

W prostocie tkwi siła

Wydanie III

Excel 2013 PL Programowanie w VBA

DLA

BYSTRZAKÓW™

Dowiedz się, jak:

- używać niezbędnych narzędzi i operacji języka VBA
- optymalnie wykorzystywać rejestrator makr w Excelu
- radzić sobie z błędami w kodzie i je eliminować
- personalizować funkcje arkusza i rozwijać aplikacje przyjazne dla użytkowników



 septem
septem.pl

John Walkenbach

Tytuł oryginalny: Excel® VBA Programming For Dummies®, 3rd Edition
Tłumaczenie: Ryszard Górnowicz, Grzegorz Kowalczyk
ISBN: 978-83-246-7953-9

Original English language edition Copyright © 2013 by John Wiley & Sons, Inc.,
Hoboken, New Jersey. All rights reserved including the right of reproduction in whole
or in part any form. This translation published by arrangement with Wiley Publishing, Inc.

Oryginalne angielskie wydanie © 2013 by John Wiley & Sons, Inc., Hoboken, New Jersey.
Wszelkie prawa, włączając prawo do reprodukcji całości lub części w jakiejkolwiek formie, zarezerwowane.
Tłumaczenie opublikowane na mocy porozumienia z Wiley Publishing, Inc.

Translation copyright © 2014 by Helion S.A.

Wiley, the Wiley logo, For Dummies, the Dummies Man logo, A Reference for the Rest of Us!,
The Dummies Way, Dummies Daily, The Fun and Easy Way, Dummies.com, Making Everything Easier,
and related trade dress are trademarks or registered trademarks of John Wiley and Sons, Inc.
and/or its affiliates in the United States and/or other countries. Used under license.

Wiley, the Wiley logo, For Dummies, the Dummies Man logo, A Reference for the Rest of Us!,
The Dummies Way, Dummies Daily, The Fun and Easy Way, Dummies.com, Making Everything Easier,
i związana z tym szata graficzna są markami handlowymi John Wiley and Sons, Inc. i/lub firm stowarzyszonych
w Stanach Zjednoczonych i/lub innych krajach. Wykorzystywane na podstawie licencji.

Polish language edition published by Wydawnictwo Helion.
Copyright © 2014.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means,
electronic or mechanical, including photocopying, recording or by any information storage retrieval system,
without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji
w jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także
kopowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich
niniejszej publikacji.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były
kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym
ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION nie ponoszą również
żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://dlabystrzakow.pl/user/opinie/e13pub_ebook
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi
w książce można znaleźć pod adresem:
<ftp://ftp.helion.pl/przyklady/e13pub.zip>

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: dlabystrzakow@dlabystrzakow.pl
WWW: <http://dlabystrzakow.pl>

Printed in Poland.

- [Poleć książkę na Facebook.com](#)
- [Kup w wersji papierowej](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

```
Sub Dedykacja_Ksiazki()
    name = Application.UserName
    reason = InputBox("Co sprawia, że " & name & "jest tak wyjątkowy?")
    msg = "Ta książka zadedykowana jest " & name
    msg = msg & " ponieważ " & reason
    MsgBox msg, vbInformation, "Dedykacja"
End Sub
```


Spis treści

O autorze	15
Podziękowania autora	17
Wstęp	19
Czy ta książka jest dla Ciebie?	19
A więc chcesz być programistą?	20
Dlaczego warto?	20
Co powinieneś wiedzieć?	21
Obowiązkowy podrozdział o konwencjach typograficznych	22
Sprawdź ustawienia zabezpieczeń	22
Jak podzielona jest książka?	24
Część I: Wstęp do programowania w VBA	24
Część II: Jak VBA współpracuje z Exceliem?	24
Część III: Podstawy programowania	24
Część IV: Komunikacja z użytkownikiem	24
Część V: Od teorii do praktyki	24
Część VI: Dekalogi	24
Ikony używane w książce	25
Pobieranie plików z przykładami	25
Co dalej?	26
Część I: Wstęp do programowania w VBA	27
Rozdział 1: Czym jest VBA?	29
No dobrze, czym jest więc VBA?	29
Co można zrobić za pomocą VBA?	30
Wprowadzanie bloków tekstu	31
Automatyzacja często wykonywanego zadania	31
Automatyzacja powtarzalnych operacji	31
Tworzenie własnego polecenia	31
Tworzenie własnego przycisku	31
Tworzenie własnych funkcji arkusza kalkulacyjnego	31
Tworzenie własnych dodatków do Excela	32
Tworzenie kompletnych aplikacji opartych na makrach	32

Plusy i minusy języka VBA	32
Plusy języka VBA	32
Minusy języka VBA	33
VBA w pigułce	33
Wycieczka po wersjach Excela	35
Rozdział 2: Szybkie zanurzenie	39
Przygotowanie do pracy	39
Plan działania	40
Stawiamy pierwsze kroki	40
Rejestrowanie makra	41
Testowanie makra	42
Podgląd kodu makra	42
Modyfikacja makra	44
Zapisywanie skoroszytów zawierających makra	45
Bezpieczeństwo makr	45
Więcej o makrze NameAndTime	47
Część II: Jak VBA współpracuje z Exceliem?	49
Rozdział 3: Praca w edytorze VBE	51
Czym jest Visual Basic Editor?	51
Uruchamianie edytora VBE	51
Zapoznanie z komponentami edytora VBE	52
Praca z oknem Project	54
Dodawanie nowego modułu VBA	55
Usuwanie modułu VBA	55
Eksportowanie i importowanie obiektów	56
Praca z oknem Code	56
Minimalizowanie i maksymalizowanie okien	56
Tworzenie modułu	57
Wprowadzanie kodu VBA do modułu	58
Bezpośrednie wprowadzanie kodu	58
Używanie rejestratora makr	61
Kopiowanie kodu VBA	63
Dostosowywanie środowiska VBA	63
Karta Editor	64
Karta Editor Format	66
Karta General	67
Karta Docking	68
Rozdział 4: Wprowadzenie do modelu obiektowego w Excelu	69
Czy Excel to obiekt?	70
Wspinaczka po hierarchii obiektów	70
Zapoznanie z kolekcjami	71
Odwoływanie się do obiektów	71
Nawigacja po hierarchii obiektów	72
Upraszczanie odwołań do obiektów	73

Właściwości i metody obiektów74
Właściwości obiektów74
Metody obiektów76
Zdarzenia obiektów77
Poszukiwanie dodatkowych informacji78
System pomocy VBA78
Narzędzie Object Browser79
Automatyczna lista właściwości i metod80
Rozdział 5: Procedury Sub i Function w języku VBA	81
Procedury Sub a funkcje81
Rzut oka na procedury Sub82
Rzut oka na procedury Function82
Nazwy procedur Sub i Function83
Uruchamianie procedur Sub83
Bezpośrednie uruchamianie procedur Sub85
Uruchamianie procedur w oknie dialogowym Makro85
Uruchamianie makr za pomocą skrótów klawiszowych86
Uruchamianie procedur przy użyciu przycisków i kształtów87
Uruchamianie procedur z poziomu innych procedur89
Uruchamianie procedur Function89
Wywoływanie funkcji z poziomu procedur Sub90
Wywoływanie funkcji z poziomu formuł arkusza90
Rozdział 6: Używanie rejestratora makr	93
Czy to rzeczywistość, czy to VBA?93
Podstawy rejestrowania makr93
Przygotowania do rejestrowania makr95
Względne czy bezwzględne?96
Rejestrowanie makr w trybie odwołań bezwzględnych96
Rejestrowanie makr w trybie odwołań względnych97
Co jest rejestrowane?98
Opcje rejestratora makr100
Nazwa makra100
Klawisz skrótu100
Przechowuj makro w101
Opis101
Czy to coś jest wydajne?101
Część III: Podstawy programowania	105
Rozdział 7: Kluczowe elementy języka VBA	107
Stosowanie komentarzy w kodzie VBA107
Używanie zmiennych, stałych i typów danych109
Pojęcie zmiennej109
Czym są typy danych w języku VBA?110
Deklarowanie zmiennych i określanie ich zasięgu111

Stale	117
Stale predefiniowane	118
Łańcuchy znaków	118
Daty i godziny	119
Instrukcje przypisania	120
Przykłady instrukcji przypisania	120
O znaku równości	121
Proste operatory	121
Praca z tablicami	123
Deklarowanie tablic	123
Tablice wielowymiarowe	124
Tablice dynamiczne	124
Stosowanie etykiet	125
Rozdział 8: Praca z obiektami Range	127
Szybka powtórka	127
Inne sposoby odwoływania się do zakresu	129
Właściwość Cells	129
Właściwość Offset	130
Wybrane właściwości obiektu Range	131
Właściwość Value	131
Właściwość Text	132
Właściwość Count	133
Właściwości Column i Row	133
Właściwość Address	133
Właściwość HasFormula	134
Właściwość Font	134
Właściwość Interior	136
Właściwości Formula i FormulaLocal	136
Właściwość NumberFormat	137
Wybrane metody obiektu Range	137
Metoda Select	137
Metody Copy i Paste	138
Metoda Clear	138
Metoda Delete	139
Rozdział 9: Praca z funkcjami VBA i arkusza kalkulacyjnego	141
Co to jest funkcja?	141
Stosowanie wbudowanych funkcji VBA	142
Przykłady funkcji VBA	142
Funkcje VBA, które robią coś więcej niż tylko zwracanie wartości	144
Odkrywanie funkcji VBA	144
Użycie funkcji arkusza kalkulacyjnego w VBA	145
Przykłady funkcji arkusza kalkulacyjnego	148
Wprowadzanie funkcji arkusza kalkulacyjnego	150
Więcej o użyciu funkcji arkusza kalkulacyjnego	151
Użycie własnych funkcji	151

Rozdział 10: Sterowanie przepływem i podejmowanie decyzji	153
Zabierz się za przepływ, kolego	153
Instrukcja GoTo	154
Decyzyje, decyzje	155
Struktura If-Then	155
Struktura Select Case	159
Entliczek, pętliczek — czyli jak używać pęli?	162
Pętle For-Next	162
Pętla Do-While	167
Pętla Do-Until	168
Użycie pętli For Each-Next z kolekcjami	168
Rozdział 11: Automatyczne procedury i zdarzenia	171
Przygotowanie do wielkiego zdarzenia	171
Czy zdarzenia są przydatne?	173
Programowanie procedur obsługi zdarzeń	173
Gdzie jest umieszczony kod VBA?	174
Tworzenie procedury obsługi zdarzenia	175
Przykłady wprowadzające	176
Zdarzenie Open dla skoroszytu	176
Zdarzenie BeforeClose dla skoroszytu	179
Zdarzenie BeforeSave dla skoroszytu	179
Przykłady zdarzeń aktywacyjnych	180
Zdarzenia aktywacji i dezaktywacji arkusza	180
Zdarzenia aktywacji i dezaktywacji skoroszytu	181
Zdarzenia aktywacji skoroszytu	182
Inne zdarzenia dotyczące arkusza	183
Zdarzenie BeforeDoubleClick	183
Zdarzenie BeforeRightClick	184
Zdarzenie Change	184
Zdarzenia niezwiązane z obiekttami	186
Zdarzenie OnTime	186
Zdarzenia naciśnięcia klawisza	188
Rozdział 12: Techniki obsługi błędów	191
Rodzaje błędów	191
Błędny przykład	192
To makro nie jest idealne	192
Makro wciąż nie jest idealne	193
Czy teraz makro jest idealne?	194
Rezygnacja z ideału	195
Inny sposób obsługi błędów	195
Korekta procedury EnterSquareRoot	195
O instrukcji On Error	196
Obsługa błędów — szczegółowe informacje	197
Wznawianie wykonywania kodu po wystąpieniu błędu	197
Obsługa błędów w pigułce	199

Kiedy ignorować błędy?	199
Rozpoznawanie określonych błędów	200
Zamierzony błąd	201
Rozdział 13: Dezynsekcja kodu, czyli jak walczyć z pluskwami	203
Rodzaje pluskiew	203
Podstawy entomologii, czyli jak zidentyfikować pluskwę	205
Metody i techniki walki z pluskwami	205
Przeglądanie kodu VBA	206
Umieszczanie funkcji MsgBox w kluczowych miejscach kodu	206
Umieszczanie polecenia Debug.Print w kluczowych miejscach kodu	208
Korzystanie z wbudowanych narzędzi Excela wspomagających odpluskwianie kodu VBA	209
Kilka słów o debuggerze	209
Ustawianie punktów przerwań w kodzie programu	209
Zastosowanie okna Watch	212
Zastosowanie okna Locals	213
Jak zredukować liczbę błędów w kodzie programu?	215
Rozdział 14: Przykłady i techniki programowania w języku VBA	217
Przetwarzanie zakresów komórek	217
Kopiowanie zakresów	218
Kopiowanie zakresu o zmiennej wielkości	219
Zaznaczanie komórek do końca wiersza lub kolumny	220
Zaznaczanie całego wiersza lub całej kolumny	221
Przenoszenie zakresów	222
Wydajne przetwarzanie komórek zaznaczonego zakresu przy użyciu pętli	222
Wydajne przetwarzanie komórek zaznaczonego zakresu przy użyciu pętli (część II)	224
Wprowadzanie wartości do komórki	225
Określanie typu zaznaczonego zakresu	226
Identyfikowanie zaznaczeń wielokrotnych	226
Zmiana ustawień Excela	227
Zmiana ustawień logicznych (opcje typu Boolean)	227
Zmiana innych opcji (typu non-Boolean)	228
Praca z wykresami	229
Metoda AddChart kontra metoda AddChart2	230
Modyfikowanie typu wykresu	231
Przechodzenie w pętli przez elementy kolekcji ChartObjects	232
Modyfikowanie właściwości wykresu	232
Zmiana formatowania wykresów	233
Jak przyspieszyć działanie kodu VBA?	234
Wyłączanie aktualizacji ekranu	234
Wyłączenie automatycznego przeliczania skoroszytu	235
Wyłączanie irytujących ostrzeżeń	236
Upraszczanie odwołań do obiektów	236
Deklarowanie typów zmiennych	237
Zastosowanie struktury With-End With	238

Część IV: Komunikacja z użytkownikiem 239

Rozdział 15: Proste okna dialogowe 241

Co zamiast formularzy UserForm?	241
Funkcja MsgBox	242
Wyświetlanie prostych okien dialogowych	243
Pobieranie odpowiedzi z okna dialogowego	243
Dostosowywanie wyglądu okien dialogowych do własnych potrzeb	244
Funkcja InputBox	247
Składnia funkcji InputBox	248
Przykład zastosowania funkcji InputBox	248
Inny rodzaj okna dialogowego InputBox	249
Metoda GetOpenFilename	250
Składnia metody GetOpenFilename	251
Przykład zastosowania metody GetOpenFilename	251
Metoda GetSaveAsFilename	253
Pobieranie nazwy folderu	254
Wyświetlanie wbudowanych okien dialogowych programu Excel	254

Rozdział 16: Wprowadzenie do formularzy UserForm 257

Kiedy używać formularzy UserForm?	257
Tworzenie formularzy UserForm — wprowadzenie	258
Praca z formularzami UserForm	259
Wstawianie nowego formularza UserForm	259
Umieszczanie formantów na formularzu UserForm	260
Modyfikacja właściwości formantów formularza UserForm	261
Przeglądanie okna Code formularza UserForm	263
Wyświetlanie formularzy UserForm	263
Pobieranie i wykorzystywanie informacji z formularzy UserForm	264
Przykład tworzenia formularza UserForm	264
Tworzenie formularza UserForm	265
Dodawanie przycisków polecień (formanty CommandButton)	265
Dodawanie przycisków opcji (formanty OptionButton)	267
Dodawanie procedur obsługi zdarzeń	268
Tworzenie makra, które wyświetla formularz na ekranie	270
Udostępnianie makra użytkownikowi	271
Testowanie działania makra	272

Rozdział 17: Praca z formantami formularza UserForm 275

Rozpoczynamy pracę z formantami formularzy UserForm	275
Dodawanie formantów	276
Wprowadzenie do właściwości formantów	277
Formanty okien dialogowych — szczegóły	278
Formant CheckBox (pole wyboru)	279
Formant ComboBox (pole kombi)	280
Formant CommandButton (przycisk polecenia)	281
Formant Frame (pole grupy)	281
Formant Image (pole obrazu)	282

Formant Label (pole etykiety)	283
Formant ListBox (pole listy)	283
Formant MultiPage	284
Formant OptionButton (przycisk opcji)	285
Formant RefEdit (pole zakresu)	286
Formant ScrollBar (pasek przewijania)	286
Formant SpinButton (pokrętło)	287
Formant TabStrip (pole karty)	288
Formant TextBox (pole tekstowe)	288
Formant ToggleButton (przycisk przełącznika)	289
Praca z formantami w oknach dialogowych	289
Zmiana rozmiarów i przenoszenie formantów w inne miejsce	289
Rozmieszczenie i wyrownianie położenia formantów w oknie dialogowym	290
Obsługa użytkowników preferujących korzystanie z klawiatury	291
Testowanie formularzy UserForm	293
Estetyka okien dialogowych	293
Rozdział 18: Techniki pracy z formularzami UserForm	295
Zastosowanie własnych okien dialogowych	295
Przykładowy formularz UserForm	296
Tworzenie okna dialogowego	296
Tworzenie kodu procedury wyświetlającej okno dialogowe	298
Udostępnianie makra użytkownikowi	299
Testowanie okna dialogowego	299
Dodawanie procedur obsługi zdarzeń	300
Sprawdzanie poprawności danych	302
Teraz okno dialogowe działa tak, jak powinno!	302
Więcej przykładów formularzy UserForm	302
Zastosowanie formantów ListBox	303
Zaznaczanie zakresów	307
Praca z wieloma grupami formantów OptionButton	309
Zastosowanie formantów SpinButton oraz TextBox	310
Wykorzystywanie formularza UserForm jako wskaźnika postępu zadania	312
Tworzenie niemodalnych okien dialogowych z wieloma kartami	315
Wyświetlanie wykresów na formularzach UserForm	317
Lista kontrolna tworzenia i testowania okien dialogowych	318
Rozdział 19: Udostępnianie makr z poziomu interfejsu użytkownika	321
Dostosowywanie Wstążki	321
Ręczne dopasowywanie Wstążki do własnych potrzeb	322
Dodawanie do Wstążki przycisku własnego makra	324
Dostosowywanie Wstążki za pomocą kodu XML	324
Dostosowywanie menu podrzędnego	329
Rodzaje obiektów CommandBar	329
Wyświetlanie wszystkich menu podrzędnego	329
Odwołania do elementów kolekcji CommandBars	330
Odwołania do formantów obiektu CommandBar	331
Właściwości formantów obiektu CommandBar	332

Przykłady zastosowania VBA do modyfikacji menu podręcznego	334
Resetowanie wszystkich wbudowanych menu podręcznych	334
Dodawanie nowego elementu do menu podręcznego Cell	335
Wyłączanie menu podręcznego	337
Tworzenie własnych pasków narzędzi	338
Część V: Od teorii do praktyki 341	
Rozdział 20: Jak tworzyć własne funkcje arkuszowe i jak przeżyć, aby o tym opowiedzieć? 343	
Dlaczego tworzymy własne funkcje?	343
Podstawowe informacje o funkcjach VBA	344
Tworzenie funkcji	345
Praca z argumentami funkcji	345
Przykłady funkcji	346
Funkcje bezargumentowe	346
Funkcje jednoargumentowe	346
Funkcje z dwoma argumentami	348
Funkcje pobierające zakres jako argument	349
Funkcje z argumentami opcjonalnymi	351
Funkcje opakowujące	353
Funkcja NumberFormat	353
Funkcja ExtractElement	354
Funkcja SayIt	354
Funkcja IsLike	355
Funkcje zwracające tablice	355
Zwarcie tablicy zawierającej nazwy miesięcy	355
Zwarcie posortowanej listy	356
Okno dialogowe Wstawianie funkcji	358
Wyświetlanie opisów funkcji	358
Opisy argumentów	360
Rozdział 21: Tworzenie dodatków 361	
No dobrze... czym zatem są dodatki?	361
Po co tworzy się dodatki?	362
Praca z dodatkami	363
Podstawy tworzenia dodatków	364
Tworzymy przykładowy dodatek	365
Konfiguracja skoroszytu	365
Testowanie skoroszytu	367
Tworzenie opisów dodatku	368
Ochrona kodu VBA	369
Tworzenie dodatku	369
Otwieranie dodatku	369
Dystrybucja dodatków	370
Modyfikowanie dodatków	371

Część VI: Dekalogi	373
Rozdział 22: Dziesięć pytań na temat VBA (wraz z odpowiedziami)	375
Rozdział 23: (Prawie) dziesięć źródeł informacji na temat Excela	379
System pomocy języka VBA	379
Wsparcie techniczne firmy Microsoft	380
Inne strony internetowe	380
Blogi poświęcone Excelowi	380
Google	381
Bing	381
Lokalne grupy użytkowników	381
Moje inne książki	381
Rozdział 24: Dziesięć rzeczy, które powinieneś robić w języku VBA i których nie powinieneś robić	383
Zawsze deklaruj wszystkie zmienne	383
Nigdy nie powinieneś mylić hasła chroniącego kod VBA z bezpieczeństwem aplikacji	384
Zawsze staraj się wyczyścić i zoptymalizować kod aplikacji	384
Nigdy nie umieszczaj wszystkiego w jednej procedurze	385
Zawsze powinieneś rozważyć zastosowanie innego oprogramowania	385
Nigdy nie zakładaj, że każdy użytkownik zezwala na uruchamianie makr	386
Zawsze staraj się eksperymentować z nowymi rozwiązaniami	386
Nigdy z góry nie zakładaj, że Twój kod będzie poprawnie działał z innymi wersjami Excela	386
Zawsze pamiętaj o użytkownikach Twojej aplikacji	387
Nigdy nie zapominaj o tworzeniu kopii zapasowych	387
Skorowidz	389

O autorze

John Walkenbach, autor wielu bestsellerów, napisał ponad 60 książek poświęconych arkuszom kalkulacyjnym. Mieszka w południowej Arizonie. W czasie, gdy nie zajmuje się Exceliem, prawdopodobnie gra na banjo.

Podziękowania autora

Jestem niezmiernie wdzięczny całemu utalentowanemu zespołowi Wiley Publishing za możliwość pisania książek o Excelu. Na specjalne podziękowania zasługuję Kelly Ewing (redaktor prowadząca) oraz Niek Otten (redaktor techniczny). Dzięki nim moja praca była zdecydowanie łatwiejsza.

Podziękowania od wydawcy oryginału

Jesteśmy dumni z tej książki. Podziel się z nami uwagami, rejestrując się na naszej stronie www.dummies.com/register/.

W wydaniu tej książki pomogli nam między innymi:

Acquisitions and Editorial

Project Editor: Kelly Ewing

(Poprzednie wydanie: Colleen Totz Diamond)

Senior Acquisitions Editor: Katie Mohr

Technical Editor: Niek Otten

Editorial Manager: Jodi Jensen

Editorial Assistant: Anne Sullivan

Sr. Editorial Assistant: Cherie Case

Cover Photo: ©Konstantin Inozemtsev/
iStockphoto

Composition Services

Project Coordinator: Katherine Crocker

Layout and Graphics: Jennifer Creasey, Christin Swinford

Proofreaders: Melissa Cossell, Jessica Kramer,

Tricia Liebig, Rob Springer

Indexer: BIM Indexing & Proofreading Services

Publishing and Editorial for Consumer Dummies

Richard Swadley, Vice President and Executive Group Publisher

Andy Cummings, Vice President and Publisher

Mary Bednarek, Executive Acquisitions Director

Mary C. Corder, Editorial Director

Publishing for Technology Dummies

Kathleen Nebenhaus, Vice President and Executive Publisher

Composition Services

Debbie Stailey, Director of Composition Services

Wstęp

Witaj, przyszły programisto Excela...

Dzięki, że kupiłeś moją książkę. Myślę, że uznasz ją za szybki i przyjemny sposób na poznanie tajników programowania w Excelu. Jeśli nawet nie masz najmniejszego pojęcia, o co chodzi w programowaniu, z tą książką w mgnieniu oka sprawisz, że Excel zatańczy, jak mu zagrasz (no, może nie aż tak szybko).

W przeciwieństwie do większości książek o programowaniu, ta napisana jest prostym językiem, tak że nawet normalni ludzie mogą ją zrozumieć. Co więcej, zawarte w niej informacje należą do gatunku „konkretów”, a nie bzdurnych wywodów, z których skorzystasz może trzy razy w życiu.

Czy ta książka jest dla Ciebie?

Odwiedź dowolną dużą księgarnię (tradycyjną lub internetową), a znajdziesz wiele książek o Excelu (o wiele za dużo, z mojego punktu widzenia). Krótki opis książki pomoże Ci zdecydować, czy jest rzeczywiście odpowiednia dla Ciebie. Ta książka:

- ✓ przeznaczona jest dla średniozaawansowanych i zaawansowanych użytkowników Excela, którzy chcą być na bieżąco z programowaniem w języku Visual Basic for Applications (VBA),
- ✓ nie wymaga żadnego wcześniejszego doświadczenia w programowaniu,
- ✓ omówione są w niej najczęściej stosowane polecenia,
- ✓ odnosi się do programowania w Excelu 2010 i 2013,
- ✓ być może czasami Cię rozbawi — są w niej nawet kreskówkowe postacie.

Jeśli używasz Excela 2000, XP lub 2003, ta książka nie jest dla Ciebie. Jeśli używasz Excela 2007, *może* być dobra, ale niektóre rzeczy się zmieniły. Prawdopodobnie lepsza będzie tu inna pozycja.

No właśnie, to *nie jest* książka o podstawach Excela. Jeśli szukasz książki o ogólnych zastosowaniach Excela, sprawdź inne pozycje wydawnictwa Helion, np. *Excel 2013 PL. Biblia Johna Walkenbacha* (tak, tak — to ja). Dostępne są również wydania tej (i nie tylko tej) książki dla poprzednich wersji Excela.

Zauważ, że tytułem tej książki nie jest *Programowanie w VBA dla Excela. Kompletny przewodnik dla bystrzaków*. Nie omawiam wszystkich aspektów programowania w Excelu, ale — jak już powiedziałem — prawdopodobnie nie będziesz chciał wiedzieć *wszystkiego* na ten temat. Jeśli po przeczytaniu tej książki poczujesz głód bardziej wyczerpujących informacji na temat programowania w Excelu, możesz sięgnąć po *Excel 2013 PL. Programowanie w VBA. Vademeum Walkenbacha Johna Walkenbacha (sic!)*, wydaną przez Helion. Zgadza się, wydania dla starszych wersji Excela są również dostępne.

A więc chcesz być programistą...

Głównym celem napisania tej książki, poza tym, że muszę zarobić na swoje rachunki, jest zapoznanie użytkowników Excela z językiem VBA — narzędziem, które pozwoli znacznie zwiększyć możliwości najpopularniejszego na świecie arkusza kalkulacyjnego. Używanie VBA wiąże się jednak z programowaniem. (Tak, to magiczne słowo na pl!)

Jeśli jesteś podobny do większości użytkowników komputerów, słowo *programista* może przywoływać wizję kogoś wyglądającego i zachowującego się zupełnie inaczej niż Ty. Być może przychodzą Ci do głowy słowa, takie jak *maniak*, *dziwak* czy *nudziarz*.

Czasy się zmieniły. Programowanie stało się dużo łatwiejsze i nawet tak zwani normalni ludzie podejmują się tej czynności. Co więcej — przyznają się do tego wśród przyjaciół i rodziny. **Programowanie** oznacza po prostu tworzenie instrukcji, które komputer wykonuje automatycznie. *Programowanie w Excelu* odnosi się do faktu, że możesz poinstruować Excel tak, by automatycznie wykonywał czynności, które normalnie robisz ręcznie. Zaoszczędzasz przy tym mnóstwo czasu i (miejmy nadzieję) unikasz błędów. Mógłbym pisać dalej, ale muszę zostawić trochę dobrego materiału na rozdział 1.

Jeśli doczytałeś do tego miejsca, mogę się założyć, że musisz zostać programistą Excela. Albo sam do tego doszedłeś, albo — co bardziej prawdopodobne — jest to polecenie Twojego szefa. Z tej książki dowiesz się tyle na temat programowania w Excelu, że nie będziesz czuł się jak idiota, gdy następnym razem wylądujesz w sali konferencyjnej razem z grupą fanatyków Excela. A gdy już dotrzesz do końca książki, uczciwie będziesz mógł powiedzieć: „O tak, trochę umiem w tym Excelu zaprogramować”.

Dlaczego warto?

Większość użytkowników Excela nie zwraca sobie głowy programowaniem w VBA. Twoje zainteresowanie tym tematem zdecydowanie plasuje Cię w elitarnej grupie. Witam wśród swoich! Jeśli nadal nie czujesz się przekonany, że opanowanie VBA jest dobrym pomysłem, przytoczę kilka dobrych powodów, dla których mógłbyś poświęcić czas na naukę programowania w Excelu. Oto one.

- ✓ **Będziesz bardziej konkurencyjny na rynku.** Czy chcesz, czy nie, aplikacje Microsoftu są niezwykle popularne. Być może już wiesz, że wszystkie programy pakietu Microsoft Office obsługują VBA. Im więcej wiesz o VBA, tym większą masz szansę na awans w pracy.

- ✓ **Pozwoli Ci to optymalnie wykorzystać środki zainwestowane w oprogramowanie** (zainwestowane przez Ciebie lub raczej Twojego pracodawcę). Używanie Excela bez znajomości VBA jest trochę jak kupno telewizora i oglądanie tylko kanałów o nieparzystych numerach.
- ✓ **Poprawisz swoją produktywność (na koniec).** Opanowanie VBA z pewnością zajmie trochę czasu, który jednak zwróci się z nawiązką — kiedy będziesz bardziej produktywny, ostatecznie zaoszczędzisz więcej czasu. To mniej więcej tak, jak się komuś mówi, że warto iść na studia.
- ✓ **To dobra rozrywka (niech będzie: czasami).** Niektórzy naprawdę lubią sprawiać, by Excel robił rzeczy, które normalnie nie są możliwe. Być może po skończeniu tej książki sam będziesz jednym z nich.

Czy teraz jesteś przekonany?

Co powinieneś wiedzieć?

Autorzy książek na ogólny mają jakieś wyobrażenie na temat docelowych czytelników. Docelowym odbiorcą tej książki jest dla mnie konglomerat dziesiątek użytkowników Excela, których spotkałem na przestrzeni lat (osobiście lub w cyberprzestrzeni). Poniższe stwierdzenia w ogólnym zarysie opisują docelowego czytelnika.

- ✓ Masz dostęp do komputera w pracy, a prawdopodobnie również w domu. Komputery te są też podłączone do internetu.
- ✓ Używasz Excela 2010 lub Excela 2013.
- ✓ Posługujesz się komputerem od wielu lat.
- ✓ Często używasz Excela w pracy i uważasz, że znasz się na nim lepiej niż przeciętna małpa.
- ✓ Musisz sprawić, by Excel robił rzeczy, których obecnie nie potrafiszkazać mu zrobić.
- ✓ Nie masz doświadczenia programistycznego lub jest ono niewielkie.
- ✓ Rozumiesz, jak przydatny może być system pomocy w Excelu. Przyjmij do wiadomości, że w tej książce nie omawiam wszystkiego. Jeśli będziesz utrzymywać dobre kontakty z systemem pomocy, będziesz w stanie uzupełnić niektóre braki.
- ✓ Musisz wykonać jakieś zadanie i masz niski poziom tolerancji dla grubych, nudnych książek z informatyki.

Obowiązkowy podrozdział o konwencjach typograficznych

Wszystkie książki z informatyki zawierają podrozdział, taki jak ten. (Myślę, że jakąś ustawą nakłada taki obowiązek.) Przeczytaj go — albo opuść.

Czasami odnoszę się do kombinacji klawiszy, co oznacza, że przytrzymujesz naciśnięty jeden klawisz, podczas gdy naciskasz drugi. Dla przykładu ***Ctrl+Z*** oznacza, że przytrzymujesz klawisz ***Ctrl***, gdy naciskasz ***Z***.

Poszczególne polecenia wybierane ze Wstążki oraz polecenia menu kontekstowego oddzielone są ukośnikiem. Dla przykładu, aby utworzyć w arkuszu zakres i nadać mu nazwę, użyj następującego polecenia:

Formuły/Nazwy zdefiniowane/Definiuj nazwę

Formuły to karta znajdująca się w górnej części Wstążki, *Nazwy zdefiniowane* to grupa poleceń, a *Definiuj nazwę* to właściwe polecenie.

W edytorze VBE nadal występują paski narzędzi oraz menu w starym stylu. Mogę więc polecić, byś wybrał polecenie *Tools/Options*. Wybierasz więc z paska menu *Tools*, a następnie polecenie *Options*.

Wszystkie teksty, które wprowadzasz z klawiatury, zapisane są czcionką o stałej szerokości znaku. Mogę więc polecić, byś wprowadził formułę $=\text{SUMA}(\text{B:B})$ do komórki A1.

Częścią programowania w Excelu jest pisanie kodu, czyli instrukcji, które program wykonuje. Również wszystkie listingi z kodem wyróżnione są w tej książce czcionką o stałej szerokości znaku, tak jak poniżej:

```
Range("A1:A12").Select
```

Niektóre wiersze kodu są zbyt długie, by zmieścić się między marginesami strony. W takich przypadkach używam standardowej w VBA sekwencji znaków do oznaczenia kontynuacji wiersza kodu, czyli znaku podkreślenia poprzedzonego spacją. Oto przykład.

```
Selection.PasteSpecial Paste:=xlValues, Operation:=xlNone,
SkipBlanks:=False, Transpose:=False
```

Wprowadzając ten kod, możesz go wpisać tak, jak powyżej, lub umieścić go w jednym wierszu (pomijając spację i znak podkreślenia).

Sprawdź ustawienia zabezpieczeń

Świat jest okrutny. Wydaje się, że ciągle jakiś cwaniak chce Cię wykorzystać lub przysporzyć Ci problemów. Świat komputerów jest równie okrutny. Prawdopodobnie słyszałeś o wirusach komputerowych, które mogą narobić paskudnych rzeczy w systemie operacyjnym. Ale czy wiedziałeś, że wirusy komputerowe mogą być przechowywane

również w pliku Excela? To prawda. W rzeczywistości dość łatwo napisać wirus komputerowy przy użyciu VBA. Nieświadomy użytkownik może otworzyć plik Excela i rozprzestrzenić wirus do innych skoroszytów i innych systemów.

Z biegiem lat Microsoft coraz bardziej troszczy się o kwestie związane z bezpieczeństwem. Dobrze, że tak jest, ale oznacza to również, że użytkownicy Excela muszą wiedzieć, jak to działa. Ustawienia zabezpieczeń możesz sprawdzić w Excelu za pomocą polecenia *Plik/Opcje/Centrum zaufania/Ustawienia Centrum zaufania*. Jest tam całe mnóstwo opcji — jeśli nawet ktoś otworzy to okno dialogowe, na ogół nigdy więcej do niego nie wraca.

Gdy klikniesz kartę *Ustawienia makr* (po lewej stronie okna dialogowego *Centrum zaufania*), do dyspozycji masz następujące opcje.

- ✓ **Wyłącz wszystkie makra bez powiadomienia** — makra nie będą działały, niezależnie od tego, co zrobisz.
- ✓ **Wyłącz wszystkie makra i wyświetl powiadomienie** — po otwarciu skoroszytu zawierającego makra zobacysz albo pasek komunikatów z przyciskiem umożliwiającym wyłączenie makr, albo — o ile okno edytora VBE jest otwarte — powiadomienie z zapytaniem, czy chcesz wyłączyć makra.
- ✓ **Wyłącz wszystkie makra oprócz makr podpisanych cyfrowo** — tylko makra z podpisem cyfrowym mogą być uruchomione (ale nawet w przypadku podpisów, których nie oznaczyłeś jako zaufane, nadal wyświetlane będzie ostrzeżenie o zabezpieczeniach).
- ✓ **Włącz wszystkie makra** — uruchamiane będą wszystkie makra bez jakichkolwiek ostrzeżeń. Opcja ta nie jest zalecana, gdyż może zostać wykonany potencjalnie niebezpieczny kod.

Rozważmy taki scenariusz. Spędzasz tydzień, pisząc w VBA piekielnie dobry program, który zrewolucjonizuje Twóją firmę. Gruntownie go testujesz i wysydasz szefowi. Ten wzywa Cię do swojego gabinetu i twierdzi, że Twoje makro nie robi absolutnie niczego. O co chodzi? Całkiem prawdopodobne, że ustawienia zabezpieczeń u Twojego szefa blokują wykonywanie makr. Możliwe też, że idąc za domyślną sugestią Microsoftu, wyłączył makra, gdy otwierał plik.

Jaki wniosek? Sam fakt, że skoroszyt Excela zawiera makro, nie gwarantuje, że zostanie ono kiedykolwiek wykonane. Zależy to od ustawień zabezpieczeń oraz od tego, czy użytkownik zdecyduje się wyłączyć lub wyłączyć makra w danym pliku.

Aby pracować z tą książką, będziesz musiał wyłączyć obsługę makr w plikach, których używasz. Osobiście radzę wybrać drugi poziom zabezpieczeń. Możesz wtedy wyłączyć makra po otwarciu pliku, który utworzyłeś. Otwierając plik pochodzący od kogoś, kogo nie znasz, powinieneś wyłączyć makra i sprawdzić kod VBA, by upewnić się, że nie zawiera złośliwych lub niebezpiecznych instrukcji. W VBA zazwyczaj łatwo zidentyfikować podejrzany kod.

Inną możliwością jest wyznaczenie zaufanego folderu. Wybierz *Plik/Opcje/Centrum zaufania/Ustawienia Centrum zaufania*. Wybierz opcję *Zaufane lokalizacje* i wskaz odpowiedni folder jako zaufaną lokalizację. Przechowuj w nim skoroszyty z zaufanych źródeł, a Excel nie będzie Cię dręczyć pytaniami, czy wyłączyć makra. Gdy na przykład pobierzesz pliki przykładowe do tej książki, możesz je umieścić w zaufanej lokalizacji.

Jak podzielona jest książka?

Książkę podzieliłem na sześć głównych części, z których każda zawiera kilka rozdziałów. Chociaż ułożyłem rozdziały w całkiem logicznym porządku, możesz je czytać w dowolnej kolejności. Oto krótka zapowiedź tego, co na Ciebie czeka.

Część I: Wstęp do programowania w VBA

Część I to tylko dwa rozdziały. W 1. zawałem wprowadzenie do języka VBA. W rozdziale 2. zabiorę Cię na interaktywną wycieczkę z przewodnikiem, na której zrobisz pierwsze kroki w programowaniu.

Część II: Jak VBA współpracuje z Exceliem?

Pisząc tę książkę, zakładam, że potrafisz już posługiwać się Exceliem. Cztery rozdziały części II pozwolą Ci lepiej zrozumieć, w jaki sposób VBA jest zaimplementowany w Excelu. Każdy z tych rozdziałów jest ważny, lepiej więc ich nie opuszczać.

Część III: Podstawy programowania

W ośmiu rozdziałach części III zapoznasz się z tym wszystkim, co stanowi sedno programowania. Być może nie musisz znać tych wszystkich informacji, ale będziesz się cieszyć, że tu są, kiedy będą potrzebne.

Część IV: Komunikacja z użytkownikiem

Jedną z najbardziej odjazdowych rzeczy podczas programowania w Excelu jest projektowanie własnych okien dialogowych (a przynajmniej ja to lubię). W pięciu rozdziałach części IV pokażę, jak utworzyć okna dialogowe wyglądające tak, jakby wyszły prosto z laboratorium oprogramowania Microsoftu.

Część V: Od teorii do praktyki

Dwa rozdziały części V stanowią zebranie informacji zawartych w poprzednich rozdziałach. Odkryjesz, jak w Excelu dodać własne przyciski do interfejsu użytkownika. Dowiesz się też, jak napisać własne funkcje arkusza kalkulacyjnego, jak tworzyć dodatki i projektować aplikacje przyjazne dla użytkownika, a nawet nawiązywać interakcję z innymi aplikacjami pakietu Office.

Część VI: Dekalogi

Zgodnie z tradycją, ostatnia część książek z serii *Dla bystrzaków* składa się z krótkich rozdziałów zawierających listy z przydatnymi informacjami. Ponieważ mam bzika na punkcie tradycji, w książce zawałem trzy takie rozdziały, które możesz przejrzeć, kiedy będzie ci wygodnie.

Ikony używane w książce

Jakaś firma zajmująca się badaniem rynku musiała kiedyś wykazać, że wydawcy książek z informatyki sprzedają więcej egzemplarzy, gdy na marginesach tych książek umieszczone są ikony. *Ikonę* to te małe obrazki, które z założenia mają zwrócić uwagę na pewne kwestie lub pomóc Ci zdecydować, czy coś jest warte przeczytania.

Nie wiem, czy wyniki tego badania są prawdziwe, ale nie będę ryzykował. Poniżej znajdziesz ikony, które napotkasz w podróży od przedniej do tyłnej okładki książki.



Gdy widzisz tę ikonę, kod omawianego przykładu jest dostępny w sieci. Pobierz go, aby zaoszczędzić sobie pisania. Więcej informacji znajdziesz w podrozdziale „Pobieranie plików z przykładami”.



Charakter materiału oznaczonego tą ikoną możesz uznać za techniczny. Może być interesujący, ale możesz go też spokojnie opuścić, jeżeli się spieszysz.



Nie opuszczaj informacji oznaczonych tą ikoną. Podczas robienia czegoś oznacza ona drogę na skróty, która zaoszczędzi Ci wiele czasu. (Być może pozwoli też wyjść z biura o przyzwoitej godzinie).



Ta ikona wskazuje, że oznaczone nią informacje powinieneś zachować w głębokich zakamarkach Twojej pamięci, aby później je wykorzystać.



Przeczytaj wszystko, co oznaczone jest taką ikoną. Inaczej narażasz się na utratę danych, wysadzenie komputera w powietrze, stopienie rdzenia reaktora nuklearnego, a nawet — zmarnowanie całego dnia.

Pobieranie plików z przykładami

Pliki z kodami źródłowymi wybranych przykładów dostępne są stronie internetowej wydawnictwa Helion. Aby je pobrać, wystarczy wpisać do wyszukiwarki adres:

<ftp://ftp.helion.pl/przyklady/e13pzb.zip>

Mając pliki z przykładami, zaoszczędzasz sobie wiele pisania. Możesz też pobawić się nimi czy poeksperymentować, zmieniając części kodu. Naprawdę bardzo zachęcam do eksperymentów z tymi plikami, gdyż jest to najlepszy sposób na opanowanie VBA.

Co dalej?

Czytając ten wstęp, zrobiłeś pierwszy krok. Teraz czas ruszyć dalej i zostać programistą (o tak, znowu to magiczne słowo na *p!*).

Jeśli jeszcze się nigdy nie programowałeś, zdecydowanie sugeruję zacząć od rozdziału 1. i czytać dalej tak długo, dopóki nie dowiesz się wystarczająco dużo, aby zrobić to, co chcesz zrobić. Pierwsze praktyczne doświadczenie zdobędziesz w rozdziale 2., będziesz więc miał złudzenie, że robisz szybkie postępy.

Mam nadzieję, że lektura tej książki sprawi Ci tyle samo przyjemności, ile dało mi jej pisanie.

Część I

Wstęp do programowania w VBA

W tej części...

- ✓ zapoznasz się z językiem Visual Basic for Applications,
- ✓ zobaczysz przykłady praktycznych zastosowań VBA,
- ✓ udasz się w podróż w czasie, by dowiedzieć się, jaki był Excel w minionych latach,
- ✓ przebrniesz przez praktyczny trening programowania w Excelu,
- ✓ zrozumiesz politykę bezpieczeństwa makr w Excelu.

Rozdział 1

Czym jest VBA?

W tym rozdziale:

- przeczytasz kilka słów o koncepcji języka VBA,
- dowiesz się, co można zrobić za pomocą VBA,
- poznasz plusy i minusy języka VBA,
- weźmiesz udział w miniwykładzie na temat historii Excela.

Jeśli nie możesz się już doczekać programowania, musisz jeszcze trochę się powstrzymać. Ten rozdział jest całkowicie pozbawiony wszelkiego rodzaju materiałów z ćwiczeniami praktycznymi, zawiera natomiast trochę niezbędnych informacji podstawowych, pomocnych na drodze do kariery programisty. Innymi słowy, rozdział ten utoruje drogę do tego, co będzie dalej, dając jednocześnie pewien obraz, jak w ogólną koncepcję wszechświata wpisuje się programowanie w Excelu. Nie jest aż tak nudny, jak może Ci się wydawać, spróbuj więc, proszę, powstrzymać chęć przeskoczenia do rozdziału 2.

No dobrze, czym jest więc VBA?

Visual Basic for Applications, w skrócie VBA, to język programowania rozwijany przez Microsoft (no wiesz, to ta firma, która co kilka lat próbuje nakłonić Cię do wykupienia nowej wersji oprogramowania Windows). W Excelu, podobnie jak innych programach pakietu Microsoft Office, zaimplementowany jest język VBA (bez dodatkowych opłat). Krótko mówiąc, VBA jest językiem używanym przez ludzi, takich jak Ty i ja, do tworzenia programów, które mogą kontrolować Excela.

Wyobraź sobie inteligentnego robota, który wie wszystko o Excelu. Robot ten potrafi odczytywać instrukcje, a także bardzo szybko i bezbłędnie posługiwać się Exceliem. Kiedy chcesz, by robot zrobił coś w Excelu, piszesz mu kilka instrukcji, używając specjalnych kodów. Potem mówisz robotowi, aby postępował według Twoich instrukcji — Ty natomiast siadasz wygodnie i pijesz lemoniadę. Mniej więcej tym jest VBA — to język kodów przeznaczony dla robotów. Zauważ jednak, że Excela nie sprzedają ani z robotem, ani z lemoniadą w pakuie.



Kilka słów o terminologii

Terminologia związana z programowaniem w Excelu może być nieco zagmatwana. Przykładowo VBA jest językiem programowania, ale służy również jako język makr. Jak nazwać coś, co pisze się w VBA, a wykonyuje w Excelu? Czy to *makro*, czy *program*? System pomocy Excela często określa procedury VBA jako makra, dlatego też i ja używam tego terminu. Niemniej jednak nazywam je również *programami*.

W całej książce używam terminu **automatyzacja**. Termin ten oznacza, że pewna sekwencja kroków jest

wykonywana automatycznie. Dla przykładu: pisząc makro, które nadaje kolor wybranym komórkom, drukuje arkusz, a następnie usuwa nadany kolor, jednocześnie *automatyzujesz* te trzy kroki.

Na wiasem mówiąc, *makro* nie jest skrótem od *Maryny Automat Kodujący Repetytywne Operacje*. Słowo to pochodzi od greckiego *makros*, które znaczy „duży” — co też będzie dobrze opisywać Twoje wynagrodzenie, gdy już będziesz doświadczonym programistą makr.

Co można zrobić za pomocą VBA?

Pewnie zdajesz sobie sprawę, że ludzie używają Excela do tysiąca różnych zadań. Oto zaledwie kilka przykładów:

- ✓ analiza danych naukowych,
- ✓ ustalanie i prognozowanie wydatków,
- ✓ przygotowywanie faktur i innych formularzy,
- ✓ prezentacja danych na wykresach,
- ✓ przechowywanie list danych, takich jak nazwiska klientów, oceny studentów czy prezenty świąteczne (dobry keks nie byłby zły),
- ✓ bla bla bla...

Moglibyśmy tak wyliczać w nieskończoność, ale myślę, że wiesz, o co chodzi. Chcę przez to powiedzieć, że Excel jest używany do wielu rozmaitych zadań, a każdy, kto czyta tę książkę, potrzebuje Excela do innych celów i odmienne ma wobec niego oczekiwania. Jedyną rzeczą, która łączy praktycznie wszystkich czytelników, jest *potrzeba automatyzacji jakiegoś aspektu Excela*. Właśnie do tego, Drogi Czytelniku, służy VBA.

Przykładowo w VBA możesz utworzyć program, który zimportuje jakieś liczby, a następnie sformatuje i wydrukuje miesięczny raport sprzedaży. Po opracowaniu i przetestowaniu programu możesz jednym poleciением uruchomić makro, a Excel automatycznie wykona wiele czasochłonnych czynności. Zamiast przeklikawać się przez nującą serię poleceń, klikasz przycisk i wchodzisz na Facebook, by miło spędzić czas, podczas gdy makro wykonuje Twoją pracę.

W następnych punktach pokrótkę opiszę niektóre popularne zastosowania makr VBA. Niejedna z tych czynności może działać Ci na nerwy.

Wprowadzanie bloków tekstu

Jeśli często wpisujesz do arkuszy kalkulacyjnych nazwę firmy, adres czy numer telefonu, możesz utworzyć makro, które zrobi to za Ciebie. Koncept ten możesz dowolnie rozszerzać, na przykład pisząc makro, które automatycznie utworzy listę wszystkich sprzedawców pracujących w Twojej firmie.

Automatyzacja często wykonywanego zadania

Załóżmy, że jesteś kierownikiem działu sprzedaży i musisz przygotowywać miesięczne raporty sprzedaży, aby zadowolić szefa. Jeśli zadanie jest proste, możesz utworzyć program w VBA, który wykonuje je za Ciebie. Twój szef będzie pod wrażeniem niezmiennie wysokiej jakości raportów i zostaniesz przeniesiony na nowe stanowisko (awans?), do którego nie masz kompletnie żadnych kwalifikacji.

Automatyzacja powtarzalnych operacji

Jeśli musisz wykonać tę samą operację na, powiedzmy, dwunastu różnych arkuszach Excela, możesz zarejestrować makro, wykonując tę czynność w pierwszym arkuszu, a następnie nakazać Excelowi powtórzenie Twojej czynności w pozostałych arkuszach, kiedy uruchomisz makro. Najlepsze w tym wszystkim jest to, że Excel nigdy nie narzeka na zmęczenie. Rejestrowanie makr w Excelu podobne jest do nagrywania filmów. Excel nie potrzebuje jednak kamery, nie trzeba też ładować baterii.

Tworzenie własnego polecenia

Często używasz tej samej sekwencji poleceń menu w Excelu? Jeśli tak, zaoszczędź kilka sekund, tworząc makro, które połączy te polecenia w jedno własne polecenie. Będziesz mógł je wywołać kliknięciem przycisku lub kombinacją klawiszy. Nie zaoszczędzisz nie wiadomo ile czasu, ale prawdopodobnie będziesz bardziej dokładny, a facet siedzący obok będzie pod wrażeniem.

Tworzenie własnego przycisku

Möesz dostosować pasek narzędzi *Szybki dostęp*, dodając przyciski uruchamiające napisane przez Ciebie makra. Pracownicy biurowi na ogół zachwycają się przyciskami, które działają cuda. A jeśli naprawdę chcesz zaimponować kolegom z pracy, możesz nawet umieścić nowe przyciski na Wstążce.

Tworzenie własnych funkcji arkusza kalkulacyjnego

Chociaż Excel zawiera setki wbudowanych funkcji, takich jak *SUMA* czy *ŚREDNIA*, możesz też tworzyć własne funkcje, które znacznie uproszczą formuły. Gwarantuję, że zdziwisz się, jakie to łatwe. (W rozdziale 20, pokażę Ci, jak się to robi). Co więcej, własne funkcje zobaczysz w oknie dialogowym *Wstawianie funkcji*, zupełnie jakby były wbudowane. Odlotowy bajer.

Tworzenie własnych dodatków do Excela

Pewnie znasz któryś z dodatków dostarczanych wraz z Exceliem. Popularnym dodatkiem jest na przykład *Analysis ToolPak*. Możesz użyć VBA, by zaprojektować własne dodatki o specjalnym przeznaczeniu. Utworzyłem dodatek *Power Utility Pak*, używając wyłącznie VBA, i ludzie z całego świata płacą mi prawdziwe pieniądze, kiedy go używają.

Tworzenie kompletnych aplikacji opartych na makrach

Jeśli jesteś gotowy, by poświęcić trochę czasu, możesz wykorzystać VBA do tworzenia dużych aplikacji wyposażonych we własną kartę Wstążki, okna dialogowe, ekranowy system pomocy i wiele innych narzędzi. W tej książce nie zajdziemy aż tak daleko; mówię o tym, by Ci uzmysłosić, jak duże są możliwości VBA.

Plusy i minusy języka VBA

W tym podrozdziale krótko przedstawię zalety języka VBA, jak również zbadam jego ciemniejszą stronę.

Plusy języka VBA

Zautomatyzować można prawie wszystkie czynności wykonywane w Excelu. Aby to zrobić, pisze się instrukcje, które potem wykonuje Excel. Automatyzacja zadań przy wykorzystaniu VBA przynosi kilka korzyści.

- ✓ Excel wykonuje zadanie zawsze w dokładnie ten sam sposób. (W większości przypadków konsekwencja jest pożądana).
- ✓ Excel wykonuje zadanie o wiele szybciej, aniżeli można to zrobić ręcznie (chyba że nazywasz się Clark Kent).
- ✓ Jeśli jesteś dobrym programistą makr, Excel zawsze wykona zadanie bezbłędnie (czego prawdopodobnie nie można powiedzieć o Tobie lub o mnie).
- ✓ Kiedy dobrze wszystko ustawisz, zadanie może wykonać osoba, która o Excelu nie wie niczego.
- ✓ Możesz robić w Excelu rzeczy, które wcześniej nie byłyby możliwe (co może przyczynić się do wzrostu Twojej popularności w całym biurze).
- ✓ Gdy masz do wykonania długie, czasochłonne zadania, nie musisz siedzieć przed komputerem i się zanudzać. Excel wykonuje pracę, a Ty możesz przesiadywać przy dystrybutorze z zimną wodą.

Minusy języka VBA

Aby było sprawiedliwie, tyle samo czasu muszę poświęcić na spisanie wad (lub potencjalnych wad) VBA.

- ✓ Musisz wiedzieć, jak pisać programy w VBA (ale właśnie po to kupiłeś tę książkę, zgadza się?). Na szczęście, nie jest to tak trudne, jak mógłbyś się spodziewać.
- ✓ Osoby, które chcą używać Twoich programów w VBA, muszą posiadać własne kopie Excela. Byłyby świetnie, gdyby można było naciągnąć przycisk, który przekształca aplikację Excela napisaną w VBA w niezależny program, ale nie jest to możliwe (i prawdopodobnie nigdy nie będzie).
- ✓ Czasami rzeczy nie idą po naszej myśli. Inaczej mówiąc, nie można ślepo założyć, że program w VBA będzie działał zawsze prawidłowo, niezależnie od sytuacji. Musisz zatem poznać świat debugowania i — jeśli inni będą używali Twoich makr — obsługi technicznej.
- ✓ VBA jest ruchomym celem. Jak wiadomo, Microsoft ciągle rozwija Excel. Pomimo wysiłków, jakie wkłada Microsoft w zachowanie kompatybilności pomiędzy wersjami, może się okazać, że kod VBA, który napisałeś, nie będzie działać poprawnie w starszych lub przyszłych wersjach Excela.

VBA w pigułce

Aby pokazać, co na Ciebie czeka, przygotowałem zwięzły opis fundamentów VBA. Wszystkie te rzeczy będę — oczywiście — opisywał aż do znudzenia w kolejnych rozdziałach książki.

- ✓ **W VBA działania wykonuje się, pisząc (lub rejestrując) kod w module VBA.** Do przeglądania i edycji modułów VBA używa się edytora VBE (*Visual Basic Editor*).
- ✓ **Moduł VBA składa się z procedur Sub.** Procedury Sub nie mają nic wspólnego z subkulturą ani z subiektywizmem. Jest to raczej kawałek kodu komputerowego wykonującego jakieś instrukcje z obiektami lub na nich (wyjaśnię niebawem). W poniższym przykładzie przedstawiam prostą procedurę typu Sub o nazwie AddEmUp. Ten zachwycający program wyświetla wynik działania 1+1:

```
Sub AddEmUp()
    Sum = 1 + 1
    MsgBox "Wynik wynosi " & Sum
End Sub
```

Procedurę typu Sub, która nie działa poprawnie, nazywamy substandardową.

- ✓ **Moduł VBA może też zawierać procedury Function.** Procedura typu Function (inaczej funkcja) zwraca pojedynczą wartość. Można ją wywołać z wnętrza innej procedury VBA lub nawet użyć jako funkcji w formule arkusza kalkulacyjnego. Poniżej przedstawiam przykładową procedurę Function o nazwie AddTwo. Funkcja ta przyjmuje dwie liczby (nazywane **argumentami**) oraz zwraca sumę tych wartości:

```
Function AddTwo(arg1, arg2)
    AddTwo = arg1 + arg2
End Function
```

Procedurę Function, która nie działa prawidłowo, nazywamy dysfunkcyjną.

- ✓ **VBA operuje na obiektach.** Excel zawiera całe dziesiątki obiektów, na których można wykonywać różne działania. Jako przykłady obiektów można wymienić skoroszyt, arkusz, zakres komórek, wykres czy też wstawiony kształt. Do dyspozycji masz dużo więcej obiektów, którymi możesz operować z poziomu kodu VBA.
- ✓ **Obiekty mają strukturę hierarchiczną.** Obiekty mogą służyć jako **kontenery** do przechowywania innych obiektów. Na samej górze hierarchii obiektów znajduje się Excel, który sam w sobie jest obiektem o nazwie Application. Obiekt Application zawiera inne obiekty, takie jak Workbook („skoroszyt”) czy Add-In („dodatek”). Obiekt Workbook może zawierać inne obiekty, takie jak Worksheet („arkusz”) czy też Chart („wykres”). Obiekt Worksheet może zawierać takie obiekty jak Range („zakres”) i PivotTable („tabela przestawna”). Takie uporządkowanie obiektów określone jest terminem **modelu obiektowego**. (Fascynaci modelu obiektowego dowiedzą się więcej o nim w rozdziale 4.).
- ✓ **Obiekty tego samego typu tworzą kolekcję (collection).** Dla przykładu, kolekcja Worksheets składa się ze wszystkich arkuszy w danym skoroszycie, kolekcja Charts natomiast składa się ze wszystkich obiektów typu Chart. Kolekcje same w sobie są obiektami.
- ✓ **Aby odwołać się do obiektu, określamy jego położenie w hierarchii obiektów za pomocą kropki w roli separatora.** Przykładowo do skoroszytu Zeszyt1.xlsx można się odwołać tak:

```
Application.Workbooks("Zeszyt1.xlsx")
```

W przykładzie tym odwołujemy się do skoroszytu Zeszyt1.xlsx należącego do kolekcji Workbooks. Kolekcja Workbooks zawarta jest w obiekcie Application, to znaczy w Excelu. Schodząc o jeden poziom niżej, możemy się odwołać do arkusza Arkusz1 zawartego w skoroszycie Zeszyt1:

```
Application.Workbooks("Zeszyt1.xlsx").Worksheets("Arkusz1")
```

Jak pokazano poniżej, możemy pójść jeszcze dalej i na kolejnym poziomie odwołać się do konkretnej komórki, na przykład A1:

```
Application.Workbooks("Zeszyt1.xlsx").Worksheets("Arkusz1").Range("A1")
```

- ✓ **Jeśli nie określisz, do jakiego kontenera odwołuje się obiekt, jako punkt odniesienia użyty zostanie obiekt aktywny.** Jeżeli Zeszyt1.xlsx jest aktywnym skoroszytem, powyższe odwołanie można skrócić do postaci:

```
Worksheets("Arkusz1").Range("A1")
```

Kiedy wiesz, że aktywnym arkuszem jest Arkusz1, odwołanie to można jeszcze bardziej uprościć:

```
Range("A1")
```

- ✓ **Obiekty mają właściwości.** **Właściwość** można sobie wyobrazić jako „ustawienie dla obiektu”. Obiekt Range posiada na przykład takie właściwości jak Value czy Address.

Właściwościami obiektu Chart są między innymi HasTitle i Type. Za pomocą VBA możesz pobierać właściwości obiektów oraz je zmieniać.

- ✓ **Aby odwoać się do właściwości obiektu, używamy kombinacji składającej się z nazwy obiektu, kropki jako separatora i nazwy właściwości.** Przykładowo do właściwości Value zawartej w komórce A1 arkusza Arkusz1 można się odwołać w następujący sposób:

```
Worksheets("Arkusz1").Range("A1").Value
```

- ✓ **Zmiennym można przypisywać wartości.** Zmienna to element o określonej nazwie, przechowujący informację. W VBA zmiennych możesz używać do przechowywania między innymi wartości, tekstu i ustawień właściwości. Aby przypisać wartość komórki A1 arkusza Arkusz1 zmiennej o nazwie Interest, użyj poniższej instrukcji:

```
Interest = Worksheets("Arkusz1").Range("A1").Value
```

- ✓ **Obiekty posiadają metody.** Metoda to operacja, którą Excel wykonuje na obiekcie. Przykładową metodą obiektu Range jest ClearContents. Jak wskazuje nazwa metody, usuwa ona zawartość komórek należących do danego zakresu.

- ✓ **Metodę wskazujemy, podając jej nazwę, poprzedzoną nazwą obiektu i znakiem kropki pomiędzy nimi.** Instrukcja z poniższego przykładu usunie całą zawartość komórki A1:

```
Worksheets("Arkusz1").Range("A1").ClearContents
```

- ✓ **VBA zawiera wszystkie elementy właściwe dla nowoczesnych języków programowania, włącznie ze zmiennymi, tablicami czy też pętlami.** Innymi słowy, o ile jesteś gotowy poświęcić trochę czasu na opanowanie tego wszystkiego, będziesz w stanie napisać kod, który działa cuda.

Czy w to wierzysz, czy nie, powyższa lista całkiem nieźle oddaje sedno VBA. Teraz nadszedł czas na więcej szczegółów — właśnie dlatego ta książka ma więcej stron...

Wycieczka po wersjach Excela

Jeśli zamierzasz pisać makra w VBA, powinieneś znać historię Excela. Wiem, że nie spodziewał się wykładu z historii, gdy sięgałeś po tę książkę. Proszę jednak o jeszcze trochę cierpliwości, bo to ważny materiał, który może okazać się hitem na następnej imprezie fanatyków Excela.

Oto lista wszystkich głównych wersji programu Excel dla Windows, jakie ujrzały światło dzienne, wraz z krótkim komentarzem dotyczącym obsługi makr w poszczególnych wersjach.

- ✓ **Excel 2.** Pierwotna wersja Excela dla systemu Windows oznaczona była jako Version 2 (a nie 1), aby zachować zgodność z numeracją wersji dla systemu Macintosh. Excel 2 pojawił się w 1987 roku. Dzisiaj nikt go nie używa, możesz więc spokojnie zapomnieć, że kiedykolwiek istniał.

- ✓ **Excel 3.** Wypuszczony na rynek pod koniec roku 1990, obsługuje język makr XLM. Tej wersji też już nikt nie używa.
- ✓ **Excel 4.** Wersja weszła na rynek na początku roku 1992. Również obsługuje makra XLM. Obecnie używa jej pewnie jakieś 10 do 12 osób — zwolennicy filozofii w stylu „dopóki działa, nie wyrzucam”.
- ✓ **Excel 5.** Ta wersja pojawiła się na początku 1994 roku i jako pierwsza obsługiwała VBA (ale wspierała również XLM). Od lat nie słyszałem, żeby ktoś używał Excela 5.
- ✓ **Excel 95.** Znany jest również jako Excel 7 (Excel 6 nie istnieje). Dystrybucja tej wersji rozpoczęła się latem 1995 roku. Posiada kilka rozszerzeń VBA oraz obsługuje język XLM. Obecnie jest rzadko używana.
- ✓ **Excel 97.** Wersja ta, znana również jako Excel 8, narodziła się w styczniu 1997 roku. Posiada wiele nowych funkcjonalności, jak również całkowicie nowy interfejs do programowania makr w języku VBA. W Excelu 97 wprowadzono też nowy format plików, których nie da się otworzyć przy użyciu wcześniejszych wersji programu. Zdarza mi się spotkać kogoś, kto nadal korzysta z tej wersji.
- ✓ **Excel 2000.** Format numeracji wersji zmieniono na czterocyfrowy. Excel 2000, znany również jako Excel 9, zadebiutował w czerwcu 1999 roku. Z punktu widzenia programisty posiada niewiele nowych funkcjonalności. Excel 2000 jest również rzadko używany.
- ✓ **Excel 2002.** Wersja ta, znana również jako Excel 10 lub Excel XP, pojawiła się pod koniec 2001 roku. Był może najważniejszą nowością jest możliwość odzyskiwania plików, gdy program się zawiesi. Excel 2002 jest nadal używany.
- ✓ **Excel 2003.** Wersja ta zawiera najmniej nowości spośród wszystkich aktualizacji Excela, które widziałem (a widziałem wszystkie). Większość zagorzałych użytkowników Excela, nie wyłączając — oczywiście — mnie, była bardzo zawiedziona tą aktualizacją. Gdy piszę te słowa, Excel 2003 nadal jest powszechnie używaną wersją. Jest to jednocześnie ostatnia „przedwstępka” wersja Excela.
- ✓ **Excel 2007.** Excel 2007 wyznaczył początek nowej ery. Porzucono w nim stary interfejs oparty na pasku menu i paskach narzędzi, a wprowadzono Wstążkę. Ja z pewnością byłem rozczałowany, że nie można jej zmodyfikować z poziomu VBA. Wersja ta miała jednak wystarczająco wiele nowości, aby mnie usatysfakcjonować; należą do nich nowy format plików oraz obsługa dużo większych arkuszy zawierających ponad milion wierszy.
- ✓ **Excel 2010.** Microsoft pobił sam siebie, wydając tę wersję. Posiada trochę pomysłowych nowości (na przykład wykresy przebiegu w czasie), w niektórych obszarach wykonano też znaczące udoskonalenia. A jeśli potrzebujesz bardzo, ale to bardzo dużych arkuszy, możesz zainstalować wersję 64-bitową. Tak czy inaczej, znowu byłem rozczałowany, bo nadal nie ma możliwości zmodyfikowania Wstążki przy użyciu VBA.
- ✓ **Excel 2013.** To najnowsza wersja, a zarazem ta sama, której używałem, pisząc to wydanie książki. Excel 2013 dostępny jest również w wersji online i na tablety. Wstążka nadal jest obecna, ale teraz ma płaski wygląd. I nadal nie można jej modyfikować z poziomu VBA!

Książka ta została napisana z myślą o Excelu 2010 i Excelu 2013. Jeśli nie posiadasz jednej z tych wersji, w niektórych miejscach możesz się pogubić.



Jaki wniosek płynie z tego miniwykładow o historii Excela? Jeśli planujesz udostępniać makra napisane w Excelu innym użytkownikom, musisz absolutnie wziąć pod uwagę, z której wersji Excela korzystają. Użytkownicy starszych wersji nie będą mogli używać funkcjonalności wprowadzonych w wersjach późniejszych. Gdy na przykład napiszesz kod, w którym odwołasz się do komórki XFD1048576 (ostatnia komórka skoroszytu), wówczas wersje starsze niż Excel 2007 zgłoszą błąd, gdyż arkusze wersji poprzedzających Excel 2007 obsługują tylko 65 536 wierszy i 255 kolumn (ostatnią komórką jest IV65536).

Excel 2010 i nowsze również posiadają nowe obiekty, metody i właściwości. Jeśli użyjesz ich w kodzie, użytkownikom starszych wersji wyświetli się powiadomienie o błędzie, gdy uruchomią Twój makro — a winić będą Ciebie.

Dobra wiadomość jest taka, że Microsoft udostępnił Office Compatibility Pack, który umożliwia użytkownikom Excela 2003 i Excela XP otwieranie i zapisywanie skoroszytów w nowym formacie plików. Produkt ten (darmowy, nawiąsem mówiąc) nie dodaje nowych funkcjonalności do wymienionych starszych wersji Excela, a jedynie umożliwia otwieranie i zapisywanie plików w nowym formacie.



Rozdział 2

Szybkie zanurzenie

W tym rozdziale:

- ▶ utworzysz użyteczne makro w VBA, wykonując praktyczny przykład krok po kroku,
- ▶ nagrasz wykonywane w Excelu operacje, używając rejestratora makr,
- ▶ sprawdzisz i przetestujesz zarejestrowany kod,
- ▶ zajmiesz się kwestiami bezpieczeństwa makr,
- ▶ zmodyfikujesz zarejestrowane makro.

Marny ze mnie pływak. Wiem jednak, że najlepszym sposobem, by wejść do zimnej wody, jest szybkie zanurzenie się — nie ma sensu przedłużać cierpienia. Idąc z nurtem tego rozdziału, od razu możesz zamoczyć nogi i uniknąć jednocześnie utonięcia w zalewie informacji.

Po dotarciu do końca rozdziału pewnie stwierdzisz, że programowanie w Excelu nie jest takie straszne. Będziesz się wtedy cieszył, że odważyłeś się zanurzyć. W tym rozdziale pokażę krok po kroku, jak utworzyć proste, ale przydatne makro VBA.

Przygotowanie do pracy

Zanim nazwiesz siebie programistą Excela, musisz przejść przez rytuały inicjacji. Chodzi o małą zmianę, jaką musisz zrobić w Excelu, aby w górnej części okna wyświetlała się nowa karta: *Deweloper*. Dodanie karty *Deweloper* w Excelu jest proste i trzeba to zrobić tylko raz. Wykonaj kolejno następujące kroki.

1. **Kliknij prawym przyciskiem myszy dowolne miejsce na Wstążce i wybierz polecenie *Dostosuj Wstążkę*.**
2. **Na karcie *Dostosowywanie Wstążki* okna dialogowego *Opcje programu Excel* odszukaj opcję *Deweloper* znajdująca się w drugiej kolumnie.**
3. **Kliknij pole przy opcji *Deweloper*, aby ją zaznaczyć.**
4. **Kliknij OK, a powrócisz do Excela z zupełnie nową kartą *Deweloper*.**

Gdy klikniesz kartę *Deweloper*, na Wstążce wyświetnią się informacje przydatne programistom (o Tobie mowa!). Na rysunku 2.1 pokazuję wygląd Wstążki w Excelu 2013 po wybraniu karty *Deweloper*.

Rysunek 2.1.
Karta Deweloper jest domyślnie ukryta, ale łatwo ją aktywować



Plan działania

W tym podrozdziale opiszę, jak możesz utworzyć swoje pierwsze makro. Makro, które opracujesz, będzie wykonywać następujące rzeczy.

- ✓ Wpisze Twoje imię i nazwisko do komórki.
- ✓ Do komórki poniżej wprowadzi aktualną datę i czas.
- ✓ Zmieni formatowanie obu komórek na pogrubienie.
- ✓ Zmieni rozmiar czcionki w obu komórkach na 16 punktów.

Makro to raczej nie zdobędzie wyróżnień na Corocznym Konkursie dla Programistów VBA, ale od czegoś trzeba zacząć. Marko wykonuje wszystkie cztery kroki za jednym razem. Na początek za pomocą rejestratora makr nagrasz operacje, które będziesz wykonywał w poszczególnych krokach. Opiszę to w następnych podrozdziałach. Następnie sprawdzisz, czy makro działa poprawnie. Na koniec wyedytujesz makro, aby wprowadzić kilka ostatecznych poprawek. Gotowy?

Stawiamy pierwsze kroki

W tym podrozdziale opiszę kroki, jakie należy zrobić przed zarejestrowaniem makra. Inaczej mówiąc, zanim zacznie się zabawa, trzeba się do niej przygotować.

1. Uruchom Excel, jeśli nie jest jeszcze otwarty.
2. Utwórz nowy, pusty skoroszyt, jeśli tak trzeba (**Crtl+N** to mój ulubiony sposób na to).
3. Kliknij kartę **Deweloper** i spojrza na przycisk **Użij odwołań względnych** w grupie **Kod**.

Jeśli kolor tego przycisku jest inny niż kolor pozostałych przycisków, wszystko w porządku. Jeśli przycisk **Użij odwołań względnych** ma taki sam kolor jak inne przyciski, wówczas trzeba go nacisnąć.

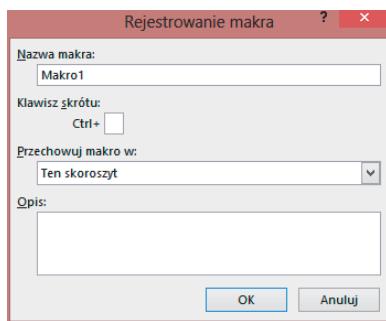
Więcej na temat przycisku **Użij odwołań względnych** napiszę w rozdziale 6. Teraz po prostu upewnij się, że opcja ta jest włączona — przycisk powinien być w innym kolorze.

Rejestrowanie makra

Czas na część praktyczną. Wykonaj uważnie następujące instrukcje.

1. **Zaznacz dowolną komórkę.**
2. **Wybierz polecenie *Deweloper/Kod/Zarejestruj makro lub kliknij przycisk rejestrowania makra na pasku statusu.***

Pojawi się okno dialogowe *Rejestrowanie makra*, co pokazano na rysunku 2.2.



Rysunek 2.2.
Przed zarejestrowaniem makra wyświetla się okno dialogowe Rejestrowanie makra

3. **Wprowadź nazwę makra.**

Excel proponuje nazwę domyślną (typu Makro1), ale lepiej użyć bardziej znaczącej nazwy. NameAndTime (bez spacji) będzie odpowiednią nazwą dla tego makra.

4. **Kliknij pole *Klawisz skrótu* i wprowadź N (wciskając Shift+N) jako klawisz skrótu.**

Określenie klawisza skrótu jest opcjonalne. Jeśli go określisz, będziesz mógł uruchomić makro, naciskając kombinację klawiszy — w tym wypadku *Ctrl+Shift+N*.

5. **Upewnij się, że na liście *Przechowuj makro w* wybrana jest opcja *Ten skoroszyt*.**
6. **Możesz wpisać coś w polu *Opis*, jeśli chcesz.**

Ten krok jest opcjonalny. Niektórzy lubią opisywać, co robi makro (lub *powinno* robić).

7. **Kliknij OK.**

Okno dialogowe *Rejestrowanie makra* zamknie się, a uruchomiony zostanie rejestrator makr. Od tego momentu Excel monitoruje wszystko, co robisz, i konwertuje to na kod VBA.

8. **Wpisz swoje imię i nazwisko do aktywnej komórki.**
9. **Przejdź do komórki poniżej i wpisz formułę:**

=TERAZ()

Formuła ta wyświetla aktualną datę i czas.

10. **Wybierz komórkę zawierającą formułę i naciśnij Crtl+C, aby skopiować ją do Schowka.**

11. Wybierz **Narzędzia główne/Schowek/Wklej/Wartości (W)**.

Polecenie to konwertuje formułę na jej wartość.

12. Wybierz komórkę z datą, a następnie naciśnij **Shift+strzałka w góre, aby zaznaczyć tę komórkę razem z komórką znajdująca się powyżej (tą, która zawiera Twoje imię i nazwisko)**.

13. Użyj kontrolek z grupy poleceń **Narzędzia główne/Czcionka**, aby zmienić formatowanie na **Pogrubienie** i ustawić wielkość czcionki na 16 punktów.

14. Wybierz polecenie **Deweloper/Kod/Zatrzymaj rejestrowanie**.

Rejestrator makr zostanie wyłączony.

Gratulacje! Właśnie utworzyłeś swoje pierwsze makro w języku VBA. Możesz zadzwonić do matki i podzielić się z nią dobrą nowiną.

Testowanie makra

Możesz teraz wypróbować makro i sprawdzić, czy działa poprawnie. Aby przetestować swoje makro, zaznacz dowolną komórkę i naciśnij **Crtl+Shift+N**.

W okamgnieniu Excel wykona makro. Twoje imię oraz aktualna data i godzina zostaną wyświetcone dużymi, pogrubionymi literami.

 Jest też inny sposób, aby uruchomić makro. Gdy klikniesz przycisk **Deweloper/Kod/Makra** (lub naciśniesz **Alt+F8**), wyświetcone zostanie okno dialogowe **Makro**. Wskaź na liście makro (w tym przypadku **NameAndTime**) i naciśnij przycisk **Uruchom**. Zanim uruchomisz makro, upewnij się, że wybrałeś komórkę, do której wpisane zostanie Twoje imię i nazwisko.

Podgląd kodu makra

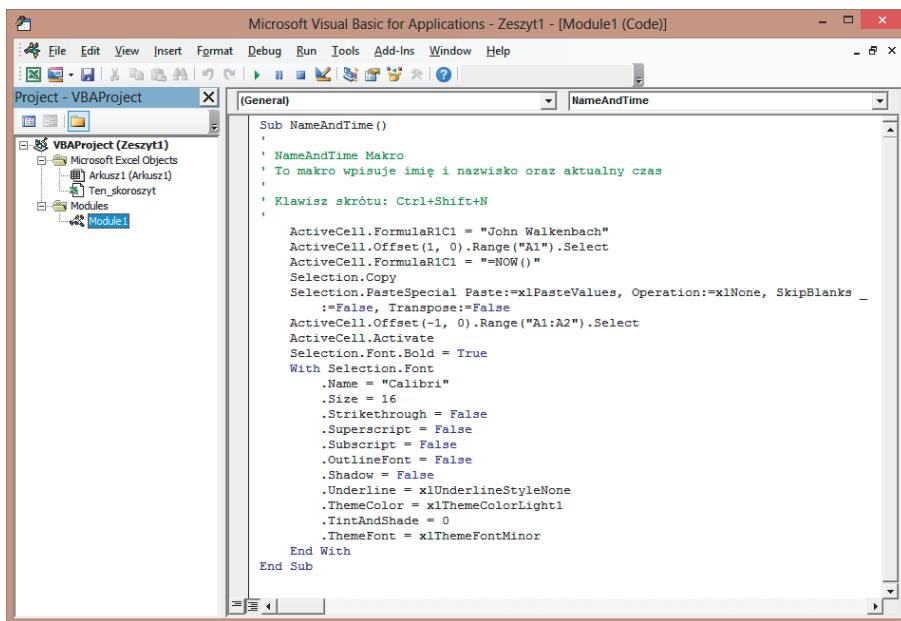
Zarejestrowałeś i przetestowałeś makro. Jeśli należysz do ciekawskich, pewnie zastanawiasz się, jak to makro wygląda. Być może nawet chciałbyś wiedzieć, gdzie jest przechowywane.

Pamiętasz, jak zacząłeś rejestrować makro? Jako miejsce do zachowania makra wskazałeś **Ten skoroszyt**. Makro jest zachowane w Twoim skoroszycie, ale aby je zobaczyć, musisz uruchomić edytor — Visual Basic Editor (w skrócie VBE).

Aby zobaczyć makro, postępuj według instrukcji.

1. Wybierz **Deweloper/Kod/Visual Basic (lub naciśnij Alt+F11)**.

Pojawi się okno programu VBE, co pokazano na rysunku 2.3. Okno to można łatwo modyfikować w zależności od indywidualnych potrzeb, dlatego u Ciebie okno edytora VBE może wyglądać nieco inaczej. Okno programu VBE zawiera kilka kolejnych okien, co może nieco przerażać. Bez obaw — przyzwyczaisz się.



Rysunek 2.3.
Edytor VBE wy-
świetla kod
VBA w module
Module1 skoro-
szytu Zeszyt1

2. W oknie edytora VBE odszukaj okno o nazwie *Project*.

Okno *Project* (znane również jako *Project Explorer*) zawiera listę wszystkich aktualnie otwartych skoroszytów i dodatków. Każdy projekt uporządkowany jest w strukturze drzewa, które można rozwinąć (aby zobaczyć więcej informacji) lub zwinąć (by ukryć część informacji).

Edytor VBE używa kilku różnych okien, a każde z nich może być otwarte lub zamknięte. Jeśli jakieś okno nie jest od razu widoczne w VBE, możesz je wyświetlić, wybierając odpowiednią opcję z menu *View*. Jeżeli przykładowo okno *Project* nie jest widoczne, możesz wybrać *View/Project Explorer* (lub nacisnąć *Ctrl+R*), aby je wyświetlić. W podobny sposób możesz wyświetlić dowolne inne okno VBE. Komponenty edytora VBE opiszę bliżej w rozdziale 3.

3. Wybierz projekt odpowiadający skoroszytowi, w którym zarejestrowałaś makro.

Jeśli nie zapisałesz skoroszytu, projekt prawdopodobnie nosi nazwę *VBAProject (Zeszyt1)*.

4. Kliknij znak plus (+) znajdujący się po lewej stronie folderu **Modules**.

Po rozwinięciu drzewa widoczny jest moduł *Module1*, który jest jedynym modułem w projekcie.

5. Kliknij dwukrotnie moduł **Module1**.

Kod VBA tego modułu wyświetlony zostanie w oknie *Code*. Na rysunku 2.3 przedstawiam wygląd tego okna na moim ekranie — możliwe, że u Ciebie nie będzie wyglądać dokładnie tak samo. Wygenerowany kod zależy od poszczególnych czynności, jakie wykonywałeś, rejestrując makro.



Hej, ja tego nie nagrałem!

W poprzednim rozdziale wspomniałem, że rejestrowanie makr jest jak nagrywanie kamerą wideo. Gdy odtwarzasz nagranie i słyszysz sam siebie, za każdym razem powtarzasz: „To nie mój głos!”. Kiedy natomiast spojrzysz na zarejestrowane przez Ciebie makro, pewnie dostrzeżesz wiele dziwnych rzeczy — nie pomyślałbyś, że to Ty je nagrałeś.

Rejestrując makro NameAndTime, zmieniłeś tylko rozmiar czcionki, podczas gdy wygenerowany kod zawiera wiele innych instrukcji formatujących czcionkę (na przykład Strikethrough, Superscript, Shadow). Nie przejmuj się, to normalne. Excel często generuje poziomie zbędny kod. W kolejnych rozdziałach dowiesz się, jak usunąć nadmierowe instrukcje z zarejestrowanego makra.

W tym momencie makro może przypominać Ci chińczyznę. Bez obaw, pokonaj jeszcze kilka rozdziałów, a wszystko będzie jasne jak słońce nad Pekinem.

Makro NameAndTime składa się z kilku instrukcji. Excel wykonuje te instrukcje jedna po drugiej, z góry na dół. Instrukcja, która poprzedzona jest apostrofem ('), jest komentarzem. Komentarze są zawarte tylko dla Twojej informacji — są ignorowane. Innymi słowy, Excel je pomija.

Pierwsza instrukcja VBA (rozpoczynająca się od słowa Sub) określa makro jako procedurę typu Sub i nadaje jej nazwę, którą podałeś jeszcze przed zarejestrowaniem makra. Czytając kod makra, pewnie będziesz w stanie rozszyfrować znaczenie niektórych jego części. Zobaczysz swoje imię i nazwisko, angielski odpowiednik formuły¹, którą wprowadziłeś, oraz wiele dodatkowych instrukcji formatujących czcionkę. Procedura Sub kończy się instrukcją End Sub.

Modyfikacja makra

Jak zapewne się domyślasz, w VBE możesz nie tylko sprawdzić kod makra, lecz również go zmienić. Choć prawdopodobnie nie masz jeszcze pojęcia, o co w nim chodzi, mogę się założyć, że będziesz umiał wykonać w kodzie następujące zmiany.

- ✓ Zmień imię i nazwisko, które jest wpisywane do aktywnej komórki. Jeśli masz psa, wpisz jego imię.
- ✓ Zmień nazwę lub rozmiar czcionki.
- ✓ Sprawdź, czy potrafisz odnaleźć właściwe miejsce na taką otwó nową instrukcję, która powoduje wyświetlanie tekstu w komórkach kursywą:

```
Selection.Font.Italic = True
```

¹ Więcej informacji o stosowaniu oryginalnych, angielskich nazw formuł i ich polskich odpowiedników znajdziesz w rozdziale 8. — przyp. tłum.



Praca w module kodu VBA przypomina pracę w edytorze tekstu (poza tym, że nie ma zawijania tekstu i nie można go formatować). Po głębszym zastanowieniu myślę, że jeszcze lepiej przyrównać ją można do pracy w Notatniku systemu Windows. Możesz nacisnąć klawisz *Enter*, aby zacząć nowy wiersz, działają również standardowe skróty klawiszowe do edycji tekstu.

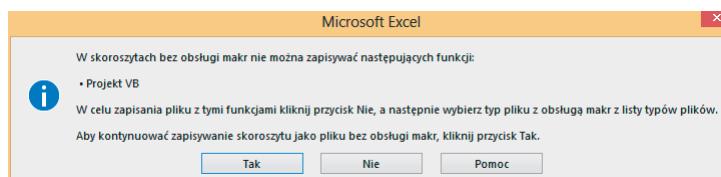
Po wprowadzeniu zmian w kodzie przeskocz z powrotem do Excela i wypróbowuj, jak działa zmodyfikowane makro. Tak samo, jak można nacisnąć *Alt+F11*, będąc w Excelu, aby uruchomić VBE, tak samo, kiedy będziesz w VBE, możesz nacisnąć *Alt+F11*, aby powrócić do Excela.

Zapisywanie skoroszytów zawierających makra

Gdy w skoroszytce przechowywane jest jedno makro lub więcej makr, plik musi być zapisany w formacie obsługującym makra. Innymi słowy, przy zapisywaniu pliku należy wybrać rozszerzenie *.xlsm* zamiast standardowego *.xlsx*.

Jeśli na przykład będziesz chciał zapisać skoroszyt zawierający makro *NameAndTime*, w oknie dialogowym *Zapisywanie jako* Excel domyślnie zaproponuje rozszerzenie *.xlsx* (format, który nie może zawierać makr). Jeśli nie zmienisz formatu na *.xlsm*, wyświetcone zostanie ostrzeżenie widoczne na rysunku 2.4. Kliknij przycisk *Nie*, a następnie z listy rozwijanej *Zapisz jako typ* wybierz *Skoroszyt programu Excel z obsługą makr (*.xlsm)*.

Rysunek 2.4.
Jeśli zawierający makra skoroszyt spróbujesz zapisać w formacie bez obsługi makr, Excel wyświetli ostrzeżenie



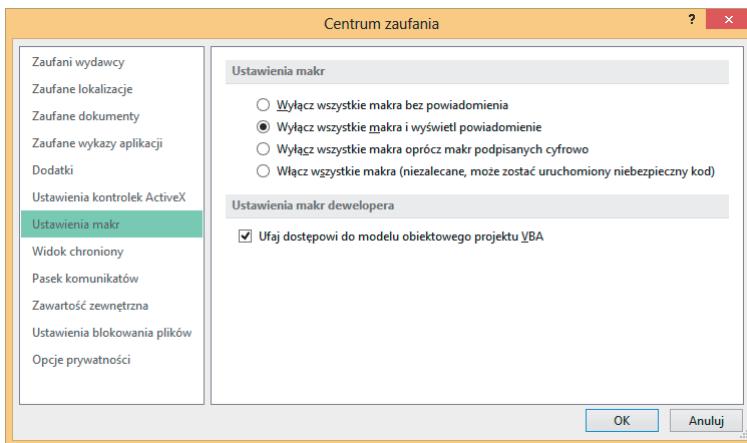
Bezpieczeństwo makr

Bezpieczeństwo makr jest w Excelu kluczową kwestią. Jest tak, ponieważ VBA jest językiem o ogromnych możliwościach — tak ogromnych, że można utworzyć makro będące w stanie wyrządzić poważne szkody w komputerze. Za pomocą makra można usuwać pliki, wysyłać informacje do innych komputerów, a nawet tak uszkodzić Windows, by uniemożliwić jego uruchomienie.

Ustawienia bezpieczeństwa makr, wprowadzone w Excelu 2007, zostały utworzone z myślą o zapobieganiu tego typu problemem.

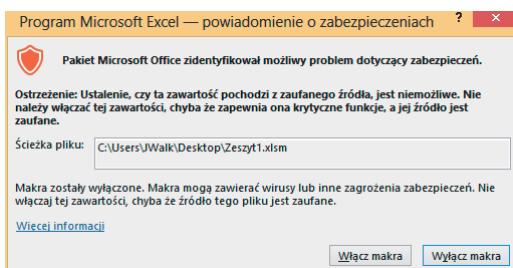
46 Część I: Wstęp do programowania w VBA

Na rysunku 2.5 przedstawiam sekcję *Ustawienia makr* okna dialogowego *Centrum zaufania*. Okno to wyświetlić można za pomocą polecenia *Deweloper/Kod/Bezpieczeństwo makr*.



Rysunek 2.5.
Sekcja Ust-
awienia makr
okna dialogo-
wego Centrum
zaufania

Domyślnie wybraną opcją w Excelu jest *Wyłącz wszystkie makra i wyświetl powiadomienie*. Gdy opcja ta jest aktywna, a plik nie jest podpisany cyfrowo ani nie jest przechowywany w zaufanej lokalizacji, wówczas podczas otwierania skoroszytu zawierającego makra Excel wyświetli ostrzeżenie, podobne do tego na rysunku 2.6. Jeśli jesteś pewien, że makro pochodzi z bezpiecznego źródła, kliknij przycisk *Włącz makra*.



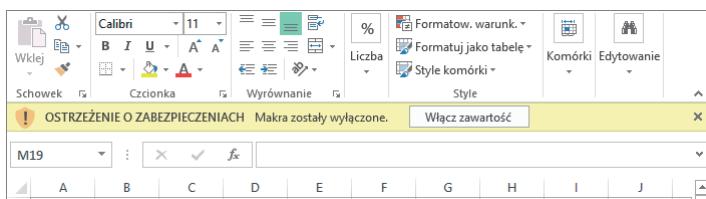
Rysunek 2.6.
Ostrzeżenie
programu Excel
o makrach
zawartych
w otwieranym
pliku

Okno pop-up widoczne na rysunku 2.6 jest wyświetlane tylko wtedy, gdy otwarty jest edytor VBE. W przeciwnym przypadku Excel wyświetla nad *Paskiem formuły* rzucające się w oczy *OSTRZEŻENIE O ZABEZPIECZENIACH*, takie jakie pokazano na rysunku 2.7. Jeśli wiesz, że skoroszyt jest bezpieczny, kliknij przycisk *Włącz zawartość*, aby włączyć makra. Aby używać skoroszytu bez obsługi makr, kliknij znak X, by zamknąć ostrzeżenie.

Excel zapamięta, czy uznałeś skoroszyt za bezpieczny. Otwierając go następnym razem, nie zobaczysz już komunikatu *OSTRZEŻENIE O ZABEZPIECZENIACH*. (Pojawi się on natomiast, mimo wszystko, w Excelu 2007).



Rysunek 2.7.
 Ostrzeżenie programu Excel o makrach zawartych we właściwie otworzonym skoroszczytce. Ostrzeżenie to jest wyświetlane, jeśli edytor VBE nie jest otwarty



Być może najlepszym sposobem na rozwiązywanie kwestii bezpieczeństwa makr jest wskazanie jednego lub kilku folderów jako *zaufanych lokalizacji*. Wszystkie skoroszczyty znajdujące się w zaufanej lokalizacji są otwierane bez ostrzeżenia o makrach. Zaufane foldery wyznaczyć można w sekcji *Zaufane lokalizacje* okna dialogowego *Centrum zaufania*.

Jeśli chcesz się dowiedzieć, co oznaczają inne ustawienia bezpieczeństwa makr, naciśnij F1, podczas gdy widoczna jest sekcja *Ustawienia makr* okna dialogowego *Centrum zaufania*. Wyświetlone zostanie okno pomocy z opisem ustawień bezpieczeństwa.

Więcej o makrze NameAndTime

Zanim skończysz tę książkę, będziesz doskonale rozumiał, jak działa makro NameAndTime. Będziesz też potrafił tworzyć bardziej wyszukane makra. Omawianie naszego przykładu zakończę kilkoma dodatkowymi uwagami dotyczącymi makra.

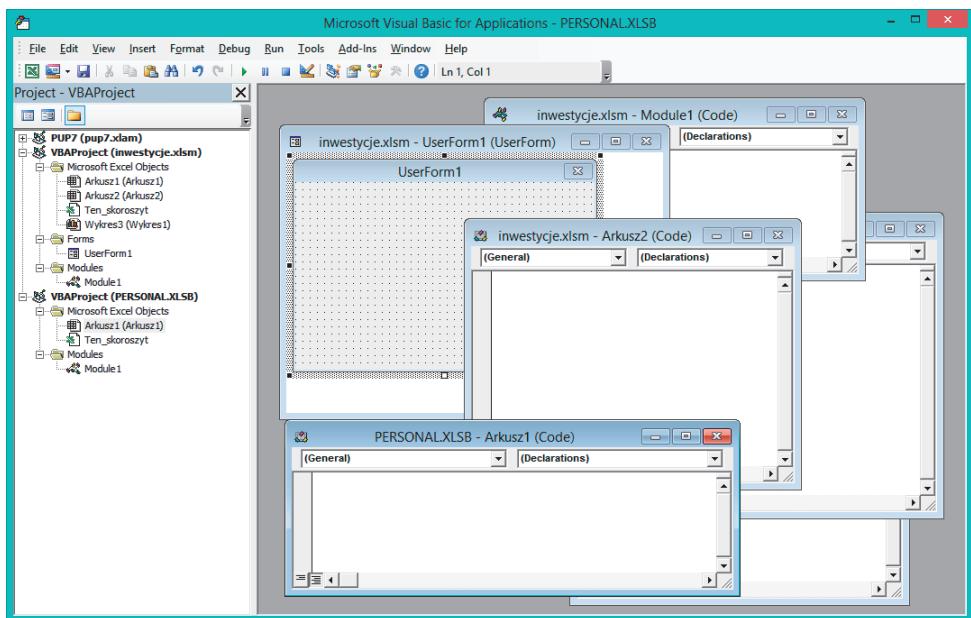
- ✓ Aby to makro działało, musi być otwarty skoroszyt, w którym jest przechowywane. Jeśli go zamkniesz, makro nie będzie działać (a naciśnięcie skrótu *Ctrl+Shift+N* nie przyniesie żadnego efektu).
- ✓ Dopóki skoroszyt zawierający makro jest otwarty, możesz uruchomić makro, podczas gdy dowolny skoroszyt jest aktywny. Innymi słowy, skoroszyt, do którego należy makro, nie musi być aktywny.
- ✓ Kod makra „nie stawia na jakość”. Makro nadpisze istniejący tekst bez żadnego ostrzeżenia, a efekty tego nie mogą zostać cofnięte.
- ✓ Przed rozpoczęciem rejestrowania makra przypiszesz mu nowy skrót klawiaturowy. Jest to tylko jedna z kilku metod uruchamiania makra. (Inne metody poznasz w rozdziale 5.).
- ✓ Zamiast rejestrować makro, możesz napisać je ręcznie. Aby to zrobić, potrzebujesz dobrej znajomości VBA. (Trochę cierpliwości, a dojdziesz do tego).
- ✓ Jako miejsce do przechowywania makra możesz wybrać *Skoroszyt makr osobistych*. Jeśli tak zrobisz, makro będzie dostępne automatycznie, kiedy tylko uruchomisz Excel. (Szczegółowe informacje o *Skoroszczycie makr osobistych* znajdziesz w rozdziale 6.).

- ✓ Możesz skonwertować skoroszyt na plik dodatku. (Więcej na ten temat w rozdziale 21.).

Gratulacje! Zostałeś wprowadzony w świat programowania w Excelu. (Wybacz, nie będzie sekretnego uścisku dłoni ani odznaczenia). Mam nadzieję, że po przeczytaniu tego rozdziału przekonałeś się, że tworzenie programu w Excelu jest czymś, co naprawdę da się zrobić — a nawet przeżyć. Czytaj dalej, a w kolejnych rozdziałach prawie na pewno znajdziesz odpowiedź na wszystkie pytania i niebawem doskonale zrozumiesz to, co robiłeś w naszym praktycznym ćwiczeniu.

Część II

Jak VBA współpracuje z Exceliem?



W tej części...

- ✓ zobaczysz, jak uzyskać dostęp do najważniejszych elementów edytora Visual Basic,
- ✓ poznasz moduły kodu VBA (gdzie przechowywane są własne instrukcje VBA użytkowników),
- ✓ zrozumiesz istotę modelu obiektowego w Excelu,
- ✓ poznasz dwa kluczowe pojęcia, czyli właściwości i metody obiektów,
- ✓ dowiesz się, jaka jest różnica pomiędzy procedurami Sub i Function,
- ✓ przejdziesz intensywny kurs rejestrowania makr w Excelu.

Rozdział 3

Praca w edytorze VBE

W tym rozdziale:

- poznasz program Visual Basic Editor,
- zaznajomisz się z komponentami edytora VBE,
- dowiesz się, z czego zbudowany jest moduł VBA,
- zrozumiesz trzy sposoby dodawania kodu VBA do modułu,
- dostosujesz interfejs VBE do własnych potrzeb.

Jako bardziej doświadczony aniżeli średnio zaawansowany użytkownik Excela prawdopodobnie sporo wiesz o skoroszytach, formułach, wykresach i innych dobrodziejstwach Excela. Nadszedł czas, by poszerzyć horyzonty i zapoznać się z całkowicie nowym aspektem Excela — edytorem Visual Basic. W tym rozdziale dowiesz się, jak pracować z edytorem VBE, a także zabierzesz się za właściwe pisane kodu VBA.

Czym jest Visual Basic Editor?

Aby nie przemęczać swoich palców, zamiast używać pełnej nazwy **Visual Basic Editor**, posługiwając się będą skrótem **VBE**. VBE jest oddzielną aplikacją, służącą do pisania i edycji makr VBA. Aplikacja ta działa równolegle z Exceliem. Pisząc *równolegle*, mam na myśli, że to Excel jest odpowiedzialny za uruchomienie VBE, gdy tego potrzebujesz.

W Excelu 2013 każdy skoroszyt wyświetlany jest w oddzielnym oknie. Niemniej jednak, istnieje tylko jedno okno VBE, które współpracuje z wszystkimi otwartymi oknami Excela.

Nie można uruchomić VBE oddzielnie. Excel musi być uruchomiony, aby edytor VBE mógł działać.



Uruchamianie edytora VBE

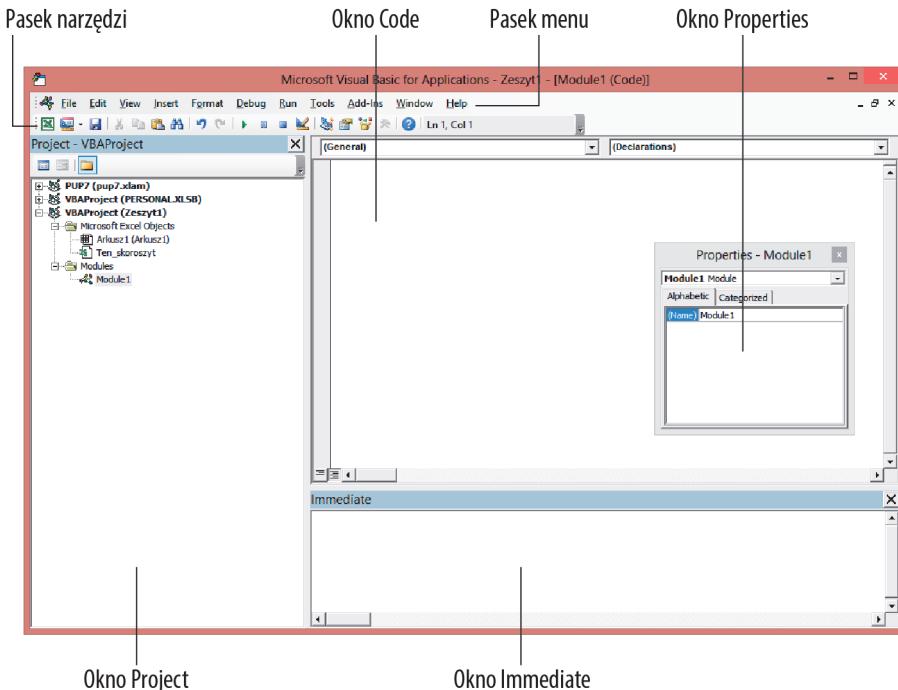
Najszybszym sposobem na uruchomienie VBE jest naciśnięcie klawiszy **Alt+F11**, podczas gdy aktywny jest program Excel. Aby wrócić do Excela, wystarczy ponownie nacisnąć **Alt+F11**. Możesz też po prostu nacisnąć przycisk *Zamknij* w pasku tytułowym edytora VBE. Gdy zamkniesz się okno VBE, aktywny będzie Excel.

Aby uruchomić VBE, można również użyć polecenia *Deweloper/Kod/Visual Basic*. Jeśli w górnej części okna Excela nie widzisz karty *Deweloper*, zajrzyj do początkowej części rozdziału 2. Objaśniam tam, jak aktywować tę bardzo przydatną kartę.

Zapoznanie z komponentami edytora VBE



Na rysunku 3.1 przedstawiam program VBE wraz ze wskazaniem jego głównych elementów. Ponieważ w edytorze VBE dzieje się wiele rzeczy, warto zmaksymalizować okno, aby widzieć jak najwięcej.



Rysunek 3.1.
VBE jest przyjacielem, który potrafi dostosować się do Twoich potrzeb

U Ciebie okno programu VBE prawdopodobnie nie będzie wyglądało dokładnie tak samo jak na rysunku 3.1. Edytor VBE zawiera kilka okien i można je dowolnie modyfikować. Okna możesz ukrywać, przemieszczać, dokować i tak dalej.

W rzeczywistości VBE zawiera jeszcze więcej komponentów niż pokazano na rysunku 3.1. Elementy te omówię w kolejnych częściach książki, kiedy będą potrzebne.

Pasek menu

Pasek menu edytora VBE działa dokładnie tak samo jak każdy inny pasek menu, jaki kiedykolwiek widziałeś. Zawiera polecenia, których używa się do wykonywania różnych operacji na poszczególnych komponentach w VBE. Przy wielu poleceniami znajdziesz również skróty klawiaturowe, które zostały im przypisane.



W VBE można używać również menu podręcznego. Kliknięcie prawym przyciskiem myszy niemalże dowolnego elementu w VBE poskutkuje wyświetleniem menu podręcznego z najczęściej używanymi poleceniami.

Pasek narzędzi

Pasek narzędzi Standard, który domyślnie znajduje się bezpośrednio pod paskiem menu (zobacz rysunek 3.1), jest jednym z czterech pasków narzędzi, jakie są dostępne w VBE. Możesz je modyfikować, przemieszczać, aktywować, ukrywać i tak dalej. Jeśli lubisz takie rzeczy, użyj polecenia *View/Tools*, aby zarządzać paskami narzędzi w VBE. Większość ludzi (nie wyłączając mnie) pozostawia je bez zmian.

Okno Project

Okno Project wyświetla diagram drzewa zawierający wszystkie skoroszyty aktualnie otwarte w Excelu (w tym dodatki i arkusze ukryte). Dwukrotne kliknięcie elementu powoduje rozwinięcie lub zwinięcie gałęzi drzewa. Okno to opiszę bardziej szczegółowo w następnym podrozdziale (zobacz „Praca z oknem Project”).

Jeśli okno *Project* nie jest widoczne, naciśnij *Ctrl+R* lub użyj polecenia *View/Project Explorer*. Aby ukryć okno *Project*, naciśnij przycisk *Zamknij* znajdujący się na pasku tytułowym tego okna. Możesz też kliknąć prawym przyciskiem myszy dowolny obszar okna i wybrać *Hide* z menu podręcznego.

Okno Code

Okno Code jest miejscem, w którym edytuje się kod języka VBA. Każdy obiekt w projekcie posiada odpowiadające mu okno *Code*. Aby wyświetlić okno z kodem źródłowym wybranego obiektu, kliknij dwukrotnie jego nazwę w oknie *Project*. Aby na przykład zobaczyć okno *Code* obiektu Arkusz1 należącego do skoroszytu Zeszt1, kliknij dwukrotnie Arkusz1 w projekcie VBAProject (Zeszt1) skoroszytu Zeszt1. Okno to będzie puste, chyba że wcześniej wprowadziłeś w nim jakieś instrukcje VBA.

Więcej na temat okien kodu źródłowego dowiesz się w podrozdziale „Praca z oknem Code”.

Okno Immediate

Okno Immediate może być widoczne lub ukryte. Jeśli nie jest widoczne, naciśnij *Ctrl+G* lub użyj polecenia *View/Immediate Window*. Aby zamknąć okno *Immediate*, kliknij przycisk *Zamknij* znajdujący się na jego pasku tytułowym lub kliknij prawym przyciskiem myszy dowolny obszar okna i wybierz *Hide* z menu podręcznego.

Co nowego w VBE?

W Excelu 2007 wprowadzony został całkowicie nowy interfejs użytkownika. Menu i paski narzędzi zastąpił nowy interfejs z bardzo praktyczną Wstążką. VBE nie doczekał się jednak liftingu — zachował klasyczny interfejs oparty na menu i paskach narzędzi.

Język VBA został zaktualizowany tak, by zapewnić obsługę nowych funkcji Excela, ale nic poza tym się nie zmieniło. Być może Microsoft w końcu zabierze się za aktualizację VBE, ale nie będę czekał z zapartym tchem.

Czymś, co naprawdę się zmieniło, jest system pomocy. Kiedyś informacje systemu pomocy przechowywane były lokalnie w komputerze użytkownika, dodatkowo istniała też możliwość uzyskania pomocy online. W Excelu 2013 cały system pomocy przeniesiony został do internetu, a dostęp do niego możliwy jest za pośrednictwem przeglądarki. Innymi słowy, aby skorzystać z systemu pomocy, musisz być podłączony do internetu.

Okno *Immediate* jest przydatne do bezpośredniego wykonywania instrukcji VBA i debugowania¹ kodu. Jeśli dopiero zaczynasz programować w VBA, okno to nie będzie Ci zbytnio potrzebne. Możesz więc spokojnie je ukryć, aby zwolnić trochę miejsca na inne elementy.

W rozdziale 13. szczegółowo opiszę pracę z oknem *Immediate*. Całkiem możliwe, że się z nim zaprzyjażnisz.

Praca z oknem Project

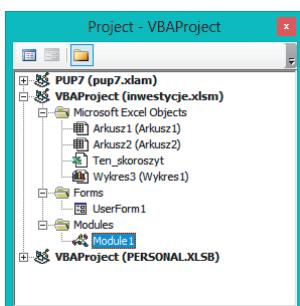
Gdy pracujesz w VBE, każdy otwarty skoroszyt i dodatek Excela jest **projektem**. Za projekt uważać można kolekcję obiektów uporządkowanych w strukturę drzewa. Aby rozwinąć drzewo projektu, kliknij pole ze znakiem plus (+) znajdujące się po lewej stronie nazwy projektu w oknie *Project*. By zwinąć drzewo projektu, kliknij znak minus (-) po lewej stronie nazwy projektu. Możesz również zwijać i rozwijać poszczególne elementy, klikając je dwukrotnie.



Gdy klikniesz dwukrotnie nazwę projektu, który jest zabezpieczony hasłem, wyświetcone zostanie okno z prośbą o podanie hasła. Jeśli go nie podasz, nie będziesz mógł rozwinąć zawartości projektu, co oznacza, że nie będziesz mógł przeglądać ani modyfikować żadnej jego części.

Na rysunku 3.2 przedstawiam okno *Project* zawierające listę trzech projektów: dodatek programu Excel o nazwie *pup7.xlam*, skoroszyt o nazwie *inwestycje.xlsm* oraz *Skoroszyt makr osobistych* (noszący zawsze nazwę *PERSONAL.XLSB*). Spośród nich tylko drzewo projektu *inwestycje.xlsm* jest rozwinięte i tylko obiekty należące do niego są widoczne.

Rysunek 3.2.
Okno Project z trzema projektami. Jeden z nich jest rozwinięty, a należące do niego obiekty są widoczne



W drzewie każdego projektu znajduje się przynajmniej jeden *węzeł* o nazwie *Microsoft Excel Objects*. Węzeł ten zawiera po jednym elemencie dla każdego arkusza w skoroszycie (każdy arkusz jest obiektem) oraz dodatkowy obiekt o nazwie *Ten_skoroszyt* (reprezentujący obiekt *Workbook*). Gdy projekt zawiera moduły VBA, drzewo projektu posiada również węzeł *Modules*. Dodatkowo, jak się przekonasz w części IV, projekt może zawierać węzeł o nazwie *Forms* zawierający obiekty formularzy *UserForm* (przechowujących własne okna dialogowe użytkownika).

¹ Proces mający na celu usunięcie błędów z programu — *przyp. tłum.*

Koncepcja obiektów może Ci się wydawać nieco zawiła. Gwarantuję jednak, że wszystko się wyjaśni w kolejnych rozdziałach. Nie przejmuj się za bardzo, jeśli jeszcze nie do końca rozumiesz, o co chodzi.

Dodawanie nowego modułu VBA

Wykonaj następujące kroki, aby dodać do projektu nowy moduł VBA.

1. Wybierz nazwę projektu w oknie *Project* edytora VBE.
2. Wybierz polecenie *Insert/Module*.

Albo:

1. Kliknij nazwę projektu prawym przyciskiem myszy.
2. Z menu podręcznego wybierz polecenie *Insert/Module*.



Gdy rejestrujesz makro, Excel automatycznie wstawia do projektu nowy moduł, w którym będzie ono przechowywane. Moduł zarejestrowanego makra będzie przechowywany w skoroszycie, który wskazałeś w liście rozwijanej *Przechowuj makro* w przed jego zarejestrowaniem.

Usuwanie modułu VBA

Czasami trzeba usunąć moduł VBA z projektu. Może on przykładowo zawierać kod, którego już nie potrzebujesz, lub nie zawiera żadnego kodu, ponieważ dodałeś nowy moduł, a potem się rozmyślisz. Aby usunąć moduł VBA z projektu, wykonaj kolejno poniższe kroki.

1. Wybierz nazwę modułu w oknie *Project*.
2. Wybierz polecenie *File/Remove xxx*, gdzie xxx jest nazwą modułu.

Albo:

1. Kliknij prawym przyciskiem myszy nazwę modułu.
2. Wybierz polecenie *Remove xxx z menu podręcznego*.

Excel, który zawsze ostrzega przed zrobieniem czegoś, czego mógłbyś żałować, zapyta Cię, czy przed usunięciem modułu chcesz wyeksportować jego kod źródłowy. Prawie zawsze odpowiesz *Nie*. (Jeśli jednak chcesz wyeksportować kod modułu, zapoznaj się z treścią następnego punktu).

Możesz usuwać moduły VBA, nie ma natomiast możliwości usuwania innych modułów kodu, takich jak moduły obiektów arkuszy i obiektu *Ten_skoroszyt*.

Eksportowanie i importowanie obiektów

Każdy obiekt w projekcie VBA można zapisać do osobnego pliku. Zapisywanie pojedynczego obiektu z projektu nazywa się **eksportowaniem**. Nie powinno Cię więc zdziwić, że obiekty można również **importować** do projektu. Możliwość eksportowania i importowania obiektów może się przydać, gdy będziesz chciał użyć wybranego obiektu (na przykład modułu VBA lub formularza UserForm) w innym projekcie. Możesz też w ten sposób wysłać kopię modułu VBA koledze lub koleżance z pracy, aby mógł on (lub ona) zimportować ten moduł do swojego projektu.

Aby wyeksportować obiekt z projektu, wykonaj następujące kroki.

1. **Zaznacz obiekt w oknie Project.**
2. **Wybierz polecenie File/Export File lub naciśnij *Ctrl+E*.**

Wyświetlone zostanie okno dialogowe z prośbą o podanie nazwy pliku. Zauważ, że obiekt nadal pozostaje w projekcie, podczas gdy wyeksportowana zostaje tylko jego kopia. W zależności od rodzaju eksportowanego obiektu, Excel sam zasugeruje właściwe rozszerzenie pliku. W rezultacie zawsze powstaje plik tekstowy. Jeśli ciekawi Cię zawartość tego pliku, możesz go otworzyć w edytorze tekstu i przejrzeć kod.

Importowanie pliku do projektu przebiega następująco.

1. **Zaznacz nazwę projektu w oknie Project.**
2. **Wybierz polecenie File/Import File lub naciśnij *Ctrl+M*.**

Wyświetli się okno dialogowe z prośbą o wskazanie pliku. Wybierz odpowiedni plik i kliknij *Otwórz*. Importować powinieneś jedynie takie pliki, które zostały wyeksportowane za pomocą polecenia *File/Export File*.



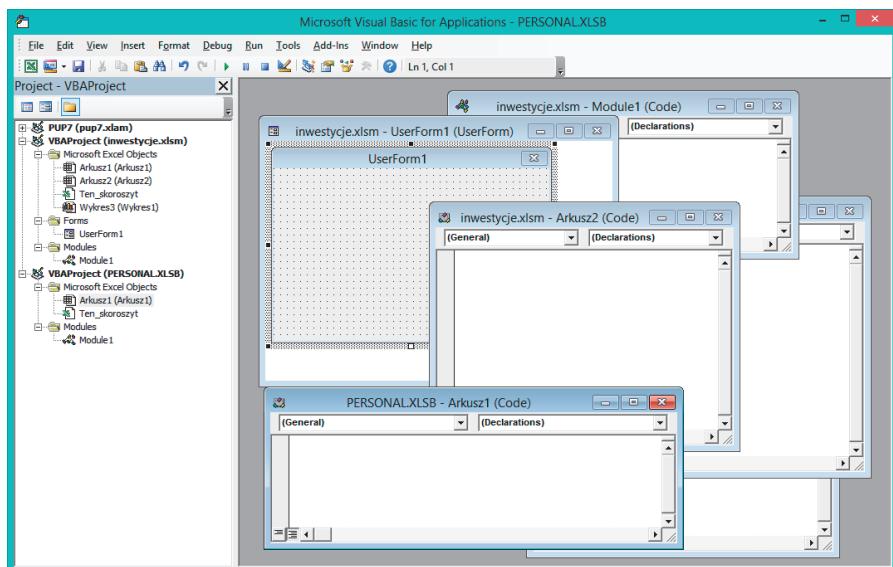
Praca z oknem Code

Gdy nabierzesz wprawy w pracy z językiem VBA, dużo czasu będziesz spędzał w oknach edycji kodu. Makra, które rejestrujesz, przechowywane są w modułach, a kod źródłowy VBA można edytować bezpośrednio w nich.

Minimalizowanie i maksymalizowanie okien

Jeśli pracujesz na wielu projektach jednocześnie, w pewnych momentach VBE może wyświetlać wiele okien *Code*. Na rysunku 3.3 przedstawiam przykład takiej sytuacji.

Okna *Code* działają podobnie jak okna skoroszytu w Excelu. Można je minimalizować, maksymalizować, ukrywać, przemieszczać, zmieniać ich rozmiar i tak dalej. Większość osób maksymalizuje aktualnie używane okno kodu, aby ułatwić sobie pracę. Widać dzięki temu więcej kodu i łatwiej skoncentrować uwagę.



Rysunek 3.3.
Nadmiar okien
Code nie wy-
gląda dobrze

Aby zmaksymalizować okno *Code*, kliknij przycisk *Maksymalizuj* na pasku tytułu tego okna (po prawej stronie, obok przycisku *X*) lub po prostu dwukrotnie kliknij pasek tytułu. Aby przywrócić pierwotny rozmiar okna, kliknij przycisk *Restore Window*. Gdy okno jest zmaksymalizowane, jego pasek tytułu nie jest widoczny, a przycisk przywracania rozmiaru okna znajduje się pod paskiem tytułu edytora VBE.

Czasami jest wygodniej, gdy widoczne są dwa lub więcej okien *Code*. Przykładowo możesz chcieć porównać kod źródłowy w dwóch modułach lub skopiować kod jednego modułu do drugiego. Możesz wówczas rozmieścić okna ręcznie lub użyć polecenia *Window/Tile Horizontally* albo *Window/Tile Vertically*, aby rozmieścić je automatycznie.

Aby szybko przełączać się pomiędzy oknami *Code*, użij skrótu *Ctrl+F6*. Każde kolejne naciśnięcie tej kombinacji klawiszy powoduje przejście do kolejnego otwartego okna *Code*. Kombinacja *Ctrl+Shift+F6* powoduje przechodzenie do kolejnych okien w odwrotnej kolejności.



Minimalizując okno, usuwasz je z pola widzenia. Możesz również kliknąć przycisk *Zamknij* (oznaczony symbolem *X*) znajdujący się na pasku tytułowym okna, aby całkowicie je zamknąć. (Zamykając okno, po prostu je ukrywasz, nie tracisz więc żadnych danych). Aby ponownie otworzyć okno, wystarczy dwukrotnie kliknąć wybrany obiekt w oknie *Project*. Nawiąsem mówiąc, praca z oknami *Code* wydaje się trudniejsza, niż jest w rzeczywistości.

Tworzenie modułu

Ogólnie mówiąc, moduł VBA może przechowywać trzy rodzaje kodu źródłowego. Oto one.

- ✓ **Deklaracje:** jedna lub kilka instrukcji języka VBA; przekazują one informacje o charakterze konfiguracyjnym. Możesz przykładowo zdeklarować typ zmiennych, których będziesz używać, lub ustawić inne opcje odnoszące się do całego modułu.

Deklaracje są w pewnym sensie instrukcjami „biernymi” — ich wykonanie nie skutkuje wywołaniem żadnych akcji.

- ✓ **Procedury Sub:** zestaw instrukcji programistycznych, których wykonanie powoduje wywołanie pewnych akcji.
- ✓ **Procedury Function (funkcje):** zestaw instrukcji programistycznych, które zwracają pojedynczą wartość (działają na zasadzie podobnej do funkcji arkusza kalkulacyjnego, takich jak SUMA).

Jeden moduł VBA może zawierać dowolną liczbę procedur Sub, funkcji i deklaracji. Właściwie *istnieje* limit, czyli około 64 000 znaków na moduł. Dla porównania, liczba znaków w tym rozdziale stanowi około połowę tego limitu. Chociaż programuję już od 15 lat, nigdy nawet nie zbliżyłem się do osiągnięcia tego limitu. Gdybym go jednak uzyskał, rozwiązanie jest proste: wystarczy wstawić nowy moduł.

To, jak zorganizujesz kod w modułach VBA, zależy wyłącznie od Ciebie. Niektórzy wolą przechowywać cały kod źródłowy należący do danej aplikacji w jednym module VBA, podczas gdy inni wolą podzielić kod na kilka różnych modułów. To kwestia gustu, tak samo jak przy meblowaniu mieszkania.

Wprowadzanie kodu VBA do modułu

Pusty moduł VBA jest jak sztuczne jedzenie, które zobaczyć można w witrynach niektórych chińskich restauracji — wygląda całkiem nieźle, ale żaden z niego pożytek. Aby zrobić coś sensownego, moduł VBA musi zawierać jakieś instrukcje. Kod źródłowy można umieścić w module VBA na trzy sposoby:

- ✓ wprowadzić instrukcje bezpośrednio,
- ✓ użyć rejestratora makr w Excelu do nagrania wykonywanych czynności i skonwertowania ich do kodu VBA (zobacz rozdział 6.),
- ✓ skopiować kod z jednego modułu i wkleić do drugiego.

Bezpośrednie wprowadzanie kodu

Czasami najlepszą drogą jest ta najbardziej bezpośrednią. Bezpośrednie wprowadzanie kodu wiąże się z... cóż, jego bezpośredniem wprowadzaniem. Inaczej mówiąc, po prostu wpisujesz kod za pomocą klawiatury. Wprowadzanie i edycja tekstu w module VBA przebiega dokładnie tak, jak możesz się tego spodziewać. Edytowany tekst możesz zaznaczać, kopiować, wycinać, wklejać i tak dalej.

Używaj klawisza tabulacji do robienia wcięć w kodzie. Wcięcia nie są konieczne, ale dzięki nim kod jest czytelniejszy, dlatego warto wyrobić sobie taki nawyk. Analizując kod przykładów w tej książce, zrozumiesz, dlaczego wcięcia w niektórych wierszach kodu mogą być pomocne.



Chwila przerwy na terminologię

Pozwole sobie na małą dygresję, aby omówić kilka terminów. W całej książce często używam terminów *procedura*, *procedura Sub*, *funkcja*, *program* czy też *podprogram*. Terminy te mogą być niejasne. Ludzie z gatunku programistów zazwyczaj używają słowa *procedura* do określenia zautomatyzowanego zadania. Z technicznego punktu widzenia, procedury dzielą się na *procedury Sub* i *procedury Function* (funkcje) — obie nazywane są czasami *podprogramami*, a nawet po prostu *programami*. Używam tych terminów wyraźnie. Istnieje jednak ważna różnica pomiędzy *procedurą Sub* a *procedurą Function*, co opiszę szczegółowo w kolejnych rozdziałach. Nie przejmuj się za bardzo terminologią — po prostu staraj się zrozumieć sens.



Pojedynczy wiersz kodu może być tak długi, jak jest to potrzebne. Niemniej jednak, prawdopodobnie będziesz chciał podzielić linijkę wiersza kodu za pomocą specjalnej sekwencji znaków oznaczających kontynuację wiersza. Aby pojedynczą linijkę kodu (zwaną **instrukcją**) można było kontynuować w następnym wierszu, pierwszy wiesz należy zakończyć spacją i znakiem podkreślenia (_). Następnie można kontynuować dalszą część instrukcji w kolejnym wierszu. Tylko nie zapominaj o spacji — sam znak podkreślenia, który nie jest poprzedzony spacją, nie zda egzaminu.

Poniżej pokazuję przykład instrukcji podzielonej na trzy wiersze.

```
Selection.Sort Key1:=Range("A1"),
Order1:=xlAscending, Header:=xlGuess, _
Orientation:=xlTopToBottom
```

Powyższa instrukcja działałaby dokładnie tak samo, gdyby została zapisana w jednym wierszu (bez znaków kontynuacji wiersza kodu). Zauważ, że użyłem wcięć w drugim i trzecim wierszu instrukcji. Wcięcia nie są konieczne, ale kiedy są, łatwiej zauważać, że wiersze te nie są oddzielnymi instrukcjami.



Biurowi inżynierowie, projektanci edytora VBE, przewidywali, że ludzie, tacy jak my, będą robić błędy. Z tego też powodu VBE umożliwia cofanie i powtarzanie kilku ostatnio wykonanych zmian. Jeśli myłykowo usunąłeś jakąś instrukcję, użij przycisku *Undo* znajdującego się na pasku menu (lub naciśnij *Ctrl+Z*), a instrukcja pojawi się z powrotem. Po skorzystaniu z funkcji *Undo* możesz użyć przycisku *Redo*, aby przywrócić zmiany, które wycofałeś. Sugeruję po eksperymentowanie z tymi poleceniami, aby zobaczyć, jak działają.

Jesteś gotowy do napisania kilku praktycznych instrukcji? Spróbuj wykonać następujące kroki.

1. **Utwórz nowy skoroszyt w Excelu.**
2. **Naciśnij *Alt+F11*, aby uruchomić VBE.**
3. **Kliknij nazwę utworzonego skoroszytu w oknie *Project*.**
4. **Wybierz polecenie *Insert/Module*, aby dodać nowy moduł VBA do projektu.**
5. **Wprowadź następujący kod do modułu:**

```
Sub GuessName()
    Msg = "Czy nazywasz się " & Application.UserName & "?"
    Ans = MsgBox(Msg, vbYesNo)
    If Ans = vbNo Then MsgBox "Przepraszam, nieważne."
    If Ans = vbYes Then MsgBox "Chyba jestem jasnowidzem!"
End Sub
```

6. Ustaw kursor w dowolnym miejscu tekstu, który wprowadziłeś, a następnie naciśnij F5, aby uruchomić procedurę.

F5 jest skrótem klawiszowym polecenia *Run/Run Sub/UserForm*. Jeśli wprowadziłeś kod poprawnie, Excel uruchomi procedurę. W efekcie będziesz mógł odpowiedzieć na pytanie postawione w prostym oknie dialogowym widocznym na rysunku 3.4. Oczywiście, tekst w oknie dialogowym będzie się różnił od tekstu widocznego na rysunku, chyba że nazywasz się tak samo jak ja.

Rysunek 3.4.
Okno dialogowe wyświetcone przez procedurę GuessName



Gdy wprowadzałeś kod z listingu w kroku 5., prawdopodobnie zauważyleś, że VBE zmienia wpisywany tekst. Gdy przykładowo wpisałeś instrukcję Sub, VBE automatycznie wstawił instrukcję End Sub. Kiedy natomiast opuścisz spację przed znakiem równości lub po nim, VBE wstawi ją za Ciebie. VBE zmienia również kolor fragmentów tekstu i ingeruje w użycie małych i wielkich liter. To całkowicie normalne zachowanie VBE — to jego sposób, by zapewnić ład i porządek w kodzie.

Jeśli wykonałeś powyższe kroki, właśnie napisałeś procedurę Sub języka VBA, nazywaną również **makrem**. Gdy naciśniesz F5, Excel wykona kod, postępując według instrukcji. Inaczej mówiąc, Excel zinterpretuje każdą instrukcję i zrobi to, co mu kazałeś. (Nie pozwól, aby ta nowo zdobyta władza uderzyła Ci do głowy). Możesz uruchomić makro tyle razy, ile chcesz — na ogół jednak traci swój urok po kilkudziesięciu razach.

Błąd komplikacji?

Istnieje możliwość, że makro GuessName nie zadziała. Gdy spróbujesz je uruchomić, Excel może się poskarżyć, wyświetlając komunikat błędu: *Compile error: Variable not defined* (Błąd komplikacji: Zmienna nie została zadeklarowana). Nie przejmuj się, jest na to prosty sposób.

Jeśli wystąpi powyższy błąd, spójrz na górną część kodu w module, a znajdziesz taką instrukcję: Option Explicit. Po prostu usuń ten wiersz, a makro

powinno zadziałać. Gdy kod rozpoczyna się od takiej instrukcji, oznacza to, że musisz zadeklarować wszystkie zmienne, jakich używasz. Temat ten przybliżę w rozdziale 7. (zobacz Deklarowanie zmiennych i określanie ich zasięgu"). Pojawienie się takiego wiersza w module oznacza, że Twój edytor VBE jest skonfigurowany tak, by dodawać go automatycznie. Nie przejmuj się tym teraz. Po prostu usuń wiersz i zapomnij o tym niemitym incydencie.

Dla formalności wymienię kilka aspektów języka VBA, jakie znalazły zastosowanie w naszym prostym makrze (omówię je wszystkie w kolejnych częściach książki). Są to:

- ✓ definiowanie procedury Sub (1. wiersz kodu),
- ✓ przypisywanie wartości zmiennym (Msg i Ans),
- ✓ konkatenacja (łączenie) łańcuchów znaków (za pomocą operatora &),
- ✓ użycie wbudowanej funkcji VBA (MsgBox),
- ✓ użycie predefiniowanych stałych (vbYesNo, vbNo, vbYes),
- ✓ użycie konstrukcji warunkowej (If ... Then, dwa razy),
- ✓ zakończenie procedury Sub (ostatni wiersz).

Nieźle, jak na początek. Nic sądzisz?

Używanie rejestratora makr

Innym sposobem dodania instrukcji do modułu VBA jest użycie rejestratora makr do „nagrania” czynności wykonywanych w Excelu. Jeśli popracowałeś nad ćwiczeniem praktycznym z rozdziału 2., posiadasz już pewne doświadczenie w używaniu tej techniki.



Tak na marginesie: nie istnieje absolutnie żaden sposób, aby „nagrać” procedurę GuessName z poprzedniego punktu za pomocą rejestratora makr. Rejestrować można tylko takie rzeczy, które można wykonać bezpośrednio w Excelu. Wyświetlanie okien dialogowych nie należy do repertuaru Excela (przywilej ten należy do VBE). Rejestrator makr jest przydatny, ale w wielu przypadkach przynajmniej część kodu będziesz musiał wprowadzić ręcznie.

Na poniższym przykładzie pokażę, jak krok po kroku zarejestrować makro, które dodaje nowy arkusz i ukrywa wszystkie kolumny i wiersze, poza pierwszymi dziesięcioma. Jeśli chcesz wykonać to ćwiczenie, utwórz nowy, pusty skoroszyt i postępuj według instrukcji.

- 1. Aktywuj dowolny arkusz skoroszytu.**
- 2. Kliknij kartę Deweloper i upewnij się, że przycisk Użij odwołań względnych nie jest aktywny.**

To makro będziemy rejestrować przy użyciu odwołań bezwzględnych.

- 3. Wybierz polecenie Deweloper/Kod/Zarejestruj makro lub kliknij ikonę znajdująca się po lewej stronie paska statusu, obok wskaźnika gotowości.**

Excel wyświetli okno dialogowe *Rejestrowanie makra*.

- 4. W oknie dialogowym Rejestrowanie makra wprowadź TenByTen jako nazwę makra oraz Shift+T jako klawisz skrótu.**

Makro będzie można uruchomić kombinacją klawiszy Ctrl+Shift+T.

- 5. Kliknij OK, aby rozpoczęć rejestrowanie.**

Excel automatycznie doda nowy moduł VBA do projektu powiązanego z aktywnym arkuszem. Od tego momentu wszystko, co robisz w Excelu, konwertowane jest

na kod VBA. Gdy rejestrujesz makro, ikona na pasku statusu przybiera kształt małego kwadratu. Jest to przypomnienie, że rejestrator makr jest uruchomiony. Klikając ten kwadrat, zatrzymasz rejestrowanie makra.

6. Kliknij ikonę Nowy arkusz po prawej stronie zakładki ostatniego arkusza.

Dodany zostanie nowy arkusz.

7. Zaznacz całą kolumnę K (jedenasta kolumna) i naciśnij *Ctrl+Shift+strzałka w prawo*. Następnie kliknij prawym przyciskiem myszy dowolną zaznaczoną kolumnę i wybierz *Ukryj* z menu podręcznego.

Excel ukryje wszystkie zaznaczone kolumny.

8. Zaznacz cały wiersz 11 i naciśnij *Ctrl+Shift+strzałka w dół*. Następnie kliknij prawym przyciskiem myszy dowolny zaznaczony wiersz i wybierz *Ukryj* z menu podręcznego.

Wszystkie zaznaczone wiersze zostaną ukryte.

9. Zaznacz komórkę A1.

10. Wybierz polecenie *Deweloper/Kod/Zatrzymaj rejestrowanie* lub kliknij ikonę (mały kwadrat) na pasku statusu.

Rejestrowanie makra zostanie zatrzymane.

Aby zobaczyć nowo zarejestrowane makro, uruchom VBE, używając skrótu *Alt+F11*. Odszukaj nazwę skoroszytu w oknie *Project*. Zauważysz, że drzewo projektu zawiera nowy moduł. Nazwa modułu zależy od tego, czy projekt zawierał już inne moduły, gdy uruchomłeś rejestrator makr. Jeśli nie, moduł będzie miał nazwę *Module1*. Kliknij dwukrotnie tę nazwę, aby zobaczyć okno *Code* z kodem źródłowym modułu.

Oto kod wygenerowany na podstawie czynności, jakie wykonałeś.

```
Sub TenByTen()
    '
    ' TenByTen Makro
    '
    ' Klawisz skrótu: Ctrl+Shift+T
    '
    Sheets.Add After:=ActiveSheet
    Columns("K:K").Select
    Range(Selection, Selection.End(xlToRight)).Select
    Selection.EntireColumn.Hidden = True
    Rows("11:11").Select
    Range(Selection, Selection.End(xlDown)).Select
    Selection.EntireRow.Hidden = True
    Range("A1").Select
End Sub
```

Aby przetestować makro, aktywuj dowolny skoroszyt i użądź skrótu, który przypisałeś do makra w kroku 4. (*Ctrl+Shift+T*).

Nie przejmuj się, jeśli nie przypisałeś żadnego skrótu. Oto sposób, w jaki możesz wyświetlić listę wszystkich dostępnych makr i uruchomić to, które Cię interesuje.

1. Wybierz polecenie *Developer/Kod/Makra*.

Wielbiciele skrótów klawiaturowych mogą nacisnąć *Alt+F8*. Obie metody powodują wyświetlenie okna dialogowego z listą dostępnych makr.

2. Wybierz makro z listy (w tym przypadku *TenByTen*).

3. Kliknij przycisk *Uruchom*.

Excel wykona makro, w wyniku czego utworzony zostanie nowy skoroszyt, w którym widocznych będzie 10 kolumn i 10 wierszy.

Gdy rejestrujesz makro, możesz wywoływać dowolną liczbę poleceń i wykonywać dowolną liczbę operacji. Excel posłusznie przetłumaczy kliknięcia myszy i uderzenia klawiszy na kod VBA.

Gdy już zarejestrujesz makro, możesz je — oczywiście — również edytować. Aby sprawdzić swoje nowe umiejętności, spróbuj tak zmodyfikować makro, aby wyświetlało 9 widocznych kolumn i 9 wierszy — w sam raz dla sudoku.

Kopiowanie kodu VBA

Ostatnią metodą dodania kodu do modułu VBA jest skopiowanie go z innego modułu lub innego miejsca (na przykład ze strony internetowej). Może się też okazać, że funkcja Sub lub Function, którą napiszałeś w jednym projekcie, przyda się również w innym projekcie. Zamiast tracić czas, ponownie wprowadzając kod, możesz aktywować moduł i użyć standardowych poleceń Schowka (*Copy* i *Paste*). Osobiście wolę używać skrótów klawiaturowych — *Ctrl+C* do kopiowania i *Ctrl+V* do wklejania. Gdy już wkleisz kod do modułu, możesz go zmodyfikować, jeśli trzeba.

Nawiasem mówiąc, w sieci możesz znaleźć mnóstwo przykładów kodu VBA. Gdybyś chciał je przetestować, zaznacz odpowiedni kod w przeglądarce i naciśnij *Ctrl+C*, aby go skopiować. Następnie aktywuj odpowiedni moduł i naciśnij *Ctrl+V*, aby wkleić kod.

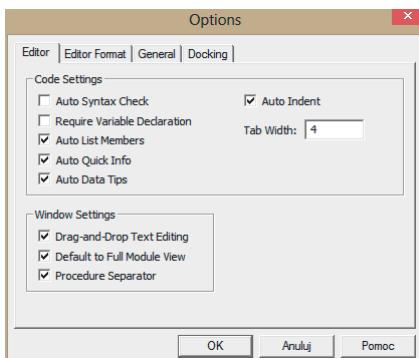
Dostosowywanie środowiska VBA

Jeśli poważnie myślisz o tworzeniu programów opartych na Excelu, spędzasz wiele czasu, pracując z modułami VBA. Aby Twоя praca była tak komfortowa, jak to tylko możliwe (nie, nie — nie zdejmuj butów), edytor VBE udostępnia całkiem sporo możliwości dostosowania środowiska programistycznego do Twoich preferencji.

Aktywuj edytor Visual Basic, a następnie wybierz *Tools/Options*. Zobaczysz okno dialogowe z czterema zakładkami: *Editor*, *Editor Format*, *General* i *Docking*. Zawierają one wiele przydatnych opcji — niektóre z nich opiszę w kolejnych punktach.

Karta Editor

Na rysunku 3.5 przedstawiam opcje dostępne po kliknięciu karty *Editor* okna dialogowego *Options*. Opcje karty *Editor* pozwalają na kontrolowanie niektórych aspektów działania edytora VBE.



Rysunek 3.5.
Karta Editor
okna dialogo-
wego Options

Opcja Auto Syntax Check

Opcja *Auto Syntax Check* określa, czy edytor VBE ma wyświetlać okno z ostrzeżeniem, w przypadku gdy wykryte zostaną błędy składni podczas wprowadzania kodu VBA. W oknie dialogowym można wówczas przeczytać krótki opis błędu. Jeśli wyłączysz tę opcję, VBE będzie oznaczał błędne instrukcje, stosując kolor czcionki odróżniający je od reszty kodu, a okna dialogowe z ostrzeżeniem nie będą wyświetlane.

Zwykle wyłączam tę opcję, ponieważ irytują mnie wyskakujące okna dialogowe, a zazwyczaj potrafię wywnioskować, dlaczego instrukcja wywołuje błąd. Ale dawno temu, zanim zostałem weteranem VBA, wskazówki te były całkiem pomocne.

Opcja Require Variable Declaration

Gdy opcja *Require Variable Declaration* jest włączona, edytor VBE wstawia na początku każdego nowego modułu VBA poniższą instrukcję:

Option Explicit

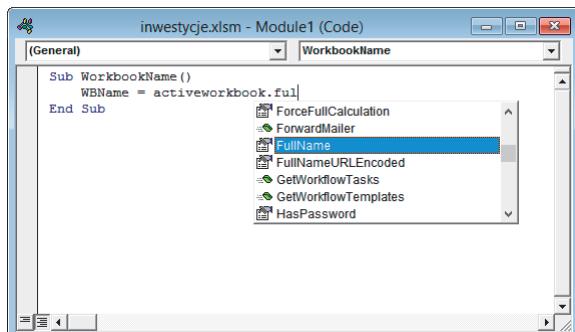
Zmiana tego ustawienia dotyczyć będzie tylko nowych modułów, ale nie już istniejących. Gdy taka instrukcja pojawi się w module, należy jawnie zadeklarować wszystkie zmienne użyte w kodzie. W rozdziale 7. napiszę, dlaczego warto wyrobić sobie taki nawyk.

Z reguły wyłączam tę opcję — wolę sam dodać instrukcję *Option Explicit*.

Opcja Auto List Members

Gdy opcja *Auto List Members* jest włączona, VBE udziela podpowiedzi, podczas gdy wprowadzasz kod VBA. Edytor automatycznie wyświetla listę możliwych zakończeń instrukcji, którą właśnie wprowadzasz. Ta magiczna sztuczka nazywana jest też czasami *IntelliSense*.

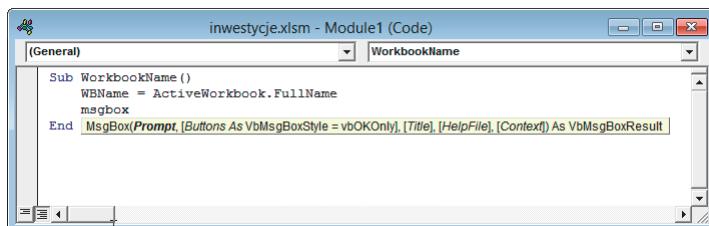
Jest to jedna z najlepszych funkcjonalności VBE i nigdy jej nie wyłączam. Na rysunku 3.6 przedstawiam przykład takiej listy (opcja nabierze większego znaczenia, gdy zaczniesz pisać instrukcje VBA).



Rysunek 3.6.
Przykład działania Auto List Members

Opcja Auto Quick Info

Jeśli opcja *Auto Quick Info* jest włączona, VBE wyświetla informacje o funkcjach i ich argumentach, gdy piszesz instrukcje. Może być to bardzo pomocne. Na rysunku 3.7 pokazuję przykład działania opcji *Auto Quick Info*. W tym przypadku podpowiada ona argumenty funkcji MsgBox.



Rysunek 3.7. Auto Quick Info podpowiada składnię funkcji MsgBox

Opcja Auto Data Tips

Jeśli opcja *Auto Data Tips* jest włączona, VBE wyświetla wartość zmiennej, nad którą umieszczony jest kursor podczas debugowania kodu. Docenisz tę opcję, gdy wkroczyś w piękny świat debugowania, który opiszę w rozdziale 13.

Opcja Auto Indent

Ustawienie *Auto Indent* określa, czy VBE automatycznie stosuje wcięcia w nowych wierszach, o głębokości takiej samej, jak w poprzedzającym wierszu. Mam bzika na punkcie wcięć, toteż nie wyłączam tej opcji.

Używaj klawisza tabulacji, nie spacji, by zrobić wcięcie w kodzie. Możesz również usuwać wcięcia, korzystając z kombinacji *Shift+Tab*. Aby zrobić wcięcie w kilku wierszach kodu jednocześnie, zaznacz te wiersze, a następnie naciśnij klawisz *Tab*.





Pasek narzędzi Edit edytora VBE (domyślnie ukryty) zawiera dwa przydatne przyciski: *Indent* i *Outdent*. Dzięki nim w szybki sposób możesz zastosować lub usunąć wcięcie dla wybranego bloku kodu. Wystarczy, że zaznacysz odpowiednie wiersze i klikniesz jeden z tych przycisków.

Opcja Drag-and-Drop Text Editing

Włączenie opcji *Drag-and-Drop Text Editing* pozwala używać myszy do kopiowania i wklejania tekstu metodą *przeciagnij i upuść (drag-and-drop)*. U mnie funkcja ta jest ciągle włączona, ale nigdy jej nie stosuję — do kopiowania i wklejania wolę używać skrótów klawiaturowych.

Opcja Default to Full Module View

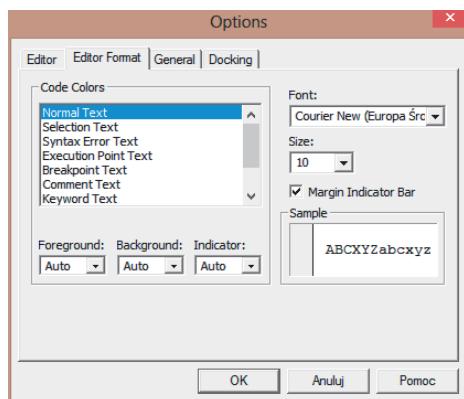
Opcja *Default to Full Module View* określa domyślne zachowanie nowych modułów (nie wpływa w żaden sposób na moduły istniejące). Gdy opcja jest włączona, kod źródłowy wszystkich procedur w module jest wyświetlany w jednym przewijanym oknie. Jeśli natomiast opcja jest wyłączona, w danym momencie widoczna może być tylko jedna procedura. Nigdy nie wyłączam tej opcji.

Opcja Procedure Separator

Gdy opcja *Procedure Separator* jest włączona, na końcu każdej procedury w oknie *Code* wyświetlany jest separator — pozioma linia oddzielająca daną procedurę od procedury następnej. Uważam, że separatorы procedur są przydatne, dlatego też zostawiam tę opcję włączoną.

Karta Editor Format

Na rysunku 3.8 przedstawiam kartę *Editor Format* okna dialogowego *Options*. Na tej karcie możesz dostosować wygląd edytora VBE do Twoich preferencji.



Rysunek 3.8.
Karta Editor
Format pozwala
zmienić wygląd
edytora VBE

Grupa ustawień *Code Colors*

Grupa ustawień *Code Colors* pozwala wybrać kolor, w jakim wyświetlany jest tekst i tło różnych elementów kodu VBA. W dużym stopniu jest to kwestia osobistych preferencji. Osobiście uważam, że domyślne kolory są całkiem niezłe. Czasami jednak bawię się trochę tymi ustawieniami, aby zmienić nieco scenerię.

Lista rozwijana *Font*

Z listy rozwijanej *Font* możesz wybrać czcionkę, jaka używana będzie w modułach VBA. Dla własnej wygody najlepiej pozostać przy czcionce o stałej szerokości znaku, takiej jak na przykład *Courier New*. Dzięki temu, że w czcionce takiej wszystkie znaki mają dokładnie taką samą szerokość, kod jest dużo czytelniejszy. Poszczególne znaki są ładnie wyrównane w pionie, można więc łatwo odnaleźć podwójne spacje (co czasem się przydaje).

Lista rozwijana *Size*

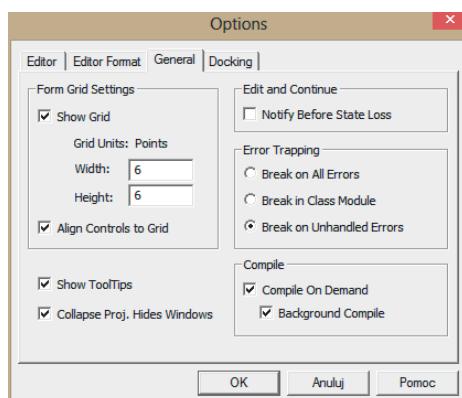
Lista rozwijana *Size* pozwala ustalić wyrażony w punktach rozmiar czcionki używanej w modułach VBA. Jest to również kwestia własnych upodobań użytkownika, na które może mieć wpływ rozdzielcość używanego ekranu i dzienna dawka pochłanianych marchewek.

Opcja *Margin Indicator Bar*

Włączenie tej opcji powoduje wyświetlanie pionowego paska wskazującego margines w oknach modułów VBA. Nie powinieneś wyłączać tej opcji. Po jej wyłączeniu nie będziesz widział wskaźników graficznych, bardzo pomocnych podczas debugowania kodu.

Karta General

Na rysunku 3.9 przedstawiam opcje dostępne na karcie *General* okna dialogowego *Options*. W prawie wszystkich przypadkach domyślne ustawienia dobrze się sprawdzają.



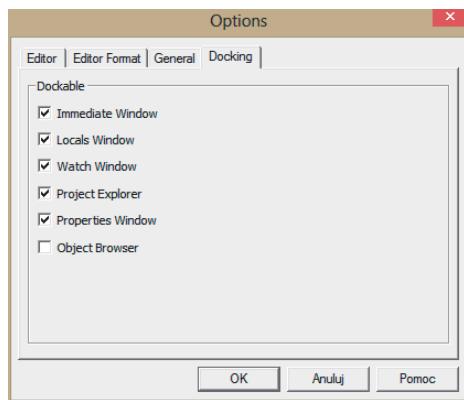
Rysunek 3.9.
Karta General
okna dialogo-
wego Options

Najważniejszym ustawieniem jest *Error Trapping*. Zdecydowanie sugeruję, abyś używał domyślnej opcji *Break On Unhandled Errors*. Jeśli wybierzesz inną opcję, nie będą działać utworzone przez Ciebie instrukcje obsługi błędów. Więcej na ten temat możesz przeczytać w rozdziale 12.

Jeśli naprawdę jesteś zainteresowany opcjami karty General, zasięgnij informacji, klikając przycisk *Pomoc*.

Karta Docking

Na rysunku 3.10 przedstawiam kartę *Docking*. Opcje na tej karcie determinują zachowanie poszczególnych okien edytora VBE. Gdy okno jest *zadokowane*, zajmuje stałe miejsce wzdłuż jednego z brzegów okna programu VBE, dzięki czemu dużo łatwiej takie okno odszukać. Jeśli wyłączysz wszystkie opcje dokowania, na ekranie będziesz miał stertę nieuporządkowanych okien, w której trudno się odnaleźć. Domyślne ustawienia z reguły się sprawdzają.



Rysunek 3.10.
Karta Docking
okna dialogo-
wego Options

Niekiedy podczas próby zadokowania okna możesz mieć wrażenie, że VBE nie chce słuchać. Jeśli będzie Ci się wydawać, że dokowanie nie do końca działa, po prostu bądź uparty, a na pewno się uda.

Rozdział 4

Wprowadzenie do modelu obiektowego w Excelu

W tym rozdziale:

- ▶ zapoznasz się z pojęciem obiektu,
- ▶ dowiesz się więcej o hierarchii obiektów w Excelu,
- ▶ poznasz kolekcje obiektów,
- ▶ odwołasz się do określonych obiektów w instrukcjach VBA,
- ▶ odczytasz i zmienisz właściwości obiektu,
- ▶ wykonasz działania z wykorzystaniem metod obiektu.

Nikomu nie jest obce słowo **obiekt**. No i świetnie, ale teraz zapomnij o definicji tego słowa, którą znasz. W świecie programowania słowo **obiekt** znaczy coś innego. Często można je spotkać jako część wyrażenia **programowanie zorientowane obiektem**, w skrócie **OOP** (*object-oriented programming*). Zgodnie z koncepcją OOP, program komputerowy postrzegany jest jako hierarchia różnorodnych obiektów, które posiadają atrybuty (lub właściwości). Na obiektach tych można również wykonywać operacje. Tak rozumiane obiekty nie są rzecznymi materialnymi — istnieją w postaci bitów i bajtów.

W tym rozdziale objaśnię **model obiektowy w Excelu**, to znaczy hierarchię obiektów zawartych w Excelu. Gdy dojdzieś do końca rozdziału, będziesz dysponował rozsądną dawką wiedzy na temat OOP oraz zrozumiesz, dlaczego znajomość koncepcji OOP jest konieczna, by zostać programistą VBA. Bądź co bądź, programowanie w Excelu w rzeczywistości sprawdza się do operowania obiektami, z których składa się program. Nic dodać, nic ująć.

Materiał w tym rozdziale może być nieco przytłaczający. Posłuchaj jednak mojej rady i postaraj się przez niego przebrnąć, nawet jeśli nie do końca od razu wszystko zrozumiesz. Wiele ważnych kwestii, jakie tutaj poruszę, wyjaśni się w późniejszych rozdziałach książki.



Czy Excel to obiekt?

Używasz Excela od dłuższego czasu, ale prawdopodobnie nigdy o nim nie myślałeś jak o obiekcie. Im więcej będziesz pracował z VBA, tym częściej będziesz spostrzegał Excel w tych kategoriach. Zrozumiesz, że Excel jest obiektem, który sam zawiera obiekty. Te obiekty z kolei zawierają jeszcze więcej kolejnych obiektów. Inaczej mówiąc, programowanie w VBA sprowadza się do pracy z hierarchią obiektów.

U szczytu tej hierarchii znajduje się obiekt Application. W tym przypadku jest nim sam Excel, jako matka wszystkich obiektów.

Wspinaczka po hierarchii obiektów

Obiekt Application, reprezentujący Excel, zawiera inne obiekty. Poniżej znajduje się lista wybranych, najważniejszych obiektów zawartych w obiekcie Application:

- ✓ Addin,
- ✓ Window,
- ✓ Workbook,
- ✓ WorksheetFunction.

Każdy obiekt zawarty w obiekcie Application może zawierać inne obiekty. Przykładowo na poniższej liście wymienione zostały obiekty, które mogą być zawarte w obiekcie Workbook:

- ✓ Chart (będący arkuszem wykresu),
- ✓ Name,
- ✓ VBProject,
- ✓ Window,
- ✓ Worksheet.

Każdy z tych obiektów może z kolei zawierać inne obiekty. Weźmy dla przykładu obiekt Worksheet, zawarty w obiekcie Workbook, który z kolei zawarty jest w obiekcie Application. Niektóre z obiektów, jakie może zawierać obiekt Worksheet, to:

- ✓ Comment,
- ✓ Hyperlink,
- ✓ Name,
- ✓ PageSetup,
- ✓ PivotTable,
- ✓ Range.

Innymi słowy, jeśli chcesz coś zrobić z zakresem (*range*) komórek na danym arkuszu (*worksheet*), być może łatwiej będzie spojrzeć na ten zakres w następujący sposób:

Range → zawarty w Worksheet → zawarty w Workbook → zawarty w Application (Excel)

Czy to zaczyna mieć sens?



Gdy zaczniesz „wykopalską”, odkryjesz, że Excel zawiera więcej obiektów, niż mógłbyś sobie wyobrazić. Nawet weterani, tacy jak ja, mogą się poczuć przytłoczeni. Dobra wiadomość jest taka, że prawie nigdy nie będziesz potrzebować większości z tych obiektów. Pracując nad problemem, będziesz mógł się skoncentrować na kilku znaczących obiektach — te z kolei często można odkryć, rejestrując makro.

Zapoznanie z kolekcjami

Kolekcje są kolejnym kluczowym zagadnieniem programowania w VBA. Kolekcja to grupa obiektów tego samego typu. Ażeby nie było tak prosto, kolekcja sama w sobie jest obiektem.

Oto kilka przykładów powszechnie używanych kolekcji.

- ✓ **Workbooks:** kolekcja wszystkich obiektów Workbook (aktualnie otwartych skoroszytów).
- ✓ **Worksheets:** kolekcja wszystkich obiektów Worksheet (arkuszy) zawartych w obiekcie Workbook (w danym skoroszycie).
- ✓ **Charts:** kolekcja wszystkich obiektów Chart (arkuszy wykresu) zawartych w obiekcie Workbook (w danym skoroszycie).
- ✓ **Sheets:** kolekcja wszystkich arkuszy (niezależnie od ich rodzaju) zawartych w obiekcie Workbook (w danym skoroszycie).

Pewnie zauważyleś, że wszystkie nazwy kolekcji występują w liczbie mnogiej, co chyba ma sens (przynajmniej ja tak uważam).

Słusznie możesz się zastanawiać, po co właściwie są kolekcje. A więc przykładowo bardzo się przydają, gdy chcesz coś zrobić nie tylko z jednym arkuszem, lecz z kilkoma lub wszystkimi. Jak się później przekonasz, VBA zawiera instrukcje pętli, przy użyciu których możesz przejść przez każdy element kolekcji i wykonać na nim pewne działania.

Odwoływanie się do obiektów

Informacje podane w poprzednich podrozdziałach miały na celu przygotowanie do następnego tematu, czyli odwoływania się do obiektów w instrukcjach języka VBA. Umiejętność odwoływania się do obiektów jest ważna, ponieważ obiekt trzeba najpierw wskazać, aby można było z nim pracować. Bądź co bądź, VBA nie potrafi czytać w myślach — na razie. Przypuszczam, że telepatyczne obiekty zostaną wprowadzone w Excelu 2016.

Można, co prawda, wykonywać operacje na całej kolekcji obiektów za jednym zamachem, najczęściej jednak będziesz chciał pracować z konkretnym obiektem w kolekcji (na przykład z wybranym arkuszem w skoroszycie). Aby odwołać się do pojedynczego obiektu z kolekcji, za nazwą kolekcji należy podać w nawiasach nazwę obiektu lub numer indeksu, na przykład tak:

```
Worksheets("Arkusz1")
```

Zauważ, że nazwę arkusza ujęto w znaki cudzysłowy. Jeśli pominiesz cudzysłowy, Excel nie będzie w stanie zidentyfikować obiektu (założy, że jest to nazwa zmiennej).

Jeśli Arkusz1 jest pierwszym (lub jedynym) arkuszem w kolekcji, możesz również użyć odwołania:

```
Worksheets(1)
```



W tym przypadku numer indeksu *nie* jest w cudzysłowach. Wniosek? Gdy odwołujesz się do obiektu za pomocą jego nazwy, używaj znaków cudzysłowy. Gdy natomiast odwołujesz się do obiektu, podając jego numer indeksu, wpisz sam numer, bez znaków cudzysłowy.

A co z arkuszami wykresów? Arkusz wykresu zawiera pojedyńczy wykres. Posiada zakładkę arkusza, ale nie jest arkuszem. Jak się okazuje, model obiektowy Excela zawiera kolekcję o nazwie Charts. W kolekcji tej znajdują się wszystkie obiekty arkuszy wykresu istniejące w skoroszycie.

By zapewnić logikę stanu rzeczy, utworzono kolejną kolekcję o nazwie Sheets. Kolekcja Sheets zawiera wszystkie arkusze (zwykłe i arkusze wykresu) należące do skoroszytu. Kolekcja Sheets jest bardzo wygodna w użyciu, gdy chcesz pracować na wszystkich arkuszach skoroszytu niezależnie od tego, czy są to arkusze zwykłe, czy arkusze wykresu.

Z tego wynika, że pojedyńczy arkusz o nazwie Arkusz1 jest członkiem dwóch kolekcji: Worksheets oraz Sheets. Odwołać się do niego możesz na jeden z dwóch sposobów:

```
Worksheets("Arkusz1")  
Sheets("Arkusz1")
```

Nawigacja po hierarchii obiektów

Praca z obiektem Application jest bardzo łatwa. Aby zacząć, wystarczy napisać:

```
Application
```

Każdy inny obiekt w modelu obiektowym Excela podlega obiektowi Application. Dostęp do tych obiektów można uzyskać, przemieszczając się w dół hierarchii i dołączając każdy napotkany po drodze obiekt za pomocą operatora kropki (.). Aby uzyskać dostęp do obiektu Workbook o nazwie Zeszyt1.xlsx, rozpocznij od obiektu Application i jezdź w dół hierarchii do obiektu kolekcji Workbooks:

```
Application.Workbooks("Zeszyt1.xlsx")
```

Aby zejść jeszcze niżej do poziomu wybranego arkusza, dodaj operator kropki i odwołaj się do obiektu kolekcji Workseehts:

```
Application.Workbooks("Zeszyt1.xlsx").Worksheets(1)
```

Jeszcze niewystarczająco daleko? Jeśli rzeczywiście chcesz odczytać wartość komórki A1 pierwszego arkusza zawartego w skoroszcycie Zeszyt1.xlsx, musisz zejść jeszcze jeden poziom niżej i odwołać się do obiektu Range:

```
Application.Workbooks("Zeszyt1.xlsx").Worksheets(1).Range("A1").Value
```

Gdy w taki sposób odwołujesz się do obiektów, nazywane jest to odwołaniem w pełni kwalifikowanym (pełnym, jednoznacznym). Dokładnie wskazujesz Excelowi, który zakres (Range) masz na myśli, w którym arkuszu i w którym skoroszcycie, nie pozostawiając w ten sposób żadnego miejsca na domysły. Wyobraźnia jest domeną ludzi, ale lepiej, by nie działała w programach komputerowych.

Tak na marginesie: w nazwach skoroszytów również występuje kropka, która oddziela nazwę pliku od rozszerzenia (na przykład Zeszyt1.xlsx). To tylko zbieg okoliczności. Kropka występująca w nazwie pliku nie ma absolutnie nic wspólnego z operatorem kropki, o którym pisałem kilka akapitów wcześniej.

Upraszczanie odwołań do obiektów

Gdyby trzeba było używać w pełni kwalifikowanych odwołań do wszystkich obiektów, tworzony kod byłby długi, a jednocześnie pewnie mniej czytelny. Na szczęście, w Excelu istnieją krótsze sposoby odwoywania się do obiektów, dzięki czemu kod zyskuje na czytelności (a Ty nie musisz tyle pisać). Obiekt Application jest zawsze przyjmowany domyślnie. Tylko w kilku przypadkach jego wskazanie ma sens. Pominiecie odwołania do obiektu Application skraca instrukcję z poprzedniego punktu do:

```
Workbooks("Zeszyt1.xlsx").Worksheets(1).Range("A1").Value
```

To już całkiem niezłe ułatwienie. Ale czekaj, to nie wszystko. Gdy jesteś pewien, że Zeszyt1.xlsx jest aktywnym skoroszytem, możesz pominąć również to odwołanie. Teraz zostaje tylko:

```
Worksheets(1).Range("A1").Value
```

Pewnie wiesz już, do czego zmierzam. A więc jaki będzie następny krok? Oczywiście, jeśli wiesz, że pierwszy arkusz jest aktualnie aktywnym arkuszem, wówczas Excel domyślnie użycie takiego odwołania, dzięki czemu możesz napisać:

```
Range("A1").Value
```

Wbrew temu, czego mógłbyś się spodziewać, Excel nie zawiera obiektu, który reprezentowałby komórkę. Komórka jest po prostu obiektem zakresu (Range), składającego się z tylko jednej komórki.



Opisane tutaj uproszczenia odwołań są świetne, ale mogą być również niebezpieczne. Co będzie, jeśli tylko myślisz, że Zeszyt1.xlsx jest aktywnym skoroszytem? Albo zostanie zgłoszony błąd, albo, co gorsza, otrzymasz w rezultacie nieprawidłową wartość, a nawet nie będziesz tego świadom. Z tego też powodu często najlepiej stosować w pełni kwalifikowane odwołania do obiektów.

W rozdziale 14. omówię konstrukcję With-End With, dzięki której możesz używać w pełni kwalifikowanych odwołań, zachowując jednocześnie czytelność kodu i zaoszczędzając sobie pisania. To, co najlepsze z obu sposobów!

Właściwości i metody obiektów

Choć umiejętność odwoływania się do obiektów jest ważna, nie da się zrobić niczego sensownego za pomocą samego odwołania do obiektu (jak w przykładach z poprzedzających podrozdziałów). Aby osiągnąć jakiś wymierny cel, trzeba zrobić jedną lub dwie rzeczy:

- ✓ odczytać lub zmodyfikować właściwości obiektu,
- ✓ określić metodę, za pomocą której wykonana zostanie operacja na obiekcie.

VBA dysponuje dosłownie tysiącami właściwości i metod obiektów, co może przytłoczyć. Zajmuję się tym wszystkim już od lat, a nadal czuję się przytłoczony. Powtórzę jednak to, co już powiedziałem: nigdy nie będziesz potrzebować większości właściwości i metod, jakie są dostępne.

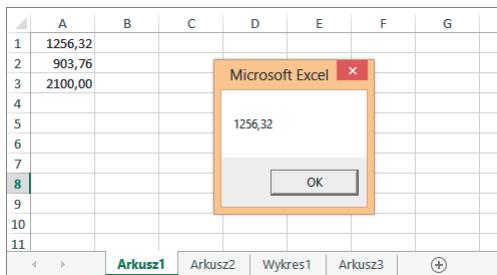
Właściwości obiektów

Każdy obiekt posiada właściwości. **Właściwości** możesz potraktować jako atrybuty opisujące obiekt, do którego należą. Właściwości obiektu określają, jak wygląda, jak się zachowuje, a nawet to, czy jest widoczny. VBA pozwala wykonywać dwie rzeczy przy wykorzystaniu właściwości:

- ✓ sprawdzać aktualną wartość (ustawienie) danej właściwości,
- ✓ zmieniać wartość właściwości.

Przykładowo obiekt jednokomórkowego zakresu Range posiada właściwość o nazwie Value. Właściwość ta przechowuje wartość, jaką zawiera komórka. Możesz napisać instrukcję VBA, aby wyświetlić właściwość Value, lub też taką instrukcję, która przypisze właściwości Value konkretną wartość. W poniższym makrze użyto wbudowanej funkcji języka VBA o nazwie MsgBox, by wyświetlić okno komunikatu z aktualną wartością komórki A1 arkusza Arkusz1 znajdującego się w aktywnym skoroszycie (zobacz też rysunek 4.1).

```
Sub ShowValue()
    Contents = Worksheets("Arkusz1").Range("A1").Value
    MsgBox Contents
End Sub
```



A screenshot of Microsoft Excel. In the center, a modal dialog box titled "Microsoft Excel" displays the value "1256,32" in its main area, with an "OK" button at the bottom. The background shows a worksheet with rows 1 through 11 and columns A through G. Row 1 contains values 1256,32, 903,76, and 2100,00. The tabs at the bottom are labeled "Arkusz1" (selected), "Arkusz2", "Wykres1", and "+".

Rysunek 4.1.
Okno komunikatu wyświetlające właściwość Value obiektu Range



Inne podejście do McObiektów, McWłaściwości i McMetod

Oto analogia, która może pomóc w zrozumieniu zależności pomiędzy obiektami, właściwościami i metodami w VBA. W analogii tej porównam Excel z siecią restauracji fast food.

Podstawową jednostką Excela jest obiekt skoroszytu (Workbook). W sieci restauracji fast food jednostką podstawową jest pojedyncza restauracja. W Excelu możesz dodać lub zamknąć skoroszyt, a wszystkie otwarte skoroszyty figurują jako Workbooks (kolekcja obiektów Workbook). Analogicznie, zarządzający siecią fast food może otworzyć restaurację lub jakąś zamknąć, a wszystkie restauracje w sieci można potraktować jak Restauracje (czyli kolekcję obiektów Restauracja).

Skoroszyt Excela jest obiektem, ale zawiera również inne obiekty, takie jak arkusze, arkusze wykresu, moduły VBA i tak dalej. Ponadto każdy obiekt w skoroszycie może zawierać swoje własne obiekty. I tak obiekt arkusza (Worksheet) może zawierać obiekty zakresu (Range), tabeli przestawnej (PivotTable), kształtu (Shape) i tym podobne.

Kontynuując nasze porównanie, mogę stwierdzić, że restauracja fast food (podobnie jak skoroszyt) zawiera obiekty, takie jak Kuchnia, Sala czy też kolekcję Stoły. Ponadto zarządzający może dodawać obiekty do obiektu Restauracja lub je z niego usuwać. Przykładowo może dodać więcej stołów do kolekcji Stoły. Każdy z tych obiektów może zawierać kolejne obiekty. Przykładowo obiekt Kuchnia posiada obiekty Kuchenka, Wentylator, SzefKuchni, Zlew i tak dalej.

Nieźle. Porównanie jest chyba całkiem udane. Zobaczmy, co z tego wyjdzie.

Obiekty Excela posiadają właściwości. Dla przykładu obiekt Range dysponuje takimi właściwościami jak Value i Name, podczas gdy obiekt Shape ma właściwości Width, Height i tak dalej. Nie zdziwi Cię, że obiekty w restauracji fast food również mają właściwości. Obiekt Kuchenka na przykład posiada takie właściwości jak Temperatura i LiczbaPalników. Obiekt Wentylator posiada własny zestaw właściwości — Włączony, ObrNaMin i tak dalej.

Obiekty Excela, poza właściwościami, posiadają również metody, które wykonują na nich pewne operacje. Przykładowo metoda ClearContents usuwa zawartość obiektu Range. Obiekty w restauracji również posiadają metody. Nietrudno wyobrazić sobie metodę ZmieńTemperaturę obiektu Kuchenka lub metodę Włącz obiektu Wentylator.

W Excelu zdarzają się metody, które zmieniają właściwości obiektu. Metoda ClearContents obiektu Range zmienia właściwość Value tego obiektu. Analogicznie, metoda ZmieńTemperaturę obiektu Kuchenka wpływa na właściwość Temperatura tego obiektu. W języku VBA można pisać procedury, które operują na obiektach, podobnie jak w restauracji kierownik może wydawać polecenia, aby oddziaływać na obiekty w restauracji („Włącz kuchenkę i podkręć wentylator”).

Gdy następnym razem pójdiesz do swojej ulubionej sieci fast food, po prostu powiedz: „Proszę użyć metody Podśmąż na obiekcie Hamburger, przy czym właściwość Cebulka proszę ustawić na Fałsz”.

Nawiasem mówiąc, MsgBox jest bardzo użyteczną funkcją. Można jej używać do wyświetlania wyników, podczas gdy Excel wykonuje instrukcje VBA.Więcej na temat tej funkcji napisz w rozdziale 15., poczekaj więc jeszcze trochę (albo przeczytaj już teraz wszystko, co znajdziesz na ten temat).

Kod z poprzedniego listingu wyświetla aktualną wartość właściwości Value zawartej w komórce. Co jednak zrobić, jeśli chcesz zmienić wartość tej właściwości? Następujące makro zmienia wartość zawartą w komórce A1, zmieniając właściwość Value tej komórki:

```
Sub ChangeValue()
    Worksheets("Arkusz1").Range("A1").Value = 994.92
End Sub
```

Po wykonaniu tej procedury wartość w komórce A1 arkusza Arkusz1 należącego do aktywnego skoroszytu wynosić będzie 994.92. Jeśli aktywny skoroszyt nie posiada arkusza o nazwie Arkusz1, wykonanie makra spowoduje wyświetlenie komunikatu o błędzie. VBA działa zgodnie z instrukcjami i nie może pracować na arkuszu, który nie istnieje.

Każdy obiekt posiada własny zestaw właściwości, mimo że niektóre z nich występują w wielu obiektach. Przykładowo wiele obiektów (lecz nie wszystkie) posiada właściwość Visible. Podobnie większość z nich posiada też właściwość Name.

Niektóre właściwości obiektów są właściwościami tylko do odczytu. Oznacza to, że wartość właściwości można odczytywać, lecz nie można jej zmieniać.

Jak już wspomniałem wcześniej w tym rozdziale, kolekcja jest również obiektem. Oznacza to, że także kolekcje posiadają właściwości. Można przykładowo określić, ile jest otwartych skoroszytów, przy użyciu właściwości Count kolekcji Workbooks. Poniższa procedura wyświetla okno komunikatu podające liczbę otwartych skoroszytów:

```
Sub CountBooks()
    MsgBox Workbooks.Count
End Sub
```

Metody obiektów

Obiekty posiadają nie tylko właściwości, lecz również metody. **Metoda** jest działaniem, jakie wykonywane jest na obiekcie. Za pomocą metody można modyfikować właściwości obiektu lub spowodować, by obiekt coś zrobił.

Oto prosty przykład, w którym użyto metody ClearContents obiektu Range, aby usunąć zawartość 12 komórek aktywnego arkusza.

```
Sub ClearRange()
    Range("A1:A12").ClearContents
End Sub
```

Niektóre metody przyjmują jeden lub więcej argumentów. **Argument** jest wartością, która bliżej określa działanie, jakie ma zostać wykonane. Argumenty metody umieszcza się za nazwą metody, oddziela się je spacją. Kilka argumentów rozdziela się przecinkiem.



Procedura z poniższego przykładu aktywuje Arkusz1 (w aktywnym skoroszycie), a następnie kopiuje zawartość komórki A1 do komórki B1 przy użyciu metody obiektu Range o nazwie Copy. W tym przykładzie metoda Copy przyjmuje jeden argument — docelowy zakres operacji kopiowania.

```
Sub CopyOne()
    Worksheets("Arkusz1").Activate
    Range("A1").Copy Range("B1")
End Sub
```

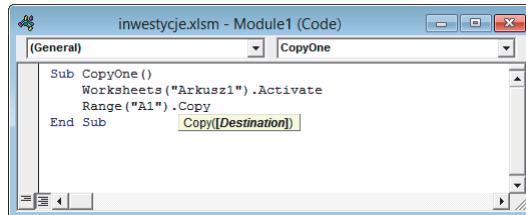
Zauważ, że odwołując się do obiektów Range, pominąłem odwołanie do arkusza. Mogłem to zrobić bez obaw, gdyż użyłem instrukcji aktywującej Arkusz1 (za pomocą metody Activate).

Innym sposobem wskazania argumentu metody jest użycie średnika i znaku równości poprzedzonych oficjalną nazwą argumentu. Używanie nazw argumentów nie jest konieczne, ale często dzięki temu pisany przez Ciebie kod może być łatwiejszy do zrozumienia. Druga instrukcja procedury CopyOne mogłaby zostać zapisana w następujący sposób.

```
Range("A1").Copy Destination:=Range("B1")
```

Zwróć uwagę na podpowiedź wyświetlającą się podczas wprowadzania instrukcji, pokazaną na rysunku 4.2. Podpowiedź ta pokazuje oficjalną nazwę argumentu.

Rysunek 4.2.
Edytor VBE wyświetla listę argumentów, gdy wprowadzasz instrukcję



Ponieważ kolekcja jest obiektem, również kolekcje posiadają metody. W następującym makrze użyta została metoda Add kolekcji Workbooks.

```
Sub AddAWorkbook()
    Workbooks.Add
End Sub
```

Jak się można spodziewać, instrukcja ta powoduje utworzenie nowego skoroszytu. Innymi słowy, do kolekcji Workbooks dodany zostaje nowy skoroszt. Po uruchomieniu makra aktywnym skoroszytem będzie nowy skoroszt.

Zdarzenia obiektów

W tym punkcie krótko poruszę jeszcze jedną rzecz, o której warto wiedzieć. Są to **zdarzenia**. Obiekty reagują na różne zdarzenia. Gdy na przykład pracujesz w Excelu i aktywujesz inny skoroszt, występuje zdarzenie Workbook Activate. Można przykładowo

napisać makro VBA wywoływanie zawsze po wystąpieniu zdarzenia `Activate` dla wybranego obiektu `Workbook`.

Excel obsługuje wiele zdarzeń, ale nie wszystkie obiekty mogą obsługiwać wszystkie zdarzenia. Są też obiekty, które nie obsługują żadnych zdarzeń. Jedynymi zdarzeniami, jakich możesz używać, są te, które zostały utworzone przez programistów Microsoft Excela. Kwestie związane ze zdarzeniami zostaną wyjaśnione w rozdziale 11., a także w części IV.

Poszukiwanie dodatkowych informacji

Można uznać, że zostałeś wprowadzony w piękny świat obiektów, właściwości, metod i zdarzeń. Więcej informacji dotyczących tych zagadnień znajdziesz w kolejnych rozdziałach. Jeśli ciągle czujesz niedosyt, możesz się zainteresować trzema innymi doskonałymi narzędziami:

- ✓ systemem pomocy VBA,
- ✓ narzędziem *Object Browser*,
- ✓ funkcjonalnością *Auto List Members*.

System pomocy VBA

W systemie pomocy języka VBA znaleźć można opisy wszystkich dostępnych obiektów, właściwości i metod. Jest to źródło informacji na temat VBA doskonale i bardziej wyczerpujące niż jakakolwiek książka na rynku. Jednocześnie jest to jednak bardzo nudna lektura.

Jeśli używasz Excela 2013, musisz mieć dostęp do internetu, aby korzystać z systemu pomocy VBA. W poprzednich wersjach ograniczenie to nie występuje.

Gdy pracujesz z modelem VBA i potrzebujesz informacji na temat wybranego obiektu, metody lub właściwości, przesuń kursor na słowo, które Cię interesuje, a następnie naciśnij F1. Po kilku sekundach Twoja przeglądarka wyświetli okno pomocy z poszukiwanymi informacjami, wraz z odsyłaczami i być może jednym lub dwoma przykładami.

Na rysunku 4.3 przedstawiam część okna systemu pomocy VBA. W tym przypadku są to informacje na temat obiektu `Worksheet`.

- ✓ Kliknij *Methods*, aby zobaczyć spis metod obiektu `Worksheet`.
- ✓ Kliknij *Properties*, aby zobaczyć kompletną listę właściwości tego obiektu.
- ✓ Kliknij *Events*, aby zobaczyć listę zdarzeń, jakie obsługuje.

Możesz użyć panelu *Contents* znajdującego się po lewej stronie, aby przeglądać informacje na różne tematy związane z obiektowym modelem Excela — doskonale zajęcie na deszczowe popołudnie.

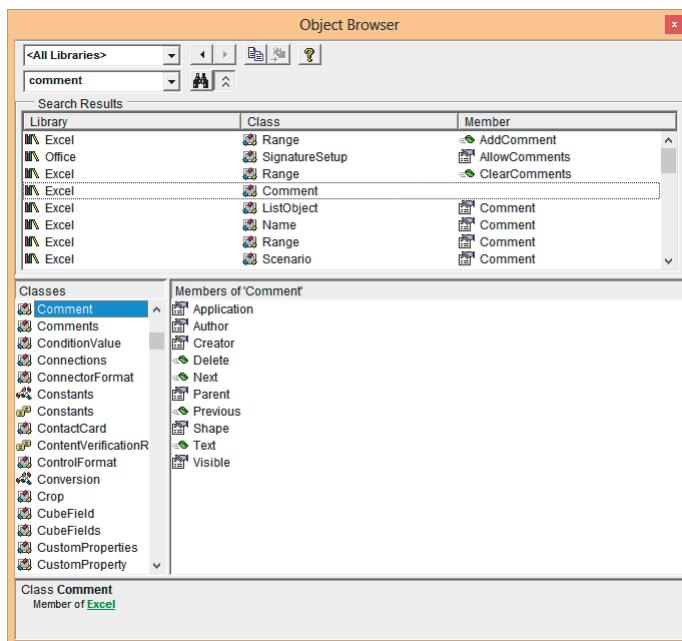


The screenshot shows the Microsoft Help interface for the 'Worksheet Members' topic. On the left, there's a navigation tree with categories like 'Office and SharePoint development', 'Office client development', etc., and under 'Worksheet Object', 'Worksheet Members' is selected. The main content area has a title 'Worksheet Members (Excel)' and a sub-header 'Office 2013 | Other Versions | 1 out of 2 rated this helpful - Rate this topic'. It defines 'Worksheet Members' as 'Represents a worksheet.' and lists 'Events', 'Methods', and 'Properties'.

Rysunek 4.3.
Przykład z systemu pomocy języka VBA

Narzędzie Object Browser

Edytor VBE wyposażony jest w narzędzie znane jako **Object Browser**. Jak sugeruje jego nazwa, narzędzie pozwala przeglądać dostępne obiekty. Aby uruchomić *Object Browser*, naciśnij klawisz F2, kiedy aktywny jest edytor VBE (lub wybierz polecenie *View/Object Browser*). Zobaczysz okno podobne do tego na rysunku 4.4.



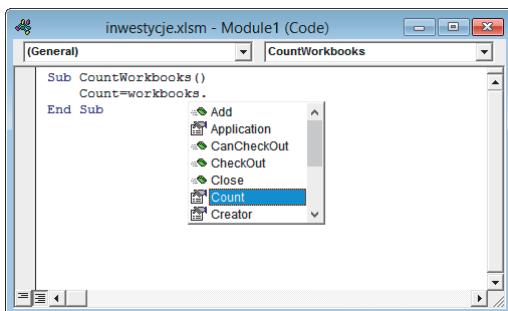
Rysunek 4.4.
Przeglądanie obiektów za pomocą narzędzia Object Browser

Lista rozwijana w górnej części okna zawiera wszystkie aktualnie dostępne biblioteki obiektów. Na rysunku 4.4 przedstawiam wszystkie biblioteki (*All Libraries*). Gdy chcesz przeglądać tylko obiekty Excela, wybierz z listy rozwijanej opcję *Excel*.

W drugiej liście rozwijanej możesz wprowadzić szukaną frazę. Jeśli na przykład chcesz zobaczyć wszystkie obiekty Excela powiązane z komentarzami, wpisz comment w pole drugiej listy rozwijanej i kliknij przycisk *Search*. Gdy zobaczyś coś, co może Cię zainteresować, zaznacz to i naciśnij *F1*, aby uzyskać więcej informacji online.

Automatyczna lista właściwości i metod

W rozdziale 3. wspomniałem o praktycznej funkcjonalności zwanej *Auto List Members*. Funkcjonalność ta odpowiada za wyświetlanie listy właściwości i metod podczas wprowadzania instrukcji. Na rysunku 4.5 przedstawiam przykład takiej listy dla kolekcji Workbooks.



Rysunek 4.5.
Funkcjonalność
Auto List
Members po-
maga rozpo-
znawać wła-
ściwości
i metody obiektu

Po wstawieniu kropki za słowem *workbooks* edytor VBE przychodzi z pomocą, wyświetlając listę właściwości i metod tej kolekcji. Po wpisaniu litery *c* lista zostaje zawężona do pozycji rozpoczęjących się od tej litery. Wybierz właściwą pozycję, naciśnij *Tab* i gotowe! Zaoszczędzisz sobie trochę pisania i ewentualnych literówek w nazwie właściwości lub metody.

Rozdział 5

Procedury Sub i Function w języku VBA

W tym rozdziale:

- ▶ zrozumiesz różnicę pomiędzy procedurami Sub i procedurami Function,
- ▶ poznasz kilka sposobów uruchamiania procedur Sub,
- ▶ poznasz dwa sposoby uruchamiania procedur Function.

W poprzednich rozdziałach kilkakrotnie wspominałem o procedurach Sub i nawiązywałem do faktu, że procedury Function również odgrywają pewną rolę w VBA. W tym rozdziale odkryję tajemnicę i wyjaśnię wszelkie wątpliwości związane z tymi pojęciami.

Procedury Sub a funkcje

Kod, który piszesz w edytorze Visual Basic, znany jest pod nazwą *procedura*. Dwoma najważniejszymi rodzajami procedur są Sub i Function.

- ✓ **Procedura Sub** to zbiór instrukcji VBA, które wykonują pewne działanie (lub działania) w Excelu.
- ✓ **Procedura Function** (inaczej **funkcja**) to zbiór instrukcji VBA, które wykonują obliczenia i zwracają pojedynczą wartość (lub tablicę).

Większość makr tworzonych w VBA to procedury Sub. Procedurę Sub można traktować tak, jakby była poleceniem — uruchomienie procedury spowoduje, że coś się wydarzy. (Co dokładnie się wydarzy, zależy — oczywiście — od instrukcji VBA zawartych w procedurze Sub.)

Funkcja jest również procedurą, ale dość istotnie różni się od procedury Sub. Pojęcie funkcji nie jest Ci obce. Excel zawiera wiele funkcji arkusza kalkulacyjnego, których używasz każdego dnia (a przynajmniej w dni pracujące). Jako przykłady można wymienić funkcje SUMA, PMT, WYSZUKAJ, PIONOWO. Funkcje te używane są w formułach. Prawie każda funkcja przyjmuje jeden lub więcej argumentów (istnieje kilka funkcji, które nie przyjmują żadnych argumentów). Funkcja taka wykonuje w tle pewne obliczenia, po czym zwraca wartość. To samo dotyczy funkcji, które tworzone są w VBA.

Rzut oka na procedury Sub

Każda procedura Sub rozpoczyna się od słowa kluczowego Sub i kończy instrukcją End Sub. Oto przykład.

```
Sub ShowMessage()
    MsgBox "To by było na tyle!"
End Sub
```

Przykład ten przedstawia procedurę o nazwie ShowMessage. Za nazwą procedury umieszczone są okrągłe nawiasy, które w większości przypadków są puste. Do procedur Sub można jednak przekazywać argumenty z innych procedur. Jeśli procedura Sub używa argumentów, wymienia się je między nawiasami.

Gdy nagrywasz makro w Excelu za pomocą rejestratora makr, w wyniku zawsze powstaje procedura Sub, która nie przyjmuje żadnych argumentów.

Jak zobacysz dalej w tym rozdziale, Excel oferuje kilka sposobów uruchamiania procedur Sub języka VBA.



Rzut oka na procedury Function

Każda procedura Function rozpoczyna się od słowa kluczowego Function i kończy instrukcją End Function. Oto prosty przykład.

```
Function CubeRoot (number)
    CubeRoot = number ^ (1 / 3)
End Function
```

Powysza funkcja nosi nazwę CubeRoot i przyjmuje jeden argument o nazwie number, zawarty w nawiasach. Funkcje mogą mieć aż 255 argumentów, mogą też nie mieć żadnego. Gdy uruchomisz tę funkcję, zwróci pojedynczą wartość — pierwiastek sześcienny argumentu przekazanego do funkcji.



VBA pozwala określić, jaki typ informacji (określany również jako typ danych) zostanie zwrócony przez procedurę Function. W rozdziale 7. znajdziesz więcej informacji na temat określania typów danych.



Procedurę Function można uruchomić tylko na dwa sposoby. Można ją wywołać z wnętrza innej procedury (Sub lub innej procedury Function) lub użyć jej w formule arkusza kalkulacyjnego.

Niezależnie od tego, jak bardzo będziesz się starać, nie zdołasz zarejestrować procedury Function, używając rejestratora makr w Excelu. Każdą procedurę Function, jaką tworzysz, musisz wprowadzić ręcznie.

Nazwy procedur Sub i Function

Podobnie jak ludzie, zwierzęta domowe i huragany, wszystkie procedury Sub i funkcje muszą się jakoś nazywać. Choć nic nie stoi na przeszkodzie, by nazwać psa Hairball Harris, tak niefrasobliwe podejście zazwyczaj nie wychodzi na dobre w przypadku nadawania nazw procedurom. Nadając nazwy procedurom, musisz przestrzegać kilku reguł.

- ✓ Możesz używać liter, cyfr oraz niektórych znaków interpunkcyjnych, lecz pierwszy znak musi być literą.
- ✓ Nazwa nie może zawierać spacji i kropek.
- ✓ Język VBA nie rozróżnia małych i dużych liter.
- ✓ Nazwa procedury nie może zawierać żadnego z następujących znaków: #, \$, %, &, @, ^, * oraz !. Innymi słowy, nazwa procedury nie może wyglądać jak przekleństwo z komiksu.
- ✓ Jeśli piszesz procedurę Function z myślą o użyciu jej w formule, unikaj nazw wyglądających jak adres komórki (na przykład A1 lub AK47). Właściwie Excel dopuszcza tego typu nazwy funkcji, ale po co komplikować sobie życie?
- ✓ Nazwy procedur nie mogą być dłuższe niż 255 znaków. (Pewnie nigdy nie przyszłyby Ci do głowy, by użyć tak długiej nazwy).

Najlepiej, gdy nazwa procedury oddaje jej cel. Dobrą praktyką jest tworzenie nazw składających się z czasownika i rzeczownika, na przykład PrzetwórzDane, WydrukujRaport, Posortuj_Tаблицę lub SprawdźNazwęPliku.

Niektórzy programiści wolą używać nazw w formie zdań, które w pełni opisują daną procedurę. Wśród przykładów mogą się znaleźć ZapiszRaportDoPlikuTekstowego czy też Odczytaj_Opcje_Drukowania_i_Wydrukuj_Raport. Użycie tak długich nazw ma swoje zalety i wady. Z jednej strony, nazwy takie bardzo dobrze opisują procedurę i są zazwyczaj jednoznaczne. Z drugiej strony, dłużej się je wprowadza. Każdy rozwija swój własny styl nadawania nazw. Tak czy inaczej, o ile makro nie ma posłużyć tylko do przetestowania czegoś na szybko, dobrym pomysłem jest nadanie mu opisowej nazwy i unikanie bardzo ogólnych nazw, takich jak ZróbTo, Aktualizuj, Napraw czy też Makrol.

Uruchamianie procedur Sub

Chociaż w tym momencie możesz jeszcze niewiele wiedzieć na temat tworzenia procedur Sub, przeskoczę trochę do przodu i napiszę, jak uruchamiać takie procedury. Jest to ważne, ponieważ procedury Sub pozostają bezużyteczne, jeśli nie wiesz, jak je uruchomić.

Przy okazji, **uruchomienie** procedury Sub oznacza to samo, co jej **wykonanie** lub **wywołanie**. Możesz używać terminologii, jaka przypadnie Ci do gustu.

Jak je uruchamiać? Procedurę Sub można wywołać na wiele sposobów. Właśnie dlatego też z ich wykorzystaniem można zrobić wiele pozytywnych rzeczy. Oto wyczerpująca lista dostępnych sposobów (a przynajmniej wszystkich tych sposobów, które mi przyszły do głowy) używanych do wywoływania procedur Sub.

- ✓ Za pomocą polecenia *Run/Run Sub/UserForm* w edytorze VBE. Excel wykonuje procedurę, w obrębie której znajduje się kursor. Istnieją dwie możliwości dla tego polecenia menu: klawisz F5 oraz przycisk *Run Sub/UserForm* na pasku narzędzi *Standard* edytora VBE. Ta metoda nie zadziała, jeśli procedura wymaga podania jednego lub większej liczby argumentów.
- ✓ Z poziomu okna dialogowego *Makro* w Excelu. Okno to można otworzyć, wybierając polecenie *Deweloper/Kod/Makra* lub *Widok/Makra/Wyświetl makra*. Można też pominać Wstążkę i po prostu użyć skrótu Alt+F8. Gdy pojawi się okno dialogowe *Makro*, wybierz konkretną procedurę Sub (makro) i kliknij przycisk *Uruchom*. Na liście tego okna pojawią się tylko te procedury, które nie przyjmują żadnych argumentów.
- ✓ Za pomocą skrótu klawiaturowego Ctrl+x lub Ctrl+Shift+x, gdzie x jest klawiszem przypisanym do danej procedury (przy założeniu, że przypisałeś skrót do danego makra).
- ✓ Klikając przycisk lub kształt w arkuszu kalkulacyjnym. Takiemu przyciskowi lub kształtowi trzeba wcześniej przypisać wybraną procedurę Sub, co bardzo łatwo można zrobić.
- ✓ Wywołując procedurę z wnętrza innej procedury Sub, którą piszesz.
- ✓ Używając przycisku dodanego do paska narzędzi *Szybki dostęp* (zobacz rozdział 19.).
- ✓ Przy użyciu własnego elementu dodanego do Wstążki (zobacz rozdział 19.).
- ✓ Poprzez wywołanie zdarzenia. Jak piszę w rozdziale 11., zdarzeniem takim może być otwarcie lub zamknięcie skoroszytu, zapisanie skoroszytu, modyfikacja komórki, aktywowanie arkusza i tym podobne.
- ✓ Z poziomu okna *Immediate* w edytorze VBE. Po prostu wpisz nazwę procedury Sub i naciśnij Enter.

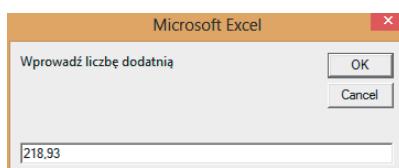
Zaprezentuję niektóre z powyższych technik w dalszej części rozdziału. Zanim jednak będę mógł to zrobić, trzeba wprowadzić procedurę Sub do modułu VBA.

1. Otwórz nowy skoroszyt.
2. Naciśnij kombinację Alt+F11, aby uruchomić VBE.
3. Wybierz skoroszyt w oknie *Project*.
4. Wybierz polecenie *Insert/Module*, aby dodać nowy moduł.
5. Do powstałego modułu wprowadź instrukcje:

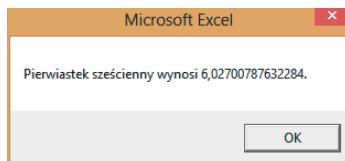
```
Sub ShowCubeRoot()
    Num = InputBox("Wprowadź liczbę dodatnią")
    MsgBox "Pierwiastek szescienny wynosi " & Num ^ (1/3) & "."
End Sub
```

Powyzsza procedura prosi użytkownika o podanie liczby, a następnie wyświetla pierwiastek szescienny tej liczby w oknie komunikatu. Na rysunkach 5.1 i 5.2 przedstawiam okna wyświetlane w czasie wykonywania procedury.

Rysunek 5.1.
Użycie wbudowanej funkcji
InputBox do wprowadzenia
liczby



Rysunek 5.2.
Wyświetlanie pierwiastka sześciennego liczby za pomocą funkcji MsgBox



Nawiasem mówiąc, *ShowCubeRoot* nie jest przykładem *dobrego* makra. Nie weryfikuje ono poprawności danych, łatwo więc doprowadzić do powstania błędu. Aby zobaczyć, co mam na myśli, spróbuj kliknąć przycisk *Cancel* w pierwszym oknie lub wprowadzić liczbę ujemną.

Bezpośrednie uruchamianie procedur Sub

Najszybszym sposobem wywołania naszej procedury jest jej uruchomienie bezpośrednio w module VBA, w którym została zdefiniowana. Wykonaj następujące kroki.

1. Uruchom edytor VBE i wybierz moduł VBA zawierający procedurę.
2. Umieść kursor w dowolnym miejscu wewnętrzku kodu procedury.
3. Naciśnij F5 (lub wybierz polecenie *Run/Run Sub/UserForm*).
4. Wprowadź liczbę w polu wyświetlonego okna i kliknij OK.

Procedura wyświetli pierwiastek sześcienny podanej liczby.

Nie można użyć polecenia *Run/Run Sub/UserForm* do wywołania procedury Sub, która przyjmuje argumenty, gdyż nie ma możliwości przekazania takich argumentów do procedury. Jedynym sposobem uruchomienia procedury przyjmującej jeden lub więcej argumentów jest wywołanie jej z poziomu innej procedury, przy czym procedura ta musi przekazać wymagane argumenty.



Uruchamianie procedur w oknie dialogowym Makro

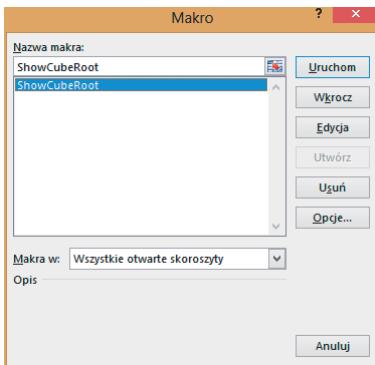
W większości przypadków procedury Sub uruchamiane są nie w edytorze VBE, lecz w Excelu. Postępuj według poniższej instrukcji, aby uruchomić makro z poziomu okna dialogowego *Makro* w Excelu.

1. Aktywuj okno Excela.

Możesz użyć ekspresowej metody *Alt+F11*.

2. Wybierz *Deweloper/Kod/Makra* (lub naciśnij *Alt+F8*).

Excel wyświetli okno dialogowe widoczne na rysunku 5.3.



Rysunek 5.3.
Okno dialogowe Makro zawiera listę wszystkich dostępnych procedur Sub

3. Wybierz makro.

4. Naciśnij przycisk *Uruchom* (albo kliknij dwukrotnie nazwę makra na liście makr).

Okno dialogowe *Makro* nie wyświetla procedur *Sub*, które przyjmują argumenty. Przyczyną tego jest brak możliwości przekazania argumentów.



Uruchamianie makr za pomocą skrótów klawiszowych

Innym sposobem na wywołanie makra jest użycie odpowiedniego skrótu klawiszowego. Aby jednak skorzystać z tej metody, trzeba najpierw taki skrót przypisać do makra.

Możesz wykorzystać okazję, by przypisać skrót klawiszowy w oknie dialogowym *Rejestrowanie makra*, gdy zaczynasz rejestrować makro. Jeśli tworzysz procedurę bez użycia rejestratora makr, skrót możesz przypisać (lub go zmodyfikować, jeśli już istnieje), postępując według następującej instrukcji.

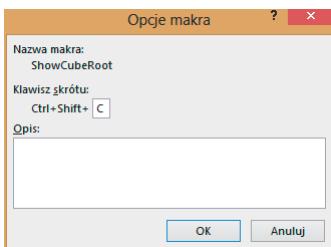
1. Wybierz polecenie *Deweloper/Kod/Makra*.

2. Wybierz odpowiednią nazwę procedury *Sub* z listy makr.

W tym przypadku procedura nosi nazwę *ShowCubeRoot*.

3. Kliknij przycisk *Opcje*.

Excel wyświetli okno dialogowe przedstawione na rysunku 5.4.



Rysunek 5.4.
Okno dialogowe
Opcje makra
powalą określić
opcje makr

4. Wprowadź wybraną literę w polu opcji *Klawisz skrótu* obok napisu *Ctrl+*.

Litera, którą wprowadzisz, odpowidać będzie kombinacji klawiszy, jakiej chcesz używać do uruchamiania makra. Jeśli na przykład wprowadzisz małe c, makro będziesz mógł uruchamiać, naciskając klawisze *Ctrl+C*. Jeśli podasz dużą literę, kombinacja klawiszy będzie dodatkowo zawierać *Shift*. Jeśli przykładowo wprowadzisz C, makro będziesz mógł uruchamiać, naciskając *Ctrl+Shift+C*.

5. Naciśnij *OK* lub *Anuluj*, aby zamknąć okno dialogowe *Opcje makra*.

Po przypisaniu skrótu klawiszowego możesz używać wybranej kombinacji klawiszy, aby uruchomić makro. Skrót klawiszowy nie zadziała, jeżeli jest przypisany do makra, w którym użyto argumentów.



Skróty, które przypisujesz makrom, nadpisują wbudowane skróty klawiszowe Excela. Jeśli przykładowo przypiszesz do makra kombinację *Ctrl+C*, nie będziesz mógł jej używać do kopiowania danych w skoroszycie. Nie jest to wielkim problemem, ponieważ Excel prawie zawsze udostępnia inne metody wykonywania poleceń.

Uruchamianie procedur przy użyciu przycisków i kształtów

Może przapaść Ci do gustu pomysł, by przypisać makro do przycisku (lub dowolnego innego kształtu) umieszczonego na arkuszu. Aby przypisać makro do przycisku, wykonaj następujące kroki.

1. Aktywuj dowolny skoroszyt.

2. Dodaj przycisk z grupy *Kontrolki formularza*.

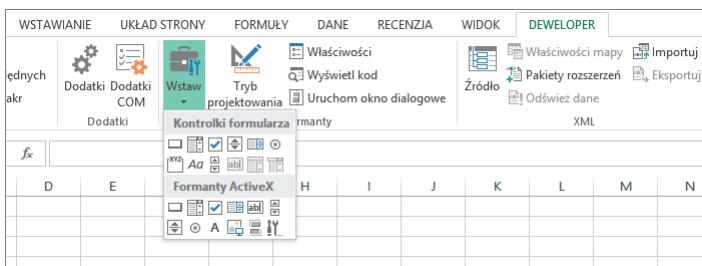
Aby wyświetlić grupę *Kontrolki formularza*, wybierz *Deweloper/Formanty/Wstaw* (zobacz rysunek 5.5).

3. Kliknij narzędzie *Przycisk* w grupie *Kontrolki formularza*.

Jest to pierwszy przycisk w pierwszym rzędzie kontrolek.

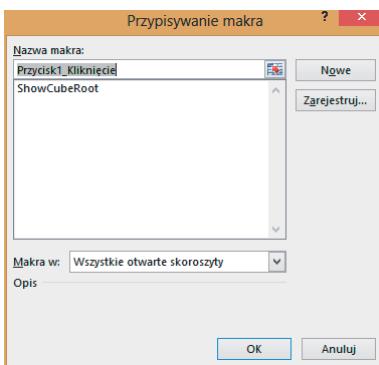
4. Przeciągnij kurSOR myszy w wybranym miejscu arkusza, by utworzyć przycisk.

Rysunek 5.5.
Kontrolki dostępne po kliknięciu poleceń Wstaw z karty Deweloper na Wstążce



Gdy już umieścisz przycisk na arkuszu, Excel — czytając w Twoich myślach — wyświetli okno dialogowe *Przypisywanie makra*, widoczne na rysunku 5.6.

Rysunek 5.6.
Gdy do arkusza wstawiasz przycisk, Excel automatycznie wyświetla okno dialogowe Przypisywanie makra



5. Wybierz makro, które chcesz przypisać do przycisku.

6. Kliknij OK.

Po przypisaniu makra do przycisku jego kliknięcie spowoduje uruchomienie makra. Istne czary!

Zauważ, że okno dialogowe *Przypisywanie makra* umożliwia również zarejestrowanie nowego makra (przycisk *Zarejestruj...*). Możesz też kliknąć przycisk *Nowe*, a Excel wstawi nową procedurę *Sub* o nazwie podanej przez Ciebie w tym oknie. Na ogół jednak do przycisku przypisuje się *istniejące* makro.

Zauważ, że podczas wstawiania przycisków wyświetlane są dwa zestawy kontrolek: *Kontrolki formularza* i *Formanty ActiveX*. Obie grupy wyglądają podobnie, ale bardzo się różnią. W praktyce *Kontrolki formularza* są łatwiejsze w użyciu.

Makro można przypisać również do dowolnego innego kształtu lub obiektu. Dla przykładu załóżmy, że chciałbyś, by makro było uruchamiane, gdy użytkownik kliknie obiekt *Prostokąt*.

1. Wstaw prostokąt do arkusza.

Aby wstawić prostokąt, użyj polecenia *Wstawianie/Ilustracje/Kształty*, z menu wybierz *Prostokąt* i narysuj go w arkuszu.



2. Kliknij prostokąt prawym przyciskiem myszy.
3. Z menu podręcznego wybierz polecenie *Przypisz makro*.
4. Wskaż wybrane makro w oknie dialogowym *Przypisywanie makra*.
5. Kliknij OK.

Po wykonaniu powyższych kroków kliknięcie prostokąta spowoduje uruchomienie przypisanego makra.

Uruchamianie procedur z poziomu innych procedur

Procedurę można uruchomić również z wnętrza innej procedury. Jeśli chcesz wypróbować ten sposób, wykonaj następujące kroki.

1. Aktywuj moduł VBA zawierający procedurę ShowCubeRoot.
2. Dodaj nową procedurę, wprowadzając poniższy kod (pod lub nad kodem procedury ShowCubeRoot — to obojętne).

```
Sub NewSub()  
    Call ShowCubeRoot  
End Sub
```

3. Uruchom makro NewSub.

Najłatwiej to zrobić, umieszczając kurSOR gdziekolwiek w obrębie kodu procedury NewSub i naciskając klawisz F5. Zauważ, że procedura NewSub wywołuje jedynie procedurę ShowCubeRoot.

Nawiasem mówiąc, słowo kluczowe Call jest opcjonalne. Instrukcja może składać się z samej nazwy procedury Sub. Uważam jednak, że użycie słowa kluczowego Call doskonale uwidacznia fakt, że wywoływaną jest procedura.

Uruchamianie procedur Function

Funkcje, w odróżnieniu od procedur Sub, mogą być wykonywane na tylko dwa sposoby:

- ✓ poprzez wywołanie funkcji z wnętrza innej procedury Sub lub Function,
- ✓ poprzez użycie funkcji w formule arkusza kalkulacyjnego.

Wypróbowujemy to na takiej oto prostej funkcji. Wprowadź ją do modułu VBA:

```
Function CubeRoot(number)  
    CubeRoot = number ^ (1/3)  
End Function
```

Dość skąpa ta funkcja — oblicza zaledwie pierwiastek sześcienny liczby przekazanej do niej jako argument. Mimo to jest dobrym punktem wyjścia na drodze do zrozumienia istoty funkcji, a ponadto ilustruje ważne zagadnienie związane z funkcjami; chodzi o to, jak zwrócić wartość. (Pamiętasz, że funkcje zwracają wartość, prawda?).

Zwróć uwagę, że jedyny wiersz kodu, który tworzy ciało funkcji, wykonuje pewne obliczenie. Wynik obliczeń (liczba podniesiona do potęgi $\frac{1}{6}$) zostaje przypisany do zmiennej `CubeRoot`. Nieprzypadkowo `CubeRoot` jest również nazwą funkcji. Aby wskazać funkcji, którą wartość ma zwrócić, należy przypisać tę wartość do zmiennej o nazwie odpowiadającej nazwie funkcji.

Wywoływanie funkcji z poziomu procedur Sub

Ponieważ nie istnieje możliwość bezpośredniego wywoływania funkcji, trzeba je wywoływać z poziomu innych procedur. Wprowadź kod poniższej prostej procedury do tego samego modułu VBA, który zawiera funkcję `CubeRoot`.

```
Sub CallerSub()
    Ans = CubeRoot(125)
    MsgBox Ans
End Sub
```

Gdy uruchomisz procedurę `CallerSub` (używając dowolnej z metod opisanych wcześniej w tym rozdziale), Excel wyświetli okno z komunikatem zawierającym wartość zmiennej `Ans`, czyli 5.

Oto, co się dzieje. Wywołana zostaje funkcja `CubeRoot`, której przekazany zostaje argument 125. Kod funkcji wykonuje obliczenia (na podstawie wartości przekazanej jako argument), a zwrócona przez funkcję wartość zostaje przypisana do zmiennej `Ans`. Następnie funkcja `MsgBox` wyświetla wartość zmiennej `Ans`.

Spróbuj zmienić argument przekazywany do funkcji `CubeRoot` i powtórnie uruchom makro `CallerSub`. Działa dokładnie tak, jak powinno — o ile tylko przekażesz funkcji prawidłowy argument (liczbę dodatnią).

Tak na marginesie, procedurę `CallerSub` można nieco uprościć. Zmienna `Ans` nie jest konieczna, o ile nie będziesz z niej korzystał w dalszej części kodu. Używając poniższej pojedynczej instrukcji, otrzymałbyś taki sam rezultat:

```
MsgBox CubeRoot(125)
```

Wywoływanie funkcji z poziomu formuł arkusza

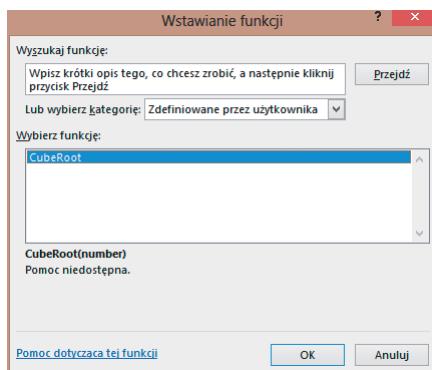
Nadszedł czas, by wywołać naszą funkcję VBA przy użyciu formuły arkusza kalkulacyjnego. Aktywuj dowolny arkusz tego samego skoroszytu, w którym zdefiniowana została funkcja `CubeRoot`. Następnie do dowolnej komórki wprowadź formułę:

```
=CubeRoot(1728)
```

W komórce wyświetlona zostanie liczba 12, która rzeczywiście jest pierwiastkiem sześciennym liczby 1728.

Jak pewnie się domyślasz, jako argumentu funkcji CubeRoot możesz użyć odwołania do komórki. Jeżeli przykładowo komórka A1 zawiera wartość, możesz wprowadzić formułę =CubeRoot(A1). W tym przypadku funkcja zwróci liczbę otrzymaną przez obliczenie pierwiastka sześciennego z wartości podanej w komórce A1.

Funkcję tę możesz stosować w arkuszu wielokrotnie. Podobnie jak wbudowane funkcje Excela, również funkcje utworzone przez Ciebie będą się pojawiały w oknie dialogowym *Wstawianie funkcji*. Kliknij przycisk *Wstaw funkcję* znajdujący się na pasku narzędzi i wybierz kategorię *Zdefiniowane przez użytkownika*. Jak pokazano na rysunku 5.7, na liście w oknie dialogowym *Wstawianie funkcji* widoczna jest Twoja własna funkcja.



Rysunek 5.7.
Funkcja Cube-
Root pojawia
się w kategorii
Zdefiniowane
przez użytkow-
nika okna dia-
logowego
Wstawianie
funkcji



Jeśli chcesz, aby w oknie dialogowym *Wstawianie funkcji* wyświetlany był opis funkcji, postępuj według poniższej instrukcji.

1. Wybierz polecenie *Developer/Kod/Makra*.

Excel wyświetli okno dialogowe *Makro*, ale funkcja CubeRoot nie pojawi się na liście. (CubeRoot jest procedurą Function, lista natomiast zawiera tylko procedury Sub). Bez obaw, czytaj dalej.

2. Wpisz słowo CubeRoot w polu *Nazwa makra*.

3. Kliknij przycisk *Opcje*.

4. Wprowadź opis funkcji w polu tekstowym *Opis*.

5. Kliknij *OK*, aby zamknąć okno dialogowe *Opcje makra*.

6. Zamknij okno dialogowe *Makro*, klikając przycisk *Anuluj*.

Opis, który wprowadziłeś, będzie się teraz pojawiał w oknie dialogowym *Wstawianie funkcji*.

Na rysunku 5.8 przedstawiam użycie funkcji CubeRoot w formułach arkusza.

92 Część II: Jak VBA współpracuje z Exceliem?

Rysunek 5.8.
Użycie funkcji
CubeRoot
w formułach

The screenshot shows a Microsoft Excel spreadsheet with data in columns A and B. Column A contains integers from 1 to 10. Column B contains the result of applying the CubeRoot function to each value in column A. The formula bar at the top shows the formula =CubeRoot(A4). The cell B4 is highlighted in green, indicating it is the active cell. The status bar at the bottom shows the text "Arkusz1".

B4	A	B	C	D	E	F	G
	1	1					
	2	1,259921					
	3	1,44225					
	4	1,587401					
	5	1,709976					
	6	1,817121					
	7	1,912931					
	8	2					
	9	2,080084					
	10	2,154435					
	11						

Być może już wszystko zaczyna składać się w całość. (Szkoda, że ja nie miałem tej książki, gdy zaczynałem). Dowiedziałeś się wiele o procedurach Sub i Function. Makra zaczniesz tworzyć w rozdziale 6., w którym szczegółowo omówię kwestie związane z opracowywaniem makr przy użyciu rejestratora makr w Excelu. W rozdziale 20. ujawnię jeszcze więcej informacji na temat funkcji.

Rozdział 6

Używanie rejestratora makr

W tym rozdziale:

- ▶ nagrasz wykonywane w Excelu czynności za pomocą wbudowanego rejestratora makr,
 - ▶ poznasz rodzaje makr, jakie można zarejestrować,
 - ▶ nauczysz się wybierać odpowiednie ustawienia przed rejestrowaniem makr.
-

Makro działające w Excelu można utworzyć na dwa sposoby:

- zarejestrować je przy użyciu rejestratora makr w Excelu,
- napisać je manualnie.

Rozdział ten poświęcony jest wszelkim aspektom związanym z używaniem rejestratora makr w Excelu. Rejestrowanie makr nie zawsze jest najlepszym podejściem, a niektórych makr po prostu nie da się zarejestrować, niezależnie od włożonego wysiłku. Zobaczysz jednak, że rejestrator makr jest, mimo wszystko, bardzo przydatny. Jeśli nawet zarejestrowane makro nie do końca będzie tym, do czego zmierzasz, rejestrator makr prawie zawsze poprowadzi Cię we właściwym kierunku.

Czy to rzeczywistość, czy to VBA?

W poprzednich edycjach tej książki porównywałem rejestrowanie makra do nagrywania za pomocą magnetyfonu. Przyszło mi jednak do głowy, że magnetofony szybko podążają śladami dinozaurów. Z tego też powodu uaktualniłem ten podrozdział — teraz rejestrowanie makr porównane zostanie do nagrywania cyfrowego filmu wideo. Analogia ta, podobnie jak poprzednia, jest trafna w kilku aspektach. W tabeli 6.1 rejestrowanie makra porównane zostało z nagrywaniem filmu.

Podstawy rejestrowania makr

Aby nagrać makro, wykonać należy kilka kroków podstawowych, które wymieniam poniżej. Kroki te opiszę bardziej szczegółowo w dalszej części rozdziału.

- 1. Określ, co dokładnie ma robić makro.**
- 2. Upewnij się, że wybrałeś prawidłowe ustawienia.**

Od tego kroku zależy, jak dobrze będzie działało makro.

Tabela 6.1. Nagrywanie filmu wideo a rejestrowanie makr

	Kamera wideo	Rejestrator makr w Excelu
Jakie wyposażenie jest potrzebne?	Kamera wideo	Komputer z zainstalowanym Exceliem
Co jest nagrywane?	Obraz i dźwięk	Działania wykonywane w Excelu
Gdzie przechowywane jest nagranie?	Na karcie pamięci	W module VBA
Jak można odtworzyć nagranie?	Odnajdując plik i naciskając przycisk <i>Odtwórz</i>	Odnajdując makro w oknie dialogowym <i>Makro</i> i klikając przycisk <i>Uruchom</i> (lub przy użyciu innej metody)
Czy można edytować nagranie?	Tak, jeśli dysponujesz odpowiednim oprogramowaniem do edycji nagrä wideo	Tak, jeśli wiesz, co robisz
Czy można skopiać nagranie?	Tak, dokładnie tak samo jak dowolny inny plik	Tak (nie są potrzebne dodatkowe narzędzia)
Czy nagranie spełnia oczekiwania?	Zależy od sytuacji i jakości wykorzystanego sprzętu	Zależy od użytych ustawień rejestratora makr
Co zrobić, jeśli popełniony zostanie błąd?	Ponownie nagrać wideo (lub je edytować, jeśli jest to możliwe)	Ponownie zarejestrować makro (lub je edytować, jeśli jest to możliwe)
Czy można uzyskać wgląd w nagranie?	Tak, otwierając plik w odpowiednim oprogramowaniu	Tak, otwierając moduł w edytorze VBE
Czy można udostępnić nagranie innym?	Tak, dobrym sposobem jest YouTube	Tak, zamieść je w poście na blogu lub innej stronie internetowej
Czy można zarobić pieniądze na utworzonym nagraniu?	Tak, jeśli jest naprawdę dobre (wymaga zazwyczaj edycji)	Możliwe, trzeba jednak najpierw spędzić wiele czasu na edycji

3. Ustal, czy odwołania do komórek w makrze mają być względne, czy bezwzględne.
 4. Kliknij przycisk *Zarejestruj makro* w lewej części paska statusu (lub wybierz polecenie *DEVELOPER/Kod/Zarejestruj makro*).
- Excel wyświetli okno dialogowe *Rejestrowanie makra*.
5. Określ nazwę makra, klawisz skrótu, miejsce przechowywania makra i jego opis.
 6. Kliknij przycisk **OK** w oknie dialogowym *Rejestrowanie makra*.

Excel automatycznie wstawi nowy moduł VBA. Od tego momentu będzie też konwertował wykonywane przez Ciebie czynności na instrukcje VBA. Na pasku statusu wyświetlany będzie kwadratowy przycisk zatrzymujący rejestrowanie makra.

7. Wykonaj czynności, które chcesz zarejestrować, posługując się myszą czy też klawiaturą.
 8. Gdy skończysz, kliknij kwadratowy biały przycisk na pasku statusu (lub wybierz polecenie **DEVELOPER/Kod/Zatrzymaj rejestrowanie**).
- Excel zatrzyma rejestrowanie wykonywanych przez Ciebie czynności.
9. Przetestuj makro, by się upewnić, że działa prawidłowo.
 10. Opcjonalnie możesz uporządkować kod, usuwając nadmiarowe instrukcje.

Rejestrator makr sprawdza się najlepiej przy prostych, niezawiłych makrach. Możesz przykładowo zarejestrować makro, które będzie odpowiednio formatować wybrany zakres komórek lub tworzyć nagłówki kolumn i wierszy w nowych arkuszach.

Za pomocą rejestratora makr można tworzyć tylko procedury Sub. Nie ma możliwości tworzenia procedur Function przy użyciu rejestratora makr.

Rejestrator makr może się również przydać podczas tworzenia bardziej złożonych procedur. Często się zdarza, że rejestruję niektóre czynności po to, by następnie skopiować wygenerowany kod do innego, bardziej złożonego makra. W większości przypadków trzeba wprowadzić pewne zmiany w wygenerowanym kodzie i dodać kilka nowych instrukcji VBA.

Rejestrator makr *nie może* wygenerować kodu dla żadnego z wymienionych niżej zadań (opiszę je dalej w tej książce), takich jak:

- ✓ wykonywanie wszelkiego rodzaju powtarzających się pętli instrukcji,
- ✓ wykonywanie wszelkiego rodzaju instrukcji warunkowych (przy użyciu konstrukcji If-Then),
- ✓ przypisywanie zmiennym wartości,
- ✓ definiowanie typów danych,
- ✓ wyświetlanie okien pop-up,
- ✓ wyświetlanie własnych okien dialogowych.

Ograniczone możliwości rejestratora makr nie umniejszają jego znaczenia. W całej książce podkreślam, że *rejestrowanie wykonywanych czynności jest być może najlepszym sposobem na opanowanie VBA*. Gdy tylko masz wątpliwości, spróbuj użyć rejestratora. Choć otrzymany rezultat może być niedokładnie tym, do czego zmierzałeś, analiza wygenerowanego automatycznie kodu może Cię poprowadzić we właściwym kierunku.

Przygotowania do rejestrowania makr

Zanim wykonasz wielki krok i uruchomisz rejestrator makr, zastanów się przez chwilę, co chcesz zrobić. Rejestrujesz makro po to, by Excel mógł automatycznie powtórzyć zarejestrowane czynności, muszą więc one być precyzyjne.



Ostateczna przydatność zarejestrowanego makra zależy od pięciu czynników.

- ✓ Jak skonfigurowane były ustawienia w skoroszycie, gdy rejestrowałeś makro?
- ✓ Co było zaznaczone, gdy uruchomiłeś rejestrator makr?
- ✓ Czy rejestrowałeś w trybie odwołań względnych, czy bezwzględnych?
- ✓ Czy odpowiednio wykonywałeś rejestrowane czynności?
- ✓ Jaki był kontekst, w którym uruchomiłeś zarejestrowane makro?

Znaczenie tych czynników stanie się jasne, gdy popracujesz z przykładami.

Względne czy bezwzględne?

Gdy rejestrujesz wykonywane czynności, Excel zazwyczaj używa bezwzględnych odwołań do komórek. (Jest to domyślny tryb rejestrowania makr). Często jednak jest to *niewłaściwy* tryb. Gdy stosujesz tryb odwołań bezwzględnych, Excel rejestruje konkretne adresy komórek. Gdy używasz trybu odwołań względnych, Excel rejestruje względne odwołania do komórek. Czytaj dalej, aby zobaczyć, jaka jest różnica.

Rejestrowanie makr w trybie odwołań bezwzględnych

Wykonaj poniższe kroki, aby zarejestrować proste makro w trybie odwołań bezwzględnych. Makro to wprowadza po prostu do arkusza nazwy trzech miesięcy.

1. Upewnij się, że przycisk **DEVELOPER/Kod/Użyj odwołań względnych** nie jest podświetlony, a następnie wybierz polecenie **DEVELOPER/Kod/Zarejestruj makro**.
2. **Jako nazwę makra wpisz** Absolute.
3. **Kliknij OK, aby rozpocząć rejestrowanie.**
4. **Aktywuj komórkę B1 i wpisz do niej** Styczeń.
5. **Przemieś się do komórki C1 i wpisz** Luty.
6. **Przemieś się do komórki D1 i wpisz** Marzec.
7. **Kliknij komórkę B1, aby powtórnie ją aktywować.**
8. **Zatrzymaj rejestrator makr.**
9. **Naciśnij Alt+F11, aby uruchomić VBE.**
10. **Spójrz na kod w module Module1.**

Excel wygeneruje poniższy kod.

```
Sub Absolute()
```

```
'
```

```
' Absolute Makro
'
Range("B1").Select
ActiveCell.FormulaR1C1 = "Styczeń"
Range("C1").Select
ActiveCell.FormulaR1C1 = "Luty"
Range("D1").Select
ActiveCell.FormulaR1C1 = "Marzec"
Range("B1").Select
End Sub
```

Po uruchomieniu makro wybiera komórkę B1 i wprowadza nazwy trzech kolejnych miesięcy do komórek z zakresu B1:D1. Następnie makro powtórnie aktywuje komórkę B1.

Te same działania wykonywane są niezależnie od tego, która komórka jest aktywna w momencie uruchomienia makra. Uruchamiając makro zarejestrowane w trybie odwołań bezwzględnych, w rezultacie zawsze otrzymamy dokładnie takie same efekty. W tym przypadku makro zawsze będzie wprowadzać nazwy pierwszych trzech miesięcy do zakresu B1:D1.

Rejestrowanie makr w trybie odwołań względnych

W niektórych przypadkach będziesz chciał, aby rejestrowane makro operowało na *względnych odwołaniach* do komórek. Możesz przykładowo chcieć, aby makro rozpoczęło wprowadzanie nazw miesięcy od komórki, która w danym momencie jest aktywna. W takim przypadku musisz zarejestrować makro w trybie odwołań względnych.

Tryb rejestrowania wykonywanych działań w Excelu możesz zmienić, klikając przycisk *Użyj odwołań względnych* w grupie *Kod* karty *DEVELOPER*. Gdy przycisk oznaczony jest innym kolorem, aktywny jest tryb odwołań względnych. Gdy kolor przycisku nie różni się od pozostałych, rejestrowanie odbywa się w trybie odwołań bezwzględnych.

Tryb rejestrowania możesz zmieniać w dowolnym czasie, również w trakcie rejestrowania makra.

Aby zobaczyć, jak działa rejestrowanie przy użyciu odwołań względnych, usuń zawartość komórek z zakresu B1:D1 i wykonaj następujące czynności.

1. **Aktywuj komórkę B1.**
2. **Wybierz polecenie *DEVELOPER/Kod/Zarejestruj makro*.**
3. **Jako nazwę makra wprowadź *Relative*.**
4. **Kliknij OK, aby rozpocząć rejestrowanie.**
5. **Kliknij przycisk *Użyj odwołań względnych*, aby przełączyć tryb rejestrowania na względny.**

Po kliknięciu tego przycisku jego kolor zmienia się na inny niż kolor pozostałych elementów Wstążki.



6. **Wpisz** Styczeń do komórki B1.
7. **Przejdź do komórki C1 i wpisz** Luty.
8. **Przejdź do komórki D1 i wpisz** Marzec.
9. **Zaznacz komórkę** B1.
10. **Zatrzymaj rejestrator makr.**

Zauważ, że sposób postępowania różni się nieco od poprzedniego przykładu. W tym przykładzie komórka wyjściowa jest aktywowana *przed* rozpoczęciem rejestrowania. Jest to ważny krok, gdy rejestrowane makro ma używać aktywnej komórki jako punktu wyjścia.

Powyzsze makro zawsze będzie rozpoczynać wprowadzanie tekstu od aktualnie aktywnej komórki. Przetestuj to. Przesuń wskaźnik komórki do dowolnej komórki i uruchom makro Relative. Nazwy miesięcy zawsze będą wprowadzane od aktywnej komórki.

Gdy podczas rejestrowania aktywny jest tryb odwołań względnych, wygenerowany kod dość znacznie różni się od kodu generowanego w trybie odwołań bezwzględnych.

```
Sub Relative()  
    ' Relative Makro  
  
    ActiveCell.FormulaR1C1 = "Styczeń"  
    ActiveCell.Offset(0, 1).Range("A1").Select  
    ActiveCell.FormulaR1C1 = "Luty"  
    ActiveCell.Offset(0, 1).Range("A1").Select  
    ActiveCell.FormulaR1C1 = "Marzec"  
    ActiveCell.Offset(0, -2).Range("A1").Select  
End Sub
```

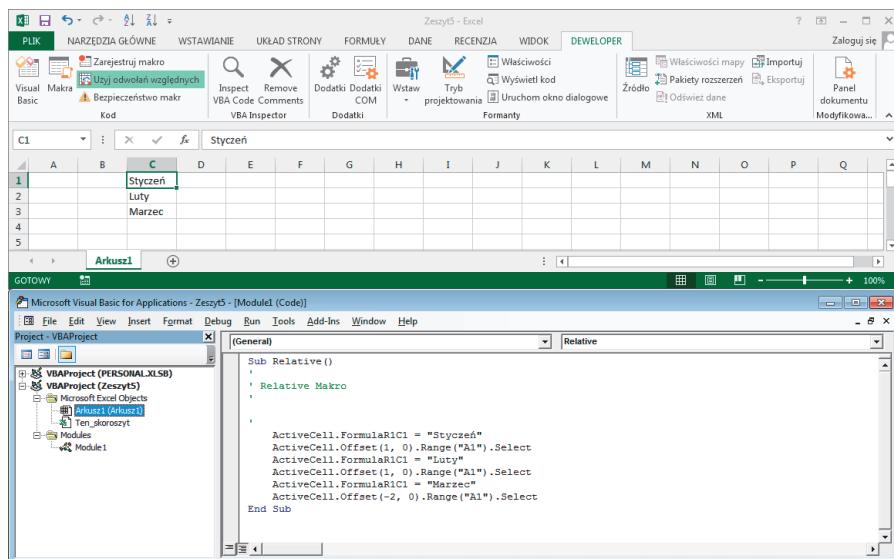
Aby przetestować to makro, zaznacz dowolną komórkę poza B1. Nazwy miesięcy zostaną wprowadzone do trzech komórek, a wprowadzanie zacznie się od komórki, którą aktywowałeś.

Zauważ, że kod wygenerowany przez makro odwołuje się do komórki A1. Może to wydawać się dziwne, ponieważ w ogóle nie używałeś tej komórki, gdy rejestrowałeś makro. Jest to po prostu efekt ubocznego sposobu działania rejestratora makr. (Kwestię tę omówię bliżej w rozdziale 8., gdzie opisana zostanie metoda Offset).



Co jest rejestrowane?

Gdy uruchamiasz rejestrator makr, Excel zaczyna konwertować wykonywane myszą i klawiaturą czynności na poprawne instrukcje VBA. Mógłbym pewnie napisać kilka stron opisu, jak Excel to robi, jednak najlepszym sposobem, by zrozumieć ten proces, jest obserwowanie rejestratora makr w akcji. (Na rysunku 6.1 zobaczysz, jak może wyglądać ekran w czasie rejestrowania makra).

**Rysunek 6.1.**

Rozmieszczenie okien wygodne podczas obserwacji działań rejestratora makr

Wykonaj następujące kroki.

- Utwórz nowy skoroszyt.**
- Upewnij się, że okno Excela nie jest zmaksymalizowane.**
- Naciśnij *Alt+F11*, aby uruchomić VBE (i upewnij się, że okno tego programu również nie jest zmaksymalizowane).**
- Dostosuj rozmiar i położenie okien Excela i VBE tak, aby oba były widoczne.**
Najlepiej będzie, jeśli okno Excela umieścisz nad oknem VBE i zminimalizujesz okna wszystkich innych uruchomionych aplikacji.
- Aktywuj okno Excela i wybierz polecenie *DEVELOPER/Kod/Zarejestruj makro*.**
- Kliknij OK, aby uruchomić rejestrator makr.**
Excel wstawi nowy moduł (o nazwie Module1) i zacznie rejestrować w nim instrukcje.
- Aktywuj okno edytora VBE.**
- Kliknij dwukrotnie Module1 w oknie Project Explorer, aby wyświetlić ten moduł w oknie Code.**

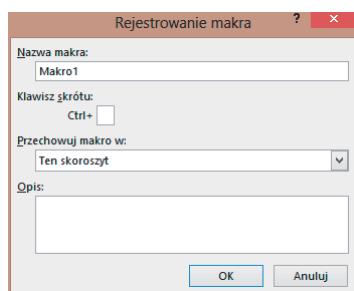
Przeskocz teraz z powrotem do Excela i pobaw się w nim chwilę. Wybieraj różne polecenia i obserwuj przy tym instrukcje generowane w oknie VBE. Zaznacz komórki, wprowadzaj dane, formatuj komórki, używaj poleceń Wstążki, utwórz wykres, zmieniaj szerokość kolumn, wykonuj operacje na obiektach graficznych i tak dalej. Po prostu poszalej! Gwarantuję, że będziesz oszołomiony, widząc na własne oczy, jak Excel wypluuwa instrukcje VBA.



Jeśli tak się składa, że masz komputer z dwoma monitami, może wygodniej Ci będzie umieścić okno Excela na jednym z nich, a okno VBE na drugim. Ja zazwyczaj tak właśnie robię.

Opcje rejestratora makr

Gdy rejestrujesz wykonywane czynności, aby wygenerować kod VBA, do dyspozycji masz kilka opcji. Przypominasz sobie, że wybranie polecenia *DEVELOPER/Kod/Zarejestruj makro* powoduje wyświetlenie okna dialogowego *Rejestrowanie makra* (zobacz rysunek 6.2), jeszcze zanim rozpoczęcie się rejestrowanie.



Rysunek 6.2.
Okno dialogowe
Rejestrowanie
makra udostępnia
kilka opcji

Okno dialogowe *Rejestrowanie makra*, przedstawione na rysunku 6.2, pozwala określić kilka aspektów makra. Opcje te opisuję w kolejnych punktach rozdziału.

Nazwa makra

W tym polu możesz wprowadzić nazwę dla procedury Sub, którą będziesz rejestrował. Domyślnie Excel stosuje nazwy Makro1, Makro2 i tak dalej dla każdego makra, które zarejestrujesz. Zazwyczaj po prostu przyjmuję domyślną nazwę. Jeśli makro działa poprawnie i chcę je zapisać, nadaję mu bardziej adekwatną nazwę, edytując wygenerowany kod w VBE. Ty jednak być może będziesz wolał nadać właściwą nazwę już na samym początku — wybór należy do Ciebie.

Klawisz skrótu

Opcja *Klawisz skrótu* umożliwia wywoływanie makra poprzez naciśnięcie odpowiedniej kombinacji klawiszy. Jeśli na przykład wprowadzisz małą literę *w*, uruchomienie makra będzie możliwe przy użyciu kombinacji *Ctrl+W*. Jeśli wprowadzisz dużą literę *W*, makro będziesz mógł przywołać do życia, naciskając *Ctrl+Shift+W*.

W dowolnym momencie możesz dodać lub zmienić klawisz skrótu, nie ma więc konieczności, by ustawać tę opcję podczas rejestrowaniem makra. Instrukcje dotyczące przypisywania skrótu klawiaturowego do istniejącego makra znajdziesz w rozdziale 5.



Przechowuj makro w

Opcja *Przechowuj makro w* to dla Excela informacja, gdzie przechować makro, które będzie rejestrować. Domyślnie zarejestrowane makro przechowywane jest w module aktywnego skoroszytu. Jeśli wolisz, możesz je zarejestrować w nowym skoroszycie (Excel otworzy czysty skoroszyt) lub też wybrać *Skoroszyt makr osobistych*.

Skoroszyt makr osobistych jest ukrytym skoroszytem, który otwierany jest automatycznie przy uruchomieniu Excela. To idealne miejsce do przechowywania makr, z których będziesz korzystał w wielu skoroszytach. *Skoroszyt makr osobistych* nazywa się *PERSONAL.XLSB*. Plik ten nie istnieje, dopóki nie wskażesz go jako lokalizacji do przechowywania rejestrowanego makra. Jeśli wprowadziłeś jakiekolwiek zmiany w tym pliku, Excel zapyta, czy je zapisać, gdy będziesz zamkał program.

Opis

Jeśli chciałbyś dodać jakiś komentarz opisujący dane makro, użyj pola tekstowego *Opis*. Możesz wpisać tu, co tylko zechcesz lub nie napisać zupełnie niczego. Moim zdaniem jest to strata czasu, bo tak samo łatwo dodaje się komentarze do zarejestrowanego makra w edytorze VBE.

Czy to coś jest wydajne?

Być może myślisz, że rejestrator makr jest w stanie wygenerować zasługujący na wyróżnienia kod VBA — lepszy niż kiedykolwiek byłbyś w stanie napisać ręcznie. Zastanów się jeszcze raz. W wielu przypadkach rejestrator makr wypluwa mnóstwo zbędnych śmieci, często też generuje kod, który nie jest wydajny.

Nie zrozum mnie źle. Jestem wiernym fanem rejestratora makr. Narzędzie to stanowi doskonałą pomoc w opanowaniu języka VBA. Nie licząc jednak prostych makr, nigdy nie użyłem zarejestrowanych makr bez wprowadzenia kliku poprawek (zazwyczaj nawet nie kilku).

Aby się przekonać, jak mało wydajny może być kod wygenerowany rejestratorem makr, wypróbuj to.

1. Uruchom rejestrator makr.
2. Wybierz polecenie **UKŁAD STRONY/Ustawienia strony/Orientacja/Pozioma**.
3. Zatrzymaj rejestrator makr.

Aby zobaczyć makro, aktywuj okno modułu **Module1**. Jedno bardzo proste polecenie wygenerowało poniższy kod.

```
Sub Makrol()
    Application.PrintCommunication = False
    With ActiveSheet.PageSetup
```

```
.PrintTitleRows = ""
.PrintTitleColumns = ""
End With
Application.PrintCommunication = True
ActiveSheet.PageSetup.PrintArea = ""
Application.PrintCommunication = False
With ActiveSheet.PageSetup
    .LeftHeader = ""
    .CenterHeader = ""
    .RightHeader = ""
    .LeftFooter = ""
    .CenterFooter = ""
    .RightFooter = ""
    .LeftMargin = Application.InchesToPoints(0.7)
    .RightMargin = Application.InchesToPoints(0.7)
    .TopMargin = Application.InchesToPoints(0.75)
    .BottomMargin = Application.InchesToPoints(0.75)
    .HeaderMargin = Application.InchesToPoints(0.3)
    .FooterMargin = Application.InchesToPoints(0.3)
    .PrintHeadings = False
    .PrintGridlines = False
    .PrintComments = xlPrintNoComments
    .PrintQuality = 600
    .CenterHorizontally = False
    .CenterVertically = False
    .Orientation = xlLandscape
    .Draft = False
    .PaperSize = xlPaperA4
    .FirstPageNumber = xlAutomatic
    .Order = xlDownThenOver
    .BlackAndWhite = False
    .Zoom = 100
    .PrintErrors = xlPrintErrorsDisplayed
    .OddAndEvenPagesHeaderFooter = False
    .DifferentFirstPageHeaderFooter = False
    .ScaleWithDocHeaderFooter = True
    .AlignMarginsHeaderFooter = True
    .EvenPage.LeftHeader.Text = ""
    .EvenPage.CenterHeader.Text = ""
    .EvenPage.RightHeader.Text = ""
    .EvenPage.LeftFooter.Text = ""
    .EvenPage.CenterFooter.Text = ""
    .EvenPage.RightFooter.Text = ""
    .FirstPage.LeftHeader.Text = ""
    .FirstPage.CenterHeader.Text = ""
    .FirstPage.RightHeader.Text = ""
    .FirstPage.LeftFooter.Text = ""
    .FirstPage.CenterFooter.Text = ""
    .FirstPage.RightFooter.Text = ""
End With
Application.PrintCommunication = True
End Sub
```

Możesz być zdziwiony ilością kodu wygenerowanego tym zaledwie jednym poleceniem. (Ja byłem, gdy pierwszy raz spróbowałem czegoś takiego). Chociaż zmieniłeś tylko jedno ustawienie strony, Excel wygenerował kod, który ustawia wiele innych pokrewnych właściwości.

Powyższy kod jest doskonałym przykładem nadgorliwości rejestratora makr. Jeśli potrzebujesz makra, które po prostu zmieni orientację strony na poziomą, możesz znacznie uprościć powyższe makro, usuwając zbędne instrukcje. Dzięki temu makro będzie nieco szybsze i dużo czytelniejsze. Tak wygląda makro po usunięciu niepotrzebnych wierszy.

```
Sub Makrol()
    With ActiveSheet.PageSetup
        .Orientation = xlLandscape
    End With
End Sub
```

Usunąłem cały kod poza wierszem, w którym ustawiana jest wartość właściwości Orientation. Makro to można jeszcze bardziej uprościć, ponieważ konstrukcja With-End With jest właściwie niepotrzebna (więcej o tej konstrukcji dowiesz się w rozdziale 14.).

```
Sub Makrol()
    ActiveSheet.PageSetup.Orientation = xlLandscape
End Sub
```

W tym przypadku makro zmienia właściwość Orientation obiektu PageSetup w aktywnym arkuszu. Wszystkie pozostałe właściwości pozostają bez zmian. Dla wyjaśnienia dodam, że xlLandscape jest predefiniowaną stałą ułatwiającą czytanie kodu. Stała ta zawiera wartość 2, dlatego też poniższa instrukcja zadziała w dokładnie ten sam sposób (choć nie jest tak czytelna).

```
ActiveSheet.PageSetup.Orientation = 2
```

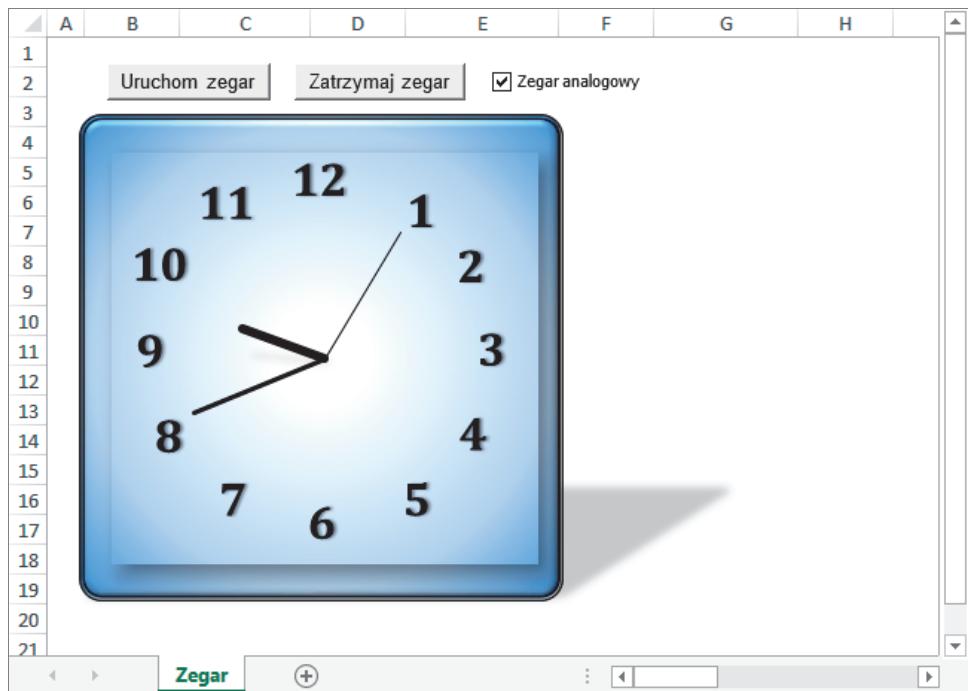
O tym już wkrótce. Predefiniowane stałe omówię w rozdziale 7.

Zamiast rejestrować takie makro, możesz je wprowadzić bezpośrednio do modułu VBA. Aby to zrobić, musisz wiedzieć, jakich obiektów, właściwości i metod należy użyć. Chociaż zarejestrowane makro nie jest zbyt genialne, rejestrując je, uświadamiasz sobie, że obiekt PageSetup zawarty jest w obiekcie Worksheet i jednocześnie posiada właściwość Orientation. Uzbrojony w taką wiedzę, po krótkiej wizycie w systemie pomocy (i prawdopodobnie kilku próbach) będziesz w stanie napisać makro ręcznie.

Rozdział ten właściwie stanowi podsumowanie wszystkiego, co dotyczy posługiwania się rejestratorem makr. Jedyną rzeczą, jakiej jeszcze brakuje, jest doświadczenie. Po pewnym czasie sam odkryjesz, które z zarejestrowanych instrukcji możesz bezpiecznie usunąć. Co więcej, będziesz wiedział, jak zmodyfikować zarejestrowane makro, by było bardziej użytkowe.

Część III

Podstawy programowania



W tej części...

- ✓ poznasz kluczowe elementy języka VBA, między innymi zmienne, stałe, typy danych, operatory i tablice,
- ✓ zaznajomisz się z obiektami Range — warto je zaliczyć do grona znajomych,
- ✓ dowiesz się, dlaczego tak ważne są funkcje VBA (jak również funkcje arkusza kalkulacyjnego),
- ✓ poznasz samą esencję programowania, czyli instrukcje warunkowe i pętle,
- ✓ zobaczysz, jak uzależnić automatyczne wykonywanie instrukcji od wystąpienia określonych zdarzeń,
- ✓ dowiesz się, jakie są rodzaje błędów i dlaczego należy zapewnić obsługę błędów,
- ✓ zobaczysz, co zrobić, gdy dobry kod robi niedobre rzeczy — zaproszenie do klubu eksterminatorów programistycznych robaków,
- ✓ zapoznasz się z wieloma przykładami — częściowo czysto dydaktycznymi, lecz w większości naprawdę z życia wziętymi.

Rozdział 7

Kluczowe elementy języka VBA

W tym rozdziale:

- dowiesz się, kiedy, dlaczego i jak używać komentarzy w kodzie,
- nauczysz się korzystać ze zmiennych i stałych,
- poznasz sposób określania typu zmiennych VBA, których używasz,
- dowiesz się, do czego czasami mogą być potrzebne etykiety w procedurach.

Jako że VBA jest prawdziwym, żywym językiem programowania, posiada wiele elementów charakterystycznych dla wszystkich języków programowania. W tym rozdziale poznasz niektóre takie elementy, czyli komentarze, zmienne, stałe, typy danych, tablice i inne dobrodziejstwa VBA. Jeśli programowałeś już w innych językach, niektóre części materiału mogą wyglądać znajomo. Jeśli w programowaniu jesteś żółtodziobem, czas zakastać rękawy i wziąć się do roboty.

Stosowanie komentarzy w kodzie VBA

Komentarz jest najprostszym rodzajem instrukcji VBA. Instrukcje mogą zawierać cokolwiek chcesz, ponieważ VBA je ignoruje. Komentarze możesz wstawiać, aby przypomnieć sobie później, dlaczego coś zrobiłeś, lub objąśnić jakieś szczegółowo sprytne instrukcje, które napisałesz.



Stosuj często komentarze z dokładnymi opisami tego, co robi dany kod. Nie zawsze jest to oczywiste po przeczytaniu samych instrukcji. Często to, co dziś jest doskonale zrozumiałe, jutro może przysporzyć Ci bólu głowy. Już ja coś o tym wiem.

Komentarze rozpoczynają się od znaku apostrofu ('). VBA ignoriuje wszelki tekst, jaki się znajdzie za tym znakiem w tym samym wierszu kodu. Komentarz może zajmować cały wiersz, ale równie dobrze może być umieszczony pod koniec wiersza kodu. W poniższym przykładzie demonstruję procedurę VBA z czterema komentarzami.

```
Sub FormatCells()
    ' Jeśli nie jest zaznaczony zakres – zakończ działanie
    If TypeName(Selection) <> "Range" Then
        MsgBox "Zaznacz zakres."
        Exit Sub
    End If
    ' Formatowanie komórek
    With Selection
        .HorizontalAlignment = xlRight
```

```
.WrapText = False ' bez zawijania tekstu  
.MergeCells = False ' bez łączenia komórek  
End With  
End Sub
```



Apostrof oznacza, że jest to komentarz, niemniej jednak istnieje jeden wyjątek od tej reguły: apostrof znajdujący się wewnętrz frazy ujętej w znaki cudzysłowu nie jest interpretowany jako oznaczenie komentarza. Tak więc w poniższej instrukcji, mimo że zawiera znak apostrofu, nie ma komentarza.

```
Msg = "Can't continue!"
```

W trakcie pisania kodu może się zdarzyć, że będziesz chciał przetestować procedurę, wyłączając chwilowo pewną instrukcję lub grupę instrukcji. Mógłbyś — oczywiście — usunąć te instrukcje i później wprowadzić je ponownie, ale to strata czasu. Lepszym rozwiązaniem jest dodanie apostrofów tak, by zamienić te instrukcje w komentarze. Wykonując procedurę, VBAignoruje wszelkie instrukcje zaczynające się od apostrofu. Aby reaktywować „zakomentowane” instrukcje, wystarczy usunąć apostrofy.



Również cały blok instrukcji można w szybki sposób przekształcić w komentarz. W edytorze VBE wybierz polecenie *View/Toolbars>Edit*, aby wyświetlić pasik narzędzi *Edit*. Zeby przekształcić blok instrukcji w komentarze, zaznacz te instrukcje i kliknij przycisk *Comment Block*. Aby usunąć apostrofy, zaznacz instrukcje i kliknij przycisk *Uncomment Block*.



Po pewnym czasie każdy wyksztalca swój własny styl komentowania. By jednak komentarze były użyteczne, powinny zawierać informacje, które nie są oczywiste już po przeczytaniu samych instrukcji.

Poniższe wskazówki mogą Ci podpowiedzieć, jak efektywnie posługiwać się komentarzami.

- ✓ Identyfikuj się jako autor. Może się to przydać, gdy dostaniesz awans i ktoś, kto przejmie Twój stanowisko, będzie mieć pytania.
- ✓ Opisz zwięźle cel każdej procedury *Sub* lub funkcji, jaką napiszesz.
- ✓ Używaj komentarzy, by śledzić zmiany, jakie wprowadzasz do procedury.
- ✓ Stosuj komentarze do oznaczania niestandardowych lub rzadkich zastosowań funkcji i konstrukcji.
- ✓ Opisuj w komentarzach przeznaczenie zastosowanych zmiennych szczególnie wtedy, kiedy ich nazwy są mało znaczące.
- ✓ Oznaczaj komentarzem wszelkiego rodzaju obejścia problemów w Excelu, jakie wypracujesz.
- ✓ Komentarze pisz równolegle z tworzonym kodem, nie zostawiaj tego zadania na koniec.
- ✓ W zależności od atmosfery panującej w pracy, pomyśl nad dodaniem czegoś śmiesznego do komentarza. Osoba, która przejmie Twój pracę, gdy już awansujesz, na pewno doceni poczucie humoru.

Używanie zmiennych, stałych i typów danych

Głównym zadaniem języka VBA jest przetwarzanie danych. VBA przechowuje dane w pamięci komputera, a ostatecznie mogą one (ale nie muszą) zostać zapisane na dysku twardym. Niektóre dane, takie jak zakresy komórek, przechowywane są w obiektach, podczas gdy nośnikami innych danych mogą być zmienne, które tworzysz.

Pojęcie zmiennej

Zmienna jest po prostu posiadającą nazwę lokalizacją w pamięci komputera, zajętą przez program do przechowywania danych. Nie ma wielu ograniczeń, jeśli chodzi o nazwy zmiennych, stosuj więc nazwy wnoszące jak najwięcej informacji o ich przeznaczeniu. Wartości przypisuje się do zmiennych za pomocą operatora znaku równości. (Więcej na ten temat znajdziesz w podrozdziale „Instrukcje przypisania”).

Nazwy zmiennych w poniższych przykładach występują zarówno po prawej, jak i po lewej stronie znaków równości. Zauważ, że w ostatnim przykładzie użyte zostały dwie zmienne.

```
x = 1
InterestRate = 0.075
LoanPayoffAmount = 243089
DataEntered = False
x = x + 1
UserName = "Jan Nowak"
Date_Started = #3/14/2013#
MyNum = YourNum * 1.25
```

Nazwy zmiennych w języku VBA muszą spełniać kilka wymogów.

- ✓ Można używać liter, cyfr i niektórych znaków przestankowych, przy czym pierwszym znakiem musi być litera.
- ✓ VBA nie rozróżnia wielkości liter.
- ✓ W nazwach zmiennych niedozwolone są spacje, kropki i operatory matematyczne.
- ✓ Nazwy zmiennych nie mogą zawierać następujących znaków: #, \$, %, &, !.
- ✓ Nazwy zmiennych nie mogą być dłuższe niż 255 znaków. Trudno jednak nawet zbliżyć się do osiągnięcia takiego limitu.

Aby nazwy zmiennych były czytelniejsze, programiści zazwyczaj stosują kombinacje małych i wielkich liter (na przykład `InterestRate`) lub znaki podkreślenia (`interest_rate`).

W VBA zastrzeżono wiele słów kluczowych, których nie można używać jako nazw zmiennych lub procedur. Do słów tych należą między innymi `Sub`, `Dim`, `With`, `End`, `Next` i `For`. Próba użycia któregoś z nich jako zmiennej może zakończyć się błędem komplikacji

(w efekcie procedury nie da się uruchomić). Jeśli więc instrukcja przypisania wywoła błąd, sprawdź ją dokładnie i upewnij się, że nazwa zmiennej nie jest słowem kluczowym.

VBA pozwala tworzyć zmienne o nazwach pokrywających się z nazwami występującymi w modelu obiektowym Excela, takimi jak Workbook czy też Range. Takie praktyki mogą się jednak przyczyniać do częstszych pomyłek. Oto przykład w pełni poprawnego, lecz kompletnie obdartego z intuicyjności makra, w którym zadeklarowano Range jako nazwę zmiennej oraz wykonano operacje na komórce o nazwie Range i arkuszu noszącym również nazwę Range.

```
Sub RangeConfusion()
    Dim Range As Double
    Range = Sheets("Range").Range("Range")
    MsgBox Range
End Sub
```

Spróbuj więc oprzeć się pokusie deklarowania zmiennych o nazwach Workbook lub Range. Zamiast tego możesz użyć czegoś w stylu MyWorkbook lub MyRange.

Czym są typy danych w języku VBA?

Gdy mówię o **typach danych**, odnoszę się do sposobu, w jaki program przechowuje dane w pamięci komputera. Mogą być one przykładowo przechowywane jako liczby całkowite, liczby rzeczywiste czy też łańcuchy znaków. Chociaż VBA potrafi uporać się z takimi szczegółami automatycznie, nie odbywa się to bez ponoszenia kosztów. (Nie zasilisz skarbonki). Pozwalając VBA dynamicznie przydzielać typy danych, godzisz się na wolniejsze wykonywanie programu i mało wydajne wykorzystanie zasobów pamięci. W przypadku niewielkich aplikacji nie stanowi to z reguły większego problemu. Projektując jednak duże lub złożone aplikacje, które mogą działać wolno lub wymagać optymalnego wykorzystania każdego wolnego bajta pamięci, musisz się dobrze orientować w typach danych.

VBA automatycznie określa typy danych, co ułatwia życie programistom. Nie wszystkie języki programowania zapewniają taki luksus. Niektóre języki programowania są *ściśle typowane*, co wymusza na programiście jawne deklarowanie typu danych każdej zmiennej, jakiej użyje.

VBA nie narzuca konieczności deklarowania używanych zmiennych, niemniej jednak jest to zdecydowanie dobra praktyka. Dalej w tym rozdziale dowiesz się, dlaczego tak jest.

VBA posiada szeroki wachlarz predefiniowanych typów danych. Tabela 7.1 stanowi zestawienie najczęściej używanych typów danych obsługiwanych w VBA.

Generalnie rzecz ujmując, najlepiej stosować takie typy danych, które zajmują jak najmniejszą liczbę bajtów, a jednocześnie mieszczą się w nich wszystkie dane, które mają być przechowywane w danych zmiennych.

Wyjątkiem od „reguły najmniejszej liczby bajtów” jest typ Long. Większość programistów używa typu Long w miejsce typu Integer, gdyż pozwala to nieco poprawić wydajność programu. W niewielkich aplikacjach różnice pomiędzy Long a Integer są jednak kompletnie niezauważalne.



Tabela 7.1. Predefiniowane typy danych języka VBA

Typ danych	Rozmiar w bajtach	Zakres wartości
Byte	1	Od 0 do 255
Boolean	2	True lub False
Integer	2	Od -32 768 do 32 767
Long	4	Od -2 147 483 648 do 2 147 483 647
Single	4	Od -3,40E38 do -1,40E-45 dla wartości ujemnych; od 1,40E-45 do 3,40E38 dla wartości dodatnich
Double	8	Od -1,79E308 do -4,94E-324 dla wartości ujemnych; od 4,94E-324 do 1,79E308 dla wartości dodatnich
Currency	8	Od -922 337 203 685 477,5808 do 922 337 203 685 477,5807
Date	8	Od 1 stycznia 100 roku do 31 grudnia 9999 roku
Object	4	Dowolne odwołanie do obiektu
String	1 na każdy znak	Zmienny
Variant	zmienny	Zmienny

Deklarowanie zmiennych i określanie ich zasięgu

Jeśli przeczytałeś poprzednie części rozdziału, dysponujesz już pewną wiedzą na temat zmiennych i ich typów. W tym punkcie dowiesz się, jak zadeklarować zmienną o określonym typie danych.

Jeśli nie zadeklarujesz typu zmiennej używanej w instrukcjach VBA, przyjęty zostanie domyślny typ danych, jakim jest Variant. Dane przechowywane jako Variant są jak kameleon — zmieniają typ w zależności od tego, co z nimi zrobisz. Jeśli zmienna typu Variant zawiera tekst, który wygląda jak liczba (na przykład "143"), można jej używać zarówno w operacjach na ciągach znaków, jak również w obliczeniach matematycznych. VBA automatycznie konwertuje dane z jednego typu na drugi. Pozostawienie VBA odpowiedzialności za obsługę typów danych może się wydawać dużym ułatwieniem — pamiętaj jednak, że odbywa się to kosztem prędkości i pamięci.

Godną polecenia praktyką jest **deklarowanie zmiennych**, zanim zostaną użyte w procedurach. Deklarowanie oznacza wskazanie typu danych dla poszczególnych zmiennych. Deklarowanie zmiennych przyspiesza działanie makra oraz zapewnia bardziej wydajne wykorzystanie pamięci. Typ Variant (domyślny typ danych) wymaga od VBA ciągłych operacji porównywania typów danych i rezerwowania większych zasobów pamięci, niż to konieczne. Kiedy określmy typ zmiennej, VBA może zaoszczędzić sobie zabawy z prowadzeniem dochodzeń. Jednocześnie program przyjmuje, że nie strzela się z armaty do muchy, zatem nie będzie rezerwować więcej miejsca w pamięci, niż wymaga tego typ zmiennej.

Aby przymusić siebie samego do deklarowania używanych zmiennych, umieść poniższe dwa słowa jako pierwszą instrukcję w module VBA:

Option Explicit

Gdy moduł rozpoczyna się od takiej instrukcji, nie powiodą się wszelkie próby uruchomienia procedury zawierającej niezadeklarowane zmienne.



Instrukcji Option Explicit używa się tylko raz — na początku modułu, przed deklaracjami jakichkolwiek procedur. Nie zapominaj, że Option Explicit obowiązuje tylko w obrębie modułu, w którym się znajduje. Jeśli w projekcie istnieje więcej niż jeden moduł, instrukcję Option Explicit należy umieścić w każdym z nich.

Załóżmy, że używasz niezadeklarowanej zmiennej (czyli zmiennej typu Variant) o nazwie CurrentRate. W pewnym miejscu zamieszczasz w swoim programie instrukcję:

CurrentRate = .075

Nazwa zmiennej zawiera literówkę (brakuje jednej litery *r*), co może być trudne do wyśledzenia. Jeśli tego nie zauważysz, Excel zinterpretuje to jako użycie innej, nowej zmiennej, rezultatem czego będą najprawdopodobniej błędne wyniki działania programu. Jeśli na początku modułu umieścisz Option Explicit, wymuszając jednocześnie zadeklarowanie zmiennej CurrentRate, Excel wygeneruje błąd, jeśli napotka błędnie zapisaną wersję takiej zmiennej.



Aby instrukcja Option Explicit była wstawiana automatycznie na początku każdego nowo dodanego modułu VBA, wystarczy włączyć opcję *Require Variable Definition* (wymuszaj deklarację zmiennych). Znajdziesz ją na karcie *Editor* (edytor) okna dialogowego *Options* (opcje — użyj polecenia *Tools/Options* w VBA). Szczerze radzę, byś tak zrobił.

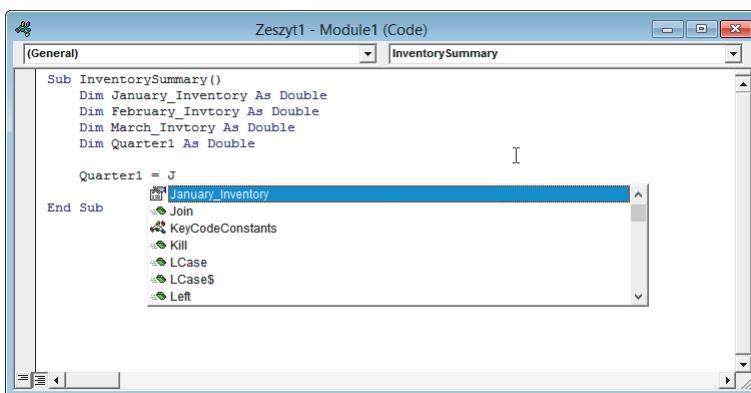


Jeśli zadeklarujesz zmienne, będziesz mógł również skorzystać z ułatwienia, dzięki któremu zaoszczędzisz sobie nieco pisania. Wprowadź po prostu pierwsze dwie lub trzy litery nazwy zmiennej, a następnie naciśnij *Ctrl+Spaja*. Jeśli wybór jest jednoznaczny, VBE uzupełni wpis za Ciebie. Jeśli nie, wyświetlona zostanie lista pasujących słów, z której będziesz mógł wybrać odpowiednie. Dodam jeszcze, że funkcjonalność ta działa również w przypadku słów kluczowych i nazw funkcji. Na rysunku 7.1 przedstawiam przykład takiej automatycznej listy.

Znasz już zalety deklarowania zmiennych, ale nadal nie wiesz, *jak* to się robi. Najczęściej stosowanym sposobem jest użycie instrukcji ze słowem kluczowym *Dim*. Oto kilka przykładów deklarowania zmiennych.

```
Dim YourName As String
Dim January Inventory As Double
Dim AmountDue As Double
Dim RowNumber As Long
Dim X
```

Pierwsze cztery zmienne zostały zadeklarowane jako określony typ danych. Ostatnia zmienna, *X*, nie została zadeklarowana jako określony typ danych, jest więc traktowana jako Variant (może być dowolnego typu).



Rysunek 7.1.
Naciśnięcie klawiszy Ctrl + spacja powoduje wyświetlenie listy nazw zmiennych, funkcji i słów kluczowych

Poza Dim w VBA występują jeszcze trzy inne słowa kluczowe używane przy deklarowaniu zmiennych:

- ✓ Static,
- ✓ Public,
- ✓ Private.

Więcej na temat słów kluczowych Dim, Static, Public i Private napiszę później, ale najpierw muszę omówić dwa inne tematy, które są tu ważne, czyli zasięg zmiennych i czas życia zmiennych.

Przypominasz sobie, że skoroszyt może zawierać dowolną liczbę modułów VBA? Podobnie moduł VBA może zawierać dowolną liczbę procedur Sub i Function. **Zasięg zmiennej** określa, w których modułach i procedurach zmienna ta może być używana. Szczegółowe informacje znajdziesz w tabeli 7.2.

Tabela 7.2. Rodzaje zasięgu zmiennych

Zasięg	Sposób deklarowania zmiennej
Tylko dana procedura	Przy użyciu instrukcji Dim lub Static w obrębie procedury, w której używana jest zmienna.
Tylko dany moduł	Przy użyciu instrukcji Dim lub Private przed pierwszą instrukcją Sub lub Function w module.
Wszystkie procedury we wszystkich modułach	Przy użyciu instrukcji Public przed pierwszą instrukcją Sub lub Function w module.

Pogubiłeś się? Czytaj dalej, a znajdziesz kilka przykładów, dzięki którym wszystko stanie się jasne.

Zmienne o zasięgu jednej procedury

Najniższym poziomem zasięgu zmiennej jest poziom procedury. (Procedurami są procedury Sub i Function). Zmienne o tak zadeklarowanym zasięgu, inaczej **zmienne lokalne**, mogą być używane tylko w obrębie procedury, w której zostały zadeklarowane. Gdy kończy się procedura, zmienność przestaje istnieć (zostanie unicestwiona), a Excel zwalnia zajmowaną przez nią pamięć. Gdy teraz ponownie uruchamiasz procedurę, zmienność powraca do życia, ale jej poprzednia wartość została utracona.

Standardowym sposobem deklarowania zmienności lokalnych jest użycie instrukcji `Dim`. Słowo `Dim` (*ciemny*) nie odnosi się do zdolności intelektualnych twórców VBA, a raczej jest to stary termin programistyczny będący skrótem od *dimension* (*rozmiar*) i oznacza po prostu rezerwowanie miejsca w pamięci dla danej zmiennej.

Poniższy przykład zawiera kilka zmienności o zasięgu jednej procedury, zadeklarowanych przy użyciu słowa kluczowego `Dim`.

```
Sub MySub()
    Dim x As Integer
    Dim First As Long
    Dim InterestRate As Single
    Dim TodaysDate As Date
    Dim UserName As String
    Dim MyValue
    ' ... [Tu znajduje się kod procedury] ...
End Sub
```

Zauważ, że w powyższym przykładzie ostatnia deklaracja `Dim` nie określa typu zmiennej `MyValue` — deklarowana jest jedynie sama zmienność. W rezultacie typem danych zmiennej `MyValue` jest Variant.

Nawiasem mówiąc, istnieje możliwość deklarowania kilku zmienności jednocześnie za pomocą pojedynczej instrukcji `Dim`. Oto przykład.

```
Dim x As Integer, y As Integer, z As Integer
Dim First As Long, Last As Double
```

W przeciwieństwie do niektórych języków, VBA nie umożliwia jednoczesnego deklarowania kilku zmienności jako określonego typu danych poprzez oddzielenie tych zmienności przecinkami. Choć przykładowo poniższa instrukcja jest całkowicie poprawna, nie deklaruje wszystkich wymienionych zmienności jako `Integer`.

```
Dim i, j, k As Integer
```

W przykładzie tym jedynie zmienność `k` jest deklarowana jako `Integer`, podczas gdy pozostały zmienności przydzielony zostaje domyślny typ danych (`Variant`).

Gdy zadeklarujesz zmienność o zasięgu jednej procedury, w pozostałych procedurach tego samego modułu możesz używać zmienności o identycznej nazwie, lecz każda z tych instancji zmiennej będzie jedyną w obrębie procedury, w której została zadeklarowana. Ogólnie rzecz ujmując, zmienność deklarowana na poziomie procedury są najbardziej wydajne, ponieważ VBA zwalnia zajmowaną przez nie pamięć w momencie, gdy kończy się procedura.

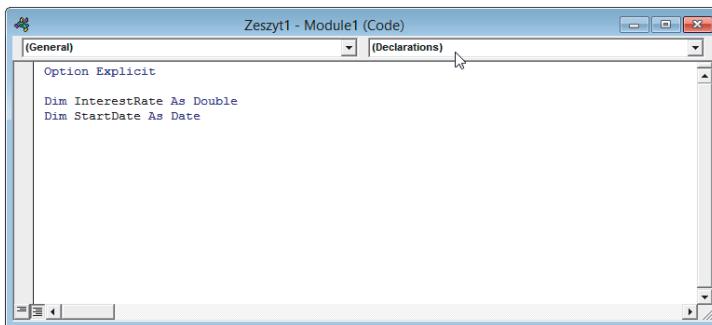


Zmienne o zasięgu jednego modułu

Czasami możesz potrzebować zmiennej, która byłaby dostępna z poziomu wszystkich procedur należących do modułu. W takim przypadku wystarczy zadeklarować zmienną (za pomocą Dim lub Private) przed pierwszym wystąpieniem instrukcji Sub lub Function — poza wszelkimi procedurami. Robi się to w sekcji *Declarations* (deklaracje), znajdującej się na początku modułu. (W sekcji tej umieszcza się również instrukcję Option Explicit).

Na rysunku 7.2 pokazuję, jak możesz się upewnić, że znajdujesz się w sekcji *Declarations*. Użyj listy rozwijanej znajdującej się po prawej stronie i przejdź bezpośrednio do sekcji *Declarations* (to nie *Eurobusiness*, nie przechodzisz przez *Start* i nie zgarniasz 400 dolarów).

Rysunek 7.2.
Każdy moduł
VBA zawiera
sekcję Declara-
tions, która
znajduje się
przed instruk-
cjami wszelkich
procedur Sub
i Function



Dla przykładu założmy, że chcesz tak zadeklarować zmienną *CurrentValue*, aby była dostępna we wszystkich procedurach danego modułu. W tym przypadku wystarczy umieścić instrukcję Dim w sekcji *Declarations*:

```
Dim CurrentValue As Double
```

Gdy zamieściszt taką deklarację (robiąc to jednocześnie we właściwym miejscu), będziesz mógł używać zmiennej *CurrentValue* w każdej procedurze należącej do modułu, a wartość tej zmiennej będzie zachowywana również pomiędzy poszczególnymi procedurami.

Zmienne o zasięgu globalnym

Jeśli chcesz zapewnić dostęp do zmiennej z poziomu wszystkich procedur należących do wszystkich modułów danego skoroszytu, zadeklaruj ją na poziomie modułu (w sekcji *Declarations*), używając słowa kluczowego Public. Oto przykład takiej deklaracji:

```
Public CurrentRate As Long
```

Po użyciu słowa kluczowego *Public* zmienna *CurrentRate* jest dostępna we wszystkich modułach skoroszytu — nie tylko w tym, w którym została zadeklarowana. Instrukcja ta musi się znaleźć przed pierwszą instrukcją procedury Sub lub Function w module.

Jeśli chcesz udostępnić zmienną również w modułach innych skoroszytów, musisz ją zadeklarować jako *Public* i ustawić odwołanie do skoroszytu, w którym zawarta została deklaracja tej zmiennej. Odwołanie możesz ustawić przy użyciu polecenia *Tools/References* (narzędzia/odwołania) w edytorze VBE. W praktyce, zmienne współdzielone pomiędzy

skoroszytami wykorzystywane są niezmiernie rzadko. Odkąd programuję w VBA, ani razu nie użyłem takiej zmiennej. Tak czy inaczej myślę, że warto wiedzieć o takiej możliwości. Kto wie, czy takie pytanie nie padnie w *Milionerach*?

Zmienne statyczne

Standardowo, kiedy zakończy się wykonywanie procedury, wartości wszystkich zawartych w niej zmiennych zostają zresetowane. Wyjątek stanowią **zmienne statyczne**, które zachowują swoje wartości również po zakończeniu wykonywania procedury. Zmienne statyczne deklaruje się w obrębie procedury. Zmienna taka może być użyteczna, jeśli chcesz na bieżąco znać liczbę wykonania danej procedury. Możesz wówczas zadeklarować zmienną statyczną i zwiększać jej wartość przy każdym wywołaniu procedury.

Jak pokazano w poniższym listingu, zmienne statyczne deklaruje się za pomocą słowa kluczowego Static.

```
Sub MySub()
    Static Counter As Integer
    Dim Msg As String
    Counter = Counter + 1
    Msg = "Liczba wywołań procedury: " & Counter
    MsgBox Msg
End Sub
```

Powyższy program na bieżąco śledzi liczbę wywołań procedury i wyświetla tę liczbę w oknie komunikatu. Wartość zmiennej Counter nie jest resetowana po dotarciu do końca procedury, lecz dopiero po zamknięciu i ponownym otwarciu skoroszytu.

Pomimo że wartość zmiennej zadeklarowanej jako Static jest przechowywana również po zakończeniu procedury, zmienna taka nie jest dostępna z poziomu innych procedur. W poprzednim przykładzie zmienna Counter i przechowywana przez nią wartość są dostępne tylko w obrębie procedury MySub. Innymi słowy, Counter jest zmienną lokalną.



Czas życia zmiennych

Nic nie jest wieczne, dotyczy to również zmiennych. Zasięg zmiennej nie tylko określa, gdzie można jej używać, lecz również determinuje okoliczności, w których zmienna usuwana jest z pamięci komputera.

Pamięć komputera można oczyścić ze wszystkich zmiennych na trzy sposoby.

- ✓ Klikając przycisk Reset (mały, kwadratowy, niebieski przycisk znajdujący się na zakładce Standard edytora VBE).
- ✓ Klikając End, gdy pojawi się okno dialogowe z komunikatem błędu czasu wykonania (*runtime error*).
- ✓ Umieszczając instrukcję End w dowolnym miejscu procedury. Instrukcja ta nie jest tożsama z instrukcjami End Sub i End Function.

Inaczej jedynie zmienne o zasięgu procedury (zmienne lokalne) zostaną usunięte z pamięci w momencie, gdy instrukcje danego makra dobiegną końca. Zmienne statyczne,

zmienne o zasięgu jednego modułu oraz zmienne globalne (publiczne) będą zachowywać swoje wartości pomiędzy kolejnymi wywołaniami procedury.



Gdy używasz zmiennych o zasięgu modułu lub globalnym, upewnij się, że w danym momencie przechowują takie wartości, jakich oczekujesz. Nigdy nie możesz mieć pewności, czy w wyniku zaistnienia jednej z wyżej opisanych sytuacji zmienne te nie utraciły przechowywanych wartości.

Stałe

Wartość zmiennej może się zmieniać w trakcie działania programu. Z reguły tak się dzieje, dlatego też nazywane są *zmiennymi*. Czasami jednak trzeba odwołać się do wartości liczbowej lub tekstowej, która nigdy się nie zmienia. W takim przypadku najlepiej sprawdzi się **stała** — element posiadający nazwę, którego wartość się nie zmienia.

Jak pokazano w następującym przykładzie, stałe deklaruje się za pomocą słowa kluczowego Const. Deklaracja stałej nadaje jej jednocześnie określona wartość.

```
Const NumQuarters As Integer = 4
Const Rate = .0725, Period = 12
Const ModName As String = "Makra Budżetu"
Public Const AppName As String = "Aplikacja Budżet"
```



Używanie stałych w miejsce wpisywanych na sztywno wartości liczbowych i tekstowych stanowi doskonałą praktykę programistyczną. Jeśli w procedurze kilkakrotnie odnosisz się do jakiejś określonej wartości (takiej jak na przykład stopa procentowa), lepiej będzie zadeklarować tę wartość jako stałą i posługiwać się nazwą stałej, niż każdorazowo wpisywać określoną liczbę. Dzięki temu kod będzie czytelniejszy i łatwiej będzie go zmodyfikować. Gdy zmieni się stopa procentowa, będziesz musiał zmienić nie wszystkie, a tylko jedną instrukcję.



Stałe mogą się różnić zasięgiem, podobnie jak zmienne. Miej na uwadze poniższe wskazówki.

- ✓ Jeśli stała ma być dostępna jedynie z poziomu danej procedury, zadeklaruj ją za instrukcją Sub lub Function tej procedury.
- ✓ Jeśli stała ma być dostępna z poziomu wszystkich procedur danego modułu, zadeklaruj ją w sekcji *Declarations* tego modułu.
- ✓ Jeśli stała ma być dostępna z poziomu wszystkich modułów danego skoroszytu, zadeklaruj ją w sekcji *Declarations* dowolnego modułu, używając słowa kluczowego Public.

W przeciwnieństwie do zmiennej, wartość przypisana stałej nie ulega zmianie. Próba użycia instrukcji VBA do zmiany wartości stałej zakończy się zgłoszeniem błędu. Nie jest to zbyt zaskakujące, gdy uwzględnimy fakt, że wartość stałej musi pozostać stała. Jeśli w trakcie wykonywania programu trzeba zmienić wartość stałej, oznacza to, że właściwie powinieneś był użyć zmiennej.

Stałe predefiniowane

Excel i VBA zawierają wiele predefiniowanych stałych, których można używać bez konieczności ich deklarowania. Rejestrator makr zazwyczaj operuje nie na konkretnych wartościach, lecz właśnie na takich stałych. Na ogół nie trzeba też znać wartości tych stałych, aby z nich korzystać. W następującym prostym przykładzie wykorzystano predefiniowaną stałą `xlCalculationManual`, aby zmienić wartość właściwości `Calculation` obiektu `Application`. Innymi słowy, tryb obliczania skoroszytu został zmieniony na ręczny.

```
Sub CalcManual()
    Application.Calculation = xlCalculationManual
End Sub
```

Stałą `xlCalculationManual` odkryłem, gdy rejestrowałem makro podczas zmiany trybu obliczania skoroszytu¹. Zajrzałem również do systemu pomocy, gdzie znalazłem poniższe informacje.

Nazwa stałej	Wartość	Opis
<code>xlCalculationAutomatic</code>	-4105	Excel automatycznie ponownie oblicza skoroszyt.
<code>xlCalculationManual</code>	-4135	Ponowne obliczenie skoroszytu następuje na polecenie użytkownika.
<code>xlCalculationSemiautomatic</code>	2	Excel automatycznie ponownie oblicza skoroszyt, ignorując przy tym zmiany w tabelach.

Rzeczywistą wartością stałej `xlCalculationManual` jest więc -4135. Oczywiście, dużo łatwiej posługiwać się nazwą zmiennej, niż spróbować zapamiętać tak dziwną wartość. Jak pokazano w tym przykładzie, wiele predefiniowanych stałych to po prostu arbitralne liczby o specjalnym znaczeniu w VBA.

Aby poznać rzeczywistą wartość predefiniowanej zmiennej, użądź okna *Immediate* w edytorze VBE i wykonaj instrukcję analogiczną do tej:

```
? xlCalculationAutomatic
```

Jeśli okno *Immediate* nie jest widoczne, skorzystaj z kombinacji klawiszy *Ctrl+G*. Znak zapytania jest odpowiednikiem polecenia *Print*.

Łańcuchy znaków

Excel może wykonywać operacje nie tylko na liczbach, lecz również na danych tekstowych. Nie zdziwi Cię więc fakt, że VBA dysponuje podobnymi umiejętnościami. Fragmenty tekstu określane są często jako **łańcuchy znaków** (*string*). W VBA rozróżnia się dwa rodzaje łańcuchów.

¹ Tryb obliczania skoroszytu można zmienić również w Excelu, wybierając *Plik/Opje/Formuły*, w grupie poleceń *Opcje obliczania — przyp. thm.*



- ✓ **Łańcuchy znaków o stałej długości** deklaruje się z podaniem określonej liczby znaków. Maksymalna długość to 65 526 znaków. To całkiem spora liczba! Dla porównania, liczba znaków w tym rozdziale to około połowa tego limitu.
- ✓ **Łańcuchy znaków o zmiennej długości** teoretycznie mogą przechowywać aż dwa miliardy znaków. Jeśli potrafisz pisać z prędkością pięciu znaków na sekundę, wystukanie dwóch miliardów znaków zajęłoby Ci prawie 4700 dni (przy założeniu, że zrezygnowałbyś z przerw na sen i posiłki).

Deklarując zmienną łańcuchową za pomocą słowa kluczowego `Dim`, możesz określić maksymalną długość łańcucha, o ile ją znasz (jest to wówczas łańcuch o stałej długości), lub też pozwolić, by długość była określana dynamicznie przez VBA (łańcuch o zmiennej długości). W poniższym przykładzie zadeklarowano zmienną `MyString` jako łańcuch o maksymalnej długości wynoszącej 50 znaków. (Aby określić liczbę znaków, maksymalnie 65 526, używa się znaku `*` (*asterisk*), czyli naszej popularnej „gwiazdki”). Zmienna `YourString` również została zadeklarowana jako łańcuch znaków, ale jego długość pozostała nieokreślona.

```
Dim MyString As String * 50
Dim YourString As String
```



Deklarując zmienne łańcuchowe o liczbie znaków przekraczającej 999, w zapisie tej liczby nie używaj spacji jako separatora tysięcy. Nawiąsem mówiąc, kiedy będziesz wprowadzał wartości liczbowe w VBA, nie używaj żadnego separatora tysięcy. Pamiętaj również, by w roli separatora dziesiętnego używać kropki zamiast przecinka, charakterystycznego dla polskiego formatowania wartości liczbowych.

Daty i godziny

Kolejnym użytecznym typem danych jest `Date`. Do przechowywania dat można wprawdzie stosować zmienne łańcuchowe, ale niemożliwe jest wówczas wykonywanie na nich operacji. Używając dedykowanego datom typu `Date`, zapewniasz sobie więcej swobody w programowaniu. Założymy przykładowo, że chcesz obliczyć liczbę dni, jakie dzielą dwie daty. Byłoby to karkołomne (jeśli nie niemożliwe) wyzwanie, jeśli daty te zachowałbyś jako zmienne łańcuchowe.

Zmienna o typie zadeklarowanym jako `Date` może przechowywać daty z zakresu od 1 stycznia 100 roku do 31 grudnia 9999 roku. Jest to przedział niemalże 10 000 lat i z pewnością wystarczy do opracowania nawet najbardziej śmiały prognozy finansowej. Typ `Date` może być również wykorzystywany do wykonywania operacji na czasie zegarowym (VBA nie posiada typu danych dedykowanego określeniom czasu zegarowego).

W poniższym przykładzie zadeklarowane zostały dwie zmienne i dwie stałe typu `Date`.

```
Dim Today As Date
Dim StartTime As Date
Const FirstDay As Date = #1/1/2013#
Const Noon = #12:00:00#
```

Jak widać, w języku VBA określenia daty i godziny umieszczane są pomiędzy dwoma krzyżykami (*hash*).



Sposób wyświetlania zmiennych typu Date zależy od formatów, jakie ustawione zostały w systemie operacyjnym. Daty wyświetlane są zgodnie z wybranym formatem daty krótkiej, godziny natomiast według formatu godziny długiej (format 24-godzinny w wersji polskiej Excela). Ustawienia te przechowywane są w rejestrze systemu Windows i można je modyfikować na karcie *Formaty* okna dialogowego *Region i język* (polecenie *Panel sterowania/Zegar, język i region/Region i język/Zmień formaty daty, godziny lub liczb* w widoku kategorii w Windows 8). Format dat i godzin wyświetlanych z poziomu VBA może się różnić, w zależności od ustawień systemu, w którym uruchomiono aplikację.

Pisząc instrukcje VBA, musisz posługiwać się jednym z amerykańskich formatów dat (na przykład *mm/dd/rrrr*). W poniższej instrukcji zmiennej *MyDate* przypisana zostaje zatem data *11 października 2013* (a nie *10 listopada 2013*), nawet jeśli w ustawieniach systemu wybrano format *dd/mm/rrrr*.

```
MyDate = #10/11/2013#
```

Gdy wyświetlisz powyższą zmienną (na przykład przy użyciu funkcji *MsgBox*), VBA pokaże datę w formacie zgodnym z ustawieniami systemu. Jeśli Twój system operacyjny używa formatu *rrrr-mm-dd* (domyślnego dla polskiej wersji językowej), zmienna *MyDate* będzie wyświetlana jako *2013-10-11*.

Instrukcje przypisania

Instrukcja przypisania to instrukcja języka VBA, która przypisuje wynik pewnego wyrażenia do zmiennej lub obiektu. System pomocy Excela definiuje termin *wyrażenie* jako:

...kombinację słów kluczowych, operatorów, zmiennych i stałych, której wynikiem jest łańcuch znaków, liczba lub obiekt. Wyrażenie może być użyte do wykonania obliczeń, operacji na znakach lub weryfikacji danych.

Sam lepiej bym tego nie ujął, nie będę więc nawet próbował.

Duża część pracy w języku VBA polega na tworzeniu (i debugowaniu) wyrażeń. Jeśli wiesz, jak opracowywać formuły w Excelu, tworzenie wyrażeń również nie przysporzy Ci trudności. Wynik formuły arkusza kalkulacyjnego wyświetlany jest w komórce, natomiast wynik wyrażenia VBA może zostać przypisany do zmiennej.

Przykłady instrukcji przypisania

W poniższych przykładach instrukcji przypisania wyrażenia znajdują się po prawej stronie znaku równości.

```
x = 1
x = x + 1
x = (y * 2) / (z * 2)
HouseCost = 375000
FileOpen = True
Range("Rok").Value = 2013
```



Wyrażenia mogą być na tyle złożone, na ile jest to potrzebne. Aby poprawić czytelność dłuższych wyrażeń, używaj znaku kontynuacji wiersza kodu (jest to znak podkreślenia poprzedzony spacją).

Wyrażenia często zawierają funkcje, takie jak predefiniowane funkcje języka VBA, funkcje skoroszytów Excela lub funkcje napisane przez Ciebie w języku VBA. Funkcje omówię w rozdziale 9.

O znaku równości

Jak widziałeś w poprzednim przykładzie, znak równości funkcjonuje w VBA jako operator przypisania. Prawdopodobnie zwykleś traktował ten znak jako matematyczny symbol równości. Dlatego też instrukcje przypisania podobne do tej mogą wprawić w niezłe osłupienie:

```
z = z + 1
```

W jakim zwariowanym świecie z równe jest samemu sobie plus 1? Odpowiedź brzmi: w żadnym z poznanych dotąd. W tym przypadku w wyniku wykonania operacji przypisania wartość zmiennej z zostaje zwiększoną o 1. Jeśli więc z równe jest 12, wykonanie tej instrukcji spowoduje, że będzie wynosić 13. Po prostu zapamiętaj, że w instrukcjach przypisania znak równości używany jest nie jako znak równości, lecz jako operator.

Proste operatory

Operatory odgrywają ważną rolę w języku VBA. Poza znakiem równości, opisanym w poprzednim punkcie, VBA udostępnia kilka innych operatorów, które znajdziesz w tabeli 7.3. Powinieneś je znać, ponieważ tych samych operatorów (z wyjątkiem Mod) używa się w formułach arkusza.

Tabela 7.3. Operatory języka VBA

Symbol operatora	Funkcja
+	Dodawanie
*	Mnożenie
/	Dzielenie
-	Odejmowanie
^	Potęgowanie
&	Konkatenacja ciągów znaków
\	Dzielenie całkowite (wynikiem jest zawsze liczba całkowita)
Mod	Arytmetyka modulo (wyznaczanie reszty z dzielenia)

Kiedy piszesz formułę w Excelu, działanie modulo (wyznaczanie reszty z dzielenia) wykonuje się przy użyciu funkcji MOD. Dla przykładu poniższa formuła zwróci wynik 2 (jest to reszta z dzielenia 12 przez 5).

```
=MOD(12;5)
```

W VBA operator Mod używany jest w sposób zaprezentowany poniżej. Po wykonaniu tej instrukcji wartością zmiennej z będzie 5.

```
z = 12 Mod 5
```



Konkatenacja jest terminem należącym do żargonu programistów i oznacza mniej więcej tyle, co po prostu *łączenie* (brzmi za to bardziej naukowo, co pewnie dodaje prestiżu). Zamiast więc mówić o *łączeniu fragmentów tekstu* (w wyniku czego powstaje nowy tekst), programiści wolą używać bardziej wysublimowanego wyrażenia *konkatenacja łańcuchów znaków*.

Jak pokazano w tabeli 7.4, VBA dysponuje pełnym wachlarzem operatorów logicznych. Z tej grupy najczęściej używane są Not, And oraz Or.

Tabela 7.4. Operatory logiczne języka VBA

Operator	Funkcja
Not	Negacja logiczna wyrażenia
And	Koniunkcja logiczna (iloczyn logiczny) dwóch wyrażeń
Or	Alternatywa logiczna (suma logiczna) dwóch wyrażeń
XoR	Alternatywa wykluczająca (różnica symetryczna) dwóch wyrażeń
Eqv	Równoważność logiczna dwóch wyrażeń
Imp	Implikacja logiczna dwóch wyrażeń

Priorytety operatorów w VBA są dokładnie takie same jak w formułach Excela. Najwyższy priorytet ma potęgowanie. Następne są mnożenie i dzielenie, a za nimi plasują się dodawanie i odejmowanie. Aby zmienić naturalną kolejność wykonywania działań, możesz użyć nawiasów. Cokolwiek znajdzie się wówczas między nawiasami, będzie miało pierwszeństwo przed innymi operatorami. Spójrz na poniższy kod.

```
x = 3
y = 2
z = x + 5 * y
```

Jaka będzie wartość z po wykonaniu tych instrukcji? Jeśli odpowiedziałeś, że 13, dostajesz złoty medal za znajomość kolejności wykonywania działań. Jeśli odpowiedziałeś, że 16, przeczytaj to: operacja mnożenia ($5 * y$) wykonywana jest w pierwszej kolejności, a jej wynik jest dodawany do x. Jeśli odpowiedziałeś, że wynikiem jest liczba inna niż 13 i 16, pozostawię to bez komentarza.

Nawiasem mówiąc, mam do zapamiętania ważniejsze rzeczy niż kolejność wykonywania działań, dlatego też często używam nawiasów również tam, gdzie nie są konieczne. Przykładowo ostatnią instrukcję przypisania normalnie zapisałbym w taki sposób:

```
z = x + (5 * y)
```



Nie obawiaj się używać nawiasów również tam, gdzie nie są wymagane, szczególnie wtedy, kiedy dzięki nim kod będzie łatwiejszy do zrozumienia. W VBA nadmiarowe nawiasy nie mają żadnego znaczenia.

Praca z tablicami

Tablice obsługiwane są w większości języków programowania. **Tablica** to grupa zmiennych dzielących tę samą nazwę. Aby odwołać się do konkretnej zmiennej z tablicy, należy podać nazwę tablicy oraz numer indeksu tej zmiennej w nawiasach. Aby na przykład przechować nazwy wszystkich miesięcy, można zdefiniować tablicę dwunastu zmiennych lańcuchowych. Jeśli nazwę tablicy określiś jako MonthNames, będziesz mógł się odwołać do jej pierwszego elementu za pomocą wyrażenia MonthNames(1), do elementu drugiego poprzez MonthNames(2) i tak dalej.

Deklarowanie tablic

Aby można było korzystać z tablicy, trzeba ją najpierw zadeklarować. Wyjątków od tej reguły nie przewidziano. W odróżnieniu od zwykłych zmiennych, VBA jest tutaj bardzo restrykcyjne. Tablicę deklaruje się za pomocą wyrażenia `Dim` lub `Public`, tak samo jak zwykłą zmienną. Należy tutaj jednak dodatkowo określić liczbę elementów tablicy. Robi się to, podając pierwszy numer indeksu, słowo **kluczowe** `To` oraz ostatni numer indeksu, całość umieszcza się jednocześnie w nawiasie. W poniższym przykładzie pokazano, jak zadeklarować tablicę składającą się ze 100 liczb całkowitych.

```
Dim MyArray(1 To 100) As Integer
```

Deklarując tablicę, możesz podać tylko górną granicę zakresu indeksów. Jeśli pominiesz numer indeksu początkowego, VBA przyjmie 0 jako wartość domyślną. Obie poniższe instrukcje deklarują zatem tę samą tablicę zawierającą 101 elementów.

```
Dim MyArray(0 To 100) As Integer
Dim MyArray(100) As Integer
```



Jeśli chcesz, aby domyślnym indeksem początkowym tablic deklarowanych w VBA było 1 (zamiast 0), zatrzymaj poniższą instrukcję w sekcji *Declarations* odpowiedniego modułu.

```
Option Base 1
```

Instrukcja ta wymusza na VBA przyjmowanie 1 jako pierwszego numeru indeksu, jeśli w deklaracji tablicy określono jedynie wartość indeksu końcowego. Przy założeniu, że użyto powyższej instrukcji, efekt wywołania następujących dwóch instrukcji jest identyczny — deklarowana jest tablica zawierająca 100 elementów.

```
Dim MyArray(1 To 100) As Integer
Dim MyArray(100) As Integer
```

Tablice wielowymiarowe

Wszystkie tablice, jakie utworzyliśmy w poprzednich przykładach, były tablicami jednowymiarowymi. Każdą z nich możesz sobie wyobrazić jako pojedynczą oś wartości. Niemniej jednak, tablice tworzone w języku VBA mogą mieć aż 60 wymiarów (w praktyce rzadko będziesz potrzebować tablicy o więcej niż dwóch lub trzech wymiarach). W poniższym przykładzie deklarowana jest dwuwymiarowa tablica 81 liczb całkowitych.

```
Dim MyArray(1 To 9, 1 To 9) As Integer
```

Tablicę tę możesz sobie wyobrazić jako tabelę o wymiarach 9×9 — w sam raz na planszę do sudoku.

Aby odwołać się do wybranego elementu takiej tablicy, należy określić dwa numery indeksów (odpowiadające numerowi wiersza i kolumny w tabeli). W poniższym przykładzie pokazano, jak przypisać wartość wybranemu elementowi tej tablicy.

```
MyArray(3, 4)= 125
```

Instrukcja ta przypisuje wartość pojedynczemu elementowi tablicy. Przyrównując tablicę do tabeli o wymiarach 9×9 , można powiedzieć, że wartość 125 została przypisana elementowi znajdującemu się w trzecim wierszu i czwartej kolumnie tabeli.

Oto jak można zadeklarować tablicę trójwymiarową, zawierającą 1000 elementów.

```
Dim MyArray(1 To 10, 1 To 10, 1 To 10) As Integer
```

Tablicę trójwymiarową przyrównać można do sześcianu. Wizualizacja tablicy o więcej niż trzech wymiarach jest już trudniejsza. Wybacz, ale jeszcze nie opanowałem tej sztuki.

Tablice dynamiczne

Möesz również tworzyć **tablice dynamiczne**. Tablica dynamiczna nie posiada odgórnie określonej liczby elementów. Tablice tego typu deklaruje się, używając pary pustych nawiasów.

```
Dim MyArray() As Integer
```

Zanim można będzie użyć takiej tablicy, trzeba posłużyć się instrukcją `ReDim`, aby przekazać do VBA informację, ile elementów posiada tablica. Zazwyczaj liczba elementów tablicy określana jest w trakcie wykonywania programu. Instrukcji `ReDim` można używać wielokrotnie, zmieniając w ten sposób rozmiar tablicy tak często, jak potrzeba. W poniższym przykładzie pokazano, jak zmienić liczbę elementów w tablicy dynamicznej. Zakładamy przy tym, że zmienna `NumElements` zawiera wartość, która została obliczona w kodzie programu.

```
ReDim MyArray(1 To NumElements)
```



Zmiana rozmiarów tablicy za pomocą instrukcji `ReDim` powoduje usunięcie wszystkich wartości, jakie były do tej pory przechowywane w jej elementach. Aby uniknąć usunięcia starych wartości, należy użyć słowa kluczowego `Preserve`. W poniższym przykładzie demonstruję, jak zachować wartości przechowywane w tablicy przy zmianie jej wymiarów.

```
ReDim Preserve MyArray(1 To NumElements)
```

Jeśli tablica `MyArray` posiada aktualnie dziesięć elementów, a wartością zmiennej `NumElements` jest 12, to po wywołaniu powyższej instrukcji pierwsze dziesięć elementów tablicy pozostanie bez zmian, a dodatkowo w tablicy pojawią się dwa nowe miejsca na dodatkowe elementy (w wyniku dopełnienia do wartości przechowywanej w zmiennej `NumElements`). Jeśli jednak wartością zmiennej `NumElements` będzie 7, to pierwsze siedem elementów zostanie zachowanych, ale pozostałe trzy zostaną unicestwione.

Temat tablic pojawi się ponownie w rozdziale 10., gdy omawiać będę działanie pętli.

Stosowanie etykiet

We wczesnych wersjach języka BASIC każdy wiersz kodu musiał rozpoczynać się od numeru wiersza. Jeśli przykładowo w latach 70. ubiegłego wieku (ubrany — oczywiście — w dzwony) napisałeś w tym języku jakiś program, mógł on wyglądać mniej więcej tak:

```
010: LET X=5
020: LET Y=3
030: LET Z=Z*Y
040: PRINT Z
050: END
```



VBA dopuszcza użycie takiego numerowania wierszy, a nawet dozwolone są etykiety tekstowe. Zazwyczaj nie używa się etykiet dla każdej linii kodu, jednak czasami etykieta taka może być przydatna. Etykietę wstawia się na przykład wtedy, gdy chce się użyć instrukcji `GoTo` (opiszę ją w rozdziale 10.). Etykieta musi zaczynać się od pierwszego znaku w wierszu, który nie może być spacją, oraz musi kończyć się średnikiem.

Informacje przekazane w tym rozdziale zostaną doprecyzowane dalej w tej książce. Jeśli chcesz dowiedzieć się więcej o elementach języka VBA, polecam system pomocy języka VBA. Znajdziesz tam tyle szczegółów, ile będziesz potrzebował (lub miał ochotę poznać).

Rozdział 8

Praca z obiektami Range

W tym rozdziale:

- dowiesz się, dlaczego obiekty Range są tak ważne,
- przeczytasz o różnych sposobach odwoływania się do zakresów,
- odkryjesz niektóre przydatniejsze właściwości obiektu Range,
- poznasz kilka ważnych metod obiektu Range.

Wtym rozdziale poszperam nieco głębiej w Excelu i omówię obiekty Range. Excel opiera się na komórkach, a obiekt Range jest kontenerem dla komórek. Dlaczego powinieneś tyle wiedzieć o obiektach Range? Ponieważ właśnie na nich opiera się duża część programowania w Excelu. Możesz mi podziękować później.

Szybka powtórka

Obiekt Range prezentuje zakres zawarty w obiekcie Worksheet. Obiekty Range, podobnie jak wszystkie inne obiekty, posiadają właściwości (możesz je sprawdzać, a czasami również zmieniać) oraz metody (wykonujące działania na obiekcie).

Obiekt Range może mieć wielkość zaledwie jednej komórki (na przykład B4), może też obejmować wszystkie 17 179 869 184 komórki skoroszytu (A1:XFD1048576).

Odwołując się do obiektu Range, adres poprzedzamy i zamykamy cudzysłowami, tak jak poniżej.

```
Range("A1:C5")
```

Cudzysłów jest potrzebny, nawet jeśli zakres składa się z jednej komórki.

```
Range("K9")
```

Jeżeli zakres posiada nazwę (utworzoną w Excelu za pomocą polecenia *Formuły/Nazwy zdefiniowane/Definiuj nazwę*), możesz użyć wyrażenia, takiego jak poniższe.

```
Range("ListaCen")
```

Dopóki nie wskażesz, do którego arkusza odwołuje się zakres, Excel zakłada, że odnosisz się do zakresu zawartego w aktywnym arkuszu. Jeśli aktywne jest cokolwiek innego niż arkusz danych (na przykład arkusz wykresu), odnalezienie odwołania zakresu nie uda się — makro wyświetli komunikat błędu i zatrzyma się.



128 Część III: Podstawy programowania

Można odwołać się do zakresu spoza aktywnego arkusza, określając w odwołaniu zakresu nazwę arkusza z aktywnego skoroszytu.

```
Worksheets("Arkusz1").Range("A1:C5")
```

Jeśli musisz się odwołać do zakresu w innym skoroszycie (to jest jakimkolwiek innym niż aktywny skoroszyt), możesz użyć podobnej instrukcji.

```
Workbooks("Budżet.xlsx").Worksheets("Arkusz1").Range("A1:C5")
```

Obiekt Range może składać się z jednego lub wielu całych wierszy albo jednej lub wielu kolumn. Do całego wiersza (w tym przypadku wiersza 3.) możesz się odwołać, używając poniższej składni.

```
Range("3:3")
```

Do całej kolumny (czwartej, w tym przykładzie) możesz się odwołać w pokazany niżej sposób.

```
Range("D:D")
```

Zakresy nieciągłe zaznacza się w Excelu, przytrzymując klawisz *Crtl* i jednocześnie zaznaczając myszą poszczególne zakresy. Zaznaczenie zakresu nieciągłego przedstawiam na rysunku 8.1. Nie powinno Cię zdziwić, że również VBA umożliwia pracę z zakresami nieciągłymi. Poniższe wyrażenie odnosi się do zakresu nieciągłego składającego się z dwóch obszarów. Zwróć uwagę, że obszary te oddzielone są przecinkiem.

```
Range("A1:B8,D9:G16")
```

A	B	C	D	E	F	G	H	I
1								
2								
3								
4								
5								
6								
7								
8								
9								
10								
11								
12								
13								
14								
15								
16								
17								
18								
19								

Rysunek 8.1.
Zaznaczenie
zakresu
nieciągłego

Miej na uwadze, że niektóre metody i właściwości użyte z zakresami nieciągłymi mogą powodować problemy. Być może będziesz musiał obsługiwać każdy obszar osobno, za pomocą pętli.



Inne sposoby odwoływania się do zakresu

Im więcej pracujesz z VBA, tym bardziej zdajesz sobie sprawę, że to całkiem dobrze przemyślany i zazwyczaj dość logiczny język (wbrew temu, co możesz o nim myśleć teraz). W wielu przypadkach VBA oferuje kilka sposobów wykonania jakiejś akcji, możesz więc wybrać najbardziej dogodne rozwiążanie Twojego problemu. W tym podrozdziale omówię kilka innych sposobów odwoływania się do zakresu komórek.



W tym rozdziale zaledwie powierzchownie opisuję część problematyki związanej z właściwościami i metodami obiektu Range. Pracując z VBA, prawdopodobnie potrzebować będziesz również innych właściwości i metod. System pomocy jest najlepszym miejscem, aby je poznać. Innym dobrym pomysłem jest rejestrowanie swoich działań i analizowanie kodu generowanego przez Excel. Pewnie powoli masz już dość słuchania tej rady, ale to naprawdę dobra rada.

Właściwość Cells

Zamiast korzystać ze słowa kluczowego Range, możesz odwołać się do zakresu przy użyciu właściwości Cells.



Zauważ, że napisałem właściwość Cells, nie natomiast obiekt Cells czy też kolekcja Cells. Mimo że właściwość Cells może wydawać się obiektem lub kolekcją, w rzeczywistości nimi nie jest. Cells jest raczej właściwością, którą VBA wpierw interpretuje, a następnie zwraca obiekt (obiekt Range, dokładniej rzecz ujmując). Nie martw się, jeśli wydaje Ci się to dziwne — nawet Microsoft zdaje się w tym gubić. W niektórych wcześniejszych wersjach Excela właściwość Cells znana była jako metoda Cells. Niezależnie od tego, czym właściwie jest, zapamiętaj po prostu, że Cells to bardzo wygodny sposób odwoływania się do zakresów.

Właściwość Cells przyjmuje dwa argumenty: numer wiersza i numer kolumny. Oba te argumenty są liczbami, nawet jeśli na ogół, odnosząc się do kolumn, używamy liter. Wyrażenie w poniższym przykładzie odwołuje się do komórki C2 w arkuszu Arkusz2.

```
Worksheets("Arkusz2").Cells(2, 3)
```

Właściwości Cells można użyć również przy odwołaniach do zakresów obejmujących wiele komórek. W poniższym przykładzie przedstawiam składnię, której należy użyć.

```
Range(Cells(1, 1), Cells(10, 8))
```

Wyrażenie to odnosi się do 80-komórkowego zakresu rozciągającego się od komórki A1 (wiersz 1., kolumna 1.) do komórki H10 (wiersz 10., kolumna 8.).

Następne dwie instrukcje dają w efekcie ten sam rezultat — obie wprowadzają wartość 99 do zakresu komórek o wymiarach 10×8. Dokładniej mówiąc, instrukcje te ustawiają właściwość Value obiektu Range.

```
Range("A1:H10").Value = 99
Range(Cells(1, 1), Cells(10, 8)).Value = 99
```



Korzyści wynikające z odwoływania się do zakresów za pomocą właściwości Cells ujawnią się, gdy jako argumentów właściwości Cells użyjesz zmiennych w miejsce liczb. Wszystko się wyjaśni, jak tylko poznasz pętle, które omówię w rozdziale 10.

Właściwość Offset

Właściwość Offset to następny wygodny sposób odwoływania się do zakresów. Właściwość ta, operując na obiekcie Range i zwracając kolejny obiekt Range, umożliwia odwołanie się do komórki oddalonej od innej komórki o określona liczbę wierszy i kolumn.

Podobnie jak właściwość Cells, właściwość Offset również przyjmuje dwa argumenty. Pierwszy argument określa liczbę wierszy, drugi natomiast liczbę kolumn, o jaką zostanie przesunięty zwrócony zakres.

Poniższe wyrażenie odwołuje się do komórki znajdującej się jeden wiersz poniżej i o dwie kolumny na prawo od komórki A1. Innymi słowy, odnosi się do komórki powszechnie znanej jako C2.

```
Range("A1").Offset(1, 2)
```

Argumentami właściwości Offset mogą być również wartości ujemne. Przesunięcie o *ujemną liczbę wierszy* odwołuje się do wiersza znajdującego się *powyżej zakresu*. Analogicznie, przesunięcie o *ujemną liczbę kolumn* odnosi się do kolumny znajdującej się *na lewo od zakresu*. Poniżej przedstawiam przykład z odwołaniem do komórki A1.

```
Range("C2").Offset(-1, -2)
```

Jak zapewne się domyślasz, możesz użyć również 0 jako jednego lub obu argumentów przesunięcia. Poniższe wyrażenie odwołuje się do komórki A1.

```
Range("A1").Offset(0, 0)
```

Oto instrukcja, która wstawi aktualną godzinę do komórki znajdującej się na prawo od komórki aktywnej.

```
ActiveCell.Offset(0, 1) = Time
```

Gdy rejestrujesz makro przy użyciu odwołań względnych, Excel często posługuje się właściwością Offset. Po przykład wróć do rozdziału 6.

Właściwość Offset przydaje się najbardziej, gdy jako argumentów używasz zmiennych zamiast konkretnych wartości. Kilka przykładów takiego użycia przesunięć przedstawię w rozdziale 10.



Wybrane właściwości obiektu Range

Obiekt Range posiada dziesiątki właściwości. Możesz pisać programy w VBA bez przerwy przez następne 12 miesięcy, a i tak nigdy nie użyjesz ich wszystkich. W tym podrozdziale opiszę pokróćce niektóre powszechnie używane właściwości obiektu Range. Szczegółowe informacje znajdziesz w systemie pomocy edytora VBE.



Niektóre właściwości obiektu Range służą *tylko do odczytu*, czyli w kodzie programu możesz sprawdzić ich wartości, ale nie możesz ich zmieniać („obejrzyj, ale nie dotykaj”). Przykładowo każdy obiekt Range ma właściwość Address, która przechowuje adres zakresu. Masz dostęp do tej właściwości, ale ponieważ jest ona tylko do odczytu, nie możesz jej zmienić. Jeśli się nad tym zastanowisz, przyznasz, że to bardzo sensowne.

Nawiasem mówiąc, podane przykłady to raczej typowe instrukcje, a nie kompletne procedury. Jeśli chciałbyś któryś z wypróbować (a powinieneś), utwórz procedurę Sub. Poza tym, wiele tych instrukcji działa poprawnie tylko wtedy, gdy aktywnym arkuszem jest arkusz danych.

Właściwość Value

Właściwość `Value` reprezentuje wartość zawartą w komórce. Jest to właściwość do odczytu i zapisu, tak więc w VBA możesz zarówno odczytywać, jak i zmieniać jej wartość.

Poniższa instrukcja wyświetla okno z komunikatem podającym wartość zawartą w komórce A1 arkusza Arkusz1.

```
MsgBox Worksheets("Arkusz1").Range("A1").Value
```

Jest rzeczą logiczną, że właściwość `Value` można odczytać jedynie w przypadku obiektu Range obejmującego tylko jedną komórkę. Poniżej przytaczam przykład instrukcji, która zgłosi błąd.

```
MsgBox Worksheets("Arkusz1").Range("A1:C3").Value
```

Możliwa jest jednak zmiana właściwości `Value` dla zakresów o dowolnych rozmiarach. Następująca instrukcja wprowadza liczbę 123 do każdej komórki w zakresie.

```
Worksheets("Arkusz1").Range("A1:C3").Value = 123
```



`Value` jest domyślną właściwością obiektu Range. Innymi słowy, jeżeli pominiiesz właściwość obiektu Range, Excel użyje właściwości `Value`. Obie poniższe instrukcje wprowadzają wartość 75 do komórki A1 aktywnego arkusza.

```
Range("A1").Value = 75
Range("A1") = 75
```



Przypisywanie wartości wielokomórkowego zakresu do zmiennej

Niezupełnie miałem rację, pisząc, że „właściwość `Value` można odczytać jedynie w przypadku obiektu `Range` obejmującego tylko jedną komórkę”. W rzeczywistości wartości zakresu obejmującego wiele komórek można przypisać zmiennej, o ile tylko jest to zmienna typu `Variant`. Jest tak, ponieważ `Variant` może zachowywać się jak tablica. Oto przykład.

```
Dim x As Variant
x = Range("A1:C3").Value
```

Teraz możesz traktować zmienną `x` jak tablicę. Przykładowo poniższa instrukcja zwraca wartość komórki B1.

```
MsgBox x(1, 2)
```

Właściwość `Text`

Właściwość `Text` zwraca łańcuch znaków przedstawiający tekst w postaci, w jakiej jest wyświetlany w komórce, czyli z uwzględnieniem formatowania. Właściwość `Text` jest tylko do odczytu. Założymy na przykład, że komórka A1 zawiera wartość 12,3 wyświetlana z dwoma miejscami po przecinku i formatowaniem walutowym (12,30 zł). Poniższa instrukcja wyświetli okno z komunikatem o treści 12,30 zł.

```
MsgBox Worksheets("Arkusz1").Range("A1").Text
```

Z kolei następna instrukcja wyświetli okno z komunikatem o treści 12,3.

```
MsgBox Worksheets("Arkusz1").Range("A1").Value
```

Gdy komórka zawiera formułę, właściwość `Text` zwraca wynik formuły. Gdy komórka zawiera tekst, wówczas właściwości `Text` oraz `Value` zawsze zwracają to samo, ponieważ tekst, w przeciwieństwie do liczb, nie może być tak sformatowany, aby był wyświetlany w inny sposób.

Zwróć uwagę, że formatowanie liczb w Excelu zależy również od ustawień regionalnych. Dla języka polskiego separatorem dziesiętnym jest przecinek. Ułamki dziesiętne będą więc wyświetlane w komórkach Excela oraz oknach komunikatu w tym formacie (na przykład 12,3). Niezależnie od ustawień regionalnych, w kodzie VBA separatorem dziesiętnym jest kropka (na przykład 12.3). Aby wstawić liczbę 12,3 do komórki A1 aktywnego arkusza, powinieneś użyć polecenia przedstawionego poniżej.

```
Range("A1").Value = 12.3
```

Właściwość Count

Właściwość **Count** zwraca liczbę komórek należących do zakresu. Zlicza ona wszystkie komórki, a nie tylko niepuste. Jak się zapewne spodziewasz, **Count** jest właściwością tylko do odczytu. Pokazana niżej instrukcja odczytuje wartość **Count** zakresu A1:C3 i wyświetla wynik (9) w oknie komunikatu.

```
MsgBox Range("A1:C3").Count
```

Właściwości Column i Row

Właściwość **Column** zwraca numer kolumny zakresu obejmującego jedną komórkę. Jej siostrzana właściwość **Row** zwraca numer wiersza jednokomórkowego zakresu. Obie są właściwościami tylko do odczytu. Przykładowo poniższa instrukcja wyświetla 6, jako że komórka F3 znajduje się w szóstej kolumnie.

```
MsgBox Sheets("Arkusz1").Range("F3").Column
```

Następne wyrażenie wyświetla 3, ponieważ komórka F3 leży w trzecim rzędzie.

```
MsgBox Sheets("Arkusz1").Range("F3").Row
```

Jeżeli obiekt **Range** zawiera więcej niż jedną komórkę, właściwość **Column** zwróci numer kolumny pierwszej komórki w zakresie, właściwość **Row** natomiast — numer wiersza pierwszej komórki w zakresie.

Nie pomył właściwości **Column** i **Row** z właściwościami **Columns** i **Rows**. Właściwości **Column** i **Row** zwracają pojedynczą wartość, podczas gdy właściwości **Columns** i **Rows** zwracają obiekt **Range**. Jaką różnicę robi jedno „s”!

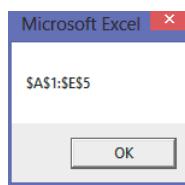
Właściwość Address

Właściwość **Address** (tylko do odczytu) zwraca adres komórek obiektu **Range** w postaci odwołania bezwzględnego (ze znakiem dolara przed literą oznaczającą kolumnę i przed numerem wiersza). Poniższa instrukcja wyświetla okno komunikatu z rysunku 8.2.

```
MsgBox Range(Cells(1, 1), Cells(5, 5)).Address
```

Rysunek 8.2.

Okno komunikatu wyświetlające właściwość **Address** zakresu o wymiarach 5×5



Właściwość HasFormula

Służąca tylko do odczytu właściwość **HasFormula** zwraca **True**, jeżeli obejmujący jedną komórkę zakres zawiera formułę. Właściwość zwraca **False**, jeżeli komórka nie zawiera formuły. Kiedy zakres zawiera więcej niż jedną komórkę, VBA zwraca **True** tylko w przypadku, gdy wszystkie komórki w zakresie zawierają formuły, lub **False**, jeśli żadna z komórek w zakresie nie zawiera formuły. Właściwość zwraca **Null**, jeżeli tylko część komórek w zakresie, lecz nie wszystkie, zawiera formuły. **Null** jest taką ziemią niczym — odpowiedź nie brzmi ani **True**, ani **False**. Znaczy to, że każda z komórek w zakresie może zawierać formułę — albo i nie.



Bądź ostrożny, pracując z właściwościami, które mogą zwracać **Null**, gdyż jedynym typem danych obsługującym **Null** jest **Variant**.

Dla przykładu założmy, że komórka A1 zawiera wartość, a komórka A2 — formułę. Poniższe instrukcje wygenerują błąd, ponieważ zakres nie składa się z samych komórek zawierających formułę lub niezawierających formuły.

```
Dim FormulaTest As Boolean
FormulaTest = Range("A1:A2").HasFormula
```

Typ danych **Boolean** obsługuje jedynie wartości **True** i **False**. Jeżeli zwrócona zostanie wartość **Null**, Excel po prostu zbuntuje się i wyświetli komunikat o wystąpieniu błędu. Aby uniknąć takiej sytuacji, najlepiej upewnić się, że zmienna **FormulaTest** zadeklarowana jest nie jako **Boolean**, lecz jako **Variant**. W kolejnym przykładzie do określenia, jakiego typu jest zmienna **FormulaTest**, użyta została przydatna funkcja języka VBA o nazwie **TypeName** (wraz z konstrukcją warunkową **If-Then-Else**). Jeżeli zakres jest mieszany pod względem zawartości formuł w komórkach, wyświetla się okno z komunikatem *Mieszany!*. W przeciwnym przypadku wyświetla się *True* lub *False*.

```
Sub CheckForFormulas()
    Dim FormulaTest As Variant
    FormulaTest = Range("A1:A2").HasFormula
    If TypeName(FormulaTest) = "Null" Then
        MsgBox "Mieszany!"
    Else
        MsgBox FormulaTest
    End If
End Sub
```

W rozdziale 10. znajdziesz więcej informacji o użyciu konstrukcji **If-Then-Else**.

Właściwość Font

Jak już wspomniałem wcześniej w tym rozdziale (patrz punkt „**Właściwość Cells**”), właściwość może zwracać obiekt. Właściwość **Font** obiektu **Range** jest następnym tego przykładem — właściwość **Font** zwraca obiekt **Font**.

Obiekt **Font**, jak pewnie się domyślasz, udostępnia wiele właściwości. Aby zmienić cechy czcionki danego zakresu, najpierw musisz odwoać się do obiektu **Font** tego zakresu, by dopiero potem manipulować właściwościami tego obiektu. Nieco to zawile, ale może następujący przykład pomoże to wyjaśnić.

W poniższej instrukcji użyto właściwości `Font` obiektu `Range`, by zwrócić obiekt `Font`. Następnie właściwości `Bold` obiektu `Font` przypisana została wartość `True`. Prosto mówiąc, instrukcja ta powoduje, że zawartość komórki wyświetlna jest pogrubioną czcionką.

```
Range("A1").Font.Bold = True
```

Prawdę mówiąc, nie musisz nawet wiedzieć, że pracujesz ze specjalnym obiektem `Font` zawartym w obiekcie `Range`. Jeśli tylko będziesz używał prawidłowej składni, wszystko będzie działać. W wielu przypadkach kod wygenerowany podczas rejestracji makr powie Ci wszystko, czego potrzebujesz, o poprawnej składni.

W rozdziale 6. znajdziesz więcej informacji na temat rejestracji makr.

Kilka słów o kolorach w Excelu

Przed pojawieniem się Excela 2007 Microsoft próbował nas przekonać, że 56 kolorów w zupełności wystarcza w pracy z arkuszami kalkulacyjnymi. Czasy się jednak zmieniają i teraz mamy do dyspozycji ponad 16 milionów kolorów — dokładnie mówiąc 16 777 216 kolorów.

Wiele obiektów posiada właściwość `Color`, która z kolei przyjmuje wartości z zakresu od 0 do 16 777 215. Nikt nie potrafi zapamiętać tylu wartości, dlatego też — na szczęście — istnieje łatwiejszy sposób określania kolorów, a mianowicie funkcja języka VBA o nazwie `RGB`. Wykorzystuje ona fakt, że każdy z tych 16 milionów kolorów można przedstawić jako połączenie różnych poziomów kolorów czerwonego, zielonego i niebieskiego. Trzy argumenty, które określamy w funkcji `RGB`, odpowiadają czerwonej, zielonej i niebieskiej składowej, a każdy z nich może przyjmować wartości z zakresu od 0 do 255.

Zauważ, że $256 \times 256 \times 256 = 16\,777\,216$. Tak się składa, że jest to liczba dostępnych kolorów. Czy to nie piękne, gdy liczby się zgadzają?

Poniżej znajduje się kilka przykładów użycia funkcji `RGB` do zmiany koloru tła komórki.

```
Range("A1").Interior.Color = RGB(0, 0, 0)  
'czarny'
```

```
Range("A1").Interior.Color = RGB(255, 0, 0)  
'czysty czerwony'
```

```
Range("A1").Interior.Color = RGB(0, 0, 255)  
'czysty niebieski'
```

```
Range("A1").Interior.Color = RGB(200, 89, 18)  
'brąz wpadający w pomarańcz'
```

```
Range("A1").Interior.Color = RGB(128, 128, 128)  
'szary o średniej jasności'
```

Jaka jest dokładna wartość zwracana przez `RGB(128, 128, 128)`? Poniższa instrukcja zdradzi, że jest to 8421504.

```
MsgBox RGB(128, 128, 128)
```

Jeśli chcesz użyć standardowych kolorów, wygodniejsze w użyciu mogą być predefiniowane stałe określające kolory: `vbBlack`, `vbRed`, `vbGreen`, `vbYellow`, `vbBlue`, `vbMagenta`, `vbCyan` czy też `vbWhite`. Instrukcja w poniższym przykładzie ustawia kolor tła komórki A1 na żółty.

```
Range("A1").Interior.Color = vbYellow
```

W Excelu 2007 wprowadzono też **kolory motywów**. Są to kolory, które pojawiają się, gdy używasz kontrolki z kolorami, takiej jak na przykład kontrolka *Kolor wypełnienia* w grupie *Czcionka* zakładki *Narzędzia główne*. Spróbuj zarejestrować makro, zmieniając kolory, a otrzymasz coś podobnego.

```
Range("A1").Interior.ThemeColor =  
x1ThemeColorAccent4
```

```
Range("A1").Interior.TintAndShade =  
0.399975585192419
```

O tak, kolejne dwie właściwości związane z kolorem do opanowania. Mamy tu kolor motywów (kolor bazowy, określony za pomocą predefiniowanej stałej) oraz wartość `TintAndShade` określającą, jak ciemny lub jasny jest kolor. Wartości `TintAndShade` leżą w zakresie od -1,0 do 1,0. Wartości dodatnie właściwości `TintAndShade` rozjaśniają kolor, a wartości ujemne powodują, że jest ciemniejszy. Gdy wybierasz kolor za pomocą właściwości `ThemeColor`, kolor się zmieni, jeśli wybierzesz inny motyw dokumentu (przy użyciu polecenia *Układ strony/Motyw/Motyw*).

Właściwość Interior

`Interior` to jeszcze jeden przykład właściwości zwracającej obiekt. Właściwość `Interior` obiektu `Range` zwraca obiekt `Interior` (dziwna nazwa, ale taka już jest). Ten typ odwoływania się do obiektu działa tak samo, jak w przypadku właściwości `Font`, którą opisałem w poprzednim punkcie.

Przykładowo poniższa instrukcja zmienia właściwość `Color` obiektu `Interior` zawartego w obiekcie `Range`.

```
Range("A1").Interior.Color = 8421504
```

Innymi słowy, instrukcja ta zmienia kolor tła komórki na szary o średniej jasności. O co chodzi? Nie wiesz, że 8421504 to szary? Aby zgłębić nieco piękny świat kolorów Excela, zajrzyj do ramki „Kilka słów o kolorach w Excelu”.

Właściwości `Formula` i `FormulaLocal`

Właściwości `Formula` i `FormulaLocal` reprezentują formułę zawartą w komórce. Są to właściwości do odczytu i zapisu, możesz więc po nie sięgnąć, by sprawdzić formułę w komórce lub też wprowadzić ją do komórki. Używając właściwości `Formula`, musisz operować oryginalnymi, angielskimi nazwami formuły, podczas gdy `FormulaLocal` pozwala posługiwać się nazwami formuły w języku odpowiadającym wersji językowej Excela (w tym wersji polskiej). Przykładowo poniższe dwie instrukcje wstawiają formułę `SUMA` (`SUM`) do komórek `A13` i `B13`.

```
Range("A13").Formula = "=SUM(A1:A12)"  
Range("B13").FormulaLocal = "=SUMA(B1:B12)"
```

Zauważ, że wartość formuły jest łańcuchem znaków, zawarta jest więc w znakach cudzysłowu.

Sprawa się nieco komplikuje, jeżeli formuła sama w sobie zawiera znaki cudzysłowu. Dla przykładu założmy, że chcesz wprowadzić poniższą formułę za pomocą VBA.

```
=SUMA(A1:A12)&" sklepów"
```

Formuła ta wyświetla wartość, po której następuje słowo `sklepów`. Aby dostosować ją do wymogów VBA, należy zastąpić każdy cudzysłów dwoma znakami cudzysłowu. Inaczej VBA pogubi się i stwierdzi, że kod zawiera błąd składni (bo rzeczywiście zawierał!). Tak więc wygląda instrukcja, która wprowadzi formułę zawierającą cudzysłowy.

```
Range("A13").FormulaLocal = "=SUMA(A1:A12)&"" sklepów"""
```

Nawiasem mówiąc, masz dostęp do właściwości `Formula` i `FormulaLocal` komórki, nawet jeśli komórka ta nie zawiera formuły. Jeśli komórka nie zawiera formuły, właściwości `Formula` i `FormulaLocal` zwracają to samo, co jej właściwość `Value`.

Jeśli chcesz się dowiedzieć, czy komórka zawiera formułę, użyj właściwości `HasFormula` (omówionej wcześniej w tym rozdziale).

Właściwość NumberFormat

Właściwość **NumberFormat** reprezentuje formatowanie liczb obiektu Range, wyrażone za pomocą kodów formatów liczbowych. Jest to właściwość do odczytu i zapisu, możesz więc sprawdzić lub zmienić jej wartość przy użyciu instrukcji VBA. Poniższe wyrażenie zmienia formatowanie liczb w kolumnie A na wartości procentowe z dwoma miejscami dziesiętnymi.

```
Columns("A:A").NumberFormat = "0.00%"
```

Postępuj według poniższych wskazówek, aby zobaczyć listę innych rodzajów formatowania liczb. Będzie jeszcze lepiej, jeśli wcześniej uruchomisz rejestrator makr.

1. Aktywuj arkusz.
2. Wywołaj okno dialogowe **Formatowanie komórek**, naciskając kombinację klawiszy **Ctrl+1**.
3. Kliknij zakładkę **Liczby**.
4. Wybierz kategorię **Niestandardowe**, by przejrzeć i zastosować kilka dodatkowych kodów formatów liczbowych.

Wybrane metody obiektu Range

Jak już się dowiedziałeś, metoda języka VBA wykonuje pewne działania. Obiekt Range zawiera dziesiątki metod, lecz i tym razem większość z nich nie będziesz potrzebować. W tym podrozdziale przedstawię kilka najczęściej używanych metod obiektu Range.

Metoda Select

Aby wybrać zakres komórek, użyj metody **Select**. Poniższa instrukcja wybiera zakres w aktywnym arkuszu.

```
Range("A1:C12").Select
```



Przed wybraniem zakresu na ogół warto użyć dodatkowej instrukcji, by zagwarantować, że aktywny jest właściwy skoroszyt. Jeśli na przykład Arkusz1 zawiera zakres, który chcesz wybrać, użyj poniższych instrukcji.

```
Sheets("Arkusz1").Activate
Range("A1:C12").Select
```

Wbrew temu, czego mógłbyś się spodziewać, kolejna instrukcja wygeneruje błąd, jeżeli Arkusz1 nie będzie aktywny przy jej wykonywaniu. Innymi słowy, należy użyć nie jednej, lecz dwóch instrukcji: jednej do aktywowania arkusza i drugiej do wybrania zakresu.

```
Sheets("Arkusz1").Range("A1:C12").Select
```



Gdy do wybierania zakresu używasz metody `GoTo` obiektu `Application`, nie musisz pamiętać, by wcześniej wybrać odpowiedni arkusz. Poniższa instrukcja aktywuje Arkusz1, a następnie wybiera zakres.

```
Application.Goto Sheets("Arkusz1").Range("A1:C12")
```

Metoda `GoTo` języka VBA jest odpowiednikiem naciśnięcia klawisza `F5` w Excelu, który wyświetla okno dialogowe *Przechodzenie do*.

Metody *Copy* i *Paste*

Operacje kopiowania i wklejania w VBA można wykonywać przy użyciu metod `Copy` i `Paste`. Zwróć uwagę, że są to metody dwóch różnych obiektów. Metoda `Copy` jest metodą obiektu `Range`, podczas gdy metoda `Paste` jest metodą obiektu `Worksheet`. To naprawdę ma sens: kopujesz zakres i wklejasz go do arkusza.

Poniższe krótkie makro (kod dzięki uprzejmości rejestratora makr) kopiuje zakres A1:A12 i wkleja go do tego samego arkusza, począwszy od komórki C1.

```
Sub CopyRange()
    Range("A1:A12").Select
    Selection.Copy
    Range("C1").Select
    ActiveSheet.Paste
End Sub
```



Zauważ, że obiekt `ActiveSheet` w poprzednim przykładzie użyty został z metodą `Paste`. Jest to specjalna wersja obiektu `Worksheet` odnosząca się do obecnie aktywnego arkusza. Zwróć również uwagę, że makro zaznacza zakres, zanim zostanie wklejony. Pamiętaj jednak, że nie musisz zaznaczać zakresów przed wykonaniem na nich operacji. W praktyce poniższa procedura wykonuje to samo zadanie, co poprzedni przykład, przy użyciu tylko jednej instrukcji.

```
Sub CopyRange2()
    Range("A1:A12").Copy Range("C1")
End Sub
```

Procedura ta wykorzystuje fakt, że metoda `Copy` może przyjmować argument określający zakres docelowy operacji kopiowania. Jest to coś, czego możesz się dowiedzieć, przeglądając system pomocy.

Metoda *Clear*

Metoda `Clear` usuwa zawartość zakresu oraz wszelkie formatowanie komórek. Jeśli na przykład chcesz się pozbyć wszystkiego w kolumnie D, zastosuj sztuczkę z poniższą instrukcją.

```
Columns("D:D").Clear
```

Powinieneś też wiedzieć o dwóch podobnych metodach. Metoda `ClearContents` usuwa zawartość zakresu, pozostawiając jednak nietknięte jego formatowanie. Z kolei metoda `ClearFormats` usuwa formatowanie zakresu, nie zmieniając zawartości komórek.

Metoda `Delete`

Usuwanie zawartości lub formatowania zakresu to nie to samo, co usuwanie samego zakresu. Gdy usuwasz *zakres*, Excel przesuwa pozostałe dookoła komórki tak, by zapełnić zakres, który usunąłeś.

W następującym przykładzie użyto metody `Delete` do usunięcia szóstego wiersza.

```
Rows("6:6").Delete
```

Gdy usuwasz zakres niebędący całą kolumną lub wierszem, Excel musi „wiedzieć”, jak przesunąć komórki. (Aby zobaczyć, jak to działa, poeksperymentuj z poleceniem *NARZĘDZIA GŁÓWNE/Komórki/Usuń*).

Poniższa instrukcja usuwa zakres, a następnie wypełnia powstałą przerwę, przesuwając pozostałe komórki w lewo.

```
Range("C6:C10").Delete xlToLeft
```

Metoda `Delete` przyjmuje argument wskazujący Excelowi, jak przesunąć pozostałe komórki. W tym przypadku jako argumentu użyłem predefiniowanej stałej (`xlToLeft`). Równie dobrze mógłbym posłużyć się inną predefiniowaną stałą, na przykład `xlUp`.

Rozdział 9

Praca z funkcjami VBA i arkusza kalkulacyjnego

W tym rozdziale:

- ▶ zastosujesz funkcje i utworzysz jeszcze bardziej zaawansowane wyrażenia,
- ▶ zastosujesz wbudowane funkcje VBA,
- ▶ użyjesz funkcji arkusza Excela w swoim kodzie VBA,
- ▶ napiszesz własne funkcje.

W poprzednich rozdziałach wspomniałem, że w swoich wyrażeniach VBA możesz stosować funkcje. Funkcje są dostępne w trzech różnych „smakach”: wbudowane funkcje języka VBA (waniliowy), wbudowane funkcje programu Excel (truskawkowy) i funkcje użytkownika napisane w języku VBA (czekoladowy). W tym rozdziale znajdziesz pełny i smakowity opis tych funkcji. Dzięki nim Twój kod VBA może wykonywać niezwykle rzeczy przy niewielkim wkładzie programowania lub nawet bez niego. Podoba Ci się ten pomysł? Zatem ten rozdział napisałem dla Ciebie.

Co to jest funkcja?

Z wyjątkiem bardzo niewielu osób, którym się wydaje, że Excel jest edytorem tekstu, wszyscy użytkownicy wykorzystują w swoich formułach funkcje arkusza kalkulacyjnego. Najpopularniejszą funkcją jest SUMA, ale oprócz niej masz do dyspozycji setki innych.

Funkcja w swojej istocie wykonuje obliczenia i zwraca wynik w postaci pojedynczej wartości. Funkcja SUMA zwraca — oczywiście — sumę zakresu wartości. Tak samo jest z funkcjami stosowanymi w wyrażeniach VBA — każda funkcja wykonuje swoją rolę i zwraca pojedynczy wynik.

Są trzy rodzaje funkcji używanych w VBA:

- ✓ funkcje wbudowane, dostępne wraz z VBA,
- ✓ funkcje arkusza kalkulacyjnego Excela,
- ✓ własne funkcje napisane przez Ciebie (lub kogoś innego) w języku VBA.

Dalej w tym rozdziale wyjaśnię różnice i (mam nadzieję) przekonam o użyteczności funkcji w kodzie VBA.

Stosowanie wbudowanych funkcji VBA

VBA oferuje wiele wbudowanych funkcji. Niektóre z nich mają argumenty, inne nie.

Przykłady funkcji VBA

W tym punkcie przedstawię dwa przykłady użycia funkcji VBA w kodzie. W wielu przykładach używam funkcji MsgBox do wyświetlania wartości w oknie dialogowym. Tak, MsgBox jest funkcją VBA — nietypową, niemniej jednak funkcją. Ta przydatna funkcja wyświetla komunikat w oknie dialogowym i również zwraca wartość. Więcej szczegółowych informacji na jej temat podaję w rozdziale 15.



Skoroszyt zawierający wszystkie przykłady jest dostępny na stronie internetowej książki.

Wyświetlenie systemowej daty i czasu

W pierwszym przykładzie wykorzystuję funkcję Date do wyświetlenia w oknie komunikatu bieżącej daty systemowej.

```
Sub ShowDate()
    MsgBox "Dzisiaj jest: " & Date
End Sub
```

Zauważ, że funkcja Date nie ma argumentu. W odróżnieniu od funkcji arkusza kalkulacyjnego, funkcje VBA, które nie mają argumentów, nie wymagają użycia pustych nawiasów. Co więcej, jeżeli wpiszesz puste nawiasy, edytor VBE szybko je usunie.

Aby odczytać czas systemowy, użyj funkcji Time. A jeżeli chcesz uzyskać pełne informacje, zastosuj funkcję Now, która zwraca datę i czas.

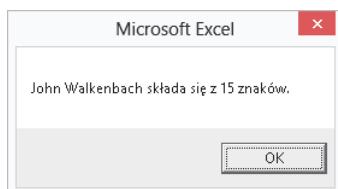
Określanie długości ciągu znaków

Poniższa procedura wykorzystuje funkcję VBA Len, zwracającą długość ciągu znaków. Funkcja ma jeden argument: ciąg znaków. Gdy uruchomisz poniższą procedurę, pojawi się okno z komunikatem zawierającym nazwę użytkownika programu Excel i liczbę znaków, z których się składa (zobacz rysunek 9.1).

```
Sub GetLength()
    Dim MyName As String
    Dim StringLength As Integer
    MyName = Application.UserName
    StringLength = Len(MyName)
    MsgBox MyName & " składa się z " & StringLength & " znaków."
End Sub
```

Excel zawiera również funkcję Dł, której możesz używać w formułach arkusza. Wersje tej samej funkcji w programie Excel i języku VBA działają tak samo.

Rysunek 9.1.
Obliczanie długości nazwy użytkownika programu Excel



Wyświetlenie nazwy miesiąca

Poniższa procedura wykorzystuje funkcję MonthName, zwracającą nazwę miesiąca. Funkcja przyjmuje jeden argument: liczbę całkowitą z przedziału od 1 do 12.

```
Sub ShowMonthName()
    Dim ThisMonth As Long
    ThisMonth = Month(Date)
    MsgBox MonthName(ThisMonth)
End Sub
```

Procedura wykorzystuje funkcję Month do odczytu bieżącego miesiąca (jako liczbę) i przypisuje ją zmiennej ThisMonth. Następnie funkcja MonthName zamienia liczbę na tekst. Jeżeli więc uruchomisz tę procedurę w kwietniu, okno z komunikatem wyświetli tekst *kwiecień*.

W zasadzie zmienna ThisMonth nie jest potrzebna. Ten sam efekt możesz osiągnąć za pomocą wyrażenia z trzema funkcjami VBA.

```
MonthName(Month(Date))
```

W tym przypadku bieżąca data jest przekazywana jako argument funkcji Month, zwracającej wartość, która jest przekazywana jako argument funkcji MonthName.

Określanie rozmiaru pliku

Poniższa procedura Sub wyświetla wielkość pliku wykonywalnego Excela, wyrazoną w bajtach. Do określenia tej wartości wykorzystana jest funkcja FileLen.

```
Sub GetFileSize()
    Dim TheFile As String
    TheFile = "C:\Program Files\Microsoft Office 15\root\office15\excel.exe"
    MsgBox FileLen(TheFile)
End Sub
```

Zwróć uwagę, że nazwa pliku jest wpisana *na stałe* (to znaczy, że w kodzie programu jest podana dokładna ścieżka pliku). To nie jest dobry sposób. Plik może nie znajdować się na dysku C albo folder programu Excel może mieć inną nazwę. Lepszym rozwiązaniem jest poniższa instrukcja.

```
TheFile = Application.Path & "\EXCEL.EXE"
```

Path jest właściwością obiektu Application i po prostu zwraca nazwę folderu, w którym zainstalowana jest aplikacja (czyli Excel), bez końcowego ukośnika.

Określenie typu wybranego obiektu

Poniższa procedura wykorzystuje funkcję `TypeName`, zwracającą typ (jako tekst) zaznaczenia w arkuszu kalkulacyjnym.

```
Sub ShowSelectionType()
    Dim SelType As String
    SelType = TypeName(Selection)
    MsgBox SelType
End Sub
```

Zaznaczeniem może być *Range* (zakres), *Picture* (obraz), *Rectangle* (prostokąt), *ChartArea* (obszar wykresu) lub innego typu obiekt, który można wybrać.



Funkcja `TypeName` jest bardzo uniwersalna. Możesz jej również używać do określenia typu zmiennej.

Funkcje VBA, które robią coś więcej niż tylko zwracanie wartości

Kilka funkcji VBA robi coś więcej niż obowiązkowe minimum. Funkcje te, zamiast jedynie zwracać wartość, wywołują przydatne efekty uboczne. W tabeli 9.1 wymieniam takie funkcje.

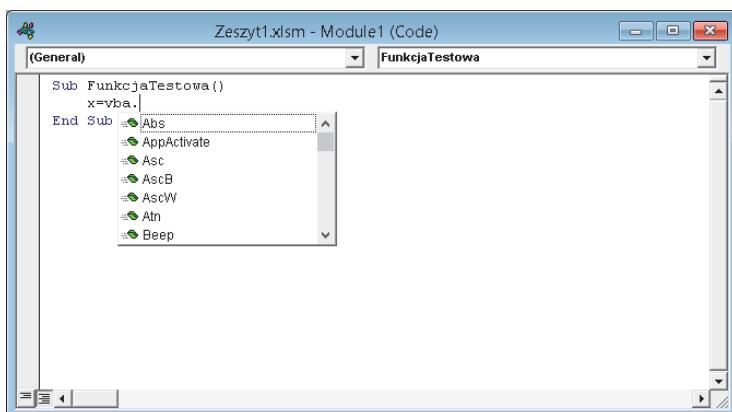
Tabela 9.1. Funkcje VBA niosące dodatkowe korzyści

Funkcja	Co robi?
<code>MsgBox</code>	Wyświetla poręczne okno dialogowe zawierające komunikat i przyciski. Funkcja zwraca kod określający przycisk kliknięty przez użytkownika. Więcej szczegółowych informacji podaję w rozdziale 15.
<code>InputBox</code>	Wyświetla proste okno dialogowe z prośbą do użytkownika o wprowadzenie danych. Funkcja zwraca dane wpisane w oknie przez użytkownika. Funkcję omawiam w rozdziale 15.
<code>Shell</code>	Uruchamia inny program. Funkcja zwraca unikatowy identyfikator zadania innego programu (lub kod błędu, jeśli nie można uruchomić tego programu).

Odkrywanie funkcji VBA

W jaki sposób możesz się dowiedzieć, jakie funkcje oferuje VBA? Dobre pytanie. Najlepszym źródłem jest system pomocy języka Visual Basic w programie Excel. Innym sposobem jest wpisanie w edytorze VBE znaków `vba` i kropki. Pojawi się lista elementów, przedstawiona na rysunku 9.2. Elementy oznaczone zielonymi ikonami są funkcjami. Jeżeli ten sposób nie działa, wybierz z menu głównego polecenie *Tools/Options*, kliknij zakładkę *Editor* i zaznacz opcję *Auto List Members*.

Sporządziłem częściową listę funkcji, którą przedstawiłem w tabeli 9.2. Pominąłem kilka bardziej wyspecjalizowanych i niezrozumiałych.



Rysunek 9.2.
Wyświetlenie
listy funkcji
VBA



Aby uzyskać szczegółowe informacje o wybranej funkcji, wpisz w module VBA jej nazwę, umieść cursor gdziekolwiek w obrębie tej nazwy i naciśnij klawisz F1.

Użycie funkcji arkusza kalkulacyjnego w VBA

Chociaż VBA oferuje przyzwoity asortyment wbudowanych funkcji, nie zawsze znajdziesz wśród nich dokładnie to, czego potrzebujesz. Na szczęście, możesz w procedurach VBA wykorzystać większość funkcji arkusza kalkulacyjnego programu Excel. Nie możesz jedynie użyć tych funkcji, które posiadają swoje odpowiedniki w VBA.

VBA udostępnia wbudowane funkcje za pomocą obiektu `WorksheetFunction`, wchodzącego w skład obiektu `Application`. Poniżej przedstawiam przykład użycia funkcji `SUM` (odpowiednik funkcji `SUMA` w polskiej wersji programu Excel) w instrukcji VBA.



```
Total = Application.WorksheetFunction.Sum(Range("A1:A12"))
```

W powyższym wyrażeniu możesz pominąć część `Application` lub `WorksheetFunction`. W obu przypadkach VBA domyśli się, co masz zamiar zrobić. Innymi słowy, poniższe wyrażenia są takie same.

```
Total = Application.WorksheetFunction.Sum(Range("A1:A12"))
Total = WorksheetFunction.Sum(Range("A1:A12"))
Total = Application.Sum(Range("A1:A12"))
```

Osobiście wolę stosować część `WorksheetFunction`, aby było jasne, że w kodzie używam funkcji arkusza Excel.

Tabela 9.2. Najbardziej przydatne wbudowane funkcje VBA

Funkcja	Co robi?
Abs	Zwraca wartość bezwzględną liczby.
Array	Zwraca obiekt typu Variant zawierający tabelę.
Choose	Zwraca wartość wybraną z listy elementów.
Chr	Zamienia kod ANSI na ciąg znaków.
CurDir	Zwraca bieżącą ścieżkę pliku.
Date	Zwraca bieżącą datę systemową.
DateAdd	Zwraca datę, do której został dodany zadany przedział czasu, na przykład po miesiącu od zadanej daty.
DateDiff	Zwraca liczbę całkowitą oznaczającą przedział czasu pomiędzy dwiema datami, na przykład liczbę miesięcy od daty urodzenia do teraz.
DatePart	Zwraca liczbę całkowitą zawierającą część zadanej daty, na przykład dzień.
DateSerial	Zamienia datę na liczbę seryjną.
DateValue	Zamienia ciąg znaków na datę.
Day	Zwraca dzień miesiąca z zadanej daty.
Dir	Zwraca nazwę pliku lub folderu zgodnego ze wzorcem.
Err	Zwraca kod błędu po jego wystąpieniu.
Error	Zwraca komunikat o błędzie, odpowiadający kodowi błędu.
Exp	Zwraca podstawę logarytmu naturalnego (liczbę e) podniesioną do potęgi.
FileLen	Zwraca liczbę bajtów w pliku.
Fix	Zwraca część całkowitą liczby.
Format	Wyświetla wyrażenie w zadanym formacie.
GetSetting	Zwraca wartość z rejestru Windows.
Hour	Zwraca część godzinową oznaczenia czasu.
InputBox	Wyświetla okno z żądaniem wprowadzenia danych przez użytkownika.
InStr	Zwraca pozycję jednego ciągu znaków w innym.
InStrRev	Zwraca pozycję jednego ciągu znaków w innym, licząc od jego końca.
Int	Zwraca część całkowitą liczby.
IsArray	Zwraca wartość <i>True</i> , jeżeli zmienna zawiera tabelę.
IsDate	Zwraca wartość <i>True</i> , jeżeli wyrażenie opisuje datę.
IsEmpty	Zwraca wartość <i>True</i> , jeżeli zmienna nie została zainicjowana.
IsError	Zwraca wartość <i>True</i> , jeżeli wyrażenie oznacza błąd.
IsMissing	Zwraca wartość <i>True</i> , jeżeli procedurze nie został przekazany opcjonalny argument.

Tabela 9.2. Najbardziej przydatne wbudowane funkcje VBA — ciąg dalszy

Funkcja	Co robi?
IsNull	Zwraca wartość <i>True</i> , jeżeli wyrażenie nie zawiera żadnych danych.
IsNumeric	Zwraca wartość <i>True</i> , jeżeli wyrażenie oznacza liczbę.
LBound	Zwraca najmniejszy indeks wymiaru tabeli.
LCase	Zwraca ciąg znaków zamieniony na małe litery.
Left	Zwraca zadaną liczbę znaków od początku ciągu tekstowego.
Len	Zwraca liczbę znaków w ciągu.
Mid	Zwraca zadaną liczbę znaków z ciągu tekstowego.
Minute	Zwraca część minutową wartości czasowej.
Month	Zwraca miesiąc z wartości daty.
MsgBox	Wyświetla okno z komunikatem i (opcjonalnie) zwraca wartość.
Now	Zwraca bieżącą systemową datę i czas.
Replace	Zamienia zadany fragment ciągu znaków na inny ciąg.
RGB	Zwraca numeryczną wartość RGB oznaczającą kolor.
Right	Zwraca określona liczbę znaków od końca ciągu.
Rnd	Zwraca losową liczbę z przedziału od 0 do 1.
Second	Zwraca część sekundową wartości czasowej.
She11	Uruchamia wykonywalny program.
Space	Zwraca ciąg znaków składający się z zadanej liczby spacji.
Split	Dzieli ciąg znaków na części, używając zadanego separatora.
Sqr	Zwraca pierwiastek kwadratowy z liczby.
String	Zwraca powtórzony znak lub ciąg znaków.
Time	Zwraca bieżący czas systemowy.
Timer	Zwraca liczbę sekund, poczawszy od północy.
TimeSerial	Zwraca wartość czasu dla zadanej godziny, minuty i sekundy.
TimeValue	Konwertuje ciąg znaków na liczbę seryjną reprezentującą czas.
Trim	Zwraca ciąg znaków bez początkowych i końcowych spacji.
TypeName	Zwraca ciąg znaków opisujący typ zmiennej.
UBound	Zwraca największy indeks wymiaru tabeli.
UCase	Zamienia ciąg znaków na wielkie litery.
Val	Zwraca wartość zawartą w ciągu znaków.
Weekday	Zwraca liczbę oznaczającą dzień tygodnia.
Year	Zwraca rok z wartości daty.

Przykłady funkcji arkusza kalkulacyjnego

W tej części pokażę, jak korzystać z funkcji arkusza w wyrażeniach VBA.

Wyszukiwanie minimalnej wartości w zadanym zakresie

Poniżej znajduje się przykład, w którym demonstruję użycie w procedurze VBA funkcji MAX arkusza. Procedura wyświetla maksymalną wartość z kolumny A bieżącego arkusza (zobacz rysunek 9.3).

```
Sub ShowMax()
    Dim TheMax As Double
    TheMax = WorksheetFunction.MAX(Range("A:A"))
    MsgBox TheMax
End Sub
```

The screenshot shows a Microsoft Excel spreadsheet with data in column A. Row 1 contains the header "Aby obejrzeć kod przykładów do rozdziału 9 naciśnij Alt+F11.". Rows 2 through 19 contain numerical values. A message box titled "Microsoft Excel" is displayed, showing the value "3892,81" and an "OK" button. The Excel ribbon at the bottom shows "Arkusz1".

A	B	C	D	E	F	G	H	I
1	Aby obejrzeć kod przykładów do rozdziału 9 naciśnij Alt+F11.							
2	2 865,77							
3	372,43							
4	2 775,52							
5	1 278,63							
6	437,69							
7	2 798,53							
8	3 892,81							
9	316,10							
10	2 988,25							
11	2 376,19							
12	1 940,09							
13	2 559,33							
14	551,45							
15	2 152,12							
16	654,88							
17	1 706,04							
18								
19								

Rysunek 9.3.
Użycie funkcji
arkusza kalku-
lacyjnego
w kodzie VBA

Aby otrzymać minimalną wartość z zakresu, możesz użyć funkcji MIN. Jak się zapewne domyślasz, w podobny sposób możesz korzystać z innych funkcji. Przykładowo do określenia k-tej największej liczby w wybranym zakresie możesz użyć funkcji LARGE (odpowiednik funkcji arkusza MAX.K): Przedstawiam odpowiednie wyrażenie.

```
SecondHighest = WorksheetFunction.LARGE(Range("A:A"), 2)
```

Zauważ, że funkcja LARGE przyjmuje dwa argumenty. Drugi argument oznacza k-tą część — w tym przypadku drugą (drugą wartość, co do wielkości).

Obliczanie raty pożyczki

W następnym przykładzie wykorzystam funkcję PMT do obliczenia raty pożyczki. W programie użyłem trzech zmiennych do przechowywania danych przekazywanych jako argumenty funkcji PMT. Obliczona rata jest wyświetlana w oknie komunikatu.

```
Sub PmtCalc()
    Dim IntRate As Double
    Dim LoanAmt As Double
    Dim Periods As Integer
    IntRate = 0.0825 / 12
    Periods = 30 * 12
    LoanAmt = 150000
    MsgBox WorksheetFunction.PMT(IntRate, Periods, -LoanAmt)
End Sub
```

Możesz również bezpośrednio umieścić wartości jako argumenty funkcji, co pokazuję w następującym przykładzie.

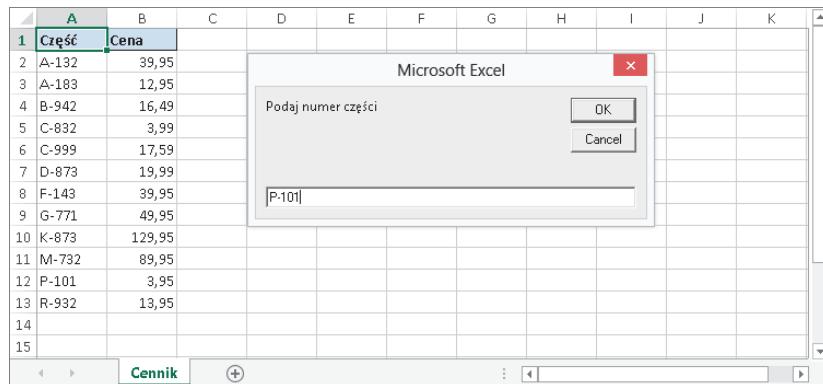
```
MsgBox WorksheetFunction.PMT(0.0825 / 12, 360, -150000)
```

Pamiętaj jednak, że użycie zmiennych do przechowywania danych sprawia, iż kod jest czytelniejszy i łatwiej go modyfikować, jeśli trzeba.

Użycie funkcji wyszukiwania

W poniższym przykładzie wykorzystuję funkcje VBA InputBox oraz MsgBox, a także funkcję VLOOKUP (odpowiednik WYSZUKAJ.PIONOWO). Kod pyta o numer części i zwraca jej cenę odszukaną w tabeli. Na rysunku 9.4 zakres A1:B13 nosi nazwę PriceList.

```
Sub GetPrice()
    Dim PartNum As Variant
    Dim Price As Double
    PartNum = InputBox("Podaj numer części")
    Sheets("Cennik").Activate
    Price = WorksheetFunction.VLOOKUP(PartNum, Range("PriceList"), 2, False)
    MsgBox PartNum & " kosztuje " & Price
End Sub
```



The screenshot shows a Microsoft Excel spreadsheet titled 'Cennik'. It contains a table with columns 'Część' (Part) and 'Cena' (Price). The data is as follows:

A	B
Część	Cena
A-132	39,95
A-183	12,95
B-942	16,49
C-832	3,99
C-999	17,59
D-873	19,99
F-143	39,95
G-771	49,95
K-873	129,95
M-732	89,95
P-101	3,95
R-932	13,95
14	
15	

An input dialog box titled 'Microsoft Excel' is displayed, asking 'Podaj numer części' (Enter part number). The value 'P-101' is entered in the input field. The 'OK' button is visible at the bottom right of the dialog.

Rysunek 9.4.
Zakres o nazwie PriceList zawiera ceny części

Ten skoroszyt możesz pobrać ze strony internetowej książki.



Procedura wykonuje następujące czynności.

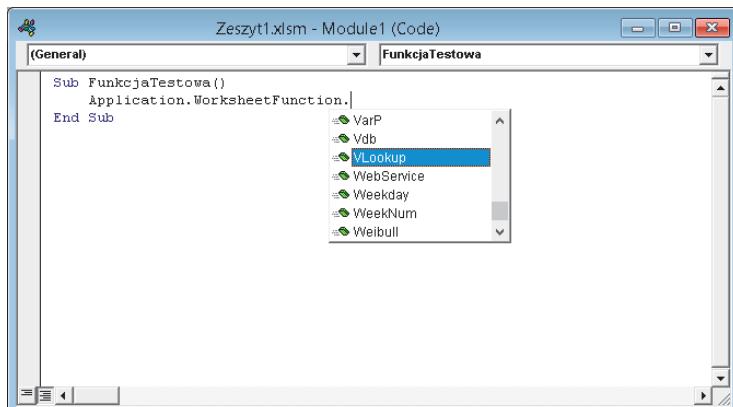
1. **Funkcja VBA InputBox prosi użytkownika o wprowadzenie numeru części.**
 2. **Instrukcja przypisuje zmiennej PartNum numer wprowadzony przez użytkownika.**
 3. **Następna instrukcja aktywuje arkusz *Cennik*, gdyby przypadkowo nie był aktywnym arkuszem w skoroszycie.**
 4. **Kod wykorzystuje funkcję VLOOKUP do wyszukania w tabeli numeru części.**
- Zauważ, że użyte w tym przykładzie argumenty funkcji są takie same jak w funkcji zastosowanej w formule arkusza. Instrukcja w tym wierszu przypisuje wynik funkcji zmiennej Price.
5. **Kod wyświetla cenę części za pomocą funkcji MsgBox.**

Procedura nie zawiera żadnej obsługi błędów i marnie kończy swoje działanie, jeżeli wpiszesz nieistniejący numer części (sprawdź to!). Jeżeli ma to być prawdziwa, używana w praktyce aplikacja, powinieneś dodać kilka instrukcji elegancko obsługujących błędy. Obsługę błędów omówię w rozdziale 12.

Wprowadzanie funkcji arkusza kalkulacyjnego

Do wstawienia funkcji arkusza kalkulacyjnego w module VBA nie możesz użyć okna dialogowego *Wstawianie funkcji*. Zamiast tego skorzystaj ze starego sposobu, czyli wpisz ręcznie jej nazwę. Możesz natomiast użyć okna *Wstawianie funkcji* do odszukania funkcji, którą chcesz zastosować, i uzyskania informacji o jej argumentach.

Możesz również wykorzystać zalety opcji edytora VBE *Auto List Members*, dzięki której w liście rozwijanej wyświetlana jest lista wszystkich funkcji arkusza. Wpisz po prostu Application.WorksheetFunction oraz kropkę. Zobaczysz listę funkcji, których możesz użyć (zobacz rysunek 9.5). Jeżeli ta funkcjonalność nie jest dostępna, wybierz polecenie menu edytora *Tools/Options*, kliknij zakładkę *Editor* i zaznacz opcję *Auto List Members*.



Rysunek 9.5.
Otwarcie listy funkcji arkusza kalkulacyjnego, których można użyć w kodzie VBA

Więcej o użyciu funkcji arkusza kalkulacyjnego

Początkujący programiści często mylą wbudowane funkcje VBA z funkcjami arkusza kalkulacyjnego. Warto pamiętać o zasadzie, że język VBA nie jest po to, aby ponownie wynaleźć koło. W większości przypadków VBA nie duplikuje funkcji arkusza.

Zamiast większości funkcji arkusza, które nie są dostępne jako metody obiektu `WorksheetFunction`, możesz użyć odpowiednika w postaci operatora lub funkcji VBA. Przykładowo funkcja arkusza `MOD` nie jest dostępna w obiekcie `WorksheetFunction`, ponieważ VBA zawiera jej odpowiednik, czyli operator `Mod`.



Jaki z tego wniosek? Jeżeli chcesz użyć jakiejś funkcji, najpierw sprawdź, czy VBA zawiera to, czego potrzebujesz. Jeżeli nie, sprawdź funkcje arkusza kalkulacyjnego. Kiedy i to nie pomoże, możesz napisać swoją własną funkcję w języku VBA.

Użycie własnych funkcji

Opisałem funkcje VBA i arkusza kalkulacyjnego programu Excel. Trzecią grupą funkcji, które możesz wykorzystać w kodzie VBA, są własne funkcje. *Własna funkcja* (zwana również funkcją zdefiniowaną przez użytkownika) jest tworzona przez Ciebie w języku (a jakże!) VBA. Aby zastosować własną funkcję, musisz ją zdefiniować w tym skoroszycie, w którym jej użyjesz, albo zdefiniować ją w tak zwanym dodatku (zobacz rozdział 21.).

Poniżej przedstawiam przykład definicji prostej funkcji (`Function`) oraz użycia jej w procedurze `Sub`.

```
Function MultiplyTwo(num1, num2) As Double
    MultiplyTwo = num1 * num2
End Function

Sub ShowResult()
    Dim n1 As Double, n2 As Double
    Dim Result As Double
    n1 = 123
    n2 = 544
    Result = MultiplyTwo(n1, n2)
    MsgBox Result
End Sub
```

Własna funkcja `MultiplyTwo` ma dwa argumenty. Procedura `ShowResult` wykorzystuje tę funkcję i przekazuje jej dwa argumenty (w nawiasach). Następnie wyświetla okno z komunikatem prezentującym wynik zwrócony przez funkcję `MultiplyTwo`.

Nie muszę chyba dodawać, że funkcja `MultiplyTwo` jest zupełnie bezużyteczna. W procedurze `ShowResult` o wiele lepiej zastosować mnożenie. Umieściłem ten przykład, aby dać Ci wyobrażenie, jak procedura `Sub` może wykorzystywać własną funkcję.

152 Część III: Podstawy programowania

Własne funkcje możesz również stosować w formułach arkusza. Jeżeli w skoroszycie jest zdefiniowana funkcja `MultiplyTwo`, możesz utworzyć na przykład poniższą formułę.

```
=MultiplyTwo(A1:A2)
```

Formuła ta zwraca iloczyn wartości w komórkach A1 i A2.

Tworzenie własnych funkcji jest ważną (i bardzo przydatną) rzeczą. Tak ważną (i przydatną), że poświęciłem jej osobny rozdział. W rozdziale 20. znajdziesz naprawdę użyteczne przykłady.

Rozdział 10

Sterowanie przepływem i podejmowanie decyzji

W tym rozdziale:

- ▶ poznasz sposoby sterowania przepływem w kodzie VBA,
- ▶ dowiesz się o okropnej instrukcji GoTo,
- ▶ zastosujesz struktury If-Then oraz Select Case,
- ▶ wykonasz pętle w swoich procedurach.

Niektóre procedury VBA rozpoczynają swoje działanie na początku kodu i wykonują linię po linii, aż do końca, nie zbaczając ze ścieżki przebiegającej z góry na dół programu. W ten sposób zawsze działają rejestrowane przez Ciebie makra. Często jednak trzeba sterować przepływem kodu, omijać niektóre instrukcje, wielokrotnie powtarzać inne i sprawdzać warunki w celu zadecydowania, co procedura ma dalej robić. Bierz zatem byka za rogi, ponieważ za chwilę odkryjesz istotę programowania.

Zabierz się za przepływ, kolego

Niektórzy nowicjusze w programowaniu nie rozumieją, w jaki sposób nierożumny komputer może podejmować inteligentne decyzje. Tajemnica leży w kilku konstrukcjach programistycznych, obsługiwanych przez większość języków programowania. W tabeli 10.1 zamieściłem krótkie zestawienie tych konstrukcji (wszystkie objaśnię dalej w tym rozdziale).

Tabela 10.1. Konstrukcje programistyczne służące do podejmowania decyzji

Konstrukcja	Jej działanie
Instrukcja GoTo	Przeskakuje do określonej instrukcji w kodzie programu.
Struktura If-Then	Wykonuje coś, jeżeli coś innego jest prawdą.
Struktura Select Case	Wykonuje różne rzeczy, w zależności od wartości czegoś.
Pętla For-Next	Wykonuje serię instrukcji zadaną liczbę razy.
Pętla Do-While	Wykonuje coś dopóty, dopóki coś innego jest prawdą.
Pętla Do-Until	Wykonuje coś dotąd, aż coś innego stanie się prawdą.

Instrukcja GoTo

Użycie instrukcji GoTo jest najprostszym sposobem zmiany przepływu programu. Po prostu przenosi ona wykonywanie kodu do innej instrukcji, poprzedzonej etykietą.

Twój kod VBA może zawierać dowolną liczbę etykiet. *Etykietą* jest ciągiem znaków zakończonym dwukropkiem.

W poniższym przykładzie pokazuję, jak działa instrukcja GoTo.

```
Sub CheckUser()
    UserName = InputBox("Podaj swoje imię: ")
    If UserName <> "Steve Ballmer" Then GoTo WrongName
    MsgBox ("Cześć, Steve... ")
    ' ...[Tu więcej kodu] ...
    Exit Sub
WrongName:
    MsgBox "Niestety, tylko Steve Ballmer może uruchomić ten program."
End Sub
```

Powyższa procedura wykorzystuje funkcję InputBox do pobrania imienia użytkownika. Następnie podejmuje decyzję: jeżeli użytkownik wprowadzi inne imię i nazwisko niż Steve Ballmer (były szef firmy Microsoft), wówczas przepływ programu przeskakuje do etykiety WrongName, po czym wyświetlony zostanie komunikat z przeprosinami i procedura zakończy działanie. Jeśli natomiast pan Ballmer uruchomili makro i wprowadzi swoje prawdziwe dane, procedura wyświetli komunikat powitalny i wykona dodatkowy kod (niepokazany w tym przykładzie).

Zwróć uwagę, że instrukcja Exit Sub kończy działanie procedury, zanim druga funkcja MsgBox dostanie szansę na wywołanie. Bez instrukcji Exit Sub wywołane byłyby obie funkcje MsgBox.

Czym jest programowanie strukturalne? Czy jest ważne?

Przebywając z programistami, przedzej czy później usłyszysz termin *programowanie strukturalne*. To pojęcie istnieje już od dziesięcioleci i programiści generalnie są zgodni, że programy strukturalne mają przewagę nad niestrukturalnymi. Czym więc jest programowanie strukturalne? Czy możesz je stosować w VBA?

Podstawą programowania strukturalnego jest założenie, że procedura lub fragment kodu powinny mieć jeden punkt wejścia i jeden punkt wyjścia. Innymi słowy, blok kodu powinien być samodzielną jednostką. Program nie może wskoczyć w środek takiej jednostki, ani nie może z niej wyjść w żadnym innym punkcie niż wyjściowy. W kodzie strukturalnym program jest wykonywany w uporządkowany, łatwy do zrozumienia sposób, w odróżnieniu od programu niestrukturalnego,

który chaotycznie skacze z jednego punktu do innego. Z tym założeniem kłoci się użycie instrukcji GoTo.

Ogólnie, program strukturalny jest bardziej czytelny i prostszy w zrozumieniu. Co ważne, jest również łatwiejszy w modyfikacji, jeżeli będzie potrzebna.

Język VBA jest językiem w pełni strukturalnym. Oferuje standardowe konstrukcje strukturalne, takie jak If-Then-Else, pętle For-Next, pętle Do-Until, pętle Do-While oraz struktury Select Case. Co więcej, obsługuje konstrukcje modułów kodu. Jeżeli jesteś początkującym programistą, powinieneś jak najszybciej przyswoić sobie dobre zwyczaje programowania strukturalnego. Koniec dyskusji.

Powyższy prosty kod działa prawidłowo, ale VBA oferuje kilka innych (i bardziej strukturalnych) wariantów dla instrukcji GoTo. Zasadniczo, powinieneś stosować tę instrukcję tylko wtedy, gdy nie ma innych sposobów na wykonanie określonej czynności. W praktyce jedynym przypadkiem, w którym musisz użyć instrukcji GoTo, jest obsługa błędów, którą opisuję w rozdziale 12.

Na marginesie, procedurę CheckUser przedstawiłem w celu zademonstrowania użycia instrukcji GoTo. Nie było moim zamiarem pokazanie efektywnej i bezpiecznej techniki programowania!



Wielu zatwardziałych programistów żywi głęboko zakorzenioną nienawiść do instrukcji GoTo, ponieważ jej użycie prowadzi do powstania trudnego w czytaniu i obsłudze „kodu spaghetti”. Dlatego rozmawiając z innymi programistami, nigdy nie wspominaj, że używasz instrukcji GoTo.

Decyzje, decyzje

Tak jak w wielu dziedzinach życia, w programie Excel podejmowanie właściwych decyzji jest kluczem do sukcesu w tworzeniu makr. Jeżeli ta książka przyniesie zamierzony skutek, wkrótce powinieneś przyjąć moją filozofię, że dobra aplikacja opiera się na podejmowaniu decyzji i działaniu według nich.

W tym podrozdziale omówię dwie struktury programistyczne, które wzbogacą Twoje procedury VBA o kilka imponujących funkcjonalności decyzyjnych; poznasz If-Then oraz Select Case.

Struktura If-Then

No dobrze, przyznam się i powiem wprost, że If-Then jest najważniejszą strukturą sterującą w VBA. Prawdopodobnie będziesz ją stosował codziennie (przynajmniej ja tak robię).

Użyj struktury If-Then wtedy, gdy chcesz warunkowo wykonać jedną lub więcej instrukcji. Opcjonalna klauzula Else, jeżeli zostanie użyta, pozwala wykonać jedną lub wiele instrukcji, gdy sprawdzany warunek nie jest prawdziwy. Poniżej przedstawiona jest prosta, opisana wcześniej, procedura CheckUser zmieniona tak, aby wykorzystywała strukturę If-Then.

```
Sub CheckUser2()
    UserName = InputBox("Podaj swoje imię: ")
    If UserName = "Steve Ballmer" Then
        MsgBox ("Cześć, Steve... ")
        ' ...[Tu więcej kodu] ...
    Else
        MsgBox "Niestety, tylko Steve Ballmer może uruchomić ten program."
    End If
End Sub
```



Zapewne zgodzisz się, że ta wersja jest znacznie łatwiejsza do zrozumienia.

Skoroszyt zawierający przykłady z tej części rozdziału jest dostępny na stronie internetowej książki.

Przykłady struktury If-Then

W poniższych przykładach przedstawiam strukturę If-Then bez opcjonalnej klauzuli Else.

```
Sub GreetMe()
    If Time < 0.5 Then MsgBox "Jaki piękny poranek"
End Sub
```

Procedura GreetMe wykorzystuje funkcję VBA Time do pobrania czasu systemowego. Jeżeli bieżący czas jest mniejszy niż 0,5 (to znaczy jest przed południem), procedura wyświetla uprzejme powitanie. Jeżeli wartość funkcji Time jest większa lub równa 0,5, kod kończy działanie i nic się nie dzieje.

Aby wyświetlić inne pozdrowienie, gdy wartość funkcji Time jest większa lub równa 0,5, możesz poniżej struktury If-Then użyć poniższej.

```
Sub GreetMe2()
    If Time < 0.5 Then MsgBox "Jaki piękny poranek"
    If Time >= 0.5 Then MsgBox "Jaki piękny dzień"
End Sub
```

Zauważ, że w drugiej instrukcji If-Then użyłem operatora `>=` (większe lub równe). Dzięki temu mamy pewność, że warunkami będzie objęty cały dzień. Gdybym użył operatora `>` (większe), a procedura została uruchomiona dokładnie o 12:00, nie pojawiłby się żaden komunikat. Taka sytuacja jest mało prawdopodobna, ale w przypadku tak ważnego programu jak ten, nie możemy pozwolić sobie na najmniejsze ryzyko.

Przykład struktury If-Then-Else

Innym sposobem rozwiązywania opisywanego zagadnienia jest użycie klauzuli Else. Poniżej znajdziesz ten sam kod, odpowiednio zmieniony i wykorzystujący strukturę If-Then-Else.

```
Sub GreetMe3()
    If Time < 0.5 Then MsgBox "Jaki piękny poranek" Else _
        MsgBox "Jaki piękny dzień"
End Sub
```

Zwróć uwagę, że w powyższym przykładzie użyłem znaku kontynuacji wiersza (podkreślenie). Instrukcja If-Then-Else jest w rzeczywistości pojedynczą instrukcją. VBA umożliwia również nieco inny sposób kodowania struktury If-Then-Else, mianowicie z zastosowaniem instrukcji End If. W takim przypadku procedura GreetMe może być napisana w następujący sposób.

```
Sub GreetMe4()
    If Time < 0.5 Then
        MsgBox "Jaki piękny poranek"
    Else
```

```
    MsgBox "Jaki piękny dzień"  
End If  
End Sub
```

W praktyce możesz umieścić dowolną liczbę instrukcji zarówno w częściach `If`, jak i `Else`. Preferuję taką właśnie składnię, ponieważ jest bardziej czytelna, a instrukcje są krótsze.

A jak rozbudować procedurę `GreetMe`, aby obsługiwała trzy warunki: poranek, środek dnia i wieczór? Masz dwie możliwości: użycie trzech instrukcji `If-Then` lub *zagnieżdzonej* struktury `If-Then-Else`. *Zagnieżdżenie* oznacza umieszczenie jednej struktury `If-Then-Else` wewnętrznie do drugiej. Pierwszy sposób, z użyciem trzech instrukcji `If-Then`, jest najprostszy.

```
Sub GreetMe5()  
    Dim Msg As String  
    If Time < 0.5 Then Msg = "poranek"  
    If Time >= 0.5 And Time < 0.75 Then Msg = "dzień"  
    If Time >= 0.75 Then Msg = "wieczór"  
    MsgBox "Jaki piękny " & Msg  
End Sub
```

Wprowadziłem zamieszanie, używając zmiennej. Zmienna `Msg` przyjmuje różne wartości w zależności od pory dnia. Instrukcja `MsgBox` wyświetla powitanie „Jaki piękny poranek”, „Jaki piękny dzień” lub „Jaki piękny wieczór”.

Poniższy kod wykonuje te same czynności, ale wykorzystuje strukturę `If-Then-Else`.

```
Sub GreetMe6()  
    Dim Msg As String  
    If Time < 0.5 Then  
        Msg = "poranek"  
    End If  
    If Time >= 0.5 And Time < 0.75 Then  
        Msg = "dzień"  
    End If  
    If Time >= 0.75 Then  
        Msg = "wieczór"  
    End If  
    MsgBox "Jaki piękny " & Msg  
End Sub
```

Użycie instrukcji Elseif

W poprzednich przykładach sprawdzane były wszystkie warunki — nawet o poranku. Nieco bardziej efektywna struktura powinna powodować wyjście z kodu, gdy tylko pierwszy warunek będzie spełniony. Przykładowo o poranku procedura powinna wyświetlić komunikat „Jaki piękny poranek” i zakończyć działanie, bez niepotrzebnego sprawdzania innych warunków.

W takich małych procedurach jak ta nie musisz się troszczyć o prędkość działania. Kiedy tworzysz większe aplikacje, w których prędkość będzie kwestią krytyczną, powinieneś pamiętać o innej składni struktury `If-Then`.

Poniżej pokazuję, jak zmienić procedurę GreetMe, aby wykorzystywała tę składnię.

```
Sub GreetMe7()
    Dim Msg As String
    If Time < 0.5 Then
        Msg = "poranek"
    ElseIf Time >= 0.5 And Time < 0.75 Then
        Msg = "dzień"
    Else
        Msg = "wieczór"
    End If
    MsgBox "Jaki piękny " & Msg
End Sub
```

Jeżeli warunek jest spełniony, VBA wykonuje odpowiednie instrukcje i struktura If kończy działanie. Innymi słowy, VBA nie traci czasu na sprawdzanie dodatkowych warunków, dzięki czemu procedura jest trochę bardziej wydajna niż w poprzednich przykładach. Ale jest to kompromis (zawsze są kompromisy), ponieważ kod jest trudniejszy do zrozumienia (ale — oczywiście — o tym już wiesz).

Inny przykład struktury If-Then

Poniżej przedstawiam inny przykład wykorzystujący strukturę If-Then. Procedura prosi użytkownika o podanie liczby i w zależności od wprowadzonej wartości wyświetla odpowiedni rabat.

```
Sub ShowDiscount()
    Dim Quantity As Integer
    Dim Discount As Double
    Quantity = InputBox("Podaj liczbę: ")
    If Quantity > 0 Then Discount = 0.1
    If Quantity >= 25 Then Discount = 0.15
    If Quantity >= 50 Then Discount = 0.2
    If Quantity >= 75 Then Discount = 0.25
    MsgBox "Rabat: " & Discount
End Sub
```

Zwróć uwagę, że w powyższym kodzie wykonywana jest każda instrukcja If-Then, a w każdej z nich jest modyfikowana zmenna Discount. Niemniej jednak ostatecznie procedura wyświetla właściwą wartość zmiennej Discount, ponieważ instrukcje If-Then umieściłem w rosnącej kolejności rabatów.

Następna procedura wykonuje to samo zadanie, ale wykorzystuje alternatywną składnię ElseIf. W tym przypadku procedura kończy działanie natychmiast po wykonaniu instrukcji zawierającej spełniony warunek.

```
Sub ShowDiscount2()
    Dim Quantity As Integer
    Dim Discount As Double
    Quantity = InputBox("Podaj liczbę: ")
    If Quantity > 0 And Quantity < 25 Then
        Discount = 0.1
    ElseIf Quantity >= 25 And Quantity < 50 Then
```

```
Discount = 0.15
ElseIf Quantity >= 50 And Quantity < 75 Then
    Discount = 0.2
ElseIf Quantity >= 75 Then
    Discount = 0.25
End If
MsgBox "Rabat: " & Discount
End Sub
```

Uważam, że wielokrotne użycie instrukcji If-Then jest raczej niewygodne. Strukturę If-Then stosuję zazwyczaj do podejmowania prostych binarnych decyzji. Jeżeli decyzja obejmuje trzy lub więcej przypadków, bardziej efektywne jest użycie struktury Select Case.

Struktura Select Case

Struktura Select Case jest przydatna podczas podejmowania decyzji obejmujących trzy przypadki lub więcej (ale sprawdza się również w przypadku dwóch i jest alternatywą dla struktury If-Then-Else).



Przykłady do tej części rozdziału są dostępne na stronie internetowej książki.

Przykład struktury Select Case

W poniższym przykładzie pokazuję zastosowanie struktury Select Case. Przedstawiam również inny sposób kodowania wcześniejszych przykładów z tego rozdziału.

```
Sub ShowDiscount3()
    Dim Quantity As Integer
    Dim Discount As Double
    Quantity = InputBox("Podaj liczbę: ")
    Select Case Quantity
        Case 0 To 24
            Discount = 0.1
        Case 25 To 49
            Discount = 0.15
        Case 50 To 74
            Discount = 0.2
        Case Is >= 75
            Discount = 0.25
    End Select
    MsgBox "Rabat: " & Discount
End Sub
```

W tym przykładzie sprawdzana jest wartość zmiennej `Quantity`. Kod rozróżnia cztery różne przypadki (0 – 24, 25 – 49, 50 – 74 oraz 75 i więcej).

Poniżej instrukcji Case można umieścić dowolną liczbę innych instrukcji, które będą wykonywane po spełnieniu danego warunku. Jeżeli użyjesz tylko jednej instrukcji, tak jak w powyższym przykładzie, możesz ją umieścić w jednym wierszu razem ze słowem Case, poprzedzoną dwukropkiem, czyli separatorem instrukcji VBA. Dzięki temu — moim zdaniem — kod jest bardziej zwarty i trochę czytelniejszy. Poniżej przedstawiam kod wykorzystujący ten format.

```
Sub ShowDiscount4 ()  
    Dim Quantity As Integer  
    Dim Discount As Double  
    Quantity = InputBox("Podaj liczbę: ")  
    Select Case Quantity  
        Case 0 To 24: Discount = 0.1  
        Case 25 To 49: Discount = 0.15  
        Case 50 To 74: Discount = 0.2  
        Case Is >= 75: Discount = 0.25  
    End Select  
    MsgBox "Rabat: " & Discount  
End Sub
```

Gdy VBA wykonuje strukturę `Select Case`, wyjście z niej następuje zaraz po znalezieniu spełnionego warunku i wykonaniu instrukcji dla tego przypadku.

Przykład zagnieżdżonej struktury `Select Case`

Jak pokazuję w następnym przykładzie, możesz zagnieździć struktury `Select Case`. Poniższy kod sprawdza bieżącą komórkę i wyświetla komunikat opisujący jej zawartość. Zauważ, że procedura zawiera trzy struktury `Select Case`, a każda z nich ma swoją własną instrukcję `End Select`.

```
Sub CheckCell()  
    Dim Msg As String  
    Select Case IsEmpty(ActiveCell)  
        Case True  
            Msg = "jest pusta."  
        Case Else  
            Select Case ActiveCell.HasFormula  
                Case True  
                    Msg = "zawiera formułę"  
                Case Else  
                    Select Case IsNumeric(ActiveCell)  
                        Case True  
                            Msg = "zawiera liczbę"  
                        Case Else  
                            Msg = "zawiera tekst"  
                    End Select  
                End Select  
            End Select  
        End Select  
    End Select  
    MsgBox "Komórka " & ActiveCell.Address & " " & Msg  
End Sub
```

Oto logika powyższego kodu.

- 1. Sprawdź, czy komórka jest pusta.**
- 2. Jeżeli komórka nie jest pusta, sprawdź, czy zawiera formułę.**
- 3. Jeżeli komórka nie zawiera formuły, sprawdź, czy zawiera wartość liczbową, czy tekstową.**

Po zakończeniu kodu zmienna `Msg` zawiera ciąg znaków opisujący zawartość komórki. Ten komunikat wyświetla funkcja `MsgBox`, co pokazuję na rysunku 10.1.

The screenshot shows a Microsoft Excel spreadsheet with a table of months and their values. A message box titled "Microsoft Excel" is displayed, stating "Komórka \$B\$13 zawiera formułę". The table has columns A and B, with rows from 1 to 13. Row 13 contains the formula `=SUM(A1:A12)`. The message box has an "OK" button. The Excel ribbon shows "Arkusz2" is selected.

A	B	C	D	E	F	G	H	I	J
1	Styczeń	198							
2	Luty	190							
3	Marczec	194							
4	Kwiecień	184							
5	Maj	192							
6	Czerwiec	186							
7	Lipiec	184							
8	Sierpień	175							
9	Wrzesień	185							
10	Październik	193							
11	Listopad								
12	Grudzień								
13	Suma:	1881							
14									
15									
16									
17									
18									
19									

Rysunek 10.1.
Komunikat wy-
świetlony przez
procedurę
CheckCell

Struktury `Select Case` możesz zagnieździć dowolnie głęboko, musisz tylko się upewnić, że każda instrukcja `Select Case` ma odpowiadającą jej instrukcję `End Select`.



Jeżeli wciąż nie jesteś przekonany, że wcięcia w kodzie są warte zachodu, spójrz na poprzedni listing, który jest dobrym przykładem. Wcięcia naprawdę sprawiają, że zagnieżdżone poziomy kodu są bardziej czytelne (przynajmniej ja tak uważam). Jeżeli mi nie wierzysz, spójrz na tę samą procedurę napisaną bez żadnych wcięć.

```
Sub CheckCell()
    Dim Msg As String
    Select Case IsEmpty(ActiveCell)
        Case True
            Msg = "jest pusta."
        Case Else
            Select Case ActiveCell.HasFormula
                Case True
                    Msg = "zawiera formułę"
                Case Else
                    Select Case IsNumeric(ActiveCell)
                        Case True
                            Msg = "zawiera liczbę"
                        Case Else
                            Msg = "zawiera tekst"
                        End Select
                    End Select
                End Select
            End Sub
    End Select
    MsgBox "Komórka " & ActiveCell.Address & " " & Msg
End Sub
```

Zupełnie niezrozumiałe, prawda?

Entliczek, pętliczek — czyli jak używać pętli?

Termin *pętla* oznacza wykonywanie bloku instrukcji kodu VBA określona liczbę razy. Po co stosować pętle? Moim zdaniem, z kilku powodów. Twój kod może:

- ✓ przeglądać zakres komórek, przetwarzając każdą z nich z osobna,
- ✓ przeglądać wszystkie otwarte skoroszyty (kolekcja Workbooks) i z każdym z nich coś robić,
- ✓ przeglądać wszystkie arkusze w skoroszycie (kolekcja Worksheets) i z każdym z nich coś robić,
- ✓ przeglądać wszystkie elementy w tabeli,
- ✓ przeglądać wszystkie znaki w komórce,
- ✓ przeglądać wszystkie wykresy w arkuszu (kolekcja ChartObjects) i z każdym z nich coś robić,
- ✓ przeglądać inne rzeczy, które mi nie przyszły do głowy.

Język VBA obsługuje różne typy pętli, a w poniższych przykładach pokazuję kilka sposobów ich użycia.

Pętle For-Next

Przykłady do tej części rozdziału są dostępne na stronie internetowej książki.



Najprostszym rodzajem pętli jest For-Next. Obiegi pętli są kontrolowane za pomocą zmiennej licznikowej, która zmienia swoją wartość, począwszy od określonej liczby i skończywszy na innej. Pomiędzy instrukcjami For i Next powtarzane jest wykonywanie instrukcji w pętli. Czytaj dalej, aby się dowiedzieć, jak to działa.

Przykład pętli For-Next

W poniższym przykładzie wykorzystuję pętlę For-Next do zsumowania pierwszego 1000 dodatnich liczb całkowitych. Zmienna Total przyjmuje na początku wartość zero. Następnie wykonywana jest pętla. Zmienna Cnt jest licznikiem pętli. Zaczyna od wartości 1 i jest zwiększana o 1 w każdym obiegu pętli. Pętla kończy działanie, gdy zmieniła Cnt osiągnie wartość 1000.

W tym przykładzie mamy tylko jedną instrukcję wewnątrz pętli. Instrukcja dodaje wartość zmiennej Cnt do zmiennej Total. Gdy pętla zakończy się, funkcja MsgBox wyświetla zsumowane liczby.

```
Sub AddNumbers()
    Dim Total As Double
    Dim Cnt As Integer
```

```
Total = 0
For Cnt = 1 To 1000
    Total = Total + Cnt
Next Cnt
MsgBox Total
End Sub
```



Ponieważ licznik pętli jest zwykłą zmienną, więc pomiędzy instrukcjami For i Next możesz napisać kod, który zmienia jego wartość. Jest to jednak *bardzo zła* praktyka. Zmiana wartości licznika wewnętrz pętli może prowadzić do nieprzewidzianych skutków. Zwróć szczególną uwagę, aby Twój kod nie zmieniał wartości licznika pętli.

Przykład pętli For-Next z instrukcją Step

W pętli For-Next możesz użyć wartości Step po to, aby licznik przeskakiwał określone wartości. Poniżej jest pokazany przykład sumujący tylko nieparzyste liczby całkowite w przedziale od 1 do 1000.

```
Sub AddOddNumbers()
    Dim Total As Double
    Dim Cnt As Integer
    Total = 0
    For Cnt = 1 To 1000 Step 2
        Total = Total + Cnt
    Next Cnt
    MsgBox Total
End Sub
```

Tym razem zmienna Cnt startuje od jedności i przybiera wartości 3, 5, 7 i tak dalej. Wartość Step określa, o jaką wartość zwiększany jest licznik. Zauważ, że górna wartość (1000) w rzeczywistości nie jest użyta, ponieważ największą wartością przyjmowaną przez licznik jest 999.

Niżej przedstawiam inny przykład wykorzystujący wartość Step równą 3. Procedura działa w aktywnym arkuszu i ustawia jasnoszare tło w co trzecim wierszu, w wierszach o numerach od 1 do 100.

```
Sub ShadeEveryThirdRow()
    Dim i As Long
    For i = 1 To 100 Step 3
        Rows(i).Interior.Color = RGB(200, 200, 200)
    Next i
End Sub
```

Na rysunku 10.2 przedstawiam wynik działania tego makra.

Przykład pętli For-Next z instrukcją Exit For

Pętla For-Next może również zawierać jedną lub więcej instrukcji Exit For. Gdy VBA napotka taką instrukcję, pętla natychmiast kończy działanie.

A	B	C	D	E	F	G	H	I	J	K
1	31	33	78	57	63	64	24	97	25	30
2	13	40	13	86	68	19	54	53	43	23
3	32	5	9	41	41	8	52	79	14	78
4	93	71	93	63	83	58	8	47	9	72
5	31	78	81	37	22	93	19	57	3	24
6	47	6	5	44	41	48	62	19	2	61
7	29	61	82	83	64	96	84	84	66	29
8	91	49	28	43	86	68	81	99	95	92
9	96	19	96	58	59	52	12	90	75	64
10	94	9	33	85	99	37	82	87	8	45
11	97	78	9	13	96	79	89	54	67	5
12	8	7	57	10	70	10	30	75	80	76
13	25	44	39	79	59	44	79	82	67	24
14	26	71	98	75	53	51	74	8	40	37
15	59	92	56	34	29	54	72	15	58	22
16	81	64	85	5	4	18	70	56	33	83
17	2	80	93	46	49	12	12	37	8	80
18	65	69	79	9	57	100	96	66	86	25
19	71	26	32	41	57	14	86	45	33	90
20	32	36	26	75	47	74	31	63	49	87

Rysunek 10.2.
Zastosowanie
pętli do usta-
wienia szarego
tła wierszy

W kolejnym przykładzie, dostępnym na stronie internetowej książki, przedstawiam użycie instrukcji Exit For. Jest to funkcja gotowa do zastosowania w formule arkusza. Funkcja przyjmuje jeden argument (zmienną o nazwie Str) i zwraca jego początkowe znaki poprzedzające cyfrę. Jeśli na przykład argument ma wartość „KBR98Z”, funkcja zwróci ciąg „KBR”.

```
Function TextPart(Str)
    TextPart = ""
    For i = 1 To Len(Str)
        If IsNumeric(Mid(Str, i, 1)) Then
            Exit For
        Else
            TextPart = TextPart & Mid(Str, i, 1)
        End If
    Next i
End Function
```

Pętla For-Next startuje od wartości 1 i kończy na liczbie oznaczającej liczbę znaków w ciągu. W kodzie użyta jest funkcja Mid zwracająca w pętli pojedynczy znak. Jeżeli jest to cyfra, wykonywana jest instrukcja Exit For i pętla kończy działanie wcześniej. Jeżeli znak nie jest cyfrą, wówczas jest dopisywany do zwracanej wartości (o tej samej nazwie, co funkcja). Pętla sprawdza wszystkie znaki w ciągu tylko w jednym przypadku, gdy ciąg przekazany w argumencie funkcji nie zawiera cyfr.

Móglbyś teraz głośno zaprotestować: „Zaraz, powiedziałeś przecież, że zawsze powinien być jeden punkt wyjścia!”. Tak, masz rację, to oznacza, że zaczynasz czuć, o co chodzi w programowaniu strukturalnym. Ale w niektórych sytuacjach ominięcie tej zasady jest słuszącą decyzją. W tym przypadku znacznie przyspiesza wykonywanie kodu, ponieważ nie ma powodu, aby kontynuować pętlę po znalezieniu pierwszej cyfry.

Jak szybkie są pętle?

Zapewne ciekaw jesteś, jak szybko VBA wykonuje pętle For-Next. Czy niektóre systemy wykonują kod znacznie szybciej niż inne? Kilka lat temu umieściłem na swojej stronie internetowej poniższą procedurę VBA i poprosiłem użytkowników o podanie swoich wyników.

```
Sub TimeTest()
    '100 milionów liczb losowych, porównać
    'i działań matematycznych
    Dim x As Long
    Dim StartTime As Single
    Dim i As Long
    x = 0
    StartTime = Timer
    For i = 1 To 100000000
        If Rnd <= 0.5 Then x = x + 1 Else x = x - 1
    Next i
    MsgBox Timer - StartTime & " sekund"
End Sub
```

Kod wykonuje pętlę 100 milionów razy i dodatkowo wykonuje w niej jakieś czynności. Generuje liczbę losową, wykonuje sprawdzenie If-Then i działanie

matematyczne. Po wykonaniu pętli wyświetlany jest komunikat z czasem, jaki upłynął podczas wykonywania kodu. Mój system wykonał go w ciągu 8,03 sekundy. Około 150 osób podało wyniki w przedziale od 2 do 30 sekund. Innymi słowy, niektóre komputery są 30 razy szybsze od innych. Warto wiedzieć.

Skoroszyt zawierający ten kod możesz pobrać ze strony internetowej książki.

Ale prawdziwe pytanie brzmi: „Ile czasu zajmie mi ręczne wykonanie tego zadania?”. Na kartce papieru napisałem cyfrę 0 i rzucałem monetą. Jeżeli wypadł orzeł, do wyniku dodawałem 1. Jeżeli reszka, odejmowałem 1. Wykonalem 10 powtórzeń, co zajęło mi 42 sekundy. A więc jeden „obieg pętli” trwał 4,2 sekundy. Na tej podstawie obliczyłem, że wykonanie tej czynności 100 milionów razy zajęłoby mi 13,3 lat, pod warunkiem że robiłbym to bez przerwy. Wniosek: mój komputer jest około 52,3 milionów razy szybszy ode mnie.

A	B	C	D
1 Czas komputera:			
2 Czas wykonania 100 milionów obiegów pętli:	8,03	sekundy	
3 Czas wykonania jednego obiegu pętli:	8,03E-08	sekundy	
4			
5 Twój szacowany czas:			
6 Czas ręcznego wykonania jednego obiegu pętli	4,2	sekundy	
7 Czas ręcznego wykonania 100 milionów obiegów pętli:	420 000 000	sekund	
8	7 000 000	minut	
9	116 667	godzin	
10	4 861	dni	
11	13,3	lat	
12			
13 Wniosek:			
14 Twój komputer jest ok.	52 303 861	razy szybszy od Ciebie.	
15			
16			
17			

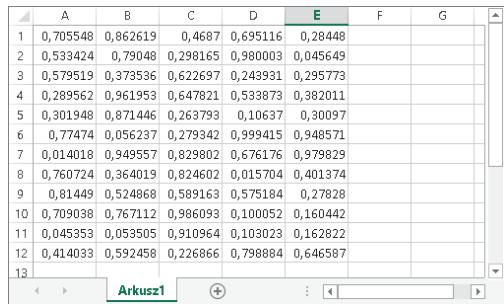
Przykład zagnieżdzonej pętli For-Next

Do tej pory wszystkie przykłady w tym rozdziale zawierały dość proste pętle. Ale wewnętrz pętli możesz umieścić dowolną liczbę instrukcji, a także zagnieżdżać jedną pętlę For-Next w innej.

W kolejnym przykładzie wykorzystam zagnieżdżone pętle For-Next do umieszczenia przypadkowych liczb w zakresie komórek o wymiarach 12 wierszy × 5 kolumn, w sposób

pokazany na rysunku 10.3. Zauważ, że kod wykonuje *wewnętrzna pętlę* (z licznikiem Row) w każdym obiegu *zewnętrznej pętli* (z licznikiem Col). Innymi słowy, kod 60 razy wykonuje instrukcję Cells(Row, Col) = Rnd.

```
Sub FillRange()
    Dim Col As Long
    Dim Row As Long
    For Col = 1 To 5
        For Row = 1 To 12
            Cells(Row, Col) = Rnd
        Next Row
    Next Col
End Sub
```



	A	B	C	D	E	F	G
1	0,705548	0,862619	0,4687	0,695116	0,28448		
2	0,533424	0,79048	0,298165	0,980003	0,045649		
3	0,579519	0,373536	0,622697	0,243931	0,295773		
4	0,289562	0,961953	0,647821	0,533873	0,382011		
5	0,301948	0,871446	0,263793	0,10637	0,30097		
6	0,77474	0,056237	0,279347	0,999415	0,948571		
7	0,014018	0,949957	0,829802	0,676176	0,979829		
8	0,760724	0,364019	0,824602	0,015704	0,401374		
9	0,81449	0,524868	0,589163	0,575184	0,27828		
10	0,709038	0,767112	0,986093	0,100052	0,160442		
11	0,045353	0,053505	0,910964	0,103023	0,162822		
12	0,414033	0,592458	0,226866	0,798884	0,646587		
13							

Arkusz1

Rysunek 10.3.
Komórki wypełnione wartościami przy użyciu zagnieżdżonych pętli For-Next

Następny przykład wykorzystuje zagnieżdżone pętle For-Next do zainicjowania trójwymiarowej tablicy wartością 100. Kod wykonuje 1000 razy instrukcję wewnętrz w wszystkich pętli (instrukcję przypisania), za każdym razem z różną kombinacją wartości zmiennych i, j oraz k.

```
Sub NestedLoops()
    Dim MyArray(10, 10, 10)
    Dim i As Integer
    Dim j As Integer
    Dim k As Integer
    For i = 1 To 10
        For j = 1 To 10
            For k = 1 To 10
                MyArray(i, j, k) = 100
            Next k
        Next j
    Next i
    'Tu są inne instrukcje
End Sub
```

Informacje o tablicach znajdziesz w rozdziale 7.

Poniżej znajduje się ostatni przykład zagnieżdżonych pętli For-Next z wartością Step. Procedura tworzy szachownicę, zmieniając naprzemiennie kolor tła komórek arkusza (zobacz rysunek 10.4).

	A	B	C	D	E	F	G	H	I	J	K
1	Red	White	Red	White	Red	White	Red	White			
2	White	Red	White	Red	White	Red	White	Red			
3	Red	White	Red	White	Red	White	Red	White			
4	White	Red	White	Red	White	Red	White	Red			
5	Red	White	Red	White	Red	White	Red	White			
6	White	Red	White	Red	White	Red	White	Red			
7	Red	White	Red	White	Red	White	Red	White			
8	White	Red	White	Red	White	Red	White	Red			
9											
10											
11											
12											

Rysunek 10.4.
Zastosowanie
pętli do utwo-
rzenia wzoru
szachownicy

Zmienna Row przyjmuje w pętli wartości od 1 do 8. Konstrukcja If-Then decyduje, która zagnieźdzona struktura For-Next ma być użyta. Dla nieparzystych wierszy licznik Col startuje od wartości 2, natomiast dla parzystych od 1. Obie pętle wewnętrzne wykorzystują wartość Step równą 2, dzięki czemu zmieniana jest co druga komórka.

```
Sub MakeCheckerboard()
    Dim Row As Integer, Col As Integer
    For Row = 1 To 8
        If WorksheetFunction.IsOdd(Row) Then
            For Col = 2 To 8 Step 2
                Cells(Row, Col).Interior.Color = 255
            Next Col
        Else
            For Col = 1 To 8 Step 2
                Cells(Row, Col).Interior.Color = 255
            Next Col
        End If
    Next Row
End Sub
```

Pętla Do-While

Język VBA obsługuje inny typ pętli zwanej Do-While. W odróżnieniu od pętli For-Next, pętla Do-While jest wykonywana dopóty, dopóki spełniony jest warunek pętli.

W poniższym przykładzie wykorzystana została pętla Do-While. Kod przyjmuje bieżącą komórkę jako punkt startowy i przemieszcza się w dół kolumny, mnożąc zawartość każdej komórki przez 2. Pętla jest wykonywana dotąd, aż kod napotka pustą komórkę.

```
Sub DoWhileDemo()
    Do While ActiveCell.Value <> Empty
        ActiveCell.Value = ActiveCell.Value * 2
    Loop
```

```

ActiveCell.Offset(1, 0).Select
Loop
End Sub

```

Pętla Do-Until

Struktura pętli Do-Until jest podobna do Do-While. Obie struktury różnią się sposobem obsługi sprawdzanego warunku. Program wykonuje pętlę Do-While, dopóki warunek jest spełniony. Natomiast pętlę Do-Until program wykonuje *dotąd*, aż zostanie spełniony warunek.

Następny przykład jest taki sam jak dla pętli Do-While, ale kod został zmieniony i wykorzystuje w nim pętlę Do-Until.

```

Sub DoUntilDemo()
    Do Until IsEmpty(ActiveCell.Value)
        ActiveCell.Value = ActiveCell.Value * 2
        ActiveCell.Offset(1, 0).Select
    Loop
End Sub

```

Użycie pętli For Each-Next z kolekcjami

Język VBA obsługuje jeszcze inny rodzaj pętli — przeglądanie każdego obiektu w kolekcji. Jak pamiętasz, kolekcja składa się z pewnej liczby obiektów tego samego typu. Przykładowo Excel zawiera kolekcję wszystkich otwartych skoroszytów (kolekcja Workbooks), a każdy skoroszyt zawiera kolekcję arkuszy (kolekcja Worksheets).



Wszystkie przykłady do tej części rozdziału są dostępne na stronie internetowej książki.

Jeżeli zamierzasz przejrzeć wszystkie obiekty w kolekcji, zastosuj strukturę For Each-Next. W poniższym przykładzie przeglądane są wszystkie arkusze w aktywnym skoroszycie i jeżeli arkusz jest pusty, jest usuwany.

```

Sub DeleteEmptySheets()
    Dim WkSht As Worksheet
    Application.DisplayAlerts = False
    For Each WkSht In ActiveWorkbook.Worksheets
        If WorksheetFunction.CountA(WkSht.Cells) = 0 Then
            WkSht.Delete
        End If
    Next WkSht
    Application.DisplayAlerts = True
End Sub

```

W tym przykładzie WkSht jest zmienną obiektową reprezentującą każdy arkusz w skoroszycie. W nazwie zmiennej nie ma nic wyjątkowego — możesz użyć dowolnej nazwy.

Kod przegląda każdy arkusz i sprawdza, czy jest pusty; zlicza też niepuste komórki. Jeżeli ich liczba jest równa zero, oznacza to, że arkusz jest pusty, po czym następuje jego usunięcie. Zwróć uwagę, że na czas wykonywania pętli przypisałem do ustawienia DisplayAlerts wartość False. Bez tej operacji Excel wyświetlałby ostrzeżenie przed usunięciem każdego arkusza.

Poniżej przedstawiam inny przykład pętli For Each-Next. Procedura wykorzystuje pętlę do ukrycia w bieżącym skoroszycie wszystkich arkuszy, oprócz aktywnego.

```
Sub HideSheets()
    Dim Sht As Worksheet
    For Each Sht In ActiveWorkbook.Worksheets
        If Sht.Name <> ActiveSheet.Name Then
            Sht.Visible = xlSheetHidden
        End If
    Next Sht
End Sub
```

Procedura HideSheets sprawdza nazwę arkusza. Jeżeli nie jest taka sama jak nazwa aktywnego arkusza, arkusz jest ukrywany. Zauważ, że właściwość Visible nie jest typu Boolean. Ta właściwość może przybierać *trzy* wartości, a w programie Excel dostępne są odpowiednie wbudowane stałe. Jeżeli jesteś ciekaw, jaka jest trzecia wartość, zajrzyj do systemu pomocy.

Co zostało ukryte, musi być w końcu odkryte, więc poniżej przedstawiam makro odkrywające wszystkie arkusze w aktywnym skoroszycie.

```
Sub UnhideSheets()
    Dim Sht As Worksheet
    For Each Sht In ActiveWorkbook.Worksheets
        Sht.Visible = xlSheetVisible
    Next Sht
End Sub
```

Nie ma nic zaskakującego w tym, że możesz tworzyć zagnieżdzone pętle For Each-Next. Procedura CountBold przegląda wszystkie komórki w używanym zakresie i wyświetla liczbę komórek, których zawartość jest zapisana pogrubioną czcionką.

```
Sub CountBold()
    Dim WBook As Workbook
    Dim WSheet As Worksheet
    Dim Cell As Range
    Dim cnt As Long
    For Each WBook In Workbooks
        For Each WSheet In WBook.Worksheets
            For Each Cell In WSheet.UsedRange
                If Cell.Font.Bold = True Then cnt = cnt + 1
            Next Cell
        Next WSheet
    Next WBook
    MsgBox "Znaleziono " & cnt & " komórek z pogrubieniem"
End Sub
```


Rozdział 11

Automatyczne procedury i zdarzenia

W tym rozdziale:

- poznasz typy zdarzeń, które mogą uruchamiać kod,
- dowiesz się, gdzie umieszczać kod VBA obsługujący zdarzenia,
- uruchomisz makro przy otwarciu i zamknięciu skoroszytu,
- uruchomisz makro przy aktywacji arkusza lub skoroszytu.

Procedurę VBA Sub możesz uruchomić na kilka sposobów. Jednym z nich jest przystosowanie jej do automatycznego uruchamiania. W tym rozdziale opiszę bardzo dokładnie tę potencjalnie przydatną funkcjonalność i wy tłumaczę, jak wszystko skonfigurować, aby makro było uruchamiane automatycznie w momencie pojawienia się określonego zdarzenia.

Przygotowanie do wielkiego zdarzenia

O jakiego rodzaju zdarzeniach tu piszę? Dobre pytanie. *Zdarzenie* to jest coś, co dzieje się w programie Excel. Poniżej wymienione są przykłady zdarzeń, które Excel rozpozna je.

- ✓ Skoroszyt został zamknięty lub otwarty.
- ✓ Okno zostało aktywowane lub dezaktywowane.
- ✓ Skoroszyt został aktywowany lub dezaktywowany.
- ✓ Do komórki zostały wprowadzone dane lub komórka została zmieniona.
- ✓ Skoroszyt został zapisany.
- ✓ Został kliknięty obiekt, na przykład przycisk.
- ✓ Został naciśnięty określony klawisz lub ich kombinacja.
- ✓ Nadeszła określona pora dnia.
- ✓ Wystąpił błąd.

Większość programistów programu Excel nie przejmuje się zdarzeniami wymienionymi na tej liście. Jednak Ty powinieneś przynajmniej wiedzieć, że takie zdarzenia istnieją, ponieważ mogą się przydać któregoś dnia. W tym rozdziale omówię większość najczęściej wykorzystywanych zdarzeń. Aby rzecz uproszczyć, będę mówić o dwóch typach zdarzeń: dotyczących skoroszytu i arkusza.

W tabeli 11.1 zamieściłem kilka bardziej użytecznych zdarzeń dotyczących skoroszytu. Jeżeli z jakiegoś powodu chciałbyś poznać pełną listę zdarzeń, możesz ją znaleźć w systemie pomocy.

Tabela 11.1. Zdarzenia dotyczące skoroszytu

Zdarzenie	Kiedy występuje?
Activate	Podczas aktywacji skoroszytu
BeforeClose	Podczas zamykania skoroszytu
BeforePrint	Podczas drukowania skoroszytu
BeforeSave	Podczas zapisywania skoroszytu
Deactivate	Podczas dezaktywacji skoroszytu
NewSheet	Podczas dodania nowego arkusza do skoroszytu
Open	Podczas otwierania skoroszytu
SheetActivate	Podczas aktywacji arkusza w skoroszycie
SheetBeforeDoubleClick	Podczas podwójnego kliknięcia komórki w skoroszycie
SheetBeforeRightClick	Podczas kliknięcia komórki w skoroszycie prawym przyciskiem myszy
SheetChange	Podczas zmiany komórki w skoroszycie
SheetDeactivate	Podczas dezaktywacji arkusza w skoroszycie
SheetSelectionChange	Podczas zmiany zaznaczenia
WindowActivate	Podczas aktywacji okna skoroszytu
WindowDeactivate	Podczas dezaktywacji okna skoroszytu

W tabeli 11.2 podaję kilka przydatnych zdarzeń dotyczących arkusza.

Tabela 11.2. Zdarzenia dotyczące arkusza

Zdarzenie	Kiedy występuje?
Activate	Podczas aktywacji arkusza
BeforeDoubleClick	Podczas podwójnego kliknięcia komórki w arkuszu
BeforeRightClick	Podczas kliknięcia komórki w arkuszu prawym przyciskiem myszy
Change	Podczas zmiany komórki w arkuszu
Deactivate	Podczas dezaktywacji arkusza
SelectionChange	Podczas zmiany zaznaczenia

Czy zdarzenia są przydatne?

W tym momencie pewnie zastanawiasz się, do czego mogą się przydać zdarzenia. Oto prosty przykład.

Założymy, że masz skoroszyt, w którym wprowadzasz wartości w kolumnie A. Twój bardzo pedantyczny szef powiedział, że chce wiedzieć, kiedy została wprowadzana każda liczba. Wprowadzenie danych jest zdarzeniem WorksheetChange. Możesz napisać makro, które reaguje na takie zdarzenie. Takie makro będzie wywoływanie, gdy zostanie zmieniony arkusz. Jeżeli zmiana zostanie wprowadzona w kolumnie A, makro wpisze obok w kolumnie B datę i czas wprowadzenia danych.

Niżej przedstawione jest takie makro. Prawdopodobnie jest o wiele prostsze, niż myślałeś, prawda?

```
Private Sub Worksheet_Change(ByVal Target As Range)
    If Target.Column = 1 Then
        Target.Offset(0, 1) = Now
    End If
End Sub
```

Przy okazji, makra reagujące na zdarzenia są bardzo grymańskie w kwestii miejsca ich utworzenia. Przykładowo powyższe makro *Worksheet_Change* *musi być umieszczone* w module kodu zmienianego skoroszytu. Jeżeli zapiszesz je w innym miejscu, nie będzie działać. Więcej na ten temat powiem później (zobacz podrozdział „*Gdzie jest umieszczony kod VBA?*”).



To, że Twój skoroszyt zawiera procedury reagujące na zdarzenia, nie oznacza, że te procedury będą rzeczywiście działać. Jak już wiesz, możliwe jest otwarcie skoroszytu z wyłączonymi makrami. W takim przypadku wszystkie makra (nawet procedury reagujące na zdarzenia) będą wyłączone. Pamiętaj o tym, gdy będziesz tworzył skoroszyty zawierające procedury obsługi zdarzeń.

Programowanie procedur obsługi zdarzeń

Procedura VBA wykonywana w wyniku wystąpienia zdarzenia jest nazywana *procedurą obsługi zdarzenia*. Są to zawsze procedury *Sub* (w odróżnieniu od procedur *Function*). Jeżeli zrozumiesz działanie całego procesu, tworzenie takich procedur będzie dość proste.

Budowanie procedur obsługi zdarzeń składa się z kilku kroków.

1. Określ zdarzenie, które ma uruchamiać procedurę.
2. Naciśnij *Alt+F11*, aby aktywować edytor Visual Basic Editor.
3. W oknie projektu VBA kliknij dwukrotnie odpowiedni obiekt umieszczony na liście *Microsoft Excel Objects*.

W przypadku zdarzeń dotyczących skoroszytu tym obiektem jest *Ten_skoroszyt*. Natomiast w przypadku arkusza — obiekt *Arkusz* (na przykład *Arkusz1*).

4. Napisz w oknie kodu odpowiedniego obiektu procedurę obsługi zdarzenia, która będzie uruchamiana, gdy takowe wystąpi.

Procedura ma specjalną nazwę oznaczającą, że jest to procedura obsługi zdarzenia.

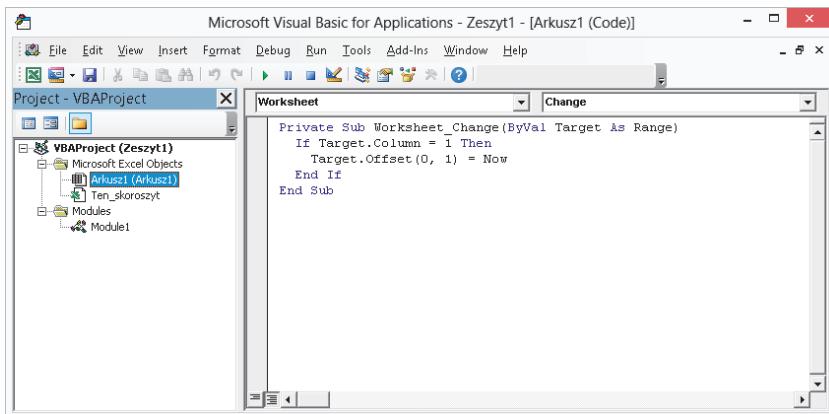
Powysze kroki staną się bardziej zrozumiałe, gdy przeczytasz ten rozdział. Zaufaj mi.

Gdzie jest umieszczony kod VBA?

Bardzo ważną rzeczą jest zrozumienie, gdzie są umieszczane Twoje procedury obsługi zdarzeń. Muszą znajdować się w oknie kodu w module odpowiedniego obiektu. Nie mogą być zapisane w standardowym module VBA. Jeżeli umieścisz swoją procedurę obsługi zdarzenia w niewłaściwym miejscu, po prostu nie będzie działać. Nie zobaczyesz nawet komunikatu o błędzie.

Na rysunku 11.1 przedstawiam okno edytora VBE zawierającego jeden projekt w oknie *Project* (podstawowe informacje o edytorze VBE podaję w rozdziale 3.). Zauważ, że cały projekt VBA w skoroszcycie *Zeszyt1* jest rozwinięty i składa się z kilku obiektów:

- ✓ jednego obiektu reprezentującego arkusz (obiekt *Arkusz1*),
- ✓ obiektu o nazwie *Ten_skoroszyt*,
- ✓ modułu VBA, który dodałem ręcznie za pomocą polecenia *Insert/Module*.



Rysunek 11.1.
Okno Project z elementami jednego projektu

Dwukrotne kliknięcie dowolnego obiektu powoduje wyświetlenie przypisanego do niego kodu, jeżeli taki jest.

Tworzone przez Ciebie procedury obsługi zdarzeń umieszczone są w oknie kodu elementu *Ten_skoroszyt* (dla zdarzeń dotyczących skoroszytu) lub w oknie odpowiedniego obiektu *Arkusz* (dla zdarzeń dotyczących arkusza).

Na rysunku 11.1 przedstawiam też okno kodu dla obiektu *Arkusz1*, w którym zdefiniowana jest jedna procedura obsługi zdarzenia. Zauważysz dwie rozwijane listy w górnej części okna kodu? Czytaj dalej, a dowiesz się dlaczego są takie przydatne.

Tworzenie procedury obsługi zdarzenia

Jeżeli jesteś gotowy do utworzenia procedury obsługi zdarzenia, edytor VBE oferuje swoją pomoc: wyświetla listę wszystkich zdarzeń dla wybranego obiektu.

W górnej części okna kodu znajdują się dwie rozwijane listy:

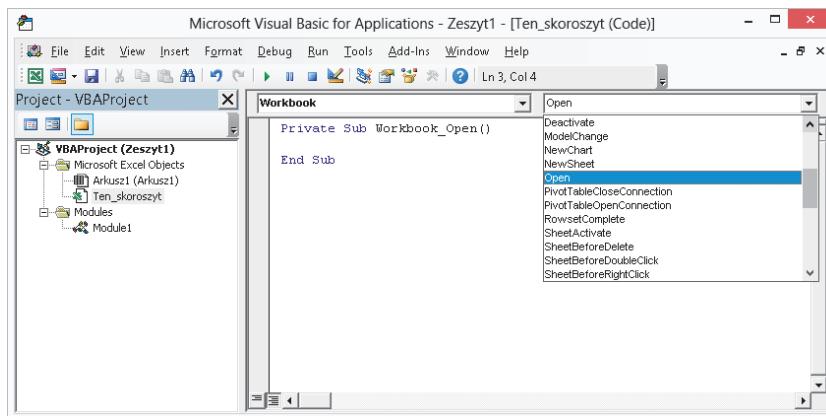
- ✓ lista *Object* (po lewej stronie),
- ✓ lista *Procedure* (po prawej stronie).

Domyślnie lista *Object* w oknie kodu zawiera napis *General*.

Jeżeli chcesz utworzyć procedurę obsługi zdarzenia dla obiektu *Ten_skoroszyt*, kliknij dwukrotnie jego nazwę w oknie *Project*, a następnie w liście *Object* wybierz pozycję *Workbook* (to jedyna możliwość).

Jeżeli chcesz utworzyć procedurę obsługi zdarzenia dla obiektu *Arkusz*, kliknij dwukrotnie jego nazwę w oknie *Project*, a następnie w liście *Object* wybierz pozycję *Worksheet* (również jest tylko jedyna możliwość).

Po wybraniu pozycji z listy *Object* możesz wybrać zdarzenie z listy *Procedure*. Na rysunku 11.2 przedstawiam kilka pozycji dotyczących zdarzeń arkusza.



Rysunek 11.2.
Wybór zdarzenia dla obiektu *Ten_skoroszyt* w oknie kodu

Po wybraniu zdarzenia z listy edytor VBE automatycznie utworzy za Ciebie procedurę obsługi zdarzenia. To bardzo przydatna funkcjonalność, ponieważ dostarcza dokładnych informacji o właściwych argumentach.

W tym miejscu powstaje małe zamieszanie. Przy pierwszym wybraniu elementu *Workbook* w liście *Object* edytor VBE zawsze zakłada, że chcesz utworzyć procedurę obsługi dla zdarzenia *Open* i tworzy taką procedurę. Jeżeli rzeczywiście masz zamiar utworzyć procedurę *Workbook_Open*, to dobrze, ale jeżeli jakąś inną, wtedy musisz usunąć pustą procedurę *Sub Workbook_Open*.

Jednak pomoc VBE na tym się kończy. Edytor wpisuje instrukcje *Sub* i *End Sub*. Natomiast napisanie kodu VBA pomiędzy tymi instrukcjami jest Twoim zadaniem.



W rzeczywistości nie musisz używać tych dwóch list, ale ułatwiają Ci pracę, ponieważ nazwa procedury obsługi zdarzenia jest bardzo ważna. Jeżeli nie wpiszesz dokładnie takiej nazwy, jak trzeba, procedura nie będzie działać. Ponadto niektóre procedury obsługi zdarzeń zawierają jeden lub więcej argumentów w instrukcji *Sub*. Nie jest możliwe, abyś pamiętała, jakie to są argumenty. Jeżeli na przykład wybierzesz zdarzenie *SheetActivate* dla obiektu *Workbook*, edytor VBE wpisze instrukcję:

```
Private Sub Workbook_SheetActivate(ByVal Sh As Object)
```

W tym przypadku *Sh* jest argumentem przekazywanym procedurze i jest to zmienna reprezentująca arkusz w aktywowanym skoroszycie. W przykładach zamieszczonych w tym rozdziale wyjaśnię to zagadnienie.

Przykłady wprowadzające

W tym podrozdziale przedstawię kilka przykładów, dzięki którym dowiesz się, na czym polega obsługa zdarzeń.

Zdarzenie *Open* dla skoroszytu

Jednym z najczęściej wykorzystywanych zdarzeń jest zdarzenie *Workbook Open*. Założmy, że posiadasz skoroszyt, którego codziennie używasz. W tym przypadku przy każdym otwarciu skoroszytu uruchamiana jest procedura *Workbook_Open*.

Procedura sprawdza dzień tygodnia. Jeżeli jest to piątek, kod wyświetli komunikat z przypomnieniem.

Wykonaj następujące kroki, aby otworzyć procedurę, która będzie uruchamiana za każdym razem, gdy pojawi się zdarzenie *Workbook Open*.

1. Otwórz skoroszyt.

Może to być dowolny skoroszyt.

2. Naciśnij *Alt+F11*, aby aktywować edytor VBE.

3. Odszukaj skoroszyt w oknie *Project*.

4. Jeżeli trzeba, kliknij dwukrotnie nazwę projektu, aby wyświetlić jego elementy.

5. Kliknij dwukrotnie element *Ten_skorosztyt*.

VBE otworzy puste okno kodu dla obiektu *Ten_skorosztyt*.

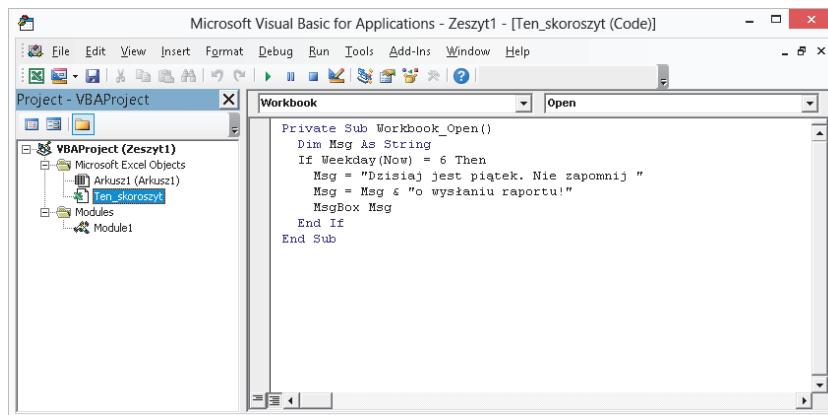
6. W oknie kodu wybierz w rozwijanej liście *Object* (po lewej) pozycję *Workbook*.

VBE wpisze początkową i końcową instrukcję procedury *Workbook_Open*.

7. Wpisz poniższe instrukcje, aby kompletna procedura obsługi zdarzenia wyglądała tak, jak poniżej:

```
Private Sub Workbook_Open()
    Dim Msg As String
    If Weekday(Now) = 6 Then
        Msg = "Dzisiaj jest piątek. Nie zapomnij "
        Msg = Msg & "o wysłaniu raportu!"
        MsgBox Msg
    End If
End Sub
```

Okno kodu będzie wyglądało tak, jak na rysunku 11.3.



Rysunek 11.3.

Powyższa procedura obsługuje zdarzenia będące uruchamiane przy otwarciu skorosztytu

Procedura *Workbook_Open* będzie automatycznie uruchamiana podczas otwarcia skorosztytu. Do określenia dnia tygodnia jest wykorzystana funkcja VBA *Weekday*. Jeżeli jest to piątek (dzień 6.), pojawia się komunikat przypominający użytkownikowi o wysłaniu raportu. Jeżeli to nie piątek, nic się nie dzieje.

Jeżeli dzisiaj nie jest piątek, możesz mieć problem z przetestowaniem tej procedury. Kiedy jednak zmienisz wartość 6 na inną liczbę odpowiadającą bieżącemu dniu tygodnia, problem zniknie.

Oczywiście, możesz zmodyfikować tę procedurę według własnego uznania. Przykładowo poniższa wersja wyświetla komunikat za każdym razem, gdy skorosztyt jest otwierany. A to po jakimś czasie staje się męczące, uwierz mi.

```
Private Sub Workbook_Open()
    Msg = "To jest fajny skoroszyt Piotra!"
    MsgBox Msg
End Sub
```

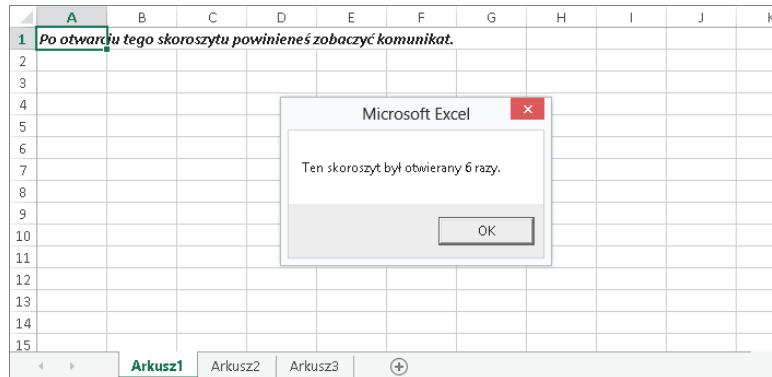
Procedura Workbook_Open może wykonywać niemal dowolne czynności. Obsługa tego zdarzenia jest często wykorzystywana do:

- ✓ wyświetlania komunikatów powitalnych (na przykład, takiego jak w fajnym skoroszycie Piotra),
- ✓ otwierania innych skoroszytów,
- ✓ aktywowania określonego arkusza w skoroszycie,
- ✓ ustawiania własnego menu użytkownika.

Poniżej przedstawiam ostatni przykład procedury Workbook_Open, wykorzystującej funkcje GetSetting oraz SaveSetting do rejestracji, jak często skoroszyt był otwierany. Funkcja SaveSetting wpisuje wartość do rejestru Windows, natomiast GetSetting odczytuje ją (szczegółowe informacje znajdziesz w systemie pomocy).

W poniższym przykładzie pobierany jest z rejestru licznik, następuje jego zwiększenie i zapis z powrotem w rejestrze. Podawana jest również użytkownikowi wartość zmiennej Cnt opisującej, ile razy skoroszyt był otwierany (zobacz 11.4).

```
Private Sub Workbook_Open()
    Dim Cnt As Long
    Cnt = GetSetting("MyApp", "Settings", "Open", 0)
    Cnt = Cnt + 1
    SaveSetting "MyApp", "Settings", "Open", Cnt
    MsgBox "Ten skoroszyt był otwierany " & Cnt & " razy."
End Sub
```



Rysunek 11.4.
Użycie procedury obsługi zdarzenia Workbook_Open do śledzenia, ile razy otwierany był skoroszyt

Zdarzenie *BeforeClose* dla skoroszytu

Poniżej przedstawiony jest przykład procedury obsługi zdarzenia *Workbook_BeforeClose*, automatycznie uruchamianej tuż przed zamknięciem skoroszytu. Procedura jest umieszczona w oknie kodu obiektu *Ten_skoroszyt*.

```
Private Sub Workbook_BeforeClose(Cancel As Boolean)
    Dim Msg As String
    Dim Ans As Integer
    Dim FName As String
    Msg = "Czy chcesz zrobić kopię zapasową tego pliku?"
    Ans = MsgBox(Msg, vbYesNo)
    If Ans = vbYes Then
        FName = "F:\BACKUP\" & ThisWorkbook.Name
        ThisWorkbook.SaveCopyAs FName
    End If
End Sub
```

Powyższy kod wykorzystuje okno komunikatu do zapytania użytkownika, czy chce utworzyć kopię zapasową skoroszytu. Jeżeli odpowiedź jest twierdząca, kod wykorzystuje metodę *SaveCopyAs* do zapisania kopii zapasowej pliku na dysku F. Jeżeli chcesz dostosować tę procedurę do swoich potrzeb, odpowiednio zmień nazwę napędu i ścieżkę.

Programiści programu Excel często wykorzystują procedurę *Workbook_BeforeClose* do posprzątania po sobie. Jeśli na przykład użyjesz procedury *Workbook_Open* do zmiany pewnych ustawień po otwarciu skoroszytu (przykładowo do ukrycia paska stanu), to wypada, abyś przy zamknięciu skoroszytu przywrócił ustawienia do pierwotnego stanu. Takie elektroniczne domowe porządki możesz wykonać za pomocą procedury *Workbook_BeforeClose*.



W zdarzeniu *Workbook BeforeClose* kryje się mała pułapka. Jeżeli przy zamknięciu programu Excel jakieś pliki zostały zmienione od momentu ostatniego zapisania, Excel wyświetli standardowy komunikat „Czy zapisać zmiany?”. Kliknięcie przycisku *Anuluj* przerwuje cały proces zamknięcia. Ale procedura *Workbook_BeforeClose* i tak zostanie wykonana.

Zdarzenie *BeforeSave* dla skoroszytu

Zdarzenie *BeforeSave* (przed zapisem), jak sama nazwa wskazuje, występuje przed zapisaniem skoroszytu, gdy użyjesz polecenia *Plik/Zapisz* lub *Plik/Zapisz jako*.

Poniższa procedura, umieszczona w oknie kodu obiektu *Ten_skoroszyt*, stanowi obsługę zdarzenia *BeforeSave*. Za każdym razem, gdy skoroszyt jest zapisywany, kod aktualizuje zawartość komórki A1 w arkuszu *Arkusz1*. Inaczej mówiąc, komórka A1 pełni funkcję licznika podającego, ile razy skoroszyt był zapisany.

```
Private Sub Workbook_BeforeSave(ByVal SaveAsUI _
    As Boolean, Cancel As Boolean)
    Dim Counter As Range
    Set Counter = Sheets("Arkusz1").Range("A1")
    Counter.Value = Counter.Value + 1
End Sub
```

Zauważ, że procedura Workbook_BeforeSave ma dwa argumenty: SaveAsUI oraz Cancel. Aby pokazać znaczenie tych argumentów, popatrzmy na poniższe makro, uruchamiane przed zapisaniem skoroszytu. Procedura zapobiega zapisaniu skoroszytu pod inną nazwą. Jeżeli użytkownik wybierze polecenie *Plik/Zapisz jako*, argument SaveAsUI przyjmie wartość True.

Kod sprawdza wartość argumentu SaveAsUI. Jeżeli ma wartość True, procedura wyświetla komunikat i nadaje argumentowi Cancel wartość True, co powoduje przerwanie czynności zapisu.

```
Private Sub Workbook_BeforeSave(ByVal SaveAsUI _  
    As Boolean, Cancel As Boolean)  
    If SaveAsUI Then  
        MsgBox "Nie możesz zapisać kopii tego skoroszytu!"  
        Cancel = True  
    End If  
End Sub
```

Pamiętaj, że procedura ta w rzeczywistości nie zapobiega zapisaniu skoroszytu pod inną nazwą. Jeżeli ktoś będzie bardzo chciał to zrobić, może po prostu otworzyć skoroszyt z wyłączonymi makrami. Gdy makra są wyłączone, to procedury obsługi zdarzeń również, co jest logiczne, ponieważ procedury te są przecież makrami.

Przykłady zdarzeń aktywacyjnych

Inną kategorię zdarzeń stanowią aktywacja i dezaktywacja obiektów — szczególnie arkuszy i skoroszytów.

Zdarzenia aktywacji i dezaktywacji arkusza

Excel potrafi wykryć moment aktywacji lub dezaktywacji określonego arkusza i uruchomić makro, gdy wystąpi któryś z tych zdarzeń. Procedura obsługi takiego zdarzenia znajduje się w oknie kodu obiektu *Arkusz*.



Możesz szybko otworzyć okno kodu dla wybranego arkusza, klikając prawym przyciskiem myszy jego zakładkę i wybierając polecenie *Wyświetl kod*.

W poniższym przykładzie przedstawiam prostą procedurę uruchamianą w momencie aktywacji określonego arkusza. Kod wyświetla okno z komunikatem zawierającym nazwę aktywnego arkusza.

```
Private Sub Worksheet_Activate()  
    MsgBox "Aktywowałś arkusz " & ActiveSheet.Name  
End Sub
```

A tu następny przykład, w którym w momencie aktywacji arkusza zaznaczana jest komórka A1.

```
Private Sub Worksheet_Activate()
    Range("A1").Activate
End Sub
```

Kod obu powyższych procedur jest tak prosty, jak tylko możliwe, ale procedury obsługujące zdarzeń mogą być dowolnie skomplikowane.

Następna procedura (umieszczona w oknie kodu arkusza *Arkusz1*) wykorzystuje zdarzenie *Deactivate* i uniemożliwia użytkownikowi wybranie innego arkusza w skoroszycie. Jeżeli *Arkusz1* zostanie dezaktywowany, użytkownik zobaczy komunikat, po czym zostanie wybrany *Arkusz1*.

```
Private Sub Worksheet_Deactivate()
    MsgBox "Musisz pozostać w Arkusz1"
    Sheets("Arkusz1").Activate
End Sub
```

Na marginesie, nie polecam używać procedur, takich jak powyższa, które przejmują kontrolę nad programem Excel. Tego typu „dyktatorskie” aplikacje mogą bardzo frustrować użytkownika i utrudniać pracę. Poza tym, oczywiście, można je łatwo obejść, wyłączając makra. Zalecam raczej poinstruowanie użytkownika, jak właściwie korzystać z aplikacji.

Zdarzenia aktywacji i dezaktywacji skoroszytu

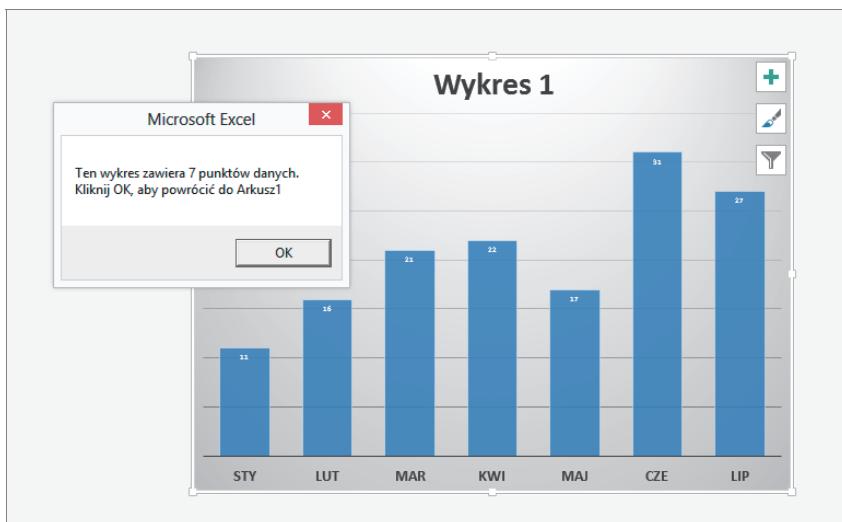
W poprzednich przykładach wykorzystałem zdarzenia dotyczące określonego arkusza. Jednak obiekt *Ten_skoroszyszt* obsługuje również zdarzenia dotyczące aktywacji i dezaktywacji dowolnego arkusza. Poniższa procedura, umieszczona w oknie kodu obiektu *Ten_skoroszyszt*, jest uruchamiana w momencie aktywacji dowolnego arkusza. Kod wyświetla komunikat zawierający nazwę aktywowanego arkusza.

```
Private Sub Workbook_SheetActivate(ByVal Sh As Object)
    MsgBox Sh.Name
End Sub
```

Procedura *Workbook_SheetActivate* posiada argument *Sh*. Jest to zmienna reprezentująca obiekt aktywnego arkusza. Okno komunikatu wyświetla właściwość *Name* obiektu *Sheet*.

Następny przykład jest umieszczony w oknie kodu obiektu *Ten_skoroszyszt*. Składa się z dwóch procedur obsługi zdarzeń. Oto one.

- ✓ **Workbook_SheetDeactivate:** wykonywana w momencie dezaktywacji dowolnego arkusza. Umieszcza w zmiennej obiektoowej dezaktywowany arkusz (słowo kluczowe *Set* tworzy zmienną obiektoową).
- ✓ **Workbook_SheetActivate:** wykonywana w momencie aktywacji dowolnego arkusza. Sprawdza typ aktywowanego arkusza (za pomocą funkcji *TypeName*). Jeżeli arkusz jest wykresem, użytkownik otrzymuje komunikat (zobacz rysunek 11.5). Jeżeli w oknie komunikatu zostanie kliknięty przycisk *OK*, aktywowany jest poprzedni arkusz (zapisany w zmiennej *oldSheet*).



Rysunek 11.5.
Jeżeli aktywowy-
wany zostanie
arkusz z wykre-
sem, użytkow-
nik otrzyma
komunikat



Skoroszyt zawierający ten kod jest dostępny na stronie internetowej książki.

```
Dim OldSheet As Object

Private Sub Workbook_SheetDeactivate(ByVal Sh As Object)
    Set OldSheet = Sh
End Sub

Private Sub Workbook_SheetActivate(ByVal Sh As Object)
    Dim Msg As String
    If TypeName(Sh) = "Chart" Then
        Msg = "Ten wykres zawiera "
        Msg = Msg & ActiveChart.SeriesCollection(1).Points.Count
        Msg = Msg & " punktów danych." & vbCrLf
        Msg = Msg & "Kliknij OK, aby powrócić do " & OldSheet.Name
        MsgBox Msg
        OldSheet.Activate
    End If
End Sub
```

Zdarzenia aktywacji skoroszytu

Excel rozpoznaje również zdarzenia występujące podczas aktywacji i dezaktywacji określonego skoroszytu. Poniższa procedura, umieszczona w oknie kodu obiektu `Ten_skoroszyt`, jest uruchamiana przy aktywacji skoroszytu. Kod po prostu maksymalizuje rozmiar okna skoroszytu.

```
Private Sub Workbook_Activate()
    ActiveWindow.WindowState = xlMaximized
End Sub
```

W kolejnym przykładzie przedstawiam procedurę Workbook_Deactivate. Jest ona wykonywana przy dezaktywacji skoroszytu. Procedura kopiuje wybrany zakres za każdym razem, gdy skoroszyt jest dezaktywowany. Może być przydatna w sytuacji, gdy kopujesz dane z wielu różnych obszarów i wklejasz je do innego skoroszytu. Gdy masz taką procedurę, możesz wybrać zakres do skopiowania, aktywować inny skoroszyt, wybrać miejsce docelowe i wkleić skopowane dane, naciskając *Ctrl+V* (lub *Enter*).

```
Private Sub Workbook_Deactivate()
    ThisWorkbook.Windows(1).RangeSelection.Copy
End Sub
```

Procedura jest bardzo prosta, ale wymagała pewnych eksperymentów, zanim zaczęła działać poprawnie. Najpierw spróbowałem tak.

```
Selection.Copy
```

Ta instrukcja nie działała tak, jak oczekiwalem. Kopiowała zakres z drugiego skoroszytu (aktywowanego po dezaktywacji pierwszego). Działało się tak dlatego, ponieważ po wystąpieniu zdarzenia dezaktywacji drugi skoroszyt stawał się aktywnym skoroszytem.

Następna instrukcja również nie działała. Wręcz wyświetlała komunikat o błędzie.

```
ThisWorkbook.ActiveSheet.Selection.Copy
```

Wreszcie przypomniałem sobie o właściwości RangeSelection obiektu Window i dopiero wtedy sztuczka się udała.

Inne zdarzenia dotyczące arkusza

W poprzednich punktach rozdziału zaprezentowałem przykłady zdarzeń aktywacji i dezaktywacji arkusza. W tym podrozdziale omówię trzy dodatkowe zdarzenia występujące w arkuszu: dwukrotnie kliknięcie komórki, kliknięcie komórki prawym przyciskiem myszy i zmianę zawartości komórki.

Zdarzenie *BeforeDoubleClick*

Możesz utworzyć procedurę, która będzie wykonywana, gdy użytkownik dwukrotnie kliknie komórkę. W kolejnym przykładzie (umieszczonym w oknie kodu obiektu Arkusz) dwukrotne kliknięcie komórki powoduje zmianę czcionki na pogrubioną (jeżeli nie jest) lub odwrotnie.

```
Private Sub Worksheet_BeforeDoubleClick
    (ByVal Target As Excel.Range, Cancel As Boolean)
        Target.Font.Bold = Not Target.Font.Bold
        Cancel = True
End Sub
```

Procedura Worksheet_BeforeDoubleClick posiada dwa argumenty — Target oraz Cancel. Argument Target reprezentuje dwukrotnie klikniętą komórkę (obiekt Range). Jeżeli argumentowi Cancel zostanie przypisana wartość True, domyślna reakcja programu na dwukrotnie kliknięcie nie będzie miała miejsca.

Domyślną reakcją programu na dwukrotnie kliknięcie jest przejście w tryb edycji komórki. Jeżeli nie chcę, aby tak było, przypisz argumentowi Cancel wartość True.

Zdarzenie BeforeRightClick

Zdarzenie BeforeRightClick jest podobne do BeforeDoubleClick z tą różnicą, że dotyczy kliknięcia komórki prawym przyciskiem myszy. Następna procedura sprawdza, czy komórka kliknięta prawym przyciskiem zawiera wartość liczbową. Jeżeli tak, kod wyświetla okno dialogowe *Formatowanie komórek* i przypisuje argumentowi Cancel wartość True (uniemożliwiając normalne wyświetlenie menu podręcznego). Jeżeli natomiast komórka nie zawiera wartości liczbowej, nic się nie dzieje — menu podręczne jest wyświetlane w zwykły sposób.

```
Private Sub Worksheet_BeforeRightClick
    (ByVal Target As Excel.Range, Cancel As Boolean)
    If IsNumeric(Target) And Not IsEmpty(Target) Then
        Application.CommandBars.ExecuteMso ("NumberFormatsDialog")
        Cancel = True
    End If
End Sub
```

Zwróć uwagę, że powyższy kod (dostępny na stronie internetowej książki) dodatkowo sprawdza, czy komórka nie jest pusta. Jest tak dlatego, ponieważ VBA traktuje puste komórki tak, jakby zawierały wartości liczbowe. Nie pytaj mnie, dlaczego, po prostu tak jest.



Zdarzenie Change

Zdarzenie Change występuje w momencie zmiany zawartości dowolnej komórki arkusza. W kolejnym przykładzie procedura Worksheet_Change skutecznie zapobiega wprowadzeniu do komórki A1 wartości innej niż liczba. Procedura jest umieszczona w oknie kodu obiektu *Arkusz*.

```
Private Sub Worksheet_Change(ByVal Target As Range)
    If Target.Address = "$A$1" Then
        If Not IsNumeric(Target) Then
            MsgBox "Wprowadź liczbę w komórce A1."
            Range("A1").ClearContents
            Range("A1").Activate
        End If
    End If
End Sub
```

Jedyny argument procedury `Worksheet_Change` (`Target`) reprezentuje zmieniony zakres. Pierwsza instrukcja sprawdza, czy zmieniona komórka ma adres `A1`. Jeżeli tak, kod korzysta z funkcji `IsNumeric` do określenia, czy komórka zawiera wartość liczbową. Jeżeli nie, pojawia się komunikat i zawartość komórki jest usuwana. Komórka `A1` jest następnie aktywowana — przydatna funkcjonalność, kiedy po wprowadzeniu danych następuje przemieszczenie do innej komórki. Jeżeli wprowadzona zmiana dotyczy komórki innej niż `A1`, nic się nie dzieje.

Czemu nie zastosować sprawdzania poprawności danych?

Być może znasz polecenie menu *DANE/Narzędzia danych/Poprawność danych*. Ta przydatna i prosta funkcjonalność sprawia, że do określonej komórki lub zakresu zostaną wprowadzone poprawne dane. Jakkolwiek to polecenie jest bardzo przydatne, nie jest jednak całkowicie niezawodne.

Spróbuj dodać do komórki sprawdzanie poprawności danych. Możesz na przykład określić, że komórka przyjmuje tylko wartości liczbowe. Ten sposób się sprawdza, o ile nie skopiujesz innej komórki i nie wkleisz wartości do sprawdzanej komórki. Operacja wklejania omija sprawdzanie poprawności. Zupełnie jakby jej nie było. Jak poważna jest to usterka, zależy od aplikacji. W następnym podpunkcie opisuję, jak wykorzystać zdarzenie `Change`, aby lepiej sprawdzać poprawność danych.



Operacja wklejania omija sprawdzanie, ponieważ Excel traktuje poprawność danych na równi z formatem komórki. Dlatego jest ona tak samo klasyfikowana jak wielkość czcionki, kolor lub inne podobne atrybuty. Podczas wklejania komórki jej format jest zastępowany formatem komórki źródłowej. Niestety, format obejmuje również reguły poprawności danych.

Zapobieganie usunięciu reguł sprawdzania poprawności danych

W procedurze przedstawionej w tym podpunkcie demonstруję, jak zapobiec skopiowaniu danych i skasowaniu przez użytkownika reguł sprawdzania poprawności. W przykładzie przyjęłem założenie, że arkusz zawiera zakres o nazwie `InputRange` i są dla niego zdefiniowane reguły sprawdzania poprawności danych (za pomocą polecenia *DANE/Narzędzia danych/Poprawność danych*). Zakres może posiadać dowolne reguły sprawdzania poprawności danych.

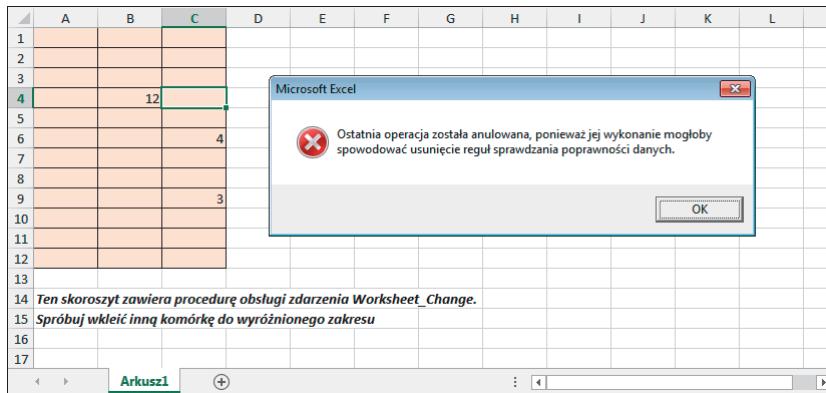


Skoroszyt zawierający ten kod jest dostępny na stronie internetowej książki.

```
Private Sub Worksheet_Change(ByVal Target As Range)
    Dim VT As Long
    'Czy wszystkie komórki w sprawdzanym zakresie
    'mają zdefiniowane reguły poprawności danych?
    On Error Resume Next
    VT = Range("InputRange").Validation.Type
    If Err.Number <> 0 Then
        Application.Undo
        MsgBox "Ostatnia operacja została anulowana, ponieważ jej wykonanie " &
        "mogłoby spowodować usunięcie reguł sprawdzania poprawności danych.", vbCritical
    End If
End Sub
```

Powysza procedura jest uruchamiana za każdym razem, gdy zostanie zmieniona zawartość komórki. Sprawdza typ reguły poprawności danych dla określonego zakresu (o nazwie *InputRange*), który może takie reguły zawierać. Jeżeli zmienna *VT* zawiera błąd, oznacza to, że jedna lub kilka komórek w zakresie *InputRange* nie zawiera reguł poprawności danych (użytkownik prawdopodobnie wkleił dane i usunął reguły). W takim przypadku kod wykonuje metodę *Undo* obiektu *Application*, wycofując ostatnią operację wykonaną przez użytkownika i wyświetla komunikat przedstawiony na rysunku 11.6.

Rysunek 11.6.
Sprawdzanie
reguł popraw-
ności danych za
pomocą proce-
dury obsługi
zdarzenia



Jaki jest efekt finalny zastosowania takiej procedury? Nie ma możliwości usunięcia reguł sprawdzania poprawności danych w wyniku kopiowania. Jak widać, niektóre błędy Excela możesz samodzielnie naprawić za pomocą VBA.

Zdarzenia niezwiązane z obiektami

Zdarzenia omówione wcześniej w tym rozdziale dotyczą obiektu arkusza lub skoroszytu. W tym podrozdziale opiszę dwa typy zdarzeń niezwiązańych z obiektami; są to upływ czasu i naciśnięcie klawisza.

Ponieważ zdarzenia dotyczące upływu czasu i naciśnięcia klawisza nie są związane z konkretnym obiektem, takim jak arkusz lub skoroszyt, dlatego ich obsługę kodujesz w zwykłym module VBA (w odróżnieniu od zdarzeń opisanych w tym rozdziale).



Zdarzenie *OnTime*

Zdarzenie *OnTime* występuje o określonej porze dnia. W poniższym przykładzie pokazuję, jak sprawić, aby Excel uruchomił wybraną procedurę, gdy pojawi się zdarzenie związane z nadaniem godziny 15:00. W tym przypadku obudzi Cię głoś robota, a na ekranie pojawi się dodatkowy komunikat.

```
Sub SetAlarm()
    Application.OnTime 0.625, "DisplayAlarm"
End Sub
```

```
Sub DisplayAlarm()
    Application.Speech.Speak ("Hej, obudź się")
    MsgBox "Czas na popołudniową przerwę!"
End Sub
```

W tym przykładzie użyłem metody `OnTime` obiektu `Application`. Zawiera ona dwa argumenty: czas (liczba 0,625 odpowiada godzinie 15:00) oraz nazwę procedury `Sub`, która ma być wykonana o zadanej godzinie (`DisplayAlarm`).

Procedura jest całkiem przydatna, jeżeli praca zaczyna Cię tak pochłaniać, że zapominasz o spotkaniach i terminach. Wystarczy, że skonfigurujesz zdarzenie `OnTime`, które o nich przypomni.



Zdaniem większości programistów (w tym autora) liczbowy zapis czasu w programie Excel jest nieczytelny. Dlatego do reprezentacji czasu warto stosować funkcję VBA `TimeValue`. Funkcja ta konwertuje ciąg znaków oznaczających czas na wartość liczbową obsługiwianą przez program Excel. Poniższa instrukcja przedstawia prostszy sposób programowania zdarzenia o godzinie 15:00.

```
Application.OnTime TimeValue("15:00:00"), "DisplayAlarm"
```

Jeżeli chcesz zaplanować zdarzenie w odniesieniu do bieżącego czasu (na przykład za 20 minut od chwili obecnej), możesz użyć też takiej instrukcji:

```
Application.OnTime Now + TimeValue("00:20:00"), "DisplayAlarm"
```

Metody `OnTime` możesz również użyć do uruchomienia procedury VBA określonego dnia. Musisz się upewnić, że Twój komputer będzie wtedy włączony i otwarty będzie skoroszyt zawierający tę procedurę. Poniższa instrukcja uruchamia procedurę `DisplayAlarm` w dniu 31 grudnia 2013 o godzinie 17:00.

```
Application.OnTime DateValue("2013-12-31 17:00"), "DisplayAlarm"
```

Ten konkretnie wiersz kodu może się przydać, aby Ci przypomnieć, że czas wraca do domu i przygotować się na powitanie Nowego Roku.

Poniżej przedstawiam inny przykład wykorzystujący zdarzenie `OnTime`. Uruchomienie procedury `UpdateClock` powoduje wpisanie do komórki A1 bieżącego czasu i zaprogramowanie następnego zdarzenia pięć sekund później. To zdarzenie ponownie uruchamia procedurę `UpdateClock`. W efekcie co pięć sekund jest wpisywany do komórki A1 aktualny czas. Aby zatrzymać obsługę zdarzeń, uruchom procedurę `StopClock` (przerywającą obsługę zdarzenia). Zwróć uwagę, że zmienna `NextTick` jest zadeklarowana na poziomie modułu i przechowuje czas wystąpienia następnego zdarzenia.

```
Dim NextTick As Date

Sub UpdateClock()
    ' Wpisanie aktualnego czasu do komórki A1
    ThisWorkbook.Sheets(1).Range("A1") = Time
    ' Ustawienie następnego zdarzenia za pięć sekund od chwili obecnej
    NextTick = Now + TimeValue("00:00:05")
    Application.OnTime NextTick, "UpdateClock"

```

OSTRZEŻENIE

```
End Sub

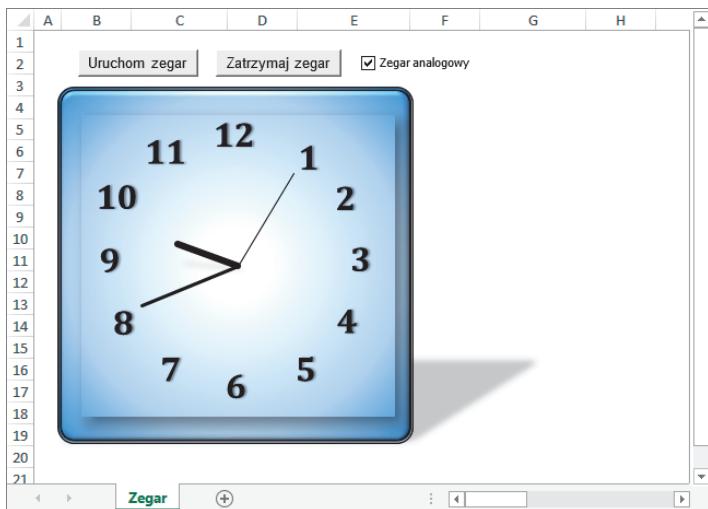
Sub StopClock()
    ' Przerwa zdarzenie OnTime (zatrzymuje zegar)
    On Error Resume Next
    Application.OnTime NextTick, "UpdateClock", , False
End Sub
```

Zdarzenie *OnTime* jest aktywne nawet po zamknięciu skoroszytu. Innymi słowy, jeżeli zamkniesz skoroszyt bez uprzedniego uruchomienia procedury *StopClock*, po pięciu sekundach skoroszyt otworzy się sam (zakładam tu, że program Excel jest wciąż otwarty). Aby temu zapobiec, zastosuj procedurę obsługi zdarzenia *Workbook_BeforeClose*, zawierającą poniższą instrukcję.

```
Call StopClock
```

Metoda *OnTime* ma dwa dodatkowe argumenty. Jeżeli zamierzasz z niej korzystać, zajrzyj do systemu pomocy w celu uzyskania szczegółowych informacji.

Gdybyś chciał zobaczyć trochę bardziej skomplikowaną aplikację, pobierz mój skoroszyt zawierający zegar analogowy, przedstawiony na rysunku 11.7. Tarcza zegara jest w rzeczywistości wykresem pokazującym bieżący czas i aktualizowanym co sekundę. Nieprzydatna aplikacja, ale fajna.



Rysunek 11.7.
Moja aplikacja z
zegarem analogo-
gowym

Zdarzenia naciśnięcia klawisza

Podczas korzystania z programu Excel nieustannie monitorowane jest wszystko, co wpisujesz. Możesz więc skonfigurować program tak, aby po naciśnięciu określonego klawisza lub ich kombinacji była uruchamiana konkretna procedura.

Kod w poniższym przykładzie przypisuje procedury do klawiszy *PgDn* i *PgUp*.

```
Sub Setup_OnKey()
    Application.OnKey "{PgDn}", "PgDn_Sub"
    Application.OnKey "{PgUp}", "PgUp_Sub"
End Sub

Sub PgDn_Sub()
    On Error Resume Next
    ActiveCell.Offset(1, 0).Activate
End Sub

Sub PgUp_Sub()
    On Error Resume Next
    ActiveCell.Offset(-1, 0).Activate
End Sub
```

Po przygotowaniu zdarzeń *OnKey* do uruchamiania procedury *Setup_OnKey* naciśnięcie klawisza *PgDn* przeniesie Cię o jeden wiersz w góre, natomiast *PgUp* — jeden wiersz w dół.

Zwróć uwagę, że kody klawiszy nie są ujęte w nawiasy okrągłe, ale klamrowe. W systemie pomocy znajdziesz pełną listę kodów klawiszy. Wyszukaj frazę *OnKey*.

W tym przykładzie zastosowałem instrukcję *On Error Resume Next*, aby ominąć zgłoszane błędy. Jeżeli przykładowo aktywna komórka znajduje się w pierwszym wierszu, wówczas przy próbie przemieszczenia się o jeden wiersz w góre zostanie zgłoszony błąd, który można bezpiecznie zignorować. Natomiast w arkuszu zawierającym wykres nie ma w ogóle aktywnej komórki.

Obsługę zdarzeń *OnKey* możesz przerwać, uruchamiając poniższą procedurę.

```
Sub Cancel_OnKey()
    Application.OnKey "{PgDn}"
    Application.OnKey "{PgUp}"
End Sub
```

 Użycie w drugim argumentie metody *OnKey* pustego ciągu znaków *nie przerywa* obsługi zdarzeń. Zamiast tego Excel po prostu ignoruje naciśnięcie zadanego klawisza. Przykładowo poniższa instrukcja powoduje, że Excel ignoruje naciśnięcie klawiszy *Alt+F4*. Znak procentu oznacza klawisz *Alt*.

```
Application.OnKey "%{F4}", ""
```

 Chociaż do przypisania procedury do skrótu klawiszowego możesz użyć metody *OnKey*, niemniej jednak powinieneś w tym celu skorzystać z okna dialogowego *Opcje makra*. Więcej szczegółowych informacji na ten temat znajdziesz w rozdziale 5.

 Jeżeli zamkniesz skoroszyt zawierający kod, ale Excel będzie otwarty, metoda *OnKey* nie zostanie zresetowana. W konsekwencji naciśnięcie skrótu klawiszowego spowoduje, że Excel automatycznie otworzy plik zawierający odpowiednie makro. Aby temu zapobiec, powinieneś w procedurze *Workbook_BeforeClose* (opisanej wcześniej w tym rozdziale) umieścić kod resetujący obsługę zdarzenia *OnKey*.

Rozdział 12

Techniki obsługi błędów

W tym rozdziale:

- poznasz różnice pomiędzy błędami programowania a błędami wykonania,
- przechwycisz i obsłużysz błędy wykonania,
- zastosujesz instrukcje VBA On Error i Resume,
- dowiesz się, jak obrócić błąd na Twoją korzyść.

Błędzić jest rzeczą ludzką, a wybaczać boską. Jeżeli programujesz w języku VBA, powinieneś wiedzieć, że istnieją dwa podstawowe rodzaje błędów — błędy programowania i błędy wykonania. Ten rozdział w całości poświęcam błędom wykonania. Błędy programowania, zwane również *pluskiami*, opisałem w rozdziale 13.

Dobrze napisany program obsługuje błędy tak, jak tańczył Fred Astaire — z gracją. Na szczęście, VBA zawiera kilka narzędzi ułatwiających wyszukiwanie błędów i ich elegancką obsługę.

Rodzaje błędów

Pracując z przykładami z tej książki, prawdopodobnie otrzymałeś jeden lub więcej komunikatów o błędach. Niektóre z nich są skutkiem źle napisanego kodu VBA, czyli niewłaściwie wpisanego słowa kluczowego lub instrukcji użytej niezgodnie ze składnią. Jeżeli popełnisz taki błąd, nie będziesz mógł nawet uruchomić procedury, dopóki go nie poprawisz.

Ten rozdział nie traktuje jednak o tego typu błędach. Omawiam w nim raczej błędy wykonania, takie które występują podczas wykonywania przez program Excel kodu VBA. Mówiąc ściślej, w tym rozdziale opisuję następujące fascynujące tematy:

- ✓ rozpoznawanie błędów,
- ✓ radzenie sobie z błędami, które wystąpią,
- ✓ wznowianie działania kodu po wystąpieniu błędu,
- ✓ zamierzone wywoływanie błędów (tak, czasami błąd jest dobrą rzeczą).

Właściwym celem obsługi błędów jest utworzenie kodu, który zapobiega, najskuteczniej jak jest to możliwe, wyświetlaniu przez program Excel komunikatów o problemach. Innymi słowy, będziesz wybaczal i obsługiwał potencjalne błędy, zanim Excel wyciągnie je na światło dzienne i wyświetli zazwyczaj mało zrozumiałą komunikat.

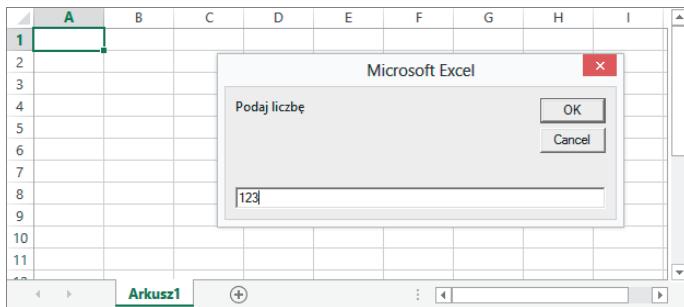
Błędny przykład

Aby wyjaśnić, w czym rzecz, przygotowałem krótkie i proste makro VBA. Takie prościutkie makro nie może spowodować błędu, prawda?

Otwórz edytor VBE, dodaj moduł i wprowadź poniższy kod.

```
Sub EnterSquareRoot()
    Dim Num As Double
    ' Poproś o liczbę
    Num = InputBox("Podaj liczbę")
    ' Umieść pierwiastek kwadratowy
    ActiveCell.Value = Sqr(Num)
End Sub
```

Na rysunku 12.1 możesz zobaczyć, że procedura prosi użytkownika o podanie wartości. Następnie wykonuje magiczne obliczenia i umieszcza w aktywnej komórce pierwiastek kwadratowy z tej wartości.



Rysunek 12.1.
Funkcja InputBox wyświetla okno dialogowe z prośbą o wprowadzenie danych

Powyższą procedurę możesz uruchomić bezpośrednio w edytorze VBE, naciskając klawisz F5. Innym sposobem jest dodanie przycisku w arkuszu (w tym celu użyj polecenia *DEVELOPER/Formanty/Wstaw* i wybierz *Przycisk* w grupie *Kontrolki formularza*) i przypisanie do niego makra (Excel zaproponuje, które makro przypisać do przycisku). Będziesz mógł wtedy uruchomić procedurę, po prostu klikając przycisk.

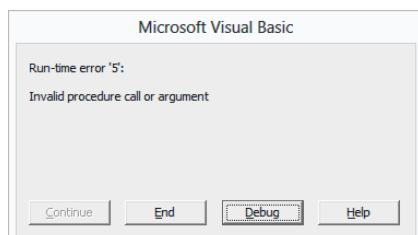


To makro nie jest idealne

Uruchom kod kilka razy i przetestuj go. Działa całkiem dobrze, prawda? W takim razie, gdy pojawi się pytanie, spróbuj wprowadzić ujemną liczbę. Ojej, wyliczenie pierwiastka kwadratowego z liczby ujemnej jest niemożliwe na tym świecie.

Na żądanie obliczenia pierwiastka kwadratowego z liczby ujemnej Excel reaguje wyświetleniem komunikatu o błędzie wykonania, pokazanym na rysunku 12.2. Teraz kliknij przycisk *End*. Jeżeli klikniesz *Debug*, Excel zawiesi wykonywanie makra po to, abyś mógł użyć narzędzi debugującego i rozpoznać błąd (narzędzia debugujące opisuję w rozdziale 13.).

Rysunek 12.2.
Excel wyświetla komunikat o błędzie, gdy procedura usuwa obliczyć pierwiastek kwadratowy z liczby ujemnej



Większość programistów uważa, że komunikaty o błędach wyświetlane przez program Excel nie są szczególnie pomocne (na przykład *Invalid procedure call or argument*, błędne wywołanie procedury lub błędny argument). Aby poprawić powyższą procedurę, powinieneś umieć wybaczyć ten błąd i z gracją go obsłużyć. Inaczej mówiąc, musisz dodać trochę kodu do obsługi błędu.

Poniżej przedstawiam zmienioną wersję procedury EnterSquareRoot.

```
Sub EnterSquareRoot2()
    Dim Num As Double
    ' Poproś o liczbę
    Num = InputBox("Podaj liczbę")
    ' Sprawdź, czy liczba jest nieujemna
    If Num < 0 Then
        MsgBox "Musisz podać liczbę dodatnią."
        Exit Sub
    End If
    ' Umieść pierwiastek kwadratowy
    ActiveCell.Value = Sqr(Num)
End Sub
```

Struktura If-Then sprawdza wartość zmiennej Num. Jeżeli jest mniejsza od zera, procedura wyświetli okno z komunikatem zawierającym informację zrozumiałą dla człowieka. Potem kod kończy działanie za sprawą instrukcji Exit Sub, więc błąd wykonania nie ma szansy wystąpić.

Makro wciąż nie jest idealne

Teraz zmieniona procedura EnterSquareRoot jest już idealna, prawda? Niezupełnie. Spróbuj zamiast liczby wpisać tekst. Albo kliknij w oknie dialogowym przycisk Cancel. Obie czynności spowodują pojawienie się błędu (*Type mismatch*, niezgodność typów). Ta mała, prosta procedura wymaga dodatkowego kodu obsługi błędów.

Poniższy zmodyfikowany kod wykorzystuje funkcję IsNumeric do upewnienia się, że zmienna Num zawiera wartość liczbową. Jeżeli użytkownik nie wprowadzi liczby, procedura wyświetli odpowiedni komunikat i zakończy działanie. Zwrót również uwagi, że zmienna Num jest teraz zadeklarowana jako Variant. Gdyby była zadeklarowana jako Double, po wprowadzeniu przez użytkownika w oknie dialogowym wartości nienumerycznej kod zgłosiłby nieobsługiwany błąd.

```

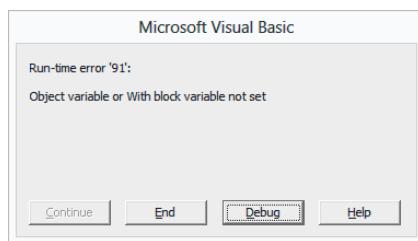
Sub EnterSquareRoot3()
    Dim Num As Variant
    ' Poproś o liczbę
    Num = InputBox("Podaj liczbę")
    ' Upewnij się, że Num zawiera liczbę
    If Not IsNumeric(Num) Then
        MsgBox "Musisz podać liczbę."
        Exit Sub
    End If
    ' Sprawdź, czy liczba jest nieujemna
    If Num < 0 Then
        MsgBox "Musisz podać liczbę dodatnią."
        Exit Sub
    End If
    ' Umieść pierwiastek kwadratowy
    ActiveCell.Value = Sqr(Num)
End Sub

```

Czy teraz makro jest idealne?

Teraz kod jest absolutnie idealny, prawda? Niezupełnie. Spróbuj uruchomić procedurę, gdy aktywnym arkuszem będzie wykres. Och, znowu błąd wykonania, tym razem jest to okropny numer 91 (zobacz rysunek 12.3). Ten błąd pojawia się, ponieważ nie istnieje aktywna komórka, jeżeli aktywny jest arkusz z wykresem lub wybrany jest obiekt inny niż zakres komórek.

Rysunek 12.3.
Uruchomienie procedury, gdy aktywny jest arkusz z wykresem, powoduje wystąpienie błędu



Kod w poniższym listingu wykorzystuje funkcję TypeName do upewnienia się, że wybranym obiektem jest zakres komórek. Jeżeli wybrany będzie jakikolwiek inny obiekt, procedura wyświetli odpowiedni komunikat i zakończy działanie.

```

Sub EnterSquareRoot4()
    Dim Num As Variant
    ' Upewnij się, że arkusz jest aktywny
    If TypeName(Selection) <> "Range" Then
        MsgBox "Wybierz komórkę do wpisania wyniku."
        Exit Sub
    End If
    ' Poproś o liczbę
    Num = InputBox("Podaj liczbę")
    ' Upewnij się, że Num zawiera liczbę

```

```

If Not IsNumeric(Num) Then
    MsgBox "Musisz podać liczbę."
    Exit Sub
End If
' Sprawdź, czy liczba jest nieujemna
If Num < 0 Then
    MsgBox "Musisz podać liczbę dodatnią."
    Exit Sub
End If
' Umieść pierwiastek kwadratowy
ActiveCell.Value = Sqr(Num)
End Sub

```

Rezygnacja z ideału

Ale teraz procedura po prostu musi być idealna. Zastanów się jeszcze, kolego.

Włącz ochronę arkusza (w tym celu użąd polecenia *RECENZJA/Zmiany/Chroń arkusz*) i uruchom kod. No tak, chroniony arkusz jest przyczyną innego błędu. A i tak prawdopodobnie nie uwzględniałem wszystkich przypadków, jakie mogą wystąpić. Czytaj dalej, aby poznać inny sposób obsługi błędów — nawet takich, których nie możesz wybaczyć.

Inny sposób obsługi błędów

W jaki sposób możesz określić i obsłużyć każdy możliwy błąd? Najczęściej nie możesz. Na szczęście, VBA oferuje inny sposób obsługi błędów.

Korekta procedury EnterSquareRoot

Przyjrzyj się poniższemu kodowi. Zmieniłem przykład z poprzedniej części, dodając uniwersalną instrukcję *On Error*, aby przechwycić wszystkie błędy, jak również sprawdzam, czy funkcja *InputBox* nie została przerwana.

Instrukcja On Error nie działa?

Jeżeli instrukcja *On Error* nie działa tak, jak powinna, powinieneś zmienić jedno z ustawień.

1. Otwórz edytor VBE.
2. Wybierz polecenie menu *Tools/Options*.
3. W oknie *Options* kliknij zakładkę **General**.
4. Sprawdź, czy nie jest zaznaczona opcja **Break on All Errors**.

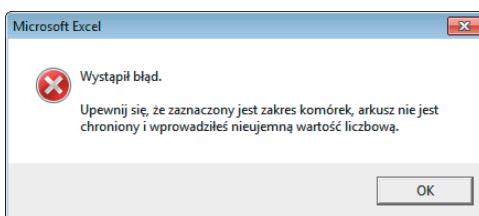
Jeżeli wybrana jest powyższa opcja, Excel ignoruje instrukcje *On Error*. Standardowo w grupie *Error Trapping* (przechwytywanie błędów) powinieneś wybrać opcję *Break on Unhandled Errors* (Przerywaj przy nieobsługiwanych błędach).

```
Sub EnterSquareRoot5()
    Dim Num As Variant
    Dim Msg As String
    ' Włącz obsługę błędów
    On Error GoTo BadEntry
    ' Poproś o liczbę
    Num = InputBox("Podaj liczbę")
    ' Wyjdź, jeżeli operacja zostanie przerwana
    If Num = "" Then Exit Sub
    ' Umieść pierwiastek kwadratowy
    ActiveCell.Value = Sqr(Num)
    Exit Sub

BadEntry:
    Msg = "Wystąpił błąd." & vbCrLf & vbCrLf
    Msg = Msg & "Upewnij się, że zaznaczony jest zakres komórek, "
    Msg = Msg & "arkusz nie jest chroniony"
    Msg = Msg & "i wprowadziłeś nieujemną wartość liczbową."
    MsgBox Msg, vbCritical
End Sub
```

Powyzszy kod przechwytuje *każdy* błąd wykonania. Skorygowana procedura EnterSquareRoot po przechwyceniu błędu wyświetli okno z komunikatem pokazanym na rysunku 12.4. Komunikat będzie zawierał opis najbardziej prawdopodobnej przyczyny błędu.

Rysunek 12.4.
Błąd wykonania procedury powoduje wyświetlenie dość pomocnego komunikatu



O instrukcji On Error

Użycie w kodzie VBA instrukcji On Error umożliwia ominięcie wbudowanych w programie Excel procedur obsługi błędów i zastosowanie własnych. W poprzednich przykładach błąd wykonania powodował przejście do instrukcji opatrzonej etykietą BadEntry. W efekcie zapobiegałeś pojawianiu się niejasnych komunikatów o błędach i mogłeś wyświetlać własne, bardziej przyjazne dla użytkownika.

Zauważ, że w podanym przykładzie instrukcja Exit Sub jest umieszczona tuż przed etykietą BadEntry. Ta instrukcja jest konieczna, ponieważ nie potrzebujesz wykonywać kodu obsługującego błąd, jeżeli taki nie wystąpi.



Obsługa błędów — szczegółowe informacje

Instrukcji On Error możesz użyć na trzy sposoby, wymienione w tabeli 12.1.

Tabela 12.1. Użycie instrukcji On Error

Składnia	Działanie
On Error GoTo etykieta	Po wykonaniu tej instrukcji VBA wznowia wykonywanie kodu od określonego wiersza. Po nazwie etykiety należy umieścić dwukropki, aby VBA mógł ją zidentyfikować.
On Error GoTo 0	Po wykonaniu tej instrukcji VBA przywraca swoją własną kontrolę błędów. Stosuj tę instrukcję po użyciu innych instrukcji On Error lub kiedy chcesz zrezygnować z obsługi błędów w procedurze.
On Error Resume Next	Po wykonaniu tej instrukcji VBA po prostu ignoriuje wszystkie błędy i wznowia wykonywanie kodu od następnej instrukcji.

Wznowianie wykonywania kodu po wystąpieniu błędu

W niektórych przypadkach wystąpienia błędu możesz jedynie potrzebować, aby kod poprawniekończył swoje działanie. Możesz na przykład wyświetlić komunikat opisujący błąd i zakończyć działanie procedury. (Procedura EnterSquareRoot5 wykorzystuje tę technikę). W innych przypadkach chciałbyś, aby wykonywanie kodu było w miarę możliwości wznowiane.

Aby wznowić działanie kodu po wystąpieniu błędu, musisz użyć instrukcji Resume. Kasuje ona stan błędu i wznowia wykonywanie kodu od zadanego miejsca. Instrukcji Resume możesz użyć na trzy sposoby, przedstawione w tabeli 12.2.

Tabela 12.2. Użycie instrukcji Resume

Składnia	Działanie
Resume	Wykonywanie kodu jest wznowiane od instrukcji, która spowodowała błąd. Stosuj tę składnię wtedy, gdy Twoja procedura obsługi błędów usunęła problem i można kontynuować wykonywanie kodu.
Resume Next	Wykonywanie kodu jest wznowiane od instrukcji następującej bezpośrednio po instrukcji, która spowodowała błąd. Błąd jest wówczas ignorowany.
Resume etykieta	Wykonywanie kodu jest wznowiane od zadanej etykiety.

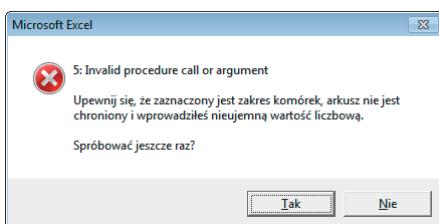
W poniższym przykładzie po wystąpieniu błędu jest wykonywana instrukcja Resume.

```
Sub EnterSquareRoot6()
    Dim Num As Variant
    Dim Msg As String
    Dim Ans As Integer
```

```
TryAgain:  
    ' Włącz obsługę błędów  
    On Error GoTo BadEntry  
    ' Poproś o liczbę  
    Num = InputBox("Podaj liczbę")  
    If Num = "" Then Exit Sub  
    ' Umieść pierwiastek kwadratowy  
    ActiveCell.Value = Sqr(Num)  
    Exit Sub  
  
BadEntry:  
    Msg = Err.Number & ": " & Error(Err.Number)  
    Msg = Msg & vbCrLf & vbCrLf  
    Msg = Msg & "Upewnij się, że zaznaczony jest zakres komórek, "  
    Msg = Msg & "arkusz nie jest chroniony."  
    Msg = Msg & "i wprowadziłeś nieujemną wartość liczbową."  
    Msg = Msg & vbCrLf & vbCrLf & "Spróbować jeszcze raz?"  
    Ans = MsgBox(Msg, vbYesNo + vbCritical)  
    If Ans = vbYes Then Resume TryAgain  
End Sub
```

Powyższa procedura zawiera drugą etykietę, czyli TryAgain. Gdy wystąpi błąd, wykonywanie kodu będzie wznowiane od etykiety BadEntry i pojawi się komunikat pokazany na rysunku 12.5. Jeżeli użytkownik kliknie przycisk *Tak*, zadziała instrukcja Resume i wykonywanie kodu zostanie przekierowane do etykiety TryAgain. Jeżeli natomiast użytkownik kliknie przycisk *Nie*, procedura zakończy działanie.

Rysunek 12.5.
Jeżeli wystąpi błąd, użytkownik może zadecydować, czy spróbować jeszcze raz



Zauważ, że komunikat o błędzie zawiera również numer błędu oraz jego „oficjalny” opis. Umieściłem go tutaj, ponieważ piszę o nim później w punkcie „Rozpoznawanie określonych błędów”.

Pamiętaj o tym, że instrukcja Resume kasuje stan błędu przed wznowieniem wykonywania kodu. Aby sprawdzić, co to oznacza, spróbuj zamienić przedostatnią instrukcję z powyższego kodu na następującą instrukcję:

```
If Ans = vbYes Then GoTo TryAgain
```

Kod nie będzie działał prawidłowo, jeżeli zamiast Resume użyjesz GoTo. Aby się o tym przekonać, wprowadź ujemną liczbę. W oknie z komunikatem kliknij *Tak* i ponownie wprowadź liczbę ujemną. Drugi błąd nie zostanie przechwycony, ponieważ nie został skasowany początkowy stan błędu.

Przykład jest dostępny na stronie internetowej książki.



Obsługa błędów w pigułce

Aby ułatwić Ci całą sprawę związaną z obsługą błędów, przygotowałem prowizoryczne podsumowanie. Blok kodu obsługi błędu ma następujące cechy.

- ✓ Rozpoczyna się od etykiety określonej w instrukcji On Error.
- ✓ Powinien być wykonywany w Twoim makro tylko wtedy, gdy wystąpi błąd. Oznacza to, że tuż przed etykietą musisz użyć instrukcji Exit Sub lub Exit Function.
- ✓ Może wymagać użycia instrukcji Resume. Jeżeli nie chcesz przerywać wykonywania procedury po wystąpieniu błędu, przed powrotem do głównego kodu musisz wykonać instrukcję Resume.

Kiedy ignorować błędy?

W niektórych przypadkach ignorowanie błędów jest całkowicie dopuszczalne. Stosuje się wtedy instrukcję On Error Resume Next.

Poniższy kod przegląda wszystkie komórki w wybranym zakresie i zamienia zawartość każdej z nich na pierwiastek kwadratowy. Jeżeli jakąś komórkę zawiera tekst lub liczbę ujemną, procedura wyświetla komunikat o błędzie.

```
Sub SelectionSqrt()
    Dim cell As Range
    If TypeName(Selection) <> "Range" Then Exit Sub
    For Each cell In Selection
        cell.Value = Sqr(cell.Value)
    Next cell
End Sub
```

W takim przypadku być może wołalibyś po prostu pominąć wszystkie komórki, których zawartość nie może być użyta do obliczenia pierwiastka kwadratowego. Z wykorzystaniem struktury If-Then możesz utworzyć kod sprawdzający możliwość wystąpienia błędu, ale możesz zastosować lepsze (i prostsze) rozwiązanie, po prostu ignorując błędy, które wystąpią podczas wykonywania procedury.

Poniższy kod realizuje to zadanie za pomocą instrukcji On Error Resume Next.

```
Sub SelectionSqrt()
    Dim cell As Range
    If TypeName(Selection) <> "Range" Then Exit Sub
    On Error Resume Next
    For Each cell In Selection
        cell.Value = Sqr(cell.Value)
    Next cell
End Sub
```

Ogólnie rzecz biorąc, instrukcję On Error Resume Next możesz stosować wtedy, gdy uważasz, że błędy są niegroźne i nie powodują niekorzystnych skutków.

Rozpoznawanie określonych błędów

Nie wszystkie błędy są takie same. Niektóre są poważne, inne mniej. Chociaż możesz ignorować błędy, które — według Ciebie — nie mają negatywnych skutków, to musisz obsługiwać inne, bardziej istotne. W niektórych przypadkach musisz rozpoznać błąd, który się pojawi.

Każdy błąd ma swój oficjalny numer. Gdy wystąpi, Excel przechowuje jego numer w obiekcie Err typu Error. Właściwość Number tego obiektu zawiera numer błędu, natomiast Description jego opis. Przykładowo poniższa instrukcja wyświetla numer i opis błędu oddzielone dwukropkiem.

```
MsgBox Err.Number & " : " & Err.Description
```

Przykład działania tej instrukcji był pokazany wcześniej na rysunku 12.5. Pamiętaj jednak, że komunikaty programu Excel o błędach nie zawsze są przydatne — ale to już wiesz.

Poniższa procedura pokazuje, jak rozpoznać błąd, który wystąpił. W tym przypadku możesz spokojnie zignorować błędy spowodowane próbą obliczenia pierwiastka kwadratowego z liczby ujemnej (czyli błąd nr 5) lub z wartością innej niż liczba (błąd nr 13). Z drugiej strony jednak, chciałbyś poinformować użytkownika, że arkusz jest chroniony i wybrany zakres zawiera jedną lub kilka zablokowanych komórek. (Inaczej użytkownik może pomyśleć, że makro zadziałało, chociaż w rzeczywistości tak się nie stało). Próba wpisania wartości do zablokowanej komórki skutkuje wystąpieniem błędu nr 1004.

```
Sub SelectionSqrt()
    Dim cell As Range
    Dim ErrMsg As String
    If TypeName(Selection) <> "Range" Then Exit Sub
    On Error GoTo ErrorHandler
    For Each cell In Selection
        cell.Value = Sqr(cell.Value)
    Next cell
    Exit Sub

ErrorHandler:
    Select Case Err.Number
        Case 5 ' Liczba ujemna
            Resume Next
        Case 13 ' Niezgodność typów
            Resume Next
        Case 1004 ' Zablokowana komórka, chroniony arkusz
            MsgBox "Komórka jest zablokowana. Spróbuj ponownie.", vbCritical,
cell.Address
            Exit Sub
        Case Else
            ErrMsg = Error(Err.Number)
            MsgBox "BŁĄD: " & ErrMsg, vbCritical, cell.Address
            Exit Sub
    End Select
End Sub
```

Gdy wystąpi błąd wykonania, realizacja kodu zostanie przeniesiona do etykiety ErrorHandler. Struktura Select Case (strukturę tę omówiłem w rozdziale 10.) sprawdza numery trzech najczęstszych błędów. Jeżeli numer błędu jest równy 5 lub 13, wykonywanie kodu jest wznowiane od następnej instrukcji. (Innymi słowy, błąd jest ignorowany). Jeżeli natomiast numer błędu jest równy 1004, kod informuje użytkownika o problemie i kończy działanie. Ostatni przypadek obejmuje wszystkie niewybaczane wcześniej błędy i wyświetlany jest odpowiedni komunikat.

Zamierzony błąd

Czasem błąd może przynieść korzyść. Wyobraź sobie na przykład makro, które działa tylko wtedy, gdy otwarty jest określony skoroszyt. Jak możesz sprawdzić, czy ten skoroszyt jest otwarty? Jednym ze sposobów jest napisanie kodu, który przegląda kolekcję Workbooks i sprawdza, czy znajduje się w niej żądany skoroszyt.

Ale jest prostszy sposób: utworzenie funkcji ogólnego przeznaczenia, przyjmującej jeden argument (nazwę skoroszytu) i zwracającej wartość True, jeżeli skoroszyt jest otwarty, lub False, jeżeli nie jest.

Oto taka funkcja.

```
Function WorkbookIsOpen(book As String) As Boolean
    Dim WBName As String
    On Error GoTo NotOpen
    WBName = Workbooks(book).Name
    WorkbookIsOpen = True
    Exit Function
NotOpen:
    WorkbookIsOpen = False
End Function
```

Funkcja wykorzystuje fakt, że Excel zgłasza błąd, gdy odwołasz się do skoroszytu, który nie jest otwarty. Poniższa instrukcja spowoduje wystąpienie błędu, jeżeli nie będzie otwarty skoroszyt *MójZeszyt.xlsx*.

```
WBName = Workbooks("MójZeszyt.xlsx").Name
```

Instrukcja On Error użyta w funkcji WorkbookIsOpen zleca VBA wznowienie wykonywania makra od etykiety NotOpen, gdy wystąpi błąd. Błąd w takim przypadku oznacza, że skoroszyt nie jest otwarty, więc funkcja zwraca wartość False. Jeżeli skoroszyt jest otwarty, błąd nie wystąpi i funkcja zwróci wartość True.

Poniżej przedstawiam inną wersję funkcji WorkbookIsOpen. Jest w niej użyta instrukcja On Error Resume Next ignorująca błąd. Natomiast kod sprawdza właściwość Number obiektu Err. Jeżeli jest równa zero, oznacza to, że błąd nie wystąpił, a to oznacza, że skoroszyt jest otwarty. Jeżeli jej wartość jest różna od zera, to znaczy, że błąd wystąpił (czyli skoroszyt nie jest otwarty).

```
Function WorkbookIsOpen(book) As Boolean
    Dim WBName As String
    On Error Resume Next
    WBName = Workbooks(book).Name
    If Err.Number = 0 Then WorkbookIsOpen = True _
        Else WorkbookIsOpen = False
End Function
```

W poniższym przykładzie pokazuję, jak wykorzystać tę funkcję w procedurze Sub.

```
Sub UpdatePrices()
    If Not WorkbookIsOpen("Ceny.xlsx") Then
        MsgBox "Otwórz najpierw skoroszyt Ceny.xlsx!"
        Exit Sub
    End If
    ' [Tutaj jest pozostały kod]
End Sub
```

Procedura `UpdatePrices` (musi znajdować się w tym samym skoroszycie, co funkcja `WorkbookIsOpen`) wywołuje funkcję `WorkbookIsOpen` i przekazuje w jej argumencie nazwę skoroszytu (`Ceny.xlsx`). Funkcja zwraca wartość `True` lub `False`. Dzięki temu, jeżeli ten skoroszyt nie jest otwarty, procedura informuje użytkownika o tym fakcie. Jeżeli natomiast skoroszyt jest otwarty, makro kontynuuje działanie.

Obsługa błędów może być trudną kwestią — może przecież wystąpić wiele różnych błędów, a nie jesteś w stanie wybaczać wszystkich. Generalnie, w miarę możliwości powinieneś przechwytywać błędy i je korygować, zanim zainterweniuje Excel. Tworzenie kodu skutecznie przechwytyującego błędy wymaga szerokiej wiedzy o programie Excel i dogłębnego zrozumienia istoty obsługi błędów w VBA. W kolejnych rozdziałach znajdziesz więcej przykładów obsługi błędów.

Rozdział 13

Dezynsekcja kodu, czyli jak walczyć z pluskwami

W tym rozdziale:

- ▶ dowiesz się, czym są pluskwy i dlaczego powinieneś z nimi walczyć,
- ▶ poznasz rodzaje pluskiew, z jakimi możesz się spotkać w swoich programach,
- ▶ poznasz techniki „odpluskwniania” kodu,
- ▶ nauczysz się korzystać z wbudowanych narzędzi VBA, wspomagających walkę z pluskwami.

Jeżeli Twoim pierwszym skojarzeniem po usłyszeniu słowa „pluskwa” jest obraz małego, krwiożerczego insektu, z całą pewnością lektura tego rozdziału powinna to skojarzenie definitelywnie zmienić. Krótko mówiąc, w świecie programistów „pluskwa” to po prostu błąd w kodzie makra, aplikacji czy dowolnego innego programu. W tym rozdziale zajmę się właśnie takimi „komputerowymi pluskwami” i napiszę, jak możesz je lokalizować, identyfikować i usuwać z kodu własnych programów.

Rodzaje pluskiew

Witam w akademii odpluskwniania. Jak już zdążyłeś się zorientować, określenie „pluskwa w kodzie programu” odnosi się do sytuacji, w której pojawia się jakiś problem z funkcjonowaniem oprogramowania. Innymi słowy, jeżeli program nie działa tak, jak tego oczekiwaneś, to znaczy, że ma pluskwy... Nikt nie zaprzeczy, że praktycznie w każdym odpowiednio rozbudowanym (i nie tylko) oprogramowaniu znajdują się pluskwy (błędy). Możemy spokojnie założyć, że populacja pluskiew Excela liczy sobie setki (jeżeli nie tysiące) sztuk. Na szczęście, znakomita większość tych błędów jest głęboko ukryta i daje o sobie znać tylko w bardzo specyficznych okolicznościach.

Jeżeli w języku VBA piszesz swoje małe i duże programy (i nie mówię tutaj o trywialnych, kilkuwierszowych procedurkach jednorazowego użytku), praktycznie zawsze w ich kodzie znajdą się jakieś błędy. Takie są po prostu realia sztuki programowania i nie ma to nic wspólnego z Twoimi umiejętnościami. Pluskwy można podzielić na kilka najważniejszych kategorii.

- ✓ **Błędy logiczne w kodzie programu.** Zazwyczaj możesz ich uniknąć, starannie analizując problem, którego rozwiążaniem zajmuje się Twój program.
- ✓ **Błędy związane z nieprawidłowym kontekstem operacji.** Taki rodzaj błędów pojawia się, kiedy próbujesz wykonać daną operację w niewłaściwym momencie. Jeśli na przykład Twój program usiłuje zapisać dane do komórek arkusza w sytuacji, kiedy aktywnym arkuszem jest arkusz wykresu (który nie posiada żadnych komórek).
- ✓ **Błędy związane z warunkami granicznymi.** Błędy graniczne występują zazwyczaj w sytuacjach, kiedy program napotyka dane, których pojawiение się nie zostało przewidziane, takie jak bardzo wielkie bądź bardzo małe liczby.
- ✓ **Błędy związane z nieprawidłowym typem danych.** Takie błędy można spotkać w sytuacji, kiedy program próbuje wykonywać określone operacje na danych nieprawidłowego typu, takie jak na przykład próba obliczenia wartości pierwiastka kwadratowego z ciągu znaków alfanumerycznych.
- ✓ **Błędy związane z niewłaściwą wersją programu.** Tego typu błędy są wynikiem niekompatybilności różnych wersji Excela. Przykładowo przygotowałaś skoroszyt zawierający makra w Excelu 2013, po czym okazało się, że nie działają one w Excelu 2003. Zazwyczaj można zapobiegać powstawaniu się takich błędów poprzez unikanie stosowania funkcji i mechanizmów specyficznych dla danej wersji Excela. Bardzo często stosowanym rozwiążaniem jest pisanie aplikacji w najstarszej wersji Excela, z której jeszcze korzystają potencjalni użytkownicy aplikacji. Aby jednak mieć pewność, że Twоя aplikacja będzie działać poprawnie, powinieneś przetestować ją na wszystkich wersjach Excela, na których będzie używana.
- ✓ **Błędy, które pozostają poza Twoją kontrolą.** Takie błędy są chyba zdedykowanie najbardziej frustrujące. Przykładem takiego błędu może być sytuacja, w której firma Microsoft wypuszcza na rynek nową wersję Excela i przy okazji dokonuje niewielkiej, nieudokumentowanej zmiany w sposobie implementacji takiej czy innej funkcji, co w efekcie powoduje, że Twój program „z nieznanych przyczyn” wylatuje w kosmos. Znane są przypadki, że tego typu błędy były powodowane przez zwykłe aktualizacje i łatki aplikacji.

Debugowanie (ang. *debugging*), czyli, inaczej mówiąc, *odpluskwanie* (obu tych terminów możesz używać zamiennie), to proces identyfikowania i naprawiania błędów w kodzie programu. Zdobycie umiejętności wyszukiwania i naprawiania błędów nie jest łatwe i z pewnością wymaga sporo czasu, stąd nie powinieneś się przejmować, jeżeli początkowo będzie Ci to sprawiać kłopoty i trudności.

Niezmiernie ważną sprawą jest dokładne zrozumienie różnicy pomiędzy *pluskwami* w kodzie programu a *błędami składniowymi* polecień tegoż kodu. Błąd składniowy to po prostu pomyłka w składni polecenia danego języka programowania. Przykładami takich błędów mogą być literówki w słowach kluczowych, pominięcie polecenia *Next* w pętli *For-Next*, brak przecinka lub średnika w określonym miejscu kodu czy brak lub nadmiar nawiasów domykających. Wszystkie błędy składniowe musisz zidentyfikować i poprawić, zanim w ogóle będziesz mógł po raz pierwszy uruchomić swoją procedurę. Pluskwy w kodzie programu to „istoty” zdedykowanie bardziej subtelne. Procedurę zawierającą pluskwę możesz swobodnie uruchomić, tyle tylko, że nie będzie ona działała zgodnie z Twoimi oczekiwaniami.

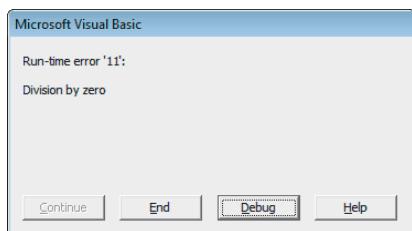


Podstawy entomologii, czyli jak zidentyfikować pluskwy

Zanim będziesz mógł rozpoczęć jakiekolwiek odpluskwianie swojego programu, musisz określić, czy rzeczywiście zagnieździła się w nim jakaś pluskwa. Obecność takiego „insektu” w kodzie możesz zazwyczaj stwierdzić na podstawie tego, że Twój program nie działa w sposób, w jaki powinieneś (ta książka jest wprost przepchniona takimi ogólnikami, nieprawdaż?). W końcu, będąc autorem programu, możesz łatwo taki fakt stwierdzić.

Pluskwy często (ale nie zawsze) wychodzą na jaw, kiedy Excel wyświetla na ekranie komunikat o wystąpieniu błędu realizacji programu. Na rysunku 13.1 przedstawiam przykład takiego komunikatu. Zwróć uwagę na fakt, że w oknie komunikatu znajduje się przycisk *Debug*. Więcej szczegółowych informacji na ten temat znajdziesz w podrozdziale zatytułowanym „Kilka słów o debuggerze”, dalej w tym rozdziale.

Rysunek 13.1.
Pojawienie się takiego komunikatu zazwyczaj oznacza, że kod VBA zawiera pluskwy



Kluczową sprawą, doskonale znaną wszystkim programistom, jest fakt, że pluskwy najczęściej „objawiają się”, kiedy się ich najmniej spodziewałeś. Kiedy na przykład Twoje makro doskonale działa z jednym zestawem danych, nigdy nie powinieneś automatycznie zakładać, że równie dobrze poradzi sobie ze wszystkimi innymi zestawami danych.

Najlepszym podejściem do problemu odpluskwiania kodu jest gruntowne testowanie działania programu w warunkach maksymalnie zbliżonych do warunków „produkcyjnych”. Pamiętaj — zmian w skoroszycie wprowadzonych przez makra VBA nie można cofnąć, stąd testowanie powinieneś zawsze przeprowadzać na kopiiach zapasowych skoroszytu. Sam zazwyczaj tworzę kilka kopii takich plików w folderze tymczasowym i dopiero na nich przeprowadzam testy.

Metody i techniki walki z pluskwami

W tym podrozdziale omówię cztery podstawowe metody odpluskwiania kodu VBA aplikacji przeznaczonych dla Excela. Oto one.

- ✓ Przeglądanie kodu VBA.
- ✓ Umieszczanie funkcji MsgBox w kluczowych miejscach kodu.

- ✓ Umieszczanie polecień Debug.Print w kluczowych miejscach kodu.
- ✓ Korzystanie z wbudowanych narzędzi Excela wspomagających odpluskwanie.

Przeglądanie kodu VBA

Prawdopodobnie najbardziej bezpośrednią metodą wyszukiwania pluskiew jest dokładne przeglądanie kodu programu i wyszukiwanie miejsc, które potencjalnie mogą sprawiać problemy. Oczywiście, taka metoda, wbrew pozorom bardzo skuteczna, wymaga od sprawdzającego dużej wiedzy i doświadczenia. Innymi słowy, musisz dobrze wiedzieć, co robisz. Jeżeli będziesz miał trochę szczęścia, doświadczonym okiem entomologa szybko zlokalizujesz pluskwę, a potem palniesz się otwartą dlonią w czoło i powiesz do siebie „Ty pacanie!...”. Kiedy już ból i opuchlizna czoła przeminą, będziesz mógł poprawić błąd w kodzie i naprawić problem.

Zauważ, że napisałem „jeżeli będziesz miał trochę szczęścia”. W praktyce tak to właśnie wygląda, ponieważ najczęściej błędy w kodzie odkrywasz, kiedy jest godzina 2 nad ranem, przesiedziałeś już nad pisaniem programu co najmniej 8 godzin i udaje Ci się utrzymać otwarte oczy już tylko dzięki sile woli wspomaganej hektolitrami wlanego w siebie kawy. Zazwyczaj w takich okolicznościach masz dużo szczęścia, jeżeli w ogóle jesteś w stanie dostrzec na ekranie monitora jakikolwiek kod, nie mówiąc już o ukrytych w nim błędach. Jak widać, nie powinieneś być zatem zaskoczony, jeżeli przeglądanie kodu po raz n-ty nie przynosi żadnego rezultatu i w żaden sposób nie przyczynia się do zlikwidowania populacji pluskiew, jakie zagnieździły się w Twoim programie.

Umieszczanie funkcji MsgBox w kluczowych miejscach kodu

Bardzo popularny problem, z którym codziennie borykają się setki, jeżeli nie tysiące programistów, polega na tym, że jedna zmienna bądź kilka zmiennych w programie przyjmuje nie takie wartości, jak mógłbyś tego oczekiwąć. W takich sytuacjach bardzo pomocną techniką może być monitorowanie wartości zmiennych podczas działania programu. Jednym ze sposobów realizacji takiego pomysłu może być tymczasowe wstawianie funkcji MsgBox w kluczowych miejscach kodu programu. Jeżeli na przykład masz zmienną o nazwie *CellCount*, możesz w odpowiednim miejscu programu umieścić następujące polecenie.

```
MsgBox CellCount
```

Dzięki temu, po uruchomieniu programu na ekranie pojawi się okno dialogowe, w którym wyświetlona będzie wartość zmiennej *CellCount*.

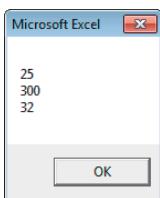
Bardzo często dobrym rozwiązaniem będzie wyświetlanie w jednym oknie dialogowym wartości dwóch lub więcej zmiennych. Polecenie przedstawione poniżej wyświetla na ekranie okno dialogowe zawierające wartości zmiennych *LoopIndex* oraz *CellCount*, oddzielone od siebie spacją.

```
MsgBox LoopIndex & " " & CellCount
```

Zwróć uwagę, że za pomocą operatora konkatenacji (&) łączymy ze sobą nazwy dwóch zmiennych, wstawiając pomiędzy ich wartości znak spacji. W przeciwnym przypadku funkcja MsgBox automatycznie połączyłaby wartości tych dwóch zmiennych w jeden ciąg znaków bez separującej spacji, co w efekcie wyglądałoby jak jedna wartość. Zamiast spacji do separowania poszczególnych wartości możesz użyć wbudowanej stałej Excela, vbNewLine, która powoduje wyświetlanie następującego po niej ciągu znaków od nowego wiersza. Polecenie przedstawione poniżej wyświetla w oknie komunikatu wartości trzech zmiennych, przy czym każda z nich wyświetlana jest w osobnym wierszu (zobacz rysunek 13.2).

```
MsgBox LoopIndex & vbNewLine & CellCount & vbNewLine & MyVal
```

Rysunek 13.2.
Zastosowanie
okna komuni-
katu do wy-
świetlania wartości
trzech zmiennych

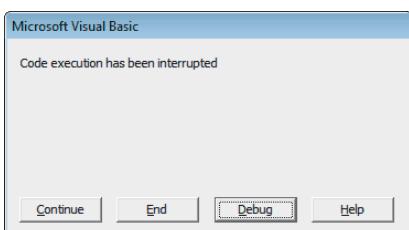


Zastosowanie takiej techniki nie jest ograniczone do monitorowania wartości zmiennych. Funkcji MsgBox możesz używać do wyświetlania podczas działania programu użytecznych informacji dowolnego rodzaju. Jeśli na przykład Twoja procedura przechodzi w pętli przez serię arkuszy w skoroszycie, możesz użyć polecenia przedstawionego poniżej do wyświetlania nazwy i typu aktywnego arkusza.

```
MsgBox ActiveSheet.Name & " " & TypeName(ActiveSheet)
```

Jeżeli w oknie komunikatu pojawi się jakaś nieoczekiwana wartość, naciśnij kombinację klawiszy **Ctrl+Break** i na ekranie pojawi się okno dialogowe z komunikatem *Code execution has been interrupted* (działanie programu zostało przerwane). Jak widać na rysunku 13.3, do wyboru masz cztery opcje.

Rysunek 13.3.
Naciśnięcie
kombinacji
klawiszy
Ctrl+Break
przerwia dzia-
łanie programu



- ✓ Naciśnij przycisk *Continue* i program wznowi działanie w miejscu, w którym zostało przerwane.
- ✓ Naciśnij przycisk *End* i program zakończy działanie.
- ✓ Naciśnij przycisk *Debug* i edytor VBE przejdzie do trybu odpluskwiania (o którym powiem więcej już za chwilę, w podrozdziale „Kilka słów o debuggerze”).

ZAPAMIĘTAJ

✓ Naciśnij przycisk *Help* i na ekranie pojawi się okno pomocy systemowej, z którego dowiesz się, że właśnie przerwałeś realizację programu, naciskając kombinację klawiszy *Ctrl+Break*. Krótko mówiąc, nie będzie to zbyt pomocne.

Podczas odpluskwiwania kodu możesz używać funkcji *MsgBox* tak często, jak będzie to potrzebne. Pamiętaj jednak, aby po zakończeniu procesu wyszukiwania i naprawy błędów usunąć wszystkie niepotrzebne wywołania tej funkcji.

Umieszczanie polecenia *Debug.Print* w kluczowych miejscach kodu

Zamiast korzystać z funkcji *MsgBox*, możesz w kluczowych miejscach kodu tymczasowo wstawiać jedno lub więcej poleceń *Debug.Print*. Za pomocą tego polecenia możesz wyświetlać w oknie *Immediate* edytora VBE wartości wybranych zmiennych. Poniżej przedstawiam przykład zastosowania polecenia, które wyświetla wartości trzech zmiennych.

```
Debug.Print LoopIndex, CellCount, MyVal
```

ZAPAMIĘTAJ

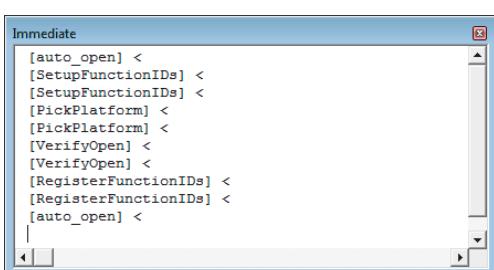
Zwróc uwagę, że nazwy poszczególnych zmiennych są oddzielone przecinkami. Za pomocą jednego polecenia *Debug.Print* możesz wyświetlić wartości dowolnej liczby zmiennych.

Polecenie *Debug.Print* wysyła wyniki swojego działania do okna *Immediate* edytora VBE. Dzieje się tak nawet wtedy, kiedy okno *Immediate* nie jest widoczne na ekranie. Jeśli trzeba, możesz je przywołać, naciskając kombinację klawiszy *Ctrl+G*.

W przeciwieństwie do funkcji *MsgBox*, polecenie *Debug.Print* nie zatrzymuje działania programu, aby zatem wiedzieć, co się w danej chwili dzieje, powinieneś bacznie przyglądać się zawartości okna *Immediate*.

Pamiętaj jednak, aby po zakończeniu procesu wyszukiwania i naprawy błędów usunąć z kodu programu wszystkie niepotrzebne polecenia *Debug.Print*. Warto zauważyć, że nawet tak duże firmy jak Microsoft czasami zapominają usunąć takie polecenia z kodu finalnych wersji swoich aplikacji. Przykładowo w kilku poprzednich wersjach programu Excel za każdym razem, kiedy ładowałesz dodatek *Analysis ToolPak*, w oknie *Immediate* pojawiało się kilka tajemniczych komunikatów (zobacz rysunek 13.4). Ten „problem” został finalnie rozwiązany dopiero w Excelu 2007.

Rysunek 13.4.
Nawet zawsze-
dowi programiści
czasami zapo-
minają o usu-
nięciu polecen-
ia *Debug.Print*
z finalnych wer-
sji programu



Korzystanie z wbudowanych narzędzi Excela wspomagających odpluskwanie kodu VBA

Projektanci Excela z pewnością dobrze wiedzą, czym są pluskwy. Zapewne właśnie dzięki temu Excel posiada szereg wbudowanych narzędzi wspomagających odpluskwanie kodu VBA. Kolejny podrozdział poświęciłem właśnie tym narzędziom.

Kilka słów o debuggerze

W tym podrozdziale omówię kilka najważniejszych zagadnień związanych z używaniem wbudowanych narzędzi Excela wspomagających odpluskwanie kodu VBA. Należy przy tym powiedzieć, że możliwości wspomnianych narzędzi są o wiele potężniejsze niż techniki opisywane w tym podrozdziale. Pamiętaj jednak, że w parze z potężnymi możliwościami idzie odpowiedzialność, a samo korzystanie z tych narzędzi wymaga pewnych wstępnych przygotowań.

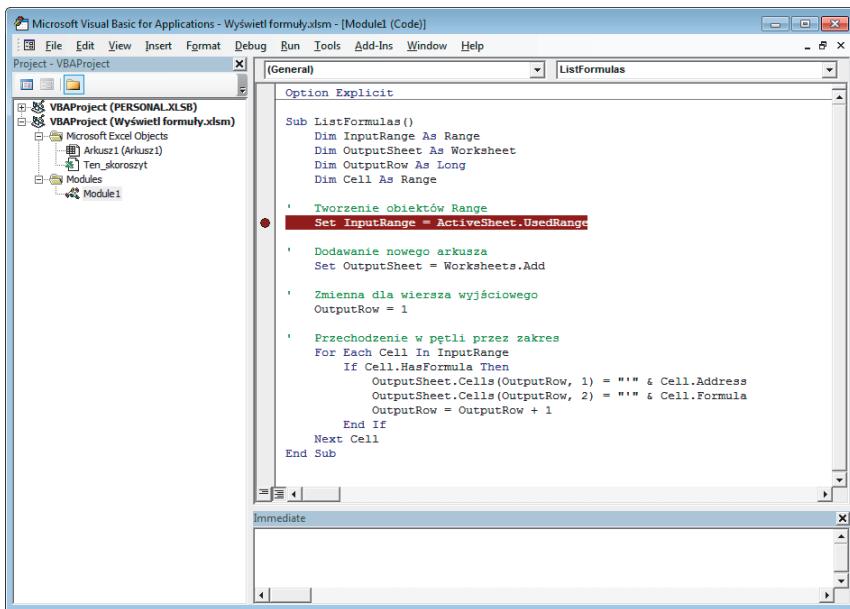
Ustawianie punktów przerwań w kodzie programu

Nieco wcześniej w tym rozdziale opisałem zastosowanie funkcji MsgBox do monitorowania wartości zmiennych podczas działania programu. W praktyce wyświetlanie okna komunikatu powoduje zatrzymanie realizacji programu, która może zostać wznowiona po naciśnięciu przez użytkownika przycisku OK.

A czy nie byłoby fajnie, jeżeli mógłbyś zatrzymać działanie programu w dowolnie wybranym miejscu, sprawdzić wartości dowolnych zmiennych i następnie wznowić działanie programu? Dokładnie to możesz zrobić poprzez ustawienie w kodzie programu VBA *punktu przerwania* (ang. *breakpoint*). Możesz to zrobić na co najmniej kilka sposobów.

- ✓ Wstaw kurSOR do wiersza polecenia, w którym chcesz ustawić punkt przerwania, i naciśnij klawisz F9.
- ✓ Kliknij lewym przyciskiem myszy szary margines, znajdujący się po lewej stronie wiersza polecenia, w którym chcesz ustawić punkt przerwania.
- ✓ Ustaw kurSOR w wierszu polecenia, w którym chcesz ustawić punkt przerwania, i z menu głównego edytora VBE wybierz polecenie *Debug/Toggle Breakpoint*.
- ✓ Kliknij wybrane polecenie prawym przyciskiem myszy i z menu podręcznego wybierz polecenie *Toggle/Breakpoint*.

Wygląd kodu VBA po ustawienia punktu przerwania został przedstawiony na rysunku 13.5. Jak widać, Excel podświetla wiersz zawierający punkt przerwania, dzięki czemu możesz go łatwo odnaleźć. Oprócz podświetlenia, z lewej strony wiersza na szarym marginesie pojawią się duża kropka.



Rysunek 13.5.
Podświetlony
wiersz oznacza
miejsce wsta-
wienia punktu
przerwania

Program po uruchomieniu jest normalnie realizowany, aż do momentu napotkania wiersza z ustawionym punktem przerwania; przed wykonaniem tego wiersza Excel przechodzi do działania w trybie *Break*, co zostaje wyróżnione pojawiением się literatu *[break]* na pasku tytułowym edytora VBE. Aby zakończyć pracę w trybie *Break* i wznowić działanie programu, naciśnij klawisz *F5* lub kliknij przycisk *Run Sub/UserForm*, znajdujący się na pasku narzędzi edytora VBE. Więcej szczegółowych informacji na ten temat znajdziesz w podpunkcie „Krokowe wykonywanie programu”, dalej w tym rozdziale.



Aby szybko usunąć wybrany punkt przerwania, kliknij lewym przyciskiem myszy kropkę (a właściwie kropę), znajdująca się z lewej strony wiersza na szarym marginesie lub ustaw kursor w wierszu punktu przerwania i naciśnij klawisz *F9*. Aby szybko usunąć wszystkie punkty przerwania z całego modułu kodu VBA, naciśnij kombinację klawiszy *Ctrl+Shift+F9*.

W języku VBA istnieje również słowo kluczowe, którego użycie w kodzie programu wymusza przejście Excela do trybu *Break*. Oto ono.

Stop

Kiedy podczas realizacji programu zostanie napotkane to słowo kluczowe, działanie programu zostaje przerwane i edytor VBE przechodzi do trybu *Break*.

Czym jest tryb *Break*? Możesz sobie go wyobrazić jako swego rodzaju zatrzymaną animację. Kod VBA przestaje działać, a bieżące polecenie zostaje wyróżnione żółtym kolorem tła wiersza. W trybie *Break* możesz:

- ✓ wpisywać polecenia VBA w oknie *Immediate* (więcej szczegółowych informacji na ten temat znajdziesz w kolejnym podrozdziale),

- ✓ naciskać klawisz *F8* w celu krokowego wykonywania kolejnych poleceń programu i sprawdzania różnych jego elementów pomiędzy kolejnymi poleceniami,
- ✓ ustawiać wskaźnik myszy nad poszczególnymi zmiennymi, aby wyświetlić ich wartość w małym okienku podpowiedzi ekranowej,
- ✓ pominąć jedno lub więcej poleceń i wznowić realizację programu od nowego miejsca (możesz nawet cofnąć się o kilka polecień wstecz),
- ✓ modyfikować wiersz polecenia i następnie wznowić realizację programu.



Na rysunku 13.6 przedstawiam kilka typowych operacji podejmowanych podczas odpluskwiania kodu. Najpierw ustawiłem punkt przerwania (zobacz podświetlony wiersz z wielką kropką z lewej strony), uruchomiłem program i następnie używałem klawisza *F8* do krokowego wykonywania kolejnych poleceń programu (zwróć uwagę na strzałkę wskazującą aktualnie wykonywane polecenie). Później użyłem okna *Immediate* do sprawdzenia kilku rzeczy i ustawiłem wskaźnik myszy na zmienną *OutputRow*, aby sprawdzić jej wartość.

 A screenshot of the Microsoft Visual Basic for Applications (VBE) interface in Break mode. The window title is "Microsoft Visual Basic for Applications - Wyświetl formuły.xlsm [break] - [Module1 (Code)]". The menu bar includes File, Edit, View, Insert, Format, Debug, Run, Tools, Add-Ins, Window, Help. The toolbar has icons for New, Open, Save, Print, and others. The left pane shows the Project Explorer with "VBAProject (PERSONAL.XLSB)" and "VBAProject (Wyświetl formuły.xlsm)" expanded, showing "Microsoft Excel Objects" like "Arkusz1 (Arkusz1)", "Arkusz2 (Arkusz2)", and "Ten_skoroszyt". Below that is "Modules" with "Module1" selected. The main code editor pane contains the following VBA code:


```

Sub ListFormulas()
    Dim InputRange As Range
    Dim OutputSheet As Worksheet
    Dim OutputRow As Long
    Dim Cell As Range

    ' Tworzenie obiektów Range
    Set InputRange = ActiveSheet.UsedRange

    ' Dodawanie nowego arkusza
    Set OutputSheet = Worksheets.Add

    ' Zmienna dla wiersza wyjściowego
    OutputRow = 1
    OutputRow = 1

    ' Przechodzenie w pętli przez zakres
    For Each Cell In InputRange
        If Cell.HasFormula Then
            OutputSheet.Cells(OutputRow, 1) = """ & Cell.Address
            OutputSheet.Cells(OutputRow, 2) = """ & Cell.Formula
            OutputRow = OutputRow + 1
        End If
    Next Cell
End Sub
  
```

 Below the code editor is the Immediate window, which displays:


```

? inputrange.Address
$A$1:$I$24

? cell.Address
$A$1
  
```

Rysunek 13.6.
Typowy wygląd
okna edytora
VBE w trybie
Break

Korzystanie z okna Immediate

Okno *Immediate* w edytorze VBE nie zawsze jest widoczne. Aby przywołać je na ekran, powinieneś nacisnąć kombinację klawiszy *Ctrl+G*.

W trybie *Break* okno *Immediate* jest szczególnie przydatne do sprawdzania wartości dowolnych zmiennych wykorzystywanych w programie. Aby na przykład sprawdzić bieżącą wartość zmiennej o nazwie *CellCount*, powinieneś w oknie *Immediate* wprowadzić polecenie przedstawione poniżej i wykonać je, naciskając klawisz *Enter*.

Print CellCount

Jeżeli chcesz, możesz zaoszczędzić kilkadziesiąt milisekund, używając znaku zapytania zamiast słowa kluczowego Print, co zostało przedstawione na poniższym przykładzie.

? CellCount

Oprócz sprawdzania wartości zmiennych, za pomocą okna Immediate możesz wykonywać wiele innych operacji. Możesz na przykład zmieniać wartości zmiennych, aktywować inny arkusz lub nawet otworzyć albo utworzyć nowy skoroszyt. Po prostu upewnij się, że wprowadzane w oknie Immediate polecenie jest poprawnym poleceniem języka VBA.

Z okna Immediate możesz również korzystać, kiedy Excel nie jest w trybie Break. Bardzo często używam tego okna do testowania małych fragmentów kodu (które mieszą się w jednym wierszu) przed dołączeniem ich do kodu moich procedur.

Krokowe wykonywanie programu

Kiedy Excel znajduje się w trybie Break, możesz wykonywać program krokowo, po jednym poleceniu na raz. Kolejne polecenia wykonywane są po każdym naciśnięciu klawisza F8. Dzięki krokowemu wykonywaniu kodu możesz używać okna Immediate do monitorowania zmiennych po każdym wierszu programu.

Za pomocą myszy możesz wskazać, które polecenie programu zostanie wykonane jako następne. Jeżeli ustawisz wskaźnik myszy w obszarze szarego marginesu po lewej stronie aktualnie wyróżnionego polecenia (zazwyczaj kolorem żółtym), wskaźnik myszy zmieni swój kształt na strzałkę skierowaną w prawo. Teraz możesz po prostu przeciągnąć wskaźnik myszy do polecenia, które powinno zostać wykonane jako następne.

Zastosowanie okna Watch

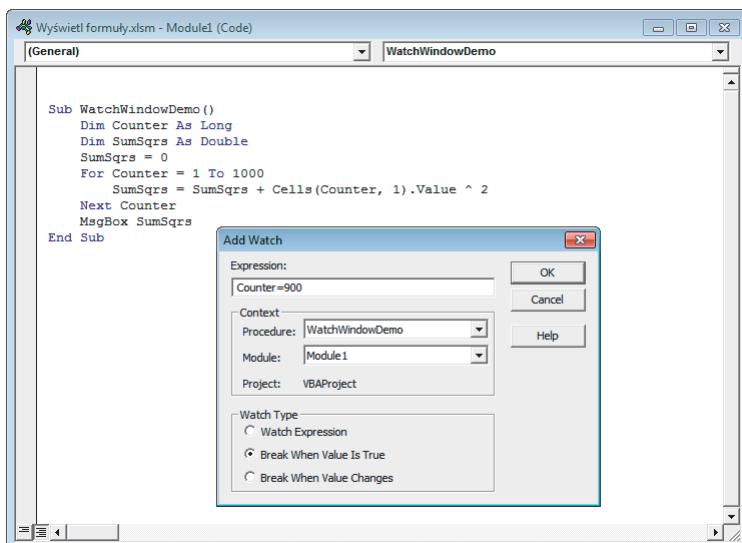
W niektórych przypadkach będziesz chciał wiedzieć, czy dana zmienna lub wyrażenie przyjmują określoną wartość. Założmy na przykład, że procedura przechodzi w pętli przez 1000 komórek. Zauważysz, że problemy zaczynają się przy 900 iteracji pętli. Cóż, w takiej sytuacji zawsze możesz wstawić do pętli punkt przerwania, ale oznaczać to będzie konieczność wykonania 899 wznowień działania programu, co prawdopodobnie szybko Ci się znudzi. Znacznie lepszym rozwiązaniem będzie zdefiniowanie *wyrażenia monitorującego* (*watch expression*).

Mögesz na przykład utworzyć wyrażenie monitorujące, które będzie przełączalo realizację procedury w tryb Break za każdym razem, kiedy dana zmienna osiągnie określoną wartość, taką jak w omawianym przypadku Counter = 900. Aby utworzyć wyrażenie monitorujące, wybierz z menu głównego edytora VBE polecenie Debug/Add Watch. Na ekranie pojawi się okno dialogowe Add Watch, przedstawione na rysunku 13.7.

Okno Add Watch składa się z trzech części.

- ✓ **Expression.** Tutaj powinieneś wprowadzić poprawne wyrażenie VBA lub nazwę zmiennej, na przykład Counter = 900 lub po prostu Counter.
- ✓ **Context.** Wybierz z odpowiedniej listy nazwę procedury i modułu, które chcesz monitorować. Zwróć uwagę, że możesz wybrać opcje All Procedures i All Modules.

Rysunek 13.7.
Okno dialogowe Add Watch pozwala na zdefiniowanie wyrażeń monitorujących, które powodują wstrzymanie realizacji programu



✓ **Watch Type.** Wybierz rodzaj monitorowania poprzez zaznaczenie odpowiedniego przycisku opcji. Wybór opcji będzie uzależniony od rodzaju wprowadzonego wcześniej wyrażenia. Pierwsza opcja, *Watch Expression*, nie powoduje przerwania działania programu; zamiast tego wartość wyrażenia jest po prostu wyświetlana, kiedy działanie programu zostanie przerwane w inny sposób.

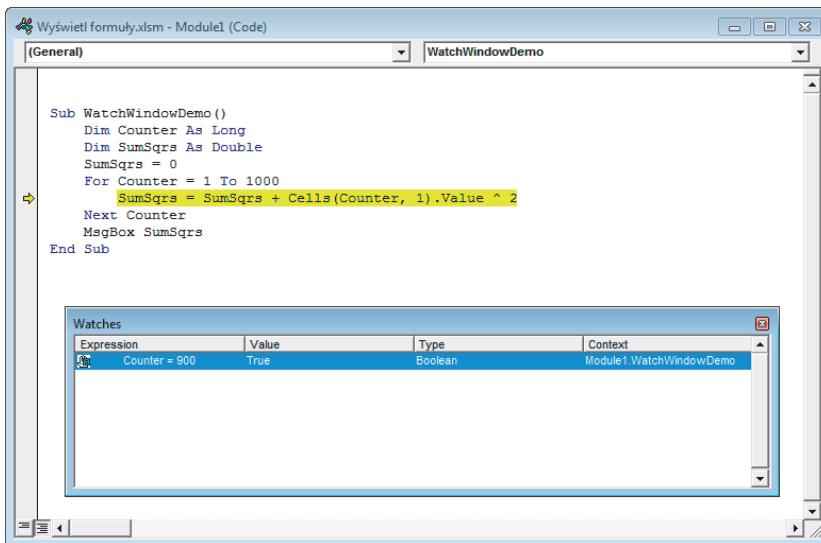
Po zdefiniowaniu wyrażenia uruchom procedurę w normalny sposób. Program będzie działał dopóty, dopóki warunek określony przez wyrażenie nie zostanie spełniony (w zależności od wybranej opcji *Watch Type*). Kiedy tak się stanie, Excel przejdzie do trybu *Break* (wybrałeś przecież opcję *Break When Value Is True*, prawda?). Od tej chwili możesz krokowo wykonywać kolejne polecenia programu lub używać okna *Immediate*.

Kiedy tworzysz wyrażenie monitorujące, edytor VBE wyświetla na ekranie okno *Watches*, przedstawione na rysunku 13.8. W tym oknie wyświetlane są wszystkie zdefiniowane przez Ciebie wyrażenia monitorujące. W naszym przykładzie widać, że zmienna *Counter* osiągnęła wartość 900, co spowodowało przerwanie działania programu i przejście Excela do trybu *Break*.

Najlepszym sposobem na zrozumienie zasad działania tego okna i powiązanego z nim mechanizmu jest samodzielne eksperymentowanie. Po pewnym czasie i nabraniu niezbędnego doświadczenia w pracy z oknem *Watches* będziesz się dziwił, jak w ogóle do tej pory mogłeś się bez niego obejść.

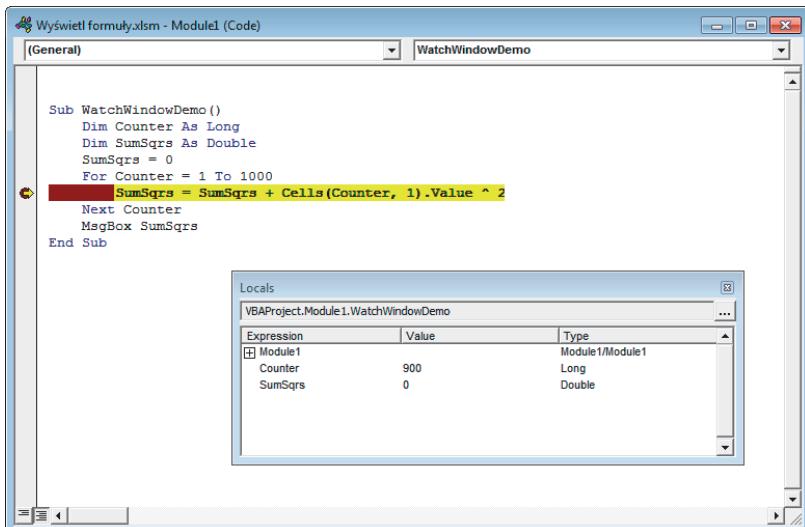
Zastosowanie okna Locals

Kolejnym, bardzo użytecznym narzędziem wspomagającym wyszukiwanie i usuwanie błędów jest okno dialogowe *Locals*. Aby je przywołać na ekran, powinieneś z menu głównego edytora VBE wybrać polecenie *View/Locals Window*. Kiedy Excel pracuje



Rysunek 13.8.
W oknie Watches wyświetlane są wszystkie wyrażenia monitorujące

w trybie Break, w oknie Locals znajdziesz listę wszystkich zmiennych lokalnych bieżącej procedury (zobacz rysunek 13.9). Bardzo miłą cechą tego okna jest to, że nie musisz do niego dodawać ręcznie żadnych wyrażeń monitorujących, pozwalających na sprawdzanie wartości wielu zmiennych — po prostu edytor VBE „odwala” tutaj całą robotę za Ciebie.



Rysunek 13.9.
W oknie Locals wyświetlane są nazwy i wartości wszystkich zmiennych lokalnych

Jak zredukować liczbę błędów w kodzie programu?

Nie jestem w stanie powiedzieć, co powinieneś zrobić, aby całkowicie wyeliminować możliwość pojawiania się błędów w Twoich programach VBA. Wyszukiwanie błędów w różnych programach może być przecież zupełnie samodzielna profesją. Spróbuje jednak przedstawić tutaj kilka porad, które mogą pomóc w zredukowaniu liczby błędów do minimum.

- ✓ **Zawsze umieszczaj dyrektywę Option Explicit na początku kodu każdego modułu.** Użycie tej dyrektywy powoduje wymuszenie deklaracji typu wszystkich zmiennych używanych w programie. Oznacza to — co prawda — nieco więcej pracy, ale w zamian pozwala na uniknięcie często popełnianych błędów związanych na przykład z literówkami w nazwach zmiennych. Dodatkowo, rozwiązanie takie ma jeszcze jedną, niebagatelną zaletę — Twoje procedury będą działały nieco szybciej.
- ✓ **Formatuj kod z użyciem wcięć.** Używanie wcięć w kodzie programu pozwala na łatwiejsze zapanowanie nad jego strukturą. Jeżeli na przykład w programie wykorzystujesz kilka zagnieżdżonych pętli For-Next, konsekwentne tworzenie wcięć dla poszczególnych poziomów znakomicie ułatwi panowanie nad strukturą takiej pętli.
- ✓ **Kiedy korzystasz z polecenia On Error Resume Next, zachowaj ostrożność.** Jak już wspomniałem w rozdziale 12, polecenie to powoduje, że w przypadku wystąpienia błędu w programie Excel po prostu go ignoriuje i przechodzi do wykonywania kolejnego polecenia. W niektórych przypadkach takie rozwiązanie może spowodować, że Excel przepuści błędy, których zignorować zdecydowanie nie powinien. Może to prowadzić do sytuacji, gdzie w Twoim kodzie pojawią się błędy, których istnienia nie będziesz nawet podejrzewał.
- ✓ **Umieszczaj w kodzie programu dużo komentarzy.** Nie ma nic bardziej frustrującego od powrotu do kodu swojego własnego programu po upływie, dajmy na to, sześciu miesięcy, i stwierdzeniu, że nie masz pojęcia, jak to działa. Dodawanie szczegółowych komentarzy do kodu już na etapie tworzenia pomoże w zaoszczędzeniu nerwów i niepotrzebnych frustracji w przyszłości.
- ✓ **Staraj się tworzyć proste funkcje i procedury.** Poprzez dzielenie kodu na małe moduły, z których każdy ma swoją jedną, dobrze zdefiniowaną rolę, znaczco ułatwisz sobie późniejszy proces odpluskowania.
- ✓ **Używaj rejestratora makr do identyfikacji właściwości i metod.** Kiedy nie pamiętam nazwy czy składni jakiejś właściwości lub metody, po prostu rejestruję nowe makro i analizuję jego kod.
- ✓ **Postaraj się zaprzyjaźnić z debuggerem Excela.** Choć początkowo może to być bardzo „szorstka” przyjaźń, jednak szybko przekonasz się, jak bardzo to narzędzie jest użyteczne. Zainwestuj trochę czasu i spróbuj go opanować.



Odpluskwianie z pewnością nie jest jednym z moich ulubionych zajęć (na liście plasuje się mniej więcej tuż przed niespodziewaną kontrolą z urzędu skarbowego), ale jest to — niestety — зло konieczne, nierozerwalnie związane z samym programowaniem. Na szczęście, w miarę jak będziesz zdobywał kolejne szlify w tworzeniu aplikacji w języku VBA, będziesz spędzał relatywnie mniej czasu na wyszukiwaniu i poprawianiu błędów, a jeżeli już będziesz musiał się zabrać za walkę z pluskwiami, z pewnością będzie ona coraz bardziej efektywna.

Rozdział 14

Przykłady i techniki programowania w języku VBA

W tym rozdziale:

- ▶ poznasz szereg przykładów technik programowania w języku VBA,
- ▶ dowiesz się, jak możesz przyspieszyć działanie kodu VBA w Twojej aplikacji.

Wierzę, że nauka programowania jest znacznie szybsza i zdecydowanie bardziej efektywna, kiedy pracujemy na konkretnych przykładach. Dobrze opracowany przykład o wiele lepiej objaśnia zagadnienie niż najbardziej rozbudowany i szczegółowy, ale teoretyczny opis. Ponieważ czytasz tę książkę, prawdopodobnie zgadzasz się ze mną w tej materii. W tym rozdziale znajdziesz szereg przykładów demonstrujących użyteczne, praktyczne techniki programowania w języku VBA.

Przykłady omawiane w tym rozdziale zostały podzielone na kilka kategorii. Oto one.

- ✓ Praca z zakresami.
- ✓ Modyfikacja ustawień Excela.
- ✓ Praca z wykresami.
- ✓ Przyspieszanie i optymalizacja działania kodu VBA.

Niektóre z prezentowanych przykładów będziesz mógł od razu wykorzystać w swoich aplikacjach, jednak w większości przypadków będą wymagały pewnego dostosowania do Twoich aplikacji.

Przetwarzanie zakresów komórek

Większość zadań, z jakimi będziesz się stykał, programując w języku VBA, będzie zapewne wymagała mniejszego bądź większego przetwarzania zakresów komórek (aby odświeżyć sobie wiadomości na temat obiektu Range, powinieneś zajrzeć do rozdziału 8.). Kiedy pracujesz z obiektami Range, powinieneś pamiętać o następujących sprawach.

- ✓ Kod VBA *nie musi zaznaczać* danego zakresu, aby go przetwarzać.
- ✓ Jeżeli kod VBA zaznacza wybrany zakres, przechowujący go skoroszyt musi być aktywny.

- ✓ Rejestrator makr nie zawsze będzie w stanie wygenerować optymalny kod VBA. Bardzo często jednak możesz za jego pomocą utworzyć bazowe makro i potem odpowiednio zmodyfikować kod tak, aby stał się bardziej efektywny.
- ✓ Zazwyczaj bardzo dobrym rozwiązaniem jest nadawanie nazw zakresom komórek wykorzystywanym w kodzie VBA. Przykładowo polecenie Range("Total") jest znacznie lepszym rozwiązaniem niż Range("D45"). Jeśli w tym drugim przypadku później wstawisz dodatkowy wiersz powyżej wiersza 45, to żeby wszystko działało poprawnie, będziesz musiał zmodyfikować makro tak, aby korzystało z nowego, poprawnego adresu komórki, a ta po wykonaniu takiej operacji będzie miała inny adres (D46). Aby nadać nazwę wybranemu zakresowi komórek, powinieneś przejść na kartę **FORMUŁY** i wybrać polecenie **Definiuj nazwę**, znajdujące się w grupie poleceń **Nazwy zdefiniowane**.
- ✓ Kiedy tworzysz makro, które będzie przetwarzalo zakres komórek zaznaczony przez użytkownika, pamiętaj, że użytkownik może zaznaczyć kilka całych kolumn czy wierszy. W większości przypadków z pewnością nie będziesz chciał, aby makro w pętli przechodziło w takiej sytuacji przez wszystkie zaznaczone komórki (włącznie z pustymi), co mogłoby zająć bardzo wiele czasu. Dobre makro powinno odszukać i przetwarzać tylko komórki, które nie są puste.
- ✓ Excel pozwala na jednocześnie zaznaczanie wielu zakresów komórek. Aby to zrobić, powinieneś zaznaczyć pierwszy zakres komórek, potem wcisnąć i przytrzymać klawisz **Ctrl**, i zaznaczać kolejne zakresy komórek przy użyciu myszy. Kod Twojej aplikacji powinien być przygotowany na takie sytuacje i podejmować odpowiednie akcje.



Skoroszyty z wybranymi przykładami omawianymi w tym rozdziale znajdziesz na stronie internetowej naszej książki.



Jeżeli chcesz samodzielnie wpisywać kod omawianych przykładów, przejdź do edytora VBE, naciśkając kombinację klawiszy **lewyAlt+F11**, a następnie wstaw nowy moduł VBA i wpisz kod prezentowanych procedur. Upewnij się, że Twój skoroszyt jest poprawnie skonfigurowany. Jeżeli na przykład kod danego przykładu odwołuje się do arkuszy o nazwach Arkusz1 i Arkusz2, upewnij się, że takie arkusze istnieją w Twoim skoroszycie.

Kopiowanie zakresów

Kopiowanie zakresów komórek może śmiało pretendować do miana jednej z najczęściej wykonywanych operacji w Excelu. Kiedy włączysz rejestrator makr i skopiujesz zakres komórek o adresie A1:A5 do zakresu B1:B5, otrzymasz następujące makro.

```
Sub CopyRange()
    Range("A1:A5").Select
    Selection.Copy
    Range("B1").Select
    ActiveSheet.Paste
    Application.CutCopyMode = False
End Sub
```

Zwróć uwagę na ostatnie polecenie. Zostało ono wygenerowane przez naciśnięcie klawisza *Esc* po skopiowaniu zakresu komórek, co spowodowało usunięcie przerywanej linii, reprezentującej na arkuszu obramowanie kopiowanego zakresu.

Przedstawione makro działa poprawnie, ale zakresy komórek można kopiować w znacznie bardziej efektywny sposób. Identyczny rezultat możesz osiągnąć za pomocą procedury, która składa się z tylko jednego wiersza polecenia i nie zaznacza żadnych komórek (dzięki czemu nie wymaga ustawiania właściwości *CutCopyMode* na wartość *False*).

```
Sub CopyRange2()
    Range("A1:A5").Copy Range("B1")
End Sub
```

Procedura przedstawiona powyżej korzysta z tego, że metoda *Copy* może używać argumentu *wywołania* reprezentującego miejsce docelowe kopiowanego zakresu. Informacje o tym znalazłem w pomocy systemowej VBA. Powyższy przykład doskonale również ilustruje fakt, że rejestrator makr nie zawsze generuje najbardziej efektywny kod.

Kopiowanie zakresu o zmiennej wielkości

W wielu przypadkach konieczne jest skopiowanie zakresu komórek, dla którego dokładna liczba wierszy i kolumn określających jego wielkość nie jest z góry znana. Przykładowo możesz dysponować skoroszytem śledzącym tygodniową sprzedaż, w którym liczba wierszy zmienia się każdego tygodnia po wprowadzeniu nowych danych.

Na rysunku 14.1 przedstawiam przykład często spotykanego typu arkusza. Znajdujący się w nim zakres komórek składa się z kilku wierszy, których liczba zmienia się każdego dnia. Ponieważ nie wiesz, jaki jest rozmiar zakresu w danej chwili, musisz utworzyć kod, który będzie działał bez używania adresu zakresu kopiowanych komórek.

	A	B	C	D	E
1	Data	Liczba sztuk	Kwota		
2	2013-08-04	181	6 697,00		
3	2013-08-05	174	5 742,00		
4	2013-08-06	201	7 437,00		
5	2013-08-07	229	8 473,00		
6	2013-08-08	203	6 496,00		
7	2013-08-09	229	7 328,00		
8	2013-08-10	213	8 094,00		
9	2013-08-11	219	8 322,00		
10	2013-08-12	236	7 780,00		
11	2013-08-13	261	8 613,00		
12	2013-08-14	262	8 646,00		
13					
14					
15					

Arkusz1 Arkusz2 (+)

Rysunek 14.1.
Przykład zakresu, który może składać się z dowolnej liczby wierszy

Makro przedstawione poniżej ilustruje sposób kopiowania zakresu komórek z arkusza Arkusz1 do arkusza Arkusz2 (począwszy od komórki A1). Makro wykorzystuje właściwość *CurrentRegion*, która zwraca obiekt *Range* odpowiadający blokowi komórek otaczających określoną komórkę (w tym przypadku o adresie A1).

```
Sub CopyCurrentRegion()
    Range("A1").CurrentRegion.Copy
    Sheets("Arkusz2").Select
    Range("A1").Select
    ActiveSheet.Paste
    Sheets("Arkusz1").Select
    Application.CutCopyMode = False
End Sub
```

Zastosowanie właściwości `CurrentRegion` jest równoważne z przejściem na kartę **NARZĘDZIA GŁÓWNE** i wybraniem polecenia **Znajdź i zaznacz/Przejdź do — specjalnie**, znajdującego się w grupie opcji **Edytowanie**, i następnie zaznaczeniem opcji **Bieżący obszar**. Aby przekonać się, jak to działa, podczas wykonywania tych poleceń powinieneś użyć rejestratora makr. Zazwyczaj wartość właściwości `CurrentRegion` reprezentuje prostokątny blok komórek otoczony przez puste wiersze i kolumny.

Oczywiście, możesz zoptymalizować kod makra przedstawionego powyżej i nie zaznaczać obszaru docelowego dla kopiowanych komórek. Makro przedstawione poniżej korzysta z faktu, że metoda `Copy` może używać argumentu wywołania reprezentującego miejsce docelowe kopiowanego zakresu.

```
Sub CopyCurrentRegion2()
    Range("A1").CurrentRegion.Copy
    Sheets("Arkusz2").Range("A1")
End Sub
```

 Jeżeli zakres komórek, który chcesz skopiować, jest tabelą (zdefiniowaną przy użyciu polecenia **WSTAWIANIE/Tabela/Tabla**), całe zadanie będzie jeszcze łatwiejsze. Każda tabela posiada swoją nazwę (na przykład `Tabela1`) i automatycznie rozszerza się w miarę dodawania nowych wierszy.

```
Sub CopyTable()
    Range("Tabela1").Copy Sheets("Arkusz2").Range("A1")
End Sub
```

Jeżeli spróbujesz wykonać procedurę przedstawioną powyżej, przekonasz się, że wiersz nagłówka tabeli nie jest kopowany, ponieważ obiekt `Tabela1` nie obejmuje tego wiersza. Jeśli chcesz, aby wiersz nagłówka również był kopowany, powinieneś zmienić odwołanie do tabeli tak, jak to zostało przedstawione poniżej.

```
Range("Tabela1[#A11]")
```

Zaznaczanie komórek do końca wiersza lub kolumny

Prawdopodobnie bardzo często używasz kombinacji klawiszy, takich jak `Ctrl + Shift + →`, czy `Ctrl + Shift + ↓`, do zaznaczania zakresów składających się ze wszystkich komórek, od komórki aktywnej, aż do końca kolumny czy wiersza. Nie jest więc chyba zaskoczeniem, że możesz napisać makro, które będzie zaznaczać komórki w podobny sposób.

Do zaznaczania całego bloku komórek możesz użyć właściwości `CurrentRegion`. Ale co powinieneś zrobić, jeżeli chcesz zaznaczyć, powiedzmy, tylko jedną kolumnę z tego bloku komórek? Na szczęście, VBA pozwala na wykonywanie takich operacji. Procedura, której kod przedstawiam poniżej, zaznacza zakres komórek, począwszy od bieżącej aktywnej komórki w dół kolumny, aż do komórki znajdującej się o jeden wiersz powyżej pierwszej pustej komórki tej kolumny. Po zaznaczeniu zakresu możesz przetwarzać go w dowolny sposób — kopiować komórki, przenosić je w inne miejsce arkusza, zmieniać formatowanie i tak dalej.

```
Sub SelectDown()
    Range(ActiveCell, ActiveCell.End(xlDown)).Select
End Sub
```

Oczywiście, taki sam obszar możesz również zaznaczyć ręcznie. Aby to zrobić, powinieneś zaznaczyć pierwszą komórkę, następnie wcisnąć i przytrzymać klawisz `Shift`, nacisnąć klawisz `End` i wreszcie nacisnąć klawisz ↓ (strzałka w dół).

W przykładzie wykorzystuję metodę `End` obiektu `ActiveCell`, która zwraca obiekt typu `Range`. Metoda `End` pobiera jeden argument określający kierunek, w którym zostanie wykonane zaznaczenie. Argumentami tej metody może być dowolna ze stałych przedstawionych poniżej:

- ✓ `xlUp`,
- ✓ `xlDown`,
- ✓ `xlToLeft`,
- ✓ `xlToRight`.

Pamiętaj, że zaznaczanie zakresu nie jest potrzebne do jego przetwarzania. Makro przedstawione poniżej zmienia czcionkę w komórkach zmiennego zakresu (pojedyncza kolumna) na pogrubioną bez uprzedniego zaznaczenia zakresu.

```
Sub MakeBold()
    Range(ActiveCell, ActiveCell.End(xlDown)) _
        .Font.Bold = True
End Sub
```

Zaznaczanie całego wiersza lub całej kolumny

Procedura przedstawiona poniżej ilustruje sposób zaznaczania kolumny, w której znajduje się aktywna komórka. Makro wykorzystuje właściwość `EntireColumn`, która zwraca obiekt typu `Range` reprezentujący całą kolumnę.

```
Sub SelectColumn()
    ActiveCell.EntireColumn.Select
End Sub
```

Jak pewnie się spodziewałeś, w języku VBA dostępna jest również właściwość `EntireRow`, która zwraca obiekt typu `Range` reprezentujący cały wiersz.

Przenoszenie zakresów

Zazwyczaj, aby przenieść zakres komórek, zaznaczasz go, wycinasz do schowka systemowego i następnie wklejasz w inne miejsce. Jeżeli użyjesz rejestratora makr do zapisania takiej operacji, przekonasz się, że wygenerowany zostanie kod podobny do przedstawionego poniżej.

```
Sub MoveRange()
    Range("A1:C6").Select
    Selection.Cut
    Range("A10").Select
    ActiveSheet.Paste
End Sub
```

Podobnie jak podczas kopiowania komórek, takie rozwiążanie nie jest najbardziej efektywnym sposobem przenoszenia zakresu komórek w inne miejsce. W praktyce taką operację możesz wykonać za pomocą procedury składającej się z jednego wiersza kodu, co przedstawię poniżej.

```
Sub MoveRange2()
    Range("A1:C6").Cut Range("A10")
End Sub
```

Makro przedstawione powyżej korzysta z faktu, że metoda `Cut` może używać argumentu wywołania reprezentującego miejsce docelowe przenoszonego zakresu. Zwrót również uwagę na fakt, że podczas przenoszenia żaden zakres komórek nie jest zaznaczany. Wskaźnik aktywnej komórki przez cały czas pozostaje w tym samym miejscu arkusza.

Wydajne przetwarzanie komórek zaznaczonego zakresu przy użyciu pętli

Jednym z zadań często wykonywanych przez makra jest sprawdzanie poszczególnych komórek zakresu i wykonywanie określonych operacji na podstawie ich zawartości. Takie makra zazwyczaj wykorzystują pętlę `For-Next`, za pomocą której przetwarzane są komórki zakresu.

Przykład przedstawiony niżej ilustruje sposób przechodzenia kolejno przez wszystkie komórki danego zakresu. W naszym przypadku przetwarzany jest aktualnie zaznaczony zakres komórek. Zmienna obiektowa o nazwie `Cell` reprezentuje aktualnie przetwarzaną komórkę. W pętli `For Each-Next` znajduje się jedno polecenie, które sprawdza aktualnie przetwarzaną komórkę i zmienia jej czcionkę na pogrubioną, jeżeli wartość przechowywana w komórce jest dodatnia.

```
Sub ProcessCells()
    Dim Cell As Range
    For Each Cell In Selection
        If Cell.Value > 0 Then Cell.Font.Bold = True
    Next Cell
End Sub
```

Taka procedura działa poprawnie, ale co się stanie, jeżeli użytkownik zaznaczy całą kolumnę lub cały wiersz? To wcale nie jest takie nieprawdopodobne, bo przecież Excel pozwala na wykonywanie operacji na całych wierszach i kolumnach. W takiej sytuacji wykonanie makra może zająć naprawdę dużo czasu, ponieważ nasza pętla przetwarza każdą komórkę zaznaczonego zakresu, a łącznie z pustymi w jednej kolumnie komórek jest aż 1 048 576... Aby zatem nasze makro było bardziej wydajne, musimy je tak zmodyfikować, żeby przetwarzane były tylko i wyłącznie komórki, które nie są puste.

Procedura przedstawiona poniżej przetwarza wyłącznie niepuste komórki zaznaczonego zakresu dzięki zastosowaniu metody `SpecialCells` (więcej szczegółowych informacji na temat tej metody znajdziesz w pomocy systemowej VBA). Nasza procedura za pomocą polecenia `Set` tworzy dwa obiekty typu `Range`: pierwszy z nich to podzakres komórek zakresu wejściowego, zawierający wyłącznie komórki z wartościami stałymi (na przykład teksty, wartości liczbowe, literaly i tak dalej), a drugi składa się z komórek zawierających formuły. Procedura przetwarza tylko komórki należące do tych podzakresów, co w efekcie powoduje pominięcie przetwarzania wszystkich pozostałych, pustych komórek zakresu wejściowego. Sprytne, prawda?

```
Sub SkipBlanks()
    Dim ConstantCells As Range
    Dim FormulaCells As Range
    Dim cell As Range
    ' Ignoruj błędy
    On Error Resume Next

    ' Przetwarzaj komórki zawierające wartości stałe
    Set ConstantCells = Selection _
        .SpecialCells(xlConstants)
    For Each cell In ConstantCells
        If cell.Value > 0 Then
            cell.Font.Bold = True
        End If
    Next cell

    ' Przetwarzaj komórki zawierające formuły
    Set FormulaCells = Selection _
        .SpecialCells(xlFormulas)
    For Each cell In FormulaCells
        If cell.Value > 0 Then
            cell.Font.Bold = True
        End If
    Next cell
End Sub
```

Procedura `SkipBlanks` działa tak samo szybko, niezależnie od tego, jaki zakres komórek zaznaczyłeś. Możesz na przykład zaznaczyć zakres składający się z kilku komórek, zaznaczyć wszystkie kolumny w danym zakresie albo wszystkie wiersze w danym zakresie, albo nawet cały arkusz. Jak widać, jest to ogromne usprawnienie w stosunku do oryginalnej procedury `ProcessCells`, którą omówiłem nieco wcześniej.

Zwrót uwagi, że w kodzie procedury użyliśmy polecenia:

```
On Error Resume Next
```

Polecenie to powoduje, że Excel po prostu ignoriuje błędy (inaczej mówiąc, jeżeli próba wykonania danego polecenia kończy się błędem, Excel ignoriuje ten błąd i po prostu przechodzi do kolejnego polecenia; więcej szczegółowych informacji na temat obsługi błędów znajdziesz w rozdziale 12.). W naszym przypadku zastosowanie polecenia On Error jest konieczne, ponieważ metoda SpecialCells generuje błąd, gdy żadna komórka nie spełnia podanego kryterium.

Zastosowanie metody SpecialCells jest równoważne z przejściem na kartę **NARZĘDZIA GŁÓWNE**, wybraniem polecenia **Znajdź i zaznacz/Przejdź do — specjalnie**, znajdującego się w grupie opcji **Edytowanie**, i następnie zaznaczeniem opcji **Stale** lub **Formuły**. Aby przekonać się, jak to działa, podczas wykonywania tych poleceń powinieneś użyć rejestratora makr i zaznaczać różne opcje.

Wydajne przetwarzanie komórek zaznaczonego zakresu przy użyciu pętli (część II)

A teraz ciąg dalszy naszej opowieści. W tym punkcie przedstawię inny sposób efektywnego przetwarzania komórek znajdujących się w zaznaczonym zakresie. Tym razem procedura będzie korzystała z właściwości UsedRange, która zwraca obiekt typu Range, reprezentujący używany zakres arkusza. Procedura korzysta również z metody Intersect, która zwraca obiekt typu Range zawierający komórki będące częścią wspólną dwóch zakresów.

Poniżej przedstawiam zmodyfikowaną wersję procedury SkipBlanks, omawianej w poprzednim punkcie.

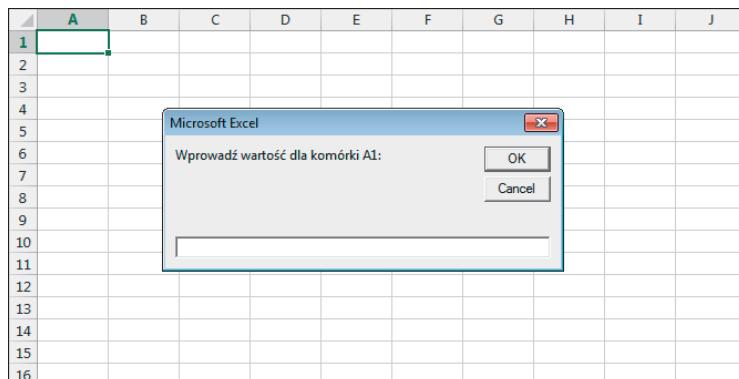
```
Sub SkipBlanks2()
    Dim WorkRange As Range
    Dim cell As Range
    Set WorkRange = Intersect(Selection, ActiveSheet.UsedRange)
    For Each cell In WorkRange
        If cell.Value > 0 Then
            cell.Font.Bold = True
        End If
    Next cell
End Sub
```

Zmienna obiektowa WorkRange zawiera komórki, które są częścią wspólną zakresu zaznaczonego przez użytkownika oraz zakresu używanych komórek arkusza. Jeśli zatem użytkownik zaznaczy całą kolumnę, zmienna WorkRange będzie zawierała tylko komórki, które znajdują się jednocześnie w zaznaczonej kolumnie i w używanym zakresie arkusza. Jak widać, jest to bardzo szybka i efektywna metoda pozwalająca na uniknięcie przetwarzania komórek znajdujących się poza zakresem używanych komórek arkusza.

Wprowadzanie wartości do komórki

Na rysunku 14.2 pokazuję, w jaki sposób możesz użyć funkcji InputBox języka VBA do pobierania od użytkownika wartości, która następnie może zostać zapisana w wybranej komórce. W procedurze przedstawionej poniżej demonstruję, jak poprosić użytkownika o podanie wartości i wstawić ją do komórki A1 aktywnego arkusza (i to wszystko za pomocą jednego polecenia).

```
Sub GetValue()
    Range("A1").Value = InputBox(
        "Wprowadź wartość dla komórki A1:")
End Sub
```



Rysunek 14.2.
Zastosowanie
funkcji Input-
Box do pobie-
rania wartości
od użytkownika

Przedstawiona procedura może jednak sprawiać pewien problem. Jeżeli użytkownik naciśnie w oknie dialogowym przycisk *Cancel*, procedura usunie wszelkie dane już znajdujące się w komórce A1, a takie zachowanie nie jest zbyt dobrą praktyką programistyczną. Naciśnięcie przycisku *Cancel* powinno po prostu usuwać z ekranu okno dialogowe bez wykonywania żadnych dodatkowych operacji.

Makro przedstawione poniżej ilustruje znacznie lepsze podejście do takiego zagadnienia i do zapamiętania wartości wprowadzonej przez użytkownika wykorzystując zmienną *x*. Jeżeli zmienna zawiera coś innego niż pusty ciąg znaków (czyli kiedy użytkownik wprowadził jakąś wartość), wartość zmiennej jest zapisywana w komórce A1. W przeciwnym przypadku procedura kończy działanie, nie wykonując żadnych dodatkowych operacji.

```
Sub GetValue2()
    Dim x as Variant
    x = InputBox("Wprowadź wartość dla komórki A1:")
    If x <> "" Then Range("A1").Value = x
End Sub
```

Zmienna *x* została zdefiniowana jako zmienna typu *Variant*, ponieważ jej wartością może być wartość albo pusty ciąg znaków (jeżeli użytkownik naciśnie przycisk *Cancel*).

Określanie typu zaznaczonego zakresu

Jeżeli zadaniem Twojego makra będzie przetwarzanie zaznaczonego zakresu, takie makro musi mieć zdolność sprawdzenia, czy przed jego wywołaniem użytkownik rzeczywiście zaznaczył zakres komórek. W przeciwnym razie, jeżeli przed wywołaniem makra zaznaczony zostanie inny obiekt (na przykład wykres lub kształt), próba wykonania makra najprawdopodobniej zakończy się niepowodzeniem. Polecenie przedstawione poniżej wykorzystuje funkcję `TypeName` języka VBA do wyświetlania na ekranie typu aktualnie zaznaczonego obiektu.

```
MsgBox TypeName(Selection)
```

Jeżeli aktualnie zaznaczony jest obiekt typu `Range`, wykonanie takiego polecenia spowoduje wyświetlenie słowa `Range`. Jeśli Twoje makro działa tylko z zakresami komórek, możesz użyć polecenia `If` do sprawdzenia, czy aktualnie zaznaczony obiekt to zakres (obiekt typu `Range`). Procedura przedstawiona poniżej sprawdza typ zaznaczonego obiektu i jeżeli nie jest to obiekt typu `Range`, na ekranie wyświetlany jest odpowiedni komunikat i procedura kończy działanie.

```
Sub CheckSelection()
    If TypeName(Selection) <> "Range" Then
        MsgBox "Zaznacz zakres komórek."
        Exit Sub
    End If
    ' ... [Tutaj wstaw dalszą część kodu procedury]
End Sub
```

Identyfikowanie zaznaczeń wielokrotnych

Jak pamiętasz, Excel pozwala na jednocześnie zaznaczanie wielu obiektów. Aby to zrobić, powinieneś podczas zaznaczania obiektów lub zakresów trzymać wcisnięty klawisz `Ctrl`. Zaznaczenia wielokrotne mogą być przyczyną problemów z wykonywaniem niektórych makr. Przykładowo nie możesz skopiować zakresu komórek, który został utworzony poprzez wiele zaznaczeń nieciągłych zakresów komórek. Jeżeli spróbujesz wykonać taką operację, Excel wyświetli na ekranie komunikat przedstawiony na rysunku 14.3.

Makro przedstawione niżej pokazuje, w jaki sposób możesz sprawdzić, czy użytkownik dokonał zaznaczenia wielokrotnego, i na tej podstawie wykonać odpowiednią operację.

```
Sub MultipleSelection()
    If Selection.Areas.Count > 1 Then
        MsgBox "Zaznaczenia wielokrotne nie są dozwolone."
        Exit Sub
    End If
    ' ... [Tutaj wstaw dalszą część kodu procedury]
End Sub
```

Przedstawiona procedura wykorzystuje metodę `Areas`, która zwraca kolekcję wszystkich zakresów w danym zaznaczeniu. Właściwość `Count` zwraca liczbę obiektów tej kolekcji.

	A	B	C	D	E	F	G	H	I	J
1	448	136	357	973	491	266	476	972		
2	314	609	958	518	195	780	300	918		
3	515	914	940	3	459	648	568	574		
4	885	189	412	232	633	65	683	321		
5	823	127	820	751	887	964	716	118		
6	313	854	829	98	310	252	924	340		
7	855	620	408	997	686	254	766	74		
8	48	290								
9	135	384								
10	421	200								
11	925	639								
12	871	607								
13	949	491								
14	771	352	291	827	458	464	489	156		
15	708	885	200	484	290	864	902	247		
16	824	889	756	877	461	145	971	48		
17	352	554	256	967	192	1	479	707		
18	299	894	165	964	426	939	72	463		
19	893	754	377	516	803	535	139	714		
20	745	782	861	740	908	563	565	54		
21	676	962	189	648	807	322	577	852		
22	156	61	795	896	308	992	1000	274		
23	139	806	737	510	521	775	825	700		
24	240	449	341	475	425	263	691	56		
25	681	821	736	223	18	634	306	217		

← → Arkusz1 Arkusz2 Arkusz3 +

Rysunek 14.3.
Excel nie lubi,
kiedy próbujesz
kopować nie-
ciągłe zakresy
komórek

Zmiana ustawień Excela

Chyba najbardziej użytecznymi makrami są proste procedury, które zmieniają jedno lub kilka ustawień Excela. Jeśli na przykład dojdziesz do wniosku, że bardzo często przywołujesz na ekran okno opcji programu Excel i zmieniasz jakieś ustawienie, taka operacja z pewnością będzie bardzo dobrym kandydatem do napisania makra oszczędzającego Twój czas i ułatwiającego zmianę takiego ustawienia.

W tym podrozdziale zaprezentuję dwa przykłady procedur, które pokazują, w jaki sposób można zmieniać ustawienia Excela. Ogólne zasady działania tych procedur możesz z powodzeniem zastosować do napisania własnych makr zmieniających inne ustawienia Excela.

Zmiana ustawień logicznych (opcje typu Boolean)

Podobnie jak wyłącznik światła, opcje logiczne (typu *Boolean*) mogą być albo włączone, albo wyłączone. Możesz na przykład utworzyć makro, które będzie włączało lub wyłączało wyświetlanie podziału arkusza na strony. Kiedy wydrukujesz arkusz (lub skorzystasz z trybu podglądu wydruku), Excel wyświetla na arkuszu przerywane linie reprezentujące miejsca podziału arkusza na strony wydruku. Niektórych użytkowników (włącznie z autorem tej książki) takie zachowanie Excela irytuje. Niestety, jedynym sposobem wyłączenia wyświetlania podziału arkusza na strony jest otarcie okna dialogowego *Opcje programu Excel*, przejście na kartę *Zaawansowane*, a następnie przewijanie zawartości tej karty, aż do odszukania opcji *Pokaż podziały stron*. Jeżeli podczas wyłączania tej opcji korzystałeś z rejestratora makr, przekonasz się, że Excel generuje poniższy kod.

```
ActiveSheet.DisplayPageBreaks = False
```

Z drugiej strony, jeżeli podczas rejestrowania makra podziały stron nie są widoczne, Excel generuje taki kod.

```
ActiveSheet.DisplayPageBreaks = True
```

Takie informacje mogą doprowadzić do wniosku, że będziesz musiał napisać aż dwa makra — jedno do włączania podglądu podziału stron, a drugie do jego wyłączenia. Na szczęście, to nieprawda. Procedura przedstawiona poniżej wykorzystuje operator Not do zmiany wartości logicznej *True* na *False* i odwrotnie. Wykonanie procedury *TogglePageBreaks* to prosty sposób na cykliczne włączanie i wyłączenie podglądu podziału stron arkusza.

```
Sub TogglePageBreaks()
    On Error Resume Next
    ActiveSheet.DisplayPageBreaks = Not _
        ActiveSheet.DisplayPageBreaks
End Sub
```

Pierwsze polecenie informuje Excel, że powinien ignorować ewentualne błędy. Przykładowo podziały stron nie są wyświetlane na arkuszach wykresów. Kiedy wprowadzisz takie polecenie i spróbujesz wykonać tę procedurę dla arkusza wykresu, na ekranie nie pojawi się komunikat o błędzie.

Techniki użytej w procedurze *TogglePageBreaks* możesz używać do zmiany dowolnych opcji logicznych (czyli takich, których wartościami są *True* albo *False*).

Zmiana innych opcji (typu non-Boolean)

Do zmiany opcji, które nie są typu logicznego, możesz używać konstrukcji *Select Case*. W przykładzie przedstawionym poniżej zmieniam tryb przeliczania skoroszytu z ręcznego na automatyczny i odwrotnie, i nakazuję wyświetlenie na ekranie komunikatu opisującego aktualny tryb przeliczania arkusza.

```
Sub ToggleCalcMode()
    Select Case Application.Calculation
        Case xlManual
            Application.Calculation = xlCalculationAutomatic
            MsgBox "Automatyczne przeliczanie skoroszytu"
        Case xlAutomatic
            Application.Calculation = xlCalculationManual
            MsgBox "Ręczne przeliczanie skoroszytu"
    End Select
End Sub
```

Techniki użytej w procedurze *ToggleCalcMode* możesz używać do zmiany dowolnych opcji, które nie posiadają wartości logicznych.

Praca z wykresami

Wykresy w Excelu są wręcz przeładowane najróżniejszymi obiektami, stąd ich przetwarzanie za pomocą kodu VBA może być niezłyym wyzwaniem.

Uruchomiłem Excel 2013, w zakresie komórek A1:A3 wpisałem kilka liczb i zaznaczyłem ten obszar arkusza. Następnie włączyłem rejestrator makr i dla tych trzech punktów danych utworzyłem prosty wykres kolumnowy. Później usunąłem wyświetlanie siatki wykresu i zmieniłem jego tytuł. Oto zarejestrowane makro.

```
Sub Macro1()
    ' Zarejestrowane w Excelu 2013
    ActiveSheet.Shapes.AddChart2(201, xlColumnClustered).Select
    ActiveChart.SetSourceData Source:=Range("Arkusz1!$A$1:$A$3")
    ActiveChart.SetElement (msoElementPrimaryValueGridLinesNone)
    ActiveChart.ChartTitle.Select
    ActiveChart.ChartTitle.Text = "To jest mój wykres"
End Sub
```

Kiedy zobaczyłem ten kod, byłem nieco zaskoczony, ponieważ nigdy wcześniej nie słyszałem o metodzie AddChart2. Okazało się, że metoda AddChart2 to nowość, która została wprowadzona w Excelu 2013. Jeżeli wykonasz podobną operację z rejestraniem makra w Excelu 2010, wynik będzie następujący.

```
Sub Macro1()
    ' Zarejestrowane w Excelu 2010
    ActiveSheet.Shapes.AddChart.Select
    ActiveChart.ChartType = xlColumnClustered
    ActiveChart.SetSourceData Source:=Range("Arkusz1!$A$1:$A$3")
    ActiveChart.Axes(xlValue).MajorGridlines.Select
    Selection.Delete
    ActiveChart.SetElement (msoElementChartTitleAboveChart)
    ActiveChart.ChartTitle.Text = "To jest mój wykres"
End Sub
```

Co to wszystko oznacza? Ano tyle, że makra zarejestrowane w Excelu 2013 po prostu nie będą działały w Excelu 2010, ale makra rejestrowane w Excelu 2010 działają w Excelu 2013. Innymi słowy, makra Excela 2010 są kompatybilne w przód (czyli zachowują zgodność z przyszłymi wersjami Excela; *forward compatibility*), a makra Excela 2013 zostały pozbawione kompatybilności wstecznej (*backward compatibility*), czyli nie zachowują zgodności z poprzednimi wersjami.



Przeciętny użytkownik Excela 2013 prawdopodobnie nie wie nic na temat kompatybilności makr w odniesieniu do tworzenia wykresów. Jeżeli jednak udostępnisz takie makro komuś, kto używa starszej wersji Excela, bardzo szybko się o tym dowiesz. Wnioski? Gdy używasz rejestratora makr do tworzenia makr przetwarzających wykresy, powinieneś przetestować takie makra na wszystkich wersjach Excela, które będą wykorzystywane do uruchamiania takiego makra.

Metoda AddChart kontra metoda AddChart2

Poniżej przedstawiam oficjalną składnię metody AddChart (metoda jest kompatybilna z Exceliem 2007 i wersjami późniejszymi).

```
.AddChart(Type, Left, Top, Width, Height)
```

A oto składnia metody AddChart2 (która jest kompatybilna wyłącznie z Exceliem 2013).

```
.AddChart2 (Style, XlChartType, Left, Top, Width, Height, NewLayout)
```

Jak widać, metoda AddChart2 pobiera kilka dodatkowych argumentów, które określają styl wykresu, typ wykresu oraz jego układ. Z drugiej strony, metoda AddChart tworzy po prostu pusty wykres, a wszystkie detale muszą być zdefiniowane za pomocą dodatkowych poleceń.

Analiza zarejestrowanego kodu ujawnia kilka rzeczy, które mogą być pomocne podczas tworzenia własnych makr przetwarzających wykresy. Jeżeli jesteś ciekawy, rzuć okiem na zmodyfikowaną ręcznie procedurę, której zadaniem jest utworzenie wykresu na bazie zaznaczonego zakresu komórek.

```
Sub CreateAChart()
    Dim ChartData As Range
    Dim ChartShape As Shape
    Dim NewChart As Chart

    ' Tworzenie zmiennych obiektowych
    Set ChartData = ActiveWindow.RangeSelection
    Set ChartShape = ActiveSheet.Shapes.AddChart
    Set NewChart = ChartShape.Chart

    With NewChart
        .ChartType = xlColumnClustered
        .SetSourceData Source:=Range(ChartData.Address)
        .SetElement (msoElementLegendRight)
        .SetElement (msoElementChartTitleAboveChart)
        .ChartTitle.Text = "To jest mój wykres"
    End With
End Sub
```

To makro jest kompatybilne z Exceliem 2007 i wersjami późniejszymi. Makro tworzy grupowany wykres kolumnowy wraz z legendą i tytułem. Jest to podstawowa wersja makra, która w łatwy sposób może być dostosowana do Twoich indywidualnych wymagań. Jednym ze sposobów może być rejestrowanie makra podczas modyfikowania wykresu i następnie używanie takiego kodu jako wzorca w swoich procedurach.

Swoją drogą, dalej w tym rozdziale omówię konstrukcję With End-With, która znakomicie ułatwia pracę z obiektami, oszczędza sporo „stukania w klawiaturę” i znakomicie przyczynia się do zwiększenia przejrzystości kodu.

Jeżeli musisz napisać makro VBA, którego zadaniem będzie przetwarzanie wykresów, musisz zapoznać się z kilkoma ważnymi określeniami. *Wykres osadzony (embedded chart)* na arkuszu to obiekt typu ChartObject. Obiekt ChartObject możesz aktywować podobnie jak aktywujesz arkusz. Polecenie przedstawione poniżej aktywuje obiekt ChartObject o nazwie Wykres 1.

```
ActiveSheet.ChartObjects("Wykres 1").Activate
```

Po aktywowaniu danego wykresu możesz się do niego odwoływać w kodzie VBA za pomocą obiektu ActiveChart. Jeżeli wykres znajduje się na osobnym arkuszu wykresu, staje się wykresem aktywnym w chwili, kiedy aktywujesz arkusz wykresu.



Obiekt ChartObject jest również obiektem typu Shape, co może być nieco mylące. W rzeczywistości, kiedy Twój kod VBA tworzy wykres, cała operacja rozpoczyna się od utworzenia nowego obiektu Shape (kształt). Wykres możesz również aktywować poprzez zaznaczenie obiektu Shape przechowującego wykres.

```
ActiveSheet.Shapes("Wykres 1").Select
```

W moich programach wolę używać obiektu ChartObject, dzięki czemu nie mam żadnych wątpliwości, że pracuję z wykresami.



Kiedy klikasz wykres osadzony lewym przyciskiem myszy, Excel zaznacza obiekt znajdujący się *wewnątrz* obiektu ChartObject. Jeżeli chcesz zaznaczyć sam obiekt ChartObject, powinieneś klikając wykres, trzymać wciśnięty klawisz *Ctrl*.

Modyfikowanie typu wykresu

A teraz przeczytasz zdanie, które może Cię nieco zdezorientować: obiekty ChartObject spełniają rolę kontenerów dla obiektów Chart. Jeśli masz jakieś wątpliwości, powinieneś to zdanie kilka razy spokojnie przeczytać i wtedy na pewno wszystko stanie się jasne.

Aby zmodyfikować wykres za pomocą VBA, nie musisz tego wykresu aktywować. Metoda Chart może zwracać wykres przechowywany w kontenerze ChartObject. Nadal niejasne? Procedury przedstawione poniżej dają taki sam efekt — zmieniają typ wykresu o nazwie *Wykres 1* na wykres powierzchniowy. Pierwsza procedura najpierw aktywuje wykres i następnie pracuje z aktywnym wykresem. Druga procedura nie aktywuje wykresu, a zamiast tego wykorzystuje właściwość Chart, która zwraca obiekt Chart zawarty w kontenerze ChartObject.

```
Sub ModifyChart1()
    ActiveSheet.ChartObjects("Wykres 1").Activate
    ActiveChart.Type = xlArea
End Sub

Sub ModifyChart2()
    ActiveSheet.ChartObjects("Wykres 1").Chart.Type = xlArea
End Sub
```

Przechodzenie w pętli przez elementy kolekcji ChartObjects

Procedura przedstawiona poniżej wprowadza zmiany do wszystkich wykresów osadzonych na aktywnym arkuszu. Procedura wykorzystuje pętlę For Each-Next do przechodzenia kolejno przez wszystkie obiekty kolekcji ChartObjects i dla każdego obiektu Chart zmienia jego właściwość Type.

```
Sub ChartType()
    Dim cht As ChartObject
    For Each cht In ActiveSheet.ChartObjects
        cht.Chart.Type = xlArea
    Next cht
End Sub
```

Makro przedstawione poniżej wykonuje taką samą operację, ale na wszystkich arkuszach wykresów w aktywnym skoroszycie.

```
Sub ChartType2()
    Dim cht As Chart
    For Each cht In ActiveWorkbook.Charts
        cht.Type = xlArea
    Next cht
End Sub
```

Modyfikowanie właściwości wykresu

Procedura przedstawiona niżej zmienia czcionkę legendy wykresu dla wszystkich wykresów osadzonych na aktywnym arkuszu. Makro wykorzystuje pętlę For-Next do przetwarzania wszystkich obiektów ChartObject.

```
Sub LegendMod()
    Dim chtObj As ChartObject
    For Each chtObj In ActiveSheet.ChartObjects
        With chtObj.Chart.Legend.Font
            .Name = "Calibri"
            .FontStyle = "Bold"
            .Size = 12
        End With
    Next chtObj
End Sub
```

Zwróć uwagę na fakt, że obiekt Font jest zawarty w obiekcie Legend, który jest zawarty w obiekcie Chart, który z kolei jest zawarty w kolekcji ChartObjects. Czy teraz rozumiesz, dlaczego to wszystko jest nazywane *hierarchią obiektów*?

Zmiana formatowania wykresów

Ten przykład odnosi się do kilku różnych typów formatowania aktywnego wykresu. Utworzyłem to makro, rejestrując moje poczynania podczas formatowania wykresu. Następnie oczyściłem nieco uzyskany kod poprzez usunięcie zbędnych wierszy.

```
Sub ChartMods()
    ActiveChart.Type = xlArea
    ActiveChart.ChartArea.Font.Name = "Calibri"
    ActiveChart.ChartArea.Font.FontStyle = "Regular"
    ActiveChart.ChartArea.Font.Size = 9
    ActiveChart.PlotArea.Interior.ColorIndex = xlNone
    ActiveChart.Axes(xlValue).TickLabels.Font.Bold = True
    ActiveChart.Axes(xlCategory).TickLabels.Font.Bold =
        True
    ActiveChart.Legend.Position = xlBottom
End Sub
```

Przed wykonaniem tego makra musisz aktywować wykres. Wykresy osadzone możesz aktywować poprzez ich kliknięcie lewym przyciskiem myszy. Aby aktywować wykres na arkuszu wykresu, kliknij kartę arkusza.

Aby upewnić się, że wykres jest zaznaczony, możesz w kodzie procedury umieścić polecenie, które będzie sprawdzało, czy wykres jest aktywny. Poniżej znajdziesz kod zmodyfikowanej procedury, która — jeżeli wykres nie jest aktywny — wyświetla na ekranie odpowiedni komunikat i kończy działanie.

```
Sub ChartMods2()
    If ActiveChart Is Nothing Then
        MsgBox "Aktywuj wykres!"
        Exit Sub
    End If
    ActiveChart.Type = xlArea
    ActiveChart.ChartArea.Font.Name = "Calibri"
    ActiveChart.ChartArea.Font.FontStyle = "Regular"
    ActiveChart.ChartArea.Font.Size = 9
    ActiveChart.PlotArea.Interior.ColorIndex = xlNone
    ActiveChart.Axes(xlValue).TickLabels.Font.Bold = True
    ActiveChart.Axes(xlCategory).TickLabels.Font.Bold =
        True
    ActiveChart.Legend.Position = xlBottom
End Sub
```

Poniżej znajdziesz kolejną wersję procedury, która wykorzystuje konstrukcję With-End With do zaoszczędzenia „klepania” w klawiaturę i (co ważniejsze) zwiększenia optymalności i przejrzystości kodu. I znów wyskakujemy nieco przed orkiestrą, ale jeżeli chcesz, możesz już teraz przeskoczyć parę stron do przodu i przeczytać opis polecenia With-End With.

```
Sub ChartMods3()
    If ActiveChart Is Nothing Then
        MsgBox "Aktywuj wykres!"
        Exit Sub
    End If
```

```
With ActiveChart
    .Type = xlArea
    .ChartArea.Font.Name = "Calibri"
    .ChartArea.Font.FontStyle = "Regular"
    .ChartArea.Font.Size = 9
    .PlotArea.Interior.ColorIndex = xlNone
    .Axes(xlValue).TickLabels.Font.Bold = True
    .Axes(xlCategory).TickLabels.Font.Bold = True
    .Legend.Position = xlBottom
End With
End Sub
```

No cóż... w zakresie zastosowania VBA do przetwarzania wykresów udało Ci się w tym rozdziale jedynie nieco „liznąć” podstawowe elementy tego rozbudowanego zagadnienia. Temat jest niezwykle szeroki, ale mam nadzieję, że to, czego dowiedziałeś się w tym rozdziale, pobudziło Twoją ciekawość i nakierowało poszukiwania we właściwym kierunku.

Jak przyspieszyć działanie kodu VBA?

VBA jest szybki, ale nie zawsze wystarczająco szybki (inna sprawa, że programy komputerowe nigdy nie są wystarczająco szybkie, przynajmniej w opinii większości użytkowników). W tym podrozdziale pokażę kilka trików i sztuczek, które będziesz mógł wykorzystać do przyspieszenia działania swoich makr.

Wyłączanie aktualizacji ekranu

Kiedy uruchomisz makro, możesz wygodnie wy ciągnąć się na fotelu i ze spokojem obserwować na ekranie jego postępy. Choć takie postępowanie może być do pewnego czasu ciekawe, to jednak, kiedy makro zostanie już napisane i przetestowane, wyświetlanie bieżących wyników działania może być irytujące i niepotrzebnie zwalniać działanie makra. Na szczęście, Excel pozwala na wyłączenie aktualizacji ekranu na czas działania makra, co może znacząco przyspieszyć jego działanie. Aby wyłączyć aktualizację ekranu, powinieneś użyć polecenia:

```
Application.ScreenUpdating = False
```

Jeżeli chcesz, aby użytkownicy widzieli, co się dzieje na ekranie podczas działania makra, powinieneś włączyć aktualizację ekranu za pomocą polecenia:

```
Application.ScreenUpdating = True
```

Aby zademonstrować różnicę w szybkości działania, powinieneś uruchomić makro przedstawione poniżej, którego zadaniem jest wypełnianie liczbami dużego zakresu komórek.

```
Sub FillRange()
    Dim r As Long, c As Long
    Dim Number As Long
```

```
Number = 0
For r = 1 To 50
    For c = 1 To 50
        Number = Number + 1
        Cells(r, c).Select
        Cells(r, c).Value = Number
    Next c
Next r
End Sub
```

Procedura zaznacza każdą komórkę zakresu i wpisuje do niej kolejną liczbę. Teraz na początku procedury wstaw polecenie przedstawione poniżej i ponownie uruchom procedurę.

```
Application.ScreenUpdating = False
```

Z pewnością zauważyleś, że zakres został wypełniony znacznie szybciej, a rezultaty nie były widoczne na ekranie, aż do zakończenia działania procedury i automatycznego przywrócenia aktualizacji ekranu.



Kiedy pracujesz nad testowaniem procedury i wyszukiwaniem błędów w kodzie, działanie programu może nagle zostać przerwane, bez automatycznego przywrócenia aktualizacji ekranu (tak, też mi się to zdarza...). W takiej sytuacji okno Excela pozostaje „martwe” i wydaje się, że program nie reaguje na Twoje prośby i groźby. Rozwiążanie tego problemu jest proste — przejdź do okna edytora VBE i w oknie *Immediate* wpisz polecenie:

```
Application.ScreenUpdating = True
```

Wyłączenie automatycznego przeliczania skoroszytu

Załóżmy, że masz skoroszyt zawierający wiele złożonych formuł. Możesz znacząco przyspieszyć działanie makra, jeżeli na czas jego realizacji przełączysz Excel w tryb ręcznego przeliczania skoroszytu. Kiedy makro zakończy działanie, powinieneś ponownie przełączyć Excel w tryb automatycznego przeliczania skoroszytu.

Polecenie przedstawione poniżej przełącza Excel w tryb ręcznego przeliczania skoroszytu.

```
Application.Calculation = xlCalculationManual
```

Aby przywrócić tryb automatycznego przeliczania skoroszytu, użyj polecenia:

```
Application.Calculation = xlCalculationAutomatic
```

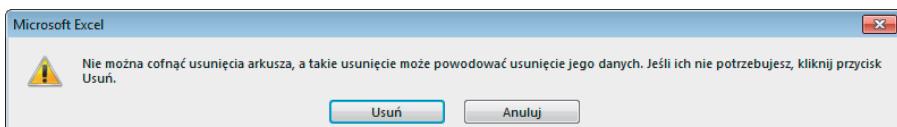


Jeżeli kod VBA Twojego makra wykorzystuje wyniki działania formuł arkuszowych, pamiętaj, że przełączenie Excela w tryb ręcznego przeliczania arkusza oznacza, iż wartości komórek nie zostaną zaktualizowane, aż do momentu, kiedy jawnie nie „poprosisz” Excela, aby to zrobił.

Wyłączanie irytujących ostrzeżeń

Jak wiesz, makra mogą automatycznie wykonywać całe mnóstwo różnych operacji. W wielu przypadkach możesz po prostu uruchomić makro i spokojnie wybrać się do kuchni i zaparzyć filiżankę swojej ulubionej kawy. Jednak niektóre operacje wykonywane przez Excel mogą spowodować wyświetlenie na ekranie komunikatu, którego potwierdzenie wymaga interakcji ze strony użytkownika. Jeśli na przykład Twoje makro próbuje usunąć arkusz, na którym znajdują się niepuste komórki, działanie makra zostanie automatycznie zatrzymane, na ekranie pojawi się komunikat przedstawiony na rysunku 14.4 i Excel będzie oczekiwał na Twoją reakcję. Obecność tego typu komunikatów oznacza, że nie możesz pozostawić Excela bez nadzoru na czas działania makra... dopóki nie poznasz pewnego triku.

Rysunek 14.4.
Możesz nakazać Excelowi zawieszenie wyświetlania takich komunikatów podczas działania makra



Oto cała sztuczka: aby uniknąć wyświetlania takich komunikatów z ostrzeżeniami, w kodzie procedury VBA umieść polecenie:

```
Application.DisplayAlerts = False
```

Excel wykonuje domyślne operacje dla tego typu komunikatów. Podczas usuwania arkusza domyślną operacją jest Delete (co właśnie przed chwilą zobaczyłeś). Jeżeli nie jesteś pewien, jaka operacja jest domyślna, przeprowadź test i przekonaj się sam.

Kiedy procedura kończy działanie, Excel automatycznie nada właściwości DisplayAlerts wartość *True* (czyli przywróci jej normalny stan). Jeżeli chcesz przywrócić wyświetlanie komunikatów przed zakończeniem działania procedury, powinieneś użyć w kodzie polecenia:

```
Application.DisplayAlerts = True
```

Upraszczanie odwołań do obiektów

Jak już sam zdążyłeś się zorientować, odwołania do obiektów mogą być bardzo rozbudowane. Przykładowo pełne, kwalifikowane odwołanie do obiektu Range może wyglądać następująco.

```
Workbooks("MójSkoroszyt.xlsx").Worksheets("Arkusz1")  
.Range("StawkaProwizji")
```

Jeżeli Twoje makro często korzysta z takiego zakresu, powinieneś rozważyć utworzenie zmiennej obiektowej za pomocą polecenia Set. Przykładowo polecenie przedstawione poniżej przypisuje obiekt Range do zmiennej obiektowej o nazwie Rate.

```
Set Rate = Workbooks("MójSkoroszyt.xlsx").Worksheets("Arkusz1")_
    .Range("StawkaProwizji")
```

Po zdefiniowaniu zmiennej obiektowej możesz zamiast długiego odwołania używać nowo utworzonej zmiennej obiektowej. Aby na przykład zmienić wartość komórki o nazwie StawkaProwizji, możesz użyć polecenia:

```
Rate.Value = .085
```

Jak widać, jest to znacznie łatwiejsze do wpisania (i zrozumienia) niż to samo polecenie w pełnej postaci.

```
Workbooks("MójSkoroszyt.xlsx").Worksheets("Arkusz1")_
    .Range("StawkaProwizji").Value = .085
```

Oprócz upraszczania kodu, zastosowanie zmiennych obiektowych powoduje również znaczące zwiększenie szybkości działania kodu Twojego makra. Wiele razy widziałem już makra, które po utworzeniu zmiennych obiektowych zwiększyły szybkość działania nawet dwukrotnie.

Deklarowanie typów zmiennych

Zazwyczaj nie musisz się martwić o typ danych, który przypisujesz do zmiennej. Excel potrafi się tym doskonale zająć. Jeżeli masz zmienną o nazwie *MyVar*, możesz do niej przypisać dowolną liczbę, a później, w dalszej części procedury, możesz do tej samej zmiennej przypisać na przykład ciąg tekstu.



Jeżeli chcesz, aby Twoje procedury VBA działały tak szybko, jak to tylko możliwe (i aby przy okazji uniknąć kilku potencjalnych i naprawdę paskudnych problemów), powinieneś zawsze poinformować Excel o tym, jakie typy danych będą przypisywane do poszczególnych zmiennych. Takie postępowanie jest nazywane *deklarowaniem typów zmiennych* (więcej szczegółowych informacji na ten temat znajdziesz w rozdziale 7.). Powinieneś jak najszybciej wyrobić sobie nawyk deklarowania wszystkich zmiennych, których używasz w swoich programach.

Ogólnie rzecz biorąc, powinieneś zawsze używać takich typów danych, które wystarczą do obsługi Twoich danych przy wykorzystaniu jak najmniejszej liczby bajtów pamięci. Kiedy VBA przetwarza dane, szybkość działania programu zależy od liczby bajtów, jakie VBA ma do „przerobienia”. Innymi słowy, im mniej bajtów zajmują dane, tym szybciej VBA może je przetwarzać. Wyjątkiem od tej reguły są dane typu *Integer* — jeżeli szybkość działania programu jest czynnikiem krytycznym, powinieneś zawsze stosować dane typu *Long*.

Jeżeli używasz zmiennych obiektowych (takich jakie opisywałem w poprzednim podrozdziale), możesz zadeklarować taką zmienną jako zmienną określonego typu obiektowego. A oto przykład takiej deklaracji.

```
Dim Rate as Range  
Set Rate = Workbooks("MójSkoroszyt.xlsx").Worksheets("Arkusz1") _  
.Range("StawkaProwizji")
```

Zastosowanie struktury With-End With

Czy chcesz ustawić szereg właściwości wybranego obiektu? Twój kod będzie działał znacznie szybciej, gdy użyjesz struktury With-End With. Dodatkową zaletą zastosowania tej struktury jest znaczne zwiększenie czytelności kodu.

Fragment kodu przedstawiony poniżej nie wykorzystuje struktury With-End With.

```
Selection.HorizontalAlignment = xlCenter  
Selection.VerticalAlignment = xlCenter  
Selection.WrapText = True  
Selection.Orientation = 0  
Selection.ShrinkToFit = False  
Selection.MergeCells = False
```

A teraz ten sam fragment kodu, ale zapisany z użyciem struktury With-End With.

```
With Selection  
    .HorizontalAlignment = xlCenter  
    .VerticalAlignment = xlCenter  
    .WrapText = True  
    .Orientation = 0  
    .ShrinkToFit = False  
    .MergeCells = False  
End With
```

Jeżeli przedstawiona struktura wydaje Ci się znajoma, jest tak prawdopodobnie dla tego, że rejestrator makr używa struktury With-End With w każdej sytuacji, w której jest to możliwe, a poza tym struktura taka pojawiła się już wcześniej w kilku przykładach w tym rozdziale.

Część IV

Komunikacja z użytkownikiem

	A	B	C	D	E	F	G	H	I	J	K	L
1	791	973	168	797	951	304	989	141	752	653	932	58
2	646	388	479	692	281	771	715	79	85	714	71	70
3	721	534	426	197	233	14	191	534	570	210	234	76
4	333	961	859							351	206	727
5	939	795	897							216	844	619
6	638	243	424							89	846	249
7	668	88	587							538	11	716
8	908	192	302							270	717	391
9	379	994	748							634	419	404
10	741	14	872	706	665	342	833	231	118	151	157	26
11	796	346	885	136	465	164	198	144	852	24	812	97
12	985	166	764	985	966	27	19	953	105	729	801	19
13	891	226	751	23	351	877	955	185	789	34	319	84
14	737	356	856	725	694	991	677	325	954	285	827	3
15	887	965	351	771	462	474	373	308	293	906	553	59
16	345	540	277	541	10	756	247	23	638	899	989	53
17	257	146	938	620	84	99	17	813	462	346	393	59
18	407	925	406	295	323	91	420	973	378	904	290	68
19	723	600	17	57	753	148	705	252	643	312	969	44
20	770	970	872	101	294	14	138	351	649	883	985	70

Arkusz1

W tej części...

- ✓ przekonasz się, jak możesz oszczędzać czas, używając wbudowanych okien dialogowych zamiast formularzy *UserForm*,
- ✓ poznasz sposoby tworzenia własnych okien dialogowych (znanych również pod nazwą formularzy *UserForm*),
- ✓ dowiesz się, jak wykorzystywać formanty okien dialogowych, takie jak przyciski, listy i pola wyboru,
- ✓ znajdziesz wiele użytecznych wskazówek i trików ułatwiających tworzenie własnych okien dialogowych,
- ✓ zobaczysz, jak można modyfikować interfejs użytkownika programu Excel tak, aby ułatwić uruchamianie własnych makr.

Rozdział 15

Proste okna dialogowe

W tym rozdziale:

- dowiesz się, jak oszczędzać czas, stosując inne rozwiązania zamiast formularzy *UserForm*,
- nauczysz się wykorzystywać funkcje *InputBox* oraz *MsgBox* do pobierania danych od użytkownika,
- zobaczysz, w jaki sposób możesz pobrać nazwę pliku i ścieżkę od użytkownika,
- poznasz sposoby pobierania nazw folderów od użytkownika,
- dowiesz się, jak napisać kod VBA, który będzie wykonywał polecenia Wstążki wyświetlające wbudowane okna dialogowe programu Excel.

Nie sposób pracować z programem Excel i nie natknąć się na okna dialogowe, które bardzo często pojawiają się na ekranie. Excel — podobnie jak większość innych programów działających pod kontrolą systemu Windows — wykorzystuje okna dialogowe do pobierania danych, potwierdzania zamiarów użytkownika czy wyświetlania na ekranie komunikatów o błędach i innych informacji. Jeżeli zajmujesz się projektowaniem makr VBA, możesz utworzyć swoje własne okna dialogowe, które będą działały w bardzo podobny sposób jak wbudowane okna dialogowe programu Excel. Takie niestandardowe okna dialogowe, tworzone przez użytkownika, noszą w języku VBA nazwę formularzy *UserForm*.

W tym rozdziale nie znajdziesz żadnych informacji na temat tworzenia formularzy *UserForm*. Zamiast tego poznasz kilka ciekawych technik, których możesz używać **zamiast** formularzy *UserForm*. Z kolei więcej szczegółowych informacji na temat formularzy *UserForm* znajdziesz w rozdziałach od 16. do 18.

Co zamiast formularzy UserForm?

Niektóre makra VBA utworzone przez Ciebie będą się zachowywały w identyczny sposób za każdym razem, kiedy je uruchomisz. Możesz na przykład napisać makro, które wpisuje do wybranego zakresu arkusza listę nazwisk pracowników Twojej firmy. Takie makro za każdym razem generuje taki sam zestaw danych i nie wymaga żadnej interakcji ze strony użytkownika.

Z drugiej strony, możesz również tworzyć makra, których sposób działania będzie się zmieniał w zależności od okoliczności, takich jak na przykład zestaw opcji wybieranych przez użytkownika. W takich sytuacjach Twoje makro może wykorzystywać niestandardowe okna dialogowe pozwalające użytkownikowi w prosty sposób wprowadzać dane, które następnie będą przetwarzane przez makro i na ich podstawie makro będzie wykonywało takie czy inne operacje.

Formularze *UserForm* są — oczywiście — bardzo efektownym i użytecznym rozwiązańiem, ale ich tworzenie zajmuje sporo czasu. Zanim w kolejnym rozdziale rozpocznę omawianie tworzenia takich formularzy, musisz poznać kilka innych ciekawych rozwiązań, które często pozwolą zaoszczędzić sporo czasu.

VBA pozwala na wyświetlanie na ekranie kilku różnych typów okien dialogowych, których możesz często z powodzeniem używać zamiast formularzy *UserForm*. Takie wbudowane okna dialogowe możesz nawet w pewnym zakresie dostosowywać do własnych potrzeb, ale pod tym względem nie oferują one takiej elastyczności jak formularze *UserForm* — nie zmienia to jednak faktu, że w wielu przypadkach znakomicie sprawdzają się w swojej roli.

W tym rozdziale poznasz następujące elementy:

- ✓ funkcję `MsgBox`,
- ✓ funkcję `InputBox`,
- ✓ metodę `GetOpenFilename`,
- ✓ metodę `GetSaveAsFilename`,
- ✓ metodę `FileDialog`.

Oprócz tego dowiesz się również, w jaki sposób wykorzystać język VBA do wyświetlania na ekranie niektórych wbudowanych, standardowych okien dialogowych programu Excel, czyli inaczej mówiąc, okien dialogowych, które w Excelu używane są do pobierania informacji od Ciebie.

Funkcja `MsgBox`

Prawdopodobnie już co najmniej kilka razy miałeś okazję spotkać się z funkcją `MsgBox` języka VBA — używałem jej w wielu przykładach w poprzednich rozdziałach tej książki. Funkcja `MsgBox`, która pobiera argumenty przedstawione w tabeli 15.1, pozwala w bardzo wygodny sposób wyświetlać informacje i pobierać proste dane od użytkownika. Dzieje się tak dlatego, że `MsgBox` to funkcja, a jak pamiętasz, funkcje po wywołaniu mogą zwracać wyniki swojego działania. W przypadku funkcji `MsgBox` pobieranie wartości, która zostanie następnie zwrocona do procedury wywołującej, odbywa się za pośrednictwem okna dialogowego. Dalej w tym podrozdziale zobaczymy, jak to działa.

Tabela 15.1. Argumenty funkcji `MsgBox`

Argument	Opis
<code>komunikat</code>	Tekst, który Excel wyświetla w oknie komunikatu.
<code>przyciski</code>	Wartość liczbową decydującą o tym, które przyciski oraz ikony pojawią się w oknie dialogowym (argument opcjonalny).
<code>tytuł</code>	Tekst, który zostanie wyświetlony na pasku tytułowym okna dialogowego (argument opcjonalny).

Poniżej zamieszczam uproszczoną wersję składni wywołania funkcji MsgBox.

```
MsgBox(komunikat[, przyciski][, tytuł])
```

Wyświetlanie prostych okien dialogowych

Funkcji MsgBox możesz używać na dwa sposoby.

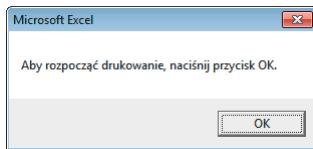
- ✓ **Aby wyświetlić na ekranie komunikat dla użytkownika.** W takiej sytuacji nie musisz przejmować się wynikami działania zwracanymi przez funkcję.
- ✓ **Aby otrzymać od użytkownika odpowiedź.** W tym przypadku wynik działania zwracany przez funkcję będzie dla Ciebie istotny. Wartość zwarcana przez funkcję zależy od tego, który przycisk w oknie dialogowym zostanie naciśnięty przez użytkownika.

Jeżeli używasz funkcji MsgBox tylko do wyświetlania komunikatów na ekranie, argumentów wywołania funkcji w kodzie programu nie powinieneś umieszczać w nawiasach. Procedura, której kod zamieszczam niżej, po prostu wyświetla odpowiedni komunikat i nie zwraca żadnego wyniku. Kiedy na ekranie pojawi się okno dialogowe z komunikatem, działanie programu zostaje zawieszone dopóty, dopóki użytkownik nie naciśnie przycisku OK.

```
Sub MsgBoxDemo()
    MsgBox "Aby rozpocząć drukowanie, naciśnij przycisk OK."
    Sheets("wyniki").PrintOut
End Sub
```

Na rysunku 15.1 przedstawiam wygląd okna dialogowego, które pojawi się na ekranie po uruchomieniu procedury. W naszym przypadku proces drukowania arkusza rozpocznie się, kiedy użytkownik naciśnie przycisk OK. Zauważysz, że nie ma tutaj żadnej możliwości anulowania operacji drukowania? W kolejnym podrozdziale dowiesz się, jak to zrobić.

Rysunek 15.1.
Proste okno dialogowe



Pobieranie odpowiedzi z okna dialogowego

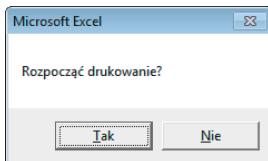
Jeżeli wyświetlasz na ekranie okno dialogowe zawierające oprócz przycisku OK również inne przyciski, prawdopodobnie będziesz chciał również wiedzieć, który przycisk został naciśnięty przez użytkownika. Masz szczęście, ponieważ funkcja MsgBox może zwracać wartość reprezentującą naciśnięty przycisk. Wynik działania funkcji MsgBox możesz przypisać do zmiennej.

W procedurze, której kod przedstawiem poniżej, wykorzystuję pewne wbudowane wartości stałe (szczegółowo opisane nieco później w tabeli 15.2), dzięki którym mogę wygodnie pracować z wartościami zwracanymi przez funkcję MsgBox.

```
Sub GetAnswer()
    Dim Ans As Integer
    Ans = MsgBox("Rozpocząć drukowanie?", vbYesNo)
    Select Case Ans
        Case vbYes
            ActiveSheet.PrintOut
        Case vbNo
            MsgBox "Operacja drukowania została anulowana!"
    End Select
End Sub
```

Na rysunku 15.2 przedstawiam wygląd takiego okna dialogowego. Kiedy procedura zostanie uruchomiona, zmiennej *Ans* zostaje przypisana wartość (odpowiednio) *vbYes* lub *vbNo*, w zależności od tego, który przycisk w oknie dialogowym zostanie naciśnięty przez użytkownika. Wyrażenie *Select Case* wykorzystuje wartość zmiennej *Ans* do wybrania operacji, która zostanie wykonana przez procedurę.

Rysunek 15.2.
Proste okno dialogowe zawierające dwa przyciski



Jeśli trzeba, możesz wykorzystać wartość zwracaną przez funkcję *MsgBox* bez przypisywania jej do zmiennej, tak jak to zostało przedstawione na przykładzie zamieszczonym poniżej.

```
Sub GetAnswer2()
    If MsgBox("Rozpocząć drukowanie?", vbYesNo) = vbYes Then
        ' ... [...kod wykonywany, gdy zostanie naciśnięty przycisk Tak...]
    Else
        ' ... [...kod wykonywany, gdy zostanie naciśnięty przycisk Nie...]
    End If
End Sub
```

Dostosowywanie wyglądu okien dialogowych do własnych potrzeb

Elastyczność argumentów reprezentujących przyciski pozwala na łatwe dostosowywanie wyglądu okien dialogowych do własnych potrzeb. Możesz wybierać przyciski, które będą wyświetlane w oknie, wybierać ikony pojawiające się obok komunikatów oraz decydować o tym, który przycisk będzie wyróżniany jako domyślny (przycisk domyślny zostaje automatycznie „kliknięty”, jeżeli użytkownik naciśnie klawisz *Enter*).

W tabeli 15.2 przedstawiam wybrane wbudowane wartości stałej, których możesz używać jako argumentów wywołania funkcji *MsgBox* reprezentujących przyciski. Oczywiście, jeśli chcesz, możesz zamiast tego korzystać z odpowiednich wartości numerycznych (ale myślę, że zastosowanie stałych jest zdecydowanie wygodniejsze).

Tabela 15.2. Stałe reprezentujące przyciski obsługiwane przez funkcję MsgBox

Stała	Wartość	Opis
vbOKOnly	0	Wyświetla jedynie przycisk <i>OK</i> .
vbOKCancel	1	Wyświetla przyciski <i>OK</i> i <i>Anuluj</i> .
vbAbortRetryIgnore	2	Wyświetla przyciski <i>Przerwij</i> , <i>Ponów próbę</i> i <i>Ignoruj</i> .
vbYesNoCancel	3	Wyświetla przyciski <i>Tak</i> , <i>Nie</i> i <i>Anuluj</i> .
vbYesNo	4	Wyświetla przyciski <i>Tak</i> i <i>Nie</i> .
vbRetryCancel	5	Wyświetla przyciski <i>Ponów próbę</i> i <i>Anuluj</i> .
vbCritical	16	Wyświetla ikonę komunikatu krytycznego.
vbQuestion	32	Wyświetla ikonę pytania.
vbExclamation	48	Wyświetla ikonę komunikatu ostrzegawczego.
vbInformation	64	Wyświetla ikonę komunikatu informacyjnego.
vbDefaultButton1	0	Domyślny jest pierwszy przycisk.
vbDefaultButton2	256	Domyślny jest drugi przycisk.
vbDefaultButton3	512	Domyślny jest trzeci przycisk.
vbDefaultButton4	768	Domyślny jest czwarty przycisk.

Aby jako argumentów wywołania funkcji użyć więcej niż jednej stałej z tego zestawu, powinieneś połączyć je ze sobą za pomocą operatora +. Gdy chcesz na przykład wyświetlić na ekranie okno dialogowe zawierające przyciski *Tak* i *Nie* oraz ikonę komunikatu ostrzegawczego, powinieneś jako drugiego argumentu wywołania funkcji MsgBox użyć wyrażenia:

```
vbYesNo + vbExclamation
```

Jeżeli jednak chcesz, aby kod Twojego programu z takiego czy innego powodu był nieco mniej zrozumiały i mniej przejrzysty, możesz zamiast powyższego wyrażenia użyć liczby 52 (czyli inaczej mówiąc, wartości wyrażenia 4 + 48).

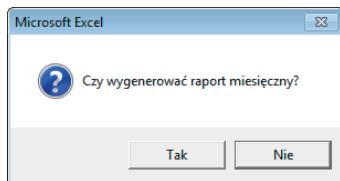
W przykładzie przedstawionym poniżej wykorzystuję kombinację wartości stałych do wyświetlania okna dialogowego zawierającego przyciski *Tak* i *Nie* (vbYesNo) oraz ikonę pytania (vbQuestion). Dodatkowo stała vbDefaultButton2 jest wykorzystywana do zdefiniowania przycisku *Nie* jako przycisku domyślnego — czyli inaczej mówiąc, przycisku, który zostanie automatycznie „kliknięty”, kiedy użytkownik naciśnie klawisz *Enter*. Dla uproszczenia i zwiększenia przejrzystości kodu najpierw przypisuję sumę stałych do zmiennej *Config*, a potem używam tej zmiennej jako drugiego argumentu wywołania funkcji MsgBox.

```
Sub GetAnswer3()
    Dim Config As Integer
    Dim Ans As Integer
    Config = vbYesNo + vbQuestion + vbDefaultButton2
```

```
Ans = MsgBox("Czy wygenerować raport miesięczny?", Config)
If Ans = vbYes Then RunReport
End Sub
```

Na rysunku 15.3 przedstawiam wygląd okna dialogowego, które zostaje wyświetcone przez Excel po uruchomieniu procedury GetAnswer3. Jeżeli użytkownik naciśnie przycisk *Tak*, program wywołuje procedurę RunReport (której kod nie został tutaj przedstawiony). Jeżeli użytkownik naciśnie przycisk *Nie* (lub naciśnie klawisz *Enter*), program zakończy działanie, nie podejmując żadnej innej akcji. Ponieważ w wywołaniu funkcji MsgBox został pominięty argument reprezentujący tytuł okna dialogowego, Excel używa tytułu domyślnego, *Microsoft Excel*.

Rysunek 15.3.
Drugi argument funkcji MsgBox odpowiada za to, jakie przyciski pojawią się w oknie dialogowym



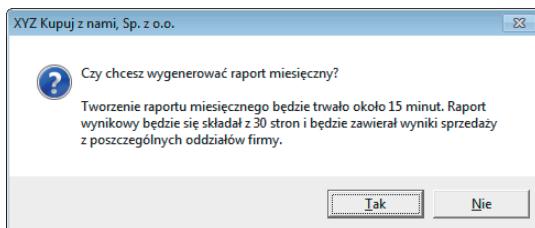
Procedura zamieszczona poniżej to kolejny przykład zastosowania funkcji MsgBox.

```
Sub GetAnswer4()
    Dim Msg As String, Title As String
    Dim Config As Integer, Ans As Integer
    Msg = "Czy chcesz wygenerować raport miesięczny?"
    Msg = Msg & vbCrLf & vbCrLf
    Msg = Msg & "Tworzenie raportu miesięcznego "
    Msg = Msg & "będzie trwało około 15 minut. Raport "
    Msg = Msg & "wynikowy będzie się składał z 30 stron "
    Msg = Msg & "i będzie zawierał wyniki sprzedaży"
    Msg = Msg & "z poszczególnych oddziałów firmy."
    Title = "XYZ Kupuj z nami, Sp. z o.o."
    Config = vbYesNo + vbQuestion
    Ans = MsgBox(Msg, Config, Title)
    If Ans = vbYes Then RunReport
End Sub
```

W tym przykładzie zilustrowany został efektywny sposób tworzenia i wyświetlania w oknie dialogowym długich, złożonych komunikatów. Do utworzenia komunikatu w serii kolejnych poleceń używamy zmiennej (*Msg*) oraz operatora konkatenacji (&). Stała *vbCrLf* powoduje wstawienie znaku łamania wiersza, który powoduje rozpoczęcie pisania od nowego wiersza (kiedy chcesz wstawić pusty wiersz, powinieneś użyć tej stałej dwukrotnie). Do zmiany tytułu okna użyty został trzeci argument wywołania funkcji *MsgBox*. Na rysunku 15.4 przedstawiam wygląd okna dialogowego, które pojawi się na ekranie po wywołaniu tej procedury w Excelu.

Rysunek 15.4.

W tym oknie dialogowym, utworzonym za pomocą funkcji MsgBox, wyświetlany jest tytuł, ikona pytania, komunikat oraz dwa przyciski



W przykładach przedstawionych powyżej wartości zwarcane przez funkcję MsgBox były reprezentowane przez stałe, takie jak vbYes czy vbNo. Oprócz wymienionych stałych, funkcja MsgBox obsługuje jeszcze kilka innych, które zostały przedstawione w tabeli 15.3.

Tabela 10.3. Stałe stosowane jako wartości zwarcane przez funkcję MsgBox

Stała	Wartość	Znaczenie
vbOK	1	Użytkownik naciągnął przycisk <i>OK</i> .
vbCancel	2	Użytkownik naciągnął przycisk <i>Anuluj</i> .
vbAbort	3	Użytkownik naciągnął przycisk <i>Przerwij</i> .
vbRetry	4	Użytkownik naciągnął przycisk <i>Ponów próbę</i> .
vbIgnore	5	Użytkownik naciągnął przycisk <i>Ignoruj</i> .
vbYes	6	Użytkownik naciągnął przycisk <i>Tak</i> .
vbNo	7	Użytkownik naciągnął przycisk <i>Nie</i> .

To w zasadzie wszystko, co na początek powinieneś wiedzieć na temat funkcji MsgBox. Pamiętaj jednak, aby używać jej ostrożnie. Zazwyczaj nie ma żadnych rozsądnych powodów do wyświetlanego komunikatów, które nie wnoszą niczego do działania programu. Przykładowo większość użytkowników z pewnością będzie zirytowana, widząc wielokrotnie w ciągu dnia komunikat: *Dzień dobry, serdecznie dziękujemy za uruchomienie naszego skoroszytu „Planuj budżet razem z nami!”*.

Funkcja InputBox

Funkcja InputBox języka VBA jest bardzo przydatna do pobierania od użytkownika pojedynczych elementów. Takie informacje mogą mieć postać wartości liczbowych, ciągów znaków czy nawet odwołań do zakresów komórek. Kiedy musisz pobrać od użytkownika tylko pojedynczą wartość, funkcja InputBox z pewnością będzie doskonałym rozwiązaniem alternatywnym do tworzenia formularza *UserForm*.

Składnia funkcji InputBox

Poniżej przedstawiam uproszczoną wersję składni wywołania funkcji InputBox.

```
InputBox(komunikat[, tytuł][, wartość_domyślna])
```

Argumenty wywołania funkcji InputBox zostały przedstawione w tabeli 15.4.

Tabela 15.4. Argumenty funkcji InputBox

Argument	Opis
komunikat	Tekst wyświetlany w oknie wprowadzania danych.
tytuł	Tekst, który zostanie wyświetlony na pasku tytułowym okna dialogowego (argument opcjonalny).
wartość_domyślna	Domyślana wartość wyświetlana w oknie dialogowym (argument opcjonalny).

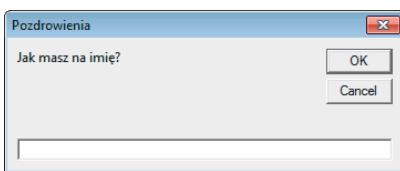
Przykład zastosowania funkcji InputBox

Poniżej przedstawiam przykład wyrażenia, które ilustruje sposób użycia funkcji InputBox.

```
TheName = InputBox("Jak masz na imię?", "Pozdrowienia")
```

Kiedy wykonasz takie polecenie języka VBA, Excel wyświetli na ekranie okno dialogowe, przedstawione na rysunku 15.5. Zwróć uwagę, że w tym przykładzie wykorzystujemy tylko dwa pierwsze argumenty wywołania tej funkcji i nie definiujemy wartości domyślnej. Kiedy użytkownik wprowadzi wybraną wartość i naciśnie przycisk OK, wpisana wartość zostanie przypisana do zmiennej *TheName*.

Rysunek 15.5.
Okno dialogowe wyświetlone za pomocą funkcji InputBox



W przykładzie przedstawionym poniżej do zdefiniowania wartości domyślnej wykorzystany został trzeci argument wywołania funkcji InputBox. W tym przypadku wartością domyślną jest nazwa użytkownika Excela (wartość właściwości *UserName* obiektu *Application*).

```
Sub GetName()
    Dim TheName As String
    TheName = InputBox("Jak się nazywasz?",
                      "Pozdrowienia", Application.UserName)
End Sub
```

W oknie dialogowym utworzonym za pomocą funkcji `InputBox` zawsze wyświetlany jest przycisk *Cancel*. Jeżeli użytkownik naciśnie przycisk *Cancel*, funkcja `InputBox` zwraca pusty ciąg znaków.

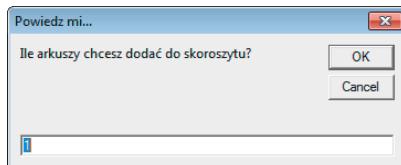


Funkcja `InputBox` języka VBA zawsze zwraca ciąg znaków, zatem jeżeli potrzebna Ci jest wartość liczbowa, Twój program będzie musiał wykonać kilka dodatkowych testów. W przykładzie przedstawionym poniżej wykorzystuję funkcję `InputBox` do pobrania liczby. Do sprawdzenia, czy wprowadzony ciąg znaków reprezentuje poprawną wartość liczbową, używam funkcji `IsNumeric`. Jeżeli wprowadzony ciąg znaków zawiera liczbę, wszystko jest w porządku. Jeżeli jednak wprowadzony ciąg znaków nie może być zinterpretowany jako wartość liczbowa, program wyświetla na ekranie okno dialogowe zawierające odpowiedni komunikat o błędzie.

```
Sub AddSheets()
    Dim Prompt As String
    Dim Caption As String
    Dim DefValue As Integer
    Dim NumSheets As String
    Prompt = "Ile arkuszy chcesz dodać do skoroszytu?"
    Caption = "Powiedz mi..."
    DefValue = 1
    NumSheets = InputBox(Prompt, Caption, DefValue)
    If NumSheets = "" Then Exit Sub 'Operacja została anulowana
    If IsNumeric(NumSheets) Then
        If NumSheets > 0 Then Sheets.Add Count:=NumSheets
    Else
        MsgBox "Podajeś niepoprawną wartość liczbową"
    End If
End Sub
```

Na rysunku 15.6 przedstawiono wygląd okna dialogowego, które pojawia się na ekranie po uruchomieniu opisywanej procedury.

Rysunek 15.6.
Kolejny przy-
kład zastoso-
wania funkcji
`InputBox`



Inny rodzaj okna dialogowego `InputBox`

Informacje przedstawiane do tej pory w tym podrozdziale odnosiły się wyłącznie do funkcji `InputBox` języka VBA. Oprócz tej funkcji, masz jednak jeszcze dostęp do metody `InputBox` obiektu `Application`.

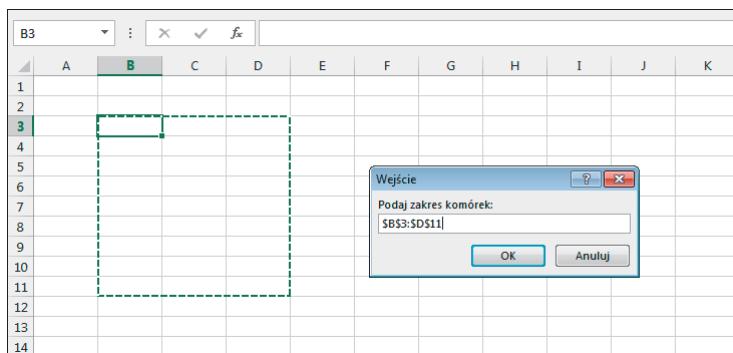
Jedną z wielu zalet wykorzystywania metody `InputBox` obiektu `Application` jest to, że dzięki temu Twój program może poprosić o podanie zakresu komórek, a użytkownik może taki zakres zdefiniować poprzez zaznaczenie odpowiedniego obszaru przy użyciu myszy.

Poniżej przedstawiam przykład prostej procedury, która prosi użytkownika o podanie zakresu komórek arkusza.

```
Sub GetRange()
    Dim Rng As Range
    On Error Resume Next
    Set Rng = Application.InputBox _
        (prompt:="Podaj zakres komórek:", Type:=8)
    If Rng Is Nothing Then Exit Sub
    MsgBox "Zaznaczyłeś następujący zakres komórek: " & Rng.Address
End Sub
```

Na rysunku 15.7 przedstawiony został sposób działania tej procedury.

Rysunek 15.7.
Zastosowanie metody InputBox obiektu Application do pobrania zakresu komórek



W tym prostym przykładzie przed zakończeniem działania procedury wyświetla adres zakresu komórek zaznaczonego przez użytkownika. W prawdziwej aplikacji kod procedury mógłby — oczywiście — wykonywać na zaznaczonym zakresie jakieś użyteczne operacje. Kolejną zaletą takiego rozwiązania jest to, że w takiej sytuacji Excel bierze na siebie całą obsługę błędów. Jeżeli wprowadzisz wartość, która z takiej czy innej przyczyny nie jest poprawna, Excel poinformuje o tym i pozwoli spróbować jeszcze raz.

Metoda `Application.InputBox` jest podobna do funkcji `InputBox` języka VBA, ale jednak obie funkcje różnią się od siebie. Więcej szczegółowych informacji na ten temat znajdziesz w pomocy systemowej programu Excel.

Metoda `GetOpenFilename`

Jeżeli Twoja procedura VBA musi poprosić użytkownika o podanie nazwy pliku, możesz do tego celu użyć funkcji `InputBox` i pozwolić użytkownikowi na wpisane takiej nazwy bezpośrednio z klawiatury. Jednak okno dialogowe generowane przez taką funkcję zazwyczaj nie jest w takiej sytuacji najlepszym rozwiązaniem, ponieważ większość użytkowników może mieć problemy z zapamiętaniem i poprawnym wpisaniem tych wszystkich ścieżek, ukośników, folderów czy nazw plików i ich rozszerzeń. Innymi słowy, wpisywanie ścieżki i nazwy pliku w taki sposób jest zdecydowanie uciążliwe i zbyt łatwo można tutaj popełnić błąd.

Znacznie lepszym rozwiązańiem takiego zagadnienia jest zastosowanie metody `GetOpenFilename` obiektu `Application`, która gwarantuje, że do Twojego programu zawsze zostanie przekazana poprawna, sprawdzona nazwa pliku wraz z całą ścieżką do niego prowadzącą. Metoda `GetOpenFilename` wyświetla na ekranie dobrze Ci znane okno dialogowe *Otwieranie*.



Metoda `GetOpenFilename` nie powoduje w żaden sposób otwarcia wybranego przez użytkownika pliku. Zamiast tego zwraca jedynie poprawną nazwę wybranego pliku w postaci ciągu znaków, która następnie może być w dowolny sposób przetwarzana w Twoim programie.

Składnia metody GetOpenFilename

Oficjalna składnia metody `GetOpenFilename` jest następująca:

```
obiekt.GetOpenFilename ([filtrPlików], [indeksFiltru], [tytuł], [tekstPrzycisku],
→[wielePlików])
```

Metoda `GetOpenFilename` może pobierać szereg opcjonalnych argumentów, które zostały przedstawione w tabeli 15.5.

Tabela 15.5. Argumenty metody GetOpenFilename

Argument	Opis
<code>filtrPlików</code>	Określa rodzaj plików, które pojawiają się w oknie dialogowym (na przykład <code>*.txt</code>). Jeśli trzeba, możesz zdefiniować kilka filtrów, spośród których użytkownik może dokonać wyboru.
<code>indeksFiltru</code>	Określa, który z podanych filtrów będzie domyślnie wyświetlany w oknie dialogowym.
<code>tytuł</code>	Tekst, który zostanie wyświetlony na pasku tytułowym okna dialogowego (argument opcjonalny).
<code>tekstPrzycisku</code>	Argument ignorowany (ma zastosowanie tylko dla Excela pracującego na komputerach Macintosh).
<code>wielePlików</code>	Jeżeli ten argument ma wartość <code>True</code> , użytkownik może zaznaczać wiele plików jednocześnie.

Przykład zastosowania metody GetOpenFilename

Argument `fileFilter` określa, co pojawia się na liście rozwijanej *Pliki typu* w oknie dialogowym *Otwieranie*. Argument ten składa się z nazw filtrów i symboli wieloznacznych definiujących rozszerzenia plików, oddzielonych od siebie przecinkami. Jeżeli ten argument zostanie pominięty w wywołaniu metody, przyjmuje następującą domyślną wartość.

```
Wszystkie pliki (*.*), *.*
```

Zwróć uwagę, że definicja filtra składa się z dwóch części, oddzielonych znakiem przecinka:

Wszystkie pliki (*.*)

oraz

.

Pierwsza część definicji filtra jest wyświetlana na liście rozwijanej *Pliki typu*, a druga część reprezentuje maskę nazw plików, które będą wyświetlane w oknie dialogowym. Przykładowo zapis *.* oznacza, że wyświetlane będą *wszystkie pliki*.

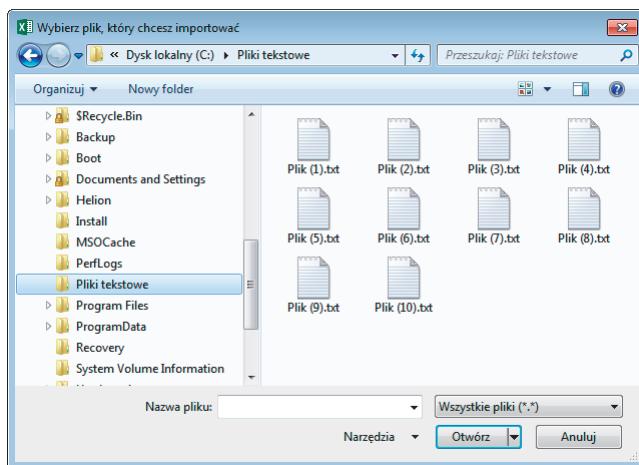
Kod procedury przedstawiony poniżej wyświetla na ekranie okno dialogowe, które prosi użytkownika o podanie nazwy pliku. Procedura definiuje pięć filtrów nazw plików. Zwróć uwagę na fakt, że w definicji filtrów zastosowane zostały znaki kontynuacji wiersza polecenia VBA; takie rozwiązanie pozwala na uproszczenie zapisu argumentu i zwiększenie przejrzystości tego dosyć złożonego polecenia.

```
Sub GetImportFileName ()  
    Dim FInfo As String  
    Dim FilterIndex As Integer  
    Dim Title As String  
    Dim FileName As Variant  
  
    ' Tworzenie listy filtrów plików  
    FInfo = "Pliki tekstowe (*.txt),*.txt," &  
        "Pliki arkusza kalkulacyjnego firmy Lotus (*.prn),*.prn," &  
        "Pliki używające przecinka jako separatora (*.csv),*.csv," &  
        "Pliki ASCII (*.asc),*.asc," &  
        "Wszystkie pliki (*.*),*.*"  
  
    ' Definiowanie filtra domyślnego (*.*)  
    FilterIndex = 5  
  
    ' Tworzenie tytułu okna dialogowego  
    Title = "Wybierz plik, który chcesz importować"  
  
    ' Pobieranie nazwy pliku  
    FileName = Application.GetOpenFilename(FInfo, _  
        FilterIndex, Title)  
  
    ' Obsługa wartości pobranej z okna dialogowego  
    If FileName = False Then  
        MsgBox "Nie wybrałeś żadnego pliku."  
    Else  
        MsgBox "Wybrałeś plik " & FileName  
    End If  
End Sub
```

Na rysunku 15.8 przedstawiam okno dialogowe, które Excel wyświetla po uruchomieniu tej procedury. Dokładny wygląd okna może się nieco różnić, w zależności od wersji systemu Windows, z której korzystasz.

Rysunek 15.8.

Metoda `GetOpenFilename` pozwala na wyświetlenie konfigurowalnego okna dialogowego i zwraca ścieżkę oraz nazwę wybranego przez użytkownika pliku; metoda nie powoduje otwarcia pliku



W rzeczywistej aplikacji po pobraniu nazwy pliku możesz wykonywać różne użyteczne operacje. Aby na przykład otworzyć wybrany plik, możesz użyć polecenia podobnego do przedstawionego poniżej.

```
Workbooks.Open FileName
```



Zwróć uwagę na fakt, że zmienna `FileName` jest zadeklarowana jako zmienna typu `Variant`. Jeżeli użytkownik naciśnie przycisk `Anuluj`, zmienna `FileName` zawiera wartość logiczną typu `Boolean` (`False`). Kiedy jednak użytkownik wybierze plik, zmienna ta zawiera ciąg znaków reprezentujący ścieżkę i nazwę pliku. Zastosowanie deklaracji typu `Variant` pozwala na obsługę obu typów danych.

Pamiętaj, że w zależności od tego, z jakiej wersji systemu Windows korzystasz, wygląd tego okna dialogowego może się różnić od przedstawionego w książce.

Metoda `GetSaveAsFilename`

Metoda `GetSaveAsFilename` działa bardzo podobnie do metody `GetOpenFilename`, z tym, że zamiast okna dialogowego *Otwieranie* wyświetla na ekranie okno dialogowe *Zapisywanie jako*. Metoda `GetSaveAsFilename` pobiera ścieżkę i nazwę pliku od użytkownika, ale nie robi niczego więcej z tą informacją. Jeżeli na przykład chcesz naprawdę zapisać plik na dysku, będziesz musiał dodać odpowiedni fragment kodu.

Składnia metody `GetSaveAsFilename` jest następująca:

```
obiekt.GetSaveAsFilename ([nazwaDomyślna], [filtrPlików], [indeksFiltra], [tytuł],  
[tekstPrzycisku])
```

Metoda `GetSaveAsFilename` może pobierać szereg opcjonalnych argumentów, które zostały przedstawione w tabeli 15.6.

Tabela 15.6. Argumenty metody GetSaveAsFilename

Argument	Opis
nazwaDomyślna	Określa domyślną nazwę pliku, która się pojawi w polu <i>Nazwa pliku</i> .
filtrPlików	Określa rodzaj plików, które pojawiają się w oknie dialogowym (na przykład *.txt). Jeśli trzeba, możesz zdefiniować kilka filtrów, spośród których użytkownik może dokonać wyboru.
indeksFiltra	Określa, który z podanych filtrów będzie domyślnie wyświetlany w oknie dialogowym.
tytuł	Tekst, który zostanie wyświetlony na pasku tytułowym okna dialogowego (argument opcjonalny).
tekstPrzycisku	Argument ignorowany (ma zastosowanie tylko dla Excela pracującego na komputerach Macintosh).

Pobieranie nazwy folderu

Czasami nie jest Ci potrzebna nazwa pliku, ale zamiast tego chciałbyś pobrać tylko nazwę folderu. W takiej sytuacji zalecanym rozwiążaniem jest użycie obiektu FileDialog.

Procedura przedstawiona poniżej wyświetla okno dialogowe, które pozwala użytkownikowi na wybranie żądanego folderu. Nazwa wybranego folderu lub komunikat o anulowaniu operacji zostają później wyświetlane na ekranie za pomocą funkcji MsgBox.

```
Sub GetAFolder()
    With Application.FileDialog(msoFileDialogFolderPicker)
        .InitialFileName = Application.DefaultFilePath & "\"
        .Title = "Wybierz lokalizację kopii zapasowej"
        .Show
        If .SelectedItems.Count = 0 Then
            MsgBox "Anulowano"
        Else
            MsgBox .SelectedItems(1)
        End If
    End With
End Sub
```

Obiekt FileDialog umożliwia określenie katalogu startowego poprzez ustawienie odpowiedniej wartości właściwości InitialFileName. W naszym przypadku kod procedury jako katalogu startowego używa domyślnej ścieżki programu Excel.

Wyświetlanie wbudowanych okien dialogowych programu Excel

Jedną z zalet języka VBA jest to, że można go używać jako narzędzia pozwalającego na naśladowanie wbudowanych poleceń Excela. Przyjrzyj się wierszowi kodu VBA przedstawionemu poniżej.

```
Range("A1:A12").Name = "MonthNames"
```

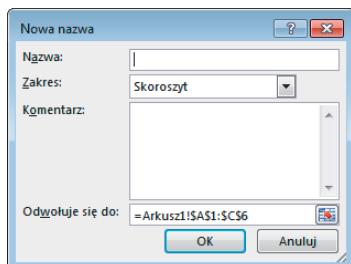
Wykonanie takiego polecenia ma dokładnie taki sam efekt jak wykonanie sekwencji poleceń: przejdź na kartę *FORMUŁY*, w grupie *Nazwy zdefiniowane* wybierz polecenie *Definiuj nazwę*, następnie w oknie dialogowym *Nowa nazwa*, które pojawi się na ekranie, wpisz w polu *Nazwa* ciąg znaków MonthNames, wpisz w polu *Odwoluje się do* zakres A1:A12 i wreszcie naciśnij przycisk *OK*.

Kiedy wykonasz przedstawione wcześniej polecenie VBA, okno dialogowe *Nowa nazwa* nie zostanie — oczywiście — wyświetcone na ekranie. Prawie zawsze będziesz chciał, aby odbywało się to w taki właśnie sposób; migające na ekranie podczas wykonywania makra okna dialogowe zazwyczaj nie są elementem pożądanym.

W niektórych przypadkach jednak będziesz chciał, aby kod programu VBA wyświetlił na ekranie jedno z wielu wbudowanych okien dialogowych Excela i pozwolił użytkownikowi na wykonanie w nim odpowiedniego wyboru. Możesz to uzyskać za pomocą wywołania z poziomu VBA odpowiedniego polecenia *Wstążki*. Poniżej przedstawiam przykład polecenia VBA, które powoduje wyświetlenie na ekranie okna dialogowego *Nowa nazwa*. Adres widoczny w polu *Odwoluje się do* odpowiada zakresowi komórek, które były zaznaczone podczas wywołania tego polecenia (zobacz rysunek 15.9).

```
Application.CommandBars.ExecuteMso "NameDefine"
```

Rysunek 15.9.
Wyświetlanie
za pomocą VBA
jednego z wielu
wbudowanych
okien dialogo-
wych Excela



Twój kod VBA nie jest w stanie pobierać żadnych informacji z tak wywołanego okna dialogowego. Jeśli na przykład wykonasz polecenie przywołujące na ekran okno dialogowe *Nowa nazwa*, Twój program nie będzie w stanie pobrać ani nazwy, która zostanie wpisana przez użytkownika, ani zakresu komórek, do których taka nazwa będzie się odwoływać.

Metoda *ExecuteMso* jest metodą obiektu *CommandBars* i pobiera tylko jeden argument, *idMso*, który reprezentuje wybrany formant *Wstążki*. Niestety, w pomocy systemowej programu Excel nie znajdziesz listy dozwolonych wartości tego argumentu. Co więcej, ponieważ *Wstążka* jest względnie nowym elementem interfejsu, to nie trzeba chyba dodawać, że kod, który wykorzystuje metodę *ExecuteMso*, nie jest kompatybilny z wersjami Excela starszymi niż 2007.

Poniżej zamieszczam kolejny przykład zastosowania metody *ExecuteMso*. Po wykonaniu takiego polecenia na ekranie pojawi się karta *Czcionka* okna dialogowego *Formatowanie komórek*.

```
Application.CommandBars.ExecuteMso "FormatCellsFontDialog"
```

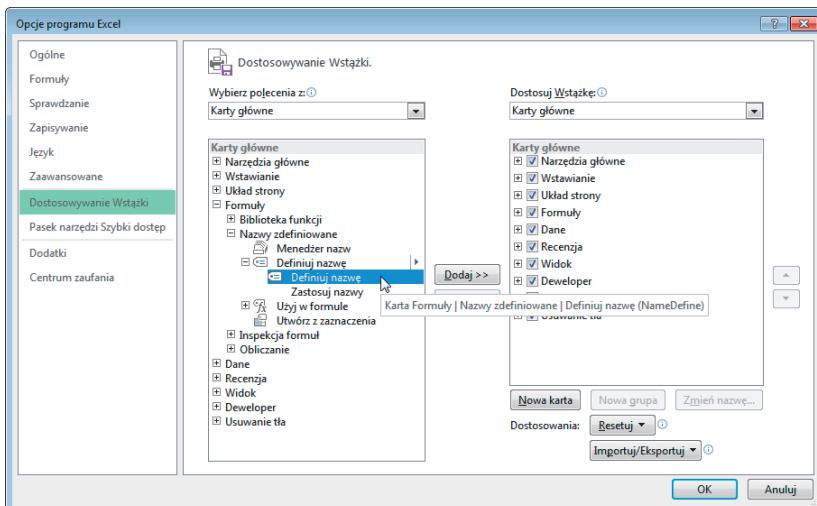
256 Część IV: Komunikacja z użytkownikiem

Jeżeli spróbowujesz wywołać wbudowane okno dialogowe w niepoprawnym kontekście, Excel wyświetli na ekranie odpowiedni komunikat o wystąpieniu błędu. I tak polecenie przedstawione poniżej przywołuje na ekran okno dialogowe *Formatowanie komórek/Liczby*.

```
Application.CommandBars.ExecuteMso ("NumberFormatsDialog")
```

Jeżeli spróbowujesz wykonać polecenie przedstawione powyżej w nieodpowiednim momencie (na przykład, kiedy aktualnie zaznaczonym elementem nie jest komórka, ale kształt), Excel wyświetli na ekranie komunikat o wystąpieniu błędu. Dzieje się tak, ponieważ to okno dialogowe działa poprawnie tylko i wyłącznie w kontekście komórek arkusza.

Excel zawiera tysiące różnych poleceń. W jaki zatem sposób możesz odnaleźć nazwę polecenia, które jest Ci potrzebne? Jednym ze sposobów jest użycie karty *Dostosowywanie Wstążki* okna dialogowego *Opcje programu Excel* (aby je przywołać na ekran, kliknij dowolne miejsce Wstążki prawym przyciskiem myszy i z menu podręcznego, które pojawi się na ekranie, wybierz polecenie *Dostosuj Wstążkę*). Praktycznie każde polecenie dostępne w Excelu jest wymienione na liście znajdującej się w lewym panelu okna. Aby znaleźć nazwę wybranego polecenia, odszukaj je na liście i ustawi nad nim wskaźnik myszy — po chwili pojawi się podpowiedź ekranowa zawierająca nazwę polecenia. Przykład został przedstawiony na rysunku 15.10.



Rysunek 15.10.
Zastosowanie
karty Dostoso-
wywanie
Wstążki do
identyfikacji na-
zwę polecenia

Rozdział 16

Wprowadzenie do formularzy UserForm

W tym rozdziale:

- ▶ dowiesz się, kiedy używać formularzy *UserForm*,
- ▶ odkryjesz tajemnice obiektów *UserForm*,
- ▶ nauczysz się wyświetlać formularze *UserForm* na ekranie,
- ▶ zobaczysz, jak utworzyć formularz *UserForm*, który współpracuje z makrami VBA.

Formularze *UserForm* są bardzo użytecznym rozwiązaniem zwłaszcza wtedy, kiedy makro VBA musi zatrzymać działanie i pobrać jakieś dane od użytkownika. Dobrym przykładem jest sytuacja, kiedy sposób działania makra zależy od szeregu opcji, które zaznacza bądź wybiera użytkownik na formularzu *UserForm*. Jeżeli program wymaga podania tylko kilku informacji (na przykład odpowiedzi *Tak/Nie* czy wpisania prostego ciągu znaków), użycie jednej z technik opisywanych wcześniej w rozdziale 15 może być w zupełności zadowalającym rozwiązaniem. Jeżeli jednak chcesz pobrać od użytkownika więcej informacji, musisz utworzyć formularz *UserForm*. W tym rozdziale poznasz podstawowe zagadnienia związane z tworzeniem takich formularzy i z pewnością będziesz zadowolony z zawarcia takiej znajomości.

Kiedy używać formularzy UserForm?

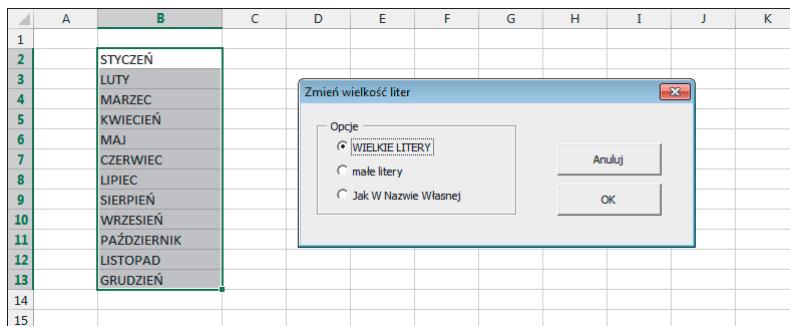
W tym podrozdziale opisuję sytuację, w której zastosowanie formularza *UserForm* może być dobrym rozwiązaniem. Makro przedstawione poniżej zamienia tekst w zaznaczonym obszarze komórek na wielkie litery. Do wykonania takiej operacji wykorzystuję wbudowaną funkcję *UCase* języka VBA.

```
Sub ChangeCase()
    Dim WorkRange As Range
    ' Zakończ pracę, jeżeli zaznaczony element nie jest zakresem komórek
    If TypeName(Selection) <> "Range" Then Exit Sub
    ' Przetwarzaj tylko tekst wpisany w komórkach, a nie formuły
    On Error Resume Next
    Set WorkRange = Selection.SpecialCells
        (x1CellTypeConstants, x1CellTypeConstants)
    For Each cell In WorkRange
```

```
    cell.Value = UCASE(cell.Value)  
    Next cell  
End Sub
```

Oczywiście, możesz łatwo zmodyfikować to makro tak, aby było jeszcze bardziej użyteczne. Przykładowo byłoby dobrze, gdyby nasze makro mogło również zmieniać tekst w komórkach na małe litery lub rozpoczynać każdy wyraz od wielkiej litery. Jednym z możliwych rozwiązań jest utworzenie dwóch dodatkowych makr — jednego do zmiany liter na małe i drugiego do zmiany tekstu tak, aby każdy wyraz rozpoczynał się od wielkiej litery. Innym rozwiązaniem jest zmodyfikowanie naszego makra bazowego tak, aby realizowało również wspomniane dwie dodatkowe opcje. Jeżeli zdecydujesz się na to drugie podejście, będziesz potrzebował jakiegoś sposobu pobrania od użytkownika informacji o tym, jakiego typu zmianę zawartości komórek nasze makro powinno wykonać.

Dobrym rozwiązańem takiego zagadnienia będzie wyświetlenie na ekranie prostego okna dialogowego, takiego jak przedstawione na rysunku 16.1. Takie okno dialogowe to, inaczej mówiąc, formularz *UserForm*, który został utworzony w edytorze VBE i następnie wyświetlony na ekranie za pomocą makra VBA. W kolejnym podrozdziale przedstawię szczegółowo, krok po kroku, procedurę tworzenia takiego okna dialogowego. Zanim jednak do tego przejdziemy, chciałbym tylko jeszcze zamieścić jedną uwagę.



Rysunek 16.1.
Informacje od użytkownika możesz pobierać za pomocą formularza *UserForm*



W oficjalnej terminologii VBA okna dialogowe noszą nazwę formularzy *UserForm*, ale formularz *UserForm* to obiekt, którego jedną z cech jest coś, co użytkownicy znają jako *okno dialogowe*. Na szczęście, takie szczegółowe rozróżnienie nie jest istotne, stąd w tej książce terminów *formularz UserForm* i *okno dialogowe* będę używać zamiennie.

Tworzenie formularzy *UserForm* — wprowadzenie

Kiedy chcesz utworzyć formularz *UserForm*, zazwyczaj będziesz musiał wykonać następujące operacje.

1. **Zastanów się, w jaki sposób okno dialogowe będzie wykorzystywane i kiedy makro VBA będzie wyświetlało okno na ekranie.**

2. Naciśnij kombinację klawiszy **Alt + F11**, aby uruchomić edytor VBE, a następnie wstaw do projektu VBA nowy obiekt **UserForm**.

Każdy obiekt *UserForm* może przechowywać pojedynczy formularz *UserForm*.

3. Umieść na formularzu **UserForm** żądane formanty.

Na formularzach *UserForm* możesz umieszczać takie formanty jak pola tekstowe, przyciski, pola wyboru, listy i inne.

4. Użyj okna **Properties** do zmiany właściwości formantów oraz samego formularza *UserForm*.

5. Napisz procedury obsługi zdarzeń dla poszczególnych formantów (na przykład makro, które będzie wykonywane, kiedy użytkownik naciśnie przycisk w oknie dialogowym).

Procedury obsługi zdarzeń są przechowywane w oknie *Code* obiektu *UserForm*.

6. Napisz procedurę (w module kodu VBA), która w odpowiednim momencie będzie wyświetlała okno dialogowe na ekranie.

Nie przejmuj się, jeżeli niektóre z opisanych wyżej kroków wydają się zupełnie niezrozumiałe. W kolejnych podrozdziałach podam bardziej szczegółowe opisy poszczególnych etapów tworzenia formularzy *UserForm*.

Kiedy projektujesz formularz *UserForm*, tworzysz dla aplikacji coś, co programiści nazywają graficznym interfejsem użytkownika (*GUI — Graphical User Interface*). Co prawda, dla akronimu *GUI* można znaleźć co najmniej jeszcze kilka innych rozwinięć, to jednak żadne inne nie będzie tutaj odpowiednie.

Teraz powinieneś poświęcić trochę czasu na zastanowienie się nad tym, jak projektowany formularz powiniem wyglądać i w jaki sposób użytkownicy będą korzystać z formantów umieszczonych w oknie dialogowym. Powinieneś spróbować przeprowadzić użytkownika przez kolejne etapy pracy z formularzem. Możesz to wykonać poprzez przemyślane zaprojektowanie układu formantów w oknie oraz zastosowanie jednoznacznych etykiet poszczególnych opcji. To tak, jak z większością zagadnień związanych z VBA — im więcej będziesz spędzał czasu na projektowaniu formularzy i tworzeniu kodu, tym łatwiej będzie Ci zbudować kolejną aplikację.

Praca z formularzami UserForm

Każde okno dialogowe, które utworzysz, będzie przechowywane w osobnym obiekcie *UserForm* — inaczej mówiąc, każdy obiekt *UserForm* może przechowywać tylko jedno okno dialogowe. Tworzenie i programowanie formularzy *UserForm* odbywa się przy użyciu edytora VBE.

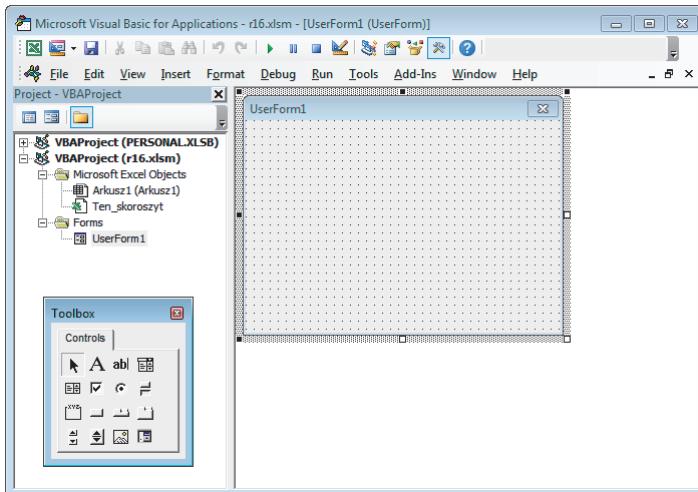
Wstawianie nowego formularza UserForm

Aby wstawić nowy formularz *UserForm* do projektu VBA, powinieneś wykonać następujące polecenia.

1. Uaktywnij edytor VBE, naciskając kombinację klawiszy **Alt+F11**.
2. W oknie **Project** zaznacz skoroszyt, z którym pracujesz.
3. Z menu głównego wybierz polecenie **Insert/UserForm**.

Edytor VBE wstawi do projektu VBA nowy obiekt *UserForm*, który będzie zawierał puste okno dialogowe o domyślnych rozmiarach.

Na rysunku 16.2 przedstawiam taki formularz *UserForm* — puste okno dialogowe. Twoim zadaniem będzie zapelnienie go odpowiednimi formantami.



Rysunek 16.2.
Nowy formularz
UserForm

Umieszczanie formantów na formularzu *UserForm*

Kiedy aktywujesz formularz *UserForm*, edytor VBA wyświetli na ekranie „pływające” okno *Toolbox*, co zostało przedstawione na rysunku 16.2. Za pomocą narzędzi umieszczonej w tym oknie możesz dodawać nowe formanty do formularza *UserForm*. Jeżeli z jakiegoś powodu okno *Toolbox* nie jest widoczne na ekranie po aktywowaniu formularza *UserForm*, możesz je przywołać, wybierając z menu głównego polecenie *View/Toolbox*.

Aby dodać nowy formant do formularza, po prostu kliknij wybrany formant w oknie *Toolbox* lewym przyciskiem myszy i przeciągnij go na okno formularza. Formant po umieszczeniu w oknie możesz dowolnie przesuwać i zmieniać jego rozmiary przy użyciu standardowych technik „myszowych”.

W tabeli 16.1 zamieszczam zestawienie formantów wraz z krótkim opisem ich możliwości i przeznaczenia. Aby przekonać się, jakie formanty są reprezentowane przez poszczególne ikony w oknie *Toolbox*, ustaw wskaźnik myszy nad wybraną ikoną i przytrzymaj przez chwilę nieruchomo — przy wskaźniku myszy pojawi się podpowiedź ekranowa z nazwą formantu.

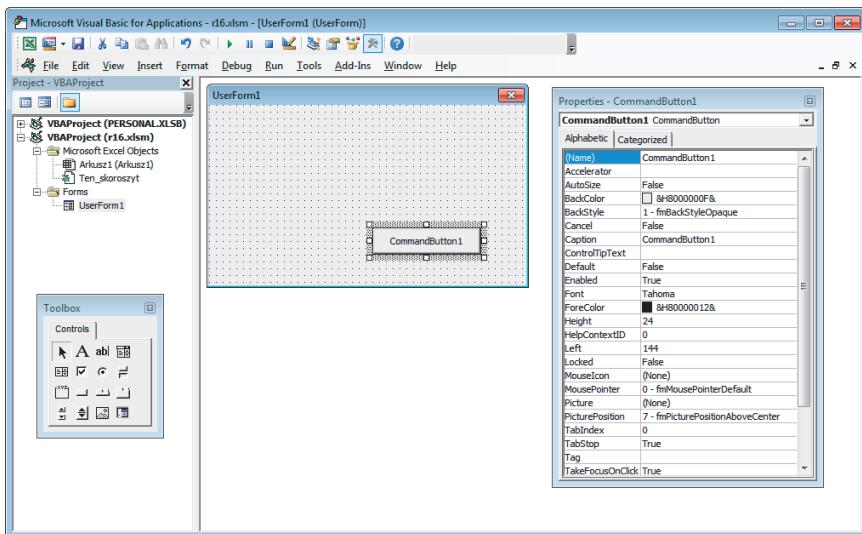
Tabela 16.1. Formanty dostępne w oknie Toolbox

Nazwa formantu	Opis
Label	<i>Etykieta</i> ; wyświetla tekst.
TextBox	<i>Pole tekstowe</i> ; pozwala użytkownikowi na wprowadzanie tekstu.
ComboBox	<i>Pole kombi</i> ; wyświetla połączenie pola tekstowego z listą rozwijaną.
ListBox	<i>Pole listy</i> ; wyświetla listę elementów.
CheckBox	<i>Pole wyboru</i> ; formant używany w przypadku opcji dwustanowych typu <i>Tak/Nie</i> , <i>Prawda/Fałsz</i> , <i>Włączony/Wyłączony</i> i tak dalej.
OptionButton	<i>Pole opcji</i> ; przydaje się, gdy użytkownik musi wybrać jedną opcję spośród kilku dostępnych. Formant ten zawsze ma postać grupy złożonej z przynajmniej dwóch opcji.
ToggleButton	<i>Przycisk przełącznika</i> ; posiada dwa stany — włączony lub wyłączony.
Frame	<i>Pole grupy</i> ; formant spełniający rolę kontenera dla innych formantów.
CommandButton	<i>Przycisk polecenia</i> ; przycisk, który użytkownik może nacisnąć.
TabStrip	<i>Pole karty</i> ; pozwala na wyświetlanie kart zawierających formanty.
MultiPage	<i>Pole strony</i> ; kontener dla innych formantów, umożliwia tworzenie okien dialogowych wyposażonych w karty.
ScrollBar	<i>Pasek przewijania</i> ; pozwala na zmianę wartości za pomocą suwaka.
SpinButton	<i>Pokrętło</i> ; pozwala na zmianę wartości za pomocą przycisków ze strzałkami.
Image	<i>Pole obrazu</i> ; przechowuje elementy graficzne.
RefEdit	<i>Pole zakresu</i> ; pozwala na wprowadzenie adresu komórki bądź wskazanie przy użyciu myszy zakresu komórek arkusza.

Modyfikacja właściwości formantów formularza UserForm

Każdy formant, który zostanie umieszczony na formularzu *UserForm* posiada szereg właściwości, mających wpływ na jego wygląd i zachowanie. Oprócz tego, sam formularz *UserForm* również posiada swój własny zestaw właściwości. Zmiany właściwości można dokonać za pomocą okna o nazwie, a jakże, *Properties* (Właściwości). Na rysunku 16.3 przedstawiam wygląd okna *Properties*, kiedy zaznaczony został formant *CommandButton*.

Okno *Properties* pojawi się na ekranie, kiedy naciśniesz klawisz *F4*. Lista właściwości widocznych w oknie jest uzależniona od tego, jaki formant był zaznaczony w chwili przywołania okna. Jeżeli teraz zaznaczyłeś inny formant, lista dostępnych właściwości ulegnie zmianie, automatycznie dostosowując się do aktualnie zaznaczonego formantu. Aby ukryć okno *Properties*, kiedy nie będzie Ci już potrzebne, powinieneś kliknąć przycisk *Zamknij*, znajdujący się po prawej stronie paska tytułowego tego okna. Pamiętaj, że kolejne naciśnięcie klawisza *F4* spowoduje ponowne pojawienie okna na ekranie.



Rysunek 16.3.
Okna Properties możesz używać do zmiany właściwości formantów formularza UserForm

Lista dostępnych właściwości formantów obejmuje między innymi takie elementy.

- ✓ Name (identyfikator formantu, używany w kodzie programu).
- ✓ Width (szerokość formantu).
- ✓ Height (wysokość formantu).
- ✓ Value (wartość formantu).
- ✓ Caption (etykieta formantu wyświetlana na ekranie).

Każdy formant posiada swój własny zestaw właściwości (aczkolwiek wiele formantów ma zbliżone lub nawet takie same właściwości). Aby zmienić wybraną właściwość przy użyciu okna *Properties*, powinieneś wykonać operacje przedstawione poniżej.

1. Upewnij się, że na formularzu *UserForm* zaznaczyłeś odpowiedni formant.
2. Sprawdź, czy okno *Properties* jest widoczne; jeżeli nie, przywołaj je na ekran, naciskając klawisz *F4*.
3. Przejdź do okna *Properties* i kliknij właściwość, którą chcesz zmodyfikować.
4. Wykonaj pożądane zmiany ustawień właściwości w prawej części okna *Properties*.

Kiedy zaznaczyłeś sam formularz *UserForm* (a nie jeden z formantów formularza), będziesz mógł użyć okna *Properties* do zmiany właściwości formularza.

W rozdziale 17. znajdziesz wszystko, co powinieneś wiedzieć o pracy z formantami okien dialogowych.



Niektóre właściwości formularzy *UserForm* są wykorzystywane jako domyślne właściwości nowych formantów dodawanych do formularza. Jeśli na przykład zmienisz właściwość *Font* formularza *UserForm*, wszystkie formanty dodawane do tego formularza będą domyślnie korzystały z tej samej czcionki. Pamiętaj, że zmiana właściwości formularza nie będzie miała żadnego wpływu na formanty, które zostały dodane już wcześniej.

Przeglądanie okna Code formularza UserForm

Każdy obiekt *UserForm* posiada swój własny moduł *Code*, który przechowuje kod VBA procedur obsługujących zdarzeń wykonywanych, kiedy użytkownik pracuje z formantami formularza. Aby wyświetlić moduł *Code*, powinieneś nacisnąć klawisz *F7*. Okno *Code* jest początkowo puste i dopiero samodzielnie musisz je wypełnić odpowiednimi procedurami. Aby powrócić do widoku okna dialogowego, powinieneś nacisnąć kombinację klawiszy *Shift+F7*.

Innym sposobem przełączania się pomiędzy oknem *Code* a edytorem okna dialogowego jest wykorzystanie przycisków *View Code* oraz *View Object*, znajdujących się na pasku narzędzi okna *Project*. Zamiast korzystać z przycisków, możesz również kliknąć formularz *UserForm* prawym przyciskiem myszy i z menu podręcznego wybrać opcję *View Code*. Gdy pracujesz z oknem *Code* i chcesz powrócić do widoku okna dialogowego, po prostu przejdź do okna *Project* i dwukrotnie kliknij nazwę formularza.

Wyświetlanie formularzy UserForm

Formularze *UserForm* można wyświetlić na ekranie z poziomu kodu VBA za pomocą metody *Show* obiektu *UserForm*.



Makro, które będzie wyświetlało okno dialogowe, musi się znajdować w module VBA, a nie w oknie *Code* formularza *UserForm*.

Procedura, której kod przedstawiam poniżej, wyświetla na ekranie okno dialogowe o nazwie *UserForm1*.

```
Sub ShowDialogBox()
    UserForm1.Show
    ' Tutaj możesz wstawić inne polecenia
End Sub
```

Kiedy Excel wyświetla okno dialogowe, makro *ShowDialogBox* wstrzymuje działanie, aż do momentu zamknięcia okna dialogowego przez użytkownika. Następnie procedura VBA wznowia działanie i wykonuje pozostałe polecenia procedury. W większości przypadków w takiej procedurze nie będzie żadnego dodatkowego kodu. Jak sam niedługo zobaczysz, procedury obsługi zdarzeń powinny zostać umieszczone w oknie *Code* formularza *UserForm*. Takie procedury zostają automatycznie wywoływane podczas pracy użytkownika z poszczególnymi formantami formularza *UserForm*.

Pobieranie i wykorzystywanie informacji z formularzy UserForm

Edytor VBE nadaje unikatową nazwę każdemu formantowi, który dodajesz do formularza. Nazwa formantu odpowiada wartości jego właściwości Name. Nazwy formantu będziesz używać do odwoływania się do danego formantu w kodzie VBA. Jeżeli na przykład umieścisz pole wyboru (formant CheckBox) na formularzu *UserForm* o nazwie UserForm1, to taki formant domyślnie otrzyma nazwę CheckBox1. Jeżeli chcesz, aby pole wyboru było domyślnie zaznaczone, możesz to ustawić za pomocą odpowiedniej właściwości w oknie *Properties*. Zamiast tego możesz — oczywiście — wykonać to z poziomu kodu VBA.

```
UserForm1.CheckBox1.Value = True
```

W większości przypadków makra VBA dla danego formularza *UserForm* będziesz pisał w module kodu tego formularza. W takiej sytuacji możesz pominąć w odwołaniu kwalifikator obiektu *UserForm* i zapisać polecenie w takiej postaci:

```
CheckBox1.Value = True
```

Kod VBA Twojego makra może — oczywiście — sprawdzać różne właściwości formantów i na podstawie ich wartości podejmować rozmaite akcje. Polecenie przedstawione poniżej wywołuje makro o nazwie PrintReport tylko wtedy, kiedy pole wyboru CheckBox1 jest zaznaczone.

```
If CheckBox1.Value = True Then Call PrintReport
```

Więcej szczegółowych informacji na ten temat znajdziesz w rozdziale 17.



Zazwyczaj dobrym rozwiązaniem będzie zmiana domyślnych nazw formantów, nadanych przez edytor VBE, na nazwy, które będą bardziej powiązane z funkcjami poszczególnych formantów. Przykładowo domyślną nazwę pola wyboru, o którym mówilem przed chwilą, możesz zmienić na cbxPrintReport. Zwrć uwagę, że nazwę pola wyboru poprzedziłem trzyliterowym prefiksem cbx, wskazującym typ formantu (CheckBox). Ostateczna decyzja, o tym czy przyjąć taką, czy inną konwencję nazywania formantów, należy tylko i wyłącznie do Ciebie.

Przykład tworzenia formularza UserForm

Przykładowy formularz *UserForm* omawiany w tym podrozdziale jest rozszerzoną wersją makra ChangeCase, o którym wspomniałem na początku tego rozdziału. Jak zapewne pamiętasz, oryginalna wersja tego makra zmieniała tekst w zaznaczonym obszarze komórek na wielkie litery. W nowej, zmodyfikowanej wersji makra użytkownik za pomocą formularza *UserForm* będzie mógł dokonać wyboru, czy zmienia tekst na wielkie litery, na małe litery, czy też może zmienia pisownię liter, tak jak w nazwie własnej.

Jak widać, nasze okno dialogowe musi pobrać od użytkownika jedną informację, reprezentującą rodzaj operacji, jaka powinna zostać wykonana na tekście. Ponieważ użytkownik ma do wyboru trzy możliwości, najlepszym rozwiązaniem będzie utworzenie okna dialogowego, zawierającego trzy przyciski opcji (formanty *OptionButton*). W oknie dialogowym będą się również musiały znaleźć się dwa dodatkowe przyciski — przycisk *OK* oraz przycisk *Anuluj*. Naciśnięcie przycisku *OK* będzie powodowało zatwierdzenie wyboru dokonanego przez użytkownika i uruchomienie kodu zmieniającego wielkość liter w zaznaczonych komórkach. Z kolei naciśnięcie przycisku *Anuluj* będzie powodowało przerwanie operacji i zakończenie działania makra bez dokonywania żadnych zmian.



Skoroszyt zawierający omawiany przykład znajdziesz na stronie internetowej tej książki. Z drugiej jednak strony, to ćwiczenie przyniesie znacznie więcej korzyści, jeżeli wykonasz wszystko samodzielnie, krok po kroku, zgodnie z instrukcjami zamieszczonymi w tym podrozdziale.

Tworzenie formularza *UserForm*

Aby utworzyć formularz *UserForm*, powinieneś wykonać polecenia przedstawione poniżej. Rozpocznij od utworzenia nowego, pustego skoroszytu.

1. **Naciśnij kombinację klawiszy *Alt+F11*, aby uruchomić edytor VBE.**
2. **Jeżeli w oknie *Project* znajduje się kilka projektów, zaznacz projekt odpowiadający skoroszytowi, którego aktualnie używasz.**
3. **Z menu głównego edytora VBE wybierz polecenie *Insert/UserForm*.**
Edytor VBE wstawi nowy obiekt *UserForm*, zawierający puste okno dialogowe.
4. **Naciśnij klawisz *F4*, aby przywołać na ekran okno *Properties*.**
5. **W oknie *Properties* zmień wartość właściwości *Caption* na ciąg znaków *Zmień wielkość liter*.**
6. **Jeżeli utworzone okno dialogowe jest zbyt duże lub zbyt małe, możesz je kliknąć i zmienić jego rozmiar, korzystając z uchwytów, które pojawią się w rogach i na krawędziach okna.**

Operację zmiany rozmiaru formularza możesz wykonać w dowolnym momencie, na przykład po umieszczeniu wszystkich formantów w oknie dialogowym.

Dodawanie przycisków poleceń (formanty *CommandButton*)

Jesteś już gotowy, aby utworzyć na formularzu przyciski *OK* i *Anuluj*? Jeżeli tak, uważnie wykonaj polecenia przedstawione poniżej.

1. **Upewnij się, że okno *Toolbox* jest widoczne na ekranie; jeżeli nie, z menu głównego edytora VBE wybierz polecenie *View/Toolbox*.**

2. Jeżeli okno *Properties* nie jest widoczne na ekranie, naciśnij przycisk *F4*, aby je wyświetlić.
3. W oknie *Toolbox* odszukaj formant *CommandButton* i przeciągnij go na formularz za pomocą myszy.

Jak możesz zauważać w oknie *Properties*, nowo utworzony przycisk otrzymał domyślną nazwę i etykietę *CommandButton1*.

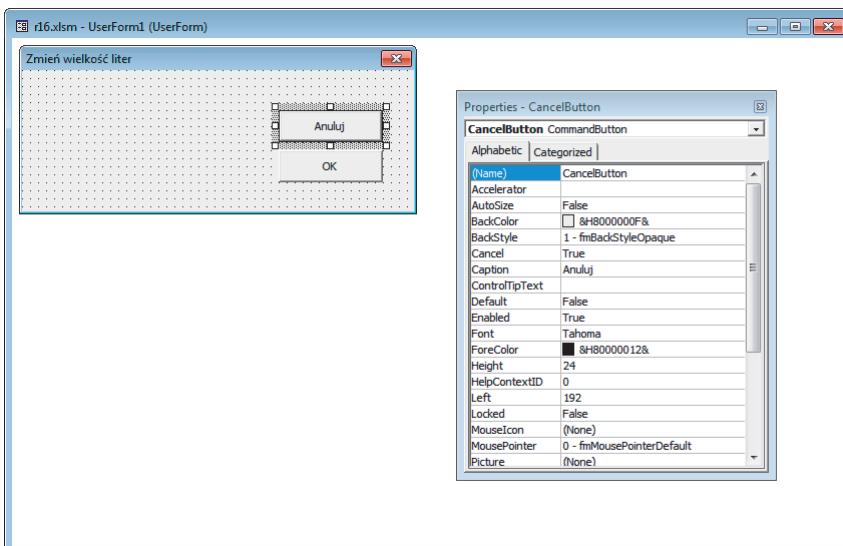
4. Upewnij się, że nowo utworzony przycisk jest zaznaczony, a następnie przejdź do okna *Properties* i zmień następujące właściwości.

<i>Właściwość</i>	<i>Zmień na:</i>
Name	OKButton
Caption	OK
Default	True

Dodaj drugi przycisk *CommandButton* do formularza *UserForm* i zmień jego następujące właściwości.

<i>Właściwość</i>	<i>Zmień na:</i>
Name	CancelButton
Caption	Anuluj
Cancel	True

5. Dostosuj rozmiar i położenie przycisków, aby Twoje okno dialogowe wyglądało mniej więcej tak, jak to zostało przedstawione na rysunku 16.4.



Rysunek 16.4.
Formularz
UserForm
z dwoma przyciskami poleceń

Dodawanie przycisków opcji (formanty OptionButton)

W tym punkcie dołączysz do formularza *UserForm* trzy przyciski opcji (formanty *OptionButton*). Zanim jednak to zrobisz, musisz umieścić na formularzu formant *Frame*, który będzie spełniał rolę kontenera dla przycisków opcji. Oczywiście, zastosowanie formantu *Frame* nie jest konieczne, ale jego użycie spowoduje, że całe okno dialogowe będzie wyglądało bardziej profesjonalnie.

1. Przejdź do okna *Toolbox* i odszukaj formant *Frame*, a następnie przeciągnij go na formularz *UserForm*.

Wykonanie takiej operacji spowoduje utworzenie kontenera (ramki), w którym umieszczone będą przyciski opcji.

2. Przejdź do okna *Properties* i zmień właściwość *Caption ramki na Opcje*.
3. Ponownie przejdź do okna *Toolbox*, odszukaj formant *OptionButton* i przeciągnij go na formularz *UserForm* tak, aby znalazł się w obrębie utworzonej wcześniej ramki.
4. Zaznacz nowo utworzony przycisk opcji, a następnie przejdź do okna *Properties* i zmień następujące właściwości formantu.

<i>Właściwość</i>	<i>Zmień na:</i>
Name	OptionUpper
Caption	WIELKIE LITERY
Accelerator	W
Value	True

Ustawienie właściwości *Value* na wartość *True* powoduje, że ten przycisk opcji będzie zaznaczony domyślnie.

5. Dodaj kolejny przycisk opcji, a następnie przejdź do okna *Properties* i zmień następujące właściwości tego formantu.

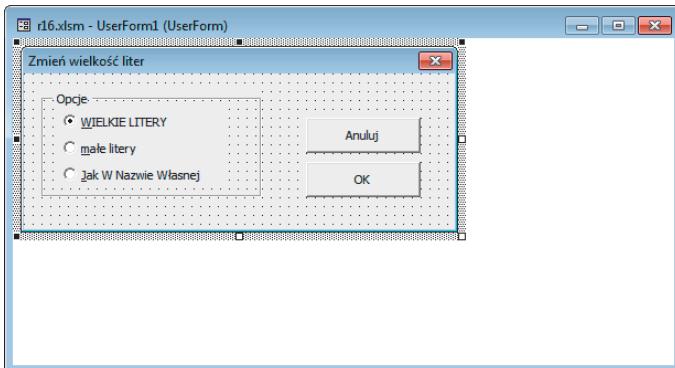
<i>Właściwość</i>	<i>Zmień na:</i>
Name	OptionLower
Caption	małe litery
Accelerator	M

6. Dodaj trzeci przycisk opcji, a następnie przejdź do okna *Properties* i zmień następujące właściwości tego formantu.

<i>Właściwość</i>	<i>Zmień na:</i>
Name	OptionProper
Caption	Jak W Nazwie Własnej
Accelerator	J

7. Dostosuj rozmiar i położenie przycisków opcji, kontenera Frame i całego okna dialogowego.

Twój formularz *UserForm* powinien wyglądać mniej więcej tak, jak na rysunku 16.5.



Rysunek 16.5.
Wygląd formularza UserForm po dodaniu trzech przycisków opcji wewnątrz kontenera Frame

Jeżeli chciałbyś już zobaczyć, jak będzie wyglądał Twój formularz *UserForm* po uruchomieniu, naciśnij klawisz *F5*. Oczywiście, żaden z formantów jeszcze nie działa, zatem aby zamknąć formularz i powrócić do edytora, musisz kliknąć czerwony przycisk *Zamknij*, znajdujący się w prawym, górnym rogu okna dialogowego.

Właściwość *Accelerator* pozwala na wskazanie, która litera w etykiecie przycisku będzie podkreślona. Co ważniejsze, właściwość ta określa jednocześnie kombinację klawiszy *lewy Alt+<klawisz>*, która będzie uaktywniała dany formant. Przykładowo opcję *male litery* możesz zaznaczyć, naciskając kombinację klawiszy *lewy Alt+M*, ponieważ to właśnie litera *M* jest podkreślona. Definiowanie klawiszy akceleratorów jest opcjonalne, aczkolwiek pamiętaj, że wielu użytkowników przy pracy z oknami dialogowymi lubi korzystać ze skrótów klawiszowych.

Możesz się teraz zastanawiać, dlaczego zdefiniowaliśmy skróty klawiszowe dla przycisków opcji, ale dla przycisków poleceń (formanty *CommandButton*) już nie. Odpowiedź jest prosta — ogólnie rzeczą biorąc, dla przycisków *OK* i *Anuluj* nigdy nie definiujemy akceleratorów, ponieważ przyciski te można domyślnie obsługiwać z klawiatury. Naciśnięcie klawisza *Enter* jest odpowiednikiem kliknięcia przycisku *OK*, ponieważ właściwość *Default* tego przycisku została ustaliona na wartość *True*. Z kolei ekuivalentem kliknięcia przycisku *Anuluj* jest naciśnięcie klawisza *Esc*. Dzieje się tak, ponieważ właściwość *Cancel* tego formantu została ustaliona na wartość *True*.

Dodawanie procedur obsługi zdarzeń

Teraz nadszedł czas na to, aby wreszcie nasz formularz *UserForm* zrobił coś użytecznego. Poniżej znajdziesz szczegółowy opis sposobu dodawania procedur obsługi zdarzeń dla przycisków *OK* i *Anuluj*.

1. Dwukrotnie kliknij lewym przyciskiem myszy przycisk *Anuluj*.

Edytor VBE aktywuje okno *Code* formularza *UserForm* i wstawi pustą procedurę.

```
Private Sub CancelButton_Click()
End Sub
```

Procedura o nazwie `CancelButton_Click` jest wykonywana, kiedy przycisk *Anuluj* zostanie kliknięty lewym przyciskiem myszy, ale — oczywiście — tylko wtedy, kiedy okno dialogowe formularza będzie uruchomione i wyświetcone na ekranie. Innymi słowy, klikanie przycisku *Anuluj*, kiedy znajdujesz się w trybie edytowania formularza, nie spowoduje uruchomienia procedury obsługi tego przycisku. Ponieważ właściwość `Cancel` przycisku *Anuluj* jest ustawiona na wartość `True`, naciśnięcie klawisza *Esc*, kiedy formularz jest widoczny na ekranie, również spowoduje wywołanie procedury `CancelButton_Click`.

2. **Wstaw do nowo utworzonej procedury następujący wiersz kodu (powinieneś umieścić go przed wierszem zawierającym polecenie `End Sub`).**

```
Unload UserForm1
```

Takie polecenie powoduje, że kiedy użytkownik kliknie przycisk *Anuluj*, formularz *UserForm* zostanie zamknięty i usunięty z pamięci komputera).

3. **Naciśnij kombinację klawiszy *Shift+F7*, aby powrócić do widoku formularza.**
4. **Dwukrotnie kliknij lewym przyciskiem myszy przycisk *OK*.**

Edytor VBE aktywuje okno *Code* formularza *UserForm* i wstawi pustą procedurę o nazwie `OKButton_Click`.

Procedura zostanie wykonana, kiedy po wyświetleniu formularza *UserForm* na ekranie użytkownik kliknie przycisk *OK*. Ponieważ właściwość `Default` tego przycisku została ustawiona na wartość `True`, naciśnięcie klawisza *Enter* również spowoduje wykonanie procedury `OKButton_Click`.

5. **Wpisz kod makra VBA przedstawiony poniżej, tak aby nasza procedura wyglądała następująco:**

```
Private Sub OKButton_Click()
    Dim WorkRange As Range
    Dim cell As Range

    ' Przetwarza tylko tekst w komórkach, a nie formuły
    On Error Resume Next
    Set WorkRange = Selection.SpecialCells _
        (x1CellTypeConstants, x1CellTypeConstants)

    ' Wielkie litery
    If OptionUpper Then
        For Each cell In WorkRange
            cell.Value = UCase(cell.Value)
        Next cell
    End If

    ' Małe litery
    If OptionLower Then
        For Each cell In WorkRange
            cell.Value = LCase(cell.Value)
        Next cell
    End If
End Sub
```

```
End If

' Liter jak w nazwach własnych
If OptionProper Then
    For Each cell In WorkRange
        cell.Value = Application. _
            WorksheetFunction.Proper(cell.Value)
    Next cell
End If
Unload UserForm1
End Sub
```

Kod przedstawiony powyżej jest rozszerzoną wersją oryginalnej procedury ChangeCase, którą prezentowałem na początku tego rozdziału. Makro składa się z trzech bloków kodu i wykorzystuje struktury instrukcji warunkowych If-Then, z których każda posiada swoją pętlę For Each-Next. W każdym przebiegu procedury wykonywany jest tylko jeden blok warunkowy, który został wybrany przez użytkownika. Ostatnie polecenie procedury zamknięcie okna dialogowego i powoduje usunięcie formularza z pamięci.

A teraz ciekawostka. Zwróć uwagę na to, że VBA posiada funkcje UCASE i LCASE, ale nie posiada funkcji pozwalających na wykonanie konwersji tekstu do pisowni, takiej jak w nazwach własnych (gdzie każdy wyraz rozpoczyna się od wielkiej litery). Z tego powodu w naszej procedurze musiałem użyć funkcji arkuszowej Proper (poprzedzonej kwalifikatorem Application.WorksheetFunction), która pozwala na wykonanie takiej konwersji.

Innym rozwiązaniem może być użycie funkcji StrConv języka VBA i podanie jako drugiego argumentu wywołania stałej vbProperCase (więcej szczegółowych informacji na ten temat znajdziesz w pomocy systemowej). Funkcja StrConv — niestety — nie jest dostępna we wszystkich wersjach Excela, stąd pisząc to makro, zdecydowałem się na użycie funkcji arkuszowej Z.WIELKIEJ.LITERY (dostępnej w VBA przy użyciu funkcji Proper). Warto zauważyć, że funkcja StrConv przy konwersji tekstu do pisowni, takiej jak w nazwach własnych, działa znacznie lepiej niż funkcja arkuszowa Z.WIELKIEJ.LITERY, która ma tendencję do zapisywania wielką literą znaków, jakie występują w tekście po znaku apostrofu (stąd na przykład angielski zwrot can't zostanie niepoprawnie zamieniony na Can'T). Funkcja StrConv nie robi takich psikusów.

Tworzenie makra, które wyświetla formularz na ekranie

Nasz projekt jest już niemal na ukończeniu. Ostatnią rzeczą, której jeszcze brakuje, jest wyświetlanie formularza. Wykonaj polecenia przedstawione poniżej, aby utworzyć procedurę, która będzie wyświetlała okno dialogowe na ekranie.

1. Przejdz do edytora VBE i z menu głównego wybierz polecenie Insert/Module.

Edytor VBE wstawi nowy moduł kodu (o nazwie *Module1*) do bieżącego projektu VBA.

2. W nowo utworzonym module kodu wpisz następującą procedurę.

```
Sub ChangeCase()
    If TypeName(Selection) = "Range" Then
        UserForm1.Show
    Else
        MsgBox "Zaznacz wybrany obszar komórek.", vbCritical
    End If
End Sub
```

Jak widać, kod procedury jest bardzo prosty. Najpierw sprawdzam, czy użytkownik zaznaczył zakres komórek. Jeżeli tak, na ekranie wyświetlany jest formularz *UserForm* (za pomocą metody *Show*). Następnie użytkownik może wejść w interakcję z oknem dialogowym i wykonywany jest kod przechowywany w oknie *Code* formularza *UserForm*. Jeżeli użytkownik nie zaznaczył zakresu komórek, zobaczy na ekranie okno komunikatu (*MsgBox*), zawierające tekst *Zaznacz wybrany obszar komórek*.

Udostępnianie makra użytkownikowi

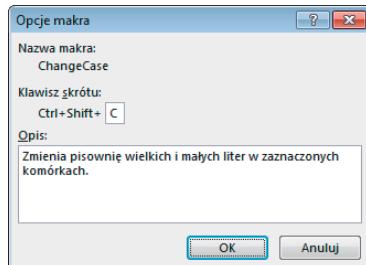
W tym momencie cały projekt powinien już działać poprawnie, jednak nadal trzeba znaleźć jakąś prostą metodę uruchamiania tego programu. Można to uzyskać, przypisując do makra *ChangeCase* odpowiedni skrót klawiszowy (na przykład kombinację klawiszy *Ctrl+Shift+C*). Aby to zrobić, wykonaj polecenia przedstawione poniżej.

1. Przejdź do głównego okna Excela, naciśkając kombinację klawiszy *Alt+F11*.
2. Przejdź na kartę *DEVELOPER* i wybierz polecenie *Kod/Makra*. Zamiast tego możesz po prostu nacisnąć kombinację klawiszy *Alt+F8*.
3. W oknie dialogowym *Makra*, które pojawi się na ekranie, zaznacz makro *ChangeCase*.
4. Naciśnij przycisk *Opcje*.

Na ekranie pojawi się okno dialogowe *Opcje makra*.

5. W polu *Klawisz skrótu* wpisz wielką literę *C*.

Całość powinna wyglądać tak, jak na rysunku 16.6.



Rysunek 16.6.
Przypisywanie
klawisza skrótu
do makra
ChangeCase

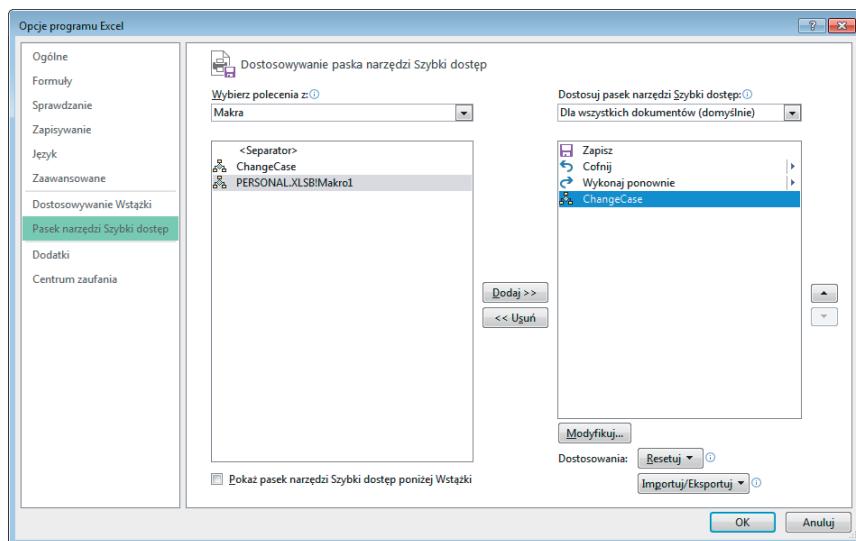
6. W polu *Opis* wprowadź krótką informację na temat przeznaczenia makra.

7. Naciśnij klawisz **OK**.

8. Kiedy powrócisz do okna dialogowego **Makro**, naciśnij przycisk **Anuluj**.

Od chwili zakończenia takiej operacji naciśnięcie kombinacji klawiszy **Ctrl+Shift+C** będzie powodowało uruchomienie makra **ChangeCase**, które w przypadku, kiedy wcześniej zaznaczony został zakres komórek, będzie wyświetlało na ekranie formularz *UserForm*.

Jeśli trzeba, możesz również udostępnić takie makro w postaci dedykowanego przycisku na pasku szybkiego dostępu w głównym oknie Excela. Aby to zrobić, kliknij prawym przyciskiem myszy pasek narzędzi *Szybki dostęp* i z menu podręcznego wybierz polecenie *Dostosuj pasek narzędzi Szybki dostęp*. Na ekranie pojawi się okno dialogowe *Opcje programu Excel*, gdzie w sekcji *Makra* znajdziesz makro **ChangeCase** (zobacz rysunek 16.7).



Rysunek 16.7.
Dodawanie makra *ChangeCase* do paska narzędzi *Szybki dostęp*

Testowanie działania makra

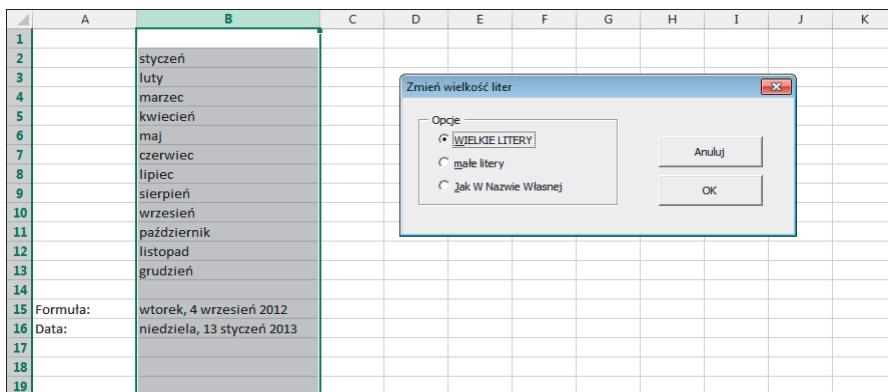
Wreszcie nadszedł czas, aby w praktyce przetestować działanie makra i okna dialogowego, a także upewnić się, że wszystko działa tak, jak tego oczekujemy. Aby to zrobić, wykonaj polecenia przedstawione poniżej.

1. Przejdź do wybranego arkusza (może to być dowolny arkusz dowolnego skoroszytu).
2. Zaznacz obszar komórek zawierających dane tekstowe.

Möżesz nawet zaznaczyć całe wiersze lub kolumny.

3. Naciśnij kombinację klawiszy **Ctrl+Shift+C**.

Na ekranie pojawi się okno formularza *UserForm*, które powinno wyglądać tak, jak to zostało przedstawione na rysunku 16.8.



Rysunek 16.8.
Działający
formularz
UserForm

4. Wybierz żądaną opcję i naciśnij przycisk OK.

Jeżeli wszystko działa, tak jak powinno, makro dokona odpowiednich zmian wielkości liter w tekście znajdującym się w zaznaczonych komórkach.

Jeżeli chcesz przetestować tę procedurę w sytuacji, kiedy przed wywołaniem makra zaznaczona jest tylko jedna komórka, przekonasz się, że zmiany zostaną dokonane we **wszystkich** komórkach arkusza. Takie zachowanie jest efektem ubocznym zastosowania metody `SpecialCells`. Jeżeli chcesz, aby makro mogło przetwarzarć również pojedyncze komórki, powinieneś zmodyfikować pierwszy blok kodu tak, jak to zostało przedstawione na listingu poniżej.

```
If Selection.Count = 1 Then
    Set WorkRange = Selection
Else
    Set WorkRange = Selection.SpecialCells
        (x1CellTypeConstants, x1CellTypeConstants)
End If
```

Na rysunku 16.9 przedstawiono wygląd arkusza po zamianie liter na wielkie. Zwróć uwagę, że wyniki działania formuły znajdującej się w komórce B15 oraz data znajdująca się w komórce B16 nie uległy zmianie. Dzieje się tak, ponieważ (jak pamiętasz) nasze makro modyfikuje zawartość tylko takich komórek, w których przechowywany jest tekst.

Jeżeli skoroszyt zawierający makro `ChangeCase` pozostaje otwarty, możesz tego makra używać w dowolnym innym skoroszycie. Jeżeli jednak zamkniesz skoroszyt zawierający makro, naciśnięcie kombinacji klawiszy `Ctrl+Shift+C` nie będzie przynosiło żadnych rezultatów.

Jeżeli makro nie działa poprawnie, powinieneś jeszcze raz przejść, krok po kroku, opisane etapy tworzenia formularza *UserForm* i powiązanych z nim procedur VBA w celu zlokalizowania i usunięcia błędu. Nie powinieneś jednak z tego powodu popadać w czarną rozpaczę, ponieważ takie wyszukiwanie błędów (w jargonie programistów nazywane *odpluskowaniem*) jest zupełnie normalnym i naturalnym etapem projektowania i tworzenia makr. Jeżeli jednak z takiego czy innego powodu nie będziesz mógł sobie poradzić z tym zadaniem, możesz po prostu pobrać cały skoroszyt zawierający to ćwiczenie ze strony internetowej tej książki i porównać z Twoim kodem, dzięki czemu szybko przekonasz się, co poszło nie tak, jak powinno.

274 Część IV: Komunikacja z użytkownikiem

	A	B	C	D
1		STYCZEŃ		
2		LUTY		
3		MARZEC		
4		KWIETIEŃ		
5		MAJ		
6		CZERWIEC		
7		LIPIEC		
8		SIERPIEŃ		
9		WRZESIEŃ		
10		PAŹDZIERNIK		
11		LISTOPAD		
12		GRUDZIEŃ		
13				
14				
15	Formula:	wtorek, 4 wrzesień 2012		
16	Data:	niedziela, 13 styczeń 2013		
17				
18				

Rysunek 16.9.

Tekst w zaznaczonych komórkach został zamieniony na wielkie litery

Rozdział 17

Praca z formantami formularza UserForm

W tym rozdziale:

- poznasz typy formantów dostępnych na formularzach *UserForm*,
- dowiesz się, jak zmieniać właściwości różnych formantów,
- nauczysz się zasad pracy z formantami formularzy *UserForm*.

Użytkownik współpracuje z niestandardowymi oknami dialogowym (czyli inaczej mówiąc, z formularzami *UserForm*) za pomocą różnego typu formantów, takich jak przyciski, pola tekstowe, przyciski opcji i inne. Kod VBA Twojego makra pobiera dane z takich formantów i na ich podstawie wykonuje różne operacje. Projektując formularz, masz do dyspozycji całkiem spory zestaw formantów — w tym rozdziale poznasz wiele tajemnic z nimi związanych.

Jeżeli uważnie wykonywałeś ćwiczenia opisywane w rozdziale 16., możesz śmiało powiedzieć, że masz już pewne doświadczenie w pracy z formularzami *UserForm*. W tym rozdziale bardzo poszerzysz swoją wiedzę.

Rozpoczynamy pracę z formantami formularzy UserForm

W tym podrozdziale opowiem, w jaki sposób możesz dodawać formanty do formularza *UserForm*, nadawać im nazwy związane z ich rolami oraz jak modyfikować ich właściwości.

Zanim będziesz mógł rozpocząć pracę z formantami, musisz najpierw utworzyć formularz *UserForm*. Aby to zrobić, przejdź do edytora VBE i z menu głównego wybierz polecenie *Insert/UserForm*. Po dodaniu formularza powinieneś się również upewnić, że w oknie *Project* zaznaczony jest właściwy projekt VBA (oczywiście, tylko wtedy, kiedy jednocześnie otworzyłeś więcej projektów).



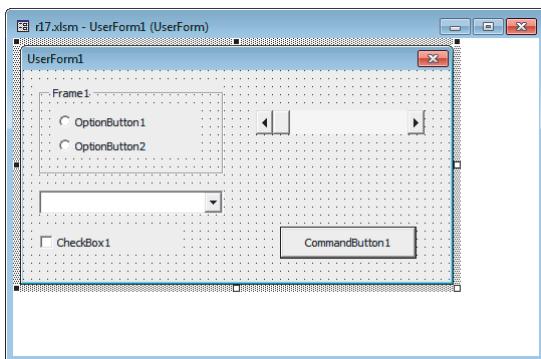
Dodawanie formantów

Co ciekawe, edytor VBE nie posiada w menu głównym żadnego polecenia, które pozwalałoby na dodawanie formantów do formularzy *UserForm*. Aby to zrobić, musisz skorzystać z „pływającego” okna dialogowego *Toolbox*, które opisałem w rozdziale 16. Zazwyczaj okno to pojawia się na ekranie automatycznie, kiedy w edytorze VBE klikniesz formularz *UserForm*. Jeżeli jednak z takiego czy innego powodu okno nie pojawi się na ekranie, możesz je przywołać, wybierając z menu głównego polecenie *View/Toolbox*.

Aby umieścić nowy formant na formularzu *UserForm*, powinieneś wykonać polecenia opisane poniżej.

- 1. W oknie *Toolbox* odszukaj i kliknij ikonę wybranego formantu.**
- 2. Przenieś wskaźnik myszy nad formularz *UserForm*, kliknij lewym przyciskiem myszy i przeciągnij mysz tak, aby „narysować” formant o żądanach rozmiarach. Po zakończeniu możesz przeciągnąć nowo utworzony formant w docelowe miejsce na formularzu.**

Zamiast tego możesz po prostu przeciągnąć wybrany formant z okna *ToolBox* do okna formularza *UserForm*, co spowoduje utworzenie formantu o domyślnych rozmiarach. Na rysunku 17.1 przedstawiam formularz *UserForm*, na którym umieściłem kilka formantów: dwa przyciski opcji (formanty *RadioButton*, zlokalizowane wewnętrz kontenera *Frame*), pole typu kombi (formant *ComboBox*), pole wyboru (formant *CheckBox*), poziomy pasek przewijania (formant *ScrollBar*) oraz przycisk polecenia (formant *CommandButton*).



Rysunek 17.1.
Formularz *UserForm*, na którym znajduje się kilka formantów

Podczas projektowania formularzy *UserForm* edytor VBE może wyświetlać na ekranie siatkę prowadnic, które znakomicie ułatwiają precyzyjne rozmieszczenie formantów we właściwych miejscach. Kiedy dodajes nowy lub przesuwasz istniejący formant w nowe miejsce, jest on automatycznie „przyciągany” do węzłów siatki. Jeżeli jednak z jakiegoś powodu nie lubisz tego mechanizmu, możesz wyłączyć wyświetlanie siatki. Aby to zrobić, wykonaj polecenia przedstawione poniżej.

- 3. Z menu głównego edytora VBE wybierz polecenie *Tools/Options*.**
- 4. W oknie *Options*, które pojawi się na ekranie, przejdź na kartę *General*.**
- 5. Usuń zaznaczenie wybranych opcji w grupie *Form Grid Settings*.**

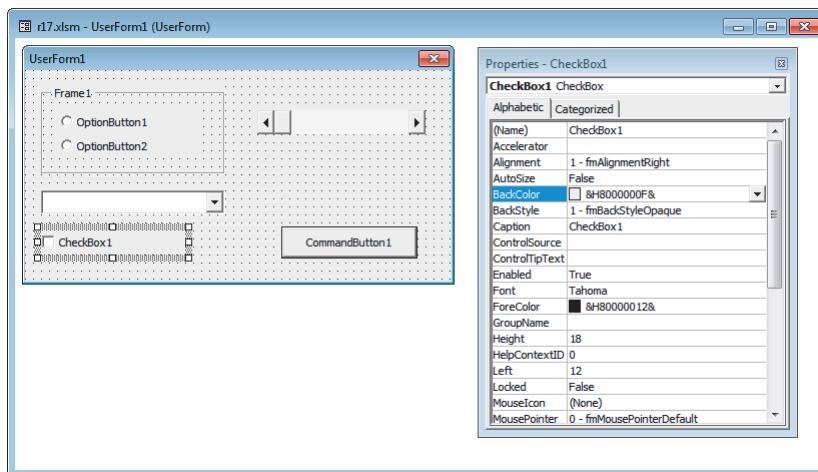


Wprowadzenie do właściwości formantów

Każdy formant umieszczany na formularzu *UserForm* ma swój zestaw właściwości, które odpowiadają za jego wygląd i zachowanie. Właściwości formantów możesz modyfikować na dwa sposoby.

- ✓ W czasie projektowania — czyli w trakcie projektowania formularza *UserForm*. Możesz to zrobić, modyfikując odpowiednie opcje w oknie *Properties*.
- ✓ W czasie działania programu — czyli po uruchomieniu makra, kiedy formularz *UserForm* zostaje wyświetlony na ekranie. Zmiany właściwości formantów w czasie działania programu możesz wykonać za pomocą odpowiednich polecen język VBA. Pamiętaj, że zmiany właściwości formantów, wprowadzone w czasie działania programu są zawsze tymczasowe, dotyczą zawartości aktualnie wyświetlanego okna dialogowego i nie mają żadnego wpływu na bazowy obiekt *UserForm*, który zaprojektowałeś.

Kiedy dodajesz nowy formant do formularza *UserForm*, niemal zawsze będziesz musiał zmodyfikować kilka jego właściwości. Takich zmian na etapie projektowania możesz dokonać za pomocą okna *Properties* (aby przywołać to okno na ekran, naciśnij klawisz F4). Na rysunku 17.2 przedstawiam wygląd okna *Properties*, wyświetlającego właściwości obiektu zazначенego na formularzu *UserForm*, czyli w tym przypadku właściwości formantu *CheckBox* (pole wyboru).



Rysunek 17.2.
Z pomocą okna *Properties* możesz zmieniać właściwości formantów już na etapie projektowania formularza *UserForm*



Aby zmienić właściwości formantu w czasie działania programu, musisz napisać odpowiedni kod VBA. Takie rozwiązanie może być potrzebne na przykład wtedy, kiedy po zaznaczeniu wybranej opcji przez użytkownika będziesz chciał ukryć część formantów. W takiej sytuacji będziesz musiał napisać parę wierszy kodu, których zadaniem będzie zmiana wartości właściwości *Visible* wybranych formantów.

Każdy formant ma swój własny zestaw właściwości, aczkolwiek niektóre z właściwości powtarzają się niemal dla wszystkich formantów (na przykład `Name`, `Width` czy `Height`). W tabeli 17.1 zamieszczam zestawienie najczęściej wykorzystywanych właściwości wielu formantów.

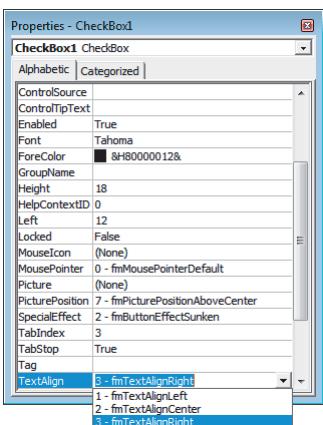
Tabela 17.1. Wspólne właściwości formantów

Właściwość	Opis
<code>Accelerator</code>	Odpowiada za to, która litera w etykiecie formantu będzie podkreślona. Dodatkowo taka litera w połączeniu z lewym klawiszem <code>Alt</code> będzie spełniała rolę skrótu klawiszowego, za pomocą którego użytkownik będzie mógł szybko wybrać taki formant.
<code>AutoSize</code>	Jeżeli ma wartość <code>True</code> , rozmiary formantu będą się automatycznie dostosowywały do rozmiarów tekstu wpisanego w jego etykiecie (właściwość <code>Caption</code>).
<code>BackColor</code>	Kolor tła formantu.
<code>BackStyle</code>	Styl tła formantu (przezroczyste lub nieprzejrzyste).
<code>Caption</code>	Etykieta formantu, zawiera tekst, który jest wyświetlany na formancie.
<code>Left</code> oraz <code>Top</code>	Wartości tych właściwości określają pozycję danego formantu w oknie formularza <code>UserForm</code> .
<code>Name</code>	Nazwa formantu, która jest używana w odwołaniach do formantu z poziomu kodu makra. Domyslnie, nazwy formantów tworzone są w oparciu o typ formantu. Nazwę formantu możesz zmienić, ale musisz przestrzegać zasad nadawania nazw i upewnić się, że nowa nazwa jest unikatowa w obrębie danego okna dialogowego.
<code>Picture</code>	Grafika, która będzie wyświetlana w miejscu formantu. Grafiką może być obraz, zdjęcie lub inny plik graficzny wskazany przez użytkownika; możesz również zaznaczyć właściwość <code>Picture</code> i wkleić do niej obraz graficzny, który wcześniej skopiowałeś do schowka systemowego.
<code>Value</code>	Wartość formantu.
<code>Visible</code>	Jeżeli ta właściwość ma wartość <code>False</code> , formant nie będzie widoczny w oknie dialogowym.
<code>Width</code> oraz <code>Height</code>	Wartości tych właściwości określają (odpowiednio) szerokość i wysokość formantu.

Kiedy zaznaczasz wybrany formant w edytorze VBE, jego właściwości automatycznie pojawiają się w oknie *Properties*. Aby zmienić wybraną właściwość, po prostu zaznacz ją w oknie *Properties* i wpisz nową wartość. Niektóre właściwości posiadają mechanizm pozwalający wybrać jedną z predefiniowanych wartości. Dobrym przykładem może być właściwość `TextAlign`; po jej wybraniu pojawia się lista rozwijana zawierająca dostępne wartości, co zostało przedstawione na rysunku 17.3.

Formanty okien dialogowych — szczegóły

W kolejnych podrozdziałach omówię poszczególne typy formantów, których możesz używać podczas tworzenia niestandardowych okien dialogowych (formularzy `UserForm`). Oczywiście, nie będę szczegółowo omawiać wszystkich właściwości każdego z dostępnych



Rysunek 17.3.
Wartości niektórych właściwości można wybierać z listy rozwijanej

formantów, ponieważ wymagałoby to napisania książki ze cztery razy grubszej niż ta, którą trzymasz w ręku (nie mówiąc już o tym, że byłaby to bardzo nudna książka).

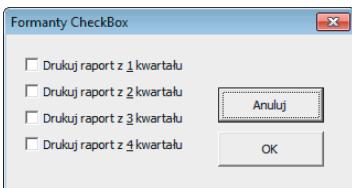
Pomoc systemowa dla formantów i ich właściwości jest bardzo rozbudowana. Aby znaleźć pełne informacje na temat wybranej właściwości danego formantu, powinieneś zaznaczyć ją w oknie *Properties* i nacisnąć klawisz *F1*.

Wszystkie przykłady omawiane w tym podrozdziale znajdziesz na stronie internetowej tej książki.



Formant CheckBox (pole wyboru)

Formant CheckBox (pole wyboru) jest używany podczas pobierania wartości dwustanowych typu *tak* lub *nie*, *prawda* lub *falsz*, *włączony* lub *wyłączony* i tym podobnych. Na rysunku 17.4 przedstawiam kilka przykładów zastosowania formantów CheckBox.



Rysunek 17.4.
Okno dialogowe z formantami CheckBox

Poniżej zamieszczam opis najbardziej użytecznych właściwości formantu CheckBox.

- ✓ **Accelerator** — znak, który pozwala użytkownikowi zmienić wartość formantu przy użyciu klawiatury. Jeśli na przykład akceleratorem formantu jest znak *A*, naciśnięcie kombinacji klawiszy *Alt+A* spowoduje zmianę wartości (stanu) pola wyboru (z niezaznaczonego na zaznaczone lub odwrotnie). W przykładzie przedstawionym na rysunku akceleratorami poszczególnych pól wyboru są kolejne cyfry reprezentujące kwartały (odpowiednio *Alt+1*, *Alt+2* i tak dalej).

- ✓ **ControlSource** — adres komórki arkusza, która jest przypisana do danego pola wyboru. Jeżeli pole wyboru jest zaznaczone, w komórce wyświetlana jest wartość *PRAWDA*. Podobnie, jeżeli pole wyboru nie jest zaznaczone, w komórce wyświetlana jest wartość *FAŁSZ*. Użycie tej właściwości jest opcjonalne i w praktyce w większości przypadków pole wyboru nie jest przypisane do żadnej komórki arkusza.
- ✓ **Value** — jeżeli ta właściwość ma wartość *True*, pole wyboru jest zaznaczone. Jeżeli ta właściwość ma wartość *False*, pole wyboru nie jest zaznaczone.

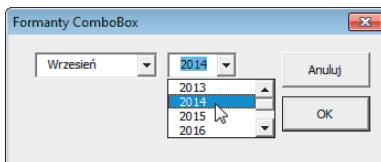


Nie powinieneś mylić formantów CheckBox (pola wyboru) z formantami OptionButton (przyciski opcji). Co prawda, do pewnego stopnia wyglądają podobnie, ale są wykorzystywane do zupełnie innych celów.

Formant ComboBox (pole kombi)

Formant ComboBox (pole kombi) jest bardzo podobny do formantu ListBox (pole listy), który opiszę już niedługo, dalej w tym podrziale. Pole kombi często jest nazywane listą rozwijaną. Inną różnicą między polem listy a polem kombi jest to, że w tym drugim przypadku użytkownik może w nim wpisywać wartości, których nie ma w zestawieniu predefiniowanych elementów listy. Na rysunku 17.5 przedstawiam okno dialogowe zawierające dwa formanty ComboBox. Formant po prawej stronie okna (reprezentujący rok) jest uaktywniony, stąd lista jego elementów jest rozwinięta.

Rysunek 17.5.
Okno dialogowe
z formantami
ComboBox



Poniżej zamieszczam opis niektórych użytecznych właściwości formantu ComboBox.

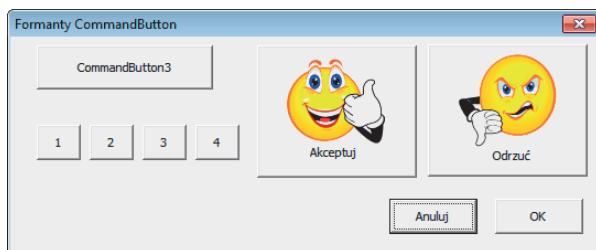
- ✓ **ControlSource** — adres komórki, która przechowuje wartość wybraną w polu kombi.
- ✓ **ListRows** — liczba elementów wyświetlnych po rozwinięciu listy.
- ✓ **ListStyle** — wygląd elementów listy.
- ✓ **RowSource** — adres zakresu komórek zawierającego listę elementów, które będą wyświetlane w polu kombi.
- ✓ **Style** — określa, czy formant będzie się zachowywał jak pole kombi, czy jak klasyczna lista rozwijana. Różnica polega na tym, że w przypadku listy rozwijanej użytkownik nie może do niej dopisywać nowych elementów.
- ✓ **Value** — zawiera element wybrany przez użytkownika z listy i wyświetlany w polu kombi.

Jeżeli elementy wyświetlane w polu kombi nie są przechowywane w arkuszu, możesz dodawać nowe elementy do listy za pomocą metody *AddItem*. Więcej szczegółowych informacji na temat tej metody znajdziesz w rozdziale 18.



Formant CommandButton (przycisk polecenia)

Formanty CommandButton to po prostu powszechnie spotykane w oknach dialogowych przyciski polecen. Taki przycisk jest zupełnie bezużyteczny, dopóki nie napiszesz odpowiedniej procedury obsługi zdarzeń, która będzie wykonywana w momencie, kiedy przycisk zostaje naciśnięty. Na rysunku 17.6 przedstawiam okno dialogowe zawierające 9 przycisków polecen. Na dwa z tych przycisków zostały nałożone elementy graficzne — taki efekt możesz uzyskać, kopując wybrany element graficzny (na przykład clipart) do schowka systemowego, a następnie wklejając go w oknie *Properties* jako wartość właściwości Picture tego formantu.



Rysunek 17.6.
Okno dialogowe
z formantami
CommandButton

Kiedy wybrany przycisk polecenia zostanie naciśnięty (kliknięty lewym przyciskiem myszy), automatycznie wykonywana jest procedura obsługi zdarzenia, której nazwa składa się z nazwy formantu, znaku podkreślenia oraz słowa kluczowego `Click`. Jeśli na przykład przycisk polecenia nosi nazwę `MyButton`, kliknięcie go spowoduje wywołanie procedury o nazwie `MyButton_Click`. Procedury obsługi zdarzeń są przechowywane w oknie *Code* formularza *UserForm*.

Poniżej zamieszczam opis niektórych użytecznych właściwości formantu `CommandButton`.

- ✓ **Cancel** — jeżeli właściwość ta ma wartość `True`, naciśnięcie klawisza `Esc` spowoduje wykonanie makra przypisanego do tego przycisku. Pamiętaj, że w całym formularzu tylko jeden przycisk polecenia powinien mieć ustawioną tę właściwość na wartość `True`.
- ✓ **Default** — jeżeli właściwość ta ma wartość `True`, naciśnięcie klawisza `Enter` spowoduje wykonanie makra przypisanego do tego przycisku. Podobnie jak w poprzednim przypadku, pamiętaj, że w całym formularzu tylko jeden przycisk polecenia powinien mieć ustawioną tę właściwość na wartość `True`.

Formant Frame (pole grupy)

Formant Frame spełnia rolę kontenera dla innych formantów. Używa się go najczęściej ze względów estetycznych bądź w celu wydzielenia logicznie ze sobą powiązanych grup formantów. Pola grupy są szczególnie użyteczne w sytuacji, kiedy w oknie dialogowym znajduje się więcej niż jedna grupa przycisków opcji (formantów `OptionButton`); zobacz punkt „Formant OptionButton”, dalej w tym rozdziale.

Poniżej zamieszczam opis niektórych użytecznych właściwości formantu Frame.

- ✓ **BorderStyle** — odpowiada za wygląd formantu Frame.
- ✓ **Caption** — pozwala zdefiniować tekst, jaki będzie wyświetlany w lewej górnej części ramki kontenera. Jeżeli nie chcesz, aby pole grupy miało etykietę, powinieneś ustawić wartość tej właściwości na pusty ciąg znaków.

Formant Image (pole obrazu)

Formanty Image pozwalają na wyświetlanie w oknie dialogowym obrazów i innych elementów graficznych. Przykładowo pola obrazu możesz użyć do wyświetlania w oknie dialogowym logo Twojej firmy. Na rysunku 17.7 przedstawiam okno dialogowe, w którym za pomocą pola obrazu wyświetlany jest sympatyczny kotek.



Rysunek 17.7.
Okno dialogowe
z formantem
Image

Poniżej zamieszczam opis niektórych użytecznych właściwości formantu Image.

- ✓ **Picture** — element graficzny, który będzie wyświetlany w polu obrazu.
- ✓ **PictureSizeMode** — właściwość ta pozwala na określenie, w jaki sposób obraz będzie wyświetlany, jeżeli jego wymiary będą inne niż wymiary pola obrazu.

Kiedy klikniesz właściwość Picture, zostaniesz poproszony o wskazanie pliku graficznego, który będzie wyświetlany w polu obrazu. Po wskazaniu takiego obrazu jego kopię zostanie osadzona i będzie przechowywana w skoroszcycie. Dzięki takiemu rozwiązaniu, kiedy udostępniasz taki skoroszycy innym użytkownikom, nie musisz dodatkowołączać kopii plików graficznych, które zostały użyte w oknach dialogowych.

Zamiast pobierać obraz graficzny z pliku, możesz skorzystać ze schowka systemowego i po prostu wkleić do właściwości Picture obraz skopiowany wcześniej do schowka. Excel zawiera rozbudowaną kolekcję elementów graficznych typu clipart. Aby wstawić wybrany clipart do arkusza, przejdź na kartę *WSTAWIANIE* (znajdziesz ją na Wstążce w głównym oknie Excela) i z grupy *Ilustracje* wybierz polecenie *Obrazy online*. Po wstawieniu obrazka do arkusza zaznacz go i naciśnij kombinację klawiszy *Ctrl+C*, aby skopiować obraz do schowka. Następnie przejdź do edytora VBE, włącz widok formularza, kliknij pole obrazu i w oknie dialogowym *Properties* odszukaj i zaznacz właściwość *Picture*.





Teraz naciśnij kombinację klawiszy *Ctrl+V*, aby wstawić do właściwości obraz graficzny ze schowka. Na koniec możesz usunąć obraz graficzny, który wstawiłeś do arkusza (jego kopia znajduje się już w polu obrazu w oknie dialogowym!).

Niektóre pliki graficzne mają bardzo duże rozmiary, stąd po wykonaniu takiej operacji rozmiar skoroszytu może dosyć znacznie wzrosnąć. Aby uzyskać najbardziej optymalne rezultaty, powinieneś używać plików graficznych o względnie niewielkich rozmiarach.

Formant Label (pole etykiety)

Formant Label po prostu wyświetla tekst w oknie dialogowym. Na rysunku 17.8 przedstawiam okno dialogowe, na którym umieszczonych zostało kilka formantów Label. Jak widać, możesz w bardzo szerokim stopniu dostosowywać wygląd takich formantów do własnych potrzeb.

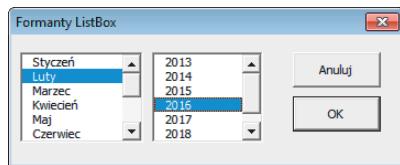


Rysunek 17.8.
Wygląd formantów Label można zmieniać w niemal doowy sposob

Formant ListBox (pole listy)

Formant ListBox (pole listy) wyświetla listę elementów, spośród których użytkownik może wybrać jedną lub więcej pozycji. Na rysunku 17.9 przedstawiam okno dialogowe zawierające dwa formanty ListBox.

Rysunek 17.9.
Okno dialogowe z formantami ListBox



Formanty ListBox są bardzo elastyczne. Możesz przykładowo zdefiniować zakres komórek arkusza, gdzie przechowywane będą elementy listy, a dodatkowo taki zakres może się składać z wielu kolumn. Oczywiście, oprócz tego możesz wypełnić listę elementami za pomocą kodu VBA.

Jeżeli formant **ListBox** nie jest wystarczająco wysoki, aby wyświetlić wszystkie elementy listy jednocześnie, wzdłuż prawej krawędzi listy automatycznie pojawi się suwak, za pomocą którego możesz przewijać zawartość listy.

Poniżej zamieszczam opis niektórych użytecznych właściwości formantu **ListBox**.

- ✓ **ControlSource** — adres komórki, która przechowuje wartość wybraną z listy przez użytkownika.
- ✓ **IntegralHeight** — jeżeli właściwość ta ma wartość *True*, wysokość formantu **ListBox** jest automatycznie dostosowywana tak, aby wyświetlać pełne wiersze tekstu podczas przewijania elementów listy. Jeżeli właściwość ma wartość *False*, podczas przewijania elementów listy niektóre wiersze mogą być widoczne częściowo. Zwróć uwagę na fakt, że kiedy właściwość ta ma wartość *True*, rzeczywista wysokość formantu **ListBox** po wyświetleniu w oknie dialogowym może być nieco inna niż ustalona podczas projektowania okna w edytorze VBE. Innymi słowy, wysokość listy może się automatycznie zmienić tak, aby ostatni element listy był wyświetlany w całości.
- ✓ **ListStyle** — odpowiada za wygląd elementów listy.
- ✓ **MultiSelect** — określa, czy użytkownik może zaznaczać więcej niż jeden element listy.
- ✓ **RowSource** — adres zakresu komórek zawierającego elementy, które będą wyświetlane na liście.
- ✓ **Value** — tekst elementu wybranego z listy przez użytkownika.

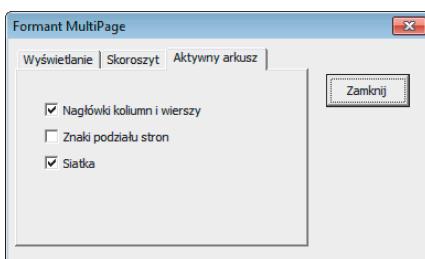


Jeżeli właściwość **MultiSelect** formantu **ListBox** jest ustawiona na wartość 1 lub 2, użytkownik może wybierać z listy więcej niż jeden element. W takiej sytuacji nie możesz zdefiniować właściwości **ControlSource**; zamiast tego będziesz musiał napisać makro sprawdzające, które elementy listy zostały zaznaczone. W rozdziale 18. dowiesz się, jak to zrobić.

Formant MultiPage

Formant **MultiPage** umożliwia tworzenie okien dialogowych wyposażonych w karty, takich jak na przykład okno *Formatowanie komórek* (które pojawia się w Excelu, kiedy naciśniesz kombinację klawiszy *Ctrl+1*). Na rysunku 17.10 przedstawiam przykład niestandardowego okna dialogowego, wykorzystującego formant **MultiPage**. W tym konkretnym przypadku formant **MultiPage** składa się z trzech kart.

Rysunek 17.10.
Zastosowanie
formantu **MultiPage**
pozwala
na tworzenie
okien dialogo-
wych podziel-
nych na karty



Poniżej zamieszczam opis niektórych użytecznych właściwości formantu MultiPage.

- ✓ **Style** — określa wygląd formantu. Karty formantu mogą być wyświetlane domyślnie (na górze formantu), wzdłuż lewej krawędzi formantu, jako przyciski lub mogą być ukryte (karty nie są wtedy widoczne, a za to, która karta jest wyświetlana, odpowiada kod VBA).
- ✓ **Value** — określa, która karta formantu jest wyświetlana na ekranie. Wartość 0 powoduje wyświetlenie pierwszej karty, wartość 1 wyświetla drugą kartę i tak dalej.



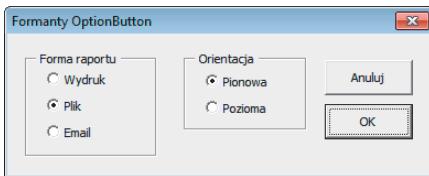
Domyślnie formant MultiPage zawiera dwie karty. Aby dodać kolejną kartę, kliknij prawym przyciskiem myszy dowolną kartę formantu i z menu podręcznego, które pojawi się na ekranie, wybierz polecenie *New Page*.

Formant OptionButton (przycisk opcji)

Formant OptionButton (przycisk opcji) przydaje się, kiedy użytkownik musi wybrać jedną opcję spośród kilku dostępnych. Formant ten zawsze ma postać grupy złożonej z przynajmniej dwóch opcji. Na rysunku 17.11 przedstawiam okno dialogowe zawierające dwie grupy przycisków opcji, z których każda jest umieszczona w swoim własnym kontenerze Frame.

Rysunek 17.11.

Dwie grupy formantów
OptionButton;
każda grupa
jest umieszczona
na swoim
własnym kon-
tenerze Frame



Poniżej zamieszczam opis niektórych użytecznych właściwości formantu OptionButton.

- ✓ **Accelerator** — znak, który pozwala użytkownikowi zmienić wartość formantu przy użyciu klawiatury. Jeśli na przykład akceleratorem formantu jest znak *C*, naciśnięcie kombinacji klawiszy *Alt+C* spowoduje wybranie tego przycisku opcji.
- ✓ **GroupName** — nazwa, która pozwala na zidentyfikowanie przycisku opcji jako formantu powiązanego z innymi przyciskami należącymi do danej grupy przycisków opcji.
- ✓ **ControlSource** — adres komórki arkusza, która jest przypisana do danego przycisku opcji. Jeżeli dana opcja jest zaznaczona, w komórce wyświetlana jest wartość *PRAWDA*. Jeżeli natomiast dana opcja nie jest zaznaczona, w komórce wyświetlana jest wartość *FAŁSZ*.
- ✓ **Value** — jeżeli ta właściwość ma wartość *True*, opcja jest zaznaczona. Jeżeli ta właściwość ma wartość *False*, opcja nie jest zaznaczona.

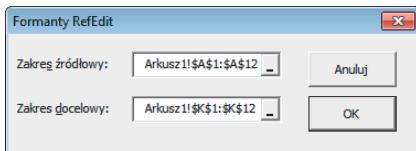


Kiedy okno dialogowe zawiera więcej niż jedną grupę przycisków opcji, dla przycisków w osobnych grupach przycisków **musisz** ustawić osobną nazwę grupy (właściwość **GroupName**). Inaczej wszystkie przyciski opcji znajdujące się w oknie dialogowym będą należały do tej samej grupy. Zamiast tego możesz również umieścić przyciski opcji z każdej grupy w osobnym kontenerze Frame, co spowoduje automatyczne przypisanie przycisków opcji z danego kontenera do jednej grupy.

Formant RefEdit (pole zakresu)

Formant RefEdit pozwala użytkownikowi na wprowadzenie adresu komórki bądź wskazanie zakresu komórek arkusza przy użyciu myszy. Na rysunku 17.12 możesz zobaczyć okno dialogowe zawierające dwa formanty RefEdit. Adres wybranej komórki lub zakresu komórek jest przechowywany we właściwości **Value** (w postaci ciągu znaków).

Rysunek 17.12.
Okno dialogowe
zawierające
dwa formanty
RefEdit



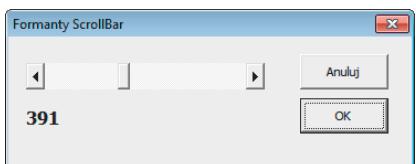
Formanty RefEdit czasami mogą powodować problemy; może się tak zdarzyć, kiedy umieścisz je na rozbudowanych, złożonych formularzach *UserForm*. Aby uniknąć kłopotów, nie powinieneś raczej umieszczać takich formantów wewnętrz kontenerów Frame i MultiPage.

Formant ScrollBar (pasek przewijania)

Kiedy umieszczasz w oknie dialogowym formant ScrollBar, czyli inaczej mówiąc, pasek przewijania, możesz wybrać, czy będzie poziomy, czy pionowy. Formant ScrollBar jest dosyć podobny do formantu SpinButton (o którym piszę nieco dalej w tym rozdziale). Podstawowa różnica między tymi formantami polega na tym, że na pasku przewijania użytkownik może szybko zmienić wartość, przesuwając przycisk suwaka. Kolejna różnica jest taka, że kliknięcie górnego przycisku pionowego paska przewijania powoduje zmniejszenie wartości, co może być niezbyt intuicyjnym rozwiązaniem — z tego powodu formanty ScrollBar nie zawsze są dobrymi zamiennikami formantów SpinButton.

Na rysunku 17.13 przedstawiam okno dialogowe zawierające poziomy pasek przewijania. Wartość suwaka jest wyświetlana za pomocą formantu Label, znajdującego się poniżej paska przewijania.

Rysunek 17.13.
Okno dialogowe
z paskiem
przewijania
i etykietą



Poniżej zamieszczam opis niektórych użytecznych właściwości formantu ScrollBar.

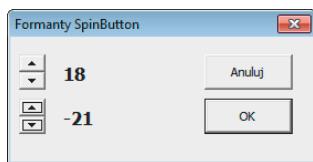
- ✓ **Value** — bieżąca wartość formantu.
- ✓ **Min** — minimalna (początkowa) wartość suwaka.
- ✓ **Max** — maksymalna (końcowa) wartość suwaka.
- ✓ **ControlSource** — adres komórki arkusza, która będzie wyświetlała bieżącą wartość formantu.
- ✓ **SmallChange** — wielkość kroku zmiany wartości formantu po naciśnięciu przycisku.
- ✓ **LargeChange** — wielkość kroku zmiany wartości formantu po kliknięciu po lewej lub prawej stronie suwaka.

Formanty ScrollBar są szczególnie użyteczne w sytuacji, kiedy użytkownik musi wybrać wartość z szerokiego zakresu wartości.

Formant SpinButton (pokrętło)

Formant SpinButton pozwala użytkownikowi na wybranie wartości poprzez klikanie przycisków ze strzałkami, znajdujących się po obu stronach formantu (jeden przycisk służy do zwiększenia wartości, a drugi do zmniejszania). Podobnie jak w przypadku pasków przewijania, pokrętła mogą być orientowane poziomo lub pionowo. Na rysunku 17.14 przedstawiam okno dialogowe, na którym umieszczone zostały dwa pionowe formanty SpinButton. Do każdego z tych formantów jest przypisany osobny formant Label, który wyświetla bieżącą wartość danego pokrętła (za pomocą odpowiednich procedur VBA).

Rysunek 17.14.
Okno dialogowe
z formantami
SpinBox



Poniżej zamieszczam opis niektórych użytecznych właściwości formantu SpinButton.

- ✓ **Value** — bieżąca wartość formantu.
- ✓ **Min** — minimalna (początkowa) wartość pokrętła.
- ✓ **Max** — maksymalna (końcowa) wartość pokrętła.
- ✓ **ControlSource** — adres komórki arkusza, która będzie wyświetlała bieżącą wartość formantu.
- ✓ **SmallChange** — wielkość kroku zmiany wartości formantu po naciśnięciu przycisku. Zazwyczaj ustawiany jest krok o wartości 1, ale możesz tutaj ustawić dowolną wartość.



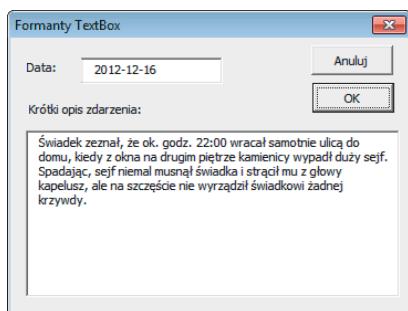
Jeżeli korzystasz z właściwości ControlSource formantu SpinButton, powinieneś pamiętać, że zmiana wartości formantu każdorazowo będzie powodowała przeliczenie arkusza. Z tego powodu arkusz zostanie przeliczony 12 razy, gdy na przykład użytkownik będzie zmieniał wartość pokrętła od 0 do 12. Jeśli korzystasz ze złożonych arkuszy kalkulacyjnych, których przeliczanie trwa długo, powinieneś raczej unikać używania właściwości ControlSource do przechowywania wartości pokrętła.

Formant TabStrip (pole karty)

Formant TabStrip jest dosyć podobny do formantu MultiPage, ale nie jest tak prosty w użyciu. Szczerze mówiąc, nie jestem pewny, dlaczego taki dziwny formant jest w ogóle dostępny. W praktyce możesz spokojnie o nim zapomnieć i zamiast niego korzystać z formantu MultiPage.

Formant TextBox (pole tekstowe)

Formant TextBox (pole tekstowe) umożliwia użytkownikowi wprowadzanie tekstu. Na rysunku 17.15 przedstawiam okno dialogowe zawierające dwa pola tekstowe.



Rysunek 17.15.
Okno dialogowe
z formantami
TextBox

Poniżej zamieszczam opis niektórych użytecznych właściwości formantu TextBox.

- ✓ **AutoSize** — jeżeli ma wartość *True*, rozmiary formantu będą się automatycznie dostosowywały do rozmiarów wpisanego tekstu.
- ✓ **ControlSource** — adres komórki, która przechowuje tekst wpisany w polu tekstowym.
- ✓ **IntegralHeight** — jeżeli właściwość ta ma wartość *True*, wysokość formantu TextBox jest automatycznie dostosowywana tak, aby wyświetlać pełne wiersze tekstu podczas przewijania elementów listy. Jeżeli właściwość ma wartość *False*, podczas przewijania elementów listy niektóre wiersze mogą być widoczne częściowo.
- ✓ **MaxLength** — maksymalna liczba znaków, które można wpisać w polu tekstowym. Jeżeli ta właściwość ma wartość 0, liczba znaków jest nieograniczona.
- ✓ **MultiLine** — jeżeli właściwość ta ma wartość *True*, pole tekstowe może wyświetlać więcej niż jeden wiersz tekstu.
- ✓ **TextAlign** — określa sposób wyrównywania tekstu w polu tekstowym.

- ✓ WordWrap — określa, czy długie wiersze wpisywane w polu tekstowym będą automatycznie zawijane.
- ✓ ScrollBars — określa rodzaj pasków przewijania widocznych w polu tekstowym. Do wyboru masz pasek poziomy, pionowy, oba paski jednocześnie lub bez pasków.

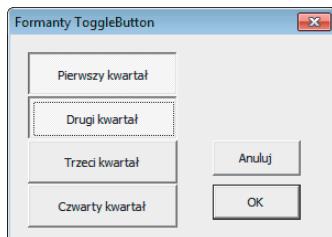


Kiedy umieszczasz w oknie dialogowym nowe pole tekstowe, właściwość WordWrap jest domyślnie ustawiona na wartość *True*, a właściwość MultiLine — na wartość *False*. Efekt? Zawijanie wierszy nie działa! Inaczej mówiąc, jeżeli chcesz, aby w polu tekstowym długie wiersze były zawijane, upewnij się, że ustawiałeś właściwość MultiLine na wartość *True*.

Formant ToggleButton (przycisk przełącznika)

Formant ToggleButton (przycisk przełącznika) posiada dwa stany: włączony lub wyłączony. Kliknięcie przycisku powoduje przełączanie pomiędzy dwoma stanami i zmianę wyglądu przycisku. Wartością formantu jest *True* (przycisk wcisnięty) lub *False* (przycisk nie jest wcisnięty). Na rysunku 17.16 przedstawiam okno dialogowe, na którym umieszczone zostały formanty ToggleButton; pierwsze dwa z nich są wcisnięte (czy jak kto woli, włączone).

Rysunek 17.16.
Okno dialogowe z formantami ToggleButton



Szczerze mówiąc, nigdy w swoich aplikacjach nie używałem tych formantów; zamiast nich zdecydowanie wolę korzystać z pól wyboru (formanty CheckBox).

Praca z formantami w oknach dialogowych

W tym podrozdziale omówię kilka podstawowych zasad pracy z formantami w oknach dialogowych.

Zmiana rozmiarów i przenoszenie formantów w inne miejsce

Po umieszczeniu formantu w oknie dialogowym możesz zmieniać jego rozmiary i przenosić w inne miejsce, korzystając ze standardowych technik posługiwania się myszą. Jeżeli chcesz bardziej precyzyjne ustawić rozmiary i położenie formantów, możesz użyć okna dialogowego *Properties* i wpisać odpowiednie wartości właściwości Height, Width, Left oraz Top.

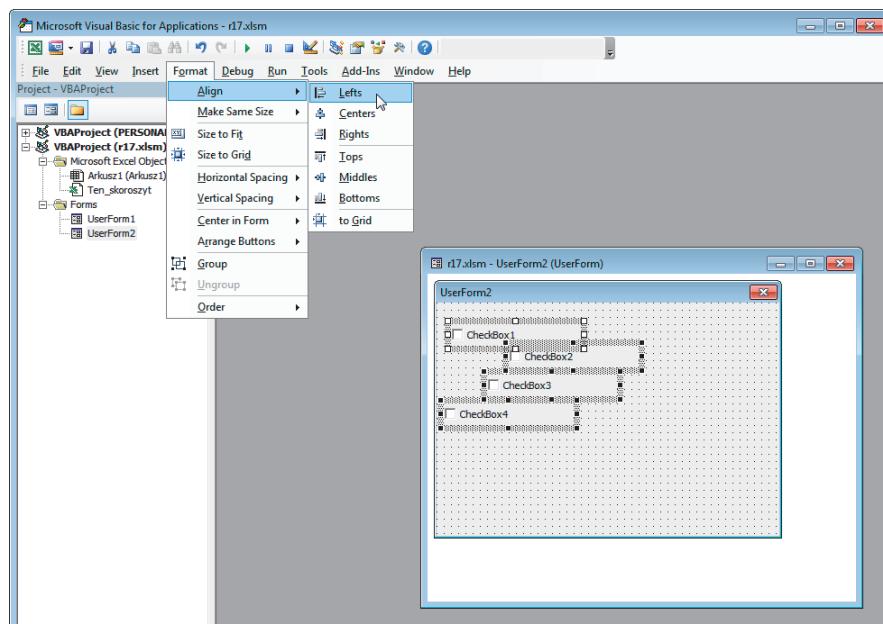


Jeśli trzeba, możesz zaznaczyć kilka formantów na raz. Aby to zrobić, po prostu wciśnij i przytrzymaj klawisz *Ctrl*, a następnie klikaj lewym przyciskiem myszy wybrane formanty. Zamiast tego możesz również wcisnąć i przytrzymać lewy przycisk myszy w wybranym miejscu formularza i następnie przeciągnąć mysz tak, aby „obrysować” formanty, które chcesz zaznaczyć. Kiedy zaznaczysz kilka formantów, w oknie *Properties* wyświetlane są tylko właściwości, które są wspólne dla wszystkich zaznaczonych formantów. Jeżeli zmienisz wartość jednej z takich właściwości, zmiana tej właściwości obejmie wszystkie zaznaczone formanty, co jest znacznie szybszą operacją niż zmiana tej właściwości w każdym z formantów z osobna.

Pamiętaj, że jeden formant może „przykrywać” inny formant; inaczej mówiąc, możesz umieszczać kilka formantów jeden na drugim i potem programowo decydować, który z nich jest w danym momencie widoczny. Jeżeli jednak nie masz naprawdę dobrego powodu, aby tak postępować, powinieneś raczej unikać takiego sposobu rozmieszczania formantów.

Rozmieszczanie i wyrównywanie położenia formantów w oknie dialogowym

W menu *Format* edytora VBE znajdziesz kilka ciekawych poleceń, które znakomicie ułatwiają precyzyjne rozmieszczanie i wyrównywanie położenia formantów w oknie dialogowym. Zanim wybierzesz jedno z takich poleceń, powinieneś najpierw zaznaczyć formanty, z którymi chcesz pracować. Działanie poszczególnych poleceń wydaje się dosyć oczywiste, stąd nie będę ich tutaj szczegółowo omawiać. Na rysunku 17.17 przedstawiam okno dialogowe z kilkoma formantami *CheckBox*, których położenie za chwilę zostanie wyrównane.



Rysunek 17.17.
Do wyrównania położenia formantów możesz użyć polecenia Format/Align



Kiedy zaznaczasz kilka formantów, zwróć uwagę na fakt, że uchwyty ostatnio zaznaczonego formantu mają kolor biały, a nie czarny, jak w pozostałych, zaznaczonych wcześniej formantach. Dzieje się tak dlatego, że formant z białymi uchwytyami będzie spełniał rolę formantu odniesienia podczas operacji zmiany położenia, rozmiaru i wyrównywania formantów, wykonywanych za pomocą polecień z menu *Format*.

Obsługa użytkowników preferujących korzystanie z klawiatury

Wielu użytkowników (wliczając w to autora tej książki) preferuje poruszanie się po oknach dialogowych za pomocą klawiatury, gdzie naciśkanie klawiszy *Tab* lub *Shift+Tab* przenosi fokus między kolejnymi formantami, a naciśnięcie odpowiedniego skrótu klawiszowego powoduje natychmiastowe aktywowanie danego formantu.

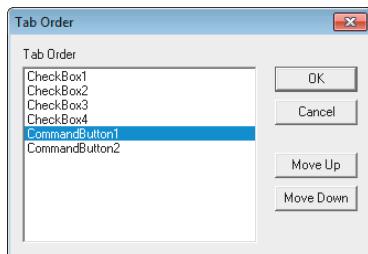
Aby upewnić się, że Twoje okno dialogowe będzie przyjazne również dla użytkowników, którzy są zapalonymi miłośnikami klawiatur, powinieneś pamiętać o dwóch sprawach. Oto one.

- ✓ Kolejność tabulacji formantów.
- ✓ Klawisze akceleratorów (skróty klawiszowe).

Zmiana kolejności tabulacji formantów

Kolejność tabulacji formantów decyduje o tym, w jakiej kolejności będą aktywowane formanty, kiedy użytkownik będzie nacisnął klawisz *Tab* lub kombinację klawiszy *Shift+Tab*. Oprócz tego, kolejność tabulacji decyduje o tym, który z formantów będzie aktywny jako pierwszy, zaraz po wyświetleniu okna dialogowego na ekranie. Jeżeli na przykład użytkownik wprowadza tekst w polu tekstowym, oznacza to, że pole to jest aktywne (posiada fokus). Jeśli teraz użytkownik kliknie przycisk opcji, to taki przycisk staje się aktywny (fokus przechodzi z pola tekstowego na przycisk opcji). Pierwszy formant w kolejności tabulacji domyślnie otrzymuje fokus po wyświetleniu okna dialogowego na ekranie.

Aby zmienić kolejność tabulacji formantów, z menu głównego edytora VBE wybierz polecenie *View/Tab Order*. Zamiast tego możesz również kliknąć edytowane okno dialogowe prawym przyciskiem myszy i wybrać polecenie *Tab Order* z menu podręcznego. W obu przypadkach po wybraniu tego polecenia na ekranie pojawi się okno dialogowe *Tab Order*, przedstawione na rysunku 17.18.



Rysunek 17.18.
Okno dialogowe
Tab Order

W oknie *Tab Order* wyświetlana jest lista wszystkich formantów znajdujących się w danym oknie dialogowym. Kolejność tabulacji formantów odpowiada kolejności formantów na liście w oknie *Tab Order*. Aby zmienić miejsce na liście wybranego formantu, zaznacz go i następnie naciśnij przycisk *Move Up* lub *Move Down*. Możesz również zaznaczyć kilka formantów (na przykład klikając lewym przyciskiem myszy z wcisniętym klawiszem *Shift* lub *Ctrl*) i od razu zmienić pozycję wszystkich zaznaczonych formantów.



Zamiast używać okna dialogowego *Tab Order*, możesz ustalić pozycję danego formantu na liście kolejności tabulacji, ustawiając odpowiednio właściwości formantu w oknie *Properties*. Pierwszy formant na liście kolejności tabulacji ma właściwość *TabIndex* ustawioną na wartość 0. Jeżeli chcesz usunąć dany formant z listy kolejności tabulacji, powinieneś ustawić jego właściwość *TabIndex* na wartość *False*.

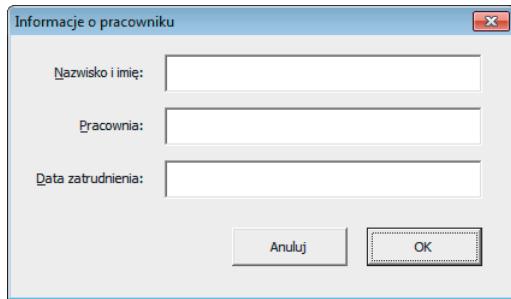


Niektóre formanty (takie jak *Frame* czy *MultiPage*) spełniają rolę kontenerów dla innych formantów. Formanty znajdujące się wewnątrz kontenerów mają swoją własną kolejność tabulacji. Aby na przykład zmienić kolejność tabulacji grupy przycisków opcji znajdujących się wewnątrz kontenera *Frame*, powinieneś najpierw zaznaczyć ten kontener, a dopiero potem wybrać z menu głównego edytora VBE polecenie *View/Tab Order*.

Definiowanie klawiszy skrótu

W większości przypadków będziesz chciał przypisać poszczególnym formantom w oknie dialogowym odpowiednie klawisze skrótu. Można to zrobić, przypisując wybraną literę do właściwości *Accelerator* danego formantu (w oknie *Properties*). Jeżeli dany formant nie posiada właściwości *Accelerator* (przykładem takiego formantu jest pole tekstowe) to i tak możesz zapewnić dostęp klawiszowy do tego pola za pomocą formantu *Label*. W praktyce robi się to tak, że przypisujesz klawisz skrótu do formantu *Label*, który następnie umieszczasz na liście kolejności tabulacji bezpośrednio przed polem tekstowym.

Na rysunku 17.19 przedstawiam formularz *UserForm* zawierający trzy pola tekstowe. Etykiety (formanty *Label*) opisujące poszczególne pola tekstowe posiadają przypisane klawisze skrótu, a dodatkowo każdy formant *Label* został umieszczony na liście kolejności tabulacji bezpośrednio przed „swim” polem tekstowym. Dzięki takiemu rozwiązaniu naciśnięcie na przykład kombinacji klawiszy *Alt+P* aktywuje pole tekstowe znajdujące się po prawej stronie etykiety *Pracownia*.



Rysunek 17.19.
Dzięki zastosowaniu etykiet możesz uzyskać dostęp z klawiatury do formantów, które nie posiadają klawiszy skrótu

Testowanie formularzy *UserForm*

Edytor VBE oferuje trzy sposoby testowania formularzy *UserForm* bez konieczności wywoływania ich z poziomu procedury VBA.

- ✓ Z menu głównego edytora VBE wybierz polecenie *Run/Run Sub/UserForm*.
- ✓ Naciśnij klawisz *F5*.
- ✓ Naciśnij przycisk *Run Sub/UserForm*, znajdujący się na standardowym pasku narzędzi edytora VBE.

Kiedy okno dialogowe (formularz *UserForm*) wyświetlane jest na ekranie w trybie testowym, możesz wypróbować kolejność tabulacji i funkcjonowanie klawiszy skrótu.

Estetyka okien dialogowych

Okna dialogowe mogą wyglądać dobrze, źle lub ich wygląd może się plasować gdzieś pośrodku między jednym a drugim. Dobrze wyglądające okna dialogowe są przyjemne dla oka, mają dobrze rozmieszczone i wyrównane formanty o starannie dobranych rozmiarach, a przeznaczenie poszczególnych formantów jest logiczne i zrozumiałe dla użytkownika. Źle wyglądające okna dialogowe zazwyczaj irytują użytkownika, a nierówne i źle wyrównane formanty, chaotycznie porozrzucane w oknie dialogowym często mogą sprawiać wrażenie, że projektant nie za bardzo wiedział, co naprawdę chce zrobić.

W miarę możliwości powinieneś redukować liczbę formantów, które znajdują się w oknie dialogowym. Jeżeli z takiego czy innego powodu musisz użyć dużej liczby formantów (arbitralnie możemy przyjąć, że „dużo” to znaczy więcej niż 10), powinieneś zastanowić się nad zastosowaniem kontenera *MultiPage*, aby podzielić zadanie, które musi wykonać użytkownik, na nieco mniejsze części.

Dobrym rozwiązaniem jest takie projektowanie formularzy *UserForm*, aby przypominały wbudowane okna dialogowe Excela. W miarę jak będziesz nabierał doświadczenia w programowaniu i tworzeniu własnych formularzy *UserForm*, będziesz w stanie zduplikować we własnych programach działanie praktycznie wszystkich mechanizmów Excela.

Rozdział 18

Techniki pracy z formularzami UserForm

W tym rozdziale:

- ▶ dowiesz się, jak korzystać z niestandardowych okien dialogowych we własnych aplikacjach,
- ▶ zobaczysz, jak w praktyce wygląda tworzenie niestandardowego okna dialogowego.

W poprzednich rozdziałach dowiedziałeś się, jak wstawić nowy formularz *UserForm* do projektu VBA, jak dodawać do niego nowe formanty oraz zmieniać właściwości formantów. Takie umiejętności — niestety — nie na wiele Ci się zdadzą, jeżeli nie będziesz wiedział, w jaki sposób używać formularza *UserForm* z poziomu kodu VBA. W tym rozdziale opiszę szereg zagadnień, które pomogą wypełnić tę lukę, oraz zaprezentuję kilka użytecznych technik i sztuczek, jakie będziesz mógł wykorzystywać podczas tworzenia swoich własnych aplikacji.

Zastosowanie własnych okien dialogowych

Kiedy w swojej aplikacji korzystasz z niestandardowych okien dialogowych, zazwyczaj tworzysz kod VBA, który realizuje następujące zadania.

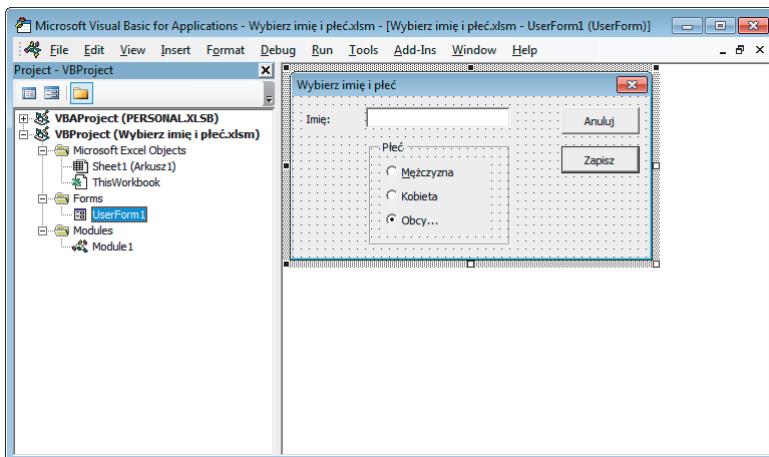
- ✓ Inicjuje formanty umieszczone na formularzu *UserForm*; na przykład możesz napisać kod, który będzie ustawał domyślne wartości dla poszczególnych formantów.
- ✓ Wyświetla okno dialogowe przy użyciu metody *Show* obiektu *UserForm*.
- ✓ Odpowiada na zdarzenia generowane przez poszczególne formanty, na przykład po naciśnięciu przycisku polecenia przez użytkownika.
- ✓ Sprawdza informacje wprowadzone przez użytkownika (o ile użytkownik nie nacisnął przycisku *Anuluj*). Takie zadanie nie zawsze jest niezbędne.
- ✓ Podejmuje różne działania na bazie informacji wprowadzonych przez użytkownika (jeżeli są poprawne).

Przykładowy formularz UserForm

W tym przykładzie przedstawię sposób implementacji w kodzie VBA pięciu zadań, o których była mowa w poprzednim podrozdziale. Użyję prostego okna dialogowego, które będzie pobierało od użytkownika dwie informacje — imię oraz płeć. Do pobierania imienia użytkownika skorzystam z pola tekstowego (formant TextBox), a płeć będzie można wybrać za pomocą jednego z trzech przycisków opcji: *Mężczyzna*, *Kobieta*, *Obcy*.... Informacje gromadzone w oknie dialogowym będą następnie zapisywane w pierwszym wolnym wierszu arkusza.

Tworzenie okna dialogowego

Na rysunku 18.1 przedstawiam gotowy formularz *UserForm*, który utworzony został na potrzeby tego ćwiczenia. Aby ułatwić sobie zadanie, powinieneś rozpocząć od utworzenia nowego, pustego skoroszytu, zawierającego tylko jeden arkusz. Następnie powinieneś wykonać polecenia opisane poniżej.



Rysunek 18.1.
Okno dialogowe umożliwia wprowadzenie imienia i wybranie płci

1. Naciśnij kombinację klawiszy *Alt+F11*, aby przejść do okna edytora VBE.
2. W oknie *Project* zaznacz pusty skoroszyt i z menu głównego wybierz polecenie *Insert/UserForm*.
Edytor VBE wstawi do projektu VBA nowy formularz *UserForm*.
3. Zmień właściwość **Caption** formularza na *Wybierz imię i płeć*.
Jeżeli okno *Properties* nie jest widoczne, naciśnij klawisz *F4*.

W oknie dialogowym powinieneś teraz wstawić osiem formantów.

- ✓ Dodaj formant Label i zmień poniższe właściwości.

Właściwość Wartość

Accelerator I

Caption Imię:

TabIndex 0

- ✓ Dodaj formant TextBox i zmień poniższe właściwości.

Właściwość Wartość

Name TextName

TabIndex 1

- ✓ Dodaj formant Frame i zmień następujące właściwości.

Właściwość Wartość

Caption Płeć

TabIndex 2

- ✓ Wewnątrz kontenera Frame umieść formant OptionButton i zmień następujące właściwości.

Właściwość Wartość

Accelerator M

Caption Mężczyzna

Name OptionMale

TabIndex 0

- ✓ Wewnątrz kontenera Frame umieść kolejny formant OptionButton i zmień następujące właściwości.

Właściwość Wartość

Accelerator K

Caption Kobieta

Name OptionFemale

TabIndex 1

- ✓ Wewnątrz kontenera Frame umieść jeszcze jeden formant OptionButton i zmień następujące właściwości.

Właściwość Wartość

Accelerator O

Caption Obcy...

Name OptionUnknown

TabIndex 2

Value True

- ✓ Poza obszarem kontenera Frame umieść formant CommandButton i zmień następujące właściwości.

Właściwość Wartość

Caption	<i>Zapisz</i>
Default	<i>True</i>
Name	<i>EnterButton</i>
TabIndex	<i>3</i>

- ✓ Umieść kolejny formant CommandButton i zmień następujące właściwości.

Właściwość Wartość

Caption	<i>Anuluj</i>
Cancel	<i>True</i>
Name	<i>CancelButton</i>
TabIndex	<i>4</i>

Jeżeli wykonujesz to ćwiczenie na komputerze w miarę czytania (a powinieneś tak robić), utworzenie formularza *UserForm* nie powinno Ci sprawić żadnych trudności. Upewnij się, że przed dodaniem formantów *OptionButton* utworzyłeś dla nich odpowiedni kontener *Frame*.

Kiedy tworzysz kilka podobnych formantów, od dodawania nowego formantu łatwiejsze może okazać się skopiowanie formantu już istniejącego. Aby skopiować formant, w trakcie przeciagania należy trzymać wciśnięty klawisz *Ctrl*.

Jeżeli nie masz czasu na samodzielne tworzenie formularza lub po prostu lubisz chodzić na skróty, możesz pobrać skoroszyt z tym przykładem ze strony internetowej tej książki.

Tworzenie kodu procedury wyświetlającej okno dialogowe

Po zakończeniu tworzenia formularza kolejnym etapem będzie napisanie procedury VBA wyświetlającej okno dialogowe na ekranie.

1. Przejdź do okna edytora VBE i z menu głównego wybierz polecenie *Insert/Module*, aby wstawić nowy moduł kodu.
2. W module kodu wpisz procedurę:

```
Sub GetData()
    UserForm1.Show
End Sub
```

Powyzsza procedura wykorzystuje metodę *Show* obiektu *UserForm* do wyświetlenia okna dialogowego na ekranie.



Udostępnianie makra użytkownikowi

Kolejnym etapem będzie udostępnienie użytkownikowi prostej metody uruchomienia naszej procedury. Aby to zrobić, wykonaj polecenia przedstawione poniżej.

1. Przejdź do głównego okna Excela.
 2. Przejdź na kartę **DEVELOPER**, naciśnij przycisk *Wstaw* znajdujący się w grupie opcji *Formanty* i z menu podręcznego wybierz *Przycisk*.
 3. Przeciągnij formant myszą tak, aby dobrać jego rozmiary i położenie na arkuszu.
- Na ekranie pojawi się okno dialogowe *Przypisywanie makra*.
4. Przypisz do przycisku makro *GetData*.
 5. Kliknij przycisk prawym przyciskiem myszy, z menu podręcznego wybierz polecenie *Edytuj tekst* i zmień etykietę przycisku na *Wprowadź dane*.



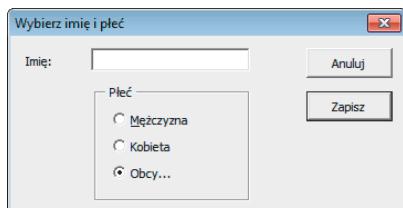
Jeżeli chcesz zrobić coś naprawdę fajnego, możesz spróbować umieścić przycisk uruchamiający makro *GetData* na pasku narzędzi *Szybki dostęp*. Aby to zrobić, kliknij prawym przyciskiem myszy pasek narzędzi *Szybki dostęp* i z menu podręcznego wybierz polecenie *Dostosuj pasek narzędzi Szybki dostęp*. Na ekranie pojawi się okno dialogowe *Opcje programu Excel*. Na liście rozwijanej *Wybierz polecenia* z wskaz opcję *Makra*. Następnie zaznacz makro *GetData* i naciśnij przycisk *Dodaj*. Jeżeli chcesz, możesz nacisnąć przycisk *Modyfikuj* i zmienić ikonę przypisaną do makra. Możesz również zrobić tak, aby ikona tego przycisku pojawiała się tylko wtedy, kiedy otwarty zostanie właściwy skoroszyt. Aby to zrobić, powinieneś przed dodaniem przycisku makra rozwinąć listę *Dostosuj pasek narzędzi Szybki dostęp*, znajdująca się w prawej, górnej części okna i zamiast opcji *Dla wszystkich dokumentów (domyślne)* wybrać opcję *Dla <nazwa_Twojego_skoroszytu>*.

Testowanie okna dialogowego

Aby sprawdzić, czy okno dialogowe działa dokładnie tak, jak tego oczekiwaliśmy, powinieneś wykonać polecenia przedstawione poniżej.

1. Przejdź do okna skoroszytu i naciśnij przycisk *Wprowadź dane*. Zamiast tego możesz — oczywiście — kliknąć przycisk uruchamiający makro *GetData*, który umieściłeś na pasku narzędzi *Szybki dostęp*.

Na ekranie pojawi się okno dialogowe przedstawione na rysunku 18.2.



Rysunek 18.2.
Po uruchomieniu makra *GetData* na ekranie pojawi się okno dialogowe

2. W polu tekstowym wpisz dowolny tekst (na przykład imię).
3. Naciśnij przycisk **Zapisz** lub **Anuluj**.
Jak zauważysz, po naciśnięciu przycisku nic się nie dzieje, co jest poniekąd zrozumiałe, ponieważ nie utworzyłeś jeszcze dotąd żadnej procedury obsługi zdarzeń.
4. Aby zamknąć okno dialogowe, naciśnij czerwony przycisk ze znakiem x, znajdujący się w prawym, górnym rogu okna.

Dodawanie procedur obsługi zdarzeń

W tym podrozdziale wyjaśnię, jak napisać procedury obsługujące zdarzenia generowane po wyświetleniu formularza *UserForm*. Aby kontynuować przykład, wykonaj następujące kroki.

1. Uaktywnij edytor VBE, naciskając kombinację klawiszy **Alt+F11**.
Upewnij się, że formularz *UserForm* został wyświetlony.
2. Dwukrotnie kliknij lewym przyciskiem myszy przycisk **Anuluj**.
Wykonanie takiej operacji spowoduje uaktywnienie okna *Code* formularza *UserForm* i wstawienie w nim pustej procedury o nazwie *CancelButton_Click*.
3. Zmodyfikuj procedurę zgodnie ze wzorem przedstawionym poniżej.

```
Private Sub CancelButton_Click()
    Unload UserForm1
End Sub
```

Powyższa procedura jest wykonywana, kiedy użytkownik naciśnie przycisk *Anuluj*, i powoduje zamknięcie oraz usunięcie okna dialogowego z pamięci.

4. Aby ponownie wyświetlić formularz *UserForm1*, naciśnij kombinację klawiszy **Shift+F7**.
5. Dwukrotnie kliknij przycisk **Zapisz** i wprowadź poniższą procedurę.

```
Private Sub EnterButton_Click()
    Dim NextRow As Long

    ' Upewnij się, że Arkusz1 jest aktywny
    Sheets("Arkusz1").Activate

    ' Określ następny pusty wiersz
    NextRow = Application.WorksheetFunction.
        CountA(Range("A:A")) + 1

    ' Przenieś imię
    Cells(NextRow, 1) = TextName.Text

    ' Przenieś płeć
    If OptionMale Then Cells(NextRow, 2) = "Mężczyzna"
    If OptionFemale Then Cells(NextRow, 2) = "Kobieta"
    If OptionUnknown Then Cells(NextRow, 2) = "Obcy..."
```

```

    ' Wyczysć formanty przed wprowadzeniem kolejnych danych
    TextName.Text = ""
    OptionUnknown = True
    TextName.SetFocus
End Sub

```

6. Uaktywnij okno Excela i ponownie naciśnij przycisk *Wprowadź dane*, aby wyświetlić formularz *UserForm*.

Okno dialogowe działa tak, jak powinno. Na rysunku 18.3 przedstawiam wygląd naszego makra w działaniu. W pierwszym wierszu arkusza umieszczone zostały nagłówki kolumn, ale nie jest to konieczne.

A	B	C	D	E	F	G	H	I	J
1 Imię	Płeć								
2 Bartek	Mężczyzna								
3 Hanka	Kobieta								
4 Henryk	Mężczyzna								
5 Horacy	Mężczyzna								
6 Janusz	Mężczyzna								
7 Jerzy	Mężczyzna								
8 Joanna	Kobieta								
9 Julia	Kobieta								
10 Kasia	Kobieta								
11 Krzysiek	Mężczyzna								
12 Sonia	Kobieta								
13 Stefan	Mężczyzna								
14 Teresa	Kobieta								
15 Tomasz	Mężczyzna								
16 Jacek	Mężczyzna								
17 Ygrwook	Obcy...								
18									
19									
20									

Rysunek 18.3.
Zastosowanie
niestandardo-
wego okna
dialogowego do
wprowadzania
danych

Wyjaśnię teraz sposób działania procedury EnterButton_Click.

- ✓ Najpierw procedura zapewnia, że został uaktywniony właściwy arkusz (Arkusz1).
- ✓ Następnie procedura używa funkcji arkuszowej COUNTA (ILE.NIEPUSTYCH) Excela, za pomocą której identyfikuje kolejną pustą komórkę kolumny A.
- ✓ Teraz procedura przenosi tekst z formantu TextBox do kolumny A...
- ✓ ...po czym przy użyciu kilku instrukcji If określa, który formant OptionButton został zaznaczony, i w kolumnie B zapisuje odpowiedni ciąg znaków (Kobieta, Mężczyzna lub Obcy...).
- ✓ Wreszcie na koniec procedura resetuje formanty w celu przygotowania okna dialogowego do wprowadzenia kolejnych danych. Zwróć uwagę, że kliknięcie OK nie powoduje zamknięcia okna dialogowego, aby ułatwić użytkownikowi wprowadzanie kolejnych danych. By zakończyć wprowadzanie danych i usunąć z pamięci formularz UserForm, powinieneś nacisnąć przycisk Anuluj.

Sprawdzanie poprawności danych

Kiedy poeksperymentujesz trochę z naszym formularzem, przekonasz się, że mamy tutaj drobny problem. Otóż nasza procedura nie gwarantuje, że użytkownik faktycznie wprowadzi w polu tekstowym dane. Poniższy kod źródłowy został umieszczony w procedurze EnterButton_Click przed instrukcją przenoszącą tekst do arkusza. Dzięki temu uzyskujemy pewność, że użytkownik poda imię (lub wpisze jakikolwiek tekst) w polu TextBox. Jeżeli pole TextBox jest puste, pojawi się odpowiedni komunikat i procedura kończy działanie, aczkolwiek okno dialogowe nadal pozostaje wyświetcone na ekranie, dzięki czemu użytkownik może szybko skorygować swój błąd.

```
' Sprawdź, czy wprowadzono imię
If TextName.Text = "" Then
    MsgBox "Musisz podać imię!"
    TextName.SetFocus
    Exit Sub
End If
```

Teraz okno dialogowe działa tak, jak powinno!

Po wykonaniu wszystkich modyfikacji przekonasz się, że okno dialogowe działa bezbłędnie (nie zapomnij o przetestowaniu skrótów klawiszowych). W rzeczywistej aplikacji prawdopodobnie będziesz chciał gromadzić większą liczbę informacji, a nie tylko imię i pleć. Niezależnie jednak od ilości wprowadzanych czy, jak kto woli, pobieranych informacji, cały czas obowiązują takie same zasady, a zmienia się jedynie liczba formantów, które będziesz musiał obsługiwać.

Powinieneś zwrócić uwagę na jeszcze jedną rzecz. Jeżeli dane nie rozpoczynają się od wiersza numer 1 lub jeżeli w obszarze danych znajdują się wiersze puste, zliczanie wierszy dla zmiennej NextRow nie będzie działać poprawnie. Funkcja COUNTA zwraca liczbę zajętych komórek w kolumnie A przy założeniu, że powyżej ostatniej zajętej komórki nie ma żadnych komórek pustych. Poniżej przedstawiam inny sposób określania następnego pustego wiersza.

```
NextRow = Cells(Rows.Count, 1).End(xlUp).Row + 1
```

Polecenie symuluje aktywowanie ostatniej komórki w kolumnie A, naciśnięcie klawisza *End*, naciśnięcie strzałki w góra i następnie przejście o jeden wiersz w dół. Jeżeli wykonasz taką operację ręcznie, wskaźnik aktywnej komórki znajdzie się w następnym pustym wierszu kolumny A — działa to poprawnie nawet w sytuacji, kiedy obszar danych nie rozpoczyna się w wierszu 1 i zawiera puste wiersze.

Więcej przykładów formularzy UserForm

Bez większego trudu mógłbym wypełnić kolejne kilkaset stron ciekawymi i przydatnymi zagadnieniami związanymi z tworzeniem i wykorzystywaniem niestandardowych okien dialogowych, ale — niestety — książka ta ma swoje ograniczenia, więc siłą rzeczy ograniczę się do przedstawienia kilku przykładów.

Zastosowanie formantów *ListBox*

Formanty *ListBox* są bardzo użyteczne, ale praca z nimi może czasami być niezłyム wyzwaniem. Przed wyświetleniem okna dialogowego, które wykorzystuje formanty *ListBox*, powinieneś wypełnić listę odpowiednimi elementami. Następnie musisz sprawdzić, które elementy zostały wybrane przez użytkownika, i na tej podstawie podać odpowiednie działania.



Kiedy pracujesz z formantami *ListBox*, musisz pamiętać o następujących właściwościach i metodach.

- ✓ **AddItem** — metoda pozwala na dodawanie nowych elementów do listy.
- ✓ **ListCount** — właściwość, która zwraca liczbę elementów listy.
- ✓ **ListBox** — właściwość, która zwraca numer indeksu elementu listy wybranego przez użytkownika lub zaznacza element listy o podanym numerze indeksu (tylko w przypadku list z wybieraniem pojedynczych elementów). Wartość właściwości **ListBox** dla pierwszego elementu listy to 0 (a nie 1).
- ✓ **MultiSelect** — właściwość określa, czy użytkownik może wybierać (zaznaczać) więcej niż jeden element listy jednocześnie.
- ✓ **RemoveAllItems** — metoda, która powoduje usunięcie wszystkich elementów listy.
- ✓ **Selected** — właściwość, która zwraca tablicę elementów listy wybranych przez użytkownika (odnosi się tylko do list, na których można zaznaczać więcej niż jeden element).
- ✓ **Value** — właściwość, która zwraca element listy zaznaczony przez użytkownika.



Większość metod i właściwości, które odnoszą się do formantów *ListBox* będzie również działać z formantami *ComboBox*. Dzięki temu, kiedy poznasz zasady pracy z formantami *ListBox*, będziesz mógł przenieść tę wiedzę na pracę z polami kombi.

Wypełnianie listy

Aby uniknąć przypadkowych problemów, rozpoczęj ćwiczenie od utworzenia nowego, pustego skoroszytu. Dla wszystkich przykładów omawianych w kolejnych podrozdziałach przyjęłem następujące założenia.

- ✓ Utworzony został formularz *UserForm*.
- ✓ Na formularzu znajduje się pole listy (formant *ListBox*) o nazwie *ListBox1*.
- ✓ Na formularzu znajduje się przycisk polecenia (formant *CommandButton*) o nazwie *OKbutton*.
- ✓ Na formularzu znajduje się przycisk polecenia (formant *CommandButton*) o nazwie *CancelButton*, do którego utworzyłeś następującą procedurę obsługi zdarzeń.

```
Private Sub CancelButton Click()
    Unload UserForm1
End Sub
```

Poniżej przedstawiam procedurę obsługi zdarzenia Initialize, którą powinieneś utworzyć w oknie *Code* formularza *UserForm*.

1. **Zaznacz formularz *UserForm* i naciśnij klawisz F7, aby aktywować okno *Code*.**

Edytor VBE wyświetli na ekranie okno *Code* formularza *UserForm*, w którym będziesz mógł rozpoczęć tworzenie procedury obsługi zdarzenia Initialize.

2. **Rozwiń listę *Procedure*, znajdująjącą się w górnej części okna *Code* i wybierz z niej zdarzenie *Initialize*.**
3. **Wpisz procedurę, której kod przedstawiony został poniżej.**

```
Sub UserForm_Initialize()
    ' Wypełnianie listy
    With ListBox1
        .AddItem "Styczeń"
        .AddItem "Luty"
        .AddItem "Marzec"
        .AddItem "Kwiecień"
        .AddItem "Maj"
        .AddItem "Czerwiec"
        .AddItem "Lipiec"
        .AddItem "Sierpień"
        .AddItem "Wrzesień"
        .AddItem "Październik"
        .AddItem "Listopad"
        .AddItem "Grudzień"
    End With
    ' Zaznacz pierwszy element listy
    ListBox1.ListIndex = 0
End Sub
```

Jest to procedura inicjująca, która jest automatycznie wykonywana za każdym razem, kiedy formularz *UserForm* jest ładowany do pamięci. Dzięki temu, kiedy wywołujesz metodę *Show* obiektu *UserForm*, nasza procedura jest wykonywana i pole listy jest wypełniane 12 elementami, dodawanymi kolejno za pomocą metody *AddItem*.

4. **Wstaw do projektu VBA nowy moduł kodu i wpisz w nim kod procedury przedstawiony poniżej.**

```
Sub ShowList()
    UserForm1.Show
End Sub
```

Identyfikowanie wybranego elementu listy

Kod przedstawiony powyżej powoduje tylko wypełnienie listy elementami i wyświetlenie formularza na ekranie. Brakuje zatem procedury sprawdzającej, który element listy został zaznaczony przez użytkownika.

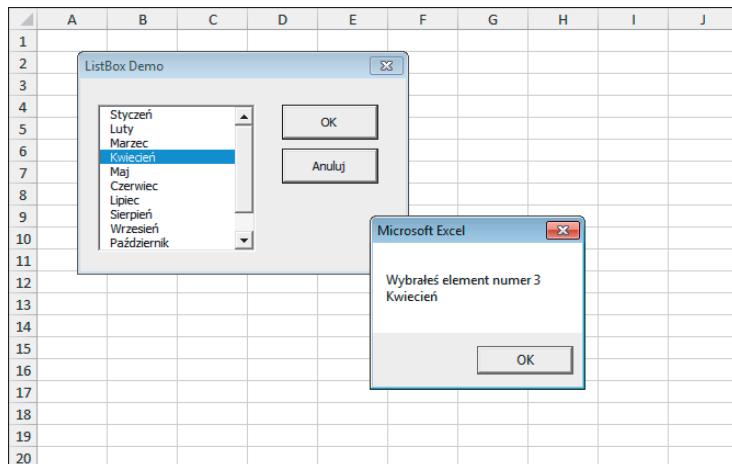
Utwórz procedurę obsługi zdarzenia `OKButton_Click`, której kod przedstawiam poniżej.

```
Private Sub OKButton_Click()
    Dim Msg As String
    Msg = "Wybrałeś element numer "
    Msg = Msg & ListBox1.ListIndex
    Msg = Msg & vbCrLf
    Msg = Msg & ListBox1.Value
    MsgBox Msg
    Unload UserForm1
End Sub
```

Powyższa procedura wyświetla na ekranie okno dialogowe, zawierające numer wybranego elementu oraz jego treść.

Jeżeli żaden element listy nie został wybrany, właściwość `ListIndex` zwraca wartość -1. W naszym przypadku jednak nigdy tak się nie stanie, ponieważ procedura inicjująca formularz `UserForm` domyślnie zaznacza pierwszy element listy, a usunięcie tego zaznaczenia bez zaznaczenia innego elementu po prostu nie jest możliwe. Z tego powodu w naszym przypadku *zawsze* będzie zaznaczony jakiś element, nawet jeżeli użytkownik nie zrobi tego wprost.

Na rysunku 18.4 przedstawiam wygląd omawianego formularza.



Rysunek 18.4.
Sprawdzanie,
który element
listy został za-
znaczony

Właściwość `ListIndex` pierwszego elementu listy ma zawsze wartość 0, a nie 1 (jak mógłbyś tego oczekiwąć). Dzieje się tak zawsze, nawet w sytuacji, kiedy w kodzie VBA użyjesz dyrektywy `Option Base 1`, zmieniającej początkowy numer indeksowania tablic na 1.

Skoroszyt z tym przykładem możesz pobrać ze strony internetowej tej książki.



Identyfikowanie wielu zaznaczonych elementów listy

Jeżeli Twoja lista jest skonfigurowana tak, że użytkownik może zaznaczać więcej niż jeden element listy, przekonasz się, że właściwość `ListIndex` zwraca numer tylko *ostatniego* zaznaczonego elementu. Aby zidentyfikować wszystkie zaznaczone elementy, powinieneś użyć właściwości `Selected`, która zwraca tablicę zawierającą zaznaczone elementy listy.



Aby umożliwić zaznaczanie wielu elementów listy, musisz ustawić właściwość `MultiSelect` na wartość 1 lub 2. Możesz to wykonać na etapie projektowania formularza w oknie `Properties` lub w czasie działania programu poprzez wykonanie polecenia takiego, jak przedstawione poniżej.

```
UserForm1.ListBox1.MultiSelect = 1
```

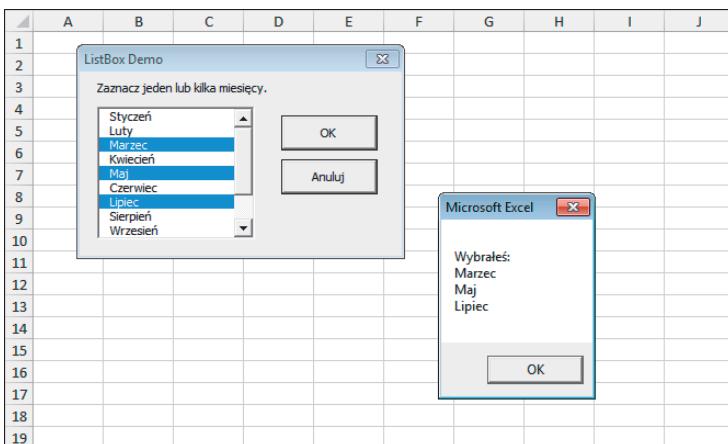
Właściwość `MultiSelect` może przyjąć jedną z trzech wartości, których znaczenie zostało przedstawione w tabeli 18.1.

Tabela 18.1. Dozwolone ustawienia właściwości MultiSelect

Wartość	Stała VBA	Znaczenie
0	<code>fmMultiSelectSingle</code>	Pozwala na zaznaczanie tylko pojedynczych elementów listy.
1	<code>fmMultiSelectMulti</code>	Klikanie elementu listy lub naciśnięcie klawisza spacji kolejno zaznacza i usuwa zaznaczenie elementu listy.
2	<code>fmMultiSelectExtended</code>	Kolejne elementy listy można zaznaczać (bądź usuwać ich zaznaczenie) poprzez przytrzymywanie wciśniętego klawisza <code>Shift</code> lub <code>Ctrl</code> podczas klikania elementów listy.

Procedura przedstawiona poniżej wyświetla na ekranie okno dialogowe zawierające zestawienie wszystkich elementów, które zostały zaznaczone na liście. Na rysunku 18.5 przedstawiam wygląd takiego okna dialogowego.

```
Private Sub OKButton Click()
    Dim Msg As String
    Dim i As Integer
    Dim Counter As Integer
    Msg = "Wybrałeś:" & vbCrLf
    For i = 0 To ListBox1.ListCount - 1
        If ListBox1.Selected(i) Then
            Counter = Counter + 1
            Msg = Msg & ListBox1.List(i) & vbCrLf
        End If
    Next i
    If Counter = 0 Then Msg = Msg & "(brak zaznaczonych elementów)"
    MsgBox Msg
    Unload UserForm1
End Sub
```



Rysunek 18.5.
Identyfikowanie
wielu zaznaczo-
nych elemen-
tów listy

Powyższa procedura wykorzystuje pętlę For-Next do przechodzenia kolejno przez wszystkie elementy pola listy. Zwróć uwagę, że pętla rozpoczyna działanie od elementu 0 (pierwszy element listy) i kończy działanie na ostatnim elemencie (który jest określany poprzez wartość właściwości ListCount pomniejszoną o 1). Jeżeli właściwość Selected bieżącego elementu ma wartość *True*, to znaczy, że element został zaznaczony. Procedura wykorzystuje również zmienną Counter do śledzenia liczby zaznaczonych elementów. Jeżeli żaden element nie został zaznaczony, konstrukcja If-Then odpowiednio zmienia komunikat wyświetlany na ekranie.

Skoroszyt z tym przykładem możesz pobrać ze strony internetowej tej książki.



Zaznaczanie zakresów

W niektórych przypadkach będziesz chciał, aby użytkownik zaznaczył w oknie dialogowym wybrany zakres komórek. Przykład zastosowania takiego mechanizmu znajdziesz w oknie dialogowym *Tworzenie tabeli*, które jest wyświetlane na ekranie po wybraniu ze Wstążki Excela polecenia *WSTAWIANIE/Tabele/Tabela*. W oknie dialogowym *Tworzenie tabeli* znajduje się pole selektora zakresów, w którym Excel automatycznie podpowiada przewidywany, jego zdaniem, zakres komórek, jakie chcesz zamienić na tabelę. Jeżeli przewidywany zakres nie jest prawidłowy, możesz do jego zmiany użyć selektora, zaznaczając komórki bezpośrednio w arkuszu.

Aby skorzystać z takiego mechanizmu w swoim oknie dialogowym, powinieneś umieścić w nim formant *RefEdit*. W przykładzie przedstawionym poniżej na ekranie wyświetlane jest okno dialogowe, w którym adres aktualnie zaznaczonego zakresu komórek jest zapisany w polu selektora adresów (zobacz rysunek 18.6). Zaznaczony zakres komórek składa się z bloku niepustych komórek, zawierającego aktywną komórkę. Użytkownik może zaakceptować proponowany zakres komórek lub go zmienić. Kiedy użytkownik naciśnie przycisk *OK*, procedura wyróżnia zaznaczony zakres komórek pogrubioną czcionką.

A	B	C	D	E	F
1	17	8	10	75	11
2	63	8	22	27	11
3	5	45	70	13	6
4	30	22	15	1	3
5	73	56	87	70	24
6	86	93	69	95	49
7	31	37	30	70	26
8	31	68	1	62	63
9	72	97	58	21	42
10	22	28	92	32	76
11	15	90	55	72	22
12	75	14	21	98	18
13	82	79	43	96	50
14	89	38	92	69	32
15	1	38	93	78	58
16	37	42	11	63	0
17	40	59	14	57	2
18	12	1	14	67	74
19	15	10	25	27	70
20	61	41	84	47	29
21	51	78	21	89	16
22	74	5	31	11	7
23					55
24					

Rysunek 18.6.
Okno dialogowe pozwala na zaznaczenie wybranego zakresu komórek

W tym przykładzie przyjąłem następujące założenia.

- ✓ Utworzony został formularz *UserForm* o nazwie *UserForm1*.
- ✓ Na formularzu znajduje się formant *CommandButton* o nazwie *OKButton*.
- ✓ Na formularzu znajduje się formant *CommandButton* o nazwie *CancelButton*.
- ✓ Na formularzu znajduje się formant *RefEdit* o nazwie *RefEdit1*.

Kod przedstawiony poniżej powinien zostać zapisany w module kodu VBA. Procedura *BoldCells* wykonuje dwa zadania: inicjuje okno dialogowe poprzez przypisanie adresu aktualnie zaznaczonego zakresu komórek do pola selektora zakresów (formant *RefEdit*) i wyświetla formularz *UserForm* na ekranie.

```
Sub BoldCells()
    ' Jeżeli arkusz nie jest aktywny, zakończ działanie
    If TypeName(ActiveSheet) <> "Worksheet" Then Exit Sub

    ' Zaznacz bieżący zakres komórek
    ActiveCell.CurrentRegion.Select

    ' Inicjuj formant RefEdit
    UserForm1.RefEdit1.Text = Selection.Address

    ' Pokaż okno dialogowe
    UserForm1.Show
End Sub
```

Procedura przedstawiona poniżej jest wykonywana po naciśnięciu przycisku *OK*. W kodzie zaimplementowany został bardzo prosty mechanizm kontroli błędów sprawdzający, czy zakres komórek przypisany do formantu *RefEdit* jest poprawny.

```

Private Sub OKButton_Click()
On Error GoTo BadRange
Range(RefEdit1.Text).Font.Bold = True
Unload UserForm1
Exit Sub
BadRange:
MsgBox "Zaznaczyłeś nieprawidłowy zakres."
End Sub

```

Jeżeli po wywołaniu procedury wystąpi błąd (co najprawdopodobniej będzie skutkiem zaznaczenia nieprawidłowego zakresu dla formantu RefEdit), sterowanie zostanie przekazane do etykiety BadRange i na ekranie pojawi się okno dialogowe z odpowiednim komunikatem. Okno formularza *UserForm* przez cały czas pozostaje otwarte, dzięki czemu użytkownik może zaznaczyć inny zakres komórek.



Jeżeli pobieranie adresu zakresu komórek jest *jedyną* funkcją Twojego formularza *UserForm*, możesz ułatwić sobie życie i zamiast formularza użyć metody *InputBox* obiektu *Application*. Więcej szczegółowych informacji na ten temat znajdziesz w rozdziale 15.

Praca z wieloma grupami formantów OptionButton

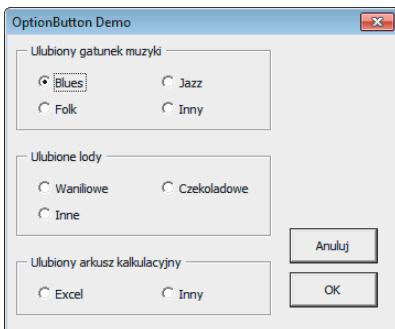
Na rysunku 18.7 przedstawiam niestandardowe okno dialogowe zawierające trzy grupy formantów *OptionButton*. Jeżeli na Twoim formularzu *UserForm* znajduje się więcej niż jeden zestaw przycisków opcji, upewnij się, że każdy z nich działa jako osobna grupa. Możesz to zrobić na dwa sposoby.

- ✓ Umieść każdy z zestawów przycisków opcji w osobnym kontenerze *Frame*. Takie rozwiązanie jest najłatwiejsze i powoduje, że okno dialogowe wydaje się dobrze zorganizowane. Zazwyczaj najpierw powinieneś utworzyć kontener *Frame* i dopiero potem umieszczać w nim kolejne przyciski opcji, aczkolwiek nic nie stoi na przeszkodzie, aby przeciągnąć do kontenera *Frame* przycisk, który już wcześniej opracowałeś.
- ✓ Upewnij się, że każdy zestaw przycisków opcji ma ustawioną unikatową nazwę we właściwości *GroupName* (możesz to zrobić za pomocą okna dialogowego *Properties*). Jeżeli wszystkie przyciski opcji z danego zestawu znajdują się w jednym kontenerze *Frame*, nie musisz się przejmować wartością właściwości *GroupName* poszczególnych przycisków.



Tylko jeden przycisk w danej grupie przycisków opcji może być zaznaczony. Aby ustawić opcję zaznaczoną domyślnie, powinieneś ustawić wartość właściwości wybranego przycisku opcji na *True*. Możesz to zrobić na etapie projektowania lub za pomocą kodu VBA przedstawionego poniżej.

```
UserForm1.OptionButton1.Value = True
```



Rysunek 18.7.
W tym oknie dialogowym znajdują się trzy grupy przycisków opcji

Skoroszyt z tym przykładem znajdziesz na stronie internetowej tej książki. W skoroszycie zamieszczony został również dodatkowy kod, który powoduje, że po naciśnięciu przycisku OK na ekranie pojawia się okno dialogowe zawierające listę wybranych opcji.

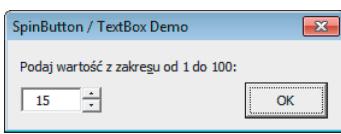


Zastosowanie formantów SpinButton oraz TextBox

Formant SpinButton (pokrętło) pozwala użytkownikowi na wybranie żąanej wartości za pomocą klikania przycisków ze strzałkami. Sam formant składa się wyłącznie z przycisków (bez żadnego tekstu), a zazwyczaj będziesz chciał, aby wybierana wartość była w jakiś sposób wyświetlana na ekranie. Jednym z możliwych rozwiązań jest zastosowanie formantu Label, jednak metoda ta ma poważną wadę — użytkownik nie może w etykietach wpisywać tekstu. Z tego powodu znacznie lepszym rozwiązaniem jest użycie do tego pola tekstowego (formant TextBox).

Formanty SpinButton oraz TextBox tworzą dosyć naturalną parę, która jest bardzo często wykorzystywana w Excelu. Aby się o tym przekonać, możesz przywołać na ekran okno dialogowe *Ustawienia strony*, gdzie znajdziesz co najmniej kilka przykładów zastosowania takich par formantów. W idealnym scenariuszu formanty SpinButton oraz TextBox są zawsze zsynchronizowane — jeżeli użytkownik kliką przycisk formantu SpinButton, jego aktualna wartość natychmiast pojawia się w polu tekstowym. Z drugiej strony, jeżeli użytkownik wpisze nową wartość w polu tekstowym, formant SpinButton powinien być od razu ustawiany na taką wartość. Na rysunku 18.8 przedstawiam niestandardowe okno dialogowe, wykorzystujące parę formantów SpinButton i TextBox.

Rysunek 18.8.
Okno dialogowe z formantami SpinButton i TextBox



W tym oknie dialogowym umieszczone zostały następujące formanty.

- ✓ Formant SpinButton o nazwie SpinButton1, którego właściwość Min została ustawiona na wartość 1, a właściwość Max na wartość 100.
- ✓ Pole tekstowe o nazwie TextBox1, umieszczone po lewej stronie formantu SpinButton.
- ✓ Formant CommandButton o nazwie OKButton.

Procedura obsługi zdarzenia Change dla formantu SpinButton została przedstawiona poniżej. Zdarzenie Change jest generowane za każdym razem, kiedy zmienia się wartość formantu SpinButton. Procedura obsługi tego zdarzenia przypisuje wartość formantu SpinButton do pola tekstowego. Aby ją utworzyć, dwukrotnie kliknij formant SpinButton lewym przyciskiem myszy, co spowoduje wyświetlenie na ekranie okna *Code* formularza *UserForm*, a następnie wpisz kod procedury.

```
Private Sub SpinButton1_Change()
    TextBox1.Text = SpinButton1.Value
End Sub
```

Procedura obsługi zdarzenia Change dla pola tekstowego, która została przedstawiona niżej, jest nieco bardziej złożona. Aby ją utworzyć, dwukrotnie kliknij formant TextBox lewym przyciskiem myszy, co spowoduje wyświetlenie na ekranie okna *Code* formularza *UserForm*, a następnie wpisz kod. Procedura jest wykonywana za każdym razem, kiedy użytkownik wpisze nową wartość w polu tekstowym.

```
Private Sub TextBox1_Change()
    Dim NewVal As Integer
    NewVal = Val(TextBox1.Text)
    If NewVal >= SpinButton1.Min And _
        NewVal <= SpinButton1.Max Then
        SpinButton1.Value = NewVal
    End Sub
```

Procedura wykorzystuje zmienną, która przechowuje tekst wpisany przez użytkownika w polu tekstowym (i zamieniony na wartość numeryczną za pomocą funkcji *Val*). Następnie procedura sprawdza, czy podana przez użytkownika wartość jest prawidłowa (inaczej mówiąc, czy znajduje się w dozwolonym zakresie). Jeżeli tak, wartość formantu SpinButton jest ustawiana na wartość wpisaną w polu tekstowym. Dzięki takiemu rozwiązaniu wartość formantu SpinButton jest zawsze równa wartości wpisanej przez użytkownika (jeśli założymy, że podana wartość mieści się w zakresie poprawnych wartości).

Jeżeli w edytorze VBE użyjesz klawisza *F8* do wykonania procedury *TextBox1_Change* programu, krok po kroku, zauważysz, że kiedy wykonywany jest wiersz kodu *SpinButton1.Value = NewVal*, natychmiast generowane jest zdarzenie Change formantu SpinButton i w efekcie procedura obsługi zdarzenia *SpinButton1_Change* ustawia odpowiednią wartość w polu tekstowym. Na szczećcie, nie powoduje to wygenerowania zdarzenia Change dla formantu TextBox, ponieważ procedura *SpinButtonChange* w tym momencie nie zmienia wartości pola tekstowego. Z pewnością możesz sobie jednak wyobrazić, że takie zjawisko może wywołać dosyć nieoczekiwane efekty w formularzu *UserForm*. Skomplikowane? Nie tak bardzo, a w praktyce powinieneś po prostu zawsze pamiętać, że jeżeli kod Twojego programu zmienia wartość właściwości *Value* danego formantu, automatycznie taka zmiana powoduje wygenerowanie zdarzenia Change dla tego formantu.

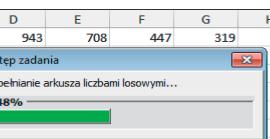




Skoroszyt z tym przykładem znajdziesz na stronie internetowej tej książki. Oprócz opisywanych wyżej procedur, znajdziesz tam parę innych fragmentów kodu, które mogą się przydać podczas tworzenia swoich makr.

Wykorzystywanie formularza *UserForm* jako wskaźnika postępu zadania

Jeżeli masz jakieś makro, którego wykonanie zajmuje dużo czasu, dobrym rozwiązaniem może być wyświetlanie na ekranie wskaźnika postępu jego realizacji, dzięki któremu użytkownicy będą wiedzieli, że makro nadal działa i Excel się nie zawiesił. Do utworzenia atrakcyjnego wskaźnika postępu zadania możesz użyć formularza *UserForm*, tak jak to zostało przedstawione na rysunku 18.9. Takie zastosowanie okna dialogowego wymaga jednak użycia kilku sztuczek, o których napiszę za chwilę.



A	B	C	D	E	F	G	H	I	J
1	223	966	320	943	708	447	319	878	360
2	349	178	16	40				899	390
3	439	379	40					927	230
4	32	761	2					485	376
5	68	319	60					576	878
6	713	365	76					577	671
7	440	295	894	40	457	578	288	376	151
8	355	602	68	488	460	781	295	765	4
9	776	144	274	621	51	286	192	81	284
10	62	658	863	706	495	112	493	94	744
11	689	529	246	817	629	559	854	150	407
12	579	801	853	980	439	483	276	560	402
13	680	445	332	561	888	803	556	242	37
14	791	874	5	909	997	276	994	612	129
15	643	711	570	242	168	504	342	722	582
16	224	800	52	793	765	204	12	654	215
17	361	476	12	199	945	723	524	157	842
18	545	79	993	138	731	805	212	541	975

Rysunek 18.9.

Formularz *UserForm* spełniający rolę wskaźnika postępu zadania dla czasochłonnego makra

Tworzenie okna dialogowego dla wskaźnika postępu

Pierwszym etapem budowania wskaźnika postępu zadania będzie utworzenie okna dialogowego. W naszym przykładzie okno dialogowe ma wyświetlać postęp działania makra, które zapisuje losowo wybrane liczby w zakresie komórek obejmującym 100 kolumn i 1000 wierszy aktywnego skoroszytu. Aby utworzyć okno dialogowe, wykonaj polecenia opisane poniżej.

- Przejdź do okna edytora VBE i wstaw nowy formularz *UserForm*.
- Ustaw właściwość *Caption* formularza na wartość *Postęp zadania*.
- Dodaj formant *Frame* i ustaw dla niego następujące właściwości.

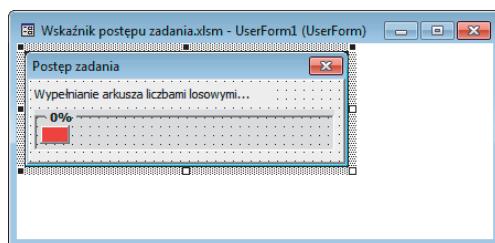
<i>Właściwość</i>	<i>Zmień na:</i>
<i>Caption</i>	0%
<i>Name</i>	<i>FrameProgress</i>
<i>SpecialEffect</i>	2 — <i>fmSpecialEffectSunken</i>
<i>Width</i>	204
<i>Height</i>	28

4. Dodaj formant Label wewnątrz kontenera Frame i ustaw dla niego następujące właściwości.

<i>Właściwość</i>	<i>Zmień na:</i>
Name	<i>LabelProgress</i>
BackColor	<i>&H000000FF&</i> (czerwony)
Caption	(brak)
SpecialEffect	<i>1 — fmSpecialEffectRaised</i>
Width	<i>20</i>
Height	<i>13</i>
Top	<i>5</i>
Left	<i>2</i>

5. Dodaj kolejny formant Label powyżej ramki i zmień jego właściwość Caption na *Wypełnianie arkusza liczbami losowymi...*.

Formularz *UserForm* powinien teraz wyglądać tak, jak na rysunku 18.10.



Rysunek 18.10.
Okno dialogowe
wskaźnika po-
stępu zadania

Procedury

W tym przykładzie wykorzystałem dwie procedury oraz jedną zmienną modułową.

- ✓ **Zmienna modułowa** jest zdefiniowana w module kodu VBA i przechowuje kopię formularza *UserForm*.

```
Dim ProgressIndicator as UserForm1
```

- ✓ **Procedura** *EnterRandomNumbers* wykonuje całe zadanie i jest wykonywana po wyświetleniu formularza na ekranie. Zwróć uwagę, że wywołuje ona procedurę *UpdateProgress*, która aktualizuje wygląd wskaźnika postępu zadania w oknie dialogowym.

```
Sub EnterRandomNumbers ()
    ' Wstawia liczby losowe do komórek aktywnego arkusza
    Dim Counter As Long
    Dim RowMax As Long, ColMax As Long
    Dim r As Long, c As Long
    Dim PctDone As Single
    ' Utwórz kopię formularza i przechowaj w zmiennej
```

```
Set ProgressIndicator = New UserForm1
' Pokaż niemodalne okno dialogowe wskaźnika postępu
ProgressIndicator.Show vbModeless
If TypeName(ActiveSheet) <> "Worksheet" Then
    Unload ProgressIndicator
    Exit Sub
End If
' Wpisywanie liczb losowych
Cells.Clear
Counter = 1
RowMax = 1000
ColMax = 100
For r = 1 To RowMax
    For c = 1 To ColMax
        Cells(r, c) = Int(Rnd * 1000)
        Counter = Counter + 1
    Next c
    PctDone = Counter / (RowMax * ColMax)
    Call UpdateProgress(PctDone)
    Next r
    Unload ProgressIndicator
    Set ProgressIndicator = Nothing
End Sub
```

✓ **Procedura** `UpdateProgress` pobiera jeden argument wywołania i aktualizuje wygląd wskaźnika postępu w oknie dialogowym.

```
Sub UpdateProgress(pct)
    With ProgressIndicator
        .FrameProgress.Caption = Format(pct, "0%")
        .LabelProgress.Width = pct * (.FrameProgress
            .Width - 10)
    End With
    ' Polecenie DoEvents jest odpowiedzialne za aktualizację zawartości formularza
    DoEvents
End Sub
```

Jak to działa?

Po uruchomieniu procedury `EnterRandomNumbers` ładuje kopię formularza `UserForm` do zmiennej modułowej o nazwie `ProgressIndicator`. Następnie procedura wyświetla na ekranie formularz `UserForm` w trybie niemodalnym (dzięki czemu kod procedury nadal może kontynuować działanie).

Procedura `EnterRandomNumbers` sprawdza typ aktywnego arkusza. Jeżeli nie jest to arkusz roboczy, zawierający komórki (a na przykład arkusz wykresu), okno formularza `UserForm` zostaje zamknięte i procedura kończy działanie bez podejmowania żadnej innej akcji. Jeżeli aktywny arkusz jest normalnym arkuszem roboczym, procedura wykonuje następujące zadania.

1. **Usuwa zawartość wszystkich komórek aktywnego arkusza.**
2. **Przechodzi w pętli kolejne wiersze i kolumny arkusza (określone przez zmienne `RowMax` i `ColMax`) i wstawia do kolejnych komórek losowe liczby.**

3. Inkrementuje wartość zmiennej *Counter* i oblicza procentowy postęp realizacji całego zadania (wynik obliczeń jest zapisywany w zmiennej *PctDone*).
4. Wywołuje procedurę *UpdateProgress*, która wyświetla postęp zadania poprzez zmianę szerokości formantu *LabelProgress* i aktualizację etykiety formantu *Frame*.
5. Po zakończeniu wypełniania arkusza procedura zamyka okno dialogowe wskaźnika postępu.

Wykorzystywanie wskaźnika postępu powoduje pewne niewielkie zmniejszenie szybkości działania całego makra, związane z koniecznością wykonywania dodatkowego kodu wyświetlającego okno dialogowe i aktualizującego wskaźnik postępu działania. Jeżeli szybkość działania makra jest czynnikiem krytycznym, powinieneś rozważyć decyzję o zastosowaniu wskaźnika postępu.



Jeżeli chcesz zaadaptować opisaną powyżej technikę do własnych potrzeb, musisz zastanowić się, jak obliczać procentowy postęp działania makra, co — oczywiście — będzie zależało od tego, jakie zadania Twoje makro realizuje. Aby całość działała poprawnie, powinieneś w regularnych odstępach czasu obliczać procentowy postęp działania makra i wywoływać procedurę *UpdateProgress*, aktualizującą wygląd wskaźnika postępu.

Skoroszyt z tym przykładem znajdziesz na stronie internetowej tej książki.

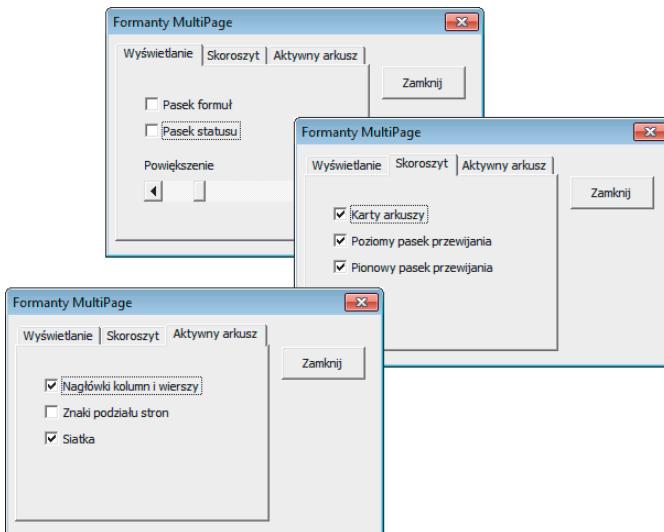
Tworzenie niemodalnych okien dialogowych z wieloma kartami

Wielokartowe okna dialogowe są bardzo użyteczne, ponieważ pozwalają na prezentowanie opcji i innych informacji w postaci mniejszych, bardziej zorganizowanych fragmentów. Okno dialogowe *Formatowanie komórek* Excela (które pojawia się na ekranie po kliknięciu komórki prawym przyciskiem myszy i wybraniu z menu podręcznego polecenia *Formatuj komórki*) jest dobrym przykładem takiego rozwiązania. Okno dialogowe, które opiszę w tym punkcie, jest podzielone na trzy karty zawierające opcje odpowiedzialne za wyświetlanie różnych elementów interfejsu Excela.

Tworzenie wielokartowych okien dialogowych jest relatywnie prostym zadaniem dzięki zastosowaniu formantu *MultiPage*. Na rysunku 18.11 przedstawiam wygląd okna dialogowego, wykorzystującego formant *MultiPage* składający się z trzech *kart*. Kiedy użytkownik kliknie przycisk kolejnej karty, zostaje ona aktywowana i w oknie wyświetlane są tylko elementy umieszczone na aktywnej karcie.

Zwróć uwagę na fakt, że jest to niemodalne okno dialogowe. Innymi słowy, kiedy okno jest wyświetcone na ekranie, użytkownik może nadal pracować z arkuszem. Wybranie każdej z dostępnych opcji ma natychmiastowy efekt, stąd nie ma potrzeby stosowania przycisku *OK*. Poniżej przedstawiam procedurę, której zadaniem jest wyświetlanie niemodalnego okna formularza *UserForm*.

```
Sub ShowDialog()
    UserForm1.Show vbModeless
End Sub
```



Rysunek 18.11.
Trzy karty formantu MultiPage

Niemodalne formularze *UserForm* w Excelu 2013 działają nieco inaczej niż w poprzednich wersjach. Dzieje się tak ze względu na nowy, jednodokumentowy interfejs Excela (gdzie każdy ze skoroszytów jest wyświetlany w swoim własnym, osobnym oknie). W poprzednich wersjach Excela niemodalne okna dialogowe były wyświetlane na ekranie niezależnie od tego, który skoroszyt był w danej chwili aktywny. W Excelu 2013 niemodalne okno dialogowe jest pokazywane tylko w skoroszycie, który był aktywny w momencie wyświetlenia okna na ekranie.



Podczas pracy z formantami MultiPage powinieneś pamiętać o następujących rzeczach.

- ✓ Używaj tylko jednego formantu MultiPage w każdym oknie dialogowym.
- ✓ Upewnij się, że korzystasz z formantu MultiPage, a nie TabStrip. Formanty TabStrip są znacznie bardziej trudniejsze w użyciu.
- ✓ Jeżeli chcesz, aby wybrane formanty (takie jak np. przyciski OK, Anuluj czy Zamknij) były widoczne przez cały czas, powinieneś umieścić je na zewnątrz kontenera MultiPage.
- ✓ Podczas projektowania kliknij kartę formantu MultiPage prawym przyciskiem myszy, aby zobaczyć menu podręczne z poleceniami pozwalającymi dodawać, usuwać, przesuwać karty lub zmieniać ich nazwy.
- ✓ Aby aktywować daną kartę podczas projektowania, po prostu kliknij ją lewym przyciskiem myszy. Potem możesz dodawać do niej formanty w normalny sposób.
- ✓ Aby zaznaczyć sam kontener MultiPage (a nie jeden z umieszczonych w nim formantów), kliknij jego obramowanie. Zwróć uwagę na zawartość okna Properties, w którym wyświetlana jest nazwa i typ zaznaczonego formantu. Formant MultiPage możesz również zaznaczyć, wybierając jego nazwę z listy rozwijanej, widocznej w górnej części okna Properties.
- ✓ Wygląd formantu MultiPage możesz zmieniać poprzez zmianę jego właściwości Style oraz TabOrientation.



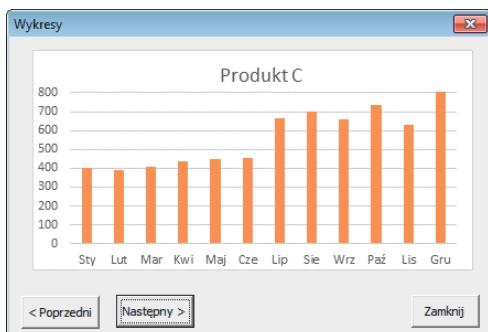
- ✓ Wartość właściwości Value formantu Multipage reprezentuje kartę, która jest aktualnie wyświetlana, jeśli na przykład kod Twojego makra ustawi właściwość Value formantu Multipage na wartość 0, na ekranie zostanie wyświetlona pierwsza karta tego formantu.

Skoroszyt z tym przykładem znajdziesz na stronie internetowej tej książki.

Wyświetlanie wykresów na formularzach UserForm

Jeżeli z takiego czy innego powodu będziesz chciał wyświetlić na formularzu *UserForm* wykres, przekonasz się, że Excel nie oferuje żadnego prostego sposobu na realizację takiego zadania. W tej sytuacji musisz po prostu być kreatywny. W tym punkcie przedstawię technikę, która pozwala na wyświetlanie jednego lub nawet kilku wykresów na formularzu *UserForm*.

Na rysunku 18.12 przedstawiam przykład okna dialogowego, w którym wyświetlony został jeden z trzech wykresów. Na formularzu *UserForm* umieszczony został formant Image. Cała sztuczka polega na tym, aby użyć kodu VBA do zapisania wykresu w postaci pliku GIF i następnie przypisania tego pliku do właściwości Picture formantu Image (co spowoduje załadowanie tego pliku z dysku). Przyciski *Poprzedni* i *Następny* umożliwiają wyświetlenie pozostałych wykresów.



Rysunek 18.12.
Wyświetlanie wykresów na formularzu UserForm



W tym przykładzie, który znajdziesz również na stronie internetowej tej książki, na arkuszu o nazwie *Wykresy* utworzone zostały trzy proste wykresy. Przyciski *Poprzedni* i *Następny* pozwalają na wybór wykresu, który będzie wyświetlany; numer wybranego wykresu jest przechowywany w publicznej zmiennej *ChartNum*, która jest dostępna z poziomu wszystkich procedur. Zadanie wyświetlania wybranego wykresu w oknie dialogowym jest realizowane przez procedurę *UpdateChart*.

```
Private Sub UpdateChart()
    Dim CurrentChart As Chart
    Dim Fname As String

    Set CurrentChart = _
```

```

Sheets("Wykresy").ChartObjects(ChartNum).Chart
CurrentChart.Parent.Width = 300
CurrentChart.Parent.Height = 150

' Zapisz wykres jako plik GIF
Fname = ThisWorkbook.Path & "\temp.gif"
CurrentChart.Export FileName:=Fname, FilterName:="GIF"

' Pokaż wykres
Image1.Picture = LoadPicture(Fname)
End Sub

```

Procedura ustawia nazwę dla zapisywanej wykresu i następnie wykorzystuje metodę Export do zapisania wykresu w postaci pliku GIF na dysku. Po wyeksportowaniu wykresu procedura stosuje funkcję LoadPicture do ustawienia wartości właściwości Picture obiektu Image.

Lista kontrolna tworzenia i testowania okien dialogowych

Na zakończenie tego rozdziału chciałbym zaprezentować listę kontrolną, której możesz używać podczas tworzenia i testowania własnych okien dialogowych.

- ✓ Czy poszczególne formanty są odpowiednio wyrównane względem siebie?
- ✓ Czy podobne formanty są jednakowej wielkości?
- ✓ Czy odległości między formantami są równe?
- ✓ Czy okno dialogowe posiada odpowiedni tytuł?
- ✓ Czy zawartość okna dialogowego nie jest zbyt przytłaczająca? Jeżeli tak, przy użyciu formantu MultiPage można pogrupować formanty na poszczególnych kartach.
- ✓ Czy każdy formant może zostać uaktywniony przy użyciu skrótu klawiszowego?
- ✓ Czy skróty klawiszowe się nie powtarzają?
- ✓ Czy poszczególne formanty są logicznie pogrupowane (według funkcji)?
- ✓ Czy kolejność tabulacji formantów jest prawidłowa? Użytkownik powinien mieć możliwość przechodzenia kolejno przez wszystkie formanty danego okna dialogowego.
- ✓ Jeżeli planujesz korzystać z tego okna dialogowego w dodatku *add-in*, czy sprawdziłeś poprawność funkcjonowania okna po zapisaniu skoroszytu w postaci dodatku?
- ✓ Czy po wciśnięciu przez użytkownika klawisza *Esc* lub kliknięciu przycisku *Zamknij* formularza *UserForm* kod źródłowy języka VBA wykonuje odpowiednie operacje?
- ✓ Czy w etykietach i innych elementach okna dialogowego nie popełniłeś literówek bądź innych błędów? Niestety, mechanizm sprawdzania poprawności pisowni Excela nie działa dla formularzy *UserForm*, stąd podczas projektowania okien dialogowych jesteś zdany tylko na własne siły.

- ✓ Czy okno dialogowe zostanie poprawnie wyświetlone w niskiej rozdzielczości ekranu (na przykład 1024×768 pikseli)? Pamiętaj, że jeżeli projektujesz formularz, korzystając z wysokiej rozdzielczości ekranu, może się okazać, że Twoje okno dialogowe jest za duże, aby mogło się zmieścić na ekranie w niższej rozdzielczości.
- ✓ Czy wszystkie pola tekstowe (formanty TextBox) zostały wyposażone w odpowiedni mechanizm sprawdzania poprawności danych wpisywanych przez użytkownika? Jeżeli masz zamiar skorzystać z właściwości WordWrap, czy na pewno ustawileś właściwość MultiLine na wartość *True*?
- ✓ Czy wszystkie formanty ScrollBar i SpinButton umożliwiają ustawienie tylko poprawnych wartości?
- ✓ Czy wszystkie formanty ListBox mają poprawnie ustawioną właściwość MultiSelect?

Najlepszym sposobem na opanowanie sztuki tworzenia własnych okien dialogowych jest po prostu budowanie własnych okien dialogowych. Rozpocznij od pustego okna dialogowego i zacznij eksperymentować z różnymi formantami i ich właściwościami. Nie zapominaj o pomocy systemowej — to najlepsze źródło szczegółowych informacji na temat poszczególnych formantów i ich tajemnic.

Rozdział 19

Udostępnianie makr z poziomu interfejsu użytkownika

W tym rozdziale:

- ▶ zobaczysz, jak za pomocą XML można dostosowywać Wstążkę do własnych potrzeb,
- ▶ nauczysz się dodawać nowe elementy do menu podręcznego,
- ▶ dowiesz się, jak dodawać nowe przyciski poleceń do paska narzędzi *Szybki dostęp*,
- ▶ poznasz sposoby ręcznego dostosowywania Wstążki do własnych potrzeb.

Przed pojawiением się na rynku pakietu Office 2007 Wstążki po prostu nie było i wszyscy użytkownicy byli przyzwyczajeni do korzystania z klasycznych menu i pasków narzędzi. Od wersji 2007 Wstążka jest nowym, powszechnie używanym graficznym interfejsem użytkownika pakietu Microsoft Office.

Mając to na uwadze, mógłbyś oczekiwać, że za pomocą języka VBA będziesz mógł modyfikować zawartość Wstążki i dostosowywać ją do własnych potrzeb. Niestety, zła informacja jest taka, że w VBA po prostu nie ma takich możliwości. Z drugiej jednak strony, dobra informacja jest taka, że z modyfikacją Wstążki innymi metodami nie jest tak źle. W tym rozdziale omówię kilka ciekawych metod pozwalających na dopasowanie interfejsu użytkownika programu Excel.

Dostosowywanie Wstążki

W tym podrozdziale opiszę wybrane sposoby dostosowywania Wstążki do własnych potrzeb. Wstążkę możesz modyfikować ręcznie, ale nie możesz wprowadzać do niej zmian za pomocą VBA. Smutne to, ale — niestety — prawdziwe. Jeżeli na przykład napiszesz nową aplikację i będziesz chciał dla niej utworzyć kilka przycisków na Wstążce, musisz takie zmiany wprowadzić na zewnątrz Excela, korzystając z czegoś o dosyć tajemniczo brzmiącej nazwie *RibbonX*.

Ręczne dopasowywanie Wstążki do własnych potrzeb

Ręczne modyfikowanie wyglądu i funkcjonalności Wstążki jest bardzo łatwe, pod warunkiem jednak, że korzystasz z Excela w wersjach 2010 lub 2013. Jeżeli używasz Excela 2007, powinieneś po prostu pominąć ten punkt, ponieważ nie dotyczy tej wersji programu.

Oto elementy Wstążki, które możesz modyfikować w Excelu 2010 i 2013.

✓ **Karty.** Modyfikacje mogą dotyczyć:

- dodawania własnych kart,
- usuwania własnych kart,
- dodawania nowych grup poleceń do karty,
- zmiany kolejności kart,
- zmiany nazw kart,
- ukrywania wbudowanych kart.

✓ **Grupy poleceń.** Modyfikacje mogą dotyczyć:

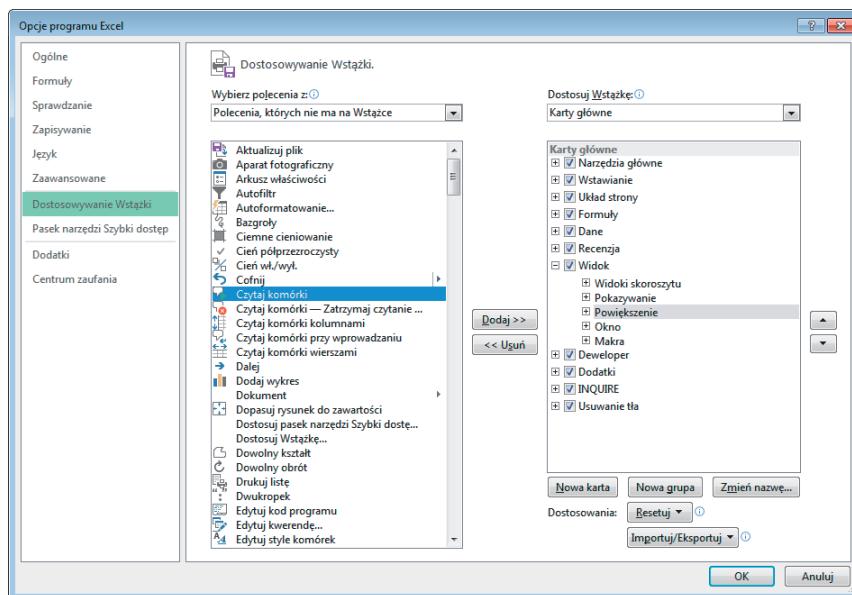
- dodawania własnych grup poleceń,
- dodawania nowych poleceń do własnych grup,
- usuwania poleceń z własnych grup,
- usuwania grup poleceń z kart,
- przenoszenia grup poleceń z jednej karty na inną,
- zmiany kolejności grup poleceń na danej karcie,
- zmiany nazw grup poleceń.

Jest to dosyć kompletna lista tego, co możesz zmienić na Wstążce, ale istnieją również operacje, których po prostu *nie możesz wykonać* (niezależnie od tego, jak bardzo będziesz się starał).

- ✓ Nie możesz usuwać wbudowanych kart (ale możesz je ukrywać).
- ✓ Nie możesz usuwać poleceń z wbudowanych grup poleceń.
- ✓ Nie możesz zmieniać kolejności poleceń we wbudowanych grupach poleceń.

Zmiany we Wstążce możesz wprowadzać za pomocą panelu *Dostosowywanie Wstążki* w oknie *Opje programu Excel* (zobacz rysunek 19.1). Najszybszą metodą wyświetlenia tego okna na ekranie jest kliknięcie prawym przyciskiem myszy w dowolnym miejscu Wstążki i wybranie z menu podręcznego polecenia *Dostosuj Wstążkę*.

Sam proces dostosowywania Wstążki do własnych potrzeb jest bardzo podobny do dostosowywania paska narzędzi *Szybki dostęp*, który omówię nieco dalej w tym rozdziale. Jedyna różnica polega na tym, że teraz musisz zadecydować, w którym miejscu Wstążki wprowadzić zmiany. Ogólnie rzecz biorąc, procedura wygląda następująco.



Rysunek 19.1.
Panel Dostosowywanie Wstążki w oknie dialogowym Opcje programu Excel

1. Rozwiń listę *Wybierz polecenia z*, znajdująca się w lewej, górnej części okna i wybierz z niej grupę zawierającą interesujące Cię polecenie.
2. W polu listy poniżej odszukaj i zaznacz wybrane polecenie.
3. Rozwiń listę *Dostosuj Wstążkę*, znajdująca się w prawej, górnej części okna, i wybierz z niej żądaną grupę kart poleceń.

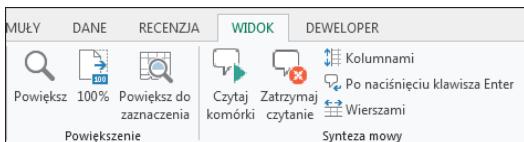
Karty główne to karty, które są zawsze widoczne na Wstążce; *Karty narzędzi* to karty kontekstowe; pojawiają się na Wstążce tylko wtedy, kiedy powiązany z nimi obiekt zostanie zaznaczony.

4. W polu listy po prawej stronie okna odszukaj i zaznacz kartę oraz grupę poleceń, w której chcesz umieścić nowe polecenie.
Aby rozwinąć hierarchiczną strukturę listy, klikaj przyciski rozwijające kolejne poziomy listy.
5. Aby dodać zaznaczone polecenie z pola po lewej stronie okna do karty lub grupy poleceń po prawej stronie okna, naciśnij przycisk *Dodaj*.

Pamiętaj, że za pomocą przycisku *Nowa karta* możesz utworzyć nową kartę poleceń na Wstążce, a przy użyciu przycisku *Nowa grupa* możesz utworzyć nową grupę poleceń na danej karcie poleceń. Nowe karty i grupy poleceń mają po utworzeniu nazwy domyślne, więc zapewne będziesz chciał je zmienić. Aby to zrobić, zaznacz wybraną kartę lub grupę poleceń i naciśnij przycisk *Zmień nazwę*.... Jeśli trzeba, możesz również zmieniać nazwy wbudowanych kart i grup poleceń.

Na rysunku 19.2 przedstawiam wygląd niestandardowej grupy poleceń (o nazwie *Synteza mowy*), którą dodałem do karty *WIDOK*. W grupie znajduje się pięć poleceń.

Rysunek 19.2.
Karta WIDOK
po dodaniu nie-
standardowej
grupy polecień
Synteza mowy



Nie możesz, co prawda, usunąć wbudowanych kart polecień, ale możesz je ukrywać. Aby to zrobić, powinieneś usunąć zaznaczenie z pola znajdującego się po lewej stronie nazwy karty.

Dodawanie do Wstążki przycisku własnego makra

Na szczęście, dodawanie przycisków własnych makr do Wstążki nie stanowi żadnego problemu. Aby to zrobić, powinieneś postępować według instrukcji przedstawionych w poprzednim punkcie, ale w punkcie 1. powinieneś z listy *Wybierz polecenia* z wybrać opcję *Makra*. Po wybraniu tej opcji w polu listy poniżej pojawią się nazwy wszystkich dostępnych makr i będziesz mógł je bez trudności dodać do Wstążki. Zanim to zrobisz, będziesz musiał wybrać kartę i grupę polecień, w której chcesz dodać przyciski makr.

Jeżeli umieścisz przycisk makra na Wstążce, będzie on widoczny nawet wtedy, kiedy zamkniesz skoroszyt zawierający to makro. W takiej sytuacji kliknięcie przycisku makra spowoduje automatycznie otwarcie odpowiedniego skoroszytu i uruchomienie makra.



Jeżeli dadasz do Wstążki przycisk polecenia uruchamiający wybrane makro, taka modyfikacja odnosi się tylko do Twojej kopii Excela. Modyfikacje wprowadzane w ten sposób we Wstążce nie są zapisywane w żadnym skoroszycie. Innymi słowy, jeżeli przekażesz teraz swój skoroszyt koleżce, zmiany wykonane ręcznie na Wstążce *nie pojawią się* w Excelu u Twojego kolegi.

Dostosowywanie Wstążki za pomocą kodu XML

W niektórych sytuacjach będziesz chciał, aby zmiany na Wstążce wykonały się automatycznie po otwarciu danego skoroszytu lub dodatku. Dzięki takiemu rozwiązaniu użytkownicy mogą mieć na przykład ułatwiony dostęp do makr, które dla nich przygotowałeś, a dodatkowo eliminuje ono potrzebę wprowadzania przez nich ręcznych zmian na Wstążce.

Automatyczne wykonywanie zmian we Wstążce podczas otwierania skoroszytu lub dodatku jest możliwe w Excelu 2007 i wersjach późniejszych, ale z góry uprzedzam, że nie jest to proste zadanie. Automatyczna modyfikacja Wstążki wymaga napisania odpowiedniego kodu XML w edytorze tekstu, skopiowania tego kodu do odpowiedniego miejsca w pliku skoroszytu, edytowania zawartości co najmniej kilku plików XML („zaszytych” wewnątrz nowego formatu plików Excela, który jest kontenerem ZIP zawierającym szereg spakowanych plików) i wreszcie napisania odpowiednich procedur VBA, obsługujących klikanie formantów, które zdefiniowałeś w plikach XML.

Pobieranie oprogramowania

Jeżeli chcesz samodzielnie wprowadzać modyfikacje do Wstążki omawiane w tym przykładzie, powinieneś pobrać z internetu program o nazwie *Custom UI Editor for Microsoft Office*. Ten mały, bezpłatny edytor znakomicie ułatwia proces wprowadzania zmian we Wstążce aplikacji pakietu Microsoft Office. Co prawda, korzystanie z tego programu nadal wymaga całkiem sporego nakładu pracy, ale jest znacznie łatwiejsze niż wprowadzanie wszystkich zmian ręcznie.

Bezpłatną kopię edytora możesz łatwo znaleźć w sieci, wpisując w wyszukiwarce sieciowej hasło *Custom UI Editor for Microsoft Office* (nie podaj tutaj konkretnych adresów stron, z których można pobrać ten program, ponieważ od czasu do czasu ulegają zmianie).

Program ma niewielkie rozmiary i, co najważniejsze, jest całkowicie bezpłatny.

Wyjaśnianie wszystkich zawiłości i procedur związanych z modyfikacją Wstążki — niestety — wykraca daleko poza ramy tej książki. Z drugiej jednak strony, chciałem przedstawić przynajmniej jeden krótki przykład ilustrujący operacje, jakie musisz wykonać w celu utworzenia nowej, własnej grupy poleceń na karcie **NARZĘDZIA GŁÓWNE**. Nowa grupa będzie nosiła nazwę *Moja grupa* i będzie zawierała jeden przycisk poleceń o nazwie *Kliknij mnie*. Naciśnięcie tego przycisku będzie uruchamiało makro o nazwie *ShowMessage*.

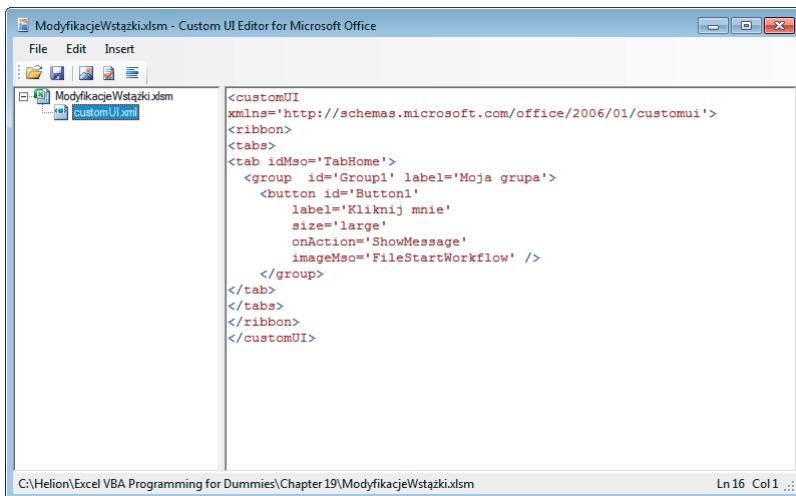
Na szczęście, istnieją odpowiednie programy, które nieco ułatwiają cały proces modyfikacji Wstążki — nie zmienia to jednak faktu, że aby z nich skorzystać, nadal musisz wykazać się wystarczającą znajomością kodu XML.



Skoroszyt z przykładami omawianymi w tym punkcie znajdziesz na stronie internetowej tej książki. Jeżeli jednak chciałbyś utworzyć taki skoroszyt samodzielnie, powinieneś wykonać polecenia opisane poniżej.

- 1. Utwórz nowy, pusty skoroszyt Excela.**
 - 2. Zapisz skoroszyt na dysku i nadaj mu nazwę *ModyfikacjeWstążki.xlsx*.**
 - 3. Zamknij skoroszyt.**
 - 4. Uruchom edytor *Custom UI Editor for Microsoft Office*.**
- Jeżeli nie posiadasz tego programu, powinieneś poszukać go w sieci, pobrać i zainstalować na swoim komputerze. Więcej szczegółowych informacji na ten temat znajdziesz w ramce „Pobieranie oprogramowania”.
- 5. Z menu głównego edytora wybierz polecenie *File/Open* i odszukaj, a następnie otwórz skoroszyt, który utworzyłeś w punkcie 2.**
 - 6. Z menu głównego edytora wybierz polecenie *Insert/Office 2007 Custom UI Part*.**
- powinieneś wybrać to polecenie nawet wtedy, kiedy używasz Excela 2010 lub Excela 2013.
- 7. Wpisz kod XML przedstawiony poniżej w panelu kodu (o nazwie *customUI.xml*), który pojawił się w oknie edytora (zobacz rysunek 19.3).**

```
<customUI xmlns='http://schemas.microsoft.com/office/2006/01/customui'>
<ribbon>
<tabs>
<tab idMso='TabHome'>
```



Rysunek 19.3.
Kod RibbonX
wyświetlony
w oknie edytora
Custom UI

```

<group id='Group1' label='Moja grupa'>
<button id='Button1' label='Kliknij mnie' size='large' onAction='ShowMessage' imageMso='FileStartWorkflow' />
</group>
</tab>
</tabs>
</ribbon>
</customUI>

```

8. Naciśnij przycisk *Validate*, znajdujący się na pasku narzędzi edytora.

Jeżeli podczas wprowadzania kodu popełniłeś jakiś błąd, na ekranie pojawi się komunikat opisujący zaistniały problem. Oczywiście, jeśli pojawiły się jakieś błędy, przed zakończeniem pracy będziesz musiał je poprawić.

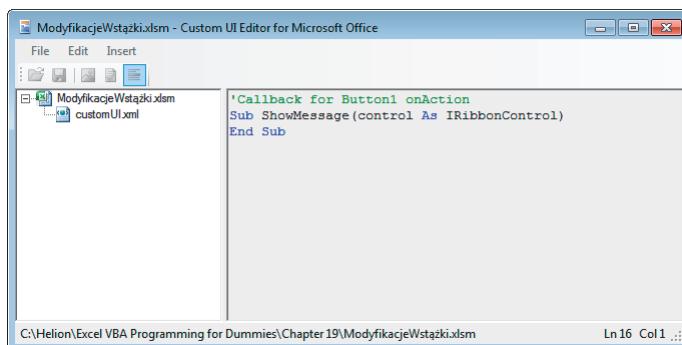
9. Naciśnij przycisk *Generate Callbacks*.

Edytor Custom UI wygeneruje szkielet procedury VBA, która będzie wykonywana po naciśnięciu przycisku na Wstążce (zobacz rysunek 19.4). Procedura nie jest wstawiana do skoroszytu, więc powinieneś sobie zapisać jej kopię, abyś mógł jej później użyć w kodzie aplikacji (jeśli masz dobrą pamięć, możesz po prostu zapamiętać kod tej procedury...).

10. Powróć do modułu *customUI.xml* i następnie z menu głównego edytora wybierz polecenie *File/Save* (zamiast tego możesz nacisnąć przycisk polecenia *Save*, znajdujący się na pasku narzędzi edytora).

11. Zamknij edytowany plik, wybierając z menu głównego polecenie *File/Close*.

Rysunek 19.4.
Szkielet procedury zwrotnej VBA, która będzie wywoływana po naciśnięciu przycisku na Wstążce



12. Otwórz skoroszyt w Excelu i przejdź na kartę NARZĘDZIA GŁÓWNE.

Na Wstążce powinieneś zobaczyć nową grupę poleceń, w której znajduje się jeden przycisk (który jednak jeszcze nie działa).

13. Naciśnij kombinację klawiszy *Alt+F11*, aby przejść do edytora VBE.

14. Wstaw nowy moduł VBA i wklej (lub wpisz) kod procedury zwrotnej VBA, która została wygenerowana w punkcie 9. Następnie dodaj polecenie MsgBox, abyś wiedział, czy po naciśnięciu przycisku na Wstążce procedura działa poprawnie.

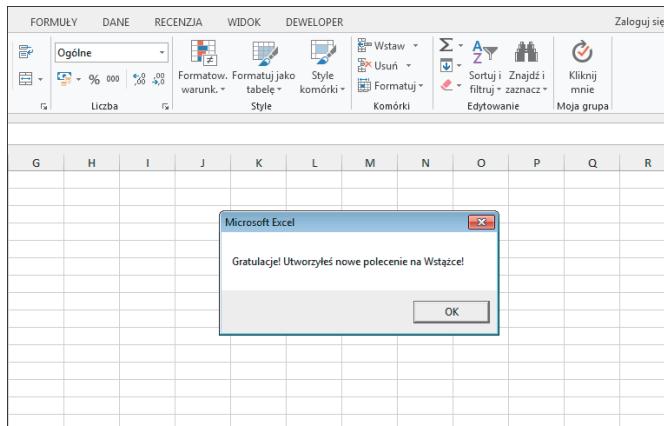
Kod procedury zwrotnej może wyglądać następująco.

```
Sub ShowMessage(control As IRibbonControl)
    MsgBox "Gratulacje! Utworzyłeś nowe polecenie na Wstążce!"
End Sub
```

15. Ponownie naciśnij kombinację klawiszy *Alt+F11*, aby powrócić do głównego okna Excela, i naciśnij nowo utworzony przycisk na Wstążce.

Jeżeli wszystko poszło tak, jak powinno, na ekranie zobaczysz okno komunikatu, przedstawione na rysunku 19.5.

Rysunek 19.5.
Dowód na to, że dodawanie nowych poleceń do Wstążki za pomocą kodu XML jest możliwe





Po wybraniu z menu głównego polecenia *Insert/Office 2007 Custom UI Part* edytor *Custom UI* wstawia szkielet kodu XML dla Excela 2007. Zauważyleś zapewne, że edytor *Custom UI* posiada również opcję do wstawiania kodu XML optymalizowanego dla Excela 2010 (i pewnie niedługo będzie miał już analogiczną opcję dla Excela 2013). Dla zachowania maksymalnej kompatybilności powinieneś jednak wybierać opcję przeznaczoną dla Excela 2007, niezależnie od tego, której wersji tego narzędzia używasz.

Z opcji wstawiania kodu XML przeznaczonego dla Excela 2010 (lub 2013) powinieneś korzystać tylko w sytuacji, kiedy musisz wykorzystać funkcje i mechanizmy, które są dostępne tylko w wersji Excela 2010 (lub Excela 2013). Jeśli trzeba, możesz również utworzyć osobny moduł kodu XML *Custom UI* dla każdej wersji Excela osobno, aczkolwiek takie rozwiązanie bardzo rzadko będzie konieczne.

Kiedy powstawała ta książka, edytor *Custom UI* nie zawierał jeszcze opcji wstawiania kodu XML optymalizowanego dla Excela 2013.

Na pewno zauważyleś, że modyfikowanie Wstążki za pomocą kodu XML wcale nie jest zadaniem prostym ani intuicyjnym. Nawet mając do pomocy dobre narzędzie (takie jak na przykład omawiany tutaj edytor *Custom UI*), nadal musisz posiadać sporą wiedzę na temat kodu XML. Jeżeli jednak zagadnienia związane z modyfikowaniem Wstążki wydają Ci się interesujące, powinieneś poszukać w internecie informacji na ten temat lub po prostu kupić dobrą książkę.

Dodawanie przycisków do paska narzędzi Szybki dostęp

Jeżeli posiadasz makro, z którego bardzo często korzystasz, możesz umieścić przycisk wywołujący takie makro na pasku narzędzi *Szybki dostęp*. Wykonanie tej operacji nie jest skomplikowane, ale musisz to zrobić ręcznie. Pasek narzędzi *Szybki dostęp* został pomysły tak, aby modyfikacje wprowadzali do niego użytkownicy Excela, a nie programiści... Aby to zrobić, wykonaj polecenia przedstawione poniżej.

1. Kliknij pasek narzędzi *Szybki dostęp* prawym przyciskiem myszy i z menu podręcznego wybierz polecenie *Dostosuj pasek narzędzi Szybki dostęp*. Na ekranie pojawi się okno dialogowe *Opcje programu Excel*.
2. Na liście rozwijanej *Wybierz polecenia z wskaź opcję Makra*.
3. Z pola listy wybierz nazwę makra, którego przycisk chcesz dodać do paska narzędzi.
4. Naciśnij przycisk *Dodaj* i przycisk polecenia wywołujący wybrane makro zostanie dodany do listy poleceń paska narzędzi *Szybki dostęp* zlokalizowanej w prawej części okna.

5. Jeżeli chcesz, możesz nacisnąć przycisk *Modyfikuj...*, aby zmienić ikonę przycisku i ewentualnie jego nazwę.

Kiedy naciśniesz przycisk makra na pasku narzędzi *Szybki dostęp*, skoroszyt zawierający makro zostanie automatycznie załadowany i otwarty (o ile nie został otwarty już wcześniej) — makro może być wykonane, kiedy otwarty jest dowolny skoroszyt.

Jeśli trzeba, możesz również tak skonfigurować przycisk makra, aby pojawiał się na pasku narzędzi *Szybki dostęp* tylko wtedy, kiedy otwarty zostanie określony skoroszyt. Aby to zrobić, powinieneś przed dodaniem makra rozwinąć listę *Dostosuj pasek narzędzi Szybki dostęp*, znajdująca się w prawej, górnej części okna opcji Excela, i wybrać z niej nazwę żądanego skoroszytu (zamiast domyślnej opcji *Dla wszystkich dokumentów*).

Jeżeli posiadasz wiele uniwersalnych makr, których możesz używać w wielu różnych skoroszytach, powinieneś pomyśleć o przechowywaniu ich w swoim osobistym skoroszycie *Personal Macro Workbook*.

Ponieważ zagadnienia związane z modyfikacją Wstążki przy użyciu kodu XML są jednak zbyt skomplikowane dla początkującego programisty VBA, dalej w tym rozdziale skoncentruję się wyłącznie na takich modyfikacjach interfejsu użytkownika programu Excel, które można wykonać za pomocą *starych* metod (czyli za pomocą kodu VBA). Co prawda, nie jest to może tak efektywne jak modyfikowanie Wstążki, ale za to dużo łatwiejsze i również pozwala na szybkie uruchamianie wybranych makr.

Dostosowywanie menu podręcznego

Przed pojawiением się Excela 2007 programiści VBA wykorzystywali obiekty CommandBar do tworzenia własnych, niestandardowych menu, pasków narzędzi i menu podręcznego (dostępnego po kliknięciu obiektu prawym przyciskiem myszy).

Począwszy od Excela 2007, obiekty CommandBar mają dosyć dziwny status. Jeżeli utworzysz procedurę VBA, która będzie modyfikowała menu lub pasek narzędzi, Excel przechwyci ten kod i po prostu zignoruje wiele jego poleceń. Zamiast wyświetlania takich dobrze zaprojektowanych i przemyślnych rozszerzeń interfejsu użytkownika Excel 2007 (i wersje późniejsze) po prostu wrzuca wszystkie modyfikacje menu i pasków narzędzi dokonywane za pomocą obiektów CommandBar na jedną, dedykowaną do tego celu kartę o nazwie **DODATKI**.

Modyfikacje menu i pasków narzędzi wyświetlane są na karcie **DODATKI** w grupach *Polecenia menu* lub *Niestandardowe paski narzędzi*. Z drugiej strony, modyfikacje wprowadzane do menu podręcznego wywoływanego kliknięciem prawym przyciskiem myszy (które również wykorzystuje obiekty CommandBar) działają — na szczęście — tak jak w poprzednich wersjach Excela i właśnie o takich zagadnieniach napiszę w kolejnych punktach.

Rodzaje obiektów CommandBar

Excel obsługuje trzy rodzaje obiektów CommandBar, rozróżnianych za pomocą ich właściwości Type. Ponieważ będziemy się zajmować modyfikacjami menu podręcznego, interesujący będzie dla nas typ 2 takich obiektów, znany również jako typ **msoBarTypePopUp** (jest to nazwa wbudowanej stałej, reprezentującej taki typ obiektów).

Wyświetlanie wszystkich menu podręcznych

Procedura przedstawiona poniżej wykorzystuje kolekcję CommandBars. Po uruchomieniu procedury wyświetla w arkuszu nazwy wszystkich dostępnych menu podręcznych — czyli obiektów CommandBar, które posiadają właściwość Type ustawioną na wartość 2 (**msoBarTypePopUp**). Dla każdego z obiektów procedura wyświetla numer indeksu (właściwość **Index**), nazwę w języku lokalnym (właściwość **NameLocal**) oraz nazwę oryginalną (właściwość **Name**), której będziemy używać w celu odwołania się do elementów kolekcji CommandBars.

```
Sub ShowShortcutMenusName()
    Dim Row As Long
    Dim cbar As CommandBar
```

```

Row = 1
For Each cbar In Application.CommandBars
    If cbar.Type = msoBarTypePopup Then
        Cells(Row, 1) = cbar.Index
        Cells(Row, 2) = cbar.NameLocal
        Cells(Row, 3) = cbar.Name
        Row = Row + 1
    End If
Next cbar
End Sub

```



Skoroszyt z tym przykładem znajdziesz na stronie internetowej tej książki. Na rysunku 19.6 przedstawiam fragment wyników działania procedury (uruchomionej w polskiej wersji Excela 2013). Możesz na przykład zobaczyć, że menu podrzczne o nazwie *Pły* ma indeks o numerze 41. To menu podrzczne o nieco dziwnej nazwie pojawia się na ekranie, kiedy klikniesz prawym przyciskiem myszy kartę arkusza w głównym oknie Excela. W zasadzie mógłbyś oczekiwać, że menu to będzie nosiło nazwę *Karty skoroszytu* (*Workbook tabs*), ale takie menu już istnieje (obiekt *CommandBar* o numerze indeksu 34) i wyświetla nazwy poszczególnych arkuszy.

A	B	C	D
1	22 Menu wykresu przestawnego	PivotChart Menu	
2	34 Karty skoroszytu	Workbook tabs	
3	35 Komórka	Cell	
4	36 Kolumna	Column	
5	37 Wiersz	Row	
6	38 Komórka	Cell	
7	39 Kolumna	Column	
8	40 Wiersz	Row	
9	41 Pły	Ply	
10	42 Komórka XLM	XLM Cell	
11	43 Dokument	Document	
12	44 Pulpit	Desktop	
13	45 Wyłącz domyślne użycie metody 'przeciągnij i upuść'	Nondefault Drag and Drop	
14	46 Autowypełnianie	AutoFill	
15	47 Przycisk	Button	
16	48 Okno dialogowe	Dialog	
17	49 Seria	Series	
18	50 Obszar kreślenia	Plot Area	
19	51 Podłoga i ściany	Floor and Walls	
20	52 Linia trendu	Trendline	
21	53 Wykres	Chart	
22	54 Formatuj serię danych	Format Data Series	

Rysunek 19.6.

Wyniki działania makra VBA wyświetlającego listę nazw wszystkich menu podrzcznych

Odwołania do elementów kolekcji CommandBars

W teorii do wybranego obiektu *CommandBar* możesz się odwołać za pomocą wartości jego indeksu w kolekcji (właściwość *Index*) lub za pomocą właściwości *Name* (do odwołania się do obiektu nie można używać właściwości *NameLocal*). Jeżeli jednak przyjrzesz się liście nazw menu podrzcznych, przedstawionej na rysunku 19.6, przekonasz się, że na przykład menu podrzczne o nazwie *Komórka* (ang. *Cell*) posiada indeksy o wartościach 35 lub 38. Dzieje się tak, ponieważ zawartość menu podrzcznego *Komórka* (ang. *Cell*) zmienia się w zależności od tego, w jakim stanie znajduje się w danej chwili Excel. Menu o indeksie 35 wyświetlane jest po kliknięciu prawym przyciskiem myszy, kiedy

Excel znajduje się w normalnym widoku arkusza, a menu o indeksie 38 pojawia się, kiedy Excel pracuje w trybie podglądu podziału stron. Do wybranego menu podrzędnego możesz odwoływać się na jeden z dwóch wspomnianych wcześniej sposobów.

```
Application.CommandBars(35)
```

lub

```
Application.CommandBars("Cell")
```



Może Ci się wydawać, że odwoływanie się do obiektów CommandBar za pomocą wartości indeksu (właściwość `Index`) jest lepszym rozwiązaniem. Niestety, nic bardziej mylnego! Problem polega na tym, że numery indeksów elementów kolekcji `CommandBars` w różnych wersjach Excela zmieniają się. Jeżeli zatem napisałeś w Excelu 2013 procedurę VBA, która dodaje nowy element o indeksie 35 do obiektu `CommandBar`, to próba wykonania takiej procedury w Excelu 2010 zakończyłaby się niepowodzeniem, ponieważ w tej wersji programu menu `Cell` posiadają numery indeksów równe 12 i 40. Krótko mówiąc, powinieneś zapomnieć o możliwości odwoływania się do obiektów `CommandBar` po numerach indeksów i zawsze używać odwołań po nazwie.

Przy użyciu kodu VBA możesz wyświetlić na ekranie dowolne, wbudowane menu podrzęenne Excela (klikanie prawym przyciskiem myszy nie jest do tego potrzebne!). Przykładowo polecenie przedstawione poniżej powoduje wyświetlenie na ekranie menu podrzędnego `Cell`.

```
Application.Commandbars("Cell").ShowPopup
```

Odwolania do formantów obiektu `CommandBar`

Obiekt `CommandBar` zawiera obiekty klasy `Control`, którymi są przyciski, menu lub elementy menu. Poniżej przedstawiam kod prostej procedury, która wyświetla etykietę pierwszego elementu menu podrzędnego `Cell`.

```
Sub ShowCaption()
    MsgBox Application.CommandBars("Cell")._
        Controls(1).Caption
End Sub
```

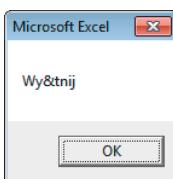
Po uruchomieniu takiej procedury na ekranie pojawi się okno dialogowe przedstawione na rysunku 19.7. Znak `&` (*ampersand*) jest używany do wskazania podkreślonej litery menu — czyli inaczej do wskazania klawisza, który jest wykorzystywany jako skrót klawiszowy do uruchamiania danego polecenia menu.



W niektórych przypadkach obiekty `Control` w menu podrzędnym zawierają inne obiekty `Control`. Przykładowo formant `Sortuj` menu podrzędnego `Cell` zawiera szereg innych formantów.

Każdy formant posiada właściwości `Name` oraz `Id`. Dostęp do wybranego formantu możesz uzyskać za pośrednictwem dowolnej z tych dwóch właściwości (aczkolwiek lokalizacja formantu za pomocą właściwości `Id` jest nieco trudniejszym zadaniem).

Rysunek 19.7.
Wyświetlanie właściwości
Caption wybranego ele-
mentu menu podręcznego



```
Sub AccessControlByName()
    MsgBox CommandBars("Cell").Controls("Kopiuj").Caption
End Sub

Sub AccessControlById()
    MsgBox CommandBars("Cell").FindControl(ID:=19, Recursive:=True).Caption
End Sub
```



Nie używaj właściwości `Caption` w odwołaniach do formantów, jeżeli tworzysz kod, który może być wykorzystywany przez użytkowników pracujących z innymi wersjami językowymi Excela. Wartość właściwości `Caption` zależy od wersji językowej Excela, stąd tak napisany kod po prostu nie będzie działał na innych systemach. Zamiast właściwości `Caption` powinieneś w takich sytuacjach używać metody `FindControl` w połączeniu z właściwością `Id` danego formantu (której wartość jest niezależna od wersji językowej Excela). Na szczęście, nazwy obiektów `CommandBar` również się nie zmieniają i występują zawsze w wersji angielskiej.

Właściwości formantów obiektu `CommandBar`

Formanty obiektu `CommandBar` posiadają cały szereg właściwości określających ich wygląd i sposób działania. Poniżej przedstawiam listę najbardziej użytecznych właściwości formantów obiektu `CommandBar`.

- ✓ `Caption` — tekst, który jest wyświetlany na formancie. Jeżeli formant ma wyłącznie postać obrazu, tekst właściwości `Caption` pojawia się, kiedy ustawisz wskaźnik myszy nad tym formantem.
- ✓ `FaceID` — liczba reprezentująca obraz wyświetlany obok tekstu formantu.
- ✓ `BeginGroup` — ma wartość `True`, jeżeli przed formantem wyświetlany jest pasek separatora.
- ✓ `OnAction` — nazwa makra VBA, które jest wykonywane po kliknięciu formantu przez użytkownika.
- ✓ `BuiltIn` — ma wartość `True`, jeżeli formant jest wbudowanym formantem programu Excel.
- ✓ `Enabled` — ma wartość `True`, jeżeli formant jest włączony.
- ✓ `Visible` — ma wartość `True`, jeżeli formant jest widoczny. Wiele menu podręcznych posiada ukryte formanty.

✓ **ToolTipText** — tekst, który pojawia się, kiedy użytkownik ustawi wskaźnik myszy nieruchomo nad formantem (nie odnosi się do menu podręcznego).

Procedura ShowShortcutMenuItems, której kod przedstawiam niżej, tworzy tabelę, w której wyświetlane są wszystkie elementy pierwszego poziomu poszczególnych menu podręcznych. Dodatkowo procedura identyfikuje formanty ukryte, umieszczając ich nazwy (właściwość **Caption**) w nawiasach <>.

```
Sub ShowShortcutMenuItems()
    Dim Row As Long, Col As Long
    Dim Cbar As CommandBar
    Dim Ctl As CommandBarControl
    Row = 1
    Application.ScreenUpdating = False
    For Each Cbar In Application.CommandBars
        If Cbar.Type = msoBarTypePopup Then
            Cells(Row, 1) = Cbar.Index
            Cells(Row, 2) = Cbar.NameLocal
            Col = 3
            For Each Ctl In Cbar.Controls
                If Ctl.Visible Then
                    Cells(Row, Col) = Ctl.Caption
                Else
                    Cells(Row, Col) = "<" & Ctl.Caption & ">"
                End If
                Col = Col + 1
            Next Ctl
            Row = Row + 1
        End If
    Next Cbar
End Sub
```

Na rysunku 19.8 przedstawiam fragment wyników działania tej procedury po uruchomieniu w Excelu 2013 (wyniki działania w Excelu 2010 będą inne). Oczywiście, nie jest to zbyt pasjonująca lektura, ale można tutaj znaleźć wiele bardzo przydatnych informacji.



Skoroszyt z tym przykładem znajdziesz na stronie internetowej tej książki. Po uruchomieniu makra przekonasz się, że wiele menu podręcznych posiada ukryte formanty.

Przedstawiłem bardzo szybki i pobiczny przegląd wybranych zagadnień związanych z kolekcjami **CommandBars**. Oczywiście, pozostało jeszcze całe mnóstwo nieporuszonych tutaj zagadnień, aczkolwiek ich omawianie wykracza daleko poza ramy tej książki. W kolejnych podrozdziałach przedstawię kilka przykładów, które pomogą Ci wyjaśnić ewentualne wątpliwości.

Wraz z prowadzeniem Wstępki, czyli nowego interfejsu użytkownika programu Excel, bardzo wiele rzeczy się zmieniło. Z punktu widzenia programistów wyszło to, jak zawsze, i niektóre zmiany są na lepsze, a niektóre na gorsze. Jakby na to nie patrzeć, możliwości kontrolowania interfejsu użytkownika z poziomu kodu VBA są po tych zmianach znacznie ograniczone.

Rysunek 19.8.
 Wyświetlanie wszystkich elementów pierwszego poziomu poszczególnych menu podręcznych

A	B	C	D
1	22 Menu wykresu przestawnego	&Ustawienia pola	&Opcje...
2	34 Karty skoroszytu	Arkusz1	<&Lista arkuszy>
3	35 Komórka	Wy&tnij	&Kopiuj
4	36 Kolumna	Wy&tnij	&Kopiuj
5	37 Wiersz	Wy&tnij	&Kopiuj
6	38 Komórka	Wy&tnij	&Kopiuj
7	39 Kolumna	Wy&tnij	&Kopiuj
8	40 Wiersz	Wy&tnij	&Kopiuj
9	41 Ply	W&staw...	Usuń
10	42 Komórka XLM	Wy&tnij	&Kopiuj
11	43 Dokument	Zapi&sz	Z&apisz jako...
12	44 Pulpit	&Nowy...	Ot&wórz...
13	45 Wyłącz domyślne użycie metody 'przeciagnij i upuść'	&Przenieś tutaj	&Kopiuj tutaj
14	46 Autowypełnianie	&Kopiuje komórki	&Wypełnij serią
15	47 Przycisk	Wy&tnij	&Kopiuj
16	48 Okno dialogowe	Wkl&ej	Kolejność &tabulacji...
17	49 Seria	Wybrany &obiekt	&Typ wykresu...
18	50 Obszar kreślenia	Wybrany &obiekt	&Typ wykresu...
19	51 Podłożo i ściany	Wybrany &obiekt	Widok &3-W...
20	52 Linia trendu	Wybrany &obiekt	&Wyczyszczać
21	53 Wykres	Wybrany &obiekt	&Wyczyszczać
22	54 Formatuj serię danych	Wybrany &obiekt	&Typ wykresu...

Przykłady zastosowania VBA do modyfikacji menu podręcznego

W tym podrozdziale zaprezentuję kilka praktycznych przykładów zastosowania VBA do modyfikacji menu podręcznego Excela. Przykłady pozwolą Ci na zorientowanie się w tym, co możesz zrobić z menu podręcznym, i — oczywiście — mogą być w dowolny sposób modyfikowane i wykorzystywane w Twoich własnych projektach.

Resetowanie wszystkich wbudowanych menu podręcznych

Procedura, której kod zamieszczam poniżej, przywraca wszystkie wbudowane paski narzędzi do ich początkowego stanu.

```
Sub ResetAll()
    Dim cbar As CommandBar
    For Each cbar In Application.CommandBars
        If cbar.Type = msoBarTypePopup Then
            cbar.Reset
            cbar.Enabled = True
        End If
    Next cbar
End Sub
```

Wykonanie powyższej procedury nie będzie miało żadnego widocznego efektu dopóty, dopóki wcześniej nie uruchomisz procedury, która w jakiś sposób dodaje, usuwa lub wyłącza elementy menu podręcznego.

Dodawanie nowego elementu do menu podręcznego Cell

W rozdziale 16. opisałem narzędzie o nazwie ChangeCase. Teraz możesz rozszerzyć nieco jego funkcjonalność poprzez umożliwienie uruchamiania go z poziomu menu podręcznego.



Skoroszyt z tym przykładem znajdziesz na stronie internetowej tej książki.

Procedura AddToShortcut, której kod zamieszczam poniżej, dodaje nowy element do menu podręcznego Cell. Jak pamiętasz, Excel posiada dwa menu podręczne o nazwie Cell. Procedura omawiana w tym przykładzie modyfikuje standardowe menu Cell, dostępne po kliknięciu komórki prawym przyciskiem myszy (a nie menu Cell dostępne w trybie podglądu podziału stron).

```
Sub AddToShortcut()
    Dim Bar As CommandBar
    Dim NewControl As CommandBarButton
    DeleteFromShortcut
    Set Bar = Application.CommandBars("Cell")
    Set NewControl = Bar.Controls.Add _
        (Type:=msoControlButton, ID:=1, _
        temporary:=True)
    With NewControl
        .Caption = "&Zmień wielkość liter..."
        .OnAction = "ChangeCase"
        .Style = msoButtonIconAndCaption
    End With
End Sub
```

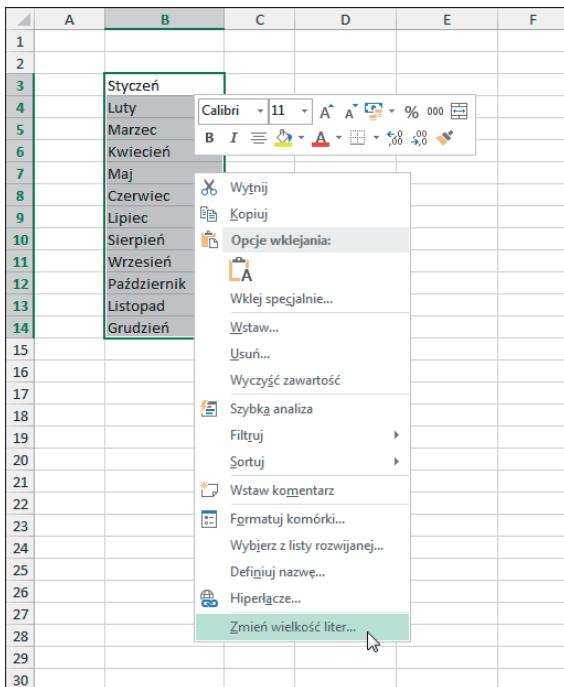


Kiedy modyfikujesz menu podręczne, wprowadzone zmiany pozostają aktywne do czasu restartu Excela. Innymi słowy, zmodyfikowanie menu podręczne nie zostanie zresetowane po zamknięciu skoroszytu zawierającego kod VBA. Z tego powodu, jeżeli tworzysz kod, który modyfikuje menu podręczne, zazwyczaj uzupełniasz go o procedury, które anulują modyfikacje i przywracają stan początkowy menu.

Procedura DeleteFromShortcut, której kod został zamieszczony poniżej, usuwa nowy element menu podręcznego Cell.

```
Sub DeleteFromShortcut()
    On Error Resume Next
    Application.CommandBars("Cell").Controls
        ("&Zmień wielkość liter...").Delete
End Sub
```

Na rysunku 19.9 przedstawiam nowy element menu podręcznego, wyświetlonego po kliknięciu komórki prawym przyciskiem myszy.



Rysunek 19.9.
Menu podręczne z dodatkową
opcją Zmień
wielkość liter...

Co nowego w Excelu 2013?

Jeżeli w poprzednich wersjach używałeś VBA do pracy z menu podręcznym, powinieneś zwrócić szczególną uwagę na to, że w Excelu 2013 zaszyły bardzo istotne zmiany.

W poprzednich wersjach, jeżeli Twój kod wprowadzał jakieś modyfikacje do menu podręcznego, to takie zmiany były widoczne we wszystkich skoroszytach. Jeśli na przykład dodałeś nowe polecenie do menu podręcznego wyświetlanego po kliknięciu komórki prawym przyciskiem myszy, taka zmiana była widoczna we wszystkich otwartych skoroszytach (oraz w tych, które otworzyłeś później). Innymi słowy, zmiany w menu podręcznym były wprowadzane na poziomie aplikacji.

Excel 2013 wykorzystuje nowy, jednodokumentowy interfejs użytkownika, co ma znaczący wpływ na sposób działania menu podręcznego. Zmiany wprowadzone w menu podręcznym działają tylko w oknie aktywnego skoroszytu. Kiedy uruchomisz kod modyfikujący

menu podręczne w aktywnym skoroszycie, menu podręczne w oknach pozostałych skoroszytów nie zostanie w żaden sposób zmienione. Takie zachowanie jest diametralnie inne od sposobu działania w poprzednich wersjach Excela.

Kolejna ciekawostka — jeżeli użytkownik otworzy skoroszyt (lub utworzy nowy skoroszyt) w sytuacji, kiedy w aktywnym skoroszycie zostało zmodyfikowane menu podręczne, nowe okno skoroszytu niejako „dziedziczy” takie ustawienia menu podręcznego z aktywnego skoroszytu. Innymi słowy, w nowym oknie wyświetlane są takie same menu podręczne jak w oknie, które było aktywne podczas tworzenia nowego okna.

Wnioski? W przeszłości, kiedy otwierałeś skoroszyt lub dodatek zawierający zmodyfikowane menu podręczne, mogłeś być pewien, że te zmiany będą widoczne we wszystkich skoroszytach. Niestety, w Excelu 2013 takiej pewności mieć już nie możesz.

Pierwsze polecenie po bloku deklaracji zmiennych wywołuje procedurę `DeleteFromShortcut`, co gwarantuje, że w menu podrzędnym pojawi się tylko jedna opcja *Zmień wielkość liter....*. Spróbuj teraz „zakomentować” ten wiersz kodu (aby to zrobić, powinieneś wpisać znak apostrofu na początku tego wiersza kodu) i uruchomić całą procedurę kilka razy. Jeżeli teraz klikniesz wybraną komórkę prawym przyciskiem myszy, zobaczyś w menu podrzędnym kilka instancji opcji *Zmień wielkość liter....* Aby się ich pozbyć, musisz kilka razy uruchomić procedurę `DeleteFromShortcut` (po jednym uruchomieniu na każdą nadmiarową opcję w menu podrzędnym).

W większości przypadków będziesz chciał dodawać i usuwać elementy menu podrzędnego całkowicie automatycznie — dodawać nowe elementy menu podrzędnego, kiedy skoroszyt jest otwierany, i usuwać je, kiedy zostanie zamknięty. Wykonanie takiego zadania jest całkiem proste..., jeżeli dokładnie przeczytałeś rozdział 11. Aby uzyskać taki efekt, wystarczy w module kodu `ThisWorkbook` utworzyć dwie proste procedury obsługi zdarzeń.

```
Private Sub Workbook_Open()
    Call AddToShortcut
End Sub

Private Sub Workbook_BeforeClose(Cancel As Boolean)
    Call DeleteFromShortcut
End Sub
```

Procedura `Workbook_Open` jest wykonywana, kiedy skoroszyt jest otwierany, a procedura `Workbook_BeforeClose` jest wykonywana przed zamknięciem skoroszytu. To jest dokładnie to, o co chodziło.

Wyłączanie menu podrzędnego

Jeżeli trzeba, możesz całkowicie wyłączyć wyświetlanie wybranego menu podrzędnego. Możesz na przykład zrobić tak, że kliknięcie komórki arkusza prawym przyciskiem myszy nie będzie wyświetlało normalnego menu podrzędnego. Procedura przedstawiona poniżej jest uruchamiana automatycznie po otwarciu skoroszytu i powoduje wyłączenie menu podrzędnego `Cell`.

```
Private Sub Workbook_Open()
    Application.CommandBars("Cell").Enabled = False
End Sub
```

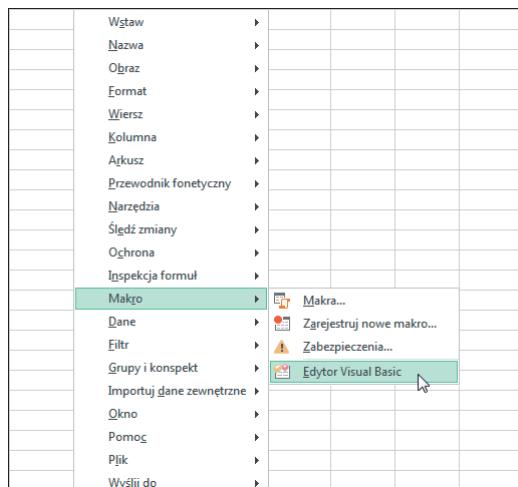
A oto kod procedury komplementarnej, która z kolei powoduje automatyczne włączenie wyświetlania menu podrzędnego `Cell` po zamknięciu skoroszytu.

```
Private Sub Workbook_BeforeClose(Cancel As Boolean)
    Application.CommandBars("Cell").Enabled = True
End Sub
```

Obie procedury muszą być umieszczone w module kodu obiektu `Ten_skoroszyt`.

Tworzenie własnych pasków narzędzi

Jeśli uważnie przeglądasz listę dostępnych obiektów CommandBar, z pewnością zauważysz jeden z nich, o nazwie Built-in Menus. Ten obiekt CommandBar zawiera wszystkie polecenia starego menu Excela 2003. Na rysunku 19.10 przedstawiam fragment tego gigantycznego menu podręcznego. Wszystkie istniejące w starym menu polecenia są dostępne na tym pasku narzędzi, aczkolwiek nie są dobrze uporządkowane.



Rysunek 19.10.
Wyświetlanie
menu podręcz-
nego Built-in
Menus

W tym przykładzie przedstawię procedurę VBA, która (do pewnego stopnia) odtwarza stare, dobre menu Excela 2003. Procedura tworzy nowy obiekt CommandBar i następnie kopiuje do niego wybrane formanty obiektu Built-in Menus.

```
Sub MakeOldMenus()
    Dim cb As CommandBar
    Dim cbc As CommandBarControl
    Dim OldMenu As CommandBar

    ' Jeżeli menu już istnieje, usuń je
    On Error Resume Next
    Application.CommandBars("Stare menu").Delete
    On Error GoTo 0

    ' Utwórz pasek narzędzi w starym stylu
    ' Ustawia ostatni argument na False, aby utworzyć bardziej kompaktowe menu
    Set OldMenu = Application.CommandBars.Add("Stare menu", , True)

    ' Kopiuje formanty z menu podręcznego "Built-in Menus"
    With CommandBars("Built-in Menus")
        .Controls("&Plik").Copy OldMenu
        .Controls("&Edycja").Copy OldMenu
        .Controls("&Widok").Copy OldMenu
        .Controls("&Wstaw").Copy OldMenu
    End With
End Sub
```

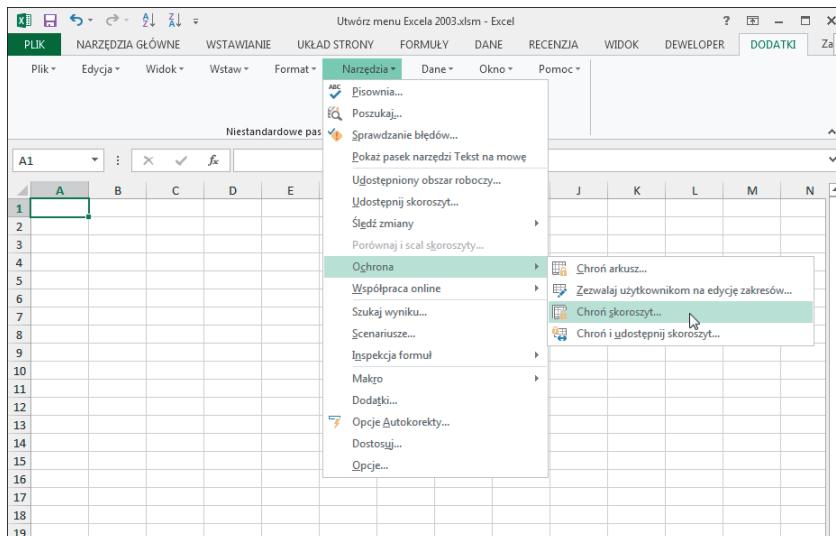
```

.Controls("F&ormat").Copy OldMenu
.Controls("&Narzędzia").Copy OldMenu
.Controls("&Dane").Copy OldMenu
.Controls("&Okno").Copy OldMenu
.Controls("&Pomoc").Copy OldMenu
End With

' Wyświetla menu, które pojawia się na karcie DODATKI
Application.CommandBars("Stare menu").Visible = True
End Sub

```

Na rysunku 19.11 przedstawiam wyniki działania procedury MakeOldMenus. Zwróć uwagę, że nowo utworzone menu pojawiło się na karcie *DODATKI*, w grupie poleceń *Niestandardowe paski narzędzi*. Dzieje się tak, ponieważ z technicznego punktu widzenia to menu jest paskiem narzędzi, który po prostu wygląda jak stare, dobre, klasyczne menu.



Rysunek 19.11.
Pasek narzędzi pretendujący do zaszczytnego miana menu Excela 2003

Nowy obiekt CommandBar nosi nazwę *Stare menu*. Procedura rozpoczyna działanie od usunięcia tego paska narzędzi, o ile pasek o takiej nazwie już istnieje. Następnie procedura rozpoczyna kopowanie formantów „pierwszego poziomu hierarchii” z obiektu CommandBar o nazwie *Built-in Menus* do obiektu *Stare menu*. Nowy pasek narzędzi jest domyślnie ukryty, stąd ostatnie polecenie procedury powoduje jego wyświetlenie.

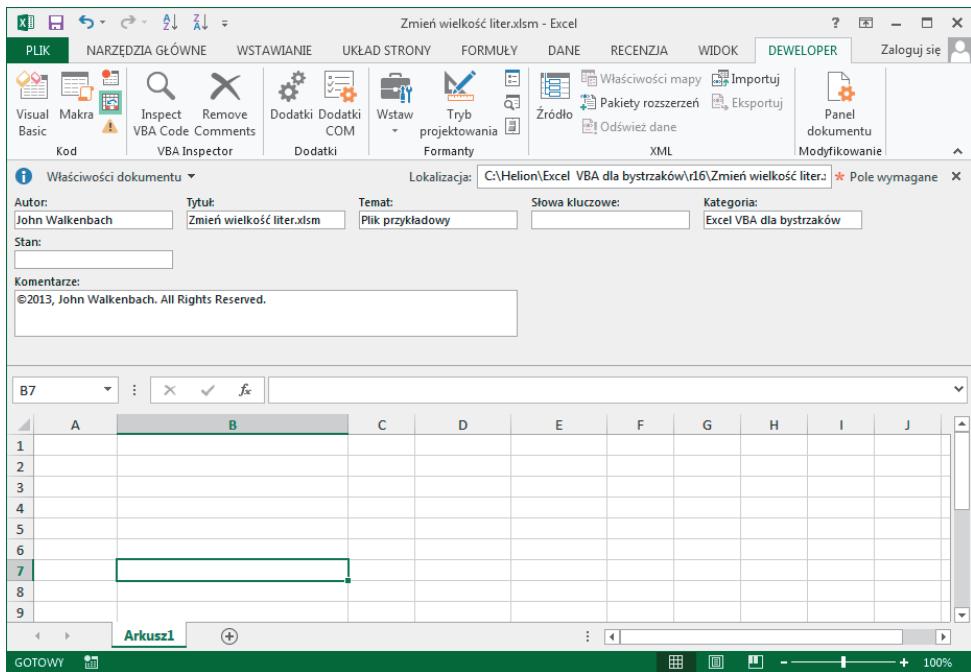
Nowe „menu” nie jest — niestety — perfekcyjne. Kilka poleceń tego menu po prostu nie będzie działać. Z pewnością zauważysz również, że w „menu” *Plik*, w miejscu, gdzie normalnie wyświetlane są nazwy ostatnio otwieranych plików, pojawia się po prostu symbol zastępczy *Miejsce na nazwę niedawno używanego pliku*. Jeżeli chcesz pozbyć się takiego paska narzędzi, możesz kliknąć go prawym przyciskiem myszy i z menu podręcznego wybrać polecenie *Usuń niestandardowy pasek narzędzi*.

Skoroszyt z tym przykładem znajdziesz na stronie internetowej tej książki.



Część V

Od teorii do praktyki



W tej części...

- ✓ przekonasz się, jak możesz tworzyć własne funkcje arkuszowe,
- ✓ dowiesz się, jak spowodować, aby niestandardowe, własne funkcje działały, tak jak wbudowane funkcje Excela,
- ✓ odkryjesz tajemnice dodatków *add-in* Excela,
- ✓ nauczysz się tworzyć proste dodatki *add-in*.

Rozdział 20

Jak tworzyć własne funkcje arkuszowe i jak przeżyć, aby o tym opowiedzieć?

W tym rozdziale:

- ▶ dowiesz się, dlaczego własne funkcje arkuszowe są tak użyteczne,
- ▶ nauczysz się używać funkcji wykorzystujących różne typy argumentów,
- ▶ zrozumiesz działanie okna dialogowego *Wstawianie funkcji*.

Dla wielu programistów jedną z głównych atrakcji języka VBA jest możliwość tworzenia własnych funkcji arkuszowych, czyli funkcji, które wyglądają i działają jak standardowe, wbudowane funkcje Excela. *Własne funkcje* mają ogromną, bezdyskusyjną zaletę — działają dokładnie w taki sposób, jak jest Ci to potrzebne (ponieważ to właśnie Ty jesteś ich autorem). Podstawowe zagadnienia związane z tworzeniem własnych funkcji omówiłem już w rozdziale 5., a w tym rozdziale będę się zajmować sztuczkami i trikami, które pozwolą Ci na jeszcze bardziej efektywne pisanie własnych funkcji.

Dlaczego tworzymy własne funkcje?

Z pewnością jesteś zaznajomiony z funkcjami arkuszowymi Excela. Nawet poczynający wiedzą, jak posługiwać się najczęściej stosowanymi funkcjami, takimi jak SUMA, ŚREDNIA czy JEŻELI. Excel 2013 posiada ponad 450 predefiniowanych funkcji arkuszowych, których możesz używać w formułach. Jeżeli jednak to nie wystarczy, możesz przy użyciu VBA tworzyć własne funkcje.

Mając do dyspozycji wszystkie funkcje Excela i języka VBA, można się zastanawiać, po co w ogóle tworzyć nowe funkcje. Odpowiedź jest prosta: aby ułatwić sobie życie. Dobrze zaprojektowane, niestandardowe funkcje są bardzo przydatne w formułach arkuszy i procedurach języka VBA. Przykładowo własne funkcje mogą bardzo często znacząco uprościć tworzone formuły, a krótsze formuły są czytelniejsze i łatwiej z nich korzystać.

Czego nie można zrobić za pomocą własnych funkcji arkuszowych?

Jeżeli tworzysz własne funkcje arkuszowe, musisz zrozumieć kluczową różnicę pomiędzy funkcjami wywoływanymi z innych procedur języka VBA a funkcjami stosowanymi w formułach arkusza. Funkcje arkuszowe w formułach muszą być *pasywne*, co oznacza, że na przykład nie mogą manipulować zakresami komórek, zmieniać formatowania ani robić wielu innych rzeczy, które można wykonywać za pomocą procedur Sub. Spróbuję to zilustrować na poniższym przykładzie.

Załóżmy, że chcesz utworzyć własną funkcję arkuszową, która w oparciu o wartość komórki zmienia kolor zawartego w niej tekstu. Możesz próbować różnych sposobów, ale z pewnością Ci się nie uda — po prostu napisanie takiej funkcji nie jest możliwe i efektem takich prób będzie zawsze komunikat o wystąpieniu błędu.

Zapamiętaj — funkcje wykorzystywane w formułach arkuszowych po prostu zwracają obliczoną wartość i nie wykonują żadnych operacji na obiektach.

Warto tutaj jednak wspomnieć o kilku wyjątkach. Poniżej przedstawiam kod funkcji, która zmienia treść komentarza przypisanego do komórki.

```
Function ChangeComment(cell, NewText)
    cell.Comment.Text = NewText
End Function
```

A oto przykład zastosowania takiej funkcji w formule. Funkcja działa tylko wtedy, kiedy do komórki A1 został przypisany komentarz. W momencie kiedy taka formula jest obliczana, komentarz zostanie zmieniony.

```
=ChangeComment(A1;"Hej. właśnie zmieniłem  
→komentarz!")
```

Nie jestem pewny, czy jest to zaplanowana właściwość Excela, czy też po prostu błąd implementacji, ale niewątpliwie jest to rzadki przykład funkcji VBA, która może zmienić coś w arkuszu.

Podstawowe informacje o funkcjach VBA

Funkcja języka VBA to rodzaj procedury, która jest przechowywana w module kodu VBA. Funkcji można używać w innych procedurach VBA lub w formułach arkusza. Własne funkcje nie mogą być tworzone za pomocą rejestratora makr, aczkolwiek rejestrator makr może być pomocny w identyfikacji odpowiednich właściwości i metod.

Moduł kodu może zawierać dowolną liczbę funkcji. Własnych funkcji arkuszowych możesz używać w formułach arkusza tak, jak innych, wbudowanych funkcji Excela. Jeżeli funkcja jest zdefiniowana w innym skoroszycie, przy wywołaniu musisz poprzedzić nazwę funkcji nazwą skoroszytu. Założymy na przykład, że utworzyłeś funkcję o nazwie DiscountPrice, która pobiera jeden argument i jest przechowywana w skoroszycie o nazwie Promocje_i_rabaty.xlsm.

Aby użyć tej funkcji w skoroszycie Promocje_i_rabaty.xlsm, powinieneś wpisać następującą formułę:

```
=DiscountPrice(A1)
```

Jeżeli jednak będziesz chciał użyć tej funkcji w innym skoroszycie, powinieneś wprowadzić formułę przedstawioną poniżej (i upewnić się, że skoroszyt Promocje_i_rabaty.xlsm jest otwarty).

```
=Promocje_i_rabaty.xlsm!DiscountPrice(A1)
```



Jeżeli własna, niestandardowa funkcja arkuszowa jest przechowywana w dodatku *add-in*, nie musisz poprzedzać jej nazwy nazwą skoroszytu. Więcej szczegółowych informacji na temat dodatków *add-in* znajdziesz w rozdziale 21.

Niestandardowe funkcje arkuszowe pojawiają się na liście dostępnych funkcji w oknie *Wstawianie funkcji*, w kategorii *Zdefiniowane przez użytkownika*. Jednym ze sposobów przywołania tego okna na ekran jest naciśnięcie kombinacji klawiszy *Shift+F3*.

Tworzenie funkcji

Pamiętaj, że nazwa funkcji zachowuje się jak zmienna. Końcową wartością tej zmiennej jest wartość zwracana przez funkcję. Aby się o tym przekonać, powinieneś dokładnie przeanalizować funkcję przedstawioną poniżej, która zwraca imię użytkownika.

```
Function FirstName()
    Dim FullName As String
    Dim FirstSpace As Integer
    FullName = Application.UserName
    FirstSpace = InStr(FullName, " ")
    If FirstSpace = 0 Then
        FirstName = FullName
    Else
        FirstName = Left(FullName, FirstSpace - 1)
    End If
End Function
```

Funkcja rozpoczyna działanie od przypisania właściwości *UserName* obiektu *Application* do zmiennej *FullName*. Następnie program wykorzystuje funkcję *InStr* języka VBA do zlokalizowania pozycji pierwszej spacji w nazwie użytkownika. Jeżeli nazwa użytkownika nie zawiera spacji, zmienna *FirstSpace* przyjmuje wartość 0, a zmiennej *FirstName* jest przypisywana pełna nazwa użytkownika. Jeżeli nazwa użytkownika *zawiera* spacje, za pomocą funkcji *Left* wycinane są z niej wszystkie znaki znajdujące się na lewo od pierwszej spacji i przypisywane do zmiennej *FirstName*.

Zwróć uwagę na fakt, że ciąg znaków *FirstName* jest nazwą funkcji i jednocześnie jest wykorzystywany jako nazwa zmiennej *w kodzie* funkcji. Końcowa wartość zmiennej *FirstName* to wartość zwracana przez funkcję. Funkcja może realizować wiele różnych obliczeń i wyznaczać różne wartości pośrednie, ale zawsze zwraca wartość, która jako ostatnia została przypisana do zmiennej reprezentującej nazwę funkcji.



Wszystkie przykłady omawiane w tym rozdziale znajdziesz w skoroszytach na stronie internetowej tej książki.

Praca z argumentami funkcji

Aby efektywnie korzystać z funkcji, musisz wiedzieć, jak pracować z ich argumentami wywołania. Argumenty to, inaczej mówiąc, informacje przekazywane do funkcji, które później są wykorzystywane przez tę funkcję do realizacji zadań przez nią spełnianych.

Zasady przedstawione poniżej odnoszą się zarówno do funkcji arkuszowych Excela, jak i do własnych funkcji VBA.

- ✓ Argumentami funkcji mogą być odwołania do komórek, zmienne (w tym tablice), stałe, literaly lub wyrażenia.
- ✓ Niektóre funkcje nie posiadają argumentów.
- ✓ Niektóre funkcje posiadają stałą liczbę wymaganych argumentów.
- ✓ Funkcje mogą posiadać zarówno wymagane, jak i opcjonalne argumenty.

Przykłady funkcji

Przykłady zaprezentowane w tym podrozdziale są ilustracją sposobów pracy z różnymi typami argumentów wywołania funkcji.

Funkcje bezargumentowe

Nie wszystkie funkcje wymagają podawania argumentów. Excel zawiera wiele wbudowanych funkcji, które nie posiadają żadnych argumentów wywołania, na przykład takich jak LOS(), DZIŚ() czy TERAZ().

Poniżej przedstawiam prosty przykład funkcji, która nie używa argumentów. Funkcja zwraca właściwość UserName obiektu Application. Wartość tej właściwości jest widoczna w polu *Nazwa użytkownika* na karcie Ogólne okna dialogowego *Opcje programu Excel*. Ten prosty, ale bardzo użyteczny przykład jest ilustracją jedynego sposobu na to, aby nazwa użytkownika Excela pojawiła się w komórce arkusza.

```
Function User()
    ' Zwraca nazwę aktualnego użytkownika programu Excel
    User = Application.UserName
End Function
```

Po wprowadzeniu poniższej formuły w komórce pojawi się nazwa aktualnego użytkownika programu Excel.

```
=User()
```

Podobnie jak w przypadku innych, wbudowanych funkcji Excela, kiedy używasz w formule arkusza funkcji bezargumentowej, musisz dołączyć do jej wywołania parę pustych nawiasów okrągłych. Inaczej Excel będzie próbował potraktować podaną nazwę funkcji jako zdefiniowaną nazwę zakresu komórek.

Funkcje jednoargumentowe

W tym punkcie omówię funkcję jednoargumentową, zaprojektowaną dla kierowników sprzedawy, którzy muszą obliczać prowizję uzyskiwaną przez podlegających im przedstawicieli handlowych. Wysokość prowizji zwiększa się w sposób nieliniowy i zależy od całkowitej

sprzedaży miesięcznej, stąd pracownicy osiągający większą sprzedaż otrzymują wyższe prowizje. Funkcja zwraca wysokość prowizji obliczoną na podstawie sprzedaży miesięcznej, która jest podawana jako jedyny wymagany argument wywołania funkcji. Obliczenia prowizji wykonywane przez naszą funkcję są oparte na danych zamieszczonych w tabeli 20.1.

Tabela 20.1. Wysokość prowizji w zależności od wartości sprzedaży

Miesięczna sprzedaż	Stawka prowizji
0 – 9999 złotych	8,0%
10 000 – 19 999 złotych	10,5%
20 000 – 39 999 złotych	12,0%
powyżej 40 000 złotych	14,0%

Istnieje co najmniej kilka metod obliczenia prowizji dla różnych wielkości sprzedaży. Jeżeli jednak nie przyłożyłeś się do opracowania dobrego rozwiązania, możesz niepotrzebnie stracić masę czasu, a co gorsza, w efekcie utworzyć niezrozumiałą, złożoną formułę, na przykład taką:

```
=JEŻELI(ORAZ(A1>=0;A1<=9999,99);A1*0,08;  
JEŻELI(ORAZ(A1>=10000;A1<=19999,99);A1*0,105;  
JEŻELI(ORAZ(A1>=20000;A1<=39999,99);A1*0,12;  
JEŻELI(A1>=40000;A1*0,14)))
```

Z kilku powodów nie jest to właściwe podejście do zagadnienia. Po pierwsze, formuła jest zbyt złożona, przez co trudno ją zrozumieć. Po drugie, wartości są na stałe wprowadzone do formuły, zatem trudno ją zmodyfikować.

Lepszą metodą (która nie wykorzystuje języka VBA) jest zastosowanie do obliczenia prowizji funkcji **WYSZUKAJ.PIONOWO** przeszukującej tabelę.

```
=WYSZUKAJ.PIONOWO(A1;Table;2)*A1
```

Innym sposobem, eliminującym konieczność użycia tabeli prowizji, może być utworzenie własnej, niestandardowej funkcji.

```
Function Commission(Sales)  
    Const Tier1 = 0.08  
    Const Tier2 = 0.105  
    Const Tier3 = 0.12  
    Const Tier4 = 0.14  
    ' Obliczanie prowizji od sprzedaży  
    Select Case Sales  
        Case 0 To 9999.99: Commission = Sales * Tier1  
        Case 10000 To 19999.99: Commission = Sales * Tier2  
        Case 20000 To 39999.99: Commission = Sales * Tier3  
        Case Is >= 40000: Commission = Sales * Tier4  
    End Select  
End Function
```

Zwróć uwagę na fakt, że stawki prowizji są zadeklarowane w naszej funkcji jako stałe, a nie zostały wpisane na sztywno w kodzie. Dzięki takiemu rozwiążaniu można bardzo łatwo modyfikować działanie funkcji w sytuacji, kiedy zmienią się wartości prowizji.

Kiedy umieścisz funkcję w module VBA, możesz jej użyć w formule arkusza lub wywołać z innych procedur języka VBA. Gdy wprowadzisz do komórki arkusza formułę przedstawioną poniżej, wynikiem jej działania będzie wartość 3000, ponieważ sprzedaż o wartości 25000 pozwala uzyskać prowizję wynoszącą 12%.

```
=Commission(25000)
```

Na rysunku 20.1 przedstawiam wygląd arkusza, w którym funkcja Commission została użyta w formułach w kolumnie C.

A	B	C	D
Nazwisko	Sprzedaż	Provizja	
1 Adamczyk	61 983,00 zł	8 677,62 zł	
2 Baran	3 506,00 zł	280,48 zł	
3 Grabowska	38 973,00 zł	4 676,76 zł	
4 Jabłorski	32 092,00 zł	3 851,04 zł	
5 Kowalski	27 354,00 zł	3 282,48 zł	
7 Lewandows	17 833,00 zł	1 872,46 zł	
8 Malinowski	41 598,00 zł	5 823,72 zł	
9 Michałski	32 000,00 zł	3 840,00 zł	
10 Nowak	5 000,00 zł	400,00 zł	
11 Olszewski	68 793,00 zł	9 631,02 zł	
12 Pawlak	31 093,00 zł	3 731,16 zł	
13 Sikora	24 509,00 zł	2 941,08 zł	
14 Zieliński	41 544,00 zł	5 816,16 zł	
15			

Rysunek 20.1.
Zastosowanie
funkcji Com-
mission w for-
mulach arkusza

Funkcje z dwoma argumentami

Dla ułatwienia podaję tu kolejny przykład będący kontynuacją poprzedniego. Wyobraź sobie, że wspomniani wcześniej hipotetyczni kierownicy sprzedaży z poprzedniego przykładu wprowadzili nową zasadę pozwalającą zwiększyć obroty. Polega ona na tym, że całkowita prowizja jest zwiększana o jeden procent po każdym kolejnym roku przepracowanym przez przedstawiciela w firmie.

Dla naszych potrzeb zmodyfikowałem funkcję Commission zdefiniowaną w poprzednim punkcie tak, aby pobierała dwa obowiązkowe argumenty. Nowy argument reprezentuje liczbę lat pracy. Nową funkcję nazwałem Commission2.

```
Function Commission2(Sales, Years)
    ' Oblicza prowizje od sprzedaży w oparciu o lata pracy
    Const Tier1 = 0.08
    Const Tier2 = 0.105
    Const Tier3 = 0.12
    Const Tier4 = 0.14
    Select Case Sales
        Case 0 To 9999.99: Commission2 = Sales * Tier1
        Case 10000 To 19999.99: Commission2 = Sales * Tier2
    End Select
End Function
```

```
Case 20000 To 39999.99: Commission2 = Sales * Tier3  
Case Is >= 40000: Commission2 = Sales * Tier4  
End Select  
Commission2 = Commission2 + (Commission2 * Years / 100)  
End Function
```

W deklaracji funkcji dodajemy jedynie drugi argument (`Years`) oraz dołączamy do kodu funkcji polecenia odpowiednio modyfikujące sposób obliczania prowizji. W tej wersji oryginalna wartość prowizji jest mnożona przez liczbę lat pracy sprzedawcy i następnie dzielona przez 100, a uzyskana w ten sposób wartość jest dodawana do oryginalnej wielkości prowizji.

Poniżej zamieszczam przykład formuły używającej funkcji `Commission2` (zakładam, że wartość sprzedaży znajduje się w komórce A1, natomiast liczba lat przepracowanych przez przedstawiciela w komórce B1).

```
=Commission2(A1, B1)
```

Na rysunku 20.2 przedstawiam wygląd arkusza wykorzystującego w formułach funkcję `Commission2`.

A	B	C	D	E
19 Nazwisko	Sprzedaż	Lata	Prowizja v2	
20 Adamczyk	61 983,00 zł	4	9 024,72 zł	
21 Baran	3 506,00 zł	3	288,89 zł	
22 Grabowska	38 973,00 zł	2	4 770,30 zł	
23 Jabłoński	32 092,00 zł	1	3 889,55 zł	
24 Kowalski	27 354,00 zł	8	3 545,08 zł	
25 Lewandowski	17 833,00 zł	3	1 928,64 zł	
26 Malinowski	41 598,00 zł	2	5 940,19 zł	
27 Michałski	32 000,00 zł	1	3 878,40 zł	
28 Nowak	5 000,00 zł	1	404,00 zł	
29 Olszewski	68 793,00 zł	4	10 016,26 zł	
30 Pawłak	31 093,00 zł	2	3 805,78 zł	
31 Sikora	24 509,00 zł	2	2 999,90 zł	
32 Zieliński	41 544,00 zł	3	5 990,64 zł	
33				

Rysunek 20.2.
Zastosowanie funkcji `Commission2` pobierającej dwa argumenty wywołania

Funkcje pobierające zakres jako argument

Używanie zakresu komórek jako argumentu wywołania funkcji nie jest wcale takie trudne — po prostu Excel zajmuje się wszystkimi potrzebnymi szczegółami technicznymi.

Poniżej przedstawiam prostą, ale bardzo użyteczną funkcję, która łączy ciągi znaków zapisane w komórkach danego zakresu. Funkcja pobiera dwa argumenty — `InRange` (zakres komórek, których zawartość ma zostać połączona) oraz `Delim` (jeden lub więcej znaków separujących, które będą wstawiane pomiędzy zawartość poszczególnych komórek).

```
Function JoinText(InRange, Delim)  
    Dim Cell As Range  
    For Each Cell In InRange  
        JoinText = JoinText & Cell.Value & Delim
```

```
Next Cell  
JoinText = Left(JoinText, Len(JoinText) - Len(Delim))  
End Function
```

Funkcja wykorzystuje pętlę For Each-Next do przechodzenia przez kolejne komórki zakresu i łączy ciągi znaków z poszczególnych komórek, wstawiając pomiędzy kolejne ciągi znaki separujące. Ostatnie polecenie funkcji usuwa końcowe znaki separujące.

Na rysunku 20.3 przedstawiam przykład zastosowania tej funkcji. Drugi argument funkcji to dwuznakowy ciąg znaków (przecinek i spacja).

	A	B	C	D	E	F	G	H
1	Styczeń							
2	Luty							
3	Marzec		Styczeń, Luty, Marzec, Kwiecień, Maj, Czerwiec					
4	Kwiecień							
5	Maj							
6	Czerwiec							
7								
8								
9								
10								
11								
12								
13								
14								
15								
...								

Rysunek 20.3.
Zastosowanie
funkcji JoinText
do łączenia
zawartości
komórek

Poniżej przedstawiam kolejny przykład funkcji wykorzystującej zakres komórek jako argument wywołania. Założmy, że chcesz obliczyć średnią pięciu największych liczb w zakresie komórek o nazwie *Dane*. Excel nie zawiera funkcji, która realizuje takie zadanie, stąd prawdopodobnie będziesz musiał napisać na przykład taką formułę.

```
=MAX.K(Dane;1)+MAX.K(Dane;2)+MAX.K(Dane;3)+  
MAX.K(Dane;4)+MAX.K(Dane;5))/5
```

Przedstawiona powyżej formuła wykorzystuje funkcję arkuszową MAX.K, która zwraca n -tą największą liczbę z zakresu komórek. Formuła dodaje pięć największych liczb w zakresie komórek o nazwie *Dane* i następnie dzieli wynik przez 5. Formuła działa dobrze, ale jest raczej nieporęczna. A co się stanie, jeżeli zmienisz zdanie i będziesz chciał policzyć średnią sześciu największych liczb? Niestety, w takiej sytuacji będziesz musiał, po pierwsze, poprawić formułę, a po drugie, upewnić się, że poprawiłeś wszystkie kopie tej formuły w całym arkuszu albo i skoroszycie.

A czy nie byłoby dużo łatwiej, gdyby Excel posiadał funkcję arkuszową o nazwie TopAvg? W takiej sytuacji mógłbyś obliczać taką średnią za pomocą następującej formuły (oczywiście, funkcja TopAvg na razie jeszcze nie istnieje).

```
=TopAvg(Dane,5)
```

Ten przykład dobrze ilustruje sytuację, w której własna, niestandardowa funkcja arkuszowa może znakomicie ułatwić życie. Funkcja TopAvg, której kod przedstawiam poniżej, zwraca średnią arytmetyczną N największych liczb w danym zakresie komórek.

```
Function TopAvg(InRange, N)
    ' Zwraca średnią arytmetyczną N największych liczb w zakresie InRange
    Dim Sum As Double
    Dim i As Long
    Sum = 0
    For i = 1 To N
        Sum = Sum + WorksheetFunction.Large(InRange, i)
    Next i
    TopAvg = Sum / N
End Function
```

Funkcja pobiera dwa argumenty — InRange (reprezentujący zakres komórek arkusza) oraz N (liczba liczb, których średnią będziemy liczyć). Funkcja rozpoczyna działanie od przypisania zmiennej Sum wartości 0. Następnie program używa pętli For-Next do obliczenia sumy N największych liczb z zakresu InRange . Zwróć uwagę, że w ciele pętli wykorzystywana jest funkcja Large (odpowiednik funkcji arkuszowej MAX.K). Wreszcie na koniec suma liczb jest dzielona przez wartość argumentu N i wynik przypisywany jest zmiennej TopAvg .



W swoich procedurach VBA możesz używać niemal wszystkich funkcji arkuszowych Excela, z wyjątkiem tych, które mają swoje odpowiedniki w języku VBA. Przykładowo język VBA posiada funkcję RND, która zwraca liczbę losową, stąd w swoich makrach nie możesz używać funkcji arkuszowej RAND.

Funkcje z argumentami opcjonalnymi

Wiele wbudowanych funkcji arkuszowych Excela używa argumentów opcjonalnych. Przykładem jest funkcja LEWY, która zwraca znaki, począwszy od lewej strony łańcucha. Jej oficjalna składnia wygląda następująco.

```
LEWY(tekst:[liczba_znaków])
```

Pierwszy argument jest wymagany, natomiast drugi jest już opcjonalny (stąd w składni funkcji zapisujemy go w nawiasach kwadratowych). Jeżeli argument opcjonalny zostanie pominięty, Excel użyje domyślnej wartości 1, zatem poniższe dwie formuły zwrócią taki sam wynik.

```
=LEWY(A1;1)  
=LEWY(A1)
```

Niestandardowe funkcje tworzone w języku VBA też mogą mieć argumenty opcjonalne. Aby zdefiniować taki argument, powinieneś przed jego nazwą dodać słowo kluczowe Optional. Po nazwie argumentu powinieneś wpisać znak równości i podać domyślną wartość argumentu. Jeżeli argument opcjonalny zostanie pominięty, VBA automatycznie użyje wartości domyślnej.

Jedna istotna uwaga: jeżeli korzystasz z argumentów opcjonalnych, ich definicje na liście argumentów wywołania funkcji muszą znaleźć się za argumentami, których podanie jest obowiązkowe.

Poniżej przedstawiam przykład niestandardowej funkcji, która wykorzystuje argument opcjonalny.

```
Function DrawOne(InRange, Optional Recalc = 0)
    ' Losowo wybiera z zakresu jedną komórkę

    Randomize
    ' Jeżeli Recalc = 1, funkcja będzie ulotna
    If Recalc = 1 Then Application.Volatile True

    ' Wybierz losową komórkę
    DrawOne = InRange(Int((InRange.Count) * Rnd + 1))
End Function
```

Funkcja losowo wybiera jedną komórkę z zakresu komórek podanego jako argument wywołania. Zakres przekazywany do funkcji jest w zasadzie tablicą (więcej szczegółowych informacji na temat tablic znajdziesz w rozdziale 7.), a funkcja po prostu losowo wybiera jeden element z tej tablicy. Jeżeli drugi argument wywołania funkcji ma wartość 1, wybrana komórka zmienia się za każdym razem, kiedy arkusz jest przeliczany (funkcja ma charakter *ulotny*). Jeżeli drugi argument wywołania ma wartość 0 (lub zostanie pominięty), funkcja nie jest ponownie przeliczana dopóty, dopóki nie zmieni się zawartość jednej z komórek zakresu wejściowego funkcji.

Wykrywanie i usuwanie błędów w funkcjach niestandardowych

Wykrywanie i usuwanie błędów w funkcjach niestandardowych może być nieco większym wyzwaniem niż w przypadku zwykłej procedury Sub. Kiedy będziesz tworzyć własne funkcje arkuszowe, przekonasz się, że ewentualne błędy w kodzie funkcji powodują tylko pojawienie się kodu błędu w komórce arkusza zawierającej formułę z Twoją funkcją (zazwyczaj Excel zwraca wtedy błąd #WARTOŚĆ!). Innymi słowy, w takiej sytuacji nie otrzymujesz normalnego komunikatu o wystąpieniu błędu wykonania, który pomaga w lokalizowaniu nieprawidłowego polecenia.

Na szczęście, zawsze masz do dyspozycji co najmniej trzy metody wykrywania błędów we własnych funkcjach arkuszowych. Oto one.

- ✓ Umieszczenie funkcji MsgBox w strategicznych miejscach kodu w celu monitorowania wartości określonych zmiennych. Okna komunikatów wywoływane z poziomu kodu funkcji pojawiają się na ekranie w normalny sposób. Powinieneś

się jednak upewnić, że w arkuszu znajduje się tylko jedna formuła używająca testowanej funkcji. W przeciwnym razie okna komunikatów będą się pojawiały dla każdej przetwarzanej formuły, co szybko może się okazać bardzo irytujące.

- ✓ Testowanie procedury poprzez wywołanie jej z procedury Sub, a nie z formuły arkuszowej. W takiej sytuacji błędy wykonania będą wyświetlane w normalny sposób, dzięki czemu będziesz mógł szybko usunąć błąd (jeżeli wiesz, jak to zrobić) lub użyć debuggera do bardziej szczegółowej analizy problemu.
- ✓ Ustawienie w kodzie funkcji pułapki (punktu przerwania) i wykonanie programu w trybie krokowym za pomocą debuggera. Po wykonaniu tej operacji dostępne będą wszystkie standardowe narzędzia wspomagające wykrywanie i usuwanie błędów. Więcej szczegółowych informacji na temat debuggera znajdziesz w rozdziale 13.

Zwróć uwagę, że w kodzie funkcji użyte zostało polecenie Randomize, które ma na celu inicjację generatora liczb pseudolosowych inną wartością za każdym razem, kiedy skoroszyt jest otwierany. Bez tego polecenia po każdym otwarciu skoroszytu funkcja zwracałaby zawsze takie same, „losowe” wartości.

Przedstawionej funkcji możesz używać do generowania szczęśliwych numerów losów w loterii, wybierania zwycięzcy z listy potencjalnych kandydatów i tak dalej.

Funkcje opakowujące

W tym podrozdziale przedstawię kilka względnie prostych, własnych funkcji arkuszowych, które mogą być bardzo użyteczne. Takie funkcje noszą nazwę *funkcji opakowujących (wrapper functions)*, ponieważ zawierają kod „opakowujący” rzeczywiste wywołania funkcji języka VBA. Innymi słowy, funkcje opakowujące pozwalają na wykorzystywanie funkcji VBA w formułach arkuszowych.

Wcześniej w tym rozdziale omówiona została następująca funkcja opakowująca.

```
Function User()
    ' Zwraca nazwę aktualnego użytkownika programu Excel
    User = Application.UserName
End Function
```

Funkcja pozwala Twoim formułom arkuszowym na dostęp do właściwości UserName obiektu Application.

Dalej w tym podrozdziale znajdziesz kilka innych, bardzo użytecznych funkcji opakowujących.

Funkcja NumberFormat

Funkcja NumberFormat po prostu wyświetla w komórce opis formatu numerycznego komórki, której adres jest argumentem wywołania funkcji. Taka funkcja może być bardzo użyteczna w sytuacji, kiedy musisz się upewnić, że wszystkim komórkom danej grupy przypisano taki sam sposób formatowania liczb.

```
Function NumberFormat(Cell)
    ' Zwraca opis formatu numerycznego komórki
    NumberFormat = Cell(1).NumberFormatLocal
End Function
```

Zwróć uwagę na wyrażenie Cell(1). Oznacza to, że jeżeli argumentem wywołania funkcji będzie zakres komórek, funkcja użycie tylko adresu pierwszej komórki.

Jeśli trzeba, możesz bardzo łatwo napisać podobne funkcje, które będą zwracały kolor tekstu w komórce, kolor tła komórki, użytą czcionkę i tak dalej.

Funkcja ExtractElement

Jest to jedna z moich ulubionych funkcji opakowujących. Funkcja ExtractElement zwraca *n*-ty element ciągu znaków, składającego się z wielu elementów oddzielonych znakiem separatora. Przykładowo formuła arkuszowa przedstawiona poniżej zwraca literał *krowa*, będący trzecim elementem łańcucha znaków, którego znakiem separatora jest spacja. Oczywiście, argumentami wywołania funkcji mogą być odwołania do komórek arkusza.

```
=ExtractElement("pies koń krowa kot"; 3; " ")
```

Poniżej zamieszczam kod tej funkcji, który spełnia rolę „opakowania” dla funkcji Split języka VBA.

```
Function ExtractElement(Txt, n, Sep)
    ' Zwraca n-ty element łańcucha tekstowego, w którym poszczególne elementy
    ' oddzielone są określonym znakiem separatora
    ExtractElement = Split(Application.Trim(Txt), Sep)(n - 1)
End Function
```

Na rysunku 20.4 przedstawiam przykłady zastosowania funkcji ExtractElement w formułach arkuszowych.

	A	B	C	D	E	F	G
1	123-45-78	2	-	45			
2	a b c d e f g	3		c			
3	a b c	d	3	c			
4	Jan Kowalski	1		Jan			
5	Jan Kowalski	2		Kowalski			
6	Jan Kowalski	3		#ARG!			
7	55/98/44/23	3	/	44			
8	1,2,3,4,5,6,7,8,9,10	5	,	5			
9	98-74-872-9823-23	3	--	872			
10							
11							

Rysunek 20.4.
Zastosowanie
funkcji
ExtractElement
do wybierania
elementów
z ciągu znaków

Funkcja SayIt

Funkcja SayIt jest prostą funkcją opakowującą dla metody Speak obiektu Application.Speech. Funkcja wykorzystuje wbudowany syntezator mowy do odczytywania „na głos” przekazanego jej argumentu.

```
Function SayIt(txt)
    ' Odczytuje argument wywołania za pomocą syntezatora mowy
    Application.Speech.Speak txt, True
End Function
```

Poniżej przedstawiam przykład zastosowania tej funkcji w formule arkuszowej.

```
=JEŻELI(C10>10000;SayIt("Przekroczyłeś budżet");"OK")
```

Formuła sprawdza zawartość komórki o adresie C10. Jeżeli jej wartość jest większa niż 10 000, funkcja „wypowiada” komunikat *Przekroczyłeś budżet*. Jeżeli wartość komórki jest mniejsza niż 10 000, w komórce wyświetlany jest tekst *OK* (i funkcja nic nie „mówi”).

Pamiętaj, że funkcji tej powinieneś używać rozsądnie. Jeżeli wykorzystasz ją w arkuszu w więcej niż jednej komórce, może się to okazać irytujące. Pamiętaj również, że funkcja będzie wywoływana za każdym razem, kiedy arkusz będzie przeliczany, więc taki „gadający” Excel może być naprawdę nieznośny, kiedy wykonujesz wiele zmian w komórkach arkusza. W praktyce taką funkcję będzie prawdopodobnie częściej służyła do wywarcia wrażenia na kolegach niż do zastosowań „produkcyjnych”.

Funkcja IsLike

Operator Like języka VBA zapewnia bardzo elastyczną metodę porównywania łańcuchów tekstu. Więcej szczegółowych informacji na ten temat znajdziesz w pomocy systemowej języka VBA. Funkcja przedstawiona poniżej właściwie odgrywa rolę funkcji opakowującej, pozwalającej na wygodne użycie wszechstronnego operatora Like w formułach arkuszowych.

```
Function IsLike(text, pattern)
    ' Zwraca wartość True, jeżeli pierwszy argument jest podobny do drugiego
    IsLike = text Like pattern
End Function
```

Funkcje zwracające tablice

Formuły tablicowe są jednym z najużyteczniejszych mechanizmów Excela. Jeżeli potrafisz posługiwać się formułami tablicowymi, z pewnością ucieczy Cię wiadomość, że możesz utworzyć funkcję VBA, która zwraca tablicę.

Zwracanie tablicy zawierającej nazwy miesięcy

Rozpocznę od przedstawienia prostego przykładu. Funkcja MonthNames zwraca 12-elementową tablicę zawierającą — jak sama nazwa funkcji wskazuje — nazwy miesięcy.

```
Function MonthNames()
    MonthNames = Array("Styczeń", "Luty", "Marzec", _
                       "Kwiecień", "Maj", "Czerwiec", "Lipiec", "Sierpień", _
                       "Wrzesień", "Październik", "Listopad", "Grudzień")
End Function
```

356 Część V: Od teorii do praktyki

Aby zastosować funkcję MonthNames w skoroszycie, musisz wprowadzić ją jako 12-komórkową formułę tablicową. Aby to zrobić, zaznacz na przykład zakres A2:L2, wpisz formułę =MonthNames() i następnie naciśnij kombinację klawiszy *Ctrl+Shift+Enter*, co spowoduje wpisanie formuły we wszystkich 12 zaznaczonych komórkach. Wynik takiej operacji został przedstawiony na rysunku 20.5.

Rysunek 20.5.
Funkcja
MonthNames
zwraca
12-elementową
tablicę zawie-
rającą nazwy
miesiący

The screenshot shows a Microsoft Excel interface. The formula bar at the top contains the formula '=monthnames()'. The spreadsheet area shows a table with 12 columns labeled A through L. Row 1 has headers A through M. Row 2 contains the values 'Styczeń' through 'Grudzień' respectively. Below the table, the ribbon menu is visible with tabs like NumberFormat, ExtractElement, Sayit, MonthNames (which is highlighted in green), Sorted, and a plus sign icon.

A	B	C	D	E	F	G	H	I	J	K	L	M
1	Styczeń	Luty	Marzec	Kwiecień	Maj	Czerwiec	Lipiec	Sierpień	Wrzesień	Październik	Listopad	Grudzień
2												
3												
4												
5												
6												
7												

Jeżeli chcesz, aby nazwy miesiąca były wyświetlane pionowo, w kolumnie, zaznacz 12 komórek wybranej kolumny i wpisz następującą formułę tablicową (nie zapomnij zakończyć jej naciśnięciem kombinacji klawiszy *Ctrl+Shift+Enter*).

```
=TRANSPONUJ(MonthNames())
```

Jeśli trzeba, możesz wybrać z tablicy pojedynczy element. Poniżej przedstawiona została formuła arkuszowa (ale nie tablicowa), która wyświetla czwarty element tablicy zwracanej przez funkcję MonthNames, czyli w tym przypadku *Kwiecień*.

```
=INDEKS(MonthNames();4)
```

Zwracanie posortowanej listy

Załóżmy, że masz listę imion, którą chcesz wyświetlić w posortowanej postaci w innym zakresie komórek. Czy nie byłoby fajnie, gdyby istniała funkcja arkuszowa mogąca wykonać taką operację za Ciebie?

Niestandardowa funkcja arkuszowa omawiana w tym punkcie realizuje właśnie takie zadanie — pobiera jako argument wywołania jednokolumnowy zakres komórek, następnie sortuje zawartość komórek i zwraca tablicę posortowanych komórek. Na rysunku 20.6 przedstawiam przykład zastosowania takiej funkcji. Zakres komórek A2:A13 zawiera listę imion. Zakres komórek C2:C13 zawiera wielokomórkową formułę tablicową (pamiętaj, wpisując formułę tablicową, musisz zakończyć ją naciśnięciem kombinacji klawiszy *Ctrl+Shift+Enter*).

```
=Sorted(A2:A13)
```

Rysunek 20.6.
Zastosowanie własnej funkcji arkuszowej do zwracania posortowanego zakresu komórek

A	B	C	D	E	F	G
1 Nieposortowane		Posortowane				
2 Jakub		Adam				
3 Kacper		Bartosz				
4 Szymon		Dawid				
5 Mateusz		Filip				
6 Filip		Jakub				
7 Michał		Kacper				
8 Bartosz		Maciej				
9 Wiktor		Mateusz				
10 Piotr		Michał				
11 Dawid		Piotr				
12 Adam		Szymon				
13 Maciej		Wiktor				
14						

Poniżej zamieszczam kod funkcji Sorted.

```
Function Sorted(Rng)
    Dim SortedData() As Variant
    Dim Cell As Range
    Dim Temp As Variant, i As Long, j As Long
    Dim NonEmpty As Long

    ' Przenoszenie danych do tablicy SortedData
    For Each Cell In Rng
        If Not IsEmpty(Cell) Then
            NonEmpty = NonEmpty + 1
            ReDim Preserve SortedData(1 To NonEmpty)
            SortedData(NonEmpty) = Cell.Value
        End If
    Next Cell

    ' Sortowanie tablicy
    For i = 1 To NonEmpty
        For j = i + 1 To NonEmpty
            If SortedData(i) > SortedData(j) Then
                Temp = SortedData(j)
                SortedData(j) = SortedData(i)
                SortedData(i) = Temp
            End If
        Next j
    Next i

    ' Transpozycja i zwracanie tablicy
    Sorted = Application.Transpose(SortedData)
End Function
```

Funkcja Sorted rozpoczyna działanie od utworzenia tablicy o nazwie SortedData, która zawiera wszystkie niezerowe wartości z zakresu komórek będącego argumentem wywołania funkcji. Następnie zawartość tej tablicy jest sortowana przy użyciu algorytmu sortowania bąbelkowego. Ponieważ tablica SortedData jest pozioma, przed zwróceniem jako wynik działania funkcji zostaje transponowana do postaci pionowej.

Funkcja Sorted może działać z zakresami o dowolnym rozmiarze dopóty, dopóki mają postać pojedynczego wiersza lub pojedynczej kolumny. Jeżeli nieposortowane dane znajdują się w wierszu, do wyświetlania posortowanych wyników w postaci wiersza powinieneś użyć funkcji arkuszowej TRANSPONUJ, co zostało przedstawione w poniższym przykładzie.

```
=TRANSPONUJ(Sorted(A16:L16))
```

Okno dialogowe Wstawianie funkcji

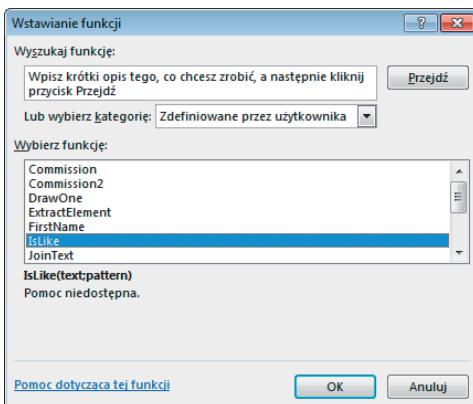
Okno dialogowe Excela *Wstawianie funkcji* jest bardzo wygodnym narzędziem, które pozwala na wybieranie funkcji arkuszowej z listy i łatwe zdefiniowanie odpowiednich argumentów wywołania funkcji. W oknie dialogowym *Wstawianie funkcji* znajdziesz również niestandardowe funkcje arkuszowe, które domyślnie wyświetlane są w kategorii *Zdefiniowane przez użytkownika*.



Funkcje zdefiniowane z dyrektywą Private (funkcje prywatne) nie pojawiają się na liście okna dialogowego *Wstawianie funkcji*. Jeżeli tworzona funkcja ma być dostępna wyłącznie dla innych procedur i funkcji języka VBA (a nie będzie używana w formułach), powinieneś przy jej deklarowaniu użyć słowa kluczowego Private.

Wyświetlanie opisów funkcji

W oknie dialogowym *Wstawianie funkcji* wyświetlane są opisy wszystkich wbudowanych funkcji Excela. Jednak w przypadku funkcji niestandardowych domyślnie w miejscu opisu funkcji wyświetlany jest lapidarny komunikat *Pomoc niedostępna*, co przedstawiam na rysunku 20.7.

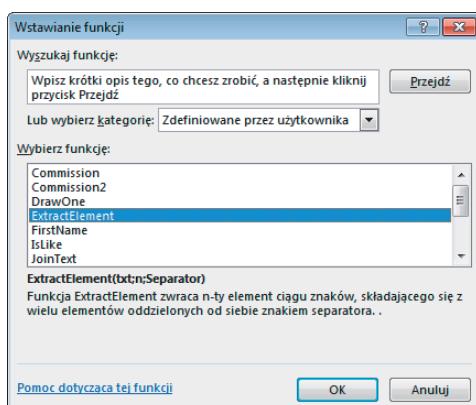


Rysunek 20.7.
Domyślnie
w oknie Wsta-
wianie funkcji
nie pojawiają
się opisy funkcji
niestandardo-
wych

Aby utworzyć opis działania własnej funkcji niestandardowej, który będzie wyświetlany w oknie dialogowym *Wstawianie funkcji*, powinieneś wykonać polecenia przedstawione poniżej (choć mogą Ci się wydawać niezbyt intuicyjne).

1. Przejdź do okna skoroszytu zawierającego niestandardową funkcję, której opis chcesz utworzyć.
2. Przejdź na kartę **DEVELOPER** i naciśnij przycisk **Makra**, znajdujący się w grupie opcji **Kod** (zamiast tego możesz nacisnąć kombinację klawiszy **Alt+F8**).
- Na ekranie pojawi się okno dialogowe *Makro*.
3. W polu **Nazwa makra** wpisz nazwę funkcji.
Zwróć uwagę na fakt, że Twojej funkcji nie ma na liście dostępnych makr, stąd musisz wpisać jej nazwę ręcznie.
4. Naciśnij przycisk **Opcje**.
Na ekranie pojawi się okno dialogowe *Opje makra*.
5. W polu **Opis** wpisz krótki opis funkcji.
6. Naciśnij przycisk **OK**.
7. Naciśnij przycisk **Anuluj**.

Od tej chwili w oknie dialogowym *Wstawianie funkcji* będzie wyświetlany nowo utworzony opis Twojej funkcji, co przedstawiam na rysunku 20.8.



Rysunek 20.8.
Opis własnej
funkcji wyświet-
lony w oknie
Wstawianie
funkcji



Funkcje niestandardowe domyślnie są przypisywane do kategorii *Zdefiniowane przez użytkownika* i w niej wyświetlane. Aby dodać wybraną funkcję do innej kategorii, musisz skorzystać z języka VBA. Polecenie przedstawione poniżej powoduje dodanie funkcji TopAvg do kategorii *Matematyczne* (czyli kategorii o numerze 3).

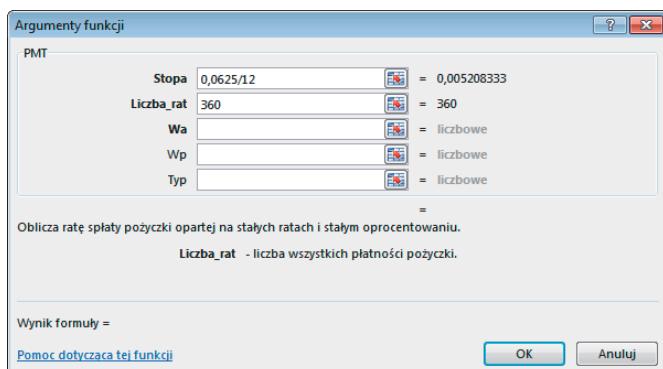
```
Application.MacroOptions Macro:="TopAvg", Category:=3
```

Więcej szczegółowych informacji na temat numerów kategorii funkcji znajdziesz w pomocy systemowej Excela. Pamiętaj, polecenie, takie jak to przedstawione powyżej, musisz wykonać tylko raz. Po jego wykonaniu (i zapisaniu skoroszytu) nowy numer kategorii zostanie permanentnie przypisany do danej funkcji.

Opisy argumentów

Kiedy w oknie *Wstawianie funkcji* wybierzesz żądaną funkcję, na ekranie pojawi się okno dialogowe *Argumenty funkcji*, w którym wyświetlane są opisy poszczególnych argumentów wywołania funkcji (zobacz rysunek 20.9). Kiedyś definiowanie opisów argumentów własnych funkcji nie było możliwe, wreszcie w wersji Excel 2010 firma Microsoft zaimplementowała ten mechanizm. Definiowanie opisów argumentów funkcji odbywa się za pomocą metody *MacroOptions*. Poniżej przedstawiam przykład kodu VBA, który dodaje opis argumentów dla funkcji *TopAvg*.

Rysunek 20.9.
Domyślnie
w oknie Argu-
menty funkcji
wyświetlane są
tylko opisy ar-
gumentów
wbudowanych
funkcji Excela



```
Sub AddArgumentDescriptions()
    Application.MacroOptions Macro:="TopAvg",
    ArgumentDescriptions:=
        Array("Zakres komórek zawierający wartości liczbowe",
              "Liczba wartości do obliczenia wartości średniej")
End Sub
```

Procedurę przedstawioną powyżej musisz uruchomić tylko raz. Po zakończeniu jej działania opisy argumentów zostaną zapisane w skoroszycie i na stałe powiązane z funkcją.

Zwróć uwagę na fakt, że opisy argumentów są podawane w kodzie procedury jako argumenty funkcji *Array*. Musisz zawsze użyć tej funkcji, nawet w sytuacji, kiedy tworzysz opis dla funkcji posiadającej tylko jeden argument.

W tym rozdziale omówiłem wiele zagadnień związanych z tworzeniem niestandardowych funkcji arkuszowych. Przykładów przedstawionych w tym rozdziale możesz używać jako wzorców do tworzenia własnych funkcji. Jak zawsze, pamiętaj, że w pomocy systemowej znajdziesz wiele interesujących i przydatnych informacji. Jeżeli chcesz się dowiedzieć, jak zwiększyć dostępność swoich funkcji poprzez umieszczenie ich w dodatku *add-in*, powinieneś zajrzeć do rozdziału 21.

Rozdział 21

Tworzenie dodatków

W tym rozdziale:

- ▶ zrozumiesz, czym są dodatki i dlaczego są tak ważne,
- ▶ dowiesz się, dlaczego powinieneś tworzyć własne dodatki,
- ▶ poznasz metody tworzenia dodatków.

Jedną z najciekawszych właściwości Excela, która najbardziej przydaje się programistom, jest możliwość tworzenia dodatków (*add-in*). W tym rozdziale postaram się objąć, dlaczego dodatki są takim ciekawym mechanizmem, i pokazać, jak możesz tworzyć dodatki za pomocą wbudowanych mechanizmów Excela.

No dobrze... czym zatem są dodatki?

Ogólnie rzecz biorąc, dodatki są mechanizmami, które wzbogacają arkusze o nowe właściwości funkcjonalne. Niektóre dodatki oferują szereg nowych funkcji arkuszowych, których możesz używać w swoich formułach, inne dostarczają nowych poleceń czy narzędzi. Jeżeli dodatek jest prawidłowo zaprojektowany, nowe mechanizmy znakomicie integrują się z oryginalnym interfejsem Excela tak, że wydają się być częścią programu.



Excel „na dzień dobry” wyposażony jest w kilka dodatków, takich jak *Analysis ToolPak* czy *Solver*. Bardzo wiele innych dodatków oferują różni dostawcy w internecie. Mój pakiet *Power Utility Pak* jest przykładem takiego właśnie dodatku (zawiera ponad 70 nowych narzędzi dla Excela, nie mówiąc już o szeregu nowych funkcji arkuszowych).

Praktycznie każdy użytkownik Excela może utworzyć dodatek, aczkolwiek wymagane do tego jest posiadanie pewnych umiejętności programowania w języku VBA. Dodatki Excela są w zasadzie inną formą skoroszytów XLSM, a różnią się od nich następującymi cechami.

- ✓ Właściwość `IsAddin` obiektu `Workbook` ma wartość *True*.
- ✓ Okno skoroszytu jest ukryte w taki sposób, że nie można go odkryć za pomocą polecenia *Odkryj* znajdującego się w grupie opcji *Okno* na karcie *WIDOK*.
- ✓ Dodatki nie należą do kolekcji `Workbooks` — zamiast tego należą do kolekcji `Addins`.

Teoretycznie na dodatek można przekształcić dowolny skoroszyt, ale w praktyce nie każdy skoroszyt dobrze się do tego nadaje. Ponieważ dodatki są zawsze ukryte, nie możesz wyświetlić arkuszy roboczych czy arkuszy wykresów zawartych w dodatku. Masz

jednak dostęp do funkcji i procedur przechowywanych w dodatku, możesz też wyświetlać okna dialogowe przechowywane w obiektach UserForm dodatków.



Dodatki Excela są zwykle zapisywane w plikach z rozszerzeniem XLAM, co pozwala je odróżnić od zwykłych skoroszytów XLSM. We wcześniejszych wersjach Excela (przed wersją 2007) dodatki zapisywane były w plikach z rozszerzeniem XLA.

Po co tworzy się dodatki?

Decyzyję o konwersji Twojej aplikacji Excela do postaci dodatku możesz podjąć ze względu na jeden lub kilka następujących powodów. Oto one.

- ✓ **Ograniczenie dostępu do kodu i arkuszy.** Kiedy rozpowszechniamy aplikację w formie dodatku i zabezpieczymy projekt VBA hasłem, użytkownicy nie będą mogli przeglądać lub modyfikować arkuszy ani kodu VBA, który jest z nimi związany. Jeżeli w aplikacji używasz własnych rozwiązań, przekształcenie jej na postać dodatku zapobiega kopiowaniu kodu przez innych użytkowników, a przynajmniej mocno utrudnia do niego dostęp. Pamiętaj jednak, że mechanizmy zabezpieczeń oferowane przez Excel nie są idealne, a poza tym istnieje wiele programów pozwalających na łamanie takich haseł.
- ✓ **Zapobieganie pomyłkom.** Jeżeli użytkownik załaduje aplikację jako dodatek, nie będą widoczne jego arkusze. Dzięki temu istnieje mniejsze prawdopodobieństwo wprowadzenia w błąd początkujących użytkowników. Dodatku, w odróżnieniu od ukrytego skoroszytu, nie można odkryć.
- ✓ **Uproszczenie dostępu do funkcji arkuszowych.** Funkcje arkuszowe definiowane w dodatku nie wymagają stosowania kwalifikatora nazwy skoroszytu. Jeżeli na przykład zapiszemy funkcję użytkownika o nazwie MOVAVG w skoroszycie Newfuncs.xlsm, użycie tej funkcji w innym skoroszycie będzie wymagało zastosowania instrukcji o następującej składni.

```
=Newfuncs.xlsm!MOVAVG(A1:A50)
```

Jeśli natomiast funkcja zostanie zapisana w dodatku, możesz skorzystać z instrukcji o prostszej składni, w której nie trzeba używać odwołań do pliku.

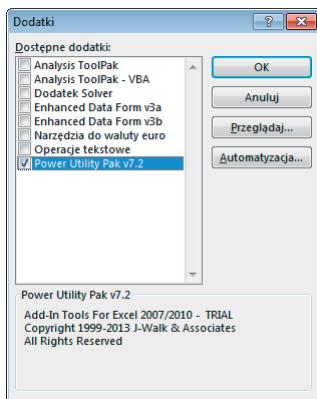
```
=MOVAVG(A1:A50)
```

- ✓ **Ułatwienie użytkownikom dostępu do właściwości aplikacji.** Po zidentyfikowaniu położenia dodatek zostanie umieszczony w oknie dialogowym *Dodatki* wraz z przyjazną nazwą i opisem działania. Użytkownik może łatwo włączyć lub wyłączyć wybrany dodatek.
- ✓ **Uzyskanie lepszej kontroli nad ładowaniem aplikacji.** Dodatki można otwierać automatycznie przy uruchomieniu Excela, niezależnie od katalogu, w którym zostały zapisane.
- ✓ **Uniknięcie wyświetlania pytań systemowych podczas zamknięcia.** Podczas zamknięcia dodatku nie jest wyświetlone okno dialogowe z pytaniem o to, czy chcesz zapisać zmiany.

Praca z dodatkami

Najlepszym sposobem ładowania i zamknięcia dodatków jest wykorzystanie okna dialogowego *Dodatki* w Excelu. Aby przywołać to okno na ekran, przejdź na kartę *PLIK* i wybierz z menu polecenie *Opcje*. Na ekranie pojawi się okno dialogowe *Opcje programu Excel*. Kliknij kategorię *Dodatki*, a następnie z listy rozwijanej *Zarządzaj* wybierz opcję *Dodatki programu Excel* i naciśnij przycisk *Przejdź*. Zamiast tego możesz skorzystać z innej metody: przejdź na kartę *DEVELOPER* i wybierz polecenie *Dodatki*, znajdujące się w grupie opcji *Dodatki*. Pamiętaj, że domyślnie karta *DEVELOPER* nie jest wyświetlana na Wstążce. Najszybszą metodą przywołania okna *Dodatki* na ekran jest po prostu naciśnięcie sekwencji klawiszy *Alt*, *Q*, *X*.

Niezależnie od tego, której metody użyjesz, na ekranie pojawi się okno dialogowe *Dodatki*, przedstawione na rysunku 21.1. Na liście *Dostępne dodatki* wyświetlane są nazwy wszystkich dostępnych dla użytkownika dodatków Excela. Zaznaczone pola wyboru obok nazwy dodatku oznaczają dodatki, które aktualnie są otwarte. Poprzez zaznaczanie i usuwanie zaznaczenia pól wyboru możesz otwierać i zamykać odpowiednie dodatki.



Rysunek 21.1.
W oknie dialogowym Dodatki wyświetlana jest lista wszystkich dostępnych dodatków

Aby dodać do listy nowy dodatek, naciśnij przycisk *Przeglądaj* i następnie odszukaj właściwy plik XLAM.



Większość dodatków możesz także otwierać za pomocą polecenia *Otwórz* z karty *PLIK* (tak jak inne, normalne skoroszyty). Jednak dodatek otwarty w taki sposób nie pojawi się na liście w oknie *Dodatki*. Co więcej, jeżeli otworzysz dodatek w taki sposób, nie będziesz mógł go zamknąć za pomocą polecenia *PLIK/Zamknij*. Jedynym sposobem usunięcia takiego dodatku będzie zamknięcie i ponowne uruchomienie Excela lub napisanie odpowiedniego makra VBA.

Otwarcie dodatku czasami nie powoduje żadnych widocznych zmian w samym Excelu, ale niemal zawsze zmienia w pewien sposób interfejs użytkownika: na Wstążce pojawia się nowe polecenie czy jedno lub kilka nowych poleceń zostaje dodanych do menu podręcznego. Przykładowo otwarcie dodatku *Analysis ToolPak* powoduje, że na karcie *DANE*, w grupie opcji *Narzędzia danych* pojawia się nowy przycisk o nazwie *Analiza warunkowa*. Jeżeli w dodatku znajdują się jedynie nowe funkcje użytkownika, pojawią się one w oknie dialogowym *Wstawianie funkcji*, a na Wstążce nie zauważysz żadnych zmian.

Podstawy tworzenia dodatków

Jak już wspominałem wcześniej, teoretycznie każdy skoroszyt można przekształcić w dodatek, ale w praktyce nie wszystkie skoroszyty się do tego nadają. Skoroszyt, który chcesz przekształcić w dodatek, musi zawierać makra (w przeciwnym przypadku taki dodatek będzie po prostu bezużyteczny). Idealnym kandydatem do konwersji na dodatek jest skoroszyt zawierający makra ogólnego przeznaczenia (zarówno funkcje, jak i procedury).

Tworzenie dodatków nie jest trudne, ale wymaga nieco dodatkowej pracy. Aby przekształcić standardowy skoroszyt na dodatek, powinieneś wykonać następujące polecenia.

1. Utwórz aplikację i sprawdź, czy wszystko działa poprawnie.

Nie zapomnij, że powinieneś udostępnić użytkownikowi metodę uruchamiania makra lub makr znajdujących się w dodatku. Możesz to zrobić, definiując na przykład odpowiedni klawisz skrótu lub modyfikując interfejs Excela (zobacz rozdział 19.). Jeżeli dodatek zawiera wyłącznie funkcje, nie musisz zwracać sobie główny metodami uruchamiania, ponieważ funkcje będą widoczne na liście w oknie dialogowym *Wstawianie funkcji*.

2. Przetestuj działanie aplikacji poprzez uruchomienie jej, kiedy aktywny jest inny skoroszyt.

Wykonanie takiego testu symuluje działanie aplikacji w sytuacji, kiedy będzie wykorzystywana jako dodatek, ponieważ dodatek nigdy nie jest aktywnym skoroszytem.

3. Aktywuj edytor VBE i zaznacz skoroszyt w oknie projektu. Z menu głównego wybierz polecenie *Tools/VBAProject Properties* i w oknie dialogowym, które pojawi się na ekranie, przejdź na kartę *Protection*. Zaznacz opcję *Lock project for viewing* i dwukrotnie wpisz hasło. Po zakończeniu naciśnij przycisk *OK*.

Ten krok będzie niezbędny tylko w sytuacji, kiedy chcesz uniemożliwić innym użytkownikom przeglądanie lub modyfikację kodu VBA czy formularzy *UserForm* dodatku.

4. Przejdz na kartę *DEVELOPER* i naciśnij przycisk *Panel dokumentu*, znajdujący się w grupie opcji *Modyfikowanie*. Na ekranie pojawi się okno dialogowe *Panel informacji o dokumencie*. Naciśnij przycisk *OK*, aby otworzyć panel *Właściwości dokumentu*.

Excel wyświetli poniżej Wstążki panel *Właściwości dokumentu*.

5. W panelu *Właściwości dokumentu* wpisz w polu *Tytuł* krótką nazwę dodatku, a w polu *Komentarze* podaj dłuższy opis przeznaczenia dodatku.

Polecenia przedstawione w punktach 4. i 5. nie są wymagane, ale powodują, że dodatki są łatwiejsze w użyciu, ponieważ opisy utworzone w panelu *Właściwości dokumentu* pojawiają się później po wybraniu dodatku w oknie dialogowym *Dodatki*.

6. Przejdz na kartę *PLIK* i wybierz polecenie *Zapisz jako*.

7. W oknie dialogowym *Zapisywanie jako* rozwiń listę *Zapisz jako typ* i wybierz z niej opcję *Dodatek programu Excel (*.xlam)*.

8. Wybierz folder, w którym chcesz zapisać dodatek.

Excel zaproponuje użycie domyślnego miejsca przechowywania dodatków (folder *Dodatki*), ale możesz zapisać swój dodatek w dowolnym innym folderze.

9. Naciśnij przycisk *Zapisz*.

Kopia Twojego skoroszytu zostanie poddana konwersji do postaci dodatku i zapisana na dysku z rozszerzeniem XLAM. Oryginalny skoroszyt pozostanie otwarty.

Tworzymy przykładowy dodatek

W tym podrozdziale omówię szereg czynności, jakie powinieneś wykonać, aby utworzyć nowy dodatek programu Excel. W przykładzie wykorzystam narzędzie *Zmień wielkość liter*, które opisałem w rozdziale 16.



Skoroszyt z tym przykładem znajdziesz na stronie internetowej tej książki. Opisywany dodatek możesz utworzyć na podstawie tego skoroszytu.

Konfiguracja skoroszytu

Przykładowy skoroszyt składa się z jednego, pustego arkusza, jednego modułu VBA oraz formularza *UserForm*. W rozdziale 19. opisywałem, w jaki sposób możesz utworzyć kod VBA, który dodaje nowe elementy menu podręcznego, jakie pojawi się po kliknięciu komórki prawym przyciskiem myszy.

Oryginalna wersja narzędzia *Zmień wielkość liter*, którą opisałem w jednym z wcześniejszych rozdziałów, posiadała trzy opcje zmiany wielkości liter: *WIELKIE LITERY*, *małe litery* oraz *Jak W Nazwie Własnej*. Na potrzeby dodatku utworzyłem jeszcze dwie nowe opcje, dzięki czemu narzędzie posiada takie same możliwości zmiany wielkości liter jak edytor Microsoft Word.

- ✓ **Opcja *Jak w zdaniu*** zmienia pierwszą literę zdania na wielką i wszystkie pozostałe litery w zdaniu na małe.
- ✓ **Opcja *zAMIANA na małe/WIELKIE*** powoduje, że wszystkie wielkie litery w zaznaczonym tekście są zamieniane na małe i odwrotnie.

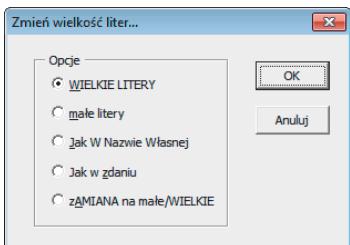
Na rysunku 21.2 przedstawiam wygląd formularza *UserForm*. Wszystkie przyciski opcji (formanty *OptionButton*) zostały umieszczone w kontenerze *Frame*. Dodatkowo na formularzu znajduje się przycisk *OK* (o nazwie *OKButton*) oraz przycisk *Anuluj* (o nazwie *CancelButton*).

Makro wykonywane po naciśnięciu przycisku *Anuluj* jest bardzo proste. Procedura przedstawiona poniżej powoduje usunięcie formularza *UserForm* z pamięci i nie wykonuje żadnych innych operacji.

```
Private Sub CancelButton_Click()
    Unload UserForm1
End Sub
```

366 Część V: Od teorii do praktyki

Rysunek 21.2.
Formularz
UserForm na-
rzędzia Zmień
wielkość liter



Kod procedury wykonywanej po naciśnięciu przycisku *OK* został przedstawiony poniżej.

```
Private Sub OKButton_Click()
    Dim TextCells As Range
    Dim cell As Range
    Dim Text As String
    Dim i As Long

    ' Utwórz obiekt zawierający stałe tekstowe
    On Error Resume Next
    Set TextCells = Selection.SpecialCells(xlConstants, _
        xlTextValues)

    ' Wyłącz aktualizowanie ekranu
    Application.ScreenUpdating = False

    ' Przeglądaj kolejne komórki (w pętli)
    For Each cell In TextCells
        Text = cell.Value
        Select Case True
            Case OptionLower 'male litery'
                cell.Value = LCase(cell.Value)
            Case OptionUpper 'WIELKIE LITERY'
                cell.Value = UCase(cell.Value)
            Case OptionProper 'Jak W Nazwie Własnej'
                cell.Value = _
                    Application.WorksheetFunction.Proper(cell.Value)
            Case OptionSentence 'Jak w zdaniu'
                Text = UCase(Left(cell.Value, 1))
                Text = Text & LCase(Mid(cell.Value, 2, Len(cell.Value)))
                cell.Value = Text
            Case OptionToggle 'zAMIEN na male/WIELKIE'
                For i = 1 To Len(Text)
                    If Mid(Text, i, 1) Like "[A-Z]" Then
                        Mid(Text, i, 1) = LCase(Mid(Text, i, 1))
                    Else
                        Mid(Text, i, 1) = UCase(Mid(Text, i, 1))
                    End If
                Next i
                cell.Value = Text
        End Select
    Next
End Sub
```

```
' Usuń okno dialogowe
Unload UserForm1
End Sub
```

Oprócz wspomnianych wcześniej dwóch nowych opcji, obecna wersja narzędzia *Zmień wielkość liter* różni się od wersji przedstawionej w rozdziale 16. jeszcze dwoma kwestiami.

- ✓ Program wykorzystuje metodę `SpecialCells` do utworzenia zmiennej obiektowej, składającej się z komórek zlokalizowanych w zaznaczonym obszarze, w których znajduje się czysty tekst (a nie formuły). Zastosowanie takiej techniki powoduje, że procedura działa nieco szybciej w sytuacji, kiedy w zaznaczonym obszarze znajduje się wiele komórek zawierających formuły. Więcej szczegółowych informacji na temat tej techniki znajdziesz w rozdziale 14.
- ✓ Opcja *Zmień wielkość liter* została również dodana do menu podręcznego, które pojawia się na ekranie po kliknięciu prawym przyciskiem myszy wiersza i kolumny. Dzięki temu możesz skorzystać z narzędzia po kliknięciu prawym przyciskiem myszy nie tylko zaznaczonego obszaru komórek, ale również dla całych zaznaczonych wierszy i dla całych zaznaczonych kolumn.

Testowanie skoroszytu

Przed konwersją skoroszytu na dodatek powinieneś go gruntownie przetestować. Aby zasymulować sposób działania dodatku, powinieneś testować swój skoroszyt w sytuacji, kiedy aktywny będzie inny skoroszyt. Pamiętaj, dodatek nigdy nie będzie aktywnym skoroszytem, stąd testowanie w opisanej wyżej sytuacji pozwoli Ci łatwiej zidentyfikować i poprawić potencjalne błędy.

1. Otwórz nowy skoroszyt i wpisz do kilku komórek dowolne informacje.

Na potrzeby takiego testu powinieneś umieścić w co najmniej kilku komórkach różne rodzaje danych, włącznie z „czystym” tekstem, wartościami liczbowymi i formułami. Zamiast tego możesz — oczywiście — otworzyć dowolny, istniejący skoroszyt i wykorzystać go do testowania naszego narzędzia — pamiętaj jednak, że zmiany wykonane za pomocą tego narzędzia nie mogą być cofnięte, więc bezpieczniej będzie, jeżeli skorzystasz z kopii istniejącego skoroszytu.

2. **Zaznacz jedną lub więcej komórek (zamiast tego możesz również zaznaczyć kilka wierszy lub kolumn).**
3. **Uruchom makro `ChangeCase`, wybierając polecenie *Zmień wielkość liter...* z menu podręcznego, które pojawi się po kliknięciu zaznaczonego obszaru prawym przyciskiem myszy.**

Jeżeli polecenie *Zmień wielkość liter...* nie pojawi się w menu podręcznym, najbardziej prawdopodobną przyczyną problemu będzie to, że podczas otwierania skoroszytu *Zmień wielkość liter.xlsx* nie zezwoliłeś na uruchamianie makr. W takiej sytuacji zamknij skoroszyt i następnie otwórz go ponownie, tym razem upewniając się, że zezwoliłeś na wykonywanie makr.

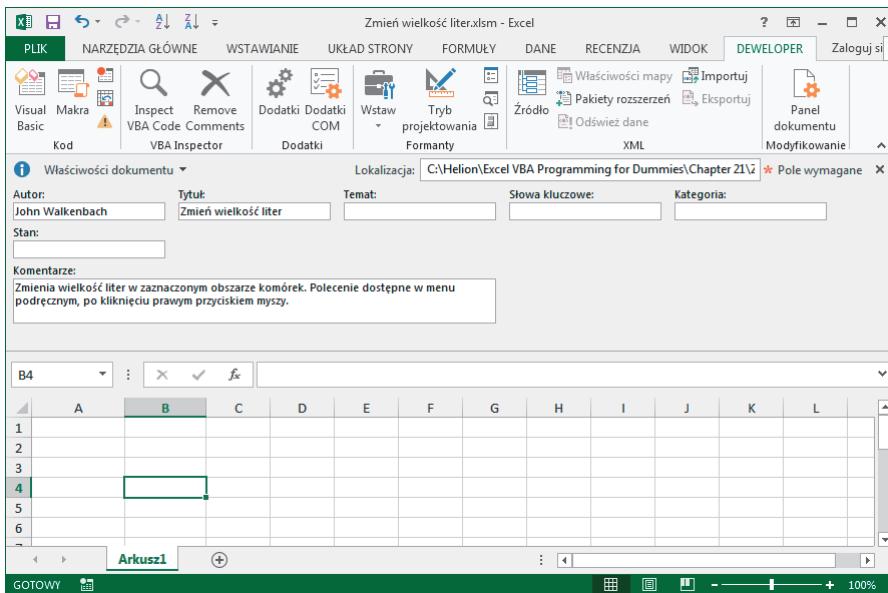


Tworzenie opisów dodatku

Utworzenie opisu przeznaczenia dodatku jest bardzo dobrą praktyką, jednak w żaden sposób nie jest to wymagane. Aby to zrobić, wykonaj polecenia przedstawione poniżej.

1. Przejdź do okna skoroszytu *Zmień wielkość liter.xlsx*.
2. Przejdź na kartę **DEVELOPER** i wybierz polecenie **Panel dokumentu**, znajdujące się w grupie opcji **Modyfikowanie**.

Excel wyświetli na ekranie, poniżej paska formuły, panel *Właściwości dokumentu* (zobacz rysunek 21.3).



Rysunek 21.3.
Opis dodatku
możesz wprowadzić w
panelu *Właściwości dokumentu*

3. W polu **Tytuł** wpisz nazwę Twojego dodatku.

Tekst wpisany w tym polu pojawi się jako nazwa dodatku na liście w oknie dialogowym *Dodatki*. W naszym przypadku wpisz *Zmień wielkość liter*.

4. W polu **Komentarze** wprowadź opis dodatku.

Tekst wpisany w tym polu pojawi się w dolnej części okna dialogowego *Dodatki* po zaznaczeniu nazwy naszego dodatku na liście. Wpisz następujący tekst: *Zmienia wielkość liter w zaznaczonym obszarze komórek. Polecenie dostępne w menu podręcznym, po kliknięciu prawym przyciskiem myszy.*

5. Naciśnij przycisk **Zamknij** , znajdujący się w prawym, górnym rogu panelu *Właściwości dokumentu*.

Ochrona kodu VBA

Jeżeli chcesz zabezpieczyć dostęp do kodu VBA hasłem, tak aby inni użytkownicy nie mieli możliwości przeglądania kodu procedur zapisanych w dodatku, powinieneś wykonać polecenia przedstawione poniżej.

1. Przejdź do edytora VBE i w oknie *Project* zaznacz skoroszyt *Zmień wielkość liter.xlsxm*.
2. Z menu głównego edytora wybierz polecenie *Tools/VBAProject Properties* i w oknie dialogowym, które pojawi się na ekranie, przejdź na kartę *Protection*.
3. Zaznacz opcję *Lock project for viewing* i następnie w polu *Password* wpisz hasło zabezpieczające. Powtórz hasło w polu *Confirm password*.
4. Naciśnij przycisk **OK**.
5. Zapisz skoroszyt, wybierając z menu głównego edytora VBE polecenie *File/Save*, lub powróć do okna skoroszytu w Excelu, przejdź na kartę *PLIK* i wybierz polecenie *Zapisz*.

Tworzenie dodatku

W tym momencie zakończyłeś procedurę testowania skoroszytu *Zmień wielkość liter.xlsxm* i upewniłeś się, że wszystko działa prawidłowo. Nadszedł zatem czas, aby utworzyć nasz dodatek. Aby to zrobić, wykonaj przedstawione niżej polecenia.

1. Jeżeli to konieczne, przejdź do okna Excela.
 2. Uaktywnij skoroszyt *Zmień wielkość liter.xlsxm*, przejdź na kartę *PLIK* i wybierz polecenie *Zapisz jako*.
- Na ekranie pojawi się okno dialogowe *Zapisywanie jako*.
3. Rozwiń listę *Zapisz jako typ* i wybierz z niej opcję *Dodatek programu Excel (*.xlam)*.
 4. Wybierz folder, w którym chcesz zapisać dodatek, i naciśnij przycisk *Zapisz*.

Kopia Twojego skoroszytu zostanie poddana konwersji do postaci dodatku i zapisana na dysku z rozszerzeniem XLAM. Oryginalny skoroszyt pozostanie otwarty.

Otwieranie dodatku

Aby uniknąć nieporozumień, powinieneś teraz zamknąć oryginalny skoroszyt XLSM, z którego utworzyłeś dodatek.

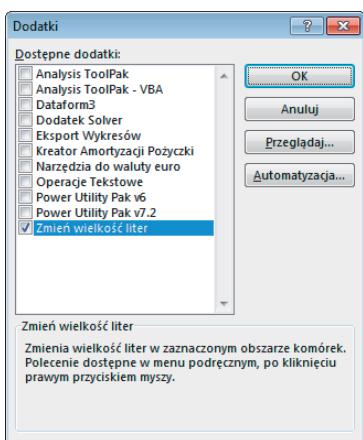
Aby otworzyć i załadować do Excela nowo utworzony dodatek, powinieneś wykonać polecenia opisane poniżej.

1. Przejdź na kartę *DEVELOPER* i naciśnij przycisk *Dodatki* znajdujący się w grupie opcji *Dodatki*. Zamiast tego możesz nacisnąć sekwencję klawiszy *Alt, Q, X*.

Na ekranie pojawi się okno dialogowe *Dodatki*.

2. Naciśnij przycisk *Przeglądaj*.
3. Odszukaj utworzony dodatek i zaznacz go.
4. Naciśnij przycisk *Otwórz*, aby zamknąć okno *Przegląda*.

Po odszukaniu i załadowaniu dodatku jego nazwa pojawi się na liście dostępnych dodatków w oknie *Dodatki*. Jak widać na rysunku 21.4, po zaznaczeniu dodatku w dolnej części okna dialogowego *Dodatki* pojawił się również opis, który wprowadziłeś w panelu *Właściwości dokumentu*.



Rysunek 21.4.
Nowy dodatek
pojawił się
w oknie dialo-
gowym Dodatki

5. Upewnij się, że pole wyboru obok nazwy Twojego dodatku jest zaznaczone.
6. Naciśnij przycisk **OK**, aby załadować dodatek i zamknąć okno dialogowe.

Excel załaduje dodatek i od tej chwili będziesz go mógł używać we wszystkich skoroszytach. Dopóki pole wyboru obok nazwy dodatku jest zaznaczone, Excel będzie automatycznie przy każdym uruchomieniu ładował Twój dodatek.

Dystrybucja dodatków

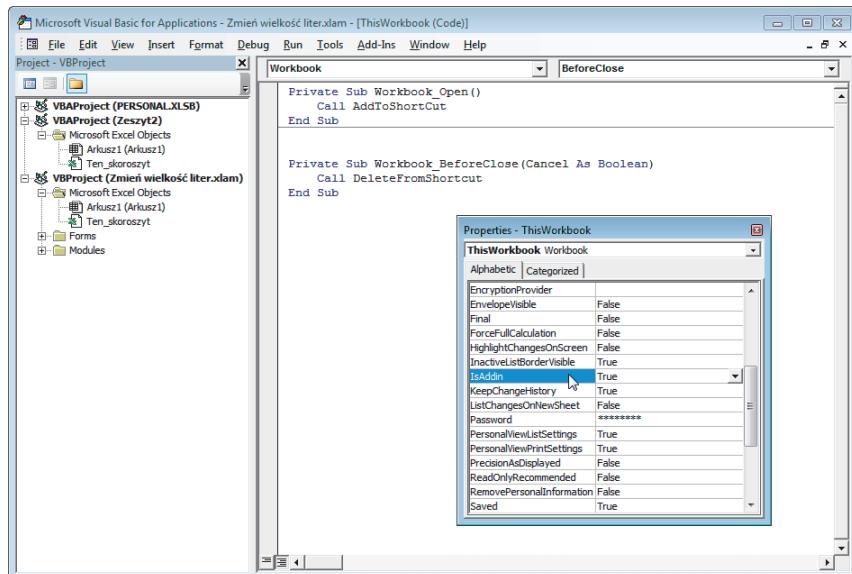
Jeżeli jesteś życzliwy, możesz udostępnić swój dodatek innym użytkownikom. Aby to zrobić, wystarczy przekazać użytkownikom kopię pliku *XLAM* wraz z instrukcją instalacji (skoroszyt *XLSM* nie jest im potrzebny). Kiedy obdarowani użytkownicy zainstalują dodatek w swoich Excelach, polecenie *Zmień wielkość liter* pojawi się w menu podręcznym, dostępnym po kliknięciu zaznaczonego obszaru prawym przyciskiem myszy. Jeśli zablokowałeś plik za pomocą hasła, użytkownicy, którzy nie znają hasła, nie będą mogli przeglądać ani modyfikować kodu VBA procedur zawartych w dodatku.

Modyfikowanie dodatków

Dodatki mogą być modyfikowane i edytowane, tak jak każdy inny skoroszyt. Pliki XLAM możesz modyfikować bezpośrednio (nie musisz pracować z oryginalną wersją dodatku w pliku XLSM). Aby to zrobić, wykonaj polecenia przedstawione poniżej.

1. Otwórz plik XLAM, o ile nie jest już otwarty.
2. Przejdź do edytora VBE.
3. Dwukrotnie kliknij nazwę projektu VBA w oknie **Project**.
4. Jeżeli to konieczne, wpisz hasło i naciśnij przycisk **OK**.
5. Wykonaj niezbędne zmiany w dodatku.
6. Zapisz plik z poziomu edytora VBE, wybierając z menu głównego polecenie **File/Save**.

Jeżeli Twój dodatek przechowuje różne informacje w swoich arkuszach, to aby zobaczyć zawartość takich arkuszy w Excelu, powinieneś ustawić właściwość **IsAddIn** dodatku na wartość *False*. Możesz to zrobić w oknie *Properties* po zaznaczeniu obiektu *Ten_skoroszyt* (zobacz rysunek 21.5). Po wprowadzeniu zmian, a przed zapisaniem pliku, musisz ponownie ustawić właściwość **IsAddIn** na wartość *True*.



Rysunek 21.5.
Zamiana dodatku
na normalny
skoroszyt

Teraz już wiesz, jak pracować z dodatkami i dlaczego warto je tworzyć.

Część VI

Dekalogi



Dekalogi
Dekalogi

W tej części...

- ✓ poznasz odpowiedzi na najczęściej zadawane pytania dotyczące języka VBA,
- ✓ poznasz dziesięć źródeł, w których znajdziesz ogromne zasoby wiedzy na temat VBA,
- ✓ dowiesz się, co możesz, a czego nie powinieneś robić w języku VBA.

Rozdział 22

Dziesięć pytań na temat VBA (wraz z odpowiedziami)

W tym rozdziale:

- ▶ przeczytasz o przechowywaniu funkcji w skoroszytach,
- ▶ dowiesz się o ograniczeniach rejestratora makr,
- ▶ poznasz przyspieszanie wykonywania kodu VBA,
- ▶ dowiesz się, co to jest pełne deklarowanie zmiennych,
- ▶ poznasz zastosowanie znaku kontynuacji wiersza kodu VBA.

Pytnia i odpowiedzi, które znajdziesz w tym rozdziale, dotyczą zagadnień, o które najczęściej pytają użytkownicy stawiający swoje pierwsze kroki w programowaniu w języku VBA.

Utworzyłem własną funkcję VBA, która będzie używana w formułach jako funkcja arkuszowa. Niestety, próba jej użycia zawsze kończy się błędem #NAZWA? Co jest nie tak?

Możliwe, że po prostu niepoprawnie wpisałeś w formule nazwę funkcji. Ale jeszcze bardziej prawdopodobne jest to, że umieściłeś kod funkcji w niewłaściwym miejscu. Własne funkcje arkuszowe muszą być definiowane w standardowych modułach kodu VBA, a nie w module kodu arkusza (na przykład Arkusz1) lub w module kodu obiektu Ten_skorosztyt. W edytorze VBE wybierz z menu głównego polecenie *Insert/Module*, aby wstawić standardowy moduł kodu VBA, a następnie wytnij kod funkcji z niewłaściwego miejsca i wklej w nowo utworzonym module kodu VBA.

Jest to bardzo często spotykany błąd, ponieważ moduły kodu arkuszy roboczych wyglądają niemal tak samo jak moduły kodu VBA. Postaraj się oprzeć pokusie wstawienia kodu funkcji w modułach arkuszy — zamiast tego po prostu poświęć kilka sekund i za pomocą polecenia *Insert/Module* wstaw do Twojego skoroszytu nowy, standardowy moduł kodu VBA.

Czy rejestratora makr VBA można użyć do rejestrowania wszystkich rodzajów makr?

Nie. Właściwie rejestrator makr nadaje się tylko do bardzo prostych makr lub do makr, które będą wykorzystywane jako baza do tworzenia bardziej złożonych procedur. Takich makr, w których wykorzystuje się zmienne, pętle oraz inne mechanizmy modyfikacji przepływu sterowania, nie można zarejestrować. Oprócz tego, rejestrator makr nie pozwala na rejestrowanie funkcji.

Wielu użytkowników wykorzystuje rejestrator makr do identyfikacji fragmentów kodu oraz odpowiednich właściwości i metod obiektów związanych z realizowanym zadaniem.

W jaki sposób mogę zabezpieczyć kod VBA moich programów przed przeglądaniem go przez innych użytkowników?

1. Przejdź do edytora VBE i z menu głównego wybierz polecenie *Tools/xxxx Properties*.

Zamiast ciągu znaków xxxx w oknie *Properties* znajdziesz nazwę swojego projektu.

2. Na ekranie pojawi się okno właściwości projektu VBA. Przejdź na kartę *Protection* i zaznacz opcję *Lock project for viewing*.
3. W polu *Password* wpisz hasło i ponownie wpisz to samo hasło w polu *Confirm password*. Po zakończeniu naciśnij przycisk **OK**.
4. Zapisz skoroszyt.

Takie rozwiązanie skutecznie uniemożliwi dostęp do kodu większości użytkowników. Najnowsze wersje Excela mają znacząco poprawione i ulepszone mechanizmy zabezpieczeń, ale pamiętaj, że hasło zawsze można złamać za pomocą wielu różnych narzędzi.

Czy za pomocą kodu VBA można zwiększać lub zmniejszać liczbę wierszy i kolumn w skoroszycie?

Nie da się napisać takiego kodu. Liczba kolumn i wierszy arkusza jest stała i nie może być zmieniana. Jeżeli jednak otworzysz w nowym Excelu skoroszyt, który został utworzony za pomocą Excela starszego niż wersja 2007, na pasku tytułowym okna Excela, obok nazwy skoroszytu, pojawi się tekst *Tryb zgodności*. Oznacza to, że taki skoroszyt ma rozmiary ograniczone do 256 kolumn i 65 536 wierszy. Jeżeli potrzebny Ci jest arkusz o większych rozmiarach, możesz wyjść z trybu zgodności poprzez zapisanie takiego „starego” skoroszytu w nowym formacie (XLSX lub XLSM), a następnie zamknięcie i ponowne otwarcie takiego pliku.

Jeśli trzeba, możesz ukrywać niepotrzebne wiersze i kolumny. Przykładowo możesz ukryć wszystkie komórki arkusza z wyjątkiem pierwszych dziesięciu kolumn i wierszy, dzięki czemu w efekcie otrzymasz skoroszyt „składający się” ze 100 komórek.

Kiedy próbuję odwołać się do arkusza w kodzie VBA, uzyskuję komunikat o błędzie „Subscript out of range” (indeks poza zakresem). Ja przecież nie używam żadnych indeksów. O co tu chodzi?

Podobnie jak w przypadku wielu innych komunikatów o wystąpieniu błędu, również i ten nie daje zbyt wielu informacji. Taki błąd powstaje podczas próby dostępu do nieistniejącego obiektu kolekcji. Przykładowo wykonanie poniższej instrukcji spowoduje powstanie błędu w przypadku, gdy aktywny skoroszyt nie zawiera arkusza o nazwie MójArkusz.

```
Set X = ActiveWorkbook.Worksheets("MójArkusz")
```

Taki błąd może się również zdarzyć w sytuacji, kiedy wydaje Ci się, że dany skoroszyt jest otwarty, ale w rzeczywistości tak nie jest (zatem taki skoroszyt nie jest częścią kolekcji Workbooks), lub kiedy po prostu popełniłeś literówkę w nazwie skoroszytu.

Czy istnieje polecenie w języku VBA, które pozwala na zaznaczenie zakresu od aktywnej komórki do ostatniej zajętej komórki w wierszu lub kolumnie? Inaczej mówiąc, w jaki sposób można za pomocą makra osiągnąć taki sam rezultat jak za pomocą naciśnięcia kombinacji klawiszy *Ctrl+Shift+↓* lub *Ctrl+Shift+→*?

Poniżej przedstawiam odpowiednik VBA naciśnięcia kombinacji klawiszy *Ctrl+Shift+↓*.

```
Range(ActiveCell, ActiveCell.End(xlDown)).Select
```

Dla innych kierunków zaznaczania, zamiast stałej *xlDown*, powinieneś użyć stałych *xlToLeft*, *xlToRight* lub *xlUp*.

Jak mogę przyspieszyć działanie mojego kodu VBA?

Poniżej znajdziesz kilka wskazówek.

- ✓ Upewnij się, że wszystkie zmienne zostały zadeklarowane przy użyciu określonych typów danych. Aby wymusić deklarowanie wszystkich zmiennych, powinieneś w sekcji deklaracji zmiennych każdego z modułów kodu użyć dyrektywy *Option Explicit*.
- ✓ Jeżeli odwołujesz się do danego obiektu (takiego jak na przykład zakres komórek) więcej niż raz, powinieneś przy użyciu polecenia *Set* utworzyć odpowiednią zmienną obiektową.
- ✓ Kiedy to tylko możliwe, przy pracy z obiektami używaj konstrukcji *With-End With*.
- ✓ Jeżeli Twoje makro zapisuje dane w skoroszycie, w którym znajduje się wiele złożonych formuł, powinieneś na czas działania makra przełączyć Excel w tryb ręcznego przeliczania arkusza (ale upewnij się, że kiedy musisz użyć wyników działania makra, arkusz zostanie przeliczony!).
- ✓ Jeżeli Twoje makro zapisuje dane w skoroszycie, wyłącz aktualizację ekranu za pomocą polecenia *Application.ScreenUpdating = False*.

Nie zapomnij przywrócić domyślnych ustawień arkusza dla dwóch ostatnich punktów, kiedy Twoje makro zakończy działanie.

Jak mogę wyświetlać w oknie komunikatów (MsgBox) kilka wierszy tekstu?

Najprostszym rozwiązaniem jest utworzenie komunikatu w zmiennej tekstowej przy użyciu stałej *vbNewLine* do wskazywania miejsca podziału wierszy. Poniżej przedstawiam przykład takiego rozwiązania.

```
Msg = "Zaznaczyłeś następujące elementy:" & vbNewLine
Msg = Msg & UserAns
MsgBox Msg
```

Utworzyłem makro, które usuwa arkusze ze skoroszytu. Jak mogę uniknąć wyświetlania ostrzeżeń przez Excel?

Wstaw polecenie przedstawione poniżej bezpośrednio przed fragmentem kodu, który usuwa arkusze ze skoroszytu.

```
Application.DisplayAlerts = False
```

Dlaczego operator kontynuacji wiersza VBA (znak podkreślenia) w moim kodzie nie działa?

Operator kontynuacji wiersza składa się z dwóch znaków — spacji i znaku podkreślenia. Upewnij się, że na końcu wiersza wpisałeś oba znaki i po znaku podkreślenia nacisnąłłeś klawisz *Enter*.

Rozdział 23

(Prawie) dziesięć źródeł informacji na temat Excela

W tym rozdziale:

- ▶ dowiesz się, dlaczego warto korzystać z systemu pomocy VBA,
- ▶ dowiesz się, jak otrzymać pomoc od firmy Microsoft,
- ▶ dowiesz się, jak znaleźć pomoc w internecie.

Książka, którą trzymasz w ręku, jest tylko wprowadzeniem do programowania Excela w języku VBA. Jeżeli nadal czujesz niepohamowany głód wiedzy, powinieneś zapoznać się z listą dodatkowych źródeł informacji na temat Excela i VBA, którą przygotowałem dla Ciebie w tym rozdziale. Dzięki tym źródłom odkryjesz nowe techniki programowania, nawiążesz kontakty z innymi użytkownikami Excela (o ile — oczywiście — chcesz...), będziesz mógł pobrać wiele plików z ciekawymi i użytecznymi przykładami zastosowań VBA, zadawać pytania na forach dyskusyjnych i wiele więcej.



Większość źródeł przedstawionych poniżej to różnego rodzaju portale, fora i inne zasoby internetowe, których adresy sieciowe mają naturalną tendencję do częstych zmian. Z tego względu adresy niektórych źródeł były prawidłowe w czasie, kiedy powstawała ta książka, ale nie mogę gwarantować, że tak jest w chwili, kiedy czytasz te słowa. Tak działa internet.

System pomocy języka VBA

Mam nadzieję, że do tej pory zdążyłeś się już zapoznać z systemem pomocy języka VBA. Moim skromnym zdaniem, jest on szczególnie przydatny do identyfikacji obiektów oraz ich właściwości i metod. Jego ogromną zaletą jest to, że jest zawsze pod ręką, jest całkowicie bezpłatny i bardzo (zazwyczaj...) dokładny. Korzystaj z niego!

Jeżeli używasz Excela 2013, pamiętaj, że w tej wersji Excela system pomocy VBA znajduje się w całości w sieci, na serwerach internetowych firmy Microsoft. Oznacza to — niestety — że aby z niego skorzystać, musisz dysponować połączeniem z internetem.

Wsparcie techniczne firmy Microsoft

Firma Microsoft oferuje szeroki zakres wariantów obsługi technicznej (niektóre z nich są bezpłatne, a za inne musisz dodatkowo zapłacić). Aby skorzystać ze wsparcia technicznego firmy Microsoft, powinieneś zajrzeć na stronę internetową:

<http://support.microsoft.com/>

Nie zapominaj również o stronie domowej pakietu Microsoft Office, gdzie znajdziesz całą masę materiałów dotyczących Excela.

<http://office.microsoft.com/>

Innym, znakomitym źródłem informacji jest strona domowa serwisu Microsoft Developer Network (MSDN). Znajdziesz tam mnóstwo informacji przeznaczonych dla programistów (tak, teraz właśnie nim jesteś!). Poniżej znajdziesz adres głównej strony internetowej serwisu, gdzie możesz rozpocząć wyszukiwanie informacji dotyczących Excela.

<http://msdn.microsoft.com/p1-p1/>

Inne strony internetowe

W internecie można znaleźć bardzo wiele stron internetowych poświęconych Excelowi i jego tajemnicom. Dobrym miejscem do rozpoczęcia poszukiwań jest strona internetowa autora tej książki, *The Spreadsheet Page*. Po wejściu na stronę możesz odwiedzić między innymi sekcję *Resources*, gdzie znajdziesz łącza prowadzące do wielu ciekawych stron dedykowanych Excelowi. Stronę internetową autora znajdziesz pod adresem:

<http://spreadsheetpage.com/>

Blogi poświęcone Excelowi

W internecie możesz znaleźć dosłownie miliony blogów poświęconych Excelowi. *Blog* to w zasadzie swego rodzaju internetowy dziennik poświęcony określonym tematowi. Bardzo wiele takich blogów jest dedykowanych Excelowi. Dosyć rozbudowaną listę takich blogów możesz znaleźć na stronie internetowej autora tej książki, pod adresem:

http://spreadsheetpage.com/index.php/excel_feeds

Google

Kiedy nasuwa mi się pytanie na dowolny temat (włącznie z pytaniami dotyczącymi programowania Excela), moją pierwszą reakcją jest uruchomienie wyszukiwarki Google — obecnie jest to chyba najpopularniejsza wyszukiwarka sieciowa na świecie:

<http://www.google.pl/>

Wpisz w polu wyszukiwania kilka fraz dotyczących Twojego pytania (na przykład zawierających słowa *Excel* i *VBA*) i zobacz, co Google znajdzie na ten temat. Zazwyczaj znajduję tam satysfakcyjne odpowiedzi przynajmniej w 90% przypadków.

Bing

Wyszukiwarka Bing to odpowiedź firmy Microsoft na ogromną popularność serwisu Google. Niektórzy użytkownicy wolą korzystać z wyszukiwarki Bing niż z Google, inni — wręcz przeciwnie. Jeżeli nigdy nie korzystałeś z wyszukiwarki Bing, zajrzyj na stronę:

<http://bing.pl/>

Lokalne grupy użytkowników

W wielu społecznościach lokalnych i środowiskach akademickich istnieją mniej lub bardziej formalne grupy użytkowników Excela, które od czasu do czasu organizują spotkania. Jeżeli w okolicy, w której mieszkasz, istnieje taka grupa, możesz spróbować się z nią skontaktować. Zazwyczaj takie grupy użytkowników są znakomitym źródłem kontaktów i pozwalają na wymianę pomysłów i koncepcji.

Moje inne książki

Jak widać, nie mogłem się powstrzymać, aby nie wykorzystać okazji do zareklamowania moich innych książek. Aby przenieść swoje umiejętności programowania w języku VBA na kolejny, bardziej zaawansowany poziom, możesz zatrudnić do moich książek *Excel 2010 PL. Programowanie w VBA. Vademeicum Walkenbacha* oraz *Excel 2013 PL. Programowanie w VBA. Vademeicum Walkenbacha* (obie zostały wydane przez Wydawnictwo Helion).

Rozdział 24

Dziesięć rzeczy, które powinieneś robić w języku VBA i których nie powinieneś robić

W tym rozdziale:

- poznasz pięć rzeczy, które zawsze powinieneś robić, tworząc aplikacje w języku VBA,
 - poznasz pięć rzeczy, których nigdy nie powinieneś robić, tworząc aplikacje w języku VBA.
-

Jeżeli czytasz ten końcowy rozdział książki, prawdopodobnie przeczytałeś wcześniej pozostałe i posiadasz już pewne doświadczenie w programowaniu Excela w języku VBA. A może po prostu pominąłeś wcześniejsze rozdziały i przeskoczyłeś od razu do ostatniego, aby dowiedzieć się, czy to rzeczywiście kamerdyner jest przestępca...

W tym rozdziale znajdziesz kilka porad, o których powinieneś pamiętać podczas tworzenia swoich własnych aplikacji w języku VBA. Postępowanie zgodnie z poniższymi sugestiami nie stanowi — oczywiście — uniwersalnego panaceum, które uchroni Cię przed popademieniem w programistyczne tarapaty, ale z pewnością pomoże Ci uniknąć wielu problemów, z którymi borykali się inni użytkownicy.

Zawsze deklaruj wszystkie zmienne

Jakie to jest wygodne... Po prostu tworzysz kod VBA bez konieczności przejmowania się tymi wszystkimi nużącyimi szczegółami związanymi z deklarowaniem wszystkich bez wyjątku zmiennych, których zamierzasz użyć w swoim programie... Niestety, mimo iż Excel pozwala na używanie zmiennych, które nie zostały zadeklarowane, takie postępowanie jest po prostu proszeniem się o kłopoty.

Pierwsze i najważniejsze przykazanie każdego programisty VBA powinno brzmieć:

Nie będziesz używał niezadeklarowanych zmiennych.

Jeżeli brakuje Ci wystarczającej samodyscypliny, powinieneś na początku każdego modułu kodu VBA umieścić dyrektywę Option Explicit. Dzięki temu nie będziesz mógł nawet uruchomić swojego programu, jeżeli znajdzie się w nim choć jedna niezadeklarowana zmienna. Tworzenie kodu bez deklaracji zmiennych ma tylko jedną „zaletę”: oszczędzasz kilka sekund czasu. W praktyce taka pozorna oszczędność może łatwo stać się źródłem koszmarów każdego programisty i gwarantuję Ci, że na wyszukiwanie i rozwiązywanie problemów spowodowanych brakiem odpowiednich deklaracji zmiennych stracisz dużo, dużo więcej czasu niż te kilka „zaoszczędzonych” sekund.

Nigdy nie powinieneś mylić hasła chroniącego kod VBA z bezpieczeństwem aplikacji

Wyobraź sobie, że właśnie spędzałeś dobrych kilka miesięcy na tworzeniu wspaniałej aplikacji dla Excela, zawierającej kilka naprawdę innowacyjnych makr. Jesteś gotowy na podzielenie się tą aplikacją ze światem, ale jednocześnie nie chciałbyś, aby inni użytkownicy mogli „podglądać” Twój kod VBA i wykorzystać Twoje „tajemnice kuchni” w swoich programach. Co robisz w takiej sytuacji? Zakładasz hasło chroniące projekt VBA Twojej aplikacji i masz problem z głowy, prawda? Niestety, nic bardziej mylnego.

Zastosowanie hasła chroniącego kod VBA Twojej aplikacji spowoduje, że większość normalnych użytkowników nie będzie mogła podglądać kodu. Niestety, możesz być pewny, że jeżeli ktoś naprawdę będzie chciał uzyskać do niego dostęp, bez większych trudności znajdzie sposób na złamanie hasła.

Wnioski? Jeżeli naprawdę musisz utrzymać kod źródłowy swojej aplikacji w tajemnicy przed innymi użytkownikami, użycie Excela jako platformy deweloperskiej z pewnością nie będzie najlepszym rozwiązaniem.

Zawsze staraj się wyczyszczyć i zoptymalizować kod aplikacji

Kiedy Twoja aplikacja zaczyna w końcu działać w zadowalający sposób, zawsze powinieneś poświęcić trochę czasu na wyczyszczenie kodu z niepotrzebnych „śmieci”. Taka operacja powinna obejmować między innymi następujące sprawy.

- ✓ Upewnij się, że każda zmienna została odpowiednio zadeklarowana.
- ✓ Upewnij się, że wcięcia poszczególnych wierszy kodu są prawidłowe i struktura kodu jest przejrzysta.
- ✓ Usuń wszelki kod pomocniczy, którego używałeś do testowania aplikacji i wyszukiwania błędów, taki jak na przykład dodatkowe polecenia MsgBox, wyświetlające wartości pośrednie, czy polecenia Debug.Print.

- ✓ Zmień niezbyt oczywiste nazwy zmiennych na coś bardziej opisowego. Jeśli na przykład w kodzie aplikacji używasz zmiennej o nazwie *MojaZmienna*, z pewnością będziesz w stanie nadać jej inną nazwę, która będzie bardziej nawiązywała do roli i przeznaczenia takiej zmiennej. Za jakiś czas serdecznie pogratulujesz sobie, że to zrobiłeś.
- ✓ Jeżeli postępujesz tak jak ja, to w modułach kodu VBA Twoich aplikacji na pewno można znaleźć co najmniej kilka „testowych” funkcji i procedur, których używałeś podczas tworzenia aplikacji do sprawdzania poprawności działania takiego czy innego mechanizmu. Procedury spełniły już swoje zadanie, więc powinieneś je usunąć.
- ✓ Podczas pisania aplikacji umieszczaj w kodzie komentarze opisujące poszczególne zmienne, procedury czy ciekawe miejsca kodu tak, abyś za pół roku był w stanie zrozumieć, jak działa Twоя aplikacja.
- ✓ Upewnij się, że teksty pojawiające się na ekranie nie zawierają błędów i literówek; dotyczy to zwłaszcza tekstów wyświetlanych na formularzach *UserForm* i w oknach komunikatów.
- ✓ Sprawdź, czy aplikacja nie zawiera zdublowanego kodu. Jeżeli znajdziesz dwie lub więcej procedur, które wykorzystują identyczny fragment kodu, pomyśl o przeniesieniu go do osobnej procedury i zastąpieniu powtarzających się fragmentów kodu wywołaniem takiej procedury.

Nigdy nie umieszczaj wszystkiego w jednej procedurze

Chcesz utworzyć zupełnie niezrozumiałą kod programu? Bardzo efektywnym sposobem na osiągnięcie takiego rezultatu jest umieszczenie całego kodu aplikacji w jednej, ogromnej procedurze. Jeżeli kiedykolwiek będziesz musiał powrócić do kodu tej aplikacji w celu dokonania zmian i modyfikacji, z pewnością w takiej sytuacji będziesz znacznie bardziej narażony na popełnienie co najmniej kilku mniej lub bardziej poważnych błędów.

Wiesz już, na czym polega problem? Dobrym wyjściem z takiej sytuacji jest — oczywiście — kod modularny. Podziel zadania realizowane przez Twoją aplikację na mniejsze części i postaraj się dla każdej z nich napisać osobną procedurę. Kiedy przyzwyczaisz się do takiego postępowania, przekonasz się, że pisanie programów, które są wolne od błędów, wcale nie jest takie trudne.

Zawsze powinieneś rozważyć zastosowanie innego oprogramowania

Excel jest niesamowicie uniwersalnym programem, ale nie można go używać do wszystkiego. Kiedy jesteś gotowy, aby rozpocząć pracę nad nowym projektem, powinieneś poświęcić nieco czasu na rozważenie wszystkich dostępnych rozwiązań. Nieco parafrując stare przysłowie, można powiedzieć, że jeżeli jedynym narzędziem, jakie posiadasz, jest Excel, to każdy problem wygląda jak makro VBA.

Nigdy nie zakładaj, że każdy użytkownik zezwala na uruchamianie makr

Jak już dobrze wiesz, Excel pozwala na załadowanie skoroszytu przy jednoczesnym zablokowaniu możliwości uruchamiania makr. W praktyce wygląda to tak, jakby projektanci najnowszych wersji Excela po prostu *chcieli*, aby użytkownicy blokowali możliwość uruchamiania makr.

Oczywiście, włączanie możliwości uruchamiania makr podczas ładowania skoroszytów pochodzących z nieznanego źródła nie jest zbyt rozsądnym rozwiązaniem. A zatem musisz znać swoich użytkowników. W niektórych środowiskach korporacyjnych możliwość uruchamiania makr VBA jest domyślnie zablokowana przez administratorów we wszystkich aplikacjach pakietu Microsoft Office i użytkownik nie ma żadnej możliwości zmiany tego ustawienia.

Jednym z rozwiązań, o których możesz pomyśleć, jest cyfrowe podpisywanie aplikacji, które udostępniasz innym użytkownikom. Dzięki temu użytkownicy mają pewność, że dana aplikacja naprawdę pochodzi od Ciebie i nie została gdzieś „po drodze” zmodyfikowana. Więcej szczegółowych informacji na temat cyfrowego podpisywania aplikacji znajdziesz w pomocy systemowej Excela.

Zawsze staraj się eksperymentować z nowymi rozwiązaniami

Kiedy pracuję nad dużym projektem w Excelu, zazwyczaj spędzam całkiem sporo czasu, pracując nad małymi i dużymi „eksperymentami” z VBA. Jeśli na przykład próbuję dowiedzieć się czegoś nowego o danym obiekcie, metodzie czy właściwości, często tworzę prostą procedurę testową i eksperymentuję z nią dopóty, dopóki nie uzyskam pewności, że wiem, jak to działa i jakie może powodować problemy.

Realizacja takich prostych eksperymentów jest zazwyczaj znacznie bardziej efektywnym rozwiązaniem niż umieszczanie w kodzie aplikacji funkcji i procedur, których działanie nie jest do końca zrozumiałe.

Nigdy z góry nie zakładaj, że Twój kod będzie poprawnie działał z innymi wersjami Excela

Obecnie w powszechnym użyciu jest co najmniej pięć różnych wersji Excela. Kiedy tworzysz nową aplikację przeznaczoną dla tego popularnego arkusza kalkulacyjnego, nie masz absolutnie żadnej gwarancji, że będzie ona bezproblemowo działała ze starszymi, czy też z nowszymi niż Twoja wersjami Excela. W niektórych przypadkach takie

niekompatybilności mogą być dosyć oczywiste (jeśli na przykład kod Twojej procedury odwołuje się do komórki o adresie XYD877322, możesz być pewny, że nie będzie działać z wersjami starszymi niż Excel 2007, ponieważ w starszych wersjach Excela maksymalne rozmiary arkusza były o wiele mniejsze). Z drugiej strony jednak, z pewnością szybko się przekonasz, że istnieją rzeczy, które w teorii powinny poprawnie działać ze starszymi wersjami Excela, ale w praktyce po prostu działać nie chcą...

Excel posiada, co prawda, bardzo wygodny mechanizm, pozwalający na sprawdzanie kompatybilności skoroszytu ze starszymi wersjami (aby go uruchomić, przejdź na kartę *PLIK*, kliknij kategorię *Informacje*, rozwiń listę *Wyszukaj problemy* i następnie z menu podręcznego wybierz polecenie *Sprawdź zgodność*), aczkolwiek sprawdza on jedynie kompatybilność samego skoroszytu i całkowicie ignoruje kod VBA. Z tego względu jedynym sposobem na upewnienie się, że Twój aplikacja działa poprawnie z innymi wersjami Excela, jest samodzielne przetestowanie jej działania na takich wersjach.

Zawsze pamiętaj o użytkownikach Twojej aplikacji

Aplikacje Excela można podzielić na dwie główne kategorie: takie, które zaprojektowałeś dla siebie, i takie, które zaprojektowałeś dla innych użytkowników. Jeżeli tworzysz aplikację, której używać będą inni użytkownicy, Twój zadanie staje się znacznie trudniejsze, ponieważ nie możesz poczynić takich samych założeń, jak w przypadku aplikacji, które tworzysz na swój własny użytk. Jeżeli na przykład jesteś jedynym użytkownikiem aplikacji, możesz sobie nieco bardziej „odpuścić” procedury obsługi błędów — po prostu jeżeli w trakcie pracy pojawi się jakiś błąd, zazwyczaj będziesz dobrze wiedział, co go spowodowało, gdzie go szukać i jak poprawić. Jeżeli jednak z aplikacją korzystają inni użytkownicy i na ekranie pojawi się taki sam błąd, oni nie będą wiedzieli, jak sobie z tym poradzić. Oprócz tego, kiedy pracujesz z własną aplikacją, zazwyczaj doskonale sobie poradzisz bez podręcznika użytkownika...

Zawsze musisz wiedzieć, jakie umiejętności posiadają przyszli użytkownicy Twojej aplikacji i jakie problemy mogą mieć z jej użytkowaniem. Spróbuj sobie wyobrazić samego siebie jako świeżo upieczonego użytkownika tej aplikacji i przewidzieć, jakie aspekty pracy z aplikacją mogą sprawiać problemy.

Nigdy nie zapominaj o tworzeniu kopii zapasowych

Nic nie jest tak irytujące, jak niespodziewana awaria dysku twardego, którego kopii zapasowej nie posiadasz. Jeżeli pracujesz nad ważnym projektem, powinieneś zadać proste pytanie, co się stanie, kiedy dzisiaj w nocy mój komputer ulegnie awarii. Jakie dane mogą zostać utracone? Jeżeli dojdzieš do wniosku, że na skutek awarii stracisz więcej niż kilka godzin pracy, powinieneś dokładniej przyjrzeć się aktualnej procedurze tworzenia kopii zapasowych... Bo przecież regularnie robisz kopie zapasowe, prawda?

Skorowidz

A

- Add-In, 34
- aplikacja
 - bezpieczeństwo, 384
 - jako niezależny program, 33
 - oparta na makrach, 32
- arkusz, 34
 - aktywacja, 180
 - aktywny, 169
 - dodawanie, 61
 - nazwa, 169
 - okno kodu, 180
 - wykres, 72
- Auto Data Tips, 65
- Auto Indent, 65
- Auto List Members, 64, 78, 80, 144, 150
- Auto Quick Info, 65
- Auto Syntax Check, 64
- automatyzacja, 30

B

- biblioteka obiektów, Patrz: obiekt biblioteka
- Bieżący obszar, 220
- blok
 - instrukcji jako komentarz, 108
 - tekstu, 31
- błąd, 76, 146
 - #WARTOŚĆ!, 352
 - czasu wykonania, 116
 - eliminowanie, 215
 - graniczny, 204
 - ignorowanie, 197, 199
 - kod, 144, 146
 - kompilacji, 60, 110
 - komunikat, 146, Patrz: komunikat o błędzie
 - numer, 200
 - obsługa, 192, 195, 196, 197, 199

- poprawianie, Patrz: odpluskwianie programowania, 191, 203, 204, 205
 - kontekstem operacji, 204
 - logiczny, 204
 - typ danych, 204
 - warunki graniczne, 204
 - wersja programu, 204
- przechwytywanie, 195
- rozpoznawanie, 200
- składni, 64, 136, 204
- Subscript out of range, 376
- wykonania, 191, 196
- zakresu, 127
- zamierzony, 201
- breakpoint, Patrz: punkt przerwania

C

- Chart, 34, 70, 71, 231, 232
- Charts, 72
- ciąg znaków, 142, 147, 249
- collection, Patrz: kolekcja
- Comment Block, 108
- Custom UI Editor for Microsoft Office, 325
- czas, 147, 186
 - systemowy, 156
 - zapis liczbowy, 187
 - zegarowy, 119
- czcionka, 134

D

- dane
 - kopiowanie, 185
 - poprawność, 185
 - tekstowe, 118
- typ, 82, 107, 110, 237
 - błędy, 204
 - Boolean, 111, 134

dane
typ
 Byte, 111
 Currency, 111
 Date, 111, 119
 definiowanie, 95
 domyślny, 111
 Double, 111
 Integer, 111
 Long, 111
 Object, 111
 predefiniowany, 111
 przydział dynamiczny, 110
 Single, 111
 String, 111
 Variant, 111, 132, 134
wklejanie, 185
data, 119, 146, 147
 część, 146
 format, 120
 krótka, 120
 obliczenia, 146
 systemowa, 146
 zamiana na liczbę seryjną, 146
debugger, 216
debugowanie, Patrz: odpluskwanie
Default to Full Module View, 66
deklaracja, 58
Deweloper, 39
dodatek, 34, 48, 54, 361, 362, 363
 Analysis ToolPak, 208, 361, 363
 bezpieczeństwo, 369
 modyfikowanie, 371
 obiekt UserForm, 362
 opis, 368
 otwieranie, 363, 369
 Power Utility Pak, 361
 Solver, 361
 tworzenie, 32, 364, 365, 369
 udostępnianie, 370
Dostosowywanie Wstążki, 256
Drag-and-Drop Text Editing, 66
drzewo, 53, 54

E

edytor VBE, Patrz: VBE
ekran
 aktualizacja, 235, 377
 wyłączenie aktualizacji, 234

element graficzny, 261
embedded chart, Patrz: wykres osadzony
etykieta, 125, 154
Excel
 ustawienia, 227, 228
 wersja, 35, 229, 387

F

folder
 nazwa, 146
 zaufany, 23

formant, 259, 260, 275
 ActiveX, 88
 CheckBox, 261, 279
 ComboBox, 261, 280, 303
 CommandButton, 261, 265, 281
 dodawanie, 276, 277
 etykieta, 278
 Frame, 261, 281, 292
 grafika, 278
 Image, 261, 282
 jakoo kontener, 292
 klawisz skrótu, 292
 kolejność tabulacji, 291
 Label, 261, 283, 287, 310
 ListBox, 261, 283, 284, 303
 MultiPage, 261, 284, 292, 294, 315, 316
 nawigacja za pomocą klawiatury, 291
 nazwa, 264, 278
 obiektu CommandBar, 331, 332
 OptionButton, 261, 267, 276, 280, 285, 309
 pozycja w oknie, 278, 289, 290
 RefEdit, 261, 286
 rozmiar, 278, 290
 ScrollBar, 261, 286
 SpinButton, 261, 287, 310
 TabStrip, 261, 288
 TextBox, 261, 288, 296, 310
 tło, 278
 ToggleButton, 261, 289
 wartość, 278
 widoczność, 278
 właściwość, 261, 262, 277, 278
 Accelerator, 278, 279, 285
 AutoSize, 278, 288
 BackColor, 278
 BackStyle, 278
 BeginGroup, 332

- BorderStyle, 282
BuiltIn, 332
Cancel, 281
Caption, 278, 282, 332
ControlSource, 280, 284, 285, 287, 288
Default, 281
Enabled, 332
FaceID, 332
GroupName, 285
Height, 278
IntegralHeight, 284, 288
LargeChange, 287
Left, 278
ListCount, 303
ListIndex, 303, 305
ListRows, 280
ListStyle, 280, 284
Max, 287
MaxLength, 288
metody, 303
Min, 287
MultiLine, 288
MultiSelect, 284, 303, 306
Name, 278
OnAction, 332
Picture, 278, 282
PictureSizeMode, 282
RowSource, 280, 284
ScrollBars, 289
Selected, 303
SmallChange, 287
Style, 280, 285
TextAlign, 278, 289
ToolTipText, 333
Top, 278
Value, 278, 280, 284, 285, 287, 303
Visible, 278, 332
Width, 278
WordWrap, 289
zmiana, 277
zaznaczanie, 290
- format
 XLA, 362
 XLAM, 362
 xlsm, 45, 376
 XLSM, 362
 xlsx, 45, 376
- formularz UserForm, Patrz: UserForm
- formula, 134
nazwa, 44
odpowiednik angielski, 44
tablicowa, 355
funkcja, 33, 58, 59, 81, 141, 344,
 Patrz też: metoda
Abs, 146
argument, 82
argumenty, 345, 349
 opcjonalne, 351
 opis, 360
arkuszowa, 343, 358, 375
 ograniczenia, 344
Array, 146
bezargumentowa, 346
Choose, 146
Chr, 146
CurDir, 146
Date, 142, 146
DateAdd, 146
DateDiff, 146
DatePart, 146
DateSerial, 146
DateValue, 146
Day, 146
Dir, 146
DŁ, 142
dwuargumentowa, 348
Err, 146
Error, 146
Exp, 146
FileLen, 143, 146
Fix, 146
Format, 146
GetOpenFilename, 250
GetSetting, 146
Hour, 146
InputBox, 144, 146, 149, 154, 195, 225, 242,
 247, 248, 249
 argumenty, 248
 pobranie liczby, 249
InStr, 146
InStrRev, 146
Int, 146
IsArray, 146
IsDate, 146
IsEmpty, 146
IsError, 146
IsMissing, 146

funkcja
IsNull, 147
IsNumeric, 147, 193
jednoargumentowa, 346
LARGE, 148
LBound, 147
LCase, 147, 270
Left, 147
Len, 142, 147
MAX, 148
Mid, 147
MIN, 148
Minute, 147
MOD, 122, 151
Month, 147
MonthName, 143
MsgBox, 74, 142, 144, 147, 149, 206, 242, 247
 argumenty, 242
 przyciski, 245
nazwa, 83, 112
Now, 142, 147
opakowująca, 353
opis, 358
pasywna, 344
PMT, 148
Proper, 270
Replace, 147
RGB, 135, 147
Right, 147
Rnd, 147
Second, 147
Shell, 144, 147
Space, 147
Split, 147
Sqr, 147
StrConv, 270
String, 147
Time, 142, 147
Timer, 147
TimeSerial, 147
TimeValue, 147, 187
Trim, 147
tworzenie, 31, 345
TypeName, 144, 147
UBound, 147
UCase, 147, 257, 270
użytkownika, 141, 151
Val, 147
VLOOKUP, 149

wbudowana
języka VBA, 141, 142, 144
podpowiedzi, 144
programu Excel, 141, 145, 150, 151
Weekday, 147
własna, Patrz: funkcja użytkownika
wykrywanie błędów, 352
WYSZUKAJ.PIONOWO, 149
wyświetlanie informacji, 65
wywołanie z procedury Sub, 352
wywoływanie, 82, 89, 90
Year, 147
Z.WIELKIEJ.LITERY, 270

G

generator liczb pseudolosowych, 353
godzina, 146
 długa, 120
 format, 119
Graphical User Interface, Patrz: GUI
GUI, 259

H

hasło, 54, 362, 369, 376, 384

I

identyfikator zadania, 144
instrukcja
 ElseIf, 157
 Exit For, 163
 Exit Sub, 154
 GoTo, 125, 153, 154, 155
 If-Then, Patrz: struktura If-Then
 On Error, 195, 196, 197
 On Error GoTo, 197
 On Error Resume, 197, 198
 On Error Resume Next, 189, 197, 199, 215
 Option Explicit, 60, 64, 112
 przypisania, 120
 ReDim, 124
 Resume, 197, 198
 Resume Next, 197
 Step, 163
 warunkowa, 95
IntelliSense, 64
interfejs użytkownika graficzny, Patrz: GUI

J

język
makr, 30
programowania, 30
XLM, 36
XML, 325

K

karta
Deweloper, 363, Patrz: Deweloper
DODATKI, 329
Plik, 363
zawierająca formanty, 261

klawisz Esc, 219

kod

ANSI, 146
spaghetti, 155

kolekcja, 34, 71

Addins, 361
ChartObject, 232
CommandBars, 330
element, 71
metoda, 77
przeglądanie, 168
Sheets, Patrz: Sheets
zakresów, 226

kolor, 135

motywu, 135
RGB, 147
standardowy, 135

TintAndShade, 135

vbBlack, 135

vbBlue, 135

vbCyan, 135

vbGreen, 135

vbMagenta, 135

vbRed, 135

vbWhite, 135

vbYellow, 135

wypełnienia, 135

kolumny ukrywanie, 61

komentarz, 44, 107, 108, 216, 385

komórka, 73

adresu wprowadzanie, 261

format, 185

niepusta, 223

pusta, 218

wartości wprowadzanie, 225

zaznaczanie, 219, 221

całego wiersza, 221

całej kolumny, 221

do końca kolumny, 220, 377

do końca wiersza, 220

komunikat

o błędzie, 146, 174

wymagający potwierdzenia, 236

komunikatem, 147

kontener, 34

kontrolka formularza, 87, 88

ksztalt, 34, 84, 87, 88

L

liczba

całkowita, 110

część całkowita, 146

e, 146

formatowanie, 132

pseudolosowa, 353

rzeczywista, 110

wartość bezwzględna, 146

lista, 261, 303

element, 304, 306

rozwijana, 261, 280

sortowanie, 356

logarytm naturalny, 146

lokalizacja zaufana, 23, 46, 47

Ł

łańcuch znaków, 110, 118, 119, 136, 142, 354

o stałej długości, 119

o zmiennej długości, 119

porównywanie, 355

M

makro, 30, 60, 82, Patrz też: procedura Sub,

program

bezpieczeństwo, 45, 47

instrukcje nadmiarowe, 44

klawisz skrótu, 100

kod, 42

lista, 63

lokalizacja, 101

modyfikacja, 44

makro
nazwa, 100
rejestrator, 58, 61, 82, 93, 95, 98, 375
ograniczenia, 95
opcje, 100
wydajność, 101, 218
rejestrowanie, 31, 41, 55, 88, 93, 95
w trybie odwołań bezwzględnych, 96
w trybie odwołań względnych, 97
testowanie, 272
ustawienia, 23, 46
menu, 331
podręczne, 329
Cell, 335
Excel 2003, 338
Excel 2013, 336
modyfikacja, 334
resetowanie, 334
wyłączanie, 337
wyświetlanie, 329
metoda, 35, 127, Patrz też: funkcja
Add, 77
AddChart, 230
AddChart2, 229, 230
Areas, 226
argument, 76
Cells, 129
Clear, 138
ClearContents, 76
Copy, 138, 219
Delete, 139
End, 221
ExecuteMso, 255
Export, 318
FileDialog, 242
GetOpenFilename, 242, 251
argumenty, 251
GetSaveAsFilename, 242, 253
InputBox, 242, 249
Intersect, 224
OnTime, 187, 188
Paste, 138
SaveCopyAs, 179
Select, 137
SpecialCells, 223, 224, 273
miesiąc, 143, 147, 355
model obiektowy, 34, 69, 110

moduł, 54, 154
Code, 263
dodawanie, 55, 94
limit znaków, 58
przewijanie w oknie, 66
sekcja Declarations, 115
tworzenie, 57, 58
usuwanie, 55

N

narzędzie Object Browser, Patrz: Object Browser

O

obiekt, 34, 69
ActiveChart, 231
Addin, 70
Add-In, Patrz: Add-In
Application, 34, 70, 72
biblioteka, 79
Chart, Patrz: Chart
ChartObject, 231
CommandBar, 329, 330, 338
formanty, 331, 332
CommandBars, 255
Comment, 70
eksportowanie, 56
Err, 200
FileDialog, 254
hierarchia, 34, 69
Hyperlink, 70
importowanie, 56
kontener, Patrz: kontener
metoda, 74, 76, 80, Patrz: metoda
Name, 70
numer indeksu, 72
odwołanie, Patrz: odwołanie
okno Code, 53
PageSetup, 70
PivotTable, 70, Patrz: PivotTable
Range, 70, Patrz: Range
Shape, 231
Ten_skoroszyt, 54
UserForm, 259
VBProject, 70
Window, 70
właściwość, Patrz: właściwość
Workbook, Patrz: Workbook

- Worksheet, Patrz: Worksheet
WorksheetFunction, 70, 145
wskaazywanie, 71
zakresu, Patrz: Range
zdarzenie, Patrz: zdarzenie
Object Browser, 78, 79
object-oriented programming, Patrz:
 programowanie zorientowane obiektowo
obsługa techniczna, 33
odpluskwanie, 33, 54, 204, 205, 208, 273
 metody, 205
 narzędzia, 209
odwołanie
 bezwzględne, 94, 96, 133
 do obiektu, 236
 do zakresu, 129, 130
 jednoznaczne, Patrz: odwołanie pełne
 pełne, 73, 74
 upraszczanie, 73
 w pełni kwalifikowane, Patrz: odwołanie pełne
 względne, 94, 96, 97, 130
Office Compatibility Pack, 37
okno
 dialogowe, 241, 253, 258
 dostosowywanie, 244
 pobieranie odpowiedzi, 243
 użytkownika, Patrz: UserForm
 wbudowane, 242, 254
Wstawianie funkcji, 358
wyświetlanie, 243
Immediate, 84
Properties, 261
Toolbox, 260
wprowadzania danych, 146
OOP, Patrz: programowanie zorientowane
 obiektowo
operator, 121
 dodawania, 121
 dzielenia, 121
 dzielenia całkowitego, 121
 konkatenacji ciągów znaków, 121, 122, 207
 kropki, 72, 73
 Like, 355
logiczny
 alternatywy, 122
 alternatywy wykluczającej, 122
 And, 122
 Eqv, 122
 Imp, 122
implikacji, 122
koniunkcji, 122
negacji, 122
Not, 122
Or, 122
równoważności, 122
XoR, 122
logiczny, 122
mnożenia, 121
Mod, 121, 122, 151
modulo, 121
odejmowania, 121
potęgowania, 121
priorytet, 122
znaku równości, 109
Option Explicit, 215, 384
- P**
- pasek
postępu zadania, Patrz: wskaźnik
 postępu zadania
przewijania, 261, 286, 289
szynkiego dostępu, 272
 umieszczań procedur, 299, 328
pętla, 95, 162, Patrz też: struktura
 Do-Until, 153, 154, 168
 Do-While, 153, 154, 167
 For Each-Next, 168, 222, 232
 For-Next, 153, 154, 162, 232
 czas wykonania, 165
 z instrukcją Exit For, 163
 z instrukcją Step, 163
 zagnieżdzona, 165
pierwiastek kwadratowy, 147
PivotTable, 34
plik
 liczba bajtów, 146
 nazwa, 146, 250
 PERSONAL.XLSB, 54, 101
 ścieżka, 146
 wielkość, 143
pluskwa, Patrz: błąd programowania
podprogram, 59
Pokaż podziały stron, 227
pokrętło, 261, 287, 310
pole
 etykiety, 261, 283
 grupy, 261, 281

- pole
karty, 261, 288
kombi, 261, 280
listy, 261, 280, 283, 303
obrazu, 261, 282
opcji, 261, 285, 296
strony, 261, 284
tekstowe, 261, 288, 289, 296, 310
wyboru, 261, 279
zakresu, 261, 286
- polecenie
Add Watch, 213
Debug.Print, 208, 384
DisplayAlerts, 236, 378
MsgBox, 384
On Error Resume Next, 223
Print, 212
Randomize, 353
Set, 237
- procedura
argumenty, 82
dysfunkcyjna, 34
Function, Patrz: funkcja
obsługi błędów
wbudowana, 196, 197
własna, 196
- obsługi zdarzenia, 173, 268
aktywacja arkusza, 180
aktywacja skoroszytu, 181
Open, 176
tworzenie, 173, 175
- obsługujące zdarzenie, 300
- separator, 66
- Sub, 33, 44, 58, 59, 60, 81, 82, 173,
Patrz też: makro
argumenty, 85, 87
nazwa, 83, 100
skrót klawiszowy, 41, 47, 86, 87, 271
tworzenie, 84
uruchamianie, 83
uruchamianie bezpośrednie, 85
uruchamianie w oknie dialogowym
Makro, 85
uruchamianie z poziomu innych
procedur, 89
uruchamianie za pomocą przycisków
i kształtów, 87, 88
uruchamianie za pomocą skrótów
klawiszowych, 86
- wywołanie, Patrz: procedura Sub
substandardowa, 33
udostępnienie użytkownikowi, 299
uruchamianie, 60
wyświetlającece okno dialogowe, 298
- Procedure Separator, 66
- program, Patrz: makro
wykonywalny, 147
wykonywanie krokowe, 211, 212
wymuszanie zatrzymania, 207
- programowanie
przykłady, 217
strukturalne, 154
zorientowane obiektywnie, 69
- projekt, 54
- przycisk, 84, 87, 331
na pasku narzędzi Szybki dostęp, 31, 84
na Wstążce, 31
opcji, 267
polecenia, 261, 281
poleceń, 265
przelącznika, 261, 289
tworzenie, 31
wstawianie, 87, 88
- pułapka, Patrz: punkt przerwania
- punkt przerwania, 210, 211, 352
usuwanie, 210
wstawianie, 209
- R**
- Range, 34, 73, 127, 129, 138, 217, 226
metoda, 137
- rata pożyczki, 148
- rejestr Windows, 146
- rejestrator makr, Patrz: makro rejestrator
- Require Variable Declaration, 64
- Require Variable Definition, 112
- RibbonX, 321
- runtime error, Patrz: błąd czasu wykonania
- S**
- Sheets, 72
- skoroszyt, 34, 54
dezaktywacja, 183
konwersja na plik dodatku, 48
kopia zapasowa, 179

- makr osobistych, 47, 54, 101
otwarty, 201
przekształcanie na dodatek, 364, 367
testowanie, 367
tryb obliczania
 automatyczny, 118
 przełączanie, 228
 ręczny, 118, 235, 377
XLSM, 361
zapisywanie, 45
zawierający makro, 45
skrót klawiszowy, 86
słowo kluczowe, 109, 112
 Call, 89
 Case, 159
 Const, 117
 Dim, 109, 113, 119, 123
 End, 109
 End Function, 82
 End Sub, 82
 End With, 103
 For, 109
 Function, 82
 Next, 109
 Preserve, 125
 Print, 212
 Private, 113
 Public, 113, 115, 123
 Static, 113
 Stop, 210
 Sub, 82, 109
 With, 103, 109
stała, 107, 117, 244
 predefiniowana, 118
 vbNewLine, 207, 377
 vbProperCase, 270
xlCalculationAutomatic, 235
xlCalculationManual, 118, 235
xlCalculationSemiautomatic, 118
xlDown, 377
xlToLeft, 377
xlToRight, 377
xlUp, 377
zasięg, 117
string, Patrz: łańcuch znaków
strona podgląd podziału, 227, 228
struktura, Patrz też: pętla
End If, 156
For Each-Next, 168, 222, 232
If-Then, 153, 155, 156, 157, 158, 199
If-Then-Else, 154, 155, 156, 157
Select Case, 153, 154, 159, 228
 zagnieżdżona, 160
With-End With, 233, 238, 377
siuwak, 261, 286
syntezator mowy, 354
system pomocy, 53, 78, 379
 formanty, 279
 funkcje wbudowane, 144
 zakres, 129
- T**
- tabela, 146, 147, 220
 kopiowanie, 220
 nazwa, 220
 przestawna, 34
 wiersz nagłówka, 220
tablica, 107, 123, 132
 deklarowanie, 123
 dynamiczna, 124
 liczba elementów, 124
 wielowymiarowa, 124
TintAndShade, 135
tryb Break, 211, 212, 214
- U**
- UserForm, 54, 241, 257, 295
 lista kontrolna, 318
 niemodalne, 315, 316
 poprawność danych, 302
 prowadnice, 276
 testowanie, 293, 299, 318
 tworzenie, 258, 259, 264, 265, 318
 właściwości, 261, 262
 wykres, 317
 wyświetlanie, 263
 wyświetlanie na ekranie, 270
 z wieloma kartami, 315
 zamienniki, 241
ustawienia regionalne, 132

V

VBA, 29
fundamenty, 33
kod, 53, 56, 57
 kopiowanie, 63
 lokalizacja, 173, 174
 optymalizacja, 234, 377, 384
 wcięcia, 58, 65, 161, 215, 384
moduł, Patrz: moduł
Project, 43
Project Explorer, Patrz: VBA Project
wady, 33
zalety, 32
VBE, 33, 42, 51, 79
 funkcje, 144
 menu podręczne, 52
okno, 52
 Code, 53, 56
 dokowanie, 68
 Immediate, 53, 54, 208, 211, 212
 Locals, 214
 Project, 53, 54
 Watch, 212, 213
pasek
 menu, 52
 narzędzi Edit, 66
 narzędzi Standard, 53
środowiska dostosowanie, 63
Tools Options, 63, 66, 67, 68
uruchamianie, 51
wygląd, 66
Visual Basic for Applications, Patrz: VBA

W

wartość
 False, 134
 Null, 134
 True, 134
watch expression, Patrz: wyrażenie monitorujące
węzeł
 Forms, 54
 Modules, 54
wiersza ukrywanie, 61
wirus, 22
właściwość, 74, 80, 127
 Accelerator, 268
 Address, 131, 133

Cells, 129
Color, 135
Column, 133
Columns, 133
Count, 133
CurrentRegion, 219, 221
DisplayAlerts, 169
EntireRow, 221
Font, 134
Formula, 136
FormulaLocal, 136
HasFormula, 134
Interior, 136
IsAddin, 361
NumberFormat, 137
Offset, 130
Path, 143
Row, 133
Rows, 133
Text, 132
ThemeColor, 135
UsedRange, 224
Value, 131
Visible, 169
Workbook, 54, 70, 71, 75
Worksheet, 34, 70, 71, 127, 138
wrapper function, Patrz: funkcja opakowująca
wskaźnik postępu zadania, 312
Wstążka, 84, 255, 321, 333
 dostosowywanie, 321, 324
 za pomocą kodu XML, 324, 329
wykres, 34, 72, 229, 230
 aktywowanie, 233
 formatowanie, 233
 na UserForm, 317
 osadzony, 231
 przetwarzanie, 231
 właściwości modyfikowanie, 232
wyrażenie, 120, 147
 monitorujące, 212, 213

Z

zabezpieczeń ustawienia, 23
zakres, 34
 cała kolumna, 128, 218
 cały wiersz, 128, 218
komórek, 34, 137, 217, 261, 307
 jako argument funkcji, 349

- kopiowanie, 218
nazwa, 218
nieciągły, 226
przenoszenie, 222
nazwa, 127
nieciągły, 128
o zmiennej wielkości kopiowanie, 219
określanie typu, 226
zaznaczenie wielokrotne, 226
zdarzenie, 77, 84, 171, 173
 Activate, 172
 aktywacyjne, 180
 BeforeClose, 172, 179
 BeforeDoubleClick, 172, 183
 BeforePrint, 172
 BeforeRightClick, 172, 184
 BeforeSave, 172, 179, 180
 Change, 172, 184
 Deactivate, 172
 dotyczące
 arkusza, 172, 180, 181, 183
 skoroszytu, 172, 176, 179, 182
 NewSheet, 172
 niezwiązane z obiekttami, 186, 188
 OnKey, 189
 OnTime, 186, 187, 188
 Open, 172, 176
 SelectionChange, 172
 SheetActivate, 172
 SheetBeforeDoubleClick, 172
 SheetBeforeRightClick, 172
 SheetChange, 172
 SheetDeactivate, 172
 SheetSelectionChange, 172
 WindowActivate, 172
 WindowDeactivate, 172
zegar analogowy, 188
- zmienna, 35, 107, 130, 157
czas życia, 116
deklarowanie, 111, 112, 215, 237, 383
globalna, 117
licznikowa, 162
lokalna, 114, 116, 214
łańcuchowa, 119
nazwa, 72, 109
niezainicjowana, 146
o zasięgu
 jednego modułu, 115, 117
 jednej procedury, Patrz: zmienna lokalna
o zasięgu globalnym, Patrz: zmienna globalna
obiektowa, 237, 377
przypisywanie wartości, 95
publiczna, Patrz: zmienna globalna
statyczna, 116
tekstowa, 377
typ, Patrz: dane typ
usuwanie z pamięci, 116
zasięg, 113, 114
znak
 $>=$, 156
 ", 121
 #, Patrz: znak krzyżyka
 &, 121, 331
 *, 121
 /, 121
 ^, 121
 +, 121
 apostrofu, 107
 cudzysłowu, 108, 127, 136
 cudzysłowu podwójnego, 136
 dolara, 133
 Esc, 219
 kontynuacji wiersza, 59, 121, 156, 378
 kropki, 72, 73, 119
 krzyżyka, 119
 łamania wiersza, 246
 nawias, 122, 142
 przecinka, 119
 równości, 77, 109, 121
 średnika, 77, 125
 zapytania, 212

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
<ftp://ftp.helion.pl/przykłady/e13pzb.zip>

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

GRUPA WYDAWNICZA

 Helion SA