Opanuj możliwości VBA z największym autorytetem w dziedzinie Excela!

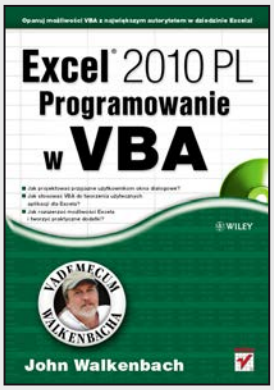
Excel® 2010 PL Programowanie w VBA

- Jak projektować przyjazne użytkownikom okna dialogowe?
- Jak stosować VBA do tworzenia użytecznych aplikacji dla Excela?
- Jak rozszerzać możliwości Excela i tworzyć praktyczne dodatki?

John Walkenbach

WILEY
Helion

Excel 2010 PL. Programowanie w VBA. Vademecum Walkenbacha



Autor: John Walkenbach
Tłumaczenie: Grzegorz Kowalczyk
ISBN: 978-83-246-2863-6
Tytuł oryginału: Excel 2010 Power Programming with VBA
Format: 172×245, stron: 1024

Opanuj możliwości VBA z największym autorytetem w dziedzinie Excela!

- Jak projektować przyjazne użytkownikom okna dialogowe?
- Jak stosować VBA do tworzenia użytecznych aplikacji dla Excela?
- Jak rozszerzać możliwości Excela i tworzyć praktyczne dodatki?

Nie należysz do osób, które onieśmiela potencjał Excela? Sprawnie tworzysz skoroszyty, wprowadzasz formuły, używasz funkcji arkuszowych i swobodnie posługujesz się Wstążką programu? Czujesz, że drzemie w nim jeszcze ogrom niezwykłych możliwości, ale nie wiesz, jak po nie sięgnąć? Najwyższa pora na naukę z Johnem Walkenbachem – najsłynniejszym ekspertem w dziedzinie Excela! Jeśli poznałeś już podstawowe funkcje tego programu, dzięki tej książce bez trudu opanujesz narzędzia zaawansowane, czyli takie, które naprawdę ułatwią i przyspieszą Twoją codzienną pracę!

Swoją naukę pod okiem mistrza zaczniesz od odświeżenia informacji na temat używania rozmaitych formuł oraz plików stosowanych i generowanych przez Excel. Zaraz potem przejdziesz do fascynującej części, poświęconej projektowaniu aplikacji w tym programie. Dowiesz się, czym taka aplikacja jest i jak szczegółowo wyglądają etapy jej tworzenia. Następnie opanujesz całą niezbędną wiedzę na temat języka VBA, aby sprawnie w nim programować oraz tworzyć funkcje i procedury. Nauczysz się również wykorzystywać jego możliwości podczas używania tabel przestawnych i wykresów. Ponadto wzbogacisz się o informacje na temat projektowania niestandardowych, przyjaznych okien dialogowych UserForm, automatycznej obsługi zdarzeń czy tworzenia praktycznych dodatków dla Excela.

- Przegląd możliwości Excela 2010
- Projektowanie aplikacji w programie Excel
- Język Visual Basic for Applications
- Zastosowanie formularzy UserForm
- Niestandardowe okna dialogowe
- Zaawansowane metody programowania
- Tabele przestawne, wykresy i obsługa zdarzeń
- Projektowanie dodatków do Excela
- Tworzenie systemów pomocy dla aplikacji
- Tworzenie aplikacji przyjaznych dla użytkownika
- Metody użycia VBA do pracy z plikami

Posiądź wiedzę profesjonalistów – wykorzystaj wszystkie możliwości Excela i poszerzaj je!

Helion SA
ul. Kościuszki 1c
44-100 Gliwice
tel. 32 230 98 63
e-mail: helion@helion.pl

Spis treści

Rozdział 11. Przykłady i techniki programowania w języku VBA

W tym rozdziale:

- Zastosowanie VBA do pracy z zakresami
- Zastosowanie VBA do pracy ze skoroszytami i arkuszami
- Tworzenie własnych funkcji i używanie ich w formułach arkusza i procedurach VBA
- Przykłady technik programowania w języku VBA
- Przykłady zastosowania funkcji interfejsu API

Nauka poprzez praktykę

Wierzę, że nauka programowania odbywa się znacznie szybciej, kiedy pracujemy na konkretnych przykładach omawianych zagadnień, a Czytelnicy poprzednich wydań książki zdecydowanie utwierdzają mnie w tym przekonaniu. Takie podejście sprawdza się zwłaszcza dla programistów pracujących z językiem VBA. Dobrze opracowany przykład o wiele lepiej objaśnia zagadnienie niż teoretyczny opis. W związku z tym zrezygnowałem z materiału referencyjnego, w którym dokładnie opisywano by wszystkie, nawet najdrobniejsze aspekty języka VBA, a zamiast tego przygotowałem przykłady demonstrujące użyteczne, praktyczne techniki programowania przy użyciu Excela.

Poprzednie rozdziały tej części książki odpowiednio przygotowały Czytelników do poznawania zagadnień omawianych w tym rozdziale, natomiast w systemie pomocy programu Excel znajdziesz wszystkie informacje, które tutaj zostały pominięte. W tym rozdziale zwiększy się nieco tempo i zaprezentowanych zostanie sporo przykładów rozwiązujących problemy spotykane w praktyce i pozwalających pogłębić wiedzę na temat języka VBA.

Przykłady omawiane w tym rozdziale zostały podzielone na sześć kategorii:

- Praca z zakresami
- Praca ze skoroszytami i arkuszami
- Techniki programowania w języku VBA
- Użyteczne funkcje, których warto używać w procedurach VBA
- Użyteczne funkcje, których możesz używać w formułach arkuszowych
- Wywołania funkcji i procedur Windows API

W kolejnych rozdziałach naszej książki znajdziesz szereg przykładów procedur dotyczących m.in. takich zagadnień, jak wykresy, tabele przestawne, zdarzenia, formularze *UserForm* i inne.

Przetwarzanie zakresów

Przykłady zamieszczone w tym podrozdziale demonstrują, w jaki sposób za pomocą języka VBA można manipulować zakresami arkusza.

W szczególności znajdziesz tutaj przykłady procedur, które pozwalają na kopiowanie i przenoszenie zakresów komórek, zaznaczanie zakresów komórek, identyfikację typów danych przechowywanych w danym zakresie komórek, wprowadzanie wartości do komórek przez użytkownika, wyszukiwanie pierwszej pustej komórki w kolumnie, zatrzymywanie makra w celu umożliwienia użytkownikowi zaznaczenia zakresu, zliczanie komórek w zakresie, przechodzenie w pętli i przetwarzanie kolejnych komórek zakresu oraz kilka innych operacji, często wykonywanych na zakresach komórek arkusza.

Kopiowanie zakresów

Rejestrator makr Excela jest bardzo przydatny nie tyle do generowania wydajnego, użytecznego kodu źródłowego, co do „odkrywania" nazw odpowiednich obiektów, metod i właściwości. Kod źródłowy generowany przez rejestrator makr nie zawsze jest optymalny i efektywny, ale zwykle pozwala uzyskać sporo przydatnych informacji.

Przykładowo po zarejestrowaniu prostej operacji kopiowania i wklejania, generowanych jest pięć wierszy kodu źródłowego języka VBA:

```
Sub Makro1()
    Range("A1").Select
    Selection.Copy
    Range("B1").Select
    ActiveSheet.Paste
    Application.CutCopyMode = False
End Sub
```

Wygenerowany kod najpierw powoduje zaznaczenie i skopiowanie komórki `A1`, a następnie, po zaznaczeniu komórki `B1`, procedura wykonuje operację wklejania. Jednak w języku VBA nie jest konieczne zaznaczanie obiektu, który będzie przetwarzany. O tej istotnej sprawie

Jak korzystać z przykładów zamieszczonych w tym rozdziale?

Nie wszystkie przykłady zamieszczone w tym rozdziale mogą spełniać rolę samodzielnych programów, ale zawsze mają postać wykonywalnych procedur, które możesz dostosować do własnych potrzeb i użyć w swoich aplikacjach.

W trakcie lektury powinieneś na bieżąco pracować z komputerem i samodzielnie testować opisywane w tym rozdziale przykłady (i nie tylko). Jeszcze lepiej będzie, jeżeli będziesz próbował samodzielnie modyfikować przykłady i sprawdzać, jaki będzie efekt tych modyfikacji. Mogę Ci zagwarantować, że takie praktyczne doświadczenia będą o wiele bardziej pomocne niż przeczytanie od deski do deski książki zawierającej tylko teoretyczną stronę programowania w języku VBA.

nie dowiedziałbyś się nigdy, gdybyś wzorował się tylko na kodzie źródłowym zarejestrowanego makra, w którym w dwóch instrukcjach została użyta metoda `Select`. Zamiast tego, możesz posłużyć się znacznie prostszą procedurą, która nie zaznacza żadnych komórek i korzysta z tego, że metoda `Copy` może użyć argumentu reprezentującego miejsce docelowe kopiowanego zakresu.

```
Sub CopyRange()
    Range("A1").Copy Range("B1")
End Sub
```

W obu powyższych makrach przyjęto założenie, że arkusz, w którym wykonywana jest operacja, jest aktywny. Aby skopiować zakres do innego arkusza lub skoroszytu, wystarczy odpowiednio zdefiniować odwołanie do zakresu docelowego. W poniższym przykładzie zakres jest kopiowany z arkusza `Arkusz1` skoroszytu *Plik1.xlsm* do arkusza `Arkusz2` skoroszytu *Plik2.xlsm*. Ponieważ odwołania są w pełni kwalifikowane, procedura zadziała niezależnie od tego, który skoroszyt będzie aktywny.

```
Sub CopyRange2()
    Workbooks("Plik1.xlsm").Sheets("Arkusz1").Range("A1").Copy _
        Workbooks("Plik2.xlsm").Sheets("Arkusz2").Range("A1")
End Sub
```

Kolejna metoda wykonania tej operacji polega na zastosowaniu zmiennych obiektowych reprezentujących zakresy, tak jak to zostało zilustrowane w kodzie poniższego przykładu:

```
Sub CopyRange3()
    Dim Rng1 As Range, Rng2 As Range
    Set Rng1 = Workbooks("Plik1.xlsm").Sheets("Arkusz1").Range("A1")
    Set Rng2 = Workbooks("Plik2.xlsm").Sheets("Arkusz2").Range("A1")
    Rng1.Copy Rng2
End Sub
```

Jak się zapewne domyślasz, kopiowanie nie jest ograniczone tylko do jednej komórki na raz. Przykładowo, procedura przedstawiona poniżej kopiuje duży zakres komórek. Zwróć uwagę na fakt, że miejsce docelowe jest tutaj identyfikowane tylko przez jedną komórkę — górną lewą komórkę wklejanego zakresu. Użycie jednej komórki działa dokładnie tak, jak podczas ręcznego kopiowania i wklejania komórek arkusza.

```
Sub CopyRange4()
    Range("A1:C800").Copy Range("D1")
End Sub
```

Przenoszenie zakresów

Instrukcje języka VBA służące do przenoszenia zakresu są bardzo podobne do instrukcji używanych podczas kopiowania zakresów, tak jak to zostało zaprezentowane na poniższym przykładzie. Różnica polega na tym, że zamiast metody `Copy` użyta została metoda `Cut`. Pamiętaj, że musisz podać tylko lokalizację górnej, lewej komórki zakresu docelowego.

W przykładzie przedstawionym poniżej 18 komórek (z zakresu `A1:C6`) przenosimy w nowy obszar, rozpoczynający się od adresu `H1`.

```
Sub MoveRange1()
    Range("A1:C6").Cut Range("H1")
End Sub
```

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 11.1 pokazano bardzo często spotykany typ arkusza. Zawarty w nim zakres składa się z kilku wierszy, których liczba zmienia się każdego tygodnia. Ponieważ nie wiesz, jaki jest adres zakresu w danej chwili, podczas pisania makra kopiującego zakres będziesz uwzględnić nieco dodatkowego kodu źródłowego.

Rysunek 11.1. *Liczba wierszy zakresu danych zmienia się każdego tygodnia*

	A	B	C	D
1	Tydzień	Sprzedaż	Liczba nowych klientów	
2	1	21093	45	
3	2	25375	49	
4	3	26180	38	
5	4	25664	32	
6	5	29325	22	
7	6	23069	23	
8	7	24281	53	

Poniższe makro 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 CopyCurrentRegion2()
    Range("A1").CurrentRegion.Copy Sheets("Arkusz2").Range("A1")
End Sub
```

Zastosowanie właściwości `CurrentRegion` jest równoważne przejściu na kartę *Narzędzia główne* i wybraniu polecenia *Znajdź i zaznacz/Przejdź do — specjalnie*, znajdującego się w grupie opcji *Edycja* i następnie zaznaczeniu opcji *Bieżący obszar* (zamiast tego możesz również nacisnąć kombinację klawiszy *Ctrl+Shift+**). Aby przekonać się, jak to działa, podczas wykonywania tych poleceń powinieneś zarejestrować makro. Zazwyczaj wartość właściwości `CurrentRegion` reprezentuje prostokątny blok komórek otoczony przez puste wiersze i kolumny.

Wskazówki dotyczące przetwarzania zakresów

W trakcie przetwarzania zakresów powinieneś pamiętać o kilku ważnych kwestiach.

- W języku VBA, do przetwarzania zakresu nie jest konieczne jego uprzednie zanaczenie.
- Nie możesz zaznaczyć zakresu, który znajduje się na nieaktywnym arkuszu, zatem, jeżeli Twoja procedura zaznacza zakres, powiązany z nim arkusz musi być aktywny. W celu uaktywnienia określonego arkusza można użyć metody `Activate` kolekcji `Worksheets`.
- Pamiętaj, że rejestrator makr nie generuje zbyt wydajnego kodu źródłowego. Najlepiej utworzyć makro przy użyciu rejestratora, a następnie jego kod źródłowy zmodyfikować w celu zwiększenia efektywności.
- W kodzie źródłowym języka VBA warto stosować nazwane zakresy. Przykładowo odwołanie `Range("Total")` jest znacznie bardziej czytelne niż odwołanie `Range("D45")`. W tym drugim przypadku dodanie wiersza powyżej wiersza 45. spowoduje zmianę adresu komórki i w konsekwencji konieczne będzie zmodyfikowanie makra tak, aby używało zakresu o poprawnym adresie (`D46`).
- Jeżeli w trakcie zaznaczania zakresów korzystasz z rejestratora makr, upewnij się, że makro rejestrowane jest przy użyciu odwołań względnych. Aby to zrobić, przejdź na kartę *Deweloper* i naciśnij przycisk *Użyj odwołań względnych*, znajdujący się w grupie opcji *Kod*.
- Po uruchomieniu makra przetwarzającego kolejne komórki aktualnie zaznaczonego zakresu, użytkownik może zaznaczać całe wiersze lub kolumny. W większości przypadków nie ma potrzeby przetwarzania wszystkich komórek zaznaczonego zakresu. Tworzone makro powinno definiować podzbiór zaznaczenia zawierający wyłącznie niepuste komórki. Więcej szczegółowych informacji na ten temat znajdziesz w podrozdziale „Wydajne przetwarzanie komórek zaznaczonego zakresu przy użyciu pętli" w dalszej części rozdziału.
- Excel pozwala zaznaczać wiele obszarów jednocześnie. Na przykład możesz zaznaczyć pierwszy zakres, wcisnąć klawisz *Ctrl* i zaznaczyć kolejny zakres. Makro powinno dokonać sprawdzenia zakresu i podjąć odpowiednią decyzję. Zapoznaj się z zawartością punktu „Określanie typu zaznaczonego zakresu" w dalszej części rozdziału.

Jeżeli zakres komórek, który chcesz skopiować jest tabelą (zdefiniowaną przy użycia polecenia *Tabela*, znajdującego się nakrcie *Narzędzia główne*, w grupie poleceń *Tabele*), możesz użyć kodu przedstawionego poniżej (który zakłada, że tabela ma nazwę `Table1`.

```
Sub CopyTable
   Range("Table1[#All]").Copy Sheets("Sheet 2").Range("A1")
End Sub
```

Zaznaczanie oraz identyfikacja różnego typu zakresów

Większość operacji wykonywanych przez instrukcje języka VBA opiera się na zakresach — poprzez definiowanie zakresów lub identyfikowanie zakresów w celu wykonywania operacji na komórkach do nich należących.

Oprócz właściwości `CurrentRegion` (o której mówiliśmy już wcześniej) powinieneś również poznać metodę `End` obiektu `Range`. Metoda ta pobiera jeden argument określający kierunek, w którym zostanie wykonane zaznaczenie. Poniższe polecenie zaznacza zakres rozpoczynający się od aktywnej komórki i kończący na ostatniej niepustej dolnej komórce:

```
Range(ActiveCell, ActiveCell.End(xlDown)).Select
```

Poniżej zamieszczamy kolejny przykład, w którym została zdefiniowana komórka będąca początkiem zakresu:

```
Range(Range("A2"), Range("A2").End(xlDown)).Select
```

Jak można się domyślić, trzy pozostałe stałe (`xlUp`, `xlToLeft`, `xlToRight`) symulują kombinacje klawiszy zaznaczające komórki w innych kierunkach.

Korzystając z właściwości `ActiveCell` w powiązaniu z metodą `End`, powinieneś zachować szczególną ostrożność. Jeżeli aktywna komórka znajduje się na końcu zakresu lub jeżeli w skład zakresu wchodzi jedna lub więcej pustych komórek, wyniki działania metody `End` mogą być zupełnie inne od oczekiwanych.

Na dołączonym dysku CD-ROM znajduje się skoroszyt (*Zaznaczanie zakresów.xlsm*) ilustrujący najczęściej spotykane rodzaje zaznaczeń zakresów. Po jego otwarciu, w menu podręcznym pojawi się nowe podmenu o nazwie *Przykłady zaznaczeń*. Poszczególne polecenia menu umożliwiają użytkownikowi zapoznanie się z przykładami różnych rodzajów zaznaczeń (patrz rysunek 11.2).

Poniższe makro, o nazwie `SelectCurrentRegion` znajduje się w przykładowym skoroszycie i symuluje naciśnięcie kombinacji klawiszy *Ctrl*+*Shift*+*:

```
Sub SelectCurrentRegion()
    ActiveCell.CurrentRegion.Select
End Sub
```

Bardzo często zaznaczanie komórek jest tylko wstępem do innych operacji, na przykład formatowania. Procedurę zaznaczającą komórki można łatwo przystosować do tego celu. Procedura przedstawiona poniżej jest prostą modyfikacją makra `SelectCurrentRegion`, która nie zaznacza komórek, a jedynie formatuje zakres zdefiniowany jako bieżący obszar otaczający aktywną komórkę. Inne procedury znajdujące się w przykładowym skoroszycie też mogą zostać przystosowane w ten sposób.

```
Sub FormatCurrentRegion()
    ActiveCell.CurrentRegion.Font.Bold = True
End Sub
```

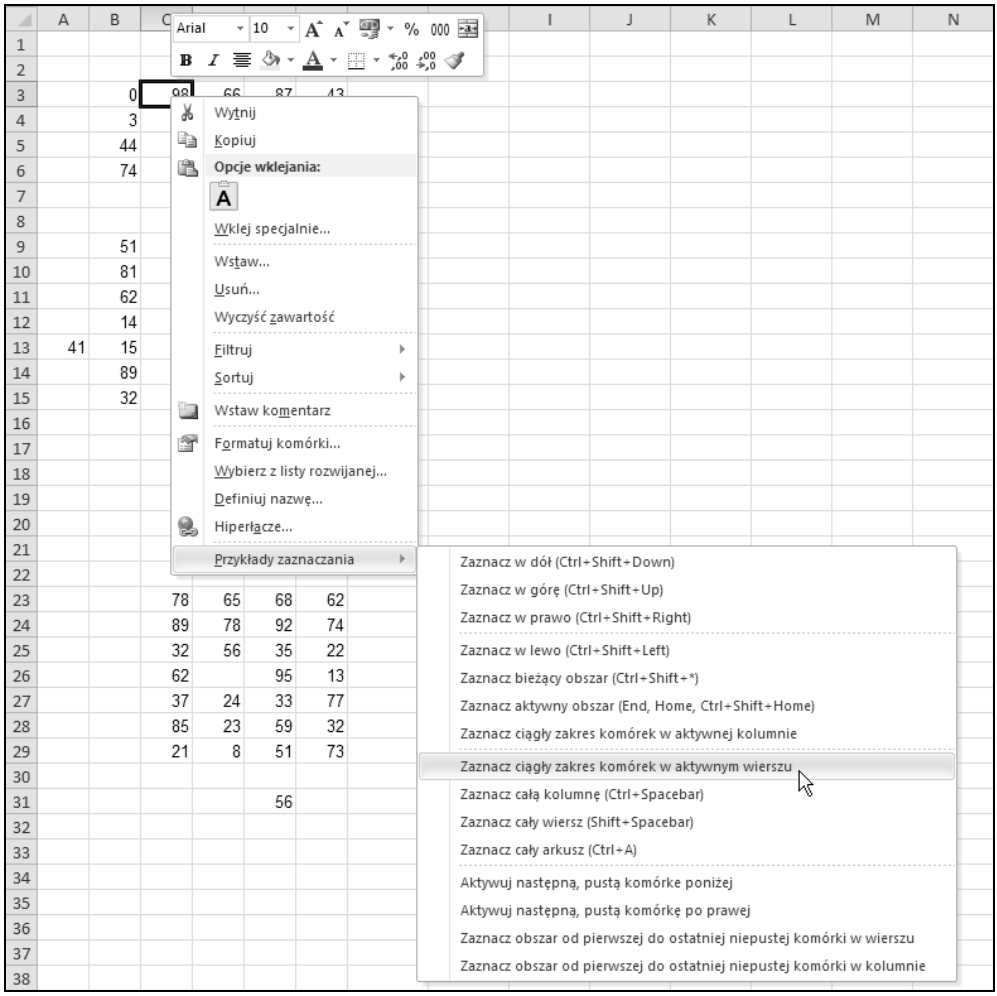
Wprowadzanie wartości do komórki

Kolejna procedura przedstawiona poniżej demonstruje, jak poprosić użytkownika o podanie wartości i wstawić ją do komórki `A1` aktywnego arkusza:

```
Sub GetValue1()
    Range("A1").Value = InputBox("Wprowadź wartość:")
End Sub
```

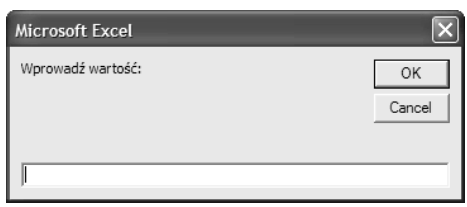
Na rysunku 11.3 pokazano wygląd okna umożliwiającego wprowadzenie wartości.

Przedstawiona procedura może sprawiać jednak pewien problem. Jeżeli użytkownik naciśnie w oknie dialogowym przycisk *Cancel*, procedura usunie wszelkie dane już znajdujące się w komórce. Poniższa zmodyfikowana wersja procedury sprawdza, czy został naciśnięty przycisk *Cancel* i jeżeli tak, nie dokonuje zmiany zawartości komórki:



Rysunek 11.2. *Niestandardowe menu podręczne w tym skoroszycie ilustruje zaznaczanie zakresów o różnej wielkości*

Rysunek 11.3. *Funkcja InputBox pobiera wartość, która zostanie umieszczona w komórce*

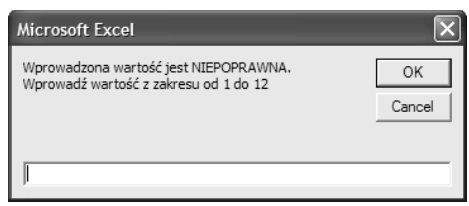



```
Sub GetValue2()
    Dim UserEntry As String
    UserEntry = InputBox("Wprowadź wartość:")
    If UserEntry <> "" Then Range("A1").Value = UserEntry
End Sub
```

W wielu przypadkach konieczne będzie sprawdzenie poprawności danych wprowadzonych przez użytkownika. Na przykład chcesz, aby użytkownik wprowadził liczbę z zakresu od 1 do 12. W poniższym przykładzie zademonstrowano jedną z metod sprawdzenia poprawności danych. Niepoprawna wartość jest ignorowana, a okno wyświetlane ponownie. Operacja jest powtarzana do momentu wprowadzenia prawidłowej wartości lub naciśnięcia przycisku *Cancel*.

```
Sub GetValue3()
    Dim UserEntry As Variant
    Dim Msg As String
    Const MinVal As Integer = 1
    Const MaxVal As Integer = 12
    Msg = "Wprowadź wartość z zakresu od " & MinVal & " do " & MaxVal
    Do
        UserEntry = InputBox(Msg)
        If UserEntry = "" Then Exit Sub
        If IsNumeric(UserEntry) Then
            If UserEntry >= MinVal And UserEntry <= MaxVal Then Exit Do
        End If
        Msg = "Wprowadzona wartość jest NIEPOPRAWNA."
        Msg = Msg & vbNewLine
        Msg = Msg & "Wprowadź wartość z zakresu od " & MinVal & " do " & MaxVal
    Loop
    ActiveSheet.Range("A1").Value = UserEntry
End Sub
```

Jeżeli użytkownik wprowadzi niepoprawną wartość, program odpowiednio zmieni treść wyświetlanego komunikatu (patrz rysunek 11.4).



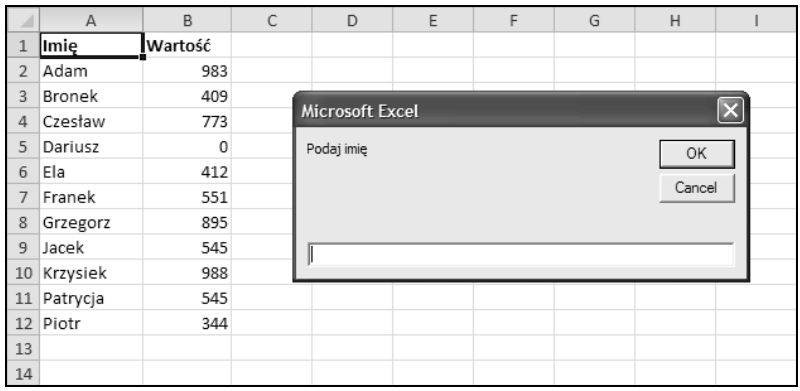
Rysunek 11.4. *Sprawdzenie poprawności danych wprowadzonych przez użytkownika przy użyciu funkcji InputBox języka VBA*

Skoroszyt zawierający wszystkie trzy przykłady (*Funkcja InputBox.xlsm*) znajdziesz na płycie CD-ROM dołączonej do książki.

Wprowadzanie wartości do następnej pustej komórki

Często wymaganą operacją jest wprowadzenie wartości do następnej pustej komórki kolumny lub wiersza. Poniższa procedura prosi użytkownika o podanie imienia i wartości, a następnie wprowadza dane do następnego pustego wiersza (patrz rysunek 11.5).

```
Sub GetData()
    Dim NextRow As Long
    Dim Entry1 As String, Entry2 As String
Do
    'Odszukaj następny pusty wiersz
    NextRow = Cells(Rows.Count, 1).End(xlUp).Row + 1
```



	A	B
1	Imię	Wartość
2	Adam	983
3	Bronek	409
4	Czesław	773
5	Dariusz	0
6	Ela	412
7	Franek	551
8	Grzegorz	895
9	Jacek	545
10	Krzysiek	988
11	Patrycja	545
12	Piotr	344
13		
14		

Rysunek 11.5. *Makro wstawiające dane do następnego pustego wiersza arkusza*

```
    ' Poproś o wprowadzenie danych
      Entry1 = InputBox("Podaj imię:")
      If Entry1 = "" Then Exit Sub
      Entry2 = InputBox("Podaj wartość:")
      If Entry2 = "" Then Exit Sub

    ' Zapisz dane w arkuszu
      Cells(NextRow, 1) = Entry1
      Cells(NextRow, 2) = Entry2
  Loop
End Sub
```

Dla uproszczenia nasza procedura w żaden sposób nie sprawdza poprawności wprowadzanych danych. Zwróć uwagę, że w procedurze nie został określony warunek zakończenia pętli. W celu jej opuszczenia użyto instrukcji `Exit Sub`, która jest wykonywana po naciśnięciu przycisku *Cancel* w oknie wprowadzania danych.

Skoroszyt zawierający oba przykłady (*NastępnaPustaKomórka.xlsm*) znajdziesz na płycie CD-ROM dołączonej do książki.

Zwróć uwagę na instrukcję określającą wartość zmiennej `NextRow`. Jeżeli nie rozumiesz, w jaki sposób procedura działa, spróbuj ręcznie wykonać realizowaną przez nią operację: uaktywnij ostatnią komórkę w kolumnie `A` (co w przypadku Excela 2010 oznacza komórkę o adresie `A1048576`), naciśnij klawisz *End* i następnie naciśnij klawisz ↑ (strzałka w górę). W efekcie zostanie zaznaczona ostatnia niepusta komórka kolumny `A`. Właściwość `Row` zwraca numer wiersza tej komórki. W celu uzyskania numeru kolejnego pustego wiersza (poniżej) wartość ta jest zwiększana o jeden. Zamiast umieszczać na „sztywno" adres ostatniej komórki wiersza `A`, użyta została metoda `Rows.Count`, dzięki czemu nasza procedura będzie poprawnie działała również z poprzednimi wersjami programu Excel (w których maksymalna liczba wierszy w arkuszu jest dużo mniejsza).

Z taką metodą zaznaczania następnej pustej komórki związany jest drobny problem. Jeżeli kolumna jest zupełnie pusta, jako następny pusty wiersz metoda wyznaczy wiersz `2`. Na szczęście dodanie odpowiedniego kodu, który będzie zapobiegał takiemu zachowaniu, nie jest trudnym zadaniem.

Wstrzymywanie działania makra w celu umożliwienia pobrania zakresu wyznaczonego przez użytkownika

Zdarzają się sytuacje, w których makro musi być w pewien sposób interaktywne. Na przykład możesz utworzyć makro, które wstrzymuje działanie, pozwalając użytkownikowi na zaznaczenie wybranego zakresu komórek. Procedura opisana w tym punkcie demonstruje sposób wykonania zadania przy użyciu metody `InputBox` Excela.

Nie należy mylić metody `InputBox` Excela z funkcją języka VBA o takiej samej nazwie. Co prawda obie funkcje mają taką samą nazwę, ale nie są takie same.

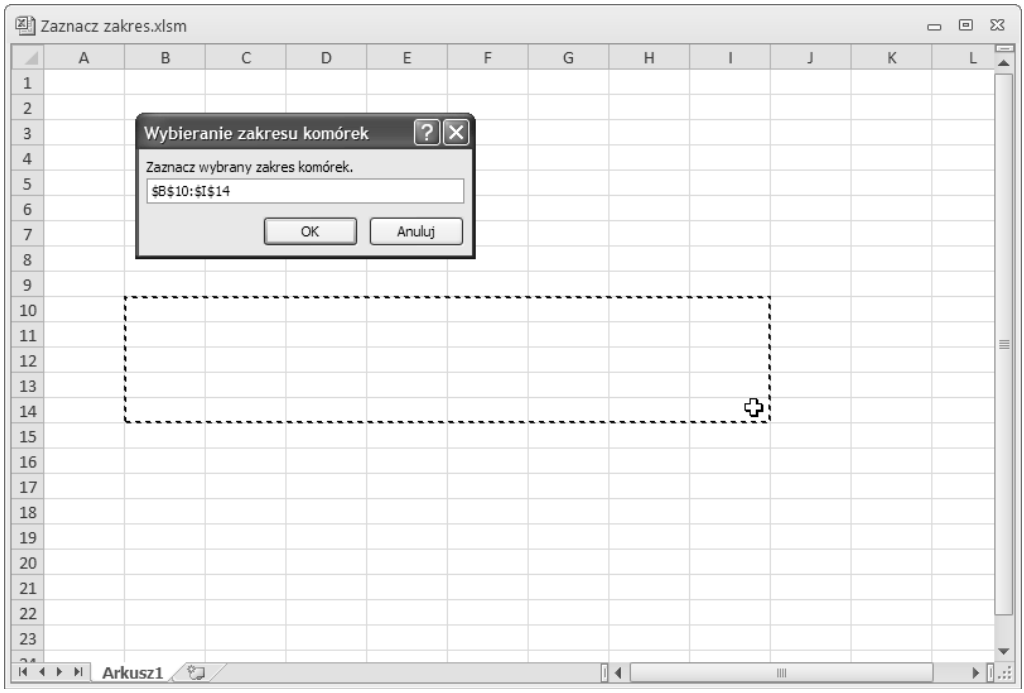
Poniższa procedura `Sub` demonstruje sposób zatrzymania pracy makra i umożliwienia użytkownikowi zaznaczenia zakresu komórek. Po wznowieniu działania procedura wstawia odpowiednią formułę do wszystkich komórek zaznaczonego zakresu.

```
Sub GetUserRange()
    Dim UserRange As Range

    Prompt = "Zaznacz wybrany zakres komórek."
    Title = "Wybieranie zakresu komórek"

'   Wyświetlanie okna dialogowego
    On Error Resume Next
    Set UserRange = Application.InputBox( _
        Prompt:=Prompt, _
        Title:=Title, _
        Default:=ActiveCell.Address, _
        Type:=8) 'Zaznaczanie zakresu komórek
    On Error GoTo 0

'   Czy okno dialogowe zostało anulowane?
    If UserRange Is Nothing Then
        MsgBox "Operacja anulowana."
    Else
        UserRange.Formula = "=RAND()"
    End If
End Sub
```

Okno dialogowe zostało przedstawione na rysunku 11.6.

Skoroszyt z tym przykładem (*Zaznacz zakres.xlsm*) znajdziesz na płycie CD-ROM dołączonej do książki.

Podanie argumentu `Type` o wartości `8` ma kluczowe znaczenie dla powyższej procedury. Oprócz tego powinieneś również zwrócić uwagę na zastosowanie instrukcji `On Error Resume Next`. To polecenie powoduje, że błąd, który wystąpi gdy użytkownik naciśnie przycisk *Anuluj*, będzie ignorowany. Jeżeli tak się stanie, nie zostanie zdefiniowana zmienna obiektowa `UserRange`. W powyższym przykładzie jest wyświetlane okno zawierające komunikat o treści *Operacja anulowana*. Gdy użytkownik naciśnie przycisk *OK*, wykonywanie makra będzie kontynuowane. Instrukcja `On Error GoTo 0` przywraca standardową obsługę błędów.



Rysunek 11.6. *Okno dialogowe użyte do wstrzymania działania makra*

Nawiasem mówiąc, sprawdzanie poprawności zaznaczonego zakresu nie jest konieczne, ponieważ zajmie się tym Excel.

Pamiętaj, podczas używania metody `InputBox` do zaznaczania zakresu komórek odświeżanie ekranu powinno być zawsze włączone. W przeciwnym wypadku nie będziesz w stanie zaznaczyć zakresu komórek. Do sterowania odświeżaniem ekranu w trakcie wykonywania makra powinieneś użyć właściwości `ScreenUpdating` obiektu `Application`.

Zliczanie zaznaczonych komórek

Można stworzyć makro przetwarzające zaznaczone komórki zakresu. Aby określić liczbę komórek w zaznaczonym zakresie (lub dowolnym innym), należy użyć właściwości `Count` obiektu `Range`. Przykładowo poniższa instrukcja wyświetla w oknie komunikatu liczbę aktualnie zaznaczonych komórek:

```
MsgBox Selection.Count
```

Jeżeli aktywny arkusz zawiera zakres o nazwie `dane`, polecenie przedstawione poniżej przypisze liczbę jego komórek zmiennej `CellCount`:

```
CellCount = Range("dane").Count
```

Możesz również określić liczbę wierszy lub kolumn w zakresie. Poniższe wyrażenie wyznacza liczbę kolumn znajdujących się w aktualnie zaznaczonym zakresie:

```
Selection.Columns.Count
```

Ze względu na fakt, że w najnowszych wersjach Excela (2007 i 2010) maksymalne rozmiary arkusza zostały znacząco powiększone, właściwość `Count` może w pewnych sytuacjach generować błąd. Właściwość `Count` wykorzystuje dane typu `Long`, zatem największa wartość, jaką ta właściwość może przechowywać, to 2 147 483 647, stąd jeżeli użytkownik zaznaczy na przykład 2048 pełnych kolumn (czyli 2 147 483 648 komórek), właściwość `Count` wygeneruje błąd. Na szczęście firma Microsoft dodała w Excelu, począwszy od wersji 2007, nową właściwość: `CountLarge`. Właściwość ta używa danych typu `Double`, które pozwalają na przechowywanie liczb z zakresu do 1,79+E308.

Wnioski? W przytłaczającej większości przypadków właściwość `Count` będzie działała najzupełniej poprawnie. Jeżeli jednak zakładasz, że będziesz zliczał arkusze zawierające naprawdę duże ilości komórek (na przykład wszystkie komórki arkusza), to zamiast właściwości `Count` powinieneś użyć właściwości `CountLarge`.

Oczywiście liczbę wierszy zakresu można również określić przy użyciu właściwości `Rows`. Poniższa instrukcja określa liczbę wierszy zakresu o nazwie `dane` i przypisuje wartość zmiennej `RowCount`:

```
RowCount = Range("dane").Rows.Count
```

Określanie typu zaznaczonego zakresu

Excel obsługuje kilka typów zaznaczeń zakresów. Oto one:

- pojedyncza komórka,
- ciągły zakres komórek,
- jedna lub więcej kolumn,
- jeden lub więcej wierszy,
- cały arkusz,
- dowolna kombinacja wyżej wymienionych typów, czyli zaznaczenie wielokrotne.

Ponieważ istnieje kilka typów zaznaczeń, w trakcie przetwarzania zakresu procedura języka VBA nie może przewidzieć, jaki jest typ zaznaczenia. Na przykład zaznaczony obszar może składać się z dwóch zakresów komórek, `A1:A10` i `C1:C10` (aby utworzyć zaznaczenie wielokrotne, podczas zaznaczania kolejnych zakresów trzymaj wciśnięty klawisz *Ctrl*).

Jeżeli zakres został zdefiniowany przez wiele zaznaczeń, obiekt `Range` będzie się składać z oddzielnych obszarów. Aby stwierdzić, czy zaznaczenie jest zaznaczeniem wielokrotnym, należy użyć metody `Areas` zwracającej kolekcję `Areas`. Kolekcja reprezentuje wszystkie obszary wchodzące w skład zakresu stworzonego poprzez zaznaczenie wielokrotne.

W celu stwierdzenia, czy wybrany zakres posiada wiele obszarów, należy zastosować wyrażenie podobne do poniższego:

```
NumAreas = Selection.Areas.Count
```

Jeżeli zmienna `NumAreas` zawiera wartość większą od `1`, zaznaczenie jest zaznaczeniem wielokrotnym.

Poniżej zamieszczono kod funkcji o nazwie AreaType, która zwraca łańcuch tekstu opisujący rodzaj zaznaczenia.

```
Function AreaType(RangeArea As Range) As String
'    Funkcja określa rodzaj zaznaczonego obszaru
    Select Case True
        Case RangeArea.Cells.CountLarge = 1
            AreaType = "Komórka"
        Case RangeArea.CountLarge = Cells.CountLarge
            AreaType = "Arkusz"
        Case RangeArea.Rows.Count = Cells.Rows.Count
            AreaType = "Kolumna"
        Case RangeArea.Columns.Count = Cells.Columns.Count
            AreaType = "Wiersz"
        Case Else
            AreaType = "Blok"
    End Select
End Function
```

Funkcja jako argument pobiera obiekt Range i zwraca jeden z pięciu łańcuchów opisujących obszar — Komórka, Arkusz, Kolumna, Wiersz lub Blok. W celu określenia, które z pięciu wyrażeń porównujących ma wartość True, funkcja korzysta z konstrukcji Select Case. Na przykład: jeżeli zakres składa się z jednej komórki, funkcja zwróci łańcuch Komórka. Jeżeli liczba komórek zakresu jest równa liczbie komórek arkusza, funkcja zwróci łańcuch Arkusz. Jeżeli liczba wierszy zakresu jest równa liczbie wierszy arkusza, funkcja zwróci łańcuch Kolumna. Jeżeli liczba kolumn zakresu jest równa liczbie kolumn arkusza, funkcja zwróci łańcuch Wiersz. Jeżeli żadne wyrażenie instrukcji Case nie będzie miało wartości True, funkcja zwróci łańcuch Blok.

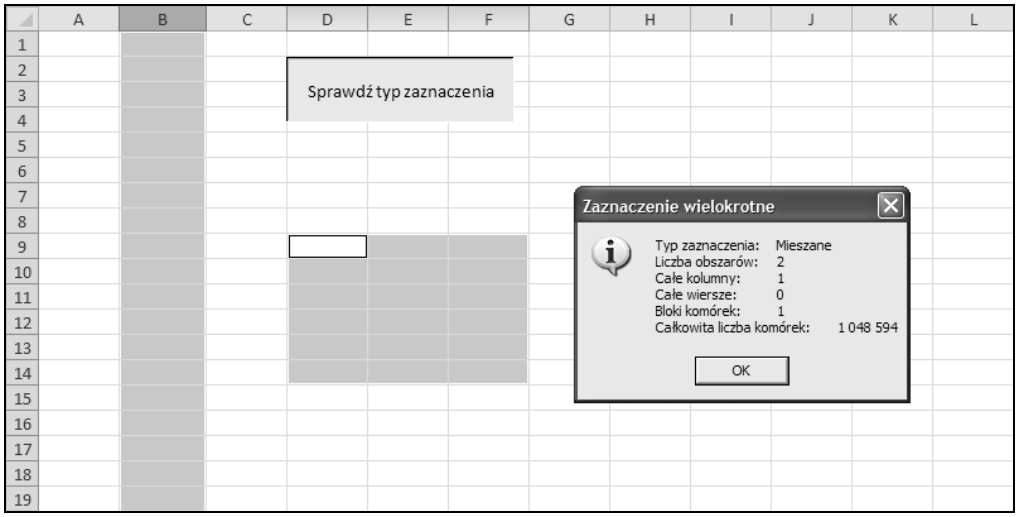
Zauważ, że do liczenia komórek użyta została właściwość CountLarge, ponieważ — jak już wspominaliśmy wcześniej — całkowita liczba zaznaczonych komórek w arkuszu może teoretycznie przekroczyć limit typu danych właściwości Count.

Skoroszyt *Zaznaczanie.xlsm* znajdujący się na dołączonym dysku CD-ROM zawiera procedurę RangeDescription, która posługuje się funkcją AreaType w celu wyświetlenia okna komunikatu opisującego typ zaznaczenia aktualnego zakresu. Na rysunku 11.7 pokazano przykład jej działania. Zrozumienie zasad działania funkcji będzie stanowiło dobre przygotowanie do wykonywania kolejnych operacji na obiektach klasy Range.

Excel pozwala na wykonanie wielu identycznych zaznaczeń. Na przykład, jeżeli trzymając wciśnięty klawisz *Ctrl*, pięciokrotnie klikniesz komórkę A1, zaznaczenie będzie złożone z pięciu identycznych obszarów. Procedura RangeDescription uwzględnia taka sytuację i nie zlicza wielokrotnie tych samych komórek.

Wydajne przetwarzanie komórek zaznaczonego zakresu przy użyciu pętli

Jednym z częściej wykonywanych przez makra zadań jest sprawdzanie poszczególnych komórek zakresu i wykonywanie określonych operacji, jeżeli komórka spełnia zadane kryterium. Kolejna procedura, której kod przedstawiamy poniżej, jest właśnie przykładem takiego makra. Procedura ColorNegaitve ustawia czerwony kolor tła dla wszystkich komórek zaznaczenia, które przechowują wartość ujemną. Kolor tła pozostałych komórek jest zerowany.



Rysunek 11.7. *Procedura AboutRangeSelection analizuje aktualnie zaznaczony zakres*

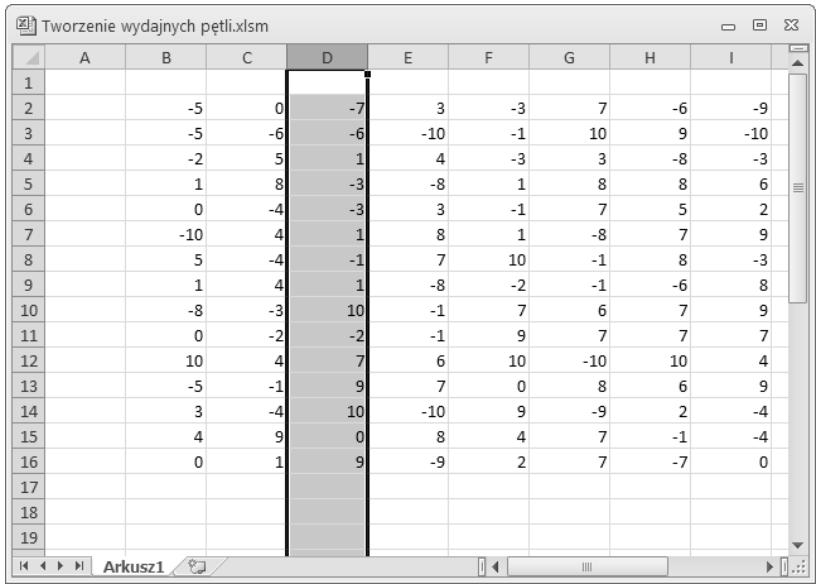
Poniższy przykład został opracowany tylko i wyłącznie w celach edukacyjnych. W zastosowaniach praktycznych o wiele lepszym rozwiązaniem będzie po prostu użycie mechanizmu formatowania warunkowego.

```
Sub ColorNegative()
'   Jeżeli wartość jest ujemna, zmienia kolor tła komórki na czerwony
    Dim cell As Range
    If TypeName(Selection) <> "Range" Then Exit Sub
    Application.ScreenUpdating = False
    For Each cell In Selection
        If cell.Value < 0 Then
            cell.Interior.ColorIndex = RGB(255, 0, 0)
        Else
            cell.Interior.ColorIndex = xlNone
        End If
    Next cell
End Sub
```

Z pewnością procedura `ColorNegative` zadziała, ale zawiera poważny błąd. Dla przykładu: co się stanie, jeżeli obszar danych na arkuszu jest bardzo mały, a użytkownik zaznaczy na przykład całą kolumnę? Albo 10 kolumn? Albo może nawet cały arkusz? Jak się zapewne sam domyślasz, nie ma żadnej potrzeby sprawdzania wszystkich pustych, nieużywanych komórek arkusza, nie mówiąc już o tym, że przy dużych zaznaczonych obszarach użytkownik z pewnością poddałby się, zanim cała procedura dobiegłaby do końca.

Lepszym rozwiązaniem jest procedura `ColorNegative2`, której kod przedstawiamy poniżej. W tej poprawionej wersji utworzyliśmy zmienną obiektową `WorkRange`, której zawartość odpowiada części wspólnej zaznaczonego obszaru i użytego obszaru arkusza. Przykładową sytuację ilustruje rysunek 11.8. Jak widać, zaznaczona jest cała kolumna `D` (czyli 1 048 576 komórek), ale użyty zakres komórek sprowadza się już tylko do obszaru `B2:I18`. Zatem przecięciem tych dwóch obszarów jest zakres `D2:D18`, co jest zdecydowanie mniejszym zakresem niż początkowe zaznaczenie. Różnica pomiędzy czasem przetwarzania 15 komórek a 1 048 576 komórek będzie naprawdę znacząca.

Rysunek 11.8. *Zastosowanie przecięcia zaznaczonego zakresu i użytego obszaru arkusza skutkuje znaczącym zmniejszeniem liczby komórek, które będziemy musieli przetwarzać*



	A	B	C	D	E	F	G	H	I
1									
2		-5	0	-7	3	-3	7	-6	-9
3		-5	-6	-6	-10	-1	10	9	-10
4		-2	5	1	4	-3	3	-8	-3
5		1	8	-3	-8	1	8	8	6
6		0	-4	-3	3	-1	7	5	2
7		-10	4	1	8	1	-8	7	9
8		5	-4	-1	7	10	-1	8	-3
9		1	4	1	-8	-2	-1	-6	8
10		-8	-3	10	-1	7	6	7	9
11		0	-2	-2	-1	9	7	7	7
12		10	4	7	6	10	-10	10	4
13		-5	-1	9	7	0	8	6	9
14		3	-4	10	-10	9	-9	2	-4
15		4	9	0	8	4	7	-1	-4
16		0	1	9	-9	2	7	-7	0
17									
18									
19									

```
Sub ColorNegative2()
'    Jeżeli wartość jest ujemna, zmienia kolor tła komórki na czerwony
    Dim WorkRange As Range
    Dim cell As Range
    If TypeName(Selection) <> "Range" Then Exit Sub
    Application.ScreenUpdating = False
    Set WorkRange = Application.Intersect(Selection, _
      ActiveSheet.UsedRange)
    For Each cell In WorkRange
        If cell.Value < 0 Then
            cell.Interior.Color = RGB(255, 0, 0)
        Else
            cell.Interior.Color = xlNone
        End If
    Next cell
End Sub
```

Procedura `ColorNegative2` jest znacznie lepsza, ale mimo to nadal nie jest tak wydajna, jak być powinna, a to z prostego powodu — nadal niepotrzebnie przetwarza puste komórki. Trzecia wersja naszej procedury, `ColorNegative3`, jest nieco dłuższa, ale jednocześnie o wiele bardziej wydajna. W celu wygenerowania dwóch podzbiorów zaznaczenia użyłem metody `SpecialCells`. Pierwszy podzbiór obejmuje jedynie komórki zawierające stałe numeryczne (`ConstantCells`), natomiast drugi — komórki przechowujące formuły numeryczne (`FormulaCells`). Komórki obu podzbiorów są następnie przetwarzane za pomocą dwóch konstrukcji `For Each ... Next`. W efekcie przetwarzane są tylko niepuste komórki zawierające wartości numeryczne, dzięki czemu uzyskujemy znaczące zwiększenie szybkości działania makra.

```
Sub ColorNegative3()
'    Jeżeli wartość jest ujemna, zmienia kolor tła komórki na czerwony
    Dim FormulaCells As Range, ConstantCells As Range
    Dim cell As Range
    If TypeName(Selection) <> "Range" Then Exit Sub
```

```
    Application.ScreenUpdating = False

'   Tworzy podzbiory oryginalnego obszaru zaznaczenia
    On Error Resume Next
    Set FormulaCells = Selection.SpecialCells(xlFormulas, xlNumbers)
    Set ConstantCells = Selection.SpecialCells(xlConstants, xlNumbers)
    On Error GoTo 0
'   Przetwarzanie komórek zawierających formuły
    If Not FormulaCells Is Nothing Then
        For Each cell In FormulaCells
            If cell.Value < 0 Then
                cell.Interior.Color = RGB(255, 0, 0)
            Else
                cell.Interior.Color = xlNone
            End If
            Next cell
    End If
'   Przetwarzanie komórek zawierających stałe wartości numeryczne
    If Not ConstantCells Is Nothing Then
        For Each cell In ConstantCells
            If cell.Value < 0 Then
                cell.Interior.Color = RGB(255, 0, 0)
            Else
                cell.Interior.Color = xlNone
            End If
        Next cell
    End If
End Sub
```

Zastosowanie instrukcji `On Error` jest konieczne, ponieważ metoda `SpecialCells` generuje błąd, gdy żadna komórka nie spełnia kryterium.

Skoroszyt z tymi przykładami (*Tworzenie wydajnych pętli.xlsm*), zawierający trzy wersje procedury `ColorNegative`, znajdziesz na płycie CD-ROM dołączonej do książki.

Usuwanie wszystkich pustych wierszy

Poniższa procedura usuwa puste wiersze aktywnego arkusza. Procedura jest szybka i wydajna, ponieważ nie sprawdza wszystkich wierszy, a jedynie wiersze używane przez zakres, który jest identyfikowany za pomocą właściwości `UsedRange` obiektu `Worksheet`.

```
Sub DeleteEmptyRows()
    Dim LastRow As Long
    Dim r As Long
    Dim Counter As Long
    Application.ScreenUpdating = False
    LastRow = ActiveSheet.UsedRange.Rows.Count + _
      ActiveSheet.UsedRange.Rows(1).Row - 1
    For r = LastRow To 1 Step -1
        If Application.WorksheetFunction.CountA(Rows(r)) = 0 Then
            Rows(r).Delete
            Counter = Counter + 1
        End If
    Next r
```

```
    Application.ScreenUpdating = True
    MsgBox Counter & " pustych wierszy zostało usuniętych."
End Sub
```

Pierwszym krokiem jest określenie ostatnio używanego wiersza, a następnie przypisanie jego numeru zmiennej `LastRow`. Nie jest to takie proste, jak mogłoby się wydawać, ponieważ używany zakres może, ale nie musi rozpoczynać się od wiersza `1`. A zatem wartość zmiennej `LastRow` jest obliczana poprzez określenie liczby wierszy używanego zakresu, dodanie numeru pierwszego wiersza zakresu i odjęcie jedynki.

W celu stwierdzenia, czy wiersz jest pusty, procedura korzysta z funkcji arkuszowej `COUNTA` (`ILE.NIEPUSTYCH`) Excela. Jeżeli dla określonego wiersza funkcja zwróci wartość `0`, oznacza to, że jest pusty. Procedura przetwarza wiersze od dołu do góry, a ponadto w pętli `For ... Next` używa ujemnej wartości skoku (`Step`). Jest to niezbędne, ponieważ operacja usuwania wierszy powoduje, że wszystkie kolejne wiersze są przesuwane w górę arkusza. Jeżeli pętla przetwarzałaby wiersze od góry do dołu, jej licznik po usunięciu wiersza nie miałby właściwej wartości.

Makro wykorzystuje również inną zmienną, `Counter`, do śledzenia liczby usuniętych wierszy. Liczba ta jest wyświetlana w oknie dialogowym, kiedy procedura kończy działanie.

Skoroszyt z tym przykładem (*Usuń puste wiersze.xlsm*) znajdziesz na płycie CD-ROM dołączonej do książki.

Powielanie wierszy

Przykład, który omówimy w tym podrozdziale, ilustruje sposób wykorzystania VBA do tworzenia duplikatów istniejących wierszy. Na rysunku 11.9 przedstawiono wygląd skoroszytu zawierającego informacje o loterii biurowej. Kolumna A przechowuje imiona graczy, kolumna B liczbę losów zakupionych przez poszczególne osoby, a w kolumnie C znajduje się liczba losowa (wygenerowana przy użyciu funkcji `RAND`). Zwycięzca zostanie wyłoniony przez sortowanie danych w kolumnie C (wygrywa osoba, do której została przypisana największa liczba losowa).

Rysunek 11.9. *Zadanie polega na powieleniu istniejących wierszy w oparciu o wartości z kolumny B*

	A	B	C	D
1	**Imię**	**Liczba biletów**	**Liczba losowa**	
2	Adam	1	0,979059879	
3	Barbara	2	0,810828269	
4	Czesław	1	0,555275531	
5	Dariusz	5	0,984923992	
6	Franek	3	0,661198742	
7	Grzegorz	1	0,983915967	
8	Hubert	1	0,908052993	
9	Inka	2	0,091112702	
10	Marek	1	0,182825299	
11	Norbert	10	0,824819877	
12	Paweł	2	0,907980166	
13	Rafał	1	0,322145504	
14	Wiesław	2	0,925320968	
15				

Nasze zadanie polega na powieleniu istniejących wierszy dla poszczególnych osób, tak aby każda z nich miała tyle osobnych wierszy, ile zakupiła losów. Na przykład Barbara kupiła 2 losy, a zatem powinniśmy dla niej utworzyć 2 osobne wiersze. Kod procedury realizującej takie zadanie został zamieszczony poniżej:

```
Sub DupeRows()
  Dim cell As Range
' Pierwsza komórka z liczbą losów
  Set cell = Range("B2")
  Do While Not IsEmpty(cell)
    If cell > 1 Then
      Range(cell.Offset(1, 0), cell.Offset(cell.Value - 1, _
        0)).EntireRow.Insert
      Range(cell, cell.Offset(cell.Value - 1, 1)).EntireRow.FillDown
    End If
    Set cell = cell.Offset(cell.Value, 0)
    Loop
End Sub
```

Zmiennej obiektowej `cell` zostaje przypisana wartość reprezentująca komórkę `B2`, czyli pierwszą komórkę, w której przechowywana jest liczba zakupionych losów. Pętla wstawia nowy wiersz i następnie kopiuje go odpowiednią ilość razy przy użyciu metody `FillDown`. Zmienna `cell` jest inkrementowana tak, aby wskazywała na liczbę losów zakupionych przez kolejną osobę i procedura kontynuuje działanie aż do momentu napotkania pierwszej pustej komórki. Na rysunku 11.10 przedstawiono wygląd arkusza po zakończeniu działania procedury.

Skoroszyt z tym przykładem (*Powielanie wierszy.xlsm*) znajdziesz na płycie CD-ROM dołączonej do książki.

Określanie, czy zakres zawiera się w innym zakresie

Poniższa funkcja `InRange` pobiera dwa argumenty (obiekty klasy `Range`) i zwraca wartość `True`, jeżeli pierwszy zakres zawiera się w drugim:

```
Function InRange(rng1, rng2) As Boolean
'   Zwraca wartość True, jeżeli rng1 jest podzbiorem rng2
    InRange = False
    If rng1.Parent.Parent.Name = rng2.Parent.Parent.Name Then
        If rng1.Parent.Name = rng2.Parent.Name Then
            If Union(rng1, rng2).Address = rng2.Address Then
                InRange = True
            End If
        End If
    End If
End Function
```

Kod funkcji `InRange` może się wydawać dość złożony, ponieważ program musi sprawdzić, czy dwa zakresy znajdują się w tym samym arkuszu i skoroszycie. Procedura posługuje się właściwością `Parent` zwracającą kontener obiektu. Przykładowo poniższe wyrażenie zwraca nazwę arkusza będącego kontenerem obiektu `rng1`, do którego jest wykonywane odwołanie:

```
rng1.Parent.Name
```

Rysunek 11.10. *Procedura dodała do arkusza nowe wiersze w oparciu o wartości w kolumnie B*

	A	B	C	D
1	**Imię**	**Liczba biletów**	**Liczba losowa**	
2	Adam	1	0,473658071	
3	Barbara	2	0,138727101	
4	Barbara	2	0,301689446	
5	Czesław	1	0,828764305	
6	Dariusz	5	0,682335281	
7	Dariusz	5	0,820466168	
8	Dariusz	5	0,765595632	
9	Dariusz	5	0,677830322	
10	Dariusz	5	0,49595584	
11	Franek	3	0,979723132	
12	Franek	3	0,950836689	
13	Franek	3	0,065496878	
14	Grzegorz	1	0,650184613	
15	Hubert	1	0,433638419	
16	Inka	2	0,6339003	
17	Inka	2	0,329331655	
18	Marek	1	0,935974583	
19	Norbert	10	0,301024128	
20	Norbert	10	0,965132769	
21	Norbert	10	0,217322826	
22	Norbert	10	0,718978857	
23	Norbert	10	0,105206684	
24	Norbert	10	0,469813354	
25	Norbert	10	0,869915296	
26	Norbert	10	0,657643784	
27	Norbert	10	0,306288531	
28	Norbert	10	0,31825869	
29	Paweł	2	0,028797204	
30	Paweł	2	0,665602312	
31	Rafał	1	0,27125745	
32	Wiesław	2	0,103203069	
33	Wiesław	2	0,370178675	
34				

Kolejne wyrażenie zwraca nazwę skoroszytu obiektu `rng1`:

```
rng1.Parent.Parent.Name
```

Funkcja `Union` języka VBA zwraca obiekt klasy `Range` reprezentujący sumę dwóch obiektów `Range`. Zakres sumy obejmuje wspólne komórki dwóch zakresów. Jeżeli adres sumy dwóch zakresów jest taki sam jak adres drugiego zakresu, oznacza to, że pierwszy zakres zawiera się w drugim.

Skoroszyt z tym przykładem (*Funkcja InRange.xlsm*) znajdziesz na płycie CD-ROM dołączonej do książki.

Określanie typu danych zawartych w komórce

Excel oferuje kilka wbudowanych funkcji arkuszowych, które pomagają w określaniu typu danych zawartych w komórce. Należy do nich zaliczyć funkcje `CZY.TEKST`, `CZY.LOGICZNA` i `CZY.BŁĄD`. Dodatkowo język VBA zawiera funkcje `IsEmpty`, `IsDate` i `IsNumeric`.

Zamieszczona poniżej funkcja `CellType` akceptuje argument `Range` i zwraca łańcuch (`Pusta`, `Tekst`, `Logiczny`, `Błąd`, `Data`, `Czas` lub `Liczba`) opisujący typ danych zawartych w górnej lewej komórce zakresu. Funkcja może zostać użyta w formule arkusza lub wywołana z innej procedury języka VBA.

```
Function CellType(Rng)
'    Zwraca typ górnej lewej komórki zakresu
    Dim TheCell As Range
    Set TheCell = Rng.Range("A1")
    Select Case True
        Case IsEmpty(TheCell)
             CellType = "Pusta"
        Case Application.IsText(TheCell)
             CellType = "Tekst"
        Case Application.IsLogical(TheCell)
             CellType = "Logiczny"
        Case Application.IsErr(TheCell)
             CellType = "Błąd"
        Case IsDate(TheCell)
             CellType = "Data"
        Case InStr(1, TheCell.Text, ":") <> 0
             CellType = "Czas"
        Case IsNumeric(TheCell)
             CellType = "Liczba"
    End Select
End Function
```

Zwróć uwagę na użycie polecenia `Set TheCell`. Funkcja `CellType` akceptuje argument `Range` dowolnej wielkości, ale ta instrukcja powoduje, że funkcja przetwarza tylko górną lewą komórkę zakresu reprezentowanego przez zmienną `TheCell`.

Skoroszyt z tym przykładem (*Funkcja CellType.xlsm*) znajdziesz na płycie CD-ROM dołączonej do książki.

Odczytywanie i zapisywanie zakresów

Wiele operacji wykonywanych w arkuszach kalkulacyjnych wymaga przenoszenia wartości z tablicy do zakresu lub z zakresu do tablicy. Z jakiegoś powodu Excel o wiele szybciej odczytuje dane z zakresu, niż je w nim zapisuje. Procedura `WriteReadRange`, której kod przedstawiamy poniżej, demonstruje porównanie względnych szybkości wykonywania operacji zapisu i odczytu zakresu.

Procedura tworzy tablicę, a następnie za pomocą pętli `For ... Next` zapisuje jej zawartość w zakresie i ponownie wczytuje ją do tablicy. Przy użyciu funkcji `Timer` języka VBA oblicza czas wymagany do wykonania każdej operacji.

```
Sub WriteReadRange()
    Dim MyArray()
    Dim Time1 As Double
    Dim NumElements As Long, i As Long
    Dim WriteTime As String, ReadTime As String
    Dim Msg As String
```

```
    NumElements = 60000
    ReDim MyArray(1 To NumElements)

'   Wypełnienie tablicy
    For i = 1 To NumElements
        MyArray(i) = i
    Next i

'   Zapis danych z tablicy do zakresu
    Time1 = Timer
    For i = 1 To NumElements
        Cells(i, 1) = MyArray(i)
    Next i
    WriteTime = Format(Timer - Time1, "00:00")

'   Odczytanie danych z zakresu i załadowanie do tablicy
    Time1 = Timer
    For i = 1 To NumElements
        MyArray(i) = Cells(i, 1)
    Next i
    ReadTime = Format(Timer - Time1, "00:00")

'   Wyświetlenie wyników
    Msg = "Czas zapisu: " & WriteTime
    Msg = Msg & vbCrLf
    Msg = Msg & "Czas odczytu: " & ReadTime
    MsgBox Msg, vbOKOnly, NumElements & " elementów"
End Sub
```

Na moim komputerze przepisanie tablicy liczącej 60 000 elementów do zakresu komórek zajęło 58 sekund, natomiast wczytanie zakresu komórek do tablicy tylko 1 sekundę.

Lepsza metoda zapisywania zakresu

W poprzednim przykładzie, aby przenieść zawartość tablicy do zakresu arkusza, użyto pętli `For ... Next`. W tym podrozdziale zademonstrujemy wydajniejszą metodę osiągnięcia tego samego celu.

Kod procedury przedstawiony poniżej ilustruje najbardziej oczywisty (ale niestety nie najwydajniejszy) sposób wypełniania zakresu danymi. Do umieszczenia danych w zakresie ponownie została użyta pętla `For ... Next`.

```
Sub LoopFillRange()
'   Wypełnia zakres przy użyciu pętli przetwarzającej komórki
    Dim CellsDown As Long, CellsAcross As Integer
    Dim CurrRow As Long, CurrCol As Integer
    Dim StartTime As Double
    Dim CurrVal As Long

'   Pobranie wymiarów
    CellsDown = Val(InputBox("Ile komórek w pionie?"))
    If CellsDown = 0 Then Exit Sub
    CellsAcross = Val(InputBox("Ile komórek w poziomie?"))
    If CellsAcross = 0 Then Exit Sub
```

```
    ' Zarejestrowanie czasu rozpoczęcia
    StartTime = Timer

    ' Przy użyciu pętli przetwarza komórki i wstawia wartości
    CurrVal = 1
    Application.ScreenUpdating = False
    For CurrRow = 1 To CellsDown
        For CurrCol = 1 To CellsAcross
            ActiveCell.Offset(CurrRow - 1, _
            CurrCol - 1).Value = CurrVal
            CurrVal = CurrVal + 1
        Next CurrCol
    Next CurrRow

    ' Wyświetla czas trwania operacji
    Application.ScreenUpdating = True
    MsgBox Format(Timer - StartTime, "00.00") & " sekund"
End Sub
```

Kolejny przykład demonstruje o wiele szybszą metodę pozwalającą na uzyskanie tego samego efektu. Procedura wstawia do tablicy poszczególne wartości, a następnie za pomocą jednej instrukcji przenosi zawartość tablicy do zakresu.

```
Sub ArrayFillRange()
    ' Wypełnienie zakresu poprzez transfer tablicy
    Dim CellsDown As Long, CellsAcross As Integer
    Dim i As Long, j As Integer
    Dim StartTime As Double
    Dim TempArray() As Long
    Dim TheRange As Range
    Dim CurrVal As Long
    ' Pobranie wymiarów
    CellsDown = Val(InputBox("Ile komórek w pionie?"))
    If CellsDown = 0 Then Exit Sub
    CellsAcross = Val(InputBox("Ile komórek w poziomie?"))
    If CellsAcross = 0 Then Exit Sub
    ' Zarejestrowanie czasu rozpoczęcia
    StartTime = Timer
    ' Przeskalowanie  wymiarów tablicy tymczasowej
    ReDim TempArray(1 To CellsDown, 1 To CellsAcross)
    ' Zdefiniowanie zakresu w arkuszu
    Set TheRange = ActiveCell.Range(Cells(1, 1), _
        Cells(CellsDown, CellsAcross))
    ' Wypełnienie tablicy tymczasowej
    CurrVal = 0
    Application.ScreenUpdating = False
    For i = 1 To CellsDown
        For j = 1 To CellsAcross
            TempArray(i, j) = CurrVal + 1
            CurrVal = CurrVal + 1
        Next j
    Next i
    ' Transfer tablicy tymczasowej do arkusza
    TheRange.Value = TempArray
    ' Wyświetlanie czas trwania operacji
    Application.ScreenUpdating = True
    MsgBox Format(Timer - StartTime, "00.00") & " sekund"
End Sub
```

W moim systemie wypełnienie zakresu o wymiarach 1000×250 komórek (250 000 komórek) przy użyciu pętli zajęło 10,05 sekundy. Metoda oparta na transferze zawartości tablicy do uzyskania identycznego efektu potrzebowała zaledwie 0,18 sekundy, czyli działała około 50 razy szybciej! Jaki wniosek wynika z tego przykładu? Jeżeli chcesz przenieść do arkusza dużą ilość danych, wszędzie, gdzie tylko jest to możliwe, unikaj stosowania pętli.

Osiągnięte czasy w dużej mierze zależą od obecności w arkuszu innych formuł. W praktyce lepsze czasy otrzymasz w sytuacji, kiedy podczas testu nie będą otwarte inne skoroszyty zawierające makra lub kiedy przełączysz Excela w tryb ręcznego przeliczania arkusza.

Na płycie CD-ROM dołączonej do książki znajdziesz skoroszyt *Wypełnianie zakresu przy użyciu pętli i tablicy.xlsm*, zawierający procedury `WriteReadRange`, `LoopFillRange` oraz `ArrayFillRange`.

Przenoszenie zawartości tablic jednowymiarowych

W poprzednim przykładzie zastosowano tablicę dwuwymiarową, dobrze sprawdzającą się w arkuszach, w których dane przechowywane są w uporządkowanej strukturze wierszy i kolumn.

Aby przenieść zawartość tablicy jednowymiarowej, zakres musi mieć orientację poziomą — czyli inaczej mówiąc, posiadać jeden wiersz z wieloma *kolumnami*. Jeżeli jednak musisz użyć zakresu pionowego, najpierw będziesz musiał dokonać transponowania tablicy z poziomej na pionową. Aby to zrobić, możesz użyć funkcji arkuszowej `TRANSPOSE` (`TRANSPONUJ`) Excela. Poniższe polecenie przenosi tablicę liczącą 100 elementów do pionowego zakresu arkusza (`A1:A100`):

```
Range("A1:A100").Value = Application.WorksheetFunction.Transpose(MyArray)
```

Funkcja arkuszowa `TRANSPONUJ` nie będzie działała z tablicami, w których przechowywanych jest więcej niż 65 536 elementów.

Przenoszenie zawartości zakresu do tablicy typu Variant

W tym podrozdziale omówimy kolejną metodę przetwarzania zawartości arkusza przy użyciu języka VBA. W poniższym przykładzie zawartość zakresu komórek jest przenoszona do dwuwymiarowej tablicy typu `Variant`. Następnie w oknach komunikatów są wyświetlane górne granice każdego wymiaru tablicy.

```
Sub RangeToVariant()
    Dim x As Variant
    x = Range("A1:L600").Value
    MsgBox UBound(x, 1)
    MsgBox UBound(x, 2)
End Sub
```

W pierwszym oknie komunikatu jest wyświetlana wartość 600 (liczba wierszy oryginalnego zakresu), natomiast w drugim — 12 (liczba kolumn). Jak sam się przekonasz, przeniesienie zawartości zakresu do tablicy typu `Variant` odbywa się prawie natychmiast.

Poniższa procedura wczytuje zawartość zakresu o nazwie `data` do tablicy typu `Variant`, wykonuje na poszczególnych elementach tablicy prostą operację mnożenia, a następnie ponownie przenosi dane zapisane w tablicy do zakresu.

```
Sub RangeToVariant2()
    Dim x As Variant
    Dim r As Long, c As Integer
'   Wczytanie danych do tablicy typu Variant
    x = Range("data").Value

'   Wykonanie pętli dla tablicy typu Variant
    For r = 1 To UBound(x, 1)
        For c = 1 To UBound(x, 2)
'           Mnożenie kolejnych elementów tablicy przez 2
            x(r, c) = x(r, c) * 2
        Next c
    Next r
'   Ponowne przeniesienie zawartości tablicy typu Variant do arkusza
    Range("data") = x
End Sub
```

Jak sam się możesz przekonać, cała procedura działa naprawdę szybko. Przetwarzanie 30 000 komórek na moim komputerze trwało poniżej jednej sekundy.

Skoroszyt z tym przykładem (*Przenoszenie tablicy typu Variant.xlsm*) znajdziesz na płycie CD-ROM dołączonej do książki.

Zaznaczanie komórek na podstawie wartości

Nasz kolejny przykład ilustruje, w jaki sposób można zaznaczać wybrane komórki w oparciu o ich wartości. Co ciekawe, Excel nie posiada swojego własnego mechanizmu, który umożliwiałby bezpośrednie wykonanie takiej operacji. Poniżej przedstawiamy kod procedury `SelectByValue`, której zadaniem jest zaznaczenie komórek zakresu zawierających wartości ujemne, aczkolwiek można to w łatwy sposób zmodyfikować i dopasować do własnych potrzeb.

```
Sub SelectByValue()
    Dim Cell As Object
    Dim FoundCells As Range
    Dim WorkRange As Range

    If TypeName(Selection) <> "Range" Then Exit Sub

'   Sprawdzamy wszystko czy tylko zaznaczenie
    If Selection.CountLarge = 1 Then
        Set WorkRange = ActiveSheet.UsedRange
    Else
       Set WorkRange = Application.Intersect(Selection, ActiveSheet.UsedRange)
    End If

'   Zredukuj liczbę przetwarzanych komórek do komórek zawierających wartości numeryczne
    On Error Resume Next
    Set WorkRange = WorkRange.SpecialCells(xlConstants, xlNumbers)
```

```
    If WorkRange Is Nothing Then Exit Sub
    On Error GoTo 0

'   Sprawdzaj w pętli kolejne komórki i dodawaj do zakresu FoundCells, jeżeli spełniają kryterium
    For Each Cell In WorkRange
        If Cell.Value < 0 Then
           If FoundCells Is Nothing Then
               Set FoundCells = Cell
           Else
               Set FoundCells = Union(FoundCells, Cell)
           End If
        End If
    Next Cell
'   Pokaż komunikat lub zaznacz komórki
    If FoundCells Is Nothing Then
       MsgBox "Nie znaleziono komórek spełniających kryterium."
    Else
        FoundCells.Select
    End If
End Sub
```

Procedura rozpoczyna działanie od sprawdzenia zaznaczonego zakresu. Jeżeli jest to pojedyncza komórka, przeszukiwany jest cały arkusz. Jeżeli zaznaczone zostały co najmniej 2 komórki, wtedy przeszukiwany jest tylko zaznaczony zakres. Zakres, który będzie przeszukiwany, jest następnie redefiniowany poprzez użycie metody `SpecialCells` do utworzenia obiektu klasy `Range`, który składa się tylko z komórek zawierających wartości numeryczne.

Kod w pętli `For ... Next` sprawdza wartości kolejnych komórek. Jeżeli komórka spełnia zadane kryterium (czyli jej zawartość jest mniejsza od 0), komórka jest dodawana za pomocą metody `Union` do obiektu `FoundCells` klasy `Range`. Zwróć uwagę na fakt, że nie możesz użyć metody `Union` dla pierwszej komórki — jeżeli zakres `FoundCells` nie zawiera żadnych komórek, próba użycia metody `Union` spowoduje wygenerowanie błędu. Właśnie dlatego w naszym programie zamieściliśmy kod sprawdzający, czy zawartość zakresu `FoundCells` to `Nothing`.

Kiedy pętla kończy swoje działanie, obiekt `FoundCells` składa się z komórek, które spełniają kryterium wyszukiwania (jeżeli nie znaleziono żadnych komórek, jego zawartością będzie `Nothing`). Jeżeli żadna komórka nie spełni kryteriów, na ekranie zostanie wyświetlone okno dialogowe z odpowiednim komunikatem. W przeciwnym razie komórki spełniające kryteria zostaną zaznaczone.

Skoroszyt z tym przykładem (*Zaznaczanie według wartości.xlsm*) znajdziesz na płycie CD-ROM dołączonej do książki.

Kopiowanie nieciągłego zakresu komórek

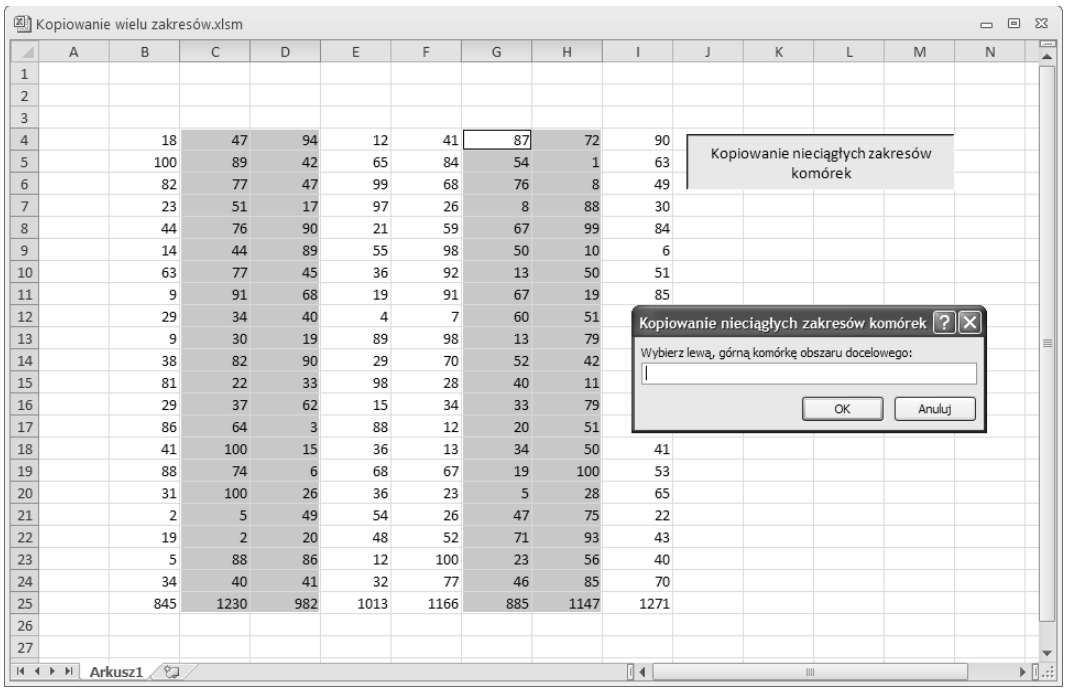
Jeżeli kiedykolwiek próbowałeś kopiować nieciągły zakres komórek, z pewnością przekonałeś się, że Excel nie obsługuje takiej operacji. Próba jej wykonania kończy się wyświetleniem komunikatu *Wykonanie tego polecenia dla kilku zakresów nie jest możliwe*.

Jedynym wyjątkiem jest tutaj sytuacja, w której próbujesz kopiować zaznaczenie wielokrotne, składające się z całych wierszy lub kolumn. Excel *pozwala* na wykonanie takiej operacji.

Kiedy napotykasz takie ograniczenia w programie Excel, zazwyczaj możesz je obejść za pomocą odpowiedniego makra. Procedura, którą omówimy poniżej, jest przykładem takiego makra, które pozwala na kopiowanie wielu zaznaczonych zakresów komórek w inne miejsce arkusza.

```
Sub CopyMultipleSelection()
    Dim SelAreas() As Range
    Dim PasteRange As Range
    Dim UpperLeft As Range
    Dim NumAreas As Long, i As Long
    Dim TopRow As Long, LeftCol As Long
    Dim RowOffset As Long, ColOffset As Long

    If TypeName(Selection) <> "Range" Then Exit Sub

'   Zapisuje poszczególne zakresy jako osobne obiekty klasy Range
    NumAreas = Selection.Areas.Count
    ReDim SelAreas(1 To NumAreas)
    For i = 1 To NumAreas
        Set SelAreas(i) = Selection.Areas(i)
    Next

'   Określa górną, lewą komórkę zaznaczonych obszarów
    TopRow = ActiveSheet.Rows.Count
    LeftCol = ActiveSheet.Columns.Count
    For i = 1 To NumAreas
        If SelAreas(i).Row < TopRow Then TopRow = SelAreas(i).Row
        If SelAreas(i).Column < LeftCol Then LeftCol = SelAreas(i).Column
    Next
    Set UpperLeft = Cells(TopRow, LeftCol)

'   Pobiera adres obszaru docelowego
    On Error Resume Next
    Set PasteRange = Application.InputBox _
      (Prompt:="Podaj adres lewej, górnej komórki obszaru docelowego: ", _
      Title:="Kopiowanie wielu zakresów", _
      Type:=8)
    On Error GoTo 0
'   Jeżeli operacja została anulowana, zakończ działanie
    If TypeName(PasteRange) <> "Range" Then Exit Sub

'   Upewnij się, że używamy tylko lewej, górnej komórki
    Set PasteRange = PasteRange.Range("A1")

'   Kopiowanie i wklejanie poszczególnych zakresów
    For i = 1 To NumAreas
        RowOffset = SelAreas(i).Row - TopRow
        ColOffset = SelAreas(i).Column - LeftCol
        SelAreas(i).Copy PasteRange.Offset(RowOffset, ColOffset)
    Next i
End Sub
```

Na rysunku 11.11 przedstawiono okno dialogowe, w którym użytkownik powinien zdefiniować adres obszaru docelowego.



Rysunek 11.11. *Zastosowanie metody InputBox programu Excel do pobierania lokalizacji komórki*

Skoroszyt z tym przykładem (*Kopiowanie wielu zakresów.xlsm*) znajdziesz na płycie CD-ROM dołączonej do książki. Dodatkowo w skoroszycie tym znajduje się inna wersja tej procedury, która ostrzega użytkownika, jeżeli w wyniku kopiowania zostaną nadpisane dane istniejące w obszarze docelowym.

Przetwarzanie skoroszytów i arkuszy

Kolejne przykłady omawiane w tym podrozdziale demonstrują metody przetwarzania skoroszytów i arkuszy przy użyciu języka VBA.

Zapisywanie wszystkich skoroszytów

Poniższa procedura przy użyciu pętli przetwarza wszystkie skoroszyty w kolekcji `Workbooks` i ponownie zapisuje każdy plik, który był już wcześniej zapisany:

```
Public Sub SaveAllWorkbooks()
    Dim Book As Workbook
    For Each Book In Workbooks
        If Book.Path <> "" Then Book.Save
    Next Book
End Sub
```

Zwróć uwagę na właściwość `Path`. Jeżeli właściwość `Path` danego skoroszytu jest pusta, oznacza to, że jego plik nigdy nie został zapisany (jest to nowy skoroszyt). Procedura ignoruje tego typu skoroszyty i zapisuje tylko te, których właściwość `Path` ma ustawioną dowolną wartość.

Zapisywanie i zamykanie wszystkich skoroszytów

Poniższa procedura przy użyciu pętli przetwarza kolekcję `Workbooks`, zapisując i zamykając wszystkie skoroszyty:

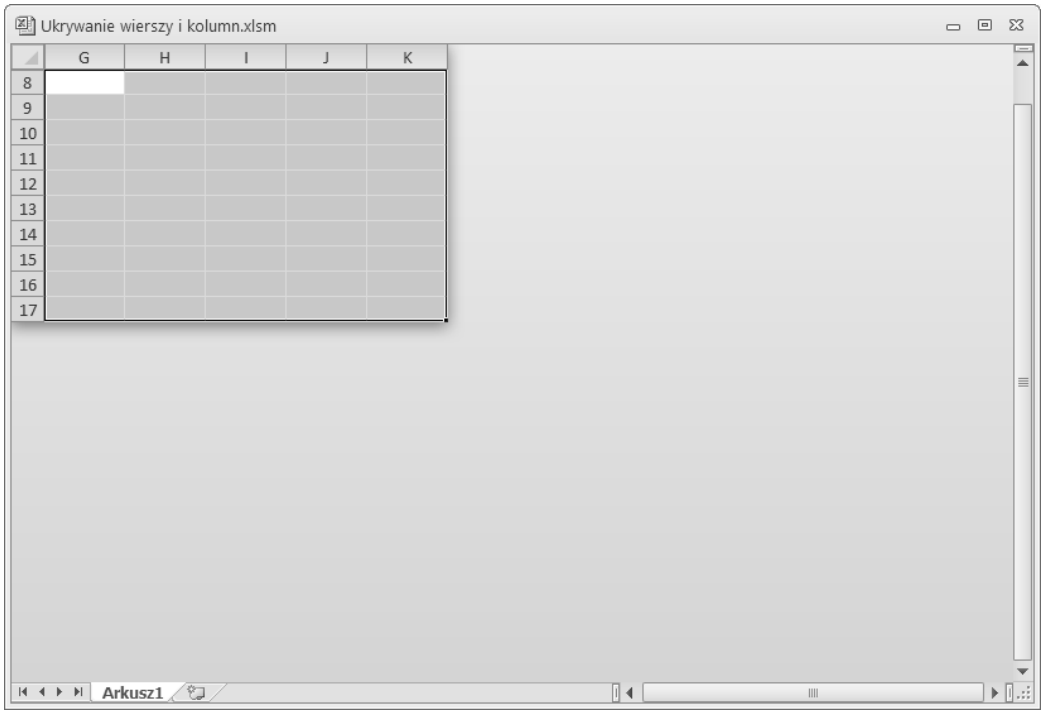
```
Sub CloseAllWorkbooks()
    Dim Book As Workbook
    For Each Book In Workbooks
        If Book.Name <> ThisWorkbook.Name Then
            Book.Close savechanges:=True
        End If
    Next Book
    ThisWorkbook.Close savechanges:=True
End Sub
```

Aby określić, czy dany skoroszyt to skoroszyt zawierający kod aktualnie wykonywanej procedury, nasza procedura używa instrukcji `If` umieszczonej w pętli `For-Next`. Jest to konieczne, ponieważ zamknięcie takiego skoroszytu spowoduje przerwanie wykonywania kodu, na skutek czego inne skoroszyty nie zostaną zapisane. Po zamknięciu wszystkich innych skoroszytów procedura zamyka również swój macierzysty skoroszyt.

Ukrywanie wszystkich komórek arkusza poza zaznaczonym zakresem

Procedura, którą omówimy w tym podrozdziale, ukrywa wszystkie komórki arkusza poza tymi, które znajdują się w aktualnie zaznaczonym zakresie. Przykład takiej sytuacji przedstawiono na rysunku 11.12.

```
Sub HideRowsAndColumns()
    Dim row1 As Long, row2 As Long
    Dim col1 As Long, col2 As Long

    If TypeName(Selection) <> "Range" Then Exit Sub

'   Jeżeli ostatni wiersz lub kolumna są ukryte, odkryj wszystko i zakończ działanie
    If Rows(Rows.Count).EntireRow.Hidden Or _
      Columns(Columns.Count).EntireColumn.Hidden Then
        Cells.EntireColumn.Hidden = False
        Cells.EntireRow.Hidden = False
        Exit Sub
    End If

    row1 = Selection.Rows(1).Row
    row2 = row1 + Selection.Rows.Count - 1
    col1 = Selection.Columns(1).Column
    col2 = col1 + Selection.Columns.Count - 1
```



Rysunek 11.12. *Wszystkie komórki arkusza poza zaznaczonym zakresem (G8:K17) zostały ukryte*

```
    Application.ScreenUpdating = False
    On Error Resume Next
'   Ukryj wiersze
    Range(Cells(1, 1), Cells(row1 - 1, 1)).EntireRow.Hidden = True
    Range(Cells(row2 + 1, 1), Cells(Rows.Count, 1)).EntireRow.Hidden = True
'   Ukryj kolumny
    Range(Cells(1, 1), Cells(1, col1 - 1)).EntireColumn.Hidden = True
    Range(Cells(1, col2 + 1), Cells(1, Columns.Count)).EntireColumn.Hidden = True
End Sub
```

Jeżeli zaznaczony obszar arkusza składa się z kilku nieciągłych zakresów komórek, bazą do ukrywania wierszy i kolumn jest pierwszy zakres.

Skoroszyt z tym przykładem (*Ukrywanie wierszy i kolumn.xlsm*) znajdziesz na płycie CD-ROM dołączonej do książki.

Synchronizowanie arkuszy

Jeżeli korzystałeś kiedykolwiek ze skoroszytów wieloarkuszowych, to wiesz zapewne, że Excel nie potrafi synchronizować danych z poszczególnych arkuszy skoroszytu. Innymi słowy, nie ma możliwości automatycznego zaznaczenia tego samego zakresu i ustawienia górnej, lewej komórki takiego multizakresu. Poniższe makro języka VBA jako bazy używa aktywnego arkusza, a na pozostałych arkuszach skoroszytu wykonuje następujące operacje:

- zaznacza taki sam zakres, jak w przypadku aktywnego arkusza;
- ustawia taką samą górną lewą komórkę okna, jak w aktywnym arkuszu.

Oto kod źródłowy naszej procedury:

```
Sub SynchSheets()
'   Ustawia we wszystkich arkuszach tę samą aktywną komórkę i lewą, górną komórkę zakresu
    If TypeName(ActiveSheet) <> "Worksheet" Then Exit Sub
    Dim UserSheet As Worksheet, sht As Worksheet
    Dim TopRow As Long, LeftCol As Integer
    Dim UserSel As String

    Application.ScreenUpdating = False
'   Zapamiętuje aktualny arkusz
    Set UserSheet = ActiveSheet

'   Zapisuje informacje z aktywnego arkusza
    TopRow = ActiveWindow.ScrollRow
    LeftCol = ActiveWindow.ScrollColumn
    UserSel = ActiveWindow.RangeSelection.Address

'   Przechodzi w pętli przez poszczególne arkusze
    For Each sht In ActiveWorkbook.Worksheets
        If sht.Visible Then ' pomija ukryte arkusze
            sht.Activate
            Range(UserSel).Select
            ActiveWindow.ScrollRow = TopRow
            ActiveWindow.ScrollColumn = LeftCol
        End If
    Next sht

'   Przywraca oryginalne ustawienie aktywnego arkusza
    UserSheet.Activate
    Application.ScreenUpdating = True
End Sub
```

Skoroszyt z tym przykładem (*Synchronizacja arkuszy.xlsm*) znajdziesz na płycie CD-ROM dołączonej do książki.

Techniki programowania w języku VBA

Następne przykłady ilustrują często używane techniki programowania w języku VBA, które możesz wykorzystać we własnych projektach.

Przełączanie wartości właściwości typu logicznego

Właściwość typu logicznego posiada wartość `True` lub `False`. Najprostsza metoda przełączania wartości takiej właściwości polega na zastosowaniu operatora `Not`. Zostało to pokazane w poniższym przykładzie, w którym przełączana jest wartość właściwości `WrapText` obiektu `Selection`:

```
Sub ToggleWrapText()
'    Włącza lub wyłącza zawijanie tekstu dla zaznaczonych komórek
    If TypeName(Selection) = "Range" Then
        Selection.WrapText = Not ActiveCell.WrapText
    End If
End Sub
```

Oczywiście w razie potrzeby możesz dowolnie zmodyfikować tę procedurę, tak aby przełączała inne właściwości typu logicznego.

Zauważ, że przełączanie jest wykonywane w oparciu o aktywną komórkę. Jeżeli po zaznaczeniu zakresu właściwości komórek mają różne wartości (np. zawartość niektórych komórek jest pogrubiona, a innych nie), taki zakres ma *mieszane* właściwości. W tym przypadku Excel w celu określenia sposobu przełączania wartości posługuje się aktywną komórką. Jeżeli na przykład zawartość aktywnej komórki jest pogrubiona, po naciśnięciu przycisku na Wstążce pogrubienie zostanie usunięte z wszystkich komórek zaznaczenia. Ta prosta procedura naśladuje zachowanie Excela, co zazwyczaj jest najlepszym rozwiązaniem.

Zauważ również, że do sprawdzenia, czy zaznaczono zakres, procedura wykorzystuje funkcję `TypeName`. Jeżeli zaznaczony obszar nie jest zakresem, nie zostanie wykonana żadna operacja.

Operator `Not` umożliwia przełączanie wartości wielu innych właściwości. Aby na przykład w arkuszu wyświetlić lub ukryć nagłówki wierszy i kolumn, należy użyć następującej instrukcji:

```
ActiveWindow.DisplayHeadings = Not ActiveWindow.DisplayHeadings
```

Aby w aktywnym arkuszu wyświetlić lub ukryć linie siatki, należy użyć następującej instrukcji:

```
ActiveWindow.DisplayGridlines = Not ActiveWindow.DisplayGridlines
```

Określanie liczby drukowanych stron

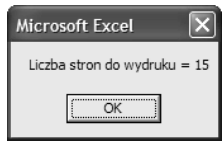
W celu określenia liczby drukowanych stron arkusza można użyć polecenia *Podgląd wydruku* Excela i w dolnej części ekranu sprawdzić liczbę stron. Procedura języka VBA przedstawiona poniżej zlicza poziome i pionowe linie podziału stron i na tej podstawie określa liczbę drukowanych stron aktywnego arkusza:

```
Sub PageCount()
    MsgBox (ActiveSheet.HPageBreaks.Count + 1) * _
      (ActiveSheet.VPageBreaks.Count + 1)
End Sub
```

Kolejna procedura języka VBA przechodzi w pętli przez wszystkie arkusze aktywnego skoroszytu i wyświetla całkowitą liczbę drukowanych stron, tak jak to zostało zilustrowane na rysunku 11.13.

```
Sub ShowPageCount()
    Dim PageCount As Integer
    Dim sht As Worksheet
```

Rysunek 11.13. *Wykorzystanie procedury VBA do wyznaczenia liczby stron do wydruku w skoroszycie*




```
    PageCount = 0
    For Each sht In Worksheets
       PageCount = PageCount + (sht.HPageBreaks.Count + 1) * _
           (sht.VPageBreaks.Count + 1)
    Next sht
    MsgBox "Liczba stron do wydruku = " & PageCount
End Sub
```

Skoroszyt z tym przykładem (*Zliczanie stron wydruku.xlsm*) znajdziesz na płycie CD-ROM dołączonej do książki.

Wyświetlanie daty i czasu

Jeżeli znasz już system liczb seryjnych używany przez Excel do przechowywania dat i godzin, nie będziesz miał żadnych problemów z zastosowaniem dat i czasu w procedurach języka VBA.

Procedura `DateAndTime` wyświetla okno komunikatu z aktualną datą i czasem, tak jak to zostało przedstawione na rysunku 11.14. W pasku tytułowym okna wyświetlany jest komunikat odpowiedni dla danej pory dnia.

Rysunek 11.14. *Okno komunikatu wyświetlające datę i czas*

Procedura jako argument funkcji `Format` wykorzystuje funkcję `Date`. Wynikiem działania procedury jest łańcuch zawierający sformatowaną datę. Aby uzyskać tak samo sformatowany czas, użyliśmy podobnego sposobu.

```
Sub DateAndTime()
'   Wyświetla aktualną datę i czas
    Dim TheDate As String, TheTime As String
    Dim Greeting As String
    Dim FullName As String, FirstName As String
    Dim SpaceInName As Integer
    TheDate = Format(Date, "Long date")
    TheTime = Format(Time, "Long time")
'   Określenie powitania w oparciu o czas
    Select Case Time
        Case Is < TimeValue("12:00"):  Greeting = "Dzień dobry "
        Case Is >= TimeValue("17:00"): Greeting = "Dobry wieczór "
        Case Else: Greeting = "Dzień dobry "
    End Select
'   Dodanie do powitania imienia użytkownika
```

```
    FullName = Application.UserName
    SpaceInName = InStr(1, FullName, " ", 1)

'   Obsługa przypadku, gdy nazwa nie zawiera spacji
    If SpaceInName = 0 Then SpaceInName = Len(FullName)
    FirstName = Left(FullName, SpaceInName)
    Greeting = Greeting & FirstName
'   Wyświetlenie komunikatu
    MsgBox TheDate & vbCrLf & TheTime, vbOKOnly, Greeting
End Sub
```

W celu zagwarantowania, że makro będzie poprawnie działało niezależnie od ustawień regionalnych systemu użytkownika, w powyższym przykładzie użyliśmy nazw formatów daty i czasu (`Long Date` i `Long Time`). Oczywiście w razie potrzeby możesz posłużyć się innymi formatami. Aby na przykład wyświetlić datę w formacie *mm/dd/rr*, możesz użyć następującego polecenia:

```
TheDate = Format(Date, "mm/dd/yy")
```

Aby uzależnić treść komunikatu wyświetlanego na pasku tytułowym okna od pory dnia, zastosowaliśmy konstrukcję `Select Case`. Wartości związane z czasem stosowane w języku VBA funkcjonują podobnie jak w Excelu. Jeżeli wartość czasu jest mniejsza od liczby `0,5` (południe), oznacza to, że jest przedpołudnie. Jeżeli z kolei wartość ta jest większa od liczby `0,7083` (godzina 17), oznacza to, że jest wieczór. W innych przypadkach jest popołudnie. Wybraliśmy proste rozwiązanie polegające na zastosowaniu funkcji `TimeValue` języka VBA, która pobierając łańcuch, zwraca wartość czasu.

Kolejna grupa instrukcji identyfikuje imię użytkownika znajdujące się w zakładce *Ogólne* okna dialogowego *Opcje*. W celu zlokalizowania pierwszej spacji zawartej w personaliach użytkownika użyłem funkcji `InStr` języka VBA. Po napisaniu procedury stwierdziłem, że nie uwzględniłem identyfikatora użytkownika, w którym nie występuje spacja. Gdy więc uruchomiłem ją w systemie używanym przez użytkownika *Nobody*, nie zadziałała prawidłowo. Jest to potwierdzeniem tezy, że nie można przewidzieć wszystkiego, i nawet najprostsze procedury mogą nie zadziałać. Nawiasem mówiąc, jeżeli nie zostaną podane personalia użytkownika, Excel zawsze użyje nazwy aktualnie zalogowanego użytkownika. W procedurze problem ten został rozwiązany poprzez przypisanie zmiennej `SpaceInName` długości pełnej nazwy użytkownika, tak aby funkcja `Left` zwróciła odpowiednią nazwę.

Funkcja `MsgBox` łączy datę i czas, a ponadto w celu wstawienia pomiędzy nimi znaku podziału stosuje wbudowaną stałą `vbCrLf`. Stała `vbOKOnly` jest predefiniowaną stałą zwracającą zero i powodującą, że w oknie komunikatu zostanie wyświetlony jedynie przycisk *OK*. Ostatni argument o nazwie `Greeting` został wcześniej zdefiniowany w procedurze.

Skoroszyt z tym przykładem (*Data i czas.xlsm*) znajdziesz na płycie CD-ROM dołączonej do książki.

Pobieranie listy czcionek

Jeżeli będziesz chciał pobrać listę wszystkich zainstalowanych czcionek, przekonasz się, że Excel nie oferuje bezpośredniej metody uzyskania takiej listy. Technika opisywana tutaj wykorzystuje fakt, że ze względu na konieczność zachowania kompatybilności

z poprzednimi wersjami, Excel 2010 nadal obsługuje stare metody i właściwości obiektów `CommandBar`, które w wersjach wcześniejszych niż 2007 były wykorzystywane do pracy z paskami narzędzi i menu.

Procedura `ShowInstalledFonts` wyświetla w kolumnie `A` aktywnego arkusza listę zainstalowanych czcionek. Procedura tworzy tymczasowy pasek narzędzi (obiekt klasy `Command` ➥`Bar`), dodaje do niego formant `Font` i odczytuje listę czcionek z właściwości tego formantu. Po zakończeniu tymczasowy pasek narzędzi jest usuwany.

```
Sub ShowInstalledFonts()
    Dim FontList As CommandBarControl
    Dim TempBar As CommandBar
    Dim i As Long

    ' Tworzy tymczasowy pasek narzędzi (obiekt klasy CommandBar)
    Set TempBar = Application.CommandBars.Add
    Set FontList = TempBar.Controls.Add(ID:=1728)

    ' Umieszcza listę czcionek w kolumnie A
    Range("A:A").ClearContents
    For i = 0 To FontList.ListCount - 1
        Cells(i + 1, 1) = FontList.List(i + 1)
    Next i

    ' Usuwa tymczasowy pasek narzędzi (obiekt klasy CommandBar)
    TempBar.Delete
End Sub
```

Opcjonalnie można też wyświetlić nazwę czcionki przy użyciu tej czcionki, tak jak to zostało zaprezentowane na rysunku 11.15. Aby to zrobić, wewnątrz pętli `For ... Next` należy umieścić następującą instrukcję:

```
Cells(i+1, 1).Font.Name = FontList.List(i + 1)
```

Trzeba jednak mieć świadomość, że zastosowanie w skoroszycie wielu czcionek spowoduje zużycie znacznej ilości zasobów systemowych, a nawet może doprowadzić do zawieszenia komputera.

Skoroszyt z tym przykładem (*Lista czcionek.xlsm*) znajdziesz na płycie CD-ROM dołączonej do książki.

Sortowanie tablicy

Co prawda Excel posiada wbudowane polecenie sortujące zakresy arkusza, ale język VBA nie dysponuje metodą sortowania tablic. Skuteczne, ale niewygodne rozwiązanie tego problemu polega na przeniesieniu zawartości tablicy do zakresu arkusza, posortowaniu jej przy użyciu polecenia Excela, a następnie wczytaniu wyniku do tablicy. Jeżeli jednak szybkość odgrywa dużą rolę, lepiej stworzyć w języku VBA procedurę sortującą.

W tym punkcie omówimy cztery różne metody sortowania:

- *Sortowanie arkuszowe* polega na przeniesieniu zawartości tablicy do zakresu arkusza, posortowaniu jej, a następnie ponownym umieszczeniu w tablicy. Jedynym argumentem procedury opartej na tej metodzie jest tablica.

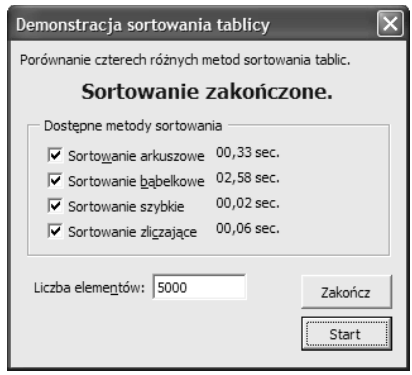
Rysunek 11.15.
Wyświetlanie listy czcionek przy użyciu tych samych czcionek

	A
1	Cambria
2	Calibri
3	Agency FB
4	Albertus MT
5	Albertus MT Lt
6	ALGERIAN
7	Antique Olive Compact
8	Antique Olive Roman
9	Apple Chancery
10	Arial
11	Arial Black
12	Arial Narrow
13	Arial Rounded MT Bold
14	Arial Unicode MS
15	AvantGarde
16	Baskerville Old Face
17	Bauhaus 93
18	Bell MT
19	Berlin Sans FB
20	Berlin Sans FB Demi
21	Bernard MT Condensed
22	Blackadder ITC
23	Bodoni
24	Bodoni MT
25	Bodoni MT Black
26	Bodoni MT Condensed
27	Bodoni MT Poster Compressed
28	Bodoni Poster
29	Bodoni PosterCompressed
30	Book Antiqua
31	Bookman
32	Bookman Old Style

- *Sortowanie bąbelkowe* jest prostą metodą sortowania (zastosowano ją też w przykładzie demonstrującym sortowanie arkusza w rozdziale 9.). Co prawda metoda sortowania bąbelkowego jest łatwa w kodowaniu, ale ma raczej powolny algorytm, zwłaszcza gdy przetwarzaniu podlega duża liczba elementów.
- *Sortowanie metodą quick-sort* (sortowanie szybkie) w porównaniu z bąbelkowym jest o wiele szybszą metodą sortowania, ale też znacznie trudniejszą do zrozumienia. Metoda może zostać wykorzystana tylko w przypadku takich typów danych, jak `Integer` lub `Long`.
- *Sortowanie zliczające* jest bardzo szybkie, ale również trudne do zrozumienia. Technika ta, podobnie jak sortowanie szybkie, działa tylko w przypadku takich typów danych, jak `Integer` lub `Long`.

Na dołączonym dysku CD-ROM znajduje się skoroszyt o nazwie *Sortowanie.xlsm*, który porównuje wyżej wymienione metody sortowania. Skoroszyt przydaje się w przypadku porównywania metod sortowania tablic o różnych rozmiarach. Oczywiście w razie potrzeby możesz skopiować z niego odpowiednie procedury i użyć ich w swoich programach.

Na rysunku 11.16 pokazano okno dialogowe naszego programu. Procedury sortujące zostały przetestowane przy użyciu tablic o siedmiu różnych rozmiarach liczących od 100 do 100 000 elementów. W tablicach zawarte były wartości losowe typu `Long`.



Rysunek 11.16. *Porównanie czasu potrzebnego do wykonania operacji sortowania tablic o różnych rozmiarach*

W tabeli 11.1 zawarłem wyniki testów. Wartość `0,00` oznacza, że sortowanie zostało zakończone prawie natychmiast w czasie krótszym niż 0,01 sekundy.

Tabela 11.1. *Czas trwania (wyrażony w sekundach) operacji sortowania tablic wypełnionych losowymi wartościami przy użyciu czterech algorytmów sortujących*

Liczba elementów tablicy	Sortowanie arkuszowe Excela	Sortowanie bąbelkowe przy użyciu języka VBA	Sortowanie szybkie przy użyciu języka VBA	Sortowanie zliczające przy użyciu języka VBA
100	0,04	0,00	0,00	0,02
500	0,02	0,01	0,00	0,01
1000	0,03	0,03	0,00	0,00
5000	0,07	0,84	0,01	0,01
10 000	0,09	3,41	0,01	0,01
50 000	0,43	79,95	0,07	0,02
100 000	0,78	301,90	0,14	0,04

Algorytm sortowania arkuszowego jest wyjątkowo szybki, zwłaszcza że operacja uwzględnia przeniesienie zawartości tablicy do arkusza, sortowanie jej i ponowne wczytanie danych do tablicy. Jeżeli tablica jest już prawie posortowana, sortowanie arkuszowe będzie jeszcze szybsze.

Algorytm sortowania bąbelkowego jest dość szybki w przypadku niewielkich tablic, ale przy większych (liczących ponad 5000 elementów) powinieneś po prostu o nim zapomnieć. Sortowanie szybkie oraz sortowanie zliczające są bardzo szybkie, ale ich funkcjonalność jest ograniczona jedynie do danych typu `Interger` oraz `Long`.

Przetwarzanie grupy plików

Jednym z częstych zastosowań makr jest oczywiście kilkakrotne powtarzanie określonej operacji. Przykład przedstawiony poniżej ilustruje, jak przy użyciu makra przetworzyć kilka różnych plików zapisanych na dysku. Procedura, która może pomóc w napisaniu

własnego makra realizującego tego typu zadanie, prosi użytkownika o podanie wzorca nazw plików, a następnie przetwarza wszystkie pliki, których nazwy są z nim zgodne. W tym przypadku operacja przetwarzania polega na zaimportowaniu pliku i wprowadzeniu grupy formuł sumujących, które opisują zawarte w nim dane.

```
Sub BatchProcess()
    Dim FileSpec As String
    Dim i As Integer
    Dim FileName As String
    Dim FileList() As String
    Dim FoundFiles As Integer
'   Określenie ścieżki i wzorca nazwy
    FileSpec = ThisWorkbook.Path & "\" & "text??.txt"
    FileName = Dir(FileSpec)

'   Czy plik został znaleziony?
    If FileName <> "" Then
        FoundFiles = 1
        ReDim Preserve FileList(1 To FoundFiles)
        FileList(FoundFiles) = FileName
    Else
        MsgBox "Nie znaleziono żadnych plików pasujących do wzorca " & FileSpec
        Exit Sub
    End If

'   Pobierz nazwy pozostałych plików
    Do
        FileName = Dir
        If FileName = "" Then Exit Do
        FoundFiles = FoundFiles + 1
        ReDim Preserve FileList(1 To FoundFiles)
        FileList(FoundFiles) = FileName & "*"
    Loop
'   Przetwarzanie kolejnych plików w pętli
    For i = 1 To FoundFiles
        Call ProcessFiles(FileList(i))
    Next i
End Sub
```

Skoroszyt z tym przykładem (*Przetwarzanie wsadowe.xlsm*) znajdziesz na płycie CD-ROM dołączonej do książki. Przykład korzysta z trzech dodatkowych plików, również znajdujących się na dysku CD-ROM. Są to: *text01.txt*, *text02.txt* i *text03.txt*. Aby zaimportować inne pliki tekstowe, będziesz musiał odpowiednio zmodyfikować kod procedury.

Nazwy plików pasujące do wzorca są przechowywane w tablicy o nazwie `FoundFiles`. Pliki są przetwarzane przy użyciu pętli `For ... Next`. W trakcie przetwarzania wewnątrz pętli jest wywoływana prosta procedura `ProcessFiles`. W celu zaimportowania pliku korzysta ona z metody `OpenText`, a następnie wstawia pięć formuł. Oczywiście zamiast poniższej można zastosować własną procedurę.

```
Sub ProcessFiles(FileName As String)
'   Importowanie pliku
    Workbooks.OpenText FileName:=FileName, _
        Origin:=xlWindows, _
        StartRow:=1, _
        DataType:=xlFixedWidth, _
```

```
        FieldInfo:= _
        Array(Array(0, 1), Array(3, 1), Array(12, 1))
'   Wprowadzanie formuł podsumowujących
    Range("D1").Value = "A"
    Range("D2").Value = "B"
    Range("D3").Value = "C"
    Range("E1:E3").Formula = "=COUNTIF(B:B,D1)"
    Range("F1:F3").Formula = "=SUMIF(B:B,D1,C:C)"
End Sub
```

Więcej szczegółowych informacji na temat pracy z plikami z poziomu języka VBA znajdziesz w rozdziale 27.

Ciekawe funkcje, których możesz użyć w swoich projektach

W tym podrozdziale zaprezentujemy kilka niestandardowych funkcji, które można albo bezpośrednio stosować w aplikacjach użytkowych, albo modyfikować, traktując je jako twórczą inspirację. Najprzydatniejsze są wtedy, gdy wywołuje się je z innej procedury języka VBA. Procedury zostały zadeklarowane przy użyciu słowa kluczowego `Private`, dzięki czemu nie będą widoczne w oknie dialogowym *Wstawianie funkcji* Excela.

Skoroszyt z tym przykładem (*Funkcje użytkowe VBA.xlsm*) znajdziesz na płycie CD-ROM dołączonej do książki.

Funkcja FileExists

Funkcja pobiera jeden argument (ścieżka pliku wraz z jego nazwą) i zwraca wartość `True`, jeżeli plik istnieje:

```
Private Function FileExists(fname) As Boolean
'   Zwraca wartość True, jeżeli istnieje plik
    FileExists = (Dir(fname) <> "")
End Function
```

Funkcja FileNameOnly

Funkcja pobiera jeden argument (ścieżka pliku wraz z jego nazwą) i zwraca tylko nazwę pliku (innymi słowy — usuwa ścieżkę pliku):

```
Private Function FileNameOnly(pname) As String
' Zwraca nazwę pliku pobraną z łańcucha złożonego ze ścieżki i nazwy pliku
   Dim temp As Variant
   length = Len(pname)
   temp = Split(pname, Application.PathSeparator)
   FileNameOnly = temp(UBound(temp))
End Function
```

Funkcja `FileNameOnly` wykorzystuje funkcję `Split` VBA, która pobiera łańcuch tekstu (oraz separator) i zwraca tabelę typu `Variant`, zawierającą elementy łańcucha znajdujące się pomiędzy znakami separatora. W tym przypadku zmienna `temp` zawiera tablicę z łańcuchami tekstu znajdującymi się pomiędzy separatorami definiowanymi przez `Application.PathSeparator` (zaznaczaj są to znaki lewego ukośnika). Inny przykład zastosowania funkcji `Split` znajdziesz w podrozdziale „Wyznaczanie n-tego elementu łańcucha" w dalszej części tego rozdziału.

Jeżeli argumentem wywołania funkcji jest ścieżka *c:\excel files\2010\backup\budget.xlsm*, funkcja zwróci łańcuch *budget.xlsm*.

Funkcja `FileNameOnly` przetwarza dowolną ścieżkę i nazwę pliku (nawet jeżeli plik nie istnieje). Jeżeli plik istnieje, poniższa funkcja oferuje prostszą metodę usuwania ścieżki i zwracania tylko nazwy pliku:

```
Private Function FileNameOnly2(pname) As String
    FileNameOnly2 = Dir(pname)
End Function
```

Funkcja PathExists

Funkcja pobiera jeden argument (ścieżka pliku) i zwraca wartość `True`, jeżeli ścieżka istnieje:

```
Private Function PathExists(pname) As Boolean
'   Zwraca wartość True, jeżeli istnieje ścieżka
    If Dir(pname, vbDirectory) = "" Then
        PathExists = False
    Else
        PathExists = (GetAttr(pname) And vbDirectory) = vbDirectory
    End If
End Function
```

Funkcja RangeNameExists

Funkcja pobiera jeden argument (nazwa zakresu) i zwraca wartość `True`, jeżeli w aktywnym skoroszycie istnieje nazwa zakresu:

```
Private Function RangeNameExists(nname) As Boolean
'   Zwraca wartość True, jeżeli istnieje nazwa zakresu
    Dim n As Name
    RangeNameExists = False
    For Each n In ActiveWorkbook.Names
        If UCase(n.Name) = UCase(nname) Then
            RangeNameExists = True
            Exit Function
        End If
    Next n
End Function
```

Inny sposób napisania takiej funkcji przedstawiono poniżej. Funkcja w tej wersji próbuje utworzyć zmienną obiektową przy użyciu nazwy zakresu. Jeżeli taka próba zakończy się wygenerowaniem błędu, oznacza to, że dana nazwa zakresu nie istnieje.

```
Private Function RangeNameExists2(nname) As Boolean
' Zwraca wartość True, jeżeli istnieje nazwa zakresu
    Dim n As Range
    On Error Resume Next
    Set n = Range(nname)
    If Err.Number = 0 Then RangeNameExists2 = True _
        Else RangeNameExists2 = False
End Function
```

Funkcja SheetExists

Funkcja pobiera jeden argument (nazwa arkusza) i zwraca wartość `True`, jeżeli w aktywnym skoroszycie istnieje arkusz o takiej nazwie:

```
Private Function SheetExists(sname) As Boolean
'   Zwraca wartość True, jeżeli w aktywnym skoroszycie istnieje arkusz o takiej nazwie
    Dim x As Object
    On Error Resume Next
    Set x = ActiveWorkbook.Sheets(sname)
    If Err = 0 Then SheetExists = True _
        Else SheetExists = False
End Function
```

Sprawdzanie, czy obiekt należy do kolekcji

Poniższa procedura `Function` jest prostą funkcją sprawdzającą, czy dany obiekt należy do kolekcji:

```
Private Function IsInCollection(Coln As Object, _
  Item As String) As Boolean
    Dim Obj As Object
    On Error Resume Next
    Set Obj = Coln(Item)
    IsInCollection = Not Obj Is Nothing
End Function
```

Funkcja pobiera dwa argumenty — kolekcję (obiekt) i element (łańcuch), który może, ale nie musi należeć do kolekcji. Funkcja próbuje utworzyć zmienną obiektową reprezentującą element kolekcji. Jeżeli próba się powiedzie, funkcja zwraca wartość `True`. W przeciwnym razie funkcja zwraca wartość `False`.

Funkcji `IsInCollection` można użyć zamiast trzech innych funkcji wymienionych w rozdziale (`RangeNameExists`, `SheetExists` i `WorkbookIsOpen`). Aby na przykład stwierdzić, czy w aktywnym skoroszycie istnieje zakres o nazwie `Data`, należy wywołać funkcję `IsInCollection` przy użyciu poniższej instrukcji:

```
MsgBox IsInCollection(ActiveWorkbook.Names, "Data")
```

Aby stwierdzić, czy otwarto skoroszyt o nazwie *budżet.xlsx*, należy użyć następującej instrukcji:

```
MsgBox IsInCollection(Workbooks, "budżet.xlsx")
```

Aby stwierdzić, czy aktywny skoroszyt zawiera arkusz o nazwie `Arkusz1`, należy użyć następującej instrukcji:

```
MsgBox IsInCollection(ActiveWorkbook.Worksheets, "Arkusz1")
```

Funkcja WorkbookIsOpen

Funkcja pobiera jeden argument (nazwa skoroszytu) i zwraca wartość `True`, jeżeli skoroszyt jest otwarty:

```
Private Function WorkbookIsOpen(wbname) As Boolean
'   Zwraca wartość True, jeżeli skoroszyt jest otwarty
    Dim x As Workbook
    On Error Resume Next
    Set x = Workbooks(wbname)
    If Err = 0 Then WorkbookIsOpen = True _
        Else WorkbookIsOpen = False
End Function
```

Pobieranie wartości z zamkniętego skoroszytu

Język VBA nie posiada metody umożliwiającej pobranie wartości z zamkniętego skoroszytu. W razie potrzeby możemy jednak skorzystać z faktu, że Excel obsługuje łącza do plików. Zamieszczona poniżej funkcja `GetValue` języka VBA pobiera wartość z zamkniętego skoroszytu. Zadanie to jest realizowane poprzez wywołanie starszego typu *makra XLM*, które było stosowane w wersjach Excela sprzed wersji 5. Na szczęście, jak widać, Excel nadal obsługuje makra tego starego typu.

```
Private Function GetValue(path, file, sheet, ref)
'   Pobiera wartość z zamkniętego skoroszytu
    Dim arg As String
'   Sprawdza, czy istnieje plik
    If Right(path, 1) <> "\" Then path = path & "\"
    If Dir(path & file) = "" Then
        GetValue = "Plik nie został znaleziony."
        Exit Function
    End If
'   Tworzenie argumentu
    arg = "'" & path & "[" & file & "]" & sheet & "'!" & _
        Range(ref).Range("A1").Address(, , xlR1C1)
'   Wykonanie makra XLM
    GetValue = ExecuteExcel4Macro(arg)
End Function
```

Funkcja `GetValue` pobiera cztery argumenty:

- `path` — ścieżka zamkniętego pliku (np. `"d:\pliki"`);
- `file` — nazwa pliku skoroszytu (np. `"budżet.xlsx"`);
- `sheet` — nazwa arkusza (np. `"Arkusz1"`);
- `ref` — odwołanie do komórki (np. `"C4"`).

Poniższa procedura `Sub` demonstruje, w jaki sposób użyć funkcji `GetValue`. Procedura wyświetla po prostu wartość w komórce `A1` arkusza `Arkusz1` pliku o nazwie `2010Budżet.xlsx` znajdującego się w katalogu `XLPliki\Budżet` na dysku `C`.

```
Sub TestGetValue()
    Dim p As String, f As String
    Dim s As String, a As String
```

```
    p = "c:\XLPliki\Budżet"
    f = "2010Budżet.xlsx"
    s = "Arkusz1"
    a = "A1"
    MsgBox GetValue(p, f, s, a)
End Sub
```

Kolejna procedura odczytuje z zamkniętego pliku 1200 wartości (zajmujących obszar 100 wierszy×12 kolumn), a następnie umieszcza je w aktywnym arkuszu:

```
Sub TestGetValue2()
    Dim p As String, f As String
    Dim s As String, a As String
    Dim r As Long, c As Long
    p = "c:\XLPliki\Budżet"
    f = "2010Budżet.xlsx"
    s = "Arkusz1"
    Application.ScreenUpdating = False
    For r = 1 To 100
        For c = 1 To 12
            a = Cells(r, c).Address
            Cells(r, c) = GetValue(p, f, s, a)
        Next c
    Next r
End Sub
```

Funkcja `GetValue` nie zadziała po zastosowaniu jej w formule arkuszowej. W praktyce nie ma jednak żadnej potrzeby umieszczania tej funkcji w jakiejkolwiek formule. W celu pobrania wartości z zamkniętego pliku wystarczy stworzyć łącze do komórki znajdującej się w takim pliku.

Skoroszyt z tym przykładem (*Pobieranie wartości z zamkniętego pliku.xlsm*) znajdziesz na płycie CD-ROM dołączonej do książki. Procedury zawarte w tym skoroszycie pobierają dane ze skoroszytu o nazwie *Mój skoroszyt.xlsx*.

Użyteczne, niestandardowe funkcje arkuszowe

W tym podrozdziale zamieściliśmy przykłady niestandardowych funkcji, które można zastosować w formułach arkusza. Pamiętaj, że takie funkcje (procedury typu `Function`), muszą zostać zdefiniowane w module VBA, a nie w modułach powiązanych z obiektami takimi jak *ThisWorkbook*, *Arkusz1* lub *UserForm1*.

Skoroszyt z przykładami omawianymi w tym podrozdziale (*Funkcje arkuszowe.xlsm*) znajdziesz na płycie CD-ROM dołączonej do książki.

Funkcje zwracające informacje o formatowaniu komórki

W tym podrozdziale znajdziesz szereg niestandardowych funkcji zwracających różnorodne informacje o sposobie formatowania komórki. Funkcje takie są przydatne na przykład w sytuacji, kiedy trzeba posortować dane w oparciu o formatowanie (np. gdy szukasz wszystkich komórek, wobec których użyto pogrubienia).

Takie funkcje nie zawsze są automatycznie przeliczane, ponieważ zmiana formatowania nie uaktywnia mechanizmu Excela wykonującego ponowne obliczenia. Aby wymusić ponowne obliczenie wszystkich skoroszytów wraz z aktualizacją wartości funkcji niestandardowych, należy nacisnąć kombinację klawiszy *Ctrl+Alt+F9*.

Innym rozwiązaniem jest umieszczenie w kodzie źródłowym funkcji następującego polecenia:

```
Application.Volatile
```

Po zastosowaniu tego polecenia naciśnięcie klawisza *F9* spowoduje ponowne przeliczenie arkusza wraz z funkcjami niestandardowymi.

Poniższa funkcja zwraca wartość `True`, jeżeli w jednokomórkowym zakresie będącym jej argumentem zastosowano pogrubienie. Jeżeli jako argument wywołania funkcji zostanie przekazany zakres, funkcja użyje lewej, górnej komórki tego zakresu:

```
Function IsBold(cell) As Boolean
'   Zwraca wartość True, jeżeli zawartość komórki została pogrubiona
    IsBold = cell.Range("A1").Font.Bold
End Function
```

Pamiętaj, że takie funkcje działają poprawnie tylko z komórkami, którym formatowanie zostało nadane bezpośrednio za pomocą poleceń i przycisków formatujących i nie będą działały w przypadku formatowania nadanego za pomocą mechanizmu formatowania warunkowego. W Excelu 2010 wprowadzony został nowy obiekt, `DisplayFormat`, który potrafi obsługiwać elementy formatowania warunkowego. Poniżej zamieszczamy nową wersję funkcji `IsBold` — zwraca ona poprawne wyniki również dla komórek, których zawartość została pogrubiona za pośrednictwem formatowania warunkowego:

```
Function IsBold(cell) As Boolean
'   Zwraca wartość True, jeżeli zawartość komórki została pogrubiona (z uwzględnieniem formatowania
    warunkowego)
    IsBold = cell.Range("A1").DisplayFormat.Font.Bold
End Function
```

Poniższa funkcja zwraca wartość `True`, jeżeli w komórce będącej jej argumentem zastosowano kursywę:

```
Function IsItalic(cell) As Boolean
'   Zwraca wartość True, jeżeli w komórce użyto kursywy
    IsItalic = cell.Range("A1").Font.Italic
End Function
```

Jeżeli w komórce zostanie zastosowane mieszane formatowanie (np. tylko wybrane znaki zostaną pogrubione), obie powyższe funkcje zwrócą błąd. Kolejna funkcja zwróci wartość `True` tylko wtedy, gdy wszystkie znaki z komórki będą pogrubione:

```
Function AllBold(cell) As Boolean
'   Zwraca wartość True, jeżeli wszystkie znaki z komórki są pogrubione
    If IsNull(cell.Font.Bold) Then
        AllBold = False
    Else
        AllBold = cell.Font.Bold
    End If
End Function
```

Funkcja `AllBold` może zostać uproszczona do następującej postaci:

```
Function AllBold (cell) As Boolean
' Zwraca wartość True, jeżeli wszystkie znaki z komórki są pogrubione
    AllBold = Not IsNull(cell.Font.Bold)
End Function
```

Funkcja `FillColor` zwraca liczbę całkowitą odpowiadającą indeksowi koloru tła komórki (czyli inaczej mówiąc, indeksowi koloru jej wypełnienia). Kolor tła komórki zwykle zależy od wybranego stylu formatowania arkusza. Jeżeli tło komórki nie zostanie wypełnione, funkcja zwraca wartość `-4142`.

Ta funkcja nie działa poprawnie kolorami tła tabel (definiowanych za pomocą polecenia *Tabela*, znajdującego się na karcie *Wstawianie*, w grupie poleceń *Tabele*). Aby uwzględnić kolory tła tabel, powinieneś użyć obiektu `DisplayFormat`, o którym wspominaliśmy wcześniej.

```
Function FillColor(cell) As Integer
'   Zwraca liczbę całkowitą odpowiadającą kolorowi tła komórki
    FillColor = cell.Range("A1").Interior.ColorIndex
End Function
```

Gadający arkusz?

Funkcja `SayIt` wykorzystuje wbudowany w Excela generator mowy to odczytywania „na głos" przekazanego jej argumentu (którym może być łańcuch tekstu lub odwołanie do wybranej komórki).

```
Function SayIt(txt)
    Application.Speech.Speak (txt)
    SayIt = txt
End Function
```

Opisywana funkcja daje całkiem interesujące efekty i może być naprawdę użyteczna. Spróbuj użyć tej funkcji na przykład w następującej formule:

```
=IF(SUM(A:A)>25000,SayIt("Cel został osiągnięty!"))
```

Jeżeli suma wartości w kolumnie `A` przekroczy wartość `25000`, usłyszysz zsyntetyzowany głos radośnie oznajmiający, że cel został osiągnięty. Metody `Speak` możesz również użyć do powiadomienia o zakończeniu długo działającej procedury. W ten sposób po uruchomieniu procedury będziesz mógł zająć się czymś innym, a Excel sam powiadomi Cię, kiedy procedura wreszcie zakończy działanie.

Wyświetlanie daty zapisania lub wydrukowania pliku

Skoroszyt Excela posiada szereg wbudowanych właściwości dokumentu, które są dostępne dla programów VBA za pośrednictwem właściwości `BuiltinDocumentProperties` obiektu `Workbook`. Poniższa funkcja zwraca datę i czas wykonania ostatniej operacji zapisu skoroszytu:

```
Function LastSaved()
    Application.Volatile
```

```
    LastSaved = ThisWorkbook. _
       BuiltinDocumentProperties("Last Save Time")
End Function
```

Data i czas ostatniego zapisu, zwracane przez tę funkcję, to dokładnie ta sama informacja, jaką znajdziesz w sekcji *Powiązane daty* widoku *Backstage*, dostępnego po przejściu na kartę *Plik* i wybraniu polecenia *Informacje*. Pamiętaj, że mechanizm automatycznego zapisu również modyfikuje te dane, stąd znacznik czasu ostatniego zapisu dokumentu niekoniecznie musi być tożsamy z tym, kiedy dany dokument został zapisany *przez użytkownika*.

Poniższa funkcja jest podobna do poprzedniej, z tym że zwraca datę i czas wykonania ostatniej operacji drukowania lub podglądu wydruku dla skoroszytu. Jeżeli skoroszyt nie był jeszcze nigdy drukowany ani użytkownik nigdy nie korzystał z opcji podglądu wydruku, funkcja zwraca błąd `#ARG!`:

```
Function LastPrinted()
    Application.Volatile
    LastPrinted = ThisWorkbook. _
      BuiltinDocumentProperties("Last Print Date")
End Function
```

Jeżeli użyjesz tych funkcji w formule, to w celu uzyskania aktualnych wartości właściwości skoroszytu może być konieczne wymuszenie wykonania ponownych obliczeń (poprzez wciśnięcie klawisza *F9*).

Istnieje całkiem sporo dodatkowych wbudowanych właściwości, ale Excel nie korzysta ze wszystkich. Na przykład próba użycia właściwości `Number of Bytes` spowoduje wygenerowanie błędu. Pełną listę wbudowanych właściwości skoroszytów znajdziesz w pomocy systemowej programu Excel.

Funkcje `LastSaved` i `LastPrinted` zostały zaprojektowane z myślą o przechowywaniu ich w skoroszycie, w którym są używane. W niektórych przypadkach funkcja może zostać umieszczona w innym skoroszycie (np. *personal.xlsb*) lub dodatku. Ponieważ obie powyższe funkcje odwołują się do właściwości `ThisWorkbook`, po zapisaniu ich w innym skoroszycie nie będą działały poprawnie. Poniżej zawarto wersje tych funkcji o bardziej uniwersalnym przeznaczeniu. Funkcje używają właściwości `Application.Caller`, która zwraca obiekt klasy `Range` reprezentujący komórkę wywołującą funkcję. Właściwość `Parent.Parent` zwraca skoroszyt (obiekt `Workbook`), czyli obiekt nadrzędny przodka obiektu `Range`. Zagadnienie to zostanie omówione dokładniej w następnym podrozdziale.

```
Function LastSaved2()
    Application.Volatile
    LastSaved2 = Application.Caller.Parent.Parent. _
        BuiltinDocumentProperties("Last Save Time")
End Function
```

Obiekty nadrzędne

Jak pamiętasz, model obiektowy Excela ma postać hierarchiczną — poszczególne obiekty są zawarte w innych obiektach. Na szczycie hierarchii znajduje się obiekt klasy `Appli`
`↪cation`. Excel zawiera inne obiekty, które są kontenerami dla kolejnych obiektów, itd. Poniższa hierarchia ilustruje miejsce, jakie w tym schemacie zajmuje obiekt `Range`:

obiekt Application
  obiekt Workbook
    obiekt Worksheet
      obiekt Range

W języku programowania obiektowego mówimy, że przodkiem (obiektem nadrzędnym) obiektu Range jest obiekt Worksheet, będący jego kontenerem. Przodkiem obiektu Work
↪sheet jest obiekt Workbook przechowujący arkusz. Z kolei obiektem nadrzędnym obiektu Workbook jest obiekt Application.

W jaki sposób wykorzystać tę informację w praktyce? Przeanalizujmy poniższą funkcję SheetName języka VBA. Funkcja pobiera jeden argument (zakres) i zwraca nazwę arkusza zawierającego zakres. Funkcja używa właściwości Parent obiektu Range. Właściwość Parent zwraca obiekt przechowujący obiekt Range.

```
Function SheetName(ref) As String
    SheetName = ref.Parent.Name
End Function
```

Kolejna funkcja WorkbookName zwraca nazwę skoroszytu zawierającego określoną komórkę. Zauważ, że funkcja dwukrotnie używa właściwości Parent. Pierwsza właściwość Parent zwraca obiekt Worksheet, a druga — obiekt Workbook.

```
Function WorkbookName(ref) As String
    WorkbookName = ref.Parent.Parent.Name
End Function
```

Poniższa funkcja AppName przenosi nas na kolejny poziom w hierarchii, trzykrotnie korzystając z właściwości Parent. Funkcja zwraca nazwę obiektu Application powiązanego z określoną komórką. Oczywiście w naszym przypadku funkcja zawsze będzie zwracała wartość Microsoft Excel.

```
Function AppName(ref) As String
    AppName = ref.Parent.Parent.Parent.Name
End Function
```

Zliczanie komórek, których wartości zawierają się pomiędzy dwoma wartościami

Poniższa funkcja o nazwie CountBetween zwraca liczbę wartości w zakresie (pierwszy argument), mieszczących się pomiędzy dwoma wartościami reprezentowanymi przez drugi i trzeci argument:

```
Function CountBetween(InRange, num1, num2) As Long
'   Zlicza wartości z przedziału od num1 do num2
    With Application.WorksheetFunction
        If num1 <= num2 Then
           CountBetween = .CountIfs(InRange, ">=" & num1, _
                InRange, "<=" & num2)
        Else
           CountBetween = .CountIfs(InRange, ">=" & num2, _
                InRange, "<=" & num1)
```

```
        End If
    End With
End Function
```

Funkcja korzysta z funkcji arkuszowej COUNTIFS (LICZ.WARUNKI) Excela i w praktyce spełnia rolę funkcji osłonowej upraszczającej tworzone formuły.

Funkcja LICZ.WARUNKI została wprowadzona w Excelu 2007, stąd taka funkcja nie będzie działała z wcześniejszymi wersjami Excela.

Poniżej zamieszczono przykład formuły korzystającej z funkcji CountBetween, zwracającej liczbę komórek zakresu A1:A100, których wartości są większe lub równe 10 i mniejsze lub równe 20:

```
=CountBetween(A1:A100, 10, 20)
```

Funkcja pobiera dwa argumenty numeryczne w dowolnej kolejności, stąd formuła przedstawiona poniżej działa dokładnie tak samo, jak jej poprzedniczka:

```
=CountBetween(A1:A100, 20, 10)
```

Zastosowanie tej funkcji języka VBA jest zdecydowanie prostsze niż wprowadzenie następującej długiej (i jak widać — dosyć złożonej) formuły:

```
=LICZ.WARUNKI(A1:A100,">=10",A1:A100"<=20")
```

Wyznaczanie ostatniej niepustej komórki kolumny lub wiersza

W tym podrozdziale zaprezentowano dwie bardzo przydatne funkcje. Pierwsza z nich, o nazwie LastInColumn, zwraca zawartość ostatniej niepustej komórki danej kolumny. Druga funkcja, LastInRow, zwraca zawartość ostatniej niepustej komórki danego wiersza. Każda z funkcji pobiera pojedynczy argument, którym jest zakres. Zakresem może być cała kolumna (funkcja LastInColumn) lub cały wiersz (funkcja LastInRow). Jeżeli przekazany argument nie jest całą kolumną lub wierszem, funkcja użyje wiersza lub kolumny określonej przez górną lewą komórkę zakresu. Poniższa przykładowa formuła zwraca wartość ostatniej, niepustej komórki kolumny B:

```
=LastInColumn(B5)
```

Kolejna formuła zwraca wartość ostatniej, niepustej komórki z wiersza 7:

```
=LastInRow(C7:D9)
```

Oto kod źródłowy funkcji LastInColumn:

```
Function LastInColumn(rng As Range)
'   Zwraca zawartość ostatniej niepustej komórki kolumny
    Dim LastCell As Range
    Application.Volatile
    With rng.Parent
        With .Cells(.Rows.Count, rng.Column)
            If Not IsEmpty(.Value) Then
                LastInColumn = .Value
            ElseIf IsEmpty(.End(xlUp)) Then
```

```
                LastInColumn = ""
            Else
                LastInColumn = .End(xlUp).Value
            End If
        End With
    End With
End Function
```

Funkcja jest dosyć złożona, dlatego poniżej przedstawiamy kilka punktów, które mogą pomóc Ci zrozumieć jej sposób działania:

- Metoda `Application.Volatile` powoduje, że funkcja zostanie wykonana każdorazowo przy obliczaniu arkusza.
- Właściwość `Rows.Count` zwraca liczbę wierszy arkusza. Zamiast na sztywno wprowadzać w kodzie źródłowym liczbę wierszy arkusza, użyłem właściwości `Count`, ponieważ kolejna wersja Excela może obsługiwać jeszcze większą niż dotychczas liczbę wierszy.
- Właściwość `rng.Column` zwraca numer kolumny górnej lewej komórki zakresu będącego wartością argumentu `rng`.
- Zastosowanie właściwości `rng.Parent` powoduje, że funkcja będzie działać poprawnie nawet wtedy, gdy argument `rng` odwołuje się do innego arkusza lub skoroszytu.
- Użycie metody `End` z argumentem `xlUp` jest równoznaczne z uaktywnieniem ostatniej komórki kolumny, wciśnięciem klawisza *End*, a następnie klawisza ↑.
- Funkcja `IsEmpty` sprawdza, czy komórka jest pusta. Jeżeli tak jest, zwraca pusty łańcuch. Gdyby funkcja `IsEmpty` nie została zastosowana, po napotkaniu pustej komórki nasza funkcja zwróciłaby wartość `0`.

Poniżej przedstawiamy kod źródłowy funkcji `LastInRow`, która jest bardzo podobna do funkcji `LastInColumn`:

```
Function LastInRow(rng As Range)
'   Zwraca zawartość ostatniej niepustej komórki wiersza
    Application.Volatile
    With rng.Parent
        With .Cells(rng.Row, .Columns.Count)
            If Not IsEmpty(.Value) Then
                LastInRow = .Value
            ElseIf IsEmpty(.End(xlToLeft)) Then
                LastInRow = ""
            Else
                LastInRow = .End(xlToLeft).Value
            End If
        End With
    End With
End Function
```

Czy dany łańcuch tekstu jest zgodny z wzorcem?

Funkcja `IsLike` jest bardzo prosta i jednocześnie bardzo użyteczna. Zwraca wartość `True`, jeżeli łańcuch tekstowy jest zgodny ze zdefiniowanym wzorcem.

Jak widać poniżej, kod funkcji jest bardzo prosty. Funkcja właściwie odgrywa rolę funkcji osłonowej, pozwalającej na wygodne użycie w tworzonych formułach wszechstronnego operatora `Like` języka VBA:

```
Function IsLike(text As String, pattern As String) As Boolean
'    Zwraca wartość True, jeżeli pierwszy argument jest podobny do drugiego
    IsLike = text Like pattern
End Function
```

Funkcja `IsLike` pobiera dwa argumenty:

- `text` — łańcuch tekstowy lub odwołanie do komórki, która go zawiera;
- `pattern` — łańcuch zawierający znaki wieloznaczne, które wymieniono w poniższej tabeli.

Znaki zawarte we wzorcu	Zawartość łańcucha text zgodna ze wzorcem
?	Dowolny pojedynczy znak
*	0 lub więcej dowolnych znaków
#	Dowolna pojedyncza cyfra (0 – 9)
[*lista_znaków*]	Dowolny pojedynczy znak znajdujący się na *liście_znaków*
[*!lista_znaków*]	Dowolny pojedynczy znak, który nie znajduje się na *liście_znaków*

Poniższa formuła zwraca wartość `TRUE`, ponieważ wzorzec * jest zgodny z dowolną liczbą znaków. Formuła zwraca wartość `TRUE`, jeżeli pierwszy argument jest dowolnym łańcuchem tekstowym rozpoczynającym się od litery `g`:

```
=IsLike("gitara", "g*")
```

Kolejna formuła zwraca wartość `TRUE`, ponieważ wzorzec ? jest zgodny z dowolnym pojedynczym znakiem. Formuła zwróci wartość `FALSE`, jeżeli wartością pierwszego argumentu będzie łańcuch `Jednostka12`:

```
= IsLike("Jednostka1", "Jednostka?")
```

Następna formuła zwraca wartość `TRUE`, ponieważ jej pierwszy argument jest pojedynczym znakiem zawartym w drugim argumencie:

```
= IsLike("a", "[aeiou]")
```

Poniższa formuła zwraca wartość `TRUE`, jeżeli komórka `A1` zawiera literę *a*, *e*, *i*, *o*, *u*, *A*, *E*, *I*, *O* lub *U*. Przetworzenie argumentów przy użyciu funkcji `UPPER` spowoduje, że formuła nie będzie rozróżniała wielkości znaków:

```
= IsLike(UPPER(A1), UPPER("[aeiou]"))
```

Poniższa formuła zwraca wartość `TRUE`, jeżeli komórka `A1` zawiera wartość rozpoczynającą się cyfrą `1` i składającą się dokładnie z trzech cyfr (czyli dowolną liczbę całkowitą z przedziału od 100 do 199):

```
= IsLike(A1, "1##")
```

Wyznaczanie n-tego elementu łańcucha

Funkcja `ExtractElement` jest niestandardową funkcją arkusza (może być też wywoływana z procedury języka VBA) wyznaczającą n-ty element łańcucha tekstowego. Jeżeli na przykład komórka zawiera poniższy tekst, w celu wydzielenia dowolnego podłańcucha zawartego pomiędzy łącznikami można użyć funkcji `ExtractElement`:

```
123-456-789-0133-8844
```

Kolejna formuła zwraca podłańcuch `0133` będący czwartym elementem łańcucha (w roli separatora w łańcuchu jest używany łącznik):

```
=ExtractElement("123-456-789-0133-8844", 4, "-")
```

Funkcja `ExtractElement` pobiera trzy argumenty:

- `Txt` — łańcuch tekstowy, z którego są wydzielane podłańcuchy (może to być literał lub odwołanie do komórki);
- `n` — liczba całkowita reprezentująca wydzielany element;
- `Separator` — pojedynczy znak spełniający funkcję separatora.

Jeżeli wartością argumentu będzie spacja, ciągi kilku spacji zostaną potraktowanych jak jedna spacja, co prawie zawsze będzie zgodne z Twoimi zamierzeniami. Jeżeli wartość argumentu `n` przekroczy liczbę elementów łańcucha, funkcja zwróci pusty łańcuch.

Oto kod źródłowy funkcji `ExtractElement` języka VBA:

```
Function ExtractElement(Txt, n, Separator) As String
'    Zwraca n-ty element łańcucha tekstowego, w którym poszczególne elementy oddziela określony znak
     separatora
     Dim AllElements As Variant
     AllElements = Split(Txt, Separator)
     ExtractElement = AllElements(n - 1)
End Function
```

Funkcja korzysta z funkcji `Split` języka VBA zwracającej tablicę typu `Variant`, która zawiera poszczególne elementy łańcucha tekstowego. Indeks tablicy rozpoczyna się od wartości `0`, a nie `1`, dlatego odwołania do kolejnych elementów tablicy są realizowane poprzez wyrażenie `n-1`.

Zamiana wartości na słowa[1]

Funkcja `SPELLDOLLARS` zwraca „słowną" wersję wartości numerycznych podanych jako argument wywołania funkcji (tak jak w dobrze każdemu znanej formułce *SŁOWNIE:* spotykanej na blankietach przelewów i wpłat czy fakturach). Na przykład formuła przedstawiona poniżej zwraca następujący łańcuch tekstu: *One hundred twenty-three and 45/100 dollars* (sto dwadzieścia trzy dolary i 45 centów):

```
=SPELLDOLLARS(123.45)
```

[1] Uwaga: funkcja opisana w tym podrozdziale zwraca wartości w języku angielskim — *przyp. tłum.*

Na rysunku 11.17 przedstawiono kilka przykładów zastosowania funkcji SPELLDOLLARS. Formuły zostały umieszczone w kolumnie C. Przykładowo formuła umieszczona w komórce C1 ma następującą postać:

```
=SPELLDOLLARS(A1)
```

Zwróć uwagę na fakt, że wartości ujemne są podawane w nawiasach.

	A	B	C	D
1	32		Thirty-Two and 00/100 Dollars	
2	37,56		Thirty-Seven and 56/100 Dollars	
3	-32		(Thirty-Two and 00/100 Dollars)	
4	-26,44		(Twenty-Six and 44/100 Dollars)	
5	-4		(Four and 00/100 Dollars)	
6	1,87341		One and 87/100 Dollars	
7	1,56		One and 56/100 Dollars	
8	1		One and 00/100 Dollars	
9	6,56		Six and 56/100 Dollars	
10	12,12		Twelve and 12/100 Dollars	
11	1000000		One Million and 00/100 Dollars	
12	10000000000		Ten Billion and 00/100 Dollars	
13	1111111111		One Billion One Hundred Eleven Million One Hundred Eleven Thousand One Hundred Eleven and 00/100 Dollars	
14				

Rysunek 11.17. *Przykłady zastosowania funkcji SPELLDOLLARS*

Funkcja SPELLDOLLARS jest zbyt złożona, aby ją tutaj zaprezentować w całości, ale skoroszyt zawierający pełny kod tej funkcji (*Funkcje arkuszowe.xlsm*) znajdziesz na płycie CD-ROM dołączonej do książki.

Funkcja wielofunkcyjna

Następny przykład prezentuje technikę, która może okazać się przydatna w niektórych sytuacjach. Sprawia ona, że pojedyncza funkcja arkusza zachowuje się jak wiele funkcji. Poniżej zamieszczamy kod źródłowy niestandardowej funkcji o nazwie StatFunction, która pobiera dwa argumenty — zakres (rng) i operację (op). W zależności od wartości argumentu op funkcja zwraca wartość obliczoną przy użyciu dowolnej z następujących funkcji arkuszowych: AVERAGE (ŚREDNIA), COUNT (ILE.LICZB), MAX, MEDIAN (MEDIANA), MIN, MODE (WYST.NAJCZĘŚCIEJ), STDEV (ODCH.STANDARDOWE), SUM (SUMA) lub VAR (WARIANCJA).

Funkcji StatFunction możesz użyć w arkuszu w następujący sposób:

```
=StatFunction(B1:B24, A24)
```

Wynik formuły zależy od zawartości komórki A24, która powinna być takim łańcuchem, jak Average, Count, Max itd. Podobną technikę kodowania możesz zastosować w przypadku innych funkcji.

```
Function StatFunction(rng, op)
    Select Case UCase(op)
        Case "SUM"
            StatFunction = WorksheetFunction.Sum(rng)
        Case "AVERAGE"
            StatFunction = WorksheetFunction.Average(rng)
        Case "MEDIAN"
            StatFunction = WorksheetFunction.Median(rng)
```

```
        Case "MODE"
            StatFunction = WorksheetFunction.Mode(rng)
        Case "COUNT"
            StatFunction = WorksheetFunction.Count(rng)
        Case "MAX"
            StatFunction = WorksheetFunction.Max(rng)
        Case "MIN"
            StatFunction = WorksheetFunction.Min(rng)
        Case "VAR"
            StatFunction = WorksheetFunction.Var(rng)
        Case "STDEV"
            StatFunction = WorksheetFunction.StDev(rng)
        Case Else
            StatFunction = CVErr(xlErrNA)
    End Select
End Function
```

Funkcja SheetOffset

Excel oferuje ograniczoną obsługę trójwymiarowych skoroszytów. Jeżeli na przykład konieczne jest odwołanie do innego arkusza skoroszytu, w formule trzeba uwzględnić nazwę arkusza. Nie stanowi to jednak dużego problemu… do momentu próby skopiowania formuły do innych arkuszy. Skopiowane formuły w dalszym ciągu odwołują się do nazwy oryginalnego arkusza, a odwołania do arkuszy nie są modyfikowane tak, jak miałoby to miejsce w prawdziwym trójwymiarowym arkuszu.

W tym podrozdziale został omówiony przykład funkcji języka VBA o nazwie `SheetOffset`, umożliwiającej stosowanie względnych odwołań do arkuszy. Na przykład w celu odwołania się do komórki `A1` poprzedniego arkusza należy użyć formuły:

```
=SheetOffset(-1, A1)
```

Pierwszy argument funkcji, który może być wartością dodatnią, ujemną lub zerem, identyfikuje względne odwołanie do arkusza. Drugi argument musi być odwołaniem do pojedynczej komórki. Po skopiowaniu formuły do innych arkuszy odwołanie względne będzie obowiązywało we wszystkich jej kopiach.

Oto kod źródłowy funkcji `SheetOffset` języka VBA:

```
Function SheetOffset(Offset As Long, Optional Cell As Variant)
'   Zwraca zawartość komórki względnie adresowanego arkusza, do której zdefiniowano odwołanie
    Dim WksIndex As Long, WksNum As Long
    Dim wks As Worksheet
    Application.Volatile
    If IsMissing(Cell) Then Set Cell = Application.Caller
    WksNum = 1
    For Each wks In Application.Caller.Parent.Parent.Worksheets
        If Application.Caller.Parent.Name = wks.Name Then
            SheetOffset = Worksheets(WksNum + Offset).Range(Cell(1).Address)
            Exit Function
        Else
            WksNum = WksNum + 1
        End If
    Next wks
End Function
```

Zwracanie maksymalnej wartości ze wszystkich arkuszy

Aby określić maksymalną wartość komórki B1 z kilku arkuszy, można zastosować formułę podobną do poniższej:

```
=MAX(Arkusz1:Arkusz4!B1)
```

Formuła zwraca maksymalną wartość komórki B1 z arkuszy Arkusz1, Arkusz4 i wszystkich, które znajdują się pomiędzy nimi.

Co się jednak stanie, gdy za arkuszem Arkusz4 zostanie wstawiony nowy arkusz Arkusz5? Formuła nie uwzględni tego automatycznie, dlatego konieczne będzie jej zmodyfikowanie w celu dodania nowego odwołania do arkusza:

```
=MAX(Arkusz1:Arkusz5!B1)
```

Funkcja MaxAllSheets pobiera jeden argument i zwraca maksymalną wartość określonej komórki z wszystkich arkuszy skoroszytu. Przykładowo poniższa formuła zwraca maksymalną wartość komórki B1 z uwzględnieniem wszystkich arkuszy skoroszytu:

```
=MaxAllSheets(B1)
```

Po dodaniu nowego arkusza nie będzie już potrzeby edytowania formuły.

```
Function MaxAllSheets(cell)
    Dim MaxVal As Double
    Dim Addr As String
    Dim Wksht As Object
    Application.Volatile
    Addr = cell.Range("A1").Address
    MaxVal = -9.9E+307
    For Each Wksht In cell.Parent.Parent.Worksheets
        If Wksht.Name = cell.Parent.Name And _
          Addr = Application.Caller.Address Then
        ' Uniknięcie odwołania cyklicznego
        Else
            If WorksheetFunction.IsNumber(Wksht.Range(Addr)) Then
                If Wksht.Range(Addr) > MaxVal Then _
                    MaxVal = Wksht.Range(Addr).Value
            End If
        End If
    Next Wksht
    If MaxVal = -9.9E+307 Then MaxVal = 0
    MaxAllSheets = MaxVal
End Function
```

W celu uzyskania dostępu do skoroszytu pętla For Each używa następującego wyrażenia:

```
cell.Parent.Parent.Worksheets
```

Obiektem nadrzędnym komórki jest arkusz, natomiast przodkiem arkusza jest skoroszyt. Wynika z tego, że pętla For Each ... Next przetwarza wszystkie arkusze skoroszytu. Pierwsza instrukcja If z pętli sprawdza, czy przetwarzana komórka zawiera funkcję. Jeżeli tak jest, to w celu uniknięcia błędu odwołania cyklicznego komórka zostanie zignorowana.

Opisana funkcja z łatwością może zostać zmodyfikowana tak, aby wykonywała inne obliczenia międzyarkuszowe oparte na takich funkcjach, jak MIN, ŚREDNIA, SUMA itd.

Zwracanie tablicy zawierającej unikatowe, losowo uporządkowane liczby całkowite

Funkcja RandomIntegers zamieszczona w tym punkcie zwraca tablicę unikatowych liczb całkowitych. Stosowana jest w wielokomórkowych formułach tablicowych.

```
{=RandomIntegers()}
```

Zaznacz zakres, a następnie wprowadź formułę (bez nawiasów klamrowych) i zatwierdź ją poprzez naciśnięcie kombinacji klawiszy *Ctrl+Shift+Enter*. Formuła zwraca tablicę zawierającą unikatowe, losowo uporządkowane liczby całkowite. Jeżeli na przykład formuła zostanie wprowadzona do zakresu złożonego z 50 komórek, jej kopie zwrócą unikatowe liczby całkowite z przedziału od 1 do 50.

Oto kod źródłowy funkcji RandomIntegers:

```
Function RandomIntegers()
    Dim FuncRange As Range
    Dim V() As Variant, ValArray() As Variant
    Dim CellCount As Double
    Dim i As Integer, j As Integer
    Dim r As Integer, c As Integer
    Dim Temp1 As Variant, Temp2 As Variant
    Dim RCount As Integer, CCount As Integer

'   Tworzy obiekt klasy Range
    Set FuncRange = Application.Caller
'   Zwraca błąd, jeżeli wartość obiektu FuncRange jest zbyt duża
    CellCount = FuncRange.Count
    If CellCount > 1000 Then
        RandomIntegers = CVErr(xlErrNA)
        Exit Function
    End If

'   Przypisanie zmiennych
    RCount = FuncRange.Rows.Count
    CCount = FuncRange.Columns.Count
    ReDim V(1 To RCount, 1 To CCount)
    ReDim ValArray(1 To 2, 1 To CellCount)
'   Wypełnienie tablicy losowymi wartościami i liczbami całkowitymi zakresu rng
    For i = 1 To CellCount
        ValArray(1, i) = Rnd
        ValArray(2, i) = i
    Next i
'   Sortowanie tablicy ValArray według wymiaru o losowej wartości
    For i = 1 To CellCount
        For j = i + 1 To CellCount
            If ValArray(1, i) > ValArray(1, j) Then
                Temp1 = ValArray(1, j)
                Temp2 = ValArray(2, j)
                ValArray(1, j) = ValArray(1, i)
```

```
                ValArray(2, j) = ValArray(2, i)
                ValArray(1, i) = Temp1
                ValArray(2, i) = Temp2
            End If
        Next j
    Next i

'   Wstawienie losowo uporządkowanych wartości do tablicy V
    i = 0
    For r = 1 To RCount
        For c = 1 To CCount
            i = i + 1
            V(r, c) = ValArray(2, i)
        Next c
    Next r
    RandomIntegers = V
End Function
```

Porządkowanie zakresu w losowy sposób

Funkcja `RangeRandomize` pobiera jeden argument będący zakresem i zwraca tablicę złożoną z losowo uporządkowanych wartości tego zakresu.

```
Function RangeRandomize(rng)
    Dim V() As Variant, ValArray() As Variant
    Dim CellCount As Double
    Dim i As Integer, j As Integer
    Dim r As Integer, c As Integer
    Dim Temp1 As Variant, Temp2 As Variant
    Dim RCount As Integer, CCount As Integer

'   Zwraca błąd, jeżeli wartość obiektu rng jest zbyt duża
    CellCount = rng.Count
    If CellCount > 1000 Then
        RangeRandomize = CVErr(xlErrNA)
        Exit Function
    End If

'   Przypisanie zmiennych
    RCount = rng.Rows.Count
    CCount = rng.Columns.Count
    ReDim V(1 To RCount, 1 To CCount)
    ReDim ValArray(1 To 2, 1 To CellCount)
'   Wypełnienie tablicy ValArray losowymi wartościami i wartościami obiektu rng
    For i = 1 To CellCount
        ValArray(1, i) = Rnd
        ValArray(2, i) = rng(i)
    Next i
'   Sortowanie tablicy ValArray według wymiaru o losowej wartości
    For i = 1 To CellCount
        For j = i + 1 To CellCount
            If ValArray(1, i) > ValArray(1, j) Then
                Temp1 = ValArray(1, j)
                Temp2 = ValArray(2, j)
                ValArray(1, j) = ValArray(1, i)
                ValArray(2, j) = ValArray(2, i)
```

```
                ValArray(1, i) = Temp1
                ValArray(2, i) = Temp2
            End If
        Next j
    Next i

'   Wstawienie losowo uporządkowanych wartości do tablicy V
    i = 0
    For r = 1 To RCount
        For c = 1 To CCount
            i = i + 1
            V(r, c) = ValArray(2, i)
        Next c
    Next r
    RangeRandomize = V
End Function
```

Jak łatwo zauważyć, kod źródłowy tej funkcji jest bardzo podobny do kodu funkcji `Random ↪Integers`.

Na rysunku 11.18 pokazano wynik działania funkcji. Formuła tablicowa zawarta w zakresie `B2:B11` ma następującą postać:

```
{=RangeRandomize(A2:A11)}
```

Formuła zwraca zawartość komórek zakresu `A2:A11`, ale uporządkowanego w losowy sposób.

Rysunek 11.18. *Funkcja RangeRandomize zwraca zawartość komórek zakresu w przypadkowej kolejności*

	A	B	C
1	Oryginał	Losowo	
2	Antylopa	Gekon	
3	Bażant	czapla	
4	Czapla	Iguana	
5	Drozd	Emu	
6	Emu	Antylopa	
7	Fretka	Drozd	
8	Gekon	Hipopotam	
9	Hipopotam	Bażant	
10	Iguana	Jastrząb	
11	Jastrząb	Fretka	
12			

Wywołania funkcji interfejsu Windows API

Jedną z najważniejszych cech języka VBA jest możliwość wywoływania funkcji przechowywanych w bibliotekach DLL (ang. *Dynamic Link Library*). W przykładach zaprezentowanych w tym podrozdziale będziemy korzystali z często używanych funkcji interfejsu API systemu Windows.

Dla uproszczenia deklaracje funkcji API przedstawiane w tym podrozdziale działają tylko w Excelu 2010 (zarówno w wersji 32-bitowej, jak i 64-bitowej), natomiast przykłady zamieszczone na dysku CD-ROM dołączonym do książki zawierają odpowiednie dyrektywy kompilatora, dzięki czemu będą poprawnie działać również we wcześniejszych wersjach Excela.

Określanie skojarzeń plików

W systemie Windows wiele typów plików jest kojarzonych z określoną aplikacją. Po wykonaniu takiego skojarzenia plik można otworzyć w powiązanej z nim aplikacji poprzez dwukrotne kliknięcie pliku.

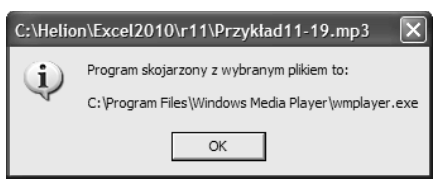
Funkcja `GetExecutable` w celu uzyskania pełnej ścieżki aplikacji skojarzonej z określonym plikiem korzysta z funkcji interfejsu API systemu Windows. Przyjmijmy, że w Twoim systemie znajduje się wiele plików o rozszerzeniu *.txt*. Jeden z nich, o nazwie *Readme.txt*, prawdopodobnie znajduje się w katalogu systemu Windows. Aby określić pełną ścieżkę aplikacji otwierającej plik po jego dwukrotnym kliknięciu, można użyć funkcji `GetExecutable`.

Deklaracje funkcji interfejsu API systemu Windows muszą zostać umieszczone na początku modułu kodu VBA.

```
Private Declare Function FindExecutableA Lib "shell32.dll" _
    (ByVal lpFile As String, ByVal lpDirectory As String, _
    ByVal lpResult As String) As Long

Function GetExecutable(strFile As String) As String
    Dim strPath As String
    Dim intLen As Integer
    strPath = Space(255)
    intLen = FindExecutableA(strFile, "\", strPath)
    GetExecutable = Trim(strPath)
End Function
```

Na rysunku 11.19 pokazano wynik wywołania funkcji `GetExecutable`, która jako argument pobrała nazwę pliku muzycznego w formacie MP3. Funkcja zwraca pełną ścieżkę aplikacji powiązanej z plikiem.

Rysunek 11.19. *Określanie ścieżki aplikacji powiązanej z określonym plikiem*

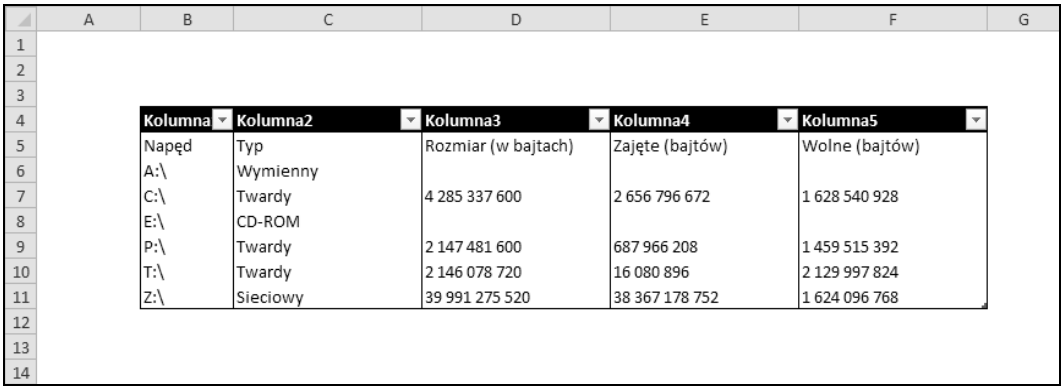


Skoroszyt z tym przykładem (*Skojarzenia plików.xlsm*) znajdziesz na płycie CD-ROM dołączonej do książki.

Pobieranie informacji o napędach dyskowych

VBA nie posiada metody pozwalającej na bezpośrednie pobieranie informacji o zainstalowanych w systemie napędach dyskowych. Jednak dzięki zastosowaniu trzech funkcji API możesz uzyskać niezbędne informacje.

Na rysunku 11.20 przedstawiono wynik działania procedury VBA, która identyfikuje wszystkie podłączone do systemu dyski, określa ich typ, sprawdza całkowitą pojemność, rozmiar użytego miejsca oraz rozmiar wolnego miejsca.



Kolumna	Kolumna2	Kolumna3	Kolumna4	Kolumna5
Napęd	Typ	Rozmiar (w bajtach)	Zajęte (bajtów)	Wolne (bajtów)
A:\	Wymienny			
C:\	Twardy	4 285 337 600	2 656 796 672	1 628 540 928
E:\	CD-ROM			
P:\	Twardy	2 147 481 600	687 966 208	1 459 515 392
T:\	Twardy	2 146 078 720	16 080 896	2 129 997 824
Z:\	Sieciowy	39 991 275 520	38 367 178 752	1 624 096 768

Rysunek 11.20. *Zastosowanie funkcji Windows API do pobierania informacji o dyskach*

Kod procedury jest dosyć długi i złożony, dlatego nie umieszczono go tutaj, ale jeżeli jesteś ciekawy, jak to działa, możesz zajrzeć do odpowiedniego skoroszytu na płycie CD-ROM dołączonej do książki.

Skoroszyt z tym przykładem (*Informacja o dyskach.xlsm*) znajdziesz na płycie CD-ROM dołączonej do książki.

Pobieranie informacji dotyczących drukarki domyślnej

W kolejnym przykładzie użyjemy funkcji interfejsu Windows API zwracającej informacje na temat domyślnej drukarki. Dane znajdują się w pojedynczym łańcuchu tekstowym. Poniższa procedura analizuje łańcuch i wyświetla informacje przy użyciu bardziej czytelnego dla użytkownika formatu:

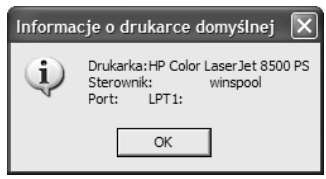
```
Private Declare Function GetProfileStringA Lib "kernel32" _
    (ByVal lpAppName As String, ByVal lpKeyName As String, _
    ByVal lpDefault As String,  ByVal lpReturnedString As String, _
    ByVal nSize As Long) As Long

Sub DefaultPrinterInfo()
    Dim strLPT As String * 255
    Dim Result As String
    Call GetProfileStringA _
       ("Windows", "Device", "", strLPT, 254)

    Result = Application.Trim(strLPT)
    ResultLength = Len(Result)
    Comma1 = InStr(1, Result, ",", 1)
    Comma2 = InStr(Comma1 + 1, Result, ",", 1)
'   Pobiera nazwę drukarki
    Printer = Left(Result, Comma1 - 1)
'   Pobiera informacje na temat sterownika
    Driver = Mid(Result, Comma1 + 1, Comma2 - Comma1 - 1)
'   Pobiera ostatnią część informacji na temat urządzenia
    Port = Right(Result, ResultLength - Comma2)
'   Tworzy komunikat
    Msg = "Drukarka:" & Chr(9) & Printer & Chr(13)
```

```
    Msg = Msg & "Sterownik:" & Chr(9) & Driver & Chr(13)
    Msg = Msg & "Port:" & Chr(9) & Port
'   Wyświetla komunikat
    MsgBox Msg, vbInformation, "Informacje o drukarce domyślnej"
End Sub
```

Co prawda właściwość `ActivePrinter` obiektu `Application` zwraca nazwę domyślnej drukarki i umożliwia jej zmianę, ale nie istnieje bezpośrednia metoda określenia, jaki sterownik i port urządzenia jest używany. Z tego właśnie powodu czasami przydatna może być nasza funkcja `GetProfileStringA`.

Na rysunku 11.21 pokazano przykładowe okno komunikatu wyświetlone przez tę procedurę.

Rysunek 11.21. *Informacja o drukarce domyślnej wyświetlona przy użyciu funkcji interfejsu API systemu Windows*



Skoroszyt z tym przykładem (*Informacja o drukarce.xlsm*) znajdziesz na płycie CD-ROM dołączonej do książki.

Pobieranie informacji o aktualnej rozdzielczości karty graficznej

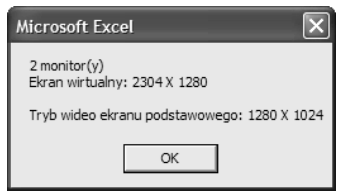
Zamieszczony w tym punkcie kod korzysta z funkcji interfejsu API w celu określenia aktualnej rozdzielczości karty graficznej używanej w systemie. Jeżeli używana aplikacja musi wyświetlić określoną ilość informacji na jednym ekranie, znajomość jego rozdzielczości może pomóc we właściwym przeskalowaniu tekstu. Oprócz tego kod procedury sprawdza liczbę monitorów podłączonych do komputera. Jeżeli podłączonych jest więcej monitorów niż jeden, procedura wyświetla rozmiary pulpitu wirtualnego.

```
Declare PtrSafe Function GetSystemMetrics Lib "user32" _
   (ByVal nIndex As Long) As Long
Public Const SM_CMONITORS = 80
Public Const SM_CXSCREEN = 0
Public Const SM_CYSCREEN = 1
Public Const SM_CXVIRTUALSCREEN = 78
Public Const SM_CYVIRTUALSCREEN = 79

Sub DisplayVideoInfo()
    Dim numMonitors As Long
    Dim vidWidth As Long, vidHeight As Long
    Dim virtWidth As Long, virtHeight As Long
    Dim Msg As String

    numMonitors = GetSystemMetrics(SM_CMONITORS)
    vidWidth = GetSystemMetrics(SM_CXSCREEN)
    vidHeight = GetSystemMetrics(SM_CYSCREEN)
    virtWidth = GetSystemMetrics(SM_CXVIRTUALSCREEN)
    virtHeight = GetSystemMetrics(SM_CYVIRTUALSCREEN)

    If numMonitors > 1 Then
        Msg = numMonitors & " monitory podłączone" & vbCrLf
```

```
        Msg = Msg & "Pulpit wirtualny: " & virtWidth & " × "
        Msg = Msg & virtHeight & vbCrLf & vbCrLf
        Msg = Msg & "Rozdzielczość głównego monitora to: "
        Msg = Msg & vidWidth & " × " & vidHeight
    Else
        Msg = Msg & "Aktualny tryb graficzny: "
        Msg = Msg & vidWidth & " × " & vidHeight
    End If
    MsgBox Msg
End Sub
```

Na rysunku 11.22 pokazano okno komunikatu zwrócone przez powyższą procedurę uruchomioną w systemie używającym dwóch monitorów.

Rysunek 11.22. *Zastosowanie funkcji interfejsu Windows API do określenia rozdzielczości karty graficznej*



Skoroszyt z tym przykładem (*Informacja o rozdzielczości karty graficznej.xlsm*) znajdziesz na płycie CD-ROM dołączonej do książki.

Dodanie dźwięku do aplikacji

Przykład, który omówiono w tym podrozdziale, pozwala na dodanie efektów dźwiękowych do aplikacji Excela, a w szczególności pozwala na odtwarzanie dźwięków zapisanych w formacie MIDI lub WAV. Możesz na przykład odtwarzać wybrany dźwięk w momencie otwierania danego okna dialogowego (lub nie…). Jeżeli będziesz chciał, aby Excel odtwarzał pliki w formacie MIDI lub WAV, w tej sekcji znajdziesz dokładnie to, czego będziesz potrzebował.

Skoroszyt z tym przykładem (*Dźwięk.xlsm*) znajdziesz na płycie CD-ROM dołączonej do książki.

Odtwarzanie plików typu WAV

Poniższy przykład zawiera deklarację funkcji interfejsu API wraz z prostą procedurą odtwarzającą plik dźwiękowy o nazwie `sound.wav`, który znajduje się w tym samym katalogu, co plik skoroszytu:

```
Private Declare Function PlaySound Lib "winmm.dll" _
    Alias "PlaySoundA" (ByVal lpszName As String, _
    ByVal hModule As Long, ByVal dwFlags As Long) As Long

Const SND_SYNC = &H0
Const SND_ASYNC = &H1
Const SND_FILENAME = &H20000

Sub PlayWAV()
```

```
    WAVFile = "sound.wav"
    WAVFile = ThisWorkbook.Path & "\" & WAVFile
    Call PlaySound(WAVFile, 0&, SND_ASYNC Or SND_FILENAME)
End Sub
```

W przykładzie plik formatu WAV jest odtwarzany asynchronicznie. Oznacza to, że w trakcie odtwarzania kontynuowane jest wykonywanie procedury. Aby zatrzymać wykonywanie kodu źródłowego podczas odtwarzania dźwięku, należy użyć następującej instrukcji:

```
Call PlaySound(WAVFile, 0&, SND_SYNC Or SND_FILENAME)
```

Odtwarzanie pliku formatu MIDI

W przypadku pliku formatu MIDI konieczne jest zastosowanie innej funkcji interfejsu API. Procedura `PlayMIDI` rozpoczyna odtwarzanie pliku formatu MIDI. Wykonanie procedury `StopMIDI` spowoduje zakończenie odtwarzania pliku. W przykładzie użyto pliku o nazwie `helion.mid`.

```
Private Declare Function mciExecute Lib "winmm.dll" _
    (ByVal lpstrCommand As String) As Long

Sub PlayMIDI()
    MIDIFile = " helion.mid"
    MIDIFile = ThisWorkbook.Path & "\" & MIDIFile
    MciExecute ("play " & MIDIFile)
End Sub

Sub StopMIDI()
    MIDIFile = "helion.mid"
    MIDIFile = ThisWorkbook.Path & "\" & MIDIFile
    MciExecute ("stop " & MIDIFile)
End Sub
```

Odtwarzanie dźwięku przy użyciu funkcji arkuszowej

Funkcja `Alarm` została stworzona z myślą o zastosowaniu w formule arkusza. Jeżeli komórka spełnia określone kryterium, funkcja, używając interfejsu API systemu Windows, odtwarza plik dźwiękowy.

```
Declare Function PlaySound Lib "winmm.dll" _
    Alias "PlaySoundA" (ByVal lpszName As String, _
    ByVal hModule As Long, ByVal dwFlags As Long) As Long

Function Alarm(Cell, Condition)
    Dim WAVFile As String
    Const SND_ASYNC = &H1
    Const SND_FILENAME = &H20000
    If Evaluate(Cell.Value & Condition) Then
        WAVFile = ThisWorkbook.Path & "\sound.wav"
        Call PlaySound(WAVFile, 0&, SND_ASYNC Or SND_FILENAME)
        Alarm = True
    Else
        Alarm = False
    End If
End Function
```

Funkcja `Alarm` pobiera dwa argumenty — odwołanie do komórki i warunek (mający postać łańcucha). Poniższa formuła używa funkcji `Alarm` do odtworzenia pliku formatu WAV, gdy wartość komórki `B13` będzie większa lub równa `1000`:

```
=ALARM(B13; ">=1000")
```

W celu stwierdzenia, czy wartość komórki spełnia określone kryterium, funkcja korzysta z funkcji `Evaluate` języka VBA. Po spełnieniu kryterium i wygenerowaniu dźwięku funkcja zwróci wartość `True`. W przeciwnym razie zwróci wartość `False`.

Funkcja `SayIt`, omawiana wcześniej w tym rozdziale, jest znacznie prostszym sposobem na wzbogacenie aplikacji w efekty dźwiękowe.

Skoroszyt z tymi przykładami (*dźwięk.xlsm*) znajdziesz na płycie CD-ROM dołączonej do książki.

Odczytywanie zawartości rejestru systemu Windows i zapisywanie w nim danych

Większość aplikacji Windows potrzebne informacje przechowuje w rejestrze systemu będącym bazą danych. Aby uzyskać dodatkowe informacje o rejestrze, należy zajrzeć do rozdziału 4. Procedury języka VBA są w stanie odczytywać dane z rejestru i zapisywać w nim nowe wartości. Aby to było możliwe, konieczne jest zastosowanie następujących deklaracji funkcji interfejsu API systemu Windows:

```
Private Declare PtrSafe Function RegOpenKeyA Lib "ADVAPI32.DLL" _
    (ByVal hKey As Long, ByVal sSubKey As String, _
    ByRef hkeyResult As Long) As Long
Private Declare PtrSafe Function RegCloseKey Lib "ADVAPI32.DLL" _
    (ByVal hKey As Long) As Long
Private Declare PtrSafe Function RegSetValueExA Lib "ADVAPI32.DLL" _
    (ByVal hKey As Long, ByVal sValueName As String, _
    ByVal dwReserved As Long, ByVal dwType As Long, _
    ByVal sValue As String, ByVal dwSize As Long) As Long

Private Declare PtrSafe Function RegCreateKeyA Lib "ADVAPI32.DLL" _
    (ByVal hKey As Long, ByVal sSubKey As String, _
    ByRef hkeyResult As Long) As Long
Private Declare PtrSafe Function RegQueryValueExA Lib "ADVAPI32.DLL" _
    (ByVal hKey As Long, ByVal sValueName As String, _
    ByVal dwReserved As Long, ByRef lValueType As Long, _
    ByVal sValue As String, ByRef lResultLen As Long) As Long
```

Utworzyłem dwie funkcje osłonowe ułatwiające korzystanie z rejestru. Są to: `GetRegistry` i `WriteRegistry`. Obie znajdują się na dołączonym dysku CD-ROM, w skoroszycie o nazwie *Rejestr systemu Windows.xlsm*. Przykładowy skoroszyt zawiera procedurę demonstrującą odczyt i zapis danych w rejestrze.

Odczyt danych z rejestru

Funkcja `GetRegistry` zwraca ustawienia znajdujące się w określonej lokalizacji rejestru. Funkcja pobiera trzy argumenty:

- `RootKey` — łańcuch reprezentujący główny klucz rejestru, który zostanie użyty. Oto możliwe łańcuchy:
  - `HKEY_CLASSES_ROOT`
  - `HKEY_CURRENT_USER`
  - `HKEY_LOCAL_MACHINE`
  - `HKEY_USERS`
  - `HKEY_CURRENT_CONFIG`
- `Path` — pełna ścieżka kategorii rejestru, która zostanie użyta.
- `RegEntry` — nazwa ustawienia, które zostanie odczytane.

Aby na przykład odnaleźć w rejestrze aktualne ustawienie powiązane z aktywnym paskiem tytułu okna, należy w sposób pokazany poniżej wywołać funkcję `GetRegistry` (wielkość znaków nazw argumentów nie jest rozróżniana):

```
RootKey = "hkey_current_user"
Path = "Control Panel\Desktop"
RegEntry = "WallPaper"
MsgBox GetRegistry(RootKey, Path, RegEntry), _
   vbInformation, Path & "\RegEntry"
```

Okno komunikatu wyświetli ścieżkę i nazwę pliku graficznego użytego w roli tapety pulpitu (jeżeli tapeta nie jest używana, funkcja zwróci pusty łańcuch).

Zapis danych w rejestrze

Funkcja `WriteRegistry` zapisuje wartość w określonej lokalizacji rejestru. Jeżeli operacja zakończy się powodzeniem, funkcja zwróci wartość `True`. W przeciwnym razie zwróci wartość `False`. Funkcja `WriteRegistry` pobiera następujące argumenty (wszystkie są łańcuchami tekstu):

- `RootKey` — łańcuch reprezentujący klucz rejestru, który zostanie użyty. Oto możliwe łańcuchy:
  - `HKEY_CLASSES_ROOT`
  - `HKEY_CURRENT_USER`
  - `HKEY_LOCAL_MACHINE`
  - `HKEY_USERS`
  - `HKEY_CURRENT_CONFIG`
- `Path` — pełna ścieżka kategorii rejestru (jeżeli ścieżka nie istnieje, zostanie utworzona).
- `RegEntry` — nazwa kategorii rejestru, w której zostanie zapisana wartość (jeżeli kategoria nie istnieje, zostanie dodana).
- `RegVal` — zapisywana wartość.

Poniżej zamieszczono przykład procedury zapisującej w rejestrze wartość reprezentującą datę i czas uruchomienia Excela. Informacja jest zapisywana w miejscu, w którym są przechowywane ustawienia dotyczące Excela.

```
Sub Workbook_Open()
    RootKey = "hkey_current_user"
    Path = "software\microsoft\office\14.0\Excel\LastStarted"
    RegEntry = "DateTime"
    RegVal = Now()
    If WriteRegistry(RootKey, Path, RegEntry, RegVal) Then
        msg = RegVal & " została zapisana w rejestrze."
    Else
        msg = "Wystąpił błąd."
    End If
    MsgBox msg
End Sub
```

Jeżeli zapiszesz tę procedurę w module `ThisWorkbook` skoroszytu makr osobistych, ustawienia będą automatycznie aktualizowane przy każdym uruchomieniu programu Excel.

Łatwiejszy sposób uzyskania dostępu do rejestru

Jeżeli w celu zapisania i odczytania danych dostęp do rejestru systemu Windows chcesz uzyskać z poziomu aplikacji Excela, nie musisz stosować funkcji interfejsu API. Zamiast nich można użyć funkcji `GetSetting` i `SaveSetting` języka VBA.

Obie funkcje zostały objaśnione w systemie pomocy, dlatego nie będę ich tutaj szczegółowo omawiał. Jednak należy wiedzieć, że funkcje te działają tylko z kluczem o następującej nazwie:

```
HKEY_CURRENT_USER\Software\VB and VBA Program Settings
```

Innymi słowy, funkcje nie mogą zostać zastosowane w celu uzyskania dostępu do *dowolnego* klucza rejestru. Funkcje te są najbardziej przydatne do zapisywania informacji o własnych aplikacjach Excela, które chcesz przechować pomiędzy kolejnymi sesjami.

Posiądź wiedzę profesjonalistów — wykorzystaj wszystkie możliwości Excela i poszerzaj je!

Excel® 2010 PL. Programowanie w VBA

Nie należysz do osób, które onieśmiela potencjał Excela? Sprawnie tworzysz skoroszyty, wprowadzasz formuły, używasz funkcji arkuszowych i swobodnie posługujesz się Wstążką programu? Czujesz, że drzemie w nim jeszcze ogrom niezwykłych możliwości, ale nie wiesz, jak po nie sięgnąć? Najwyższa pora na naukę z Johnem Walkenbachem — najsłynniejszym ekspertem w dziedzinie Excela! Jeśli poznałeś już podstawowe funkcje tego programu, dzięki tej książce bez trudu opanujesz narzędzia zaawansowane, czyli takie, które naprawdę ułatwią i przyspieszą Twoją codzienną pracę!

Swoją naukę pod okiem mistrza zaczniesz od odświeżenia informacji na temat używania rozmaitych formuł oraz plików stosowanych i generowanych przez Excel. Zaraz potem przejdziesz do fascynującej części, poświęconej projektowaniu aplikacji w tym programie. Dowiesz się, czym taka aplikacja jest i jak szczegółowo wyglądają etapy jej tworzenia. Następnie opanujesz całą niezbędną wiedzę na temat języka VBA, aby sprawnie w nim programować oraz tworzyć funkcje i procedury. Nauczysz się również wykorzystywać jego możliwości podczas używania tabel przestawnych i wykresów. Ponadto wzbogacisz się o informacje na temat projektowania niestandardowych, przyjaznych okien dialogowych UserForm, automatycznej obsługi zdarzeń czy tworzenia praktycznych dodatków dla Excela.

- Przegląd możliwości Excela 2010
- Projektowanie aplikacji w programie Excel
- Język Visual Basic for Applications
- Zastosowanie formularzy UserForm
- Niestandardowe okna dialogowe
- Zaawansowane metody programowania
- Tabele przestawne, wykresy i obsługa zdarzeń
- Projektowanie dodatków do Excela
- Tworzenie systemów pomocy dla aplikacji
- Tworzenie aplikacji przyjaznych dla użytkownika
- Metody użycia VBA do pracy z plikami

Vademecum Walkenbacha

Sięgnij po mistrzowskie umiejętności! Oto seria podręczników, w których najsłynniejszy ekspert w dziedzinie Excela, John Walkenbach, pokazuje, jak wykrzesać z tego programu maksimum możliwości! Te adresowane do średnio i bardzo zaawansowanych użytkowników książki pozwalają wyjść poza świat standardowych narzędzi i dają praktyczną wiedzę o tym, jak rozszerzać i dopasowywać funkcjonalność Excela do własnych potrzeb! **Fascynują Cię formuły, tworzenie makr, VBA czy projektowanie złożonych, przyjaznych dla użytkownika aplikacji? Nikt nie nauczy Cię więcej o Excelu niż sam mistrz Walkenbach!**

John Walkenbach

jeden z największych autorytetów w dziedzinie Excela. Autor kilkudziesięciu książek i setek artykułów na temat tego programu oraz twórca wielokrotnie nagradzanego pakietu narzędziowego *Power Utility Pak*, przeznaczonego dla Excela. W 2000 roku zdobył prestiżowy tytuł Microsoft MVP, który jest przyznawany corocznie najbardziej aktywnym specjalistom.

Helion

Sprawdź najnowsze promocje:
http://helion.pl/promocje

Książki najchętniej czytane:
http://helion.pl/bestsellery

Zamów informacje o nowościach:
http://helion.pl/nowosci

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

helion.pl
księgarnia internetowa

Nr katalogowy: 5840

Księgarnia internetowa:
http://helion.pl

Zamówienia telefoniczne:
0 801 339900
0 601 339900

Cena xx,00 zł

ISBN 978-83-246-2863-6

9 788324 628636

Informatyka w najlepszym wydaniu