helion.pl
KSIĘGARNIA INTERNETOWA · WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

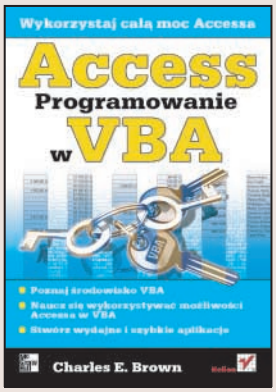
KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

Access. Programowanie w VBA



Autor: Charles E. Brown
Tłumaczenie: Krzysztof Masłowski (wstęp, rozdz. 1 – 8), Grzegorz Werner (rozdz. 9 – 21, dod. A)
ISBN: 83-7361-807-4
Tytuł oryginału: Access VBA Programming
Format: B5, stron: 408

Wykorzystaj całą moc Accessa

- Poznaj środowisko VBA
- Naucz się wykorzystywać możliwości Accessa w VBA
- Stwórz wydajne i szybkie aplikacje

MS Access jest jednym z najczęściej wykorzystywanych systemów zarządzania bazami danych. Jest łatwy w obsłudze, posiada spore możliwości i nie wymaga poznawania złożonych języków manipulacji danymi. Czasem jednak jego podstawowe możliwości nie wystarczają do realizacji niektórych zadań. W takich sytuacjach należy sięgnąć po VBA (Visual Basic for Applications), czyli narzędzie programistyczne pozwalające na tworzenie „aplikacji dla aplikacji" – programów integrujących się z Accessem i wykorzystujących jego funkcje, ale w sposób dokładnie taki, jaki jest potrzebny w określonym przypadku.

Książka „Access. Programowanie w VBA" opisuje zasady programowania w Accessie z wykorzystaniem VBA. Przedstawia zasady projektowania aplikacji i pracy ze środowiskiem programistycznym VBA, korzystania z elementów i funkcji Accessa oraz tworzenia formularzy i raportów. Znajdziesz w niej także bardziej zaawansowane techniki – tworzenie aplikacji klient-serwer, mechanizmy wymiany danych pomiędzy aplikacjami pakietu MS Office oraz łączenie Accessa z SQL Serverem.

- Projektowanie aplikacji
- Dostosowanie Accessa do własnych potrzeb
- Środowisko programistyczne VBA oraz język Visual Basic for Applications
- Tworzenie formularzy, raportów, menu i pasków narzędziowych
- Bezpieczeństwo baz danych
- Aplikacje WWW i zastosowanie języka XML
- Tworzenie aplikacji wielodostępnych

Jeśli nie wystarczają Ci standardowe możliwości Accessa, sięgnij po VBA – w ten sposób wykorzystasz prawdziwe bogactwo Accessa.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

O Autorach **9**

Wstęp **11**

Część I Poznanie środowiska MS Accessa **15**

Rozdział 1. Wstęp do VBA for Applications **17**
Historia Microsoft Accessa 17
Tworzenie aplikacji w Accessie 19
Okno Baza danych 19
Makra 21
Moduły 22
Przyszłość Microsoft Accessa 23
Podsumowanie 24

Rozdział 2. Projektowanie aplikacji Accessa **25**
Etapy procesu tworzenia aplikacji 26
Określanie i analizowanie wymagań 26
Analizowanie wymagań 27
Ocena i oszacowanie wymagań 28
Projektowanie i tworzenie programu 29
Model encja-relacja 30
Tłumaczenie modelu E-R na accessową bazę danych 31
Wdrażanie i testowanie 34
Przygotowanie dokumentacji 35
Utrzymanie systemu 36
Podsumowanie 37

Rozdział 3. Używanie Accessa bez VBA **39**
Architektura systemu 40
Rozumienie aplikacji bazodanowej 41
Praca bez VBA 42
Relacje 43
Zdarzenia 45
Stwórz makro 46
Przekształcanie makr w kod VBA 49
Konwencje nazw 50
Szablony predefiniowane 50
Menadżer panelu przełączania 51
Podsumowanie 55

Rozdział 4. Przystosowywanie środowiska Accessa do własnych potrzeb 57
Interakcja z Accessem 57
Dostosowywanie paska poleceń 59
Tworzenie własnego paska narzędziowego 62
Dostosowywanie menu podręcznych 63
Tworzenie menu 64
Przypisywanie kodu do elementu dostosowanego menu 65
Podsumowanie 68

Część II Poznanie środowiska VBA 69

Rozdział 5. Wprowadzenie do środowiska programistycznego VBA 71
Wprowadzenie do VBA 71
Historia VBA 72
Model VBA 72
Moduły 73
Procedury 73
Koncepcja obiektów 74
Obiekty ADO 76
Obiekty VBA 77
Obiekty Accessa 77
Edytor VBA 78
Zaprzęgnięcie edytora VBA do pracy 80
Podsumowanie 84

Rozdział 6. Podstawy programowania w VBA 85
Podstawy programowania 85
Tworzenie modułów standardowych 86
Tworzenie procedur 88
Wejście i wyjście 94
Struktury sterujące 96
Struktury decyzyjne 96
Pętle 101
Tablice 106
Składniki tablicy 106
Czyszczenie tablic 111
IsArray 111
Podsumowanie 112

Rozdział 7. Zrozumienie działania edytora VBA 113
Otwieranie edytora VBA 113
Najważniejsze elementy i cechy systemu menu 115
Menu Edit 115
Menu View 115
Menu Tools 116
Menu podręczne (Shortcut) 116
Paski narzędziowe Debug, Edit i UserForm 117
Eksplorator projektu 118
Zarządzanie modułami 119
Wstawianie i kasowanie modułów 119
Importowanie i eksportowanie modułów 120
Okno Properties 121
Przeglądarka obiektów 121
Biblioteki 124
Referencje 125

Używanie opcji edytora VBA ... 125
Karta Editor ... 126
Karta Editor Format ... 129
Karty General i Docking ... 130
Sięganie po pomoc ... 131
Rozpoczynanie nowej procedury w VBA ... 133
Podsumowanie ... 134

Rozdział 8. Składniki języka VBA ... 135
Obiekty w VBA ... 135
Stałe wewnętrzne ... 136
Podstawowe programy VBA używające formularzy ... 137
Ustawianie ogniska ... 138
Znajdowanie rekordu ... 141
Obiekt Me ... 143
Sprawdzanie poprawności rekordu ... 144
Łączenie z Accessem ... 145
SQL (strukturalny język zapytań) ... 146
SQL — podstawy ... 147
SQL i VBA ... 150
Prezentacja wyników ... 153
Podsumowanie ... 155

Rozdział 9. Procedury ... 157
Deklarowanie procedury ... 157
Funkcje ... 159
Parametry ... 160
Parametry opcjonalne ... 162
Parametry nazwane ... 163
Projekt i zasięg ... 164
Zmienne globalne i statyczne ... 165
Zmienne globalne ... 165
Zmienne statyczne ... 166
Zmiana struktury bazy danych za pomocą procedury ... 167
Konstruowanie tabeli ... 167
Podsumowanie ... 169

Rozdział 10. Usuwanie usterek z kodu VBA ... 171
Podstawowe informacje o obsłudze błędów ... 171
Błędy wykonania ... 172
Obiekt Err ... 176
Kolekcja Errors ... 177
Narzędzia do usuwania usterek z kodu ... 180
Okno Immediate i punkty wstrzymania ... 181
Asercje ... 183
Okno Locals ... 184
Okno Watch ... 185
Podsumowanie ... 186

Rozdział 11. Funkcje ... 187
Podprogramy i funkcje ... 187
Funkcje wbudowane ... 188
MsgBox ... 188
InputBox ... 192
Funkcje daty i czasu ... 193
Funkcje finansowe ... 195
Podsumowanie ... 197

Część III Interakcja z VBA 199

Rozdział 12. Formularze 201
Formularze i dane 201
Dynamiczne przypisywanie zestawów rekordów 205
Dołączanie i edytowanie danych 207
Sterowanie formantami na formularzach 208
Podsumowanie 214

Rozdział 13. Raporty 215
Anatomia raportu 215
Kreator raportów 218
Raporty specjalne 220
Wykresy 221
Etykiety adresowe 224
Wywoływanie raportu z kodu VBA 226
Tworzenie raportu metodą programową 226
Tworzenie pustego raportu 226
Dodawanie formantów do raportu 228
Podsumowanie 231

Rozdział 14. Menu i paski narzędzi 233
Paski poleceń 233
Tworzenie paska narzędzi 235
Menu 240
Podmenu 243
Podsumowanie 244

Rozdział 15. Modyfikowanie środowiska Accessa 245
Właściwości startowe 245
Modyfikowanie opcji 247
Karta Widok 249
Karta Ogólne 249
Karta Edytowanie/Znajdowanie 250
Karta Klawiatura 250
Karta Arkusz danych 251
Karta Formularze/Raporty 251
Karta Strony 252
Karta Zaawansowane 252
Karta Międzynarodowe 253
Karta Sprawdzanie błędów 253
Karta Pisownia 253
Karta Tabele/Kwerendy 254
Podsumowanie 255

Część IV Zaawansowane techniki programowania 257

Rozdział 16. Bezpieczeństwo bazy danych 259
Access i bezpieczeństwo 259
Interfejs użytkownika 259
Ustawianie hasła 260
Programowe tworzenie hasła 261
Ochrona kodu VBA 262
Kompilowanie kodu do pliku .mde 263
Zabezpieczenia na poziomie użytkownika 263
Modyfikowanie kont użytkowników metodą programową 266

Bezpieczeństwo i Access 2003 ... 268
Jet Sandbox ... 269
Zabezpieczenia makr ... 269
Podsumowanie ... 273

Rozdział 17. Access i środowisko Microsoft Office ... 275
Współdzielone obiekty ... 275
Obiekt FileSearch ... 276
Obiekt CommandBar ... 277
Łączenie Accessa z innymi programami pakietu Microsoft Office ... 280
Łączenie się z Excelem ... 281
Łączenie się z Outlookiem ... 283
Łączenie się z Wordem ... 284
Podsumowanie ... 286

Rozdział 18. Access i sieć WWW ... 287
Mechanika sieci WWW ... 287
Strony dostępu do danych ... 288
Generowanie strony dostępu do danych ... 289
Zapisywanie strony DAP ... 293
Tworzenie strony dostępu do danych w widoku projektu ... 295
Strony dostępu do danych i VBA ... 297
XML (Extensible Markup Language) ... 298
Podsumowanie ... 301

Rozdział 19. Rozbudowa ... 303
Microsoft Access i SQL Server ... 304
Access Data Projects ... 304
Rozbudowa bazy danych Accessa ... 306
Praca z plikiem ADP ... 310
Procedury przechowywane ... 311
Widoki ... 311
Podsumowanie ... 313

Część V Tworzenie aplikacji ... 315

Rozdział 20. Aplikacje wielodostępne ... 317
Współdzielenie bazy danych ... 317
Obsługa współzawodnictwa i konfliktów ... 318
Ustawianie interwału odświeżania ... 319
Ustawianie interwału aktualizacji ... 319
Blokowanie rekordów w interfejsie Accessa ... 320
Ustawianie opcji sposobem programowym ... 322
Blokowanie rekordów w ADO ... 323
Zwiększanie wydajności aplikacji ... 324
Oddzielanie danych od innych obiektów Accessa ... 325
Centralizacja i decentralizacja ... 329
Kompilowanie kodu ... 329
Optymalizowanie kodu VBA ... 331
Podsumowanie ... 333

Rozdział 21. Nie tylko Microsoft Access ... 335
VBA i Microsoft Office ... 335
Visual Basic ... 340
VBScript ... 347
Platforma Microsoft .NET ... 349
Podsumowanie ... 350

Dodatki 351

Dodatek A Obiekt DoCmd 353

AddMenu 354
ApplyFilter 354
Beep 355
CancelEvent 356
Close 356
CopyDatabaseFile 358
CopyObject 358
DeleteObject 359
DoMenuItem 360
Echo 360
FindNext 361
FindRecord 362
GoToControl 364
GoToPage 364
GoToRecord 365
Hourglass 366
Maximize 366
Minimize 367
MoveSize 367
OpenDataAccessPage 368
OpenDiagram 369
OpenForm 369
OpenFunction 371
OpenModule 371
OpenQuery 372
OpenReport 373
OpenStoredProcedure 374
OpenTable 375
OpenView 376
OutputTo 377
PrintOut 379
Quit 380
Rename 380
RepaintObject 382
Requery 382
Restore 382
RunCommand 383
RunMacro 383
RunSQL 384
Save 385
SelectObject 386
SendObject 387
SetMenuItem 388
SetWarnings 389
ShowAllRecords 389
ShowToolbar 390
TransferDatabase 391
TransferSpreadsheet 391
TransferSQLDatabase 391
TransferText 391

Skorowidz 393

Rozdział 6.

Podstawy programowania w VBA

W rozdziale 5 poznałeś środowisko VBA, jego strukturę, koncepcję i edytor. Teraz zamierzamy przedstawić podstawy programowania oraz specyficzną składnię języka VBA. Wrócimy do koncepcji omówionych w rozdziale 5 i przyjrzymy się im dokładniej, zaczynając od przeglądu miejsc, gdzie kod jest przechowywany, oraz typów bloków, na jakie jest dzielony. Czyniąc to, szczegółowo zajmiemy się również problemem zmiennych. Trudno jest mówić o zmiennych, pomijając ich szczególny typ nazywany tablicą (*array*).

Zamierzamy także przyjrzeć się kilku wbudowanym narzędziom VBA, znacznie ułatwiającym wykonywanie zadań. Zakończymy nasze rozważania, przyglądając się różnym typom struktur programistycznych i ich stosowaniu w środowisku VBA.

Podstawy programowania

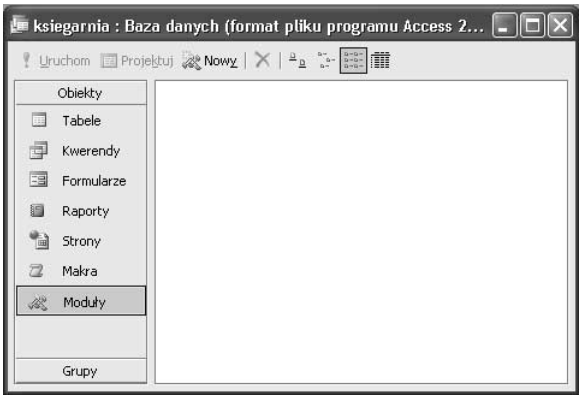
Jak już powiedzieliśmy w rozdziale 5, kod VBA jest zapisywany i przechowywany w modułach. Zapewne pamiętasz, że istnieją dwa podstawowe typy modułów: powiązane z raportami i formularzami oraz ogólne, przechowujące procedury używane w całym projekcie.

Istnieje jeszcze jeden typ zwany *modułem klasy*, który zawiera kod powiązany z obiektem. O modułach będziemy jeszcze mówić w pozostałej części książki.

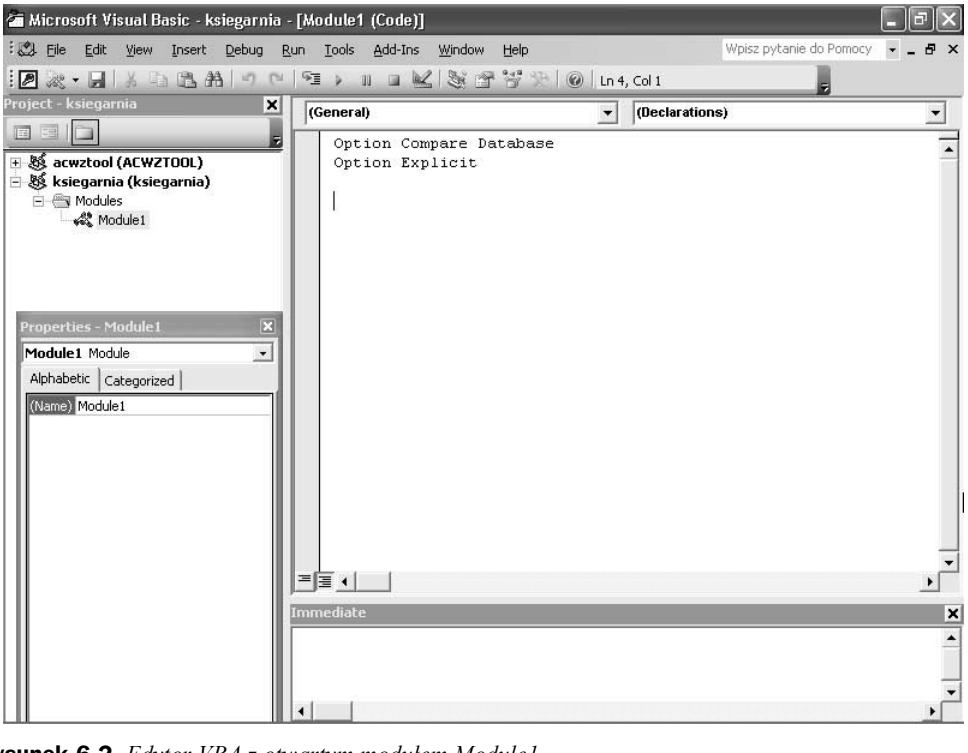
Zanim zaczniesz pisać kod VBA, musisz stworzyć moduł. Ponieważ moduły dla formularzy i raportów są tworzone wraz z formularzami i raportami, tutaj skupimy się na tworzeniu modułu standardowego.

Tworzenie modułów standardowych

Istnieje kilka różnych sposobów tworzenia modułów standardowych. Najłatwiej przejść do kategorii *Moduły* w oknie dialogowym *Baza danych*, jak to zostało pokazane na rysunku 6.1. Wystarczy wówczas jedynie kliknąć przycisk *Nowy*, aby zostać przeniesionym do edytora VBA, a w nim do wnętrza stworzonego modułu, co zostało pokazane na rysunki 6.2.



Rysunek 6.1. *Okno Baza danych po wybraniu kategorii Moduły*

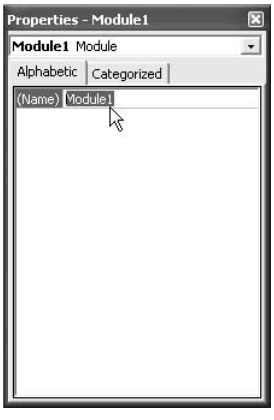


Rysunek 6.2. *Edytor VBA z otwartym modułem Module1*

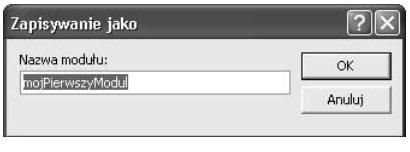
VBA nada nowemu modułowi tymczasową nazwę `Module1`. Zapewne zachcesz użyć nazwy bardziej opisowej. Możesz ją nadać w oknie *Properties* edytora VBA lub bezpośrednio w oknie *Baza danych*.

Możesz zaznaczyć nazwę `Module1` w oknie *Properties* tak, jak to widać na rysunku 6.3, i wpisać inną nazwę, powiedzmy `mojPierwszyModul` Gdy dokonasz zmiany w oknie *Properties*, nazwa zostanie zmieniona również w oknie *Project* w edytorze VBA i w oknie *Baza danych* w Accessie (w kategorii *Moduły*).

Rysunek 6.3.
Property Name (czyli Właściwość Nazwa)

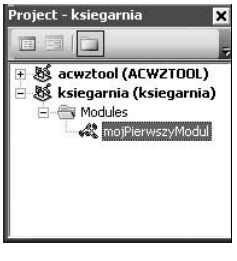


Dodatkowo powinieneś na pasku narzędziowym kliknąć przycisk *Save*. Zostaniesz poproszony o potwierdzenie nazwy:



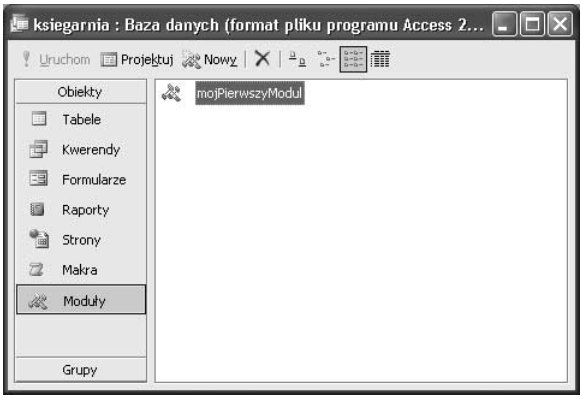
O tym, że nazwa została zmieniona, możesz się teraz upewnić w oknie *Project* (rysunek 6.4) i w oknie *Baza danych* (rysunek 6.5).

Rysunek 6.4.
Okno Project po zmianie nazwy modułu





Nazwę modułu możesz również zmienić, klikając ją prawym przyciskiem myszy w oknie dialogowym *Baza danych* i wybierając z menu podręcznego polecenie *Zmień nazwę*.



Rysunek 6.5. *Okno Baza danych po zmianie nazwy modułu*

Tworzenie procedur

Przypominam z rozdziału 5, że większość kodu VBA jest zapisywana w blokach zwanych procedurami. Dzielimy je na dwa rodzaje: procedury typu `Sub` (podprogramy) i procedury typu `Function` (funkcje). Podprogramy wykonują zadania, ale nie zwracają żadnej wartości, zaś funkcje zwracają wartości.

W większości przypadków na górze modułu będziesz widzieć dwie następujące linie:

```
Option Compare Database
Option Explicit
```

Są to tak zwane *deklaracje ogólne* modułu. Wszelki kod wpisany w tej sekcji będzie wpływał na wszystkie procedury w całym module.

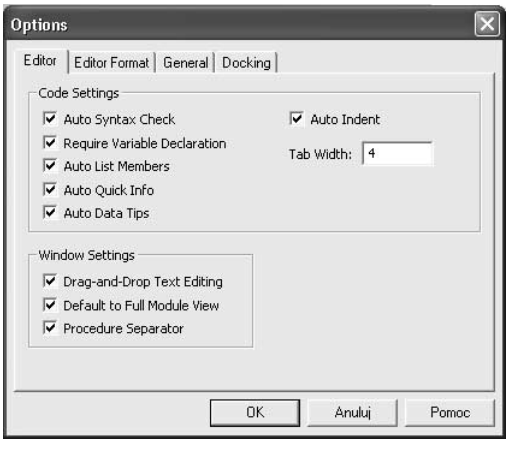
Linia `Option Compare` umożliwia wybranie jednego z trzech sposobów sortowania łańcuchów znakowych wewnątrz kodu:

- `Option Compare Database` — powoduje sortowanie w takim samym porządku jak sortowanie rekordów bazy danych i jest opcją najczęściej stosowaną.
- `Option Compare Binary` — powoduje sortowanie oparte na binarnych wartościach znaków. Wielkie litery są wtedy ustawiane przed małymi.
- `Option Compare Twxt` — powoduje sortowanie rozpoznające wielkie i małe litery z uwzględnieniem języka lokalnego.

W tej książce będziemy stosowali domyślne ustawienie `Option Compare Database`.

Linia `Option Explicit` jest, moim zdaniem, bardzo ważna. Wkrótce przekonasz się, że może ochronić Cię przed wieloma błędami kodowania, zmuszając do jawnego deklarowania zmiennych przed użyciem.

Jeżeli ta opcja nie pojawia się automatycznie wraz z tworzeniem modułu, wymuś to, wydając polecenie *Tools/Options* i na karcie *Editor* zaznaczając opcję *Require Variable Declaration*, tak jak na rysunku 6.6.



Rysunek 6.6. *Opcja Require Variable Declaration*

O procedurze myśl jak o miniprogramie mającym wykonać jedno wybrane zadanie. Na przykład zadaniem procedury może być dodanie do siebie dwóch liczb. Zawsze, gdy zechcesz dodać te dwie liczby, będziesz mógł wywołać tę procedurę. Nie mając procedury, za każdym razem, gdy należałoby wykonać to samo zadanie, musiałbyś pisać kod, w którym powtarzałby się ten sam ciąg poleceń.

Wewnątrz procedury deklarujesz zmienne, używasz pętli i poleceń `If`, a nawet wywołujesz inne procedury. Omówimy po kolei te wszystkie przypadki.

Na początku deklarujesz procedurę w sposób pokazany poniżej. Oczywiście, jej nazwę wybierasz wedle własnej woli.

```
Sub addNumbers()
End Sub
```

Zauważ, że edytor VBA nakreśli poziomą linię oddzielającą nową procedurę od deklaracji ogólnych. Pomaga to wyróżniać i oddzielać procedury i nie ma żadnego wpływu na ich działanie.

Jest ważne, abyś deklarując procedurę, zaczynał od słowa `Sub` i kończył deklarację nawiasami. Podkreślam, że to ważne, bo uważam, iż od początku należy stosować właściwą składnię. Jeżeli pominiesz nawiasy, edytor VBA wstawi je za Ciebie, gdy tylko naciśniesz *Enter*.

Zwykle w nawiasach umieszcza się wszelkie argumenty, jakich oczekuje procedura, ale nawet gdy nie ma żadnych argumentów, puste nawiasy muszą pozostać.

Zaraz po naciśnięciu klawisza *Enter* edytor VBA doda, zamykającą procedurę, linię `End Sub`. Cały napisany przez Ciebie kod musisz umieścić między wyrażeniem otwierającym i zamykającym.

Na kolejnych kartach książki będę robił uwagi o tym, jakie zwyczaje i praktyki programistyczne uważam za dobre. Choć nie są one bezwzględnie obowiązujące, są akceptowane przez wielu programistów i stały się standardami przemysłowymi.

Zgodnie z pierwszym zwyczajem, nazwa nadawana procedurze musi opisywać jej działanie, np. `addNumbers` (dodaj liczby). Każdy programista od razu rozpozna, do czego ta procedura służy. Dodatkowo, choć VBA nie rozróżnia liter wielkich i małych, czasami musi współdziałać z programami, które to robią. Ogólna konwencja nazw, wspomniana w rozdziale 4, nakazuje rozpoczynanie nazwy od małej litery, wyróżnianie wielkimi literami początków wyrazów w zbitkach słownych, niestosowanie spacji i zaczynanie nazw raczej od liter niż cyfr. Rozpoczynanie nazw od liter, a nie od cyfr to w wielu językach, również w VBA, więcej niż zwyczaj — to obowiązująca zasada, narzucona przez reguły języka VBA.

Drugi zwyczaj to zalecenie wcinania kodu procedury. Dzięki temu łatwo odszukać, gdzie się procedura rozpoczyna, a gdzie kończy. Aby zrobić wcięcie, zwykle trzykrotnie naciskam klawisz spacji lub używam klawisza Tab. Edytor VBA zapamięta wcięcie i automatycznie będzie je stosował do następnych linii.

Trzecim zwyczajem jest staranne komentowanie kodu. Możesz to z łatwością robić, wpisując pojedynczy apostrof na początku linii. Dzięki temu VBA zignoruje tę linię, traktując ją jako komentarz.

Poniższy przykład pokazuje wcięcie i komentarz:

```
Sub addNumbers()
    'Deklaracja zmiennych
End Sub
```

Zauważ kolor, jakiego VBA używa do wyróżnienia komentarza. Rozumienie znaczenia kolorów używanych przez edytor VBA podczas kodowania pomoże Ci śledzić, co się z kodem dzieje. Na przykład wyświetlenie tekstu na czerwono oznacza błąd składni.

Co oznaczają poszczególne kolory, możesz zobaczyć, wydając polecenie *Tools/Options*. Po otwarciu okna dialogowego *Options* karta *Editor Format* pokaże znaczenie poszczególnych kolorów, a ponadto pozwoli na dokonanie zmian schematu.

Deklarowanie zmiennych

Dwoma zasadniczymi składnikami procedur są zmienne i metody. Mówiąc po prostu, *zmienna* to „kawałek” informacji przechowywanej gdzieś w pamięci komputera. Może to być liczba, litera lub całe zdanie. Określenie miejsca przechowywanej informacji jest możliwe dzięki nadanej nazwie. Powiedzmy, że w kodzie mamy taką linię:

```
number = 23
```

Od tego momentu, jeżeli kod gdziekolwiek odwoła się do nazwy `number`, w to miejsce zostanie wstawiona liczba 23. Oczywiście, później możesz zmiennej o nazwie `number` nadać inną wartość.

Aby zmienna działała prawidłowo, powinieneś zadeklarować, jakiego typu informację ma przechowywać (w miarę uczenia się przyczyny tego staną się dla Ciebie oczywiste). W tabeli 6.1 zostały podane typy zmiennych i ich zastosowanie.

Tabela 6.1. *Typy zmiennych*

Typ zmiennej	Opis
`Boolean`	Zmienna `Boolean` zwraca wartości `True` lub `False` (`Prawda` lub `Fałsz`). Można je również wyrazić liczbowo: `0 = False`, `-1 = True`.
`Byte`	To jeden z najrzadziej używanych typów zmiennych. Może przechowywać tylko pojedynczą wartość od 0 do 255.
`Currency`	Robi to, co opisuje nazwa (waluta). Przechowuje wartości walutowe z czterema miejscami po przecinku, od –922 337 203 685 477,5808 do 922 337 203 685 477,5807.
`Date`	Przechowuje daty i czas. Ciekawe, że dopuszczalne lata to od 100 do 9999.
`Double`	To jeden z dwóch typów zmiennych zmiennoprzecinkowych. Drugi to `Single`. Zmienne `Double` służą do przechowywania bardzo długich liczb w zakresie od $-1.79769313486231 * 10^{308}$ do $-4.94065645841247 * 10^{-324}$ dla wartości ujemnych i od $4.94065645841247 * 10^{-324}$ do $1.79769313486232* 10^{308}$ dla wartości dodatnich.
`Integer`	To jeden z dwóch typów zmiennych całkowitych. Drugi to `Long`. Zmienne `Integer` przechowują liczby z zakresu od –32 768 do 32 767.
`Long`	`Long` to drugi to zmiennych całkowitych. Pierwszy to `Integer`. Zmienne `Long` przechowują liczby z zakresu od –2 147 483 648 do 2 147 483 657.
`Object`	Służy do przechowywania obiektów, które mają być później użyte.
`Single`	To drugi typ liczb zmiennoprzecinkowych. Pierwszym był `Double`.
`String`	To najczęściej używany typ zmiennych. Zmienna typu `String` może przechowywać do 2 miliardów znaków.

Do deklarowania zmiennych służy słowo kluczowe `Dim`. Na przykład:

```
Dim number As Integer
```

Ta deklaracja oznacza, że zmienna o nazwie `number` będzie przechowywała wyłącznie dane typu `Integer`. Zauważ, że jeszcze nie nadaliśmy zmiennej żadnej wartości (i edytor by nam na to nie pozwolił). Jedynie zadeklarowaliśmy nazwę i typ zmiennej. Gdzieś w dalszej części kodu umieścimy linię w rodzaju:

```
number = 32
```



Pamiętaj, że VBA nie odróżnia liter wielkich i małych. Nazwy zmiennej `number`, `Number` i `numBer` będą rozpoznawane jako ta sama nazwa.

Nadając zmiennej wartość, musisz zdawać sobie sprawę z wielu rzeczy. Po pierwsze, wartość nadawana zmiennej typu `String` musi być wzięta w cudzysłów[1]. Prawidłowe przypisanie wartości zmiennej typu `String` może wyglądać następująco:

```
Dim lastName As String
lastName = "Kowalski"
```

[1] Nie stosujemy cudzysłowów drukarskich — *przyp. tłum.*

Ponadto musisz pamiętać, że wartość nadawana zmiennej typu `Date` musi być ujęta w znaki #. Prawidłowe przypisanie wartości zmiennej typu `Date` może wyglądać następująco[2]:

```
Dim thisDate As Date
thisDate = #10/08/03#
```

W rozdziale 5 omówiliśmy konwencję nazw połączonych z obiektami bazy danych. Te same zasady odnoszą się do zmiennych. Pamiętaj, że nie są to zasady obowiązujące, lecz tylko konwencje przyjęte przez większość programistów. Przedrostki nazw są podane w tabeli 6.2.

Tabela 6.2. *Przedrostki zmiennych*

Typ zmiennej	Przedrostek
`Boolean`	`bln`
`Byte`	`byt`
`Currency`	`cur`
`Date`	`dat`
`Double`	`dbl`
`Integer`	`int`
`Long`	`lng`
`Object`	`obj`
`Single`	`sng`
`String`	`str`

Dobrym pomysłem jest, aby przy nadawaniu nazw obiektom jakiegokolwiek typu stosować nazwy opisowe. W ten sposób kod staje się „samoopisującym".

Oto przykład prawidłowego zadeklarowania zmiennej daty:

```
Dim datThisDate As Date
```

Wróćmy do procedury `addNumbers`, dodając deklaracje trzech zmiennych:

```
Sub addNumbers()
     'Deklaracja zmiennych
     Dim intNumber1 As Integer
     Dim intNumber2 As Integer
     Dim intSum As Integer
End Sub
```

Powinieneś już rozumieć znaczenie tych deklaracji. W zmiennych `intNumber1` i `int`
➥`Number2` zostaną zapamiętane dwie liczby. Po dodaniu ich suma zostanie zapamiętana w zmiennej `intSum`.

[2] Data podawana w stylu amerykańskim: miesiąc, dzień, rok — *przyp. tłum.*

Variant

Typ zmiennej, o jakim jeszcze nie mówiliśmy, to `Variant`. Zależnie od tego, z jakim programistą rozmawiasz, dowiesz się albo że jest to potężne narzędzie programistyczne, albo że to rozwiązanie dla niechlujnych programistów.

`Variant` pozwala VBA na samodzielne określanie, jaki typ danych jest zapamiętywany. Jest to domyślny typ zmiennych, używany, jeżeli w deklaracji zmiennej pominiesz klauzulę `as` *`typ_zmiennej`*. Przedrostkiem oznaczającym ten typ zmiennej jest `var`.

Przykładowa deklaracja zmiennej typu `Variant` może mieć postać:

```
Dim varMyData
```

Ponieważ pominęliśmy parametr `as` *`typ_zmiennej`*, zostanie użyty domyślny typ `Variant`. Taki sam efekt dałoby użycie deklaracji:

```
Dim varMyData As Variant
```

Powiedzmy, że tej zmiennej przypiszesz wartość w następujący sposób:

```
varMyData = "To jest zmienna Variant"
```

VBA przekształci wówczas typ `varMyData` na `String`. Jeżeli potem dokonasz kolejnego przypisania wartości:

```
varMyData = 12
```

VBA zmieni typ `varMyData` na `Integer`.

Jak się przekonamy w trakcie dalszej nauki, czasem prowadzi to do sytuacji, gdy zmienna `Variant` staje się zmienną nieoczekiwanego typu lub, co gorsza, końcowym rezultatem jest błąd. Wielu programistów uważa, że użycie zbyt wielu zmiennych typu `Variant` powoduje zbędne zajmowanie pamięci i spowalnia działanie kodu. Zanim więc zdecydujesz się na używanie zmiennych `Variant`, rozważ dobrze wszystkie „za" i „przeciw".

Najlepiej, abyś jako początkujący traktował zmienne `Variant` z wielką ostrożnością i stosował przede wszystkim jawne deklaracje typów podstawowych.

Constant

Wielokrotnie chcesz zadeklarować wartości, które nie będą się zmieniać. Używamy do tego typu `Constant`, deklarowanego za pomocą słowa kluczowego `Const`, zamiast `Dim`, stosowanego w normalnych deklaracjach.

Oto przykład:

```
Const conNumber1 As Integer = 12
Const conDate As Date = #3/2/2004#
```

Zauważ użycie przedrostka `con` zamiast standardowego przedrostka typu. Pamiętaj także, że deklarując stałą, musisz przypisać jej wartość, gdyż jeżeli tego nie zrobisz, przy opuszczaniu linii zostanie zgłoszony błąd składni.

Wejście i wyjście

Zatem wiesz już, jak za pomocą zmiennych zachowywać dane. Ale w jaki sposób wprowadzić informację, jaka ma być zapamiętana przez zmienną? Albo jak odczytać informacje zapamiętane przez zmienne? Pod wieloma względami właśnie tego będziemy się uczyć w pozostałej części książki. Ale dla przetestowania naszego kodu zacznijmy od paru najprostszych technik.

Jednym z najprostszych sposobów pobrania informacji od użytkownika i podstawienia jej pod zmienną jest użycie wbudowanej funkcji `InputBox`. Spowoduje to wyświetlenie prostego okna dialogowego z wezwaniem.

Kod naszego prostego przykładu uzupełnij pokazanymi niżej liniami, napisanymi pismem pogrubionym:

```
Sub addNumbers()
    'Deklaracja zmiennych
    Dim intNumber1 As Integer
    Dim intNumber2 As Integer
    Dim intSum As Integer

    'Użycie InputBox do wprowadzenia liczb
    intNumber1 = InputBox("Wpisz pierwszą liczbę")
    intNumber2 = InputBox("Wpisz drugą liczbę")
End Sub
```

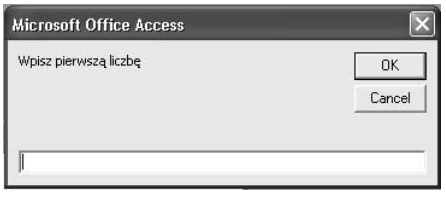
Zauważ, że dla jasności dodałem komentarz informujący, co robię.

Wyrażenie `intNumber1 = InputBox` jest nazywane *wyrażeniem przypisania*, gdyż przypisuje do zmiennej wartość podaną przez użytkownika. W tym przypadku zostały stworzone dwa przypisania — jedno dla `intNumber1`, a drugie dla `intNumber2`.

Możesz uruchomić kod, klikając przycisk *Run*:



Spowoduje to wyświetlenie pokazanego tu okna dialogowego do wprowadzenia pierwszej liczby (do wprowadzenia drugiej liczby zostanie wyświetlone oddzielne okno).



Jeżeli wykonałeś poprzednie kroki, kod powinien działać, zapamiętując w zmiennych podane liczby. Jednakże w żaden sposób nie informuje nas o tym. Są dwa sposoby stworzenia wyjścia. Pierwszy to dopisanie dwóch linii, napisanych poniżej pismem pogrubionym:

```
Sub addNumbers()
    'Deklaracja zmiennych
    Dim intNumber1 As Integer
    Dim intNumber2 As Integer
    Dim intSum As Integer

    'Użycie InputBox do wprowadzenia liczb
    intNumber1 = InputBox("Wpisz pierwszą liczbę")
    intNumber2 = InputBox("Wpisz drugą liczbę")

    'Dodanie obu liczb
    intSum = intNumber1 + intNumber2

    'Wyjście informacji
    Debug.Print "Podane liczby to " & intNumber1 & " i " & _
       intNumber2
End Sub
```

Jeżeli teraz uruchomisz kod, ponownie zostaną wyświetlone dwa okienka, te same co poprzednio, zaś komunikat wyjściowy pojawi się w oknie *Immediate* widocznym na dole okna edytora VBA.



To okno służy nam jako tymczasowe i przydatne jest w początkowej fazie testowania kodu.

Warto tu omówić kilka spraw. Po pierwsze linia:

```
Debug.Print "Podane liczby to " & intNumber1 & " i " & _
  intNumber2
```

jest przykładem *konkatenacji*, gdyż łączy w całość różne składniki. Tekst znakowy jest ujęty w cudzysłowy, zaś zmienne nie. Oba typy są oddzielone znakiem `&`.

Zauważ, że linia została podzielona spacją i znakiem podkreślenia. VBA wymaga tego, jeżeli jedno wyrażenie jest dzielone na wiele linii. W książce będziemy tak robili z powodu wymagań typograficznych. Jeżeli chcesz, możesz zapisywać wyrażenia kodu w jednej linii.

Innym sposobem formatowania wyjścia danych jest technika, z którą już się spotkaliśmy w rozdziale 4. Możesz użyć okna komunikatów, jeżeli poprzednią linię wyjścia zastąpisz następującą:

```
MsgBox "Podane liczby to " & intNumber1 & " i " & intNumber2
```

Spowoduje to wyświetlenie okna komunikatu:



A więc masz już swój pierwszy działający program napisany w VBA.

Struktury sterujące

Komputer działa sekwencyjnie. Wykonuje pierwsze polecenie i jeżeli nie ma żadnego błędu, przechodzi do następnego. Nazywamy to *strukturą sekwencyjną*. Ale co masz zrobić, jeżeli nie chcesz, aby kolejnym krokiem było wykonanie następnej linii kodu? W pewnych sytuacjach możesz zechcieć przekazać sterowanie programem do innego bloku linii kodu. Albo zechcesz, aby pewien blok linii kodu był powtarzany wielokrotnie, aż do wystąpienia określonej sytuacji (albo określoną liczbę razy).

Mechanizmy służące realizacji tych zadań nazywamy *strukturami sterującymi*. Struktury te dzielimy na dwie bardzo szerokie kategorie: struktury decyzyjne i struktury powtarzające. Jak się zaraz przekonasz, struktury decyzyjne pozwalają programowi na podejmowanie decyzji. Najpopularniejszą z nich jest struktura `If...Then`. Ale VBA ma ich więcej: `If...Then...Else`, `ElseIf`, `Select Case` i `IIf`.

Struktury powtarzające powodują wielokrotne wykonanie bloku linii — albo określoną liczbę razy, albo do zaistnienia sytuacji, która spowoduje przerwanie pętli.

W VBA mamy dwie główne struktury powtarzające: `For...Next` i `Do...Loop`. W ramach tych dwóch struktur zasadniczych istnieje wiele wariacji.

Struktury decyzyjne

Rozważmy pseudokod dla porannych czynności. Dla określenia działania w pewnym momencie możesz napisać:

```
If(jeżeli) pada deszcz
Then(to) wezmę parasol
Else(a w przeciwnym wypadku) po prostu pójdę do samochodu
```

Wyrazy pogrubione to słowa kluczowe konieczne do podjęcia decyzji.

Rozpoczynamy strukturę decyzyjną od porównania dwóch wartości. Czy A jest równe B? Czy A jest większe od B? Czy B jest prawdziwe? Są to tak zwane *wyrażenia warunkowe*. Symbole używane w tych wyrażeniach są podane w tabeli poniżej.

=	jest równe
<>	jest różne
>	jest większe od
<	jest mniejsze od
>=	jest większe lub równe
<=	jest mniejsze lub równe

Zobaczmy, jak prosta struktura decyzyjna działa w procedurze:

```
Sub ifTest()
    Dim intNum As Integer
    Dim strMessage As String
```

```
    intNum = 12

    If intNum > 10 Then
        strMessage = "Liczba jest równa " & intNum
    End If

    Debug.Print strMessage

End Sub
```

W tym kodzie należy zauważyć kilka rzeczy. Linia zawierająca wyrażenie warunkowe rozpoczyna się od `If` i kończy na `Then`. Ponadto, tak jak wszystkie struktury w VBA, również struktura warunkowa `If` musi się kończyć instrukcją `End`. W tym przypadku jest to `End If`.

Jeżeli uruchomisz tę przykładową procedurę, w oknie *Immediate* zobaczysz wynik pokazany na rysunku poniżej:



Rozważmy kilka problemów. Kod działa dobrze, ale co by się stało, gdyby liczba nie była większa od 10? A co ważniejsze, skoro wartość zmiennej jest sztywno ustalona, będzie ona *zawsze* większa od 10. Przyjrzyjmy się temu bliżej.

Za pomocą polecenia `Else` możesz wewnątrz struktury `If` umieścić ścieżkę alternatywną. Zmodyfikujmy nieco nasz kod:

```
Sub ifTest()
    Dim intNum As Integer
    Dim strMessage As String

    intNum = 9

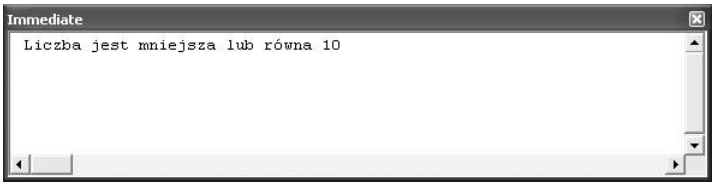
    If intNum > 10 Then
        strMessage = "Liczba jest większa od 10"
    Else
        strMessage = "Liczba jest mniejsza lub równa 10"
    End If

    Debug.Print strMessage

End Sub
```

Zauważ, że teraz nadaliśmy zmiennej `intNum` nową wartość oraz dodaliśmy instrukcję `Else`. Ponieważ instrukcja `Else` jest częścią instrukcji `If`, nie potrzeba dodatkowej instrukcji `End`.

Ponieważ wartość `intNum` jest mniejsza od 10, po uruchomieniu procedury zostanie włączona instrukcja `Else`, dając wynik pokazany na rysunku na następnej stronie.



Oczywiście, tak jak poprzednio możesz użyć funkcji wbudowanych `InputBox` i `MsgBox` do wprowadzania liczby i wyświetlania wyjściowego komunikatu.

AND, OR, NOT

W niektórych przypadkach zechcesz sprawdzić spełnienie warunków złożonych. Czy `intNum <1 OR intNum >= 10`? Słowa kluczowe `AND`, `OR` i `NOT` są nazywane *operatorami logicznymi* i służą do sprawdzania warunków złożonych.

W przypadku operatora logicznego `AND` oba wyrażenia logiczne muszą być prawdziwe, aby warunek `If` był spełniony.

Poniżej została podana tabela wszystkich możliwych wyników logicznych dla operatora `AND`.

Pierwszy argument logiczny	Drugi argument logiczny	Wynik
True	True	True
True	False	False
False	True	False
False	False	False

Jeżeli zostanie użyty operator `OR`, tylko jeden argument logiczny musi być prawdziwy, aby warunek `If` został spełniony. Następna tabela podaje wszystkie możliwe wyniki logiczne dla operatora `OR`.

Pierwszy argument logiczny	Drugi argument logiczny	Wynik
True	True	True
True	False	True
False	True	True
False	False	False

Ostatni operator logiczny `NOT` zwraca — zgodnie z oczekiwaniami — wartość przeciwną. Jeżeli wyrażenie warunkowe (A NIE jest równe B) jest prawdziwe, to jego zaprzeczenie `NOT` (A NIE jest równe B) jest fałszywe i odwrotnie — jeżeli A jest równe B, wyrażenie jest fałszywe, a wynik ostateczny prawdziwy. Nie przejmuj się, jeżeli potrzebujesz nieco czasu, aby sobie to wszystko wyobrazić.

Będziemy nadal zajmować się tą samą co poprzednio procedurą `ifTest` i użyjemy jej do wyjaśnienia wielokrotnego użycia instrukcji `If/Else`. Chcemy również ograniczyć wprowadzanie przez użytkownika liczb do zakresu od 1 do 15.

Zmień swój kod, aby wyglądał tak jak pokazany poniżej.

```
Option Compare Database
Option Explicit
Private intNum As Integer
---------------------------

Sub ifTest()
     Dim strMessage As String

     intNum = InputBox("Wpisz liczbę od 1 do 15", _
      "Testowanie struktury If")

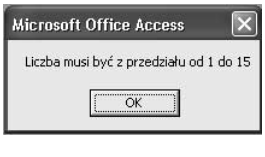
    If intNum >= 1 And intNum <= 15 Then
        ifTest2
    Else
        MsgBox "Liczba musi być z przedziału od 1 do 15"
    End If

End Sub
---------------------------
Sub ifTest2()
    If intNum > 10 Then
        MsgBox intNum & " jest liczbą większą od 10"
    Else
        MsgBox intNum & " jest liczbą mniejszą lub równą 10"
    End If
End Sub
```

Zauważ, że musiałeś przedefiniować `intNum` na zmienną globalną. Jeszcze o tym nie mówiliśmy, ale przypomnij sobie z początku tej części, że wszystko, co jest umieszczone w części deklaracji globalnych, obowiązuje dla całego modułu.

Procedura `ifTest` sprawdza, czy wartość `intNum` zawiera się w zakresie od 1 do 15. Jeżeli tak, wywołuje procedurę `ifTest2` zawierającą kod, którego używaliśmy poprzednio, a jeżeli nie — wyświetla komunikat dla użytkownika. Jeżeli jedna procedura uruchamia inną, nazywamy to *wywołaniem procedury*.

Spróbuj uruchomić ten kod wielokrotnie. Jeżeli wprowadzisz liczbę z przedziału od 1 do 15, otrzymasz komunikat, że jest ona większa albo mniejsza lub równa 10. Jeżeli wprowadzona liczba będzie spoza przedziału od 1 do 15, zostanie wyświetlony komunikat pokazany poniżej.



Struktura ElseIf

Za pomocą `ElseIf` możesz połączyć kilka struktur `If`. Na przykład:

```
Sub ifTest()
    Dim intNum As Integer
    intNum = 12
```

```
    If intNum = 1 Then
        Debug.Print "To jest najmniejsza liczba"
    ElseIf intNum = 15 Then
        Debug.Print "To jest największa liczba"
    Else
        Debug.Print "Ta liczba jest większa od 1 i mniejsza od 15"
    End If
End Sub
```

Możesz używać tylu poleceń `ElseIf`, ilu potrzebujesz do sprawdzenia potrzebnej liczby warunków.

Struktura Select Case

Jeżeli będziesz musiał wielokrotnie stosować `ElseIf`, powinieneś zastanowić się nad użyciem struktury `Select Case`. Dzięki temu kod stanie się czytelniejszy. Użycie `ElseIf` z poprzedniego przykładu można w następujący sposób zastąpić kodem korzystającym z `Select Case`:

```
Sub ifTest()
    Dim intNum As Integer
    intNum = 2
    Select Case intNum
    Case 1
        Debug.Print "To jest najmniejsza liczba"
    Case 15
        Debug.Print "To jest największa liczba"
    Case Else
        Debug.Print "Ta liczba jest większa od 1 i mniejsza od 15"
    End Select
End Sub
```

Oczywiście, możesz dodać więcej przypadków, zależnie od potrzeby. VBA będzie badać wszystkie przypadki, aż znajdzie ten, który odpowiada wartości `intNum` i wykona zapisane w nim instrukcje. Jeżeli żaden przypadek nie będzie odpowiadał wartości `intNum`, wykonane zostaną instrukcje domyślnego przypadku `Case Else`. Użycie `Case Else` jest opcjonalne, ale bardzo zalecane, aby struktura zawsze obejmowała wszystkie możliwe przypadki.

Jeżeli te same instrukcje mają być wykonane dla wielu przypadków, możesz użyć następującej składni:

```
Case 1, 2, 3...
```

lub

```
Case 1 To 4
```

IIf

Przyjrzymy się jeszcze szybko strukturze `IIf`, której nazwa pochodzi od Immediate If[3]. `IIf` jest przydatne, gdy chcemy nadać ostateczną wartość zmiennej, gdyż jego składnia jest samo wystarczalna. Prawidłowa składnia wygląda tak:

[3] Natychmiastowe (bezpośrednie) jeżeli — *przyp. tłum.*

```
IIf(warunek, wartość dla True, wartość dla False)
```

Kod roboczy może wyglądać mniej więcej tak:

```
strMessage = IIf(intNum > 10, "Liczba jest większa od 10", _
"Liczba nie jest większa od 10")
```

Struktura `IIf` działa dość wolno i dlatego nie jest zbyt często używana w dużych programach.

Teraz zwróćmy uwagę na struktury sterujące drugiego rodzaju, czyli pętle.

Pętle

Pętli używamy, gdy pewien blok instrukcji kodu ma być wykonywany albo określoną liczbę razy, albo dopóty, dopóki coś się nie zdarzy. Jeżeli liczba wykonań pętli jest określona, pętla jest sterowana przez licznik. Mówimy wówczas o *powtarzaniu sterowanym licznikiem*. Drugim rodzajem są pętle wykonywane tak długo, jak długo warunek jest spełniony. Mówimy wówczas o *powtarzaniu sterowanym warunkiem*. Przyjrzymy się obu typom.

Pętla For...Next

`For...Next` jest przykładem pętli sterowanej licznikiem. Za pomocą rzeczywistych liczb lub zmiennych możesz sterować następującymi elementami pętli:

- **Counter** (licznik) to serce pętli. Śledzi liczbę powtórzeń wykonanych przez pętlę.
- **Start** to wartość początkowa licznika. Rzadko się zdarza ustawianie innej wartości startowej niż 1 (czasami robi się to w przypadku różnych obliczeń matematycznych).
- **End** (koniec) to liczba wyznaczająca koniec pętli. Gdy licznik przekroczy tę wartość, wykonywanie pętli kończy się.
- **Step** (krok) wyznacza, o ile wartość licznika zwiększa się przy każdym wykonaniu pętli. Użycie kroku jest opcjonalne.

Oto przykład składni pętli `For...Next`:

```
Dim intCounter As Integer
For intCounter = 1 To 25
    ...
Next
```

Pamiętaj, że `Start` i `End` mogą być zmiennymi. Zauważ, że ostatnią linią pętli jest `Next`, a nie `End`, jak w strukturach, z jakimi dotychczas mieliśmy do czynienia. Jest to instrukcja zwiększenia wartości licznika o jeden krok.

Do zadeklarowania pętli możemy użyć również takiej składni:

```
For intCounter = 1 To 25 Step 5
```

Taka definicja powoduje zwiększenie licznika pętli, w tym przypadku intCounter, o pięć po każdym wykonaniu pętli. W rezultacie tego pętla ta wykona się pięć razy (intCounter będzie miał kolejne wartości 1, 6, 11, 16, 21).

Nie jest rzeczą niezwykłą użycie struktury If … Then wewnątrz petli, tak jak w poniższym przykładzie:

```
Sub forTest()
    Dim intCounter As Integer

    For intCounter = 1 To 10
        If (intCounter Mod 2) = 0 Then
            Debug.Print intCounter & " jest liczbą parzystą"
        Else
            Debug.Print intCounter & " jest liczbą nieparzystą"
        End If
    Next
End Sub
```

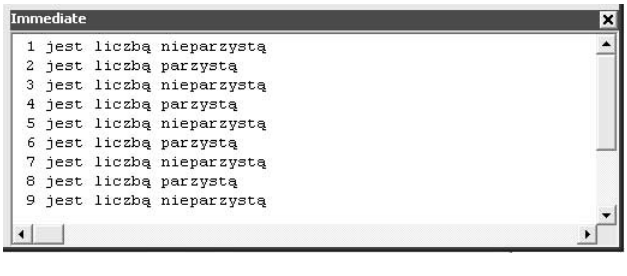
Już słyszę mojego wydawcę pojękującego, że wprowadziłem tu kilka rzeczy dodatkowych. Zatem przyjrzyjmy się dokładniej użytemu kodowi.

Jest to pętla `For...Next` mająca wykonać 10 powtórzeń. Zadaniem instrukcji wewnątrz pętli jest sprawdzenie, czy jest to przebieg parzysty, czy nieparzysty. Do tego celu posłużyła mi struktura `If...Then...Else`. Mamy zatem strukturę `If...Then...Else` wewnątrz struktury `For...Next`. Jeżeli jedna struktura występuje wewnątrz innej, mówimy o *strukturach zagnieżdżonych*, które są w programowaniu powszechnie stosowane.

Użyte wyrażenie warunkowe wykonuje specyficzne dzielenie. Użyłem słowa kluczowego `Mod`, które jest skrótem od *modulus*. `A Mod B` zwraca resztę z dzielenia liczby A przez liczbę B. Jeżeli liczbę parzystą podzielimy przez 2, resztą z dzielenia będzie 0. Zatem jeżeli licznik zostanie podzielony przez 2 i resztą z dzielenia będzie 0, warunek `If` zostanie spełniony i kolejna instrukcja wykonana. Jeżeli reszta z dzielenia będzie różna od zera, zostanie wykona instrukcja z części `Else`.

`Next` po prostu zwiększa wartość licznika.

Gdy uruchomisz tę procedurę, w oknie *Immediate* pojawią się następujące wyniki:



Zajmijmy się teraz innym rodzajem pętli.

Pętla Do

`Do` jest pętlą z powtarzaniem sterowanym warunkiem. Innymi słowy — pętlą działającą dopóty, dopóki określony warunek jest spełniony (na przykład coś jest czemuś równe lub od czegoś większe).

Istnieją dwa rodzaje pętli `Do` — `Do While` i `Do Until`. Przyjrzyjmy się im.

Do While

Pętla `Do While` sprawdza, czy warunek jest spełniony. Jeżeli tak, instrukcje pętli są wykonywane. Dla przykładu przyjrzyjmy się poniższej procedurze.

```
Sub doTest()
    Dim intCounter As Integer
    Dim intTest As Integer

    intTest = 1
    intCounter = 1

    Do While intTest = 1
        Debug.Print "To jest pętla numer " & intCounter

        If intCounter >= 5 Then
            intTest = 0
        End If

        intCounter = intCounter + 1
    Loop
End Sub
```

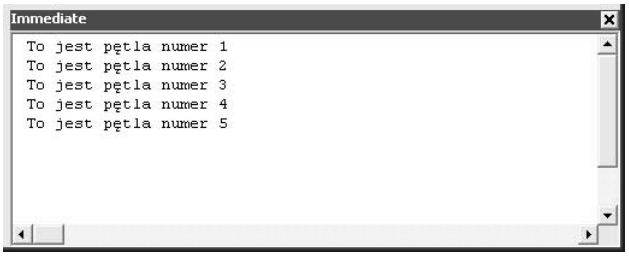
Jak widzisz, inaczej niż omawiana wcześniej `For...Next`, pętla `Do While` nie jest samowystarczalna. W tym przypadku musiałeś nadać wartości dwóm zmiennym — jedna posłużyła jako licznik, zaś druga jako zmienna testowa. W tym przypadku chciałem, aby procedura zakończyła działanie po wykonaniu piątego powtórzenia pętli.

Przeciwnie niż w pętli `For`, tutaj przebieg nie jest sterowany licznikiem, a powtarzanie pętli trwa, dopóki wartość `intTest` jest równa 1. Wewnątrz pętli zagnieździłem strukturę `If...Then`, sprawdzającą wartość licznika. Po osiągnięciu przez nią wartości 5 struktura `If` spowodowała zmianę wartości `intTest` na 0, co zakończyło działanie pętli.

Ponieważ sprawdzany jest warunek, a nie liczba iteracji, w pętlach `Do` nie musisz używać licznika. W tym przypadku wprowadziłem go, dodając zmienną `intCounter` i podnosząc jej wartość w ostatniej linii. Można to zastąpić wieloma innymi rozwiązaniami.

Zauważ, że ta struktura kończy się słowem `Loop`. Pamiętaj, że stosowanie licznika nie jest tu wymagane.

Jeżeli uruchomisz pętlę `Do While`, wyniki otrzymane w oknie *Immediate* powinny wyglądać jak na rysunku na następnej stronie.



Powstaje pytanie, co się stanie, jeżeli z jakiegoś powodu wartość zmiennej `inTest` nigdy nie będzie równa 1. Czy pętla `Do` w ogóle zostanie wykonana? Odpowiedź brzmi: nie. Nie zostanie wykonana. Co zatem począć, aby pętla została wykonana choć jeden raz?

```
Sub ifTest()
    Dim strMessage As String

    intNum = InputBox("Wpisz liczbę od 1 do 15", _
      "Testowanie struktury If")

    If intNum >= 1 And intNum <= 15 Then
        ifTest2
    Else
        MsgBox "Liczba musi być z przedziału od 1 do 15"
    End If

End Sub
---------------------------
Sub ifTest2()
    If intNum > 10 Then
        MsgBox intNum & " jest liczbą większą od 10"
    Else
        MsgBox intNum & " jest liczbą mniejszą od 10"
    End If
End Sub
```

Zauważ, że procedura `ifTest` zawiera sprawdzenie, czy użytkownik wprowadził liczbę z przedziału od 1 do 15, a jeżeli wprowadzona liczba będzie spoza tego zakresu, program po prostu zostanie zakończony i będziesz musiał uruchomić go ponownie. Chcemy, aby prośba o wprowadzenie liczby była ponawiana do czasu, aż zostanie wpisana liczba spełniająca ustalone kryteria.

Zmień procedurę `ifTest` w następujący sposób:

```
Sub ifTest()
    Dim strMessage As String
    Dim intTest As Integer

    intTest = 1

    Do
        intNum = InputBox("Wpisz liczbę od 1 do 15", _
          "Testowanie struktury If")
```

```
        If intNum >= 1 And intNum <= 15 Then
            intTest = 0
            ifTest2
        Else
            MsgBox "Liczba musi być z przedziału od 1 do 15"
        End If
    Loop While intTest = 1

End Sub
```

Zauważ, że nie użyliśmy licznika w żadnej postaci. Zamiast tego jest sprawdzana wartość zmiennej `intTest`. Interesujące, *gdzie* to sprawdzanie się odbywa. Inaczej niż w poprzednim przykładzie, możemy sprawdzanie warunku umieścić na końcu pętli, a nie na początku. Dzięki temu pętla zawsze zostanie wykonana przynajmniej jeden raz, co okazuje się bardzo przydatne w sytuacjach podobnych do zilustrowanej.

Do Until

Jest to pętla nieznacznie różniącą się od poprzednio omówionej. W `Do While` pętla jest wykonywana, dopóki warunek jest spełniony, zaś pętla `Do Until` działa, dopóki warunek nie zostanie spełniony.

Podana poprzednio procedura, pokazująca działanie pętli `Do While`, po niewielkich zmianach może korzystać z pętli `Do Until`:

```
Sub doTest()
    Dim intCounter As Integer
    Dim intTest As Integer

    intTest = 1
    intCounter = 1

    Do Until intTest <> 1
        Debug.Print "To jest pętla numer " & intCounter

        If intCounter >= 5 Then
            intTest = 0
        End If

        intCounter = intCounter + 1
    Loop
End Sub
```

Zauważ, że teraz pętla będzie działać dopóty, dopóki wartość `intTest` będzie *różna* od 1. Poza tym wszystko w tym przykładzie jest takie samo jak w poprzednim. Wynik działania powinien być taki sam jak w przykładzie z pętlą `Do While`.

Podobnie jak w pętli `Do While` możesz sprawdzanie warunku umieścić na końcu, dzięki czemu pętla będzie zawsze wykonywana przynajmniej jeden raz.

W dalszej części książki będziemy często używać pętli. Tak jak zmienne są one integralną częścią większości programów.

Tablice

Na wiele sposobów dyskusja o tablicach ściśle dotyczy zmiennych. *Tablica* to zmienna zawierająca wiele wartości. Liczba wartości przechowywanych przez tablicę musi być z góry określona i zadeklarowana. Musisz się także nauczyć odwoływania do wartości wewnątrz tablicy.

Gdy to pisałem, wydawca spytał: „A co z tablicami dynamicznymi?" Jak się potem przekonasz, tablice dynamiczne są poniekąd oszustwem. Nadal musisz z góry zadeklarować liczbę wartości zapamiętywanych wewnątrz tablicy. Jedyną różnicą jest to, że deklarację robisz przed użyciem, podczas działania programu, a nie przy pisaniu kodu.

Dowiesz się także, jak właściwie rezerwować (alokować) pamięć tak, aby Twoje tablice nie zajmowały zbyt wiele pamięci.

Składniki tablicy

Każda wartości w tablicy jest jej *elementem*. Ponieważ zmienna tablicowa zawiera wiele elementów, musisz wiedzieć, jak poszczególne z nich wybierać i jak się do nich odwoływać. Możesz to robić za pomocą liczby zwanej *indeksem*. Najczęściej pierwszy element tablicy ma indeks równy 0.

Gdybyś mógł zajrzeć „za kulisy" tablicy `strName` zawierającej nazwiska, zobaczyłbyś mniej więcej taki obraz:

```
strName        (0)  "Jan Kowalski"
               (1)  "Jacek Dydziuk"
               (2)  "Rozalia Bielska"
               (3)  "Anita Leparska"
               (4)  "Bohdan Gawerski"
```

Zauważ, że tablica zawiera pięć elementów, choć największą wartością indeksu jest 4. Powtarzam, że zwykle indeks pierwszego elementu jest równy 0 (w dalszej części tego rozdziału poznasz kilka odstępstw od tej zasady).

Jeżeli chcesz pobrać z tablicy i wydrukować imię i nazwisko Anity Leparskiej, musisz użyć instrukcji

```
Print strName(3)
```

Anita jest na pozycji odpowiadającej indeksowi 3, co — aby nie było zbyt prosto — oznacza czwarty element tablicy. Jest to przyczyną wielu problemów programistycznych i nieco dalej poznamy sposób korygowania tej komplikacji.

W VBA mamy dwa typy tablic:

- **Tablice statyczne** — w których liczba elementów, zwana długością tablicy, jest z góry określona i pozostaje niezmienna.
- **Tablice dynamiczne** — których zmienna długość nie jest ustalana z góry.

Tablice statyczne deklaruje się podobnie jak zmienne — z jedną niewielką różnicą:

```
Dim intMyScores(10) As Integer
```

Widząc coś takiego, musisz być ostrożny. Zapewne myślisz, że w ten sposób deklarujesz 10-elementową tablicę. W rzeczywistości tak zadeklarowana tablica ma 11 elementów o indeksach od 0 do 10 (indeks o najmniejszej wartości nazywamy *dolną granicą*, a indeks o największej wartości *górną granicą*).

W przypadku naszej tablicy dolna granica, czyli najmniejszy indeks, wynosi 0.

Musisz o tym pamiętać, aby właściwie alokować pamięć potrzebną do przechowywania tablicy.

Jeżeli chcesz, możesz w procedurze zadeklarować wiele tablic:

```
Dim strName(6) As String, intMyScores(10) As Integer
```

Domyślnie najmniejszy indeks jest równy 0, łańcuchy znaków są inicjowane jako puste, a zmienne całkowite z wartością zero.

Zobaczmy przykład. W tej procedurze stworzysz dwie pętle `For...Next`. Pierwsza spowoduje wypełnienie tablicy, zaś druga wydrukowanie jej zawartości. Kod tej procedury jest następujący:

```
Sub arrayTest()
    Dim i As Integer
    Dim intMyScores(10) As Integer

    For i = 0 To 10
        intMyScores(i) = InputBox("Wpisz wartość elementu " & i, _
          "Test tablicy statycznej")
    Next

    For i = 0 To 10
        Debug.Print "Wartość elementu tablicy z indeksem " & i & " to liczba " & _
          intMyScores(i)
    Next
End Sub
```

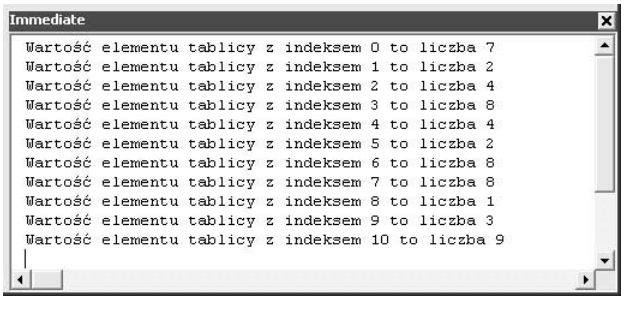
Programiści lubią używać małej litery `i` jako zmiennej reprezentującej indeks tablicy. To tylko przyjęta konwencja. Tutaj użyliśmy jej w podwójnej roli: licznika pętli `For...Next` oraz indeksu tablicy. Zauważ, że do elementu tablicy zawsze odwołujemy się przez jej nazwę z numerem elementu, czyli indeksem ujętym w nawiasy. W podanym przykładzie jako indeksu używamy licznika pętli `i`, co pozwala nam na wypełnienie tablicy.

Małym przyjemnym dodatkiem jest konkatenacja tekstu objaśniającego i indeksu w komunikacie wzywającym do podania liczby, dzięki czemu wiadomo, który element tablicy wypełniamy.

Okno wprowadzania wartości powinno wyglądać mniej więcej tak:



Po wprowadzeniu elementów zostaje uruchomiona druga pętla `For`, wyświetlająca wartości w oknie *Immediate*.



Rozmiar tablicy statycznej jest deklarowany bezpośrednio w kodzie. Mówiąc inaczej, jest to robione w *czasie projektowania*.

W podanym przykładzie jest jeden problem. Jeżeli błędnie zadeklarujesz dolną lub górną granicę, może to doprowadzić do błędu wykonania. VBA częściowo pomaga Ci tego uniknąć za pomocą dwóch wbudowanych funkcji: `LBound(`*`array name`*`)` i `UBound ➥(`*`array name`*`)`, które zwracają granice tablicy.

Składnię pętli z poprzedniego przykładu możesz zmienić następująco:

```
For i = LBound(intMyScores) To UBound(intMyScores)
    intMyScores(i) = InputBox("Wpisz wartość elementu " & i, _
      "Test tablicy statycznej")
Next
```

Tablice dynamiczne

Wielu programistów uważa tablice dynamiczne VBA poniekąd za fikcję. W zasadzie są to nadal tablice statyczne, ale ich rozmiary nie są deklarowane przed uruchomieniem programu. Tak więc jedyną różnicą jest to, kiedy rozmiary tablic są deklarowane.

Zaczynasz od zadeklarowanie pustej tablicy, na przykład:

```
Sub arrayTest()
    Dim i As Integer
    Dim intMyScores() As Integer
    Dim intArraySize As Integer

    intArraySize = InputBox("Ile wyników zamierzasz wpisać?", "Rozmiar tablicy")
```

```
    ReDim intMyScores(intArraySize)

    For i = 0 To intArraySize
        intMyScores(i) = InputBox("Wpisz wartość elementu " & i, _
          "Test tablicy statycznej")
    Next

    For i = 0 To intArraySize
        Debug.Print "Wartość elementu tablicy z indeksem " & i & " to liczba " & _
          intMyScores(i)
    Next
End Sub
```

Zauważ, że najpierw zadeklarowaliśmy `intMyScores` jako pustą tablicę. Potem za pomocą słowa kluczowego `ReDim` zadeklarowaliśmy tablicę statyczną z górną granicą określoną przez zmienną `intArraySize`, której wartość została wprowadzona przez użytkownika.

Następnie zmienna `intArraySize` posłużyła do sterowania pętlą.

Śledząc ten przykład, dostrzeżesz sprzeczność. Jeżeli wpiszesz, że chcesz wprowadzić 5 wyników, skończy się na wprowadzeniu 6, gdyż wartość indeksu rozpoczyna się od 0. To częsty błąd początkujących programistów.

Poprawiając nieco kod, możesz pierwszemu elementowi tablicy przypisać indeks 1 zamiast 0. To ułatwi koordynację działań. Przyjrzyjmy się następującemu kodowi:

```
Sub arrayTest()
    Dim i As Integer
    Dim intMyScores() As Integer
    Dim intArraySize As Integer

    intArraySize = InputBox("Ile wyników zamierzasz wpisać?", "Rozmiar tablicy")

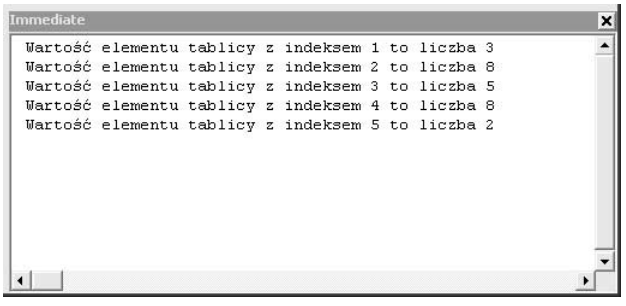
    ReDim intMyScores(1 To intArraySize)

    For i = 1 To intArraySize
        intMyScores(i) = InputBox("Wpisz wartość elementu " & i, _
          "Test tablicy statycznej")
    Next

    For i = 1 To intArraySize
        Debug.Print "Wartość elementu tablicy z indeksem " & i & " to liczba " & _
          intMyScores(i)
    Next
End Sub
```

Jeżeli uruchomisz ten kod, w oknie *Immediate* zobaczysz wyniki wyświetlone jak na pierwszym rysunku na następnej stronie.

`intArraySize` jest teraz górną granicą tablicy, zaś 1 — granicą dolną. Potem mamy dwie pętle `For...Next` startujące od 1 (pamiętaj, że teraz nie istnieje element 0).

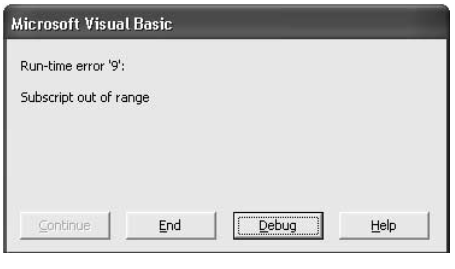


Istnieje również inna technika pozwalająca na rozpoczęcie indeksowania od 1. W sekcji deklaracji ogólnej możesz napisać `Option Base 0` lub `Option Base 1`. W ten sposób ustalisz dolną granicę tablic w danym module. Masz do wyboru tylko dwie wartości: 0 i 1.

Poza granicami

Co się stanie, jeżeli programując, zrobisz błąd, w wyniku którego nastąpi próba uzyskania dostępu do większej liczby elementów tablicy, niż zostało zadeklarowane w instrukcji `Dim` lub `ReDim`?

VBA nie odkryje tego błędu, dopóki program nie zostanie uruchomiony. Tego rodzaju błąd jest nazywany *błędem wykonania* (*runtime error*). Jeżeli nastąpi, zostanie wyświetlony komunikat podobny do pokazanego poniżej:



Kliknięcie przycisku *Debug* spowoduje pokazanie linii kodu uznanej przez program za błędną.

Bądź jednak ostrożny! Wskazana linia jest tą, na której program się załamał, jednakże zablokowanie wykonania programu może być wynikiem błędu popełnionego o wiele wcześniej. Mogłeś zmiennej określającej wielkość nadać złą wartość lub pomylić się w deklaracji `Dim` lub `ReDim`. Znalezienie przyczyny błędu może wymagać nieco pracy detektywistycznej.

Zmniejszanie tablicy

Co się stanie, jeżeli zadeklarowałeś górną granicę tablicy równą 10, ale nadałeś wartość jedynie czterem elementom?

Pamiętaj, że 10 określa pozycję ostatniego elementu tablicy. Jednakże w przeciwieństwie do przekroczenia wielkości tablicy, nie ma żadnych zasad nakazujących wykorzystanie wszystkich jej elementów. Użycie jedynie czterech z nich nie spowoduje żadnych problemów. Ale „za kulisami" tkwi pewien problem.

Jako początkujący nie musisz się tym przejmować, ale ważnym zagadnieniem w programowaniu jest kontrola wykorzystania zasobów. Pojemność pamięci komputera jest skończona. Jeżeli deklarujesz, że Twoja tablica będzie miała 10 elementów, VBA rezerwuje pamięć odpowiedniej wielkości. Oznacza to, że część pamięci pozostanie niewykorzystana. Jest to straszne marnowanie zasobów.

Zapewne Twoim pierwszym odruchem będzie próba użycia `ReDim` do zmiany wielkości deklaracji, ale to spowoduje inny, raczej poważny problem. Gdy w poprzednim przykładzie użyliśmy deklaracji `ReDim`, tablica nadal nie miała żadnych elementów. Jeżeli użyjesz `ReDim` przy wypełnionej tablicy, zostanie ona wymazana i stworzona od nowa. Istnieje spore prawdopodobieństwo, że nie będziesz z tego zadowolony.

VBA pozwala nam na wyjście z kłopotu przez połączenie `ReDim` z innym słowem kluczowym, jak to zostało pokazane poniżej:

```
ReDim Preserve inMyScores(4)
```

Słowo kluczowe `Preserve` powoduje ponowną alokację pamięci, pozostawiając jej elementy nietknięte.

Czyszczenie tablic

Czasami będziesz chciał zachować deklarację tablicy, ale wyczyścić jej elementy. Możesz to zrobić z łatwością za pomocą słowa kluczowego `Erase`, tak jak to zostało pokazane poniżej:

```
Erase intMyScores
```

Powoduje to wyczyszczenie zawartości tablicy z zachowaniem jej deklaracji.

Różne rzeczy mogą się zdarzyć, zależnie od typu tablicy. Jeżeli jest to tablica numeryczna, jej elementom zostanie nadana wartość 0. Jeżeli jest to tablica tekstowa (łańcuchów znaków), jej elementom zostanie nadana wartość "". Jest to pusty łańcuch znaków. W przypadku tablicy typu `Boolean` jej elementy zostaną ustawione na `False`.

IsArray

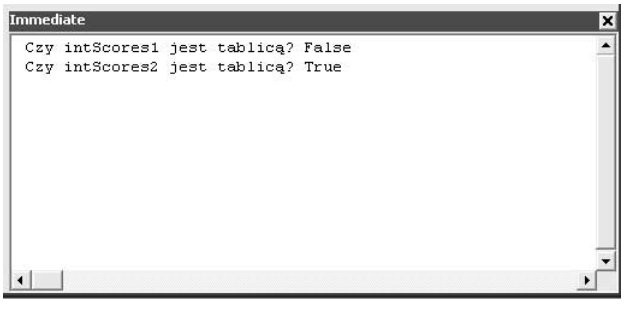
Jak można sprawdzić, czy zmienna jest tablicą? VBA posiada małą wygodną funkcję do testowania zmiennych. Przyjrzyj się poniższemu kodowi.

```
Sub arrayTest()
    Dim intScores1 As Integer
    Dim intScores2(4) As Integer
```

```
    Debug.Print "Czy intScores1 jest tablicą? " & IsArray(intScores1)
    Debug.Print "Czy intScores2 jest tablicą? " & IsArray(intScores2)

End Sub
```

Rezultat działania tego kodu będzie następujący:



Jak widzisz, `IsArray` jest funkcją typu `Boolean`. Zwraca wartość `True` lub `False`. W pierwszym przypadku powyżej wynikiem jest `False`, gdyż pierwsza zmienna nie została zadeklarowana jako tablica. W drugim przypadku wynikiem jest `True`, gdyż druga zmienna jest tablicą.

Podsumowanie

Wiele się nauczyliśmy w tym rozdziale. Poznaliśmy zmienne oraz różne struktury VBA. Pomimo to nasza znajomość podstaw VBA jest wciąż dalece niepełna. Podczas dalszej lektury książki poznamy kolejne elementy.

W pozostałych rozdziałach tej części z jeszcze większą uwagą i bardziej szczegółowo przyjrzymy się edytorowi VBA i poznanym już strukturom.