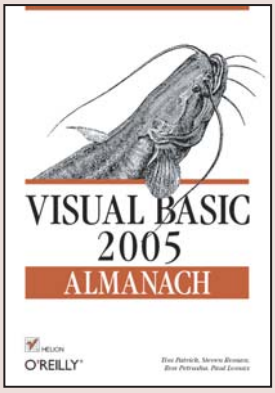
Visual Basic 2005. Almanach



Tim Patrick, Steven Roman, Ron Petrusha, Paul Lomax
Tłumaczenie: Mikołaj Szczepaniak
ISBN: 83-246-0475-8
Tytuł oryginału: Visual Basic 2005 in a Nutshell
Format: B5, stron: 880
Przykłady na ftp: 7 kB

Visual Basic po raz pierwszy pojawił się na rynku w roku 1991 jako połączenie oferowanego przez Microsoft języka QBasic z mechanizmem projektowania graficznego interfejsu użytkownika. Od tej pory przeszedł sporą ewolucję, zyskując jednocześnie ogromne grono zwolenników. W roku 2001 wprowadzono na rynek platformę programistyczną .NET wraz z całkowicie zmienioną i odnowioną wersją Visual Basica pod nazwą Visual Basic .NET. Visual Basic dla platformy .NET był językiem w pełni obiektowym i oferował znacznie większe możliwości niż jego poprzednicy. Visual Basic 2005 to najnowsze wcielenie tego popularnego języka programowania.

Książka „Visual Basic 2005. Almanach" to kompleksowe omówienie wszystkich zagadnień związanych z programowaniem w tym języku. Czytając ją, poznasz genezę platformy .NET, jej składniki i konstrukcję oraz słowa kluczowe języka Visual Basic. Przeczytasz o programowaniu obiektowym, klasach platformy .NET, obsłudze zdarzeń oraz typach uniwersalnych. Dalsza część książki to niezwykle przydatne każdemu programiście zestawienie dokumentacji wszystkich istotnych wyrażeń, procedur, funkcji i obiektów Visual Basica zawierające omówienie składni i argumentów, wskazówki dotyczące sposobu stosowania omawianego elementu języka, przykłady kodu oraz opisy nieudokumentowanych zachowań.

W książce omówiono:

- Podstawowe wiadomości o platformie .NET
- Zasady programowania obiektowego
- Zmienne, typy danych i operatory
- Struktura programów w Visual Basic
- Klasy platformy .NET
- Typy uniwersalne
- Obsługa błędów i wyjątków
- Leksykon elementów języka Visual Basic 2005

Zostań ekspertem w dziedzinie programowania w Visual Basicu

Wydawnictwo Helion
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

ROZDZIAŁ 4.

Zmienne i typy danych

Mechanizmy odpowiedzialne za przetwarzanie danych są sercem wszystkich aplikacji programowych. Moglibyśmy oczywiście przetwarzać dane tak jak robi to procesor komputera, czyli bit po bicie, jednak praca nad odpowiednimi algorytmami szybko by nas znużyła — właśnie dlatego takie języki jak Visual Basic oferują rozmaite **typy danych** oraz implementacje narzędzi zarządzających danymi (z których każdy bazuje na jakimś podzbiorze możliwych wartości danych). W niniejszym rozdziale zajmiemy się typami danych, sposobami zarządzania danymi przez te typy oraz technikami ich przetwarzania w języku Visual Basic i platformie .NET.

Pojęcie „typ danych" nie jest tożsame z bardziej ogólnym terminem „typ" wykorzystywanym w rozmaitych aspektach w tej i innych publikacjach poświęconych technologii .NET. Cała koncepcja platformy .NET bazuje na pojęciu **typu**, czyli podstawowej konstrukcji danych obejmującej klasy, struktury, delegacje i inne wysokopoziomowe elementy wykorzystywane w procesie konstruowania aplikacji oraz podczas przekazywania danych pomiędzy programami. Typy danych dostępne w platformie .NET bazują właśnie na tych ogólnie rozumianych typach (podobnie zresztą jak klasy budowane przez samych programistów). Typy danych oferują stosunkowo niewielki, ale bardzo istotny zbiór narzędzi przetwarzania danych pogrupowanych według podzbiorów możliwych wartości (zarządzanych w ramach poszczególnych typów danych).

Typy danych

Środowisko uruchomieniowe wspólnego języka platformy .NET (CLR) obejmuje między innymi wspólny system typów (CTS), który definiuje typy danych obsługiwane właśnie przez środowisko CLR. Wszystkie języki programowania przystosowane do współpracy z platformą .NET muszą implementować przynajmniej podzbiory typów danych środowiska CLR (część języków implementuje wszystkie, np. Visual Basic począwszy od wydania z 2005 roku).

W technologii .NET typy danych mają postać specjalnych klas i struktur, których egzemplarze reprezentują wartości danych należące do ograniczonych przedziałów (właściwych dla poszczególnych typów). Przykładowo, typ danych `Byte` oferuje możliwość reprezentowania i zarządzania 8-bitowymi wartościami całkowitoliczbowymi bez znaku, czyli liczbami z przedziału od `0` do `255`. Egzemplarze tego typu nie mogą zawierać wartości spoza tego przedziału (podzbioru), ale przynajmniej z tym konkretnym podzbiorem radzą sobie znakomicie. Platforma .NET oferuje typy danych dla tych podzbiorów wartości, które są najczęściej wykorzystywane

przez programistów podczas wytwarzania aplikacji. Za pomocą tych typów można reprezentować niemal dowolne kombinacje danych. Jeśli predefiniowane typy danych platformy .NET nie odpowiadają naszym potrzebom, możemy te typy wykorzystać w roli bloków składających się na naszą własną klasę zarządzającą danymi.

Platforma .NET Framework implementuje blisko dwadzieścia podstawowych typów danych, z których większość zaprojektowano z myślą o przetwarzaniu liczb całkowitych i zmiennoprzecinkowych. Rdzenne typy danych Visual Basica (znane jeszcze sprzed wprowadzenia platformy .NET) stanowią tylko opakowania dla wspomnianych typów podstawowych systemu CTS. Przykładowo, stosowany w Visual Basicu typ danych `Integer` jest opakowaniem struktury `System.Int32`. Jedną z wartości struktury `Int32` jest stała `MaxValue` reprezentująca maksymalną wartość numeryczną, którą można składować w tym typie danych. Oznacza to, że choć składowa `MaxValue` nie jest oficjalną częścią Visual Basica, pełna zależność typu danych `Integer` od wspomnianego typu podstawowego `Int32` systemu CTS powoduje, że poniższa para wyrażeń zostanie prawidłowo skompilowana i wykonana:

```
Dim usesInt32 As Integer
MsgBox(usesInt32.MaxValue)    ' Wyświetla 2147483647
```

Przed wydaniem wersji 2005 tylko niektóre spośród podstawowych typów danych platformy .NET były implementowane w Visual Basicu. Nawet mimo braku odpowiednich opakowań Visual Basica, wcześniejsze wydania tego języka (oczywiście już w ramach platformy .NET) udostępniały „nieopakowane" typy danych systemu CTS. Ponieważ podstawowe typy danych platformy .NET mają postać klas i struktur, można z nich korzystać w Visual Basicu dokładnie tak jak z innych klas czy struktur.

Typy wartościowe i referencyjne

Typy danych w języku Visual Basic można podzielić pomiędzy dwie dość ogólne kategorie: **typy wartościowe** (ang. *data types*) oraz **typy referencyjne** (ang. *reference types*). Typy wartościowe różnią się od typów referencyjnych przede wszystkim sposobem składowania w pamięci. Pamięć przydzielana zmiennej typu wartościowego zawiera właściwą wartość tej zmiennej. Przykładowo, w czasie wykonywania poniższego wyrażenia:

```
Dim simpleValue As Integer = 5
```

zostanie zarezerwowane miejsce w pamięci potrzebne do składowania przypisywanej wartości `5`. Inaczej jest w przypadku typów referencyjnych, gdzie pod adresami pamięci przydzielonymi tym zmiennym są składowane adresy innych bloków pamięci, w których znajdują się właściwe dane. Mechanizm stosowany w typach referencyjnych przypomina trochę usługę pocztową polegającą na przesyłaniu pod inny adres listów oryginalnie kierowanych do określonego odbiorcy. Przykładowo, dla poniższej deklaracji typu referencyjnego:

```
Dim somewhereElse As New MyCustomClass
```

kompilator Visual Basica utworzy w pamięci egzemplarz klasy `MyCustomClass`, po czym przypisze zmiennej `somewhereElse` rzeczywisty adres w pamięci tego egzemplarza. Programistom, którzy mają doświadczenie w pracy ze wskaźnikami oferowanymi przez takie języki jak C++, zrozumienie mechanizmu stosowanego w Visual Basicu nie powinno sprawić najmniejszych problemów, ponieważ oba rozwiązania są bardzo podobne.

Najkrócej mówiąc, zmienne typów wartościowych **zawierają** dane, natomiast zmienne typów referencyjnych tylko na te dane **wskazują**.

Różnice pomiędzy typami wartościowymi a typami referencyjnymi powodują szereg konsekwencji — jedną z najważniejszych jest sposób wykonywania popularnych operacji przypisania. Przeanalizujmy poniższą klasę, która zawiera tylko jedno pole składowe:

```
Public Class SimpleClass
   Public Age As Short
End Class
```

oraz strukturę będącą odpowiednikiem tej klasy:

```
Structure SimpleStruct
   Public Age As Short
End Structure
```

W przeciwieństwie do struktur, które są typami wartościowymi, klasy są typami referencyjnymi. Poniższy kod ilustruje podstawową różnicę w korzystaniu z tej pary podobnych, ale mimo wszystko różnych typów:

```
'----- Deklaruje cztery zmienne, po dwie dla każdego z typów.
Dim refType1 As SimpleClass
Dim refType2 As SimpleClass
Dim valType1 As SimpleStruct
Dim valType2 As SimpleStruct

'----- W pierwszej kolejności skoncentrujemy się na typach referencyjnych.
'      Przypisanie refType2 = refType1 spowoduje, że obie zmienne refType2
'      będą wskazywały na ten sam adres w pamięci. Od tej pory zmiany
'      składowych zmiennej refType1 będą miały wpływ na składowe zmiennej
'      refType2 i odwrotnie. Obie zmienne współdzielą ten sam egzemplarz.
refType1 = New SimpleClass
refType1.Age = 20
refType2 = refType1
refType2.Age = 30
Debug.WriteLine(refType1.Age)  '--> Wyświetla 30
Debug.WriteLine(refType2.Age)  '--> Wyświetla 30

'----- Przyjrzyjmy się teraz typom wartościowym. Przypisanie valType2 = valType1
'      powoduje wykonanie kopii składowych zmiennej valType1. Od tej pory zmiany
'      składowych jednej z tych zmiennych nie będą miały wpływu na wartości
'      składowych drugiej zmiennej.
valType1 = New SimpleStruct
valType1.Age = 20
valType2 = valType1
valType2.Age = 30
Debug.Writeline(valType1.Age)  '--> Wyświetla 20
Debug.Writeline(valType2.Age)  '--> Wyświetla 30
```

W pewnym sensie obie operacje przypisania jednej zmiennej drugiej zmiennej realizują to samo zadanie — kopiują wartość reprezentowaną przez zmienną użytą po prawej stronie do zmiennej na lewo od operatora przypisania. Ponieważ jednak rzeczywistą wartością typu referencyjnego (w tym przypadku zmiennej `refType1`) jest adres w pamięci, właśnie adres został skopiowany do zmiennej `refType2`. Ponieważ obie zmienne wskazują na ten sam obszar w pamięci (miejsce przechowywania składowych obiektu klasy `SimpleClass`), w praktyce zmienne `refType1` i `refType2` współdzielą jeden zbiór składowych.

Przypisanie jednej zmiennej typu wartościowego (`valType1`) innej zmiennej typu wartościowego (`valType2`) także polega na skopiowaniu wartości zmiennej użytej na prawo od operatora przypisania do zmiennej użytej z lewej strony tego operatora. Różnica polega na tym, że zmienna `valType1` zawiera rzeczywiste składowe (nie adres miejsca ich przechowywania w pamięci). Oznacza to, że zmienna docelowa `valType2` zawiera odrębną kopię tych składowych (w tym przypadku jedynej składowej `Age`).

Wyzerowanie zmiennej typu referencyjnego wymaga przypisania jej wartości `Nothing`. Typy wartościowe zawsze reprezentują jakieś wartości (nawet jeśli są to same zera), zatem w ich przypadku przypisywanie „wartości" `Nothing` nie jest możliwe.

Wszystkie podstawowe typy danych Visual Basica, które zarządzają wartościami numerycznymi (czyli np. `Integer` oraz `Double`), są typami wartościowymi. Typ danych `String` jest co prawda przykładem typu referencyjnego, jednak z perspektywy programisty funkcjonuje dokładnie tak jak typy wartościowe. Za każdym razem gdy przypisujemy jedną zmienną łańcuchową innej, wbrew pozorom w łańcuchu docelowym nie jest zapisywana referencja do pierwszego łańcucha (jak w przypadku innych typów referencyjnych). Wynika to z faktu, że stosowana w Visual Basicu implementacja typu danych `String` każdorazowo (zarówno podczas przypisywania, jak i modyfikowania) tworzy zupełnie nowy egzemplarz oryginalnego łańcucha.

Podstawowe typy danych Visual Basica: przegląd

Język programowania Visual Basic począwszy od wersji 2005 implementuje wszystkie podstawowe typy danych platformy .NET Framework (a konkretnie wspólnego systemu typów, CTS). Okazuje się, że podstawowe typy danych oferują bardzo szeroki zakres funkcjonalności i pozwalają zarządzać niemal wszystkimi kategoriami danych. Typy danych należące do systemu CTS można podzielić na pięć grup (według rodzaju zarządzanych danych):

Dane logiczne
: Pojedynczy typ danych, który obejmuje tylko jeden bit reprezentujący prawdę lub fałsz (odpowiednio `True` lub `False`).

Dane znakowe
: Visual Basic oferuje typy danych zarządzające zarówno pojedynczymi znakami, jak i długimi łańcuchami znaków.

Dane czasu i daty
: Pojedynczy typ danych zarządzający wartościami reprezentującymi zarówno datę, jak i godzinę.

Dane zmiennoprzecinkowe
: Rozmaite typy danych reprezentujące wartości zmiennoprzecinkowe — każdy z tych typów zarządza ograniczonym podzbiorem liczb wymiernych. Niektóre z tych typów oferują większą precyzję matematyczną od pozostałych.

Dane całkowitoliczbowe
: Ta kategoria obejmuje wiele całkowitoliczbowych typów danych, które umożliwiają składowanie liczb całkowitych należących do właściwego (zależnego od typu) przedziału (od wartości minimalnej do maksymalnej). Część typów danych całkowitoliczbowych obsługuje wartości ujemne.

W dalszej części tego punktu przedstawimy definicje i stosowne komentarze do każdej z wymienionych powyżej kategorii typów danych obsługiwanych przez Visual Basic.

Typ danych Boolean

Podstawowe fakty
: Podstawowy typ .NET: `System.Boolean`

  Implementacja: Typ wartościowy (struktura)

Ilość zajmowanej pamięci: 2 bajty

Zakres wartości: `True` lub `False`

Typ danych `Boolean` może reprezentować tylko dwie wartości (`True` lub `False`). W języku Visual Basic mamy do dyspozycji słowa kluczowe `True` i `False`, które są stosowane w roli wartości zmiennych typu `Boolean`. Zmiennym tego typu można też przypisywać wyniki dowolnych operacji logicznych.

Kiedy wartość numeryczna jest konwertowana na wartość typu `Boolean`, wszystkie wartości różne od zera są przekształcane w wartość `True`, a jedyną wartością tłumaczoną na `False` jest zero. W razie konwersji w drugą stronę wartość `False` jest zamieniana na zero, natomiast wartość `True` jest zamieniana na liczbę `-1`. (Ta część funkcjonalności odróżnia Visual Basica od pozostałych języków platformy .NET, które konwertują wartość `True` na `1`. W Visual Basicu zastosowano w tej roli liczbę `-1`, aby zapewnić zgodność wstecz. W razie współdzielenia danych logicznych pomiędzy komponentami napisanymi w różnych językach platformy .NET, środowisko uruchomieniowe .NET Framework i tak automatycznie przekształci te wartości w sposób gwarantujący pełną zgodność).

Typ danych Byte

Podstawowe fakty

Podstawowy typ .NET: `System.Byte`

Implementacja: Typ wartościowy (struktura)

Ilość zajmowanej pamięci: 1 bajt

Zakres wartości: Od `0` do `255` (bez znaku)

Typ danych `Byte` jest najmniejszym dostępnym w Visual Basicu typem całkowitoliczbowym bez znaku. Mimo bardzo niewielkiego przedziału obsługiwanych wartości zmienne typu `Byte` doskonale zdają egzamin podczas pracy z nieprzetworzonymi danymi binarnymi.

Typ danych Char

Podstawowe fakty

Podstawowy typ .NET: `System.Char`

Implementacja: Typ wartościowy (struktura)

Ilość zajmowanej pamięci: 2 bajty

Zakres wartości: Kody znaku z przedziału od `0` do `65535` (bez znaku)

Typ danych Char reprezentuje pojedynczy, 16-bitowy znak Unicode. Wszystkie znaki w platformie .NET są reprezentowane właśnie za pomocą kodów 16-bitowych, co oczywiście wystarczy do obsługi języków wymagających stosowania 2-bajtowego zbioru znaków (ang. *Double-Byte Character Set — DBCS), czyli np. języka japońskiego. Wersje Visual Basica sprzed* wprowadzenia platformy .NET nie zawierały żadnego odpowiednika typu danych `char`.

Stosując stałe wartości typu `Char`, należy do nich dołączać (za cudzysłowem zamykającym) pojedynczą literę `c`:

```
Dim singleLetter As Char = "A"c
```

Zmienna łańcuchowa (typu `String`), która zawiera tylko jeden znak, **nie** jest tożsama ze zmienną typu `Char` zawierającą ten sam znak. Ponieważ są to dwa zupełnie różne typy, ewentualne przenoszenie danych pomiędzy zmiennymi tych typów wymaga stosownej konwersji (jeśli włączono opcję `Option Strict`).

Typ danych DateTime

Podstawowe fakty

Podstawowy typ .NET: `System.DateTime`

Implementacja: Typ wartościowy (struktura)

Ilość zajmowanej pamięci: 8 bajtów

Zakres wartości: Od 1 stycznia 1 roku n.e. do 31 grudnia 9999 roku n.e. (w kalendarzu gregoriańskim)

Wartości reprezentujące daty i czas mają postać 64-bitowych, długich liczb całkowitych zgodnych ze standardem IEEE. Za pomocą tak długich liczb można reprezentować daty z przedziału od 1 stycznia 1 roku n.e. do 31 grudnia 9999 roku n.e. oraz godziny z przedziału od 0:00:00 do 23:59:59. Odpowiednie wartości reprezentują liczbę „tyknięć", które upłynęły od 1 stycznia 1 roku n.e. Każde takie „tyknięcie" odpowiada 100 nanosekundom.

Stałe daty należy umieszczać pomiędzy dwoma znakami krzyżyków (#):

```
Dim independenceDay As Date = #7/4/1776#
```

Typ danych Decimal

Podstawowe fakty

Podstawowy typ .NET: `System.Decimal`

Implementacja: Typ wartościowy (struktura)

Ilość zajmowanej pamięci: 12 bajtów

Zakres wartości: `+/-79 228 162 514 264 337 593 543 950 335` bez części dziesiętnej; `+/-7.9228162514264337593543950335` z 28 miejscami po przecinku; najmniejsza wartość różna od zera wynosi `+/-0.0000000000000000000000000001`

Wartości typu danych `Decimal` są składowane w formie 96-bitowych liczb całkowitych ze znakiem; reprezentacja tych wartości dodatkowo obejmuje wewnętrzny (obsługiwany i stosowany w pełni automatycznie) współczynnik skali z przedziału od `0` do `28`. Takie rozwiązanie zapewnia wysoki poziom precyzji matematycznej w przypadku liczb należących do odpowiedniego przedziału wartości (typ `Decimal` szczególnie dobrze sprawdza się w przypadku danych walutowych).

Na końcu wartości stałych typu `Decimal` należy umieszczać albo pojedynczą literę `D`, albo znak `@`:

```
Dim startingValue As Decimal = 123.45D
Dim endingValue As Decimal = 543.21@
```

Znak `@` można stosować także do oznaczania deklarowanych zmiennych jako egzemplarzy typu `Decimal`:

```
Dim startingValue@ = 123.45D
```

Składowe typu danych `Decimal` nazwane `MaxValue` i `MinValue` reprezentują granice obsługiwanego przedziału wartości (odpowiednio wartość maksymalną i minimalną).

W implementacjach Visual Basica sprzed wprowadzenia technologii .NET typ danych `Decimal` w praktyce nie stanowił odrębnego, autonomicznego typu danych — był podtypem typu danych `Variant`. Dopiero w wersjach kwalifikujących Visual Basic do rodziny języków platformy .NET zaimplementowano typ `Decimal` w formie pełnowartościowego typu danych.

Typ danych Double

Podstawowe fakty

Podstawowy typ .NET: `System.Double`

Implementacja: Typ wartościowy (struktura)

Ilość zajmowanej pamięci: 8 bajtów

Zakres wartości: Od `-1.79769313486231E+308` do `-4.94065645841247E-324` w przypadku liczb ujemnych; od `4.94065645841247E-324` do `1.79769313486232E+308` w przypadku liczb dodatnich

Wartości typu `Double` są zgodne ze standardem IEEE dla 64-bitowych (8-bajtowych) liczb zmiennoprzecinkowych podwójnej precyzji ze znakiem. Mimo ogromnego przedziału obsługiwanych wartości, stosując typ danych `Double` musimy się liczyć z utratą precyzji w pewnych obliczeniach matematycznych.

Stałe egzemplarze typu danych `Double` należy oznaczać wielką literą `R` lub znakiem krzyżyka (#) dołączanym bezpośrednio po wartościach numerycznych:

```
Dim startingValue As Double = 123.45R
Dim endingValue As Double = 543.21#
```

Znak krzyżyka (#) można stosować także do oznaczania deklarowanych zmiennych jako egzemplarzy typu `Double`:

```
Dim startingValue# = 123.45R
```

Typ danych Int32 (Integer)

Podstawowe fakty

Podstawowy typ .NET: `System.Int32`

Implementacja: Typ wartościowy (struktura)

Ilość zajmowanej pamięci: 4 bajty

Zakres wartości: Od `-2 147 483 648` do `2 147 483 647`

Typ danych `Integer` umożliwia reprezentowanie 32-bitowych liczb całkowitych ze znakiem. Właśnie tyle wynosi rdzenna długość słowa w procesorach 32-bitowych, zatem stosowanie tego rodzaju wartości powinno się przekładać na nieznacznie wyższą wydajność w porównaniu z pozostałymi typami całkowitoliczbowymi.

W wersjach Visual Basica sprzed wprowadzenia platformy .NET zmienne i stałe typu `Integer` zajmowały tylko 16 bitów i — tym samym — mogły służyć do reprezentowania liczb całkowitych z dużo mniejszego przedziału. W wersjach języka Visual Basic wchodzących w skład rodziny języków platformy .NET dodatkowo mamy do dyspozycji typ danych `Short`, czyli 16-bitowy typ całkowitoliczbowy ze znakiem.

Stałe egzemplarze typu `Integer` można opcjonalnie oznaczać wielką literą `I` lub znakiem procenta (`%`) dołączanym na końcu wartości numerycznej:

```
Dim startingValue As Integer = 123I
Dim endingValue As Integer = 543%
```

Znak procenta (`%`) można stosować także do oznaczania deklarowanych zmiennych jako egzemplarzy typu `Integer`:

```
Dim startingValue% = 123I
```

Typ danych Int64 (Long)

Podstawowe fakty

Podstawowy typ .NET: `System.Int64`

Implementacja: Typ wartościowy (struktura)

Ilość zajmowanej pamięci: 8 bajtów

Zakres wartości: Od `–9 223 372 036 854 775 808` do `9 223 372 036 854 775 807`

`Long` jest 64-bitowym typem danych całkowitoliczbowych ze znakiem. W wersjach Visual Basica sprzed wprowadzenia platformy .NET zmienne i stałe typu `Long` zajmowały tylko 32 bity i tym samym mogły służyć do reprezentowania liczb całkowitych z dużo mniejszego przedziału. W wersjach języka Visual Basic wchodzących w skład rodziny języków platformy .NET mamy do dyspozycji typ danych `Integer`, czyli 32-bitowy typ całkowitoliczbowy ze znakiem.

Stałe egzemplarze typu `Long` należy oznaczać wielką literą `L` lub znakiem `&` dołączanym na końcu wartości numerycznej:

```
Dim startingValue As Long = 123L
Dim endingValue As Long = 543&
```

Znak `&` można stosować także do oznaczania deklarowanych zmiennych jako egzemplarzy typu `Long`:

```
Dim startingValue& = 123L
```

Stosując znak `&` do oznaczania stałych typu `Long`, w żadnym razie nie należy pozostawiać spacji pomiędzy wartością całkowitoliczbową a tym znakiem, ponieważ znak `&` dodatkowo pełni w Visual Basicu funkcję operatora konkatenacji łańcuchów.

Typ danych Object

Podstawowe fakty

Podstawowy typ .NET: `System.Object`

Implementacja: Typ referencyjny (klasa)

Ilość zajmowanej pamięci: 4 bajty

Zakres wartości: Zmienna typu Object może reprezentować dowolny typ

`Object` jest uniwersalnym typem danych, co oznacza, że zmienna typu `Object` może się odwoływać do danych (wskazywać na dane) dowolnego innego typu danych. Przykładowo, egzemplarz klasy `Object` może wskazywać na wartość typu `Long`, wartość typu `String` lub egzemplarz dowolnej innej klasy.

```
Dim amazingVariable As Object
amazingVariable = 123L
amazingVariable = "Czyż to nie wspaniałe?"
amazingVariable = New MyCustomClass
```

Warto pamiętać, że ze stosowaniem zmiennych typu `Object` wiążą się pewne koszty w wymiarze wydajności oprogramowania. Visual Basic nie może związać właściwych składowych reprezentujących dane ze zmienną typu `Object` w czasie kompilacji, co oznacza, że odpowiednie łączenie musi nastąpić w czasie wykonywania programu — to z kolei powoduje, że przetwarzanie metod związanych z tym obiektem wymaga większej ilości kodu. Opisywana technika bywa nazywana **późnym wiązaniem** (ang. *late binding*). Deklarowanie obiektów z właściwymi (konkretnymi) typami danych skutkuje **wczesnym wiązaniem** (ang. *early binding*), ponieważ za zarządzanie związkami wszystkich składowych odpowiada kompilator. Przykładowo, poniższy fragment kodu:

```
Dim lateBound As Object
. . .
lateBound = New MyCustomClass
lateBound.SomeMethod(  )
```

wymusi na aplikacji dopasowanie zmiennej `lateBound` do składowej `SomeMethod` klasy `MyCustomClass` w czasie wykonywania. Przedstawione rozwiązanie jest więc dużo mniej wydajne od mechanizmu zastosowanego poniżej:

```
Dim earlyBound As MyCustomClass
. . .
earlyBound = New MyCustomClass
earlyBound.SomeMethod(  )
```

W wersjach języka Visual Basic sprzed wprowadzenia technologii .NET programista miał do dyspozycji funkcję `VarType`, która identyfikowała konkretny podtyp wartości typu `Variant`. Okazuje się, że funkcja `VarType` istnieje także w kolejnych wersjach Visual Basica .NET i służy do identyfikacji faktycznych typów zmiennych lub wartości. Klasa `System.Object` (i klasy potomne, czyli wszystkie klasy w technologii .NET) dodatkowo udostępnia metodę `GetType`, która zwraca informację o prawdziwym typie danego obiektu. Chociaż wymienione rozwiązania można stosować dla dowolnych typów danych, ich przydatność jest szczególnie widoczna właśnie w przypadku obiektów typu `Object`.

Typ danych SByte

Podstawowe fakty

Podstawowy typ .NET: `System.SByte`

Implementacja: Typ wartościowy (struktura)

Ilość zajmowanej pamięci: 1 bajt

Zakres wartości: Od `-128` do `127`

Nowość w wersji 2005. `SByte` jest najmniejszym typem danych całkowitoliczbowych ze znakiem obsługiwanym przez Visual Basica. Typ `SByte` jest więc odpowiednikiem opisanego wcześniej typu danych `Byte` (bez znaku) obejmującym znak.

`SByte` jest jednym z czterech typów danych dodanych do języka programowania Visual Basic w wydaniu z roku 2005, które nie są wymienione w minimalnej specyfikacji wspólnego języka (CLS). W związku z tym, komponenty i aplikacje zbudowane według zaleceń tego minimalnego standardu mogą nie być kompatybilne z aplikacjami wykorzystującymi ten typ danych.

Typ danych Int16 (Short)

Podstawowe fakty

Podstawowy typ .NET: `System.Int16`

Implementacja: Typ wartościowy (struktura)

Ilość zajmowanej pamięci: 2 bajty

Zakres wartości: Od `-32768` do `32767`

`Short` jest 16-bitowym typem danych całkowitoliczbowych ze znakiem. W wersjach języka Visual Basic sprzed wprowadzenia technologii .NET funkcję 16-bitowego typu całkowitoliczbowego ze znakiem pełnił inny typ danych, `Integer`; typ danych `Short` w ogóle nie występował w tamtych wydaniach Visual Basica.

Stałe egzemplarze typu `Short` należy oznaczać wielką literą `S` dołączaną na końcu wartości numerycznej:

```
Dim startingValue As Short = 123S
```

Typ danych Single

Podstawowe fakty

Podstawowy typ .NET: `System.Single`

Implementacja: Typ wartościowy (struktura)

Ilość zajmowanej pamięci: 4 bajty

Zakres wartości: Od `-3.4O2823E+38` do `-1.4O1298E-45` w przypadku liczb ujemnych; od `1.4O1298E-45` do `3.4O2823E+38` w przypadku liczb dodatnich

Wartości typu `Single` są zgodne ze standardem IEEE dla 32-bitowych (4-bajtowych) liczb zmiennoprzecinkowych pojedynczej precyzji ze znakiem. Mimo stosunkowo dużego przedziału obsługiwanych wartości, stosując typ danych `Single`, musimy się liczyć z utratą precyzji w pewnych obliczeniach matematycznych.

Stałe egzemplarze typu danych `Single` należy oznaczać wielką literą `F` lub znakiem wykrzyknika (`!`) dołączanym bezpośrednio po wartościach numerycznych:

```
Dim startingValue As Single = 123.45F
Dim endingValue As Single = 543.21!
```

Znak wykrzyknika (`!`) można stosować także do oznaczania deklarowanych zmiennych jako egzemplarzy typu `Single`:

```
Dim startingValue! = 123.45F
```

Typ danych String

Podstawowe fakty

Podstawowy typ .NET: `System.String`

Implementacja: Typ referencyjny (klasa)

Ilość zajmowanej pamięci: 10 + (2 * *długość_łańcucha*) bajtów

Zakres wartości: Od zera do około dwóch miliardów znaków Unicode

Typ danych `String` umożliwia reprezentowanie łańcuchów znakowych zmiennej długości złożonych maksymalnie z około dwóch miliardów znaków.

Wszystkie łańcuchy w technologii .NET są niezmienne (ang. *immutable*). Oznacza to, że wartości raz przypisanej zmiennej łańcuchowej nie można zmieniać. Kiedy modyfikujemy zawartość łańcucha, typ danych `String` zwraca nowy egzemplarz uwzględniający wprowadzone zmiany.

Zmienna `String` zawierająca pojedynczy znak nie jest tożsama ze zmienną typu `Char` zawierającą ten sam pojedynczy znak. Ponieważ są to dwa zupełnie różne typy, ewentualne przenoszenie danych pomiędzy zmiennymi tych typów wymaga stosownej konwersji (jeśli włączono opcję `Option Strict`).

Typ danych UInt32 (UInteger)

Podstawowe fakty

Podstawowy typ .NET: `System.Int32`

Implementacja: Typ wartościowy (struktura)

Ilość zajmowanej pamięci: 4 bajty

Zakres wartości: Od `0` do `4 294 967 295` (bez znaku)

Nowość w wersji 2005. **`UInteger`** jest 32-bitowym typem danych całkowitoliczbowych bez znaku. Typ `UInteger` jest więc odpowiednikiem bez znaku opisanego wcześniej typu danych `Integer` (ze znakiem).

`UInteger` jest jednym z czterech typów danych dodanych do języka programowania Visual Basic w wydaniu z roku 2005, które nie są wymienione w minimalnej specyfikacji wspólnego języka (CLS). W związku z tym, komponenty i aplikacje zbudowane według zaleceń tego minimalnego standardu mogą nie być kompatybilne z aplikacjami wykorzystującymi ten typ danych.

Typ danych UInt64 (ULong)

Podstawowe fakty

Podstawowy typ .NET: `System.Int64`

Implementacja: Typ wartościowy (struktura)

Ilość zajmowanej pamięci: 8 bajtów

Zakres wartości: Od `0` do `18 446 744 073 709 551 615` (bez znaku)

Nowość w wersji 2005. **`ULong`** jest 64-bitowym typem danych całkowitoliczbowych bez znaku. Typ `ULong` jest więc odpowiednikiem bez znaku opisanego wcześniej typu danych `Long` (ze znakiem).

`ULong` jest jednym z czterech typów danych dodanych do języka programowania Visual Basic w wydaniu z roku 2005, które nie są wymienione w minimalnej specyfikacji wspólnego języka (CLS). W związku z tym, komponenty i aplikacje zbudowane według zaleceń tego minimalnego standardu mogą nie być kompatybilne z aplikacjami wykorzystującymi ten typ danych.

Typ danych UInt16 (UShort)

Podstawowe fakty

Podstawowy typ .NET: `System.Int64`

Implementacja: Typ wartościowy (struktura)

Ilość zajmowanej pamięci: 2 bajty

Zakres wartości: Od `0` do `65535` (bez znaku)

Nowość w wersji 2005. **`UShort`** jest 16-bitowym typem danych całkowitoliczbowych bez znaku. Typ `UShort` jest więc odpowiednikiem bez znaku opisanego wcześniej typu danych `Short` (ze znakiem).

`UShort` jest jednym z czterech typów danych dodanych do języka programowania Visual Basic w wydaniu z roku 2005, które nie są wymienione w minimalnej specyfikacji wspólnego języka (CLS). W związku z tym, komponenty i aplikacje zbudowane według zaleceń tego minimalnego standardu mogą nie być kompatybilne z aplikacjami wykorzystującymi ten typ danych.

Typy danych definiowane przez użytkownika

Chociaż pojedyncze zmienne w zdecydowanej większości przypadków zaspokajają potrzeby programistów, nierzadko bardziej efektywnym rozwiązaniem jest łączenie wielu podstawowych wartości danych w ramach logicznych grup. Te **niestandardowe typy danych (definiowane przez użytkownika)** rozszerzają zbiór podstawowych typów danych o nowe typy dostosowane do potrzeb konkretnych programów.

Wersje języka Visual Basic sprzed wprowadzenia technologii .NET oferowały możliwość tworzenia typów danych definiowanych przez użytkownika za pośrednictwem wyrażenia `Type`. Budowane w ten sposób struktury danych były zwykłymi grupami zmiennych pozbawionymi jakiejkolwiek funkcjonalności (oczywiście poza możliwością ustawiania i odczytywania wartości reprezentowanych w poszczególnych składowych). W wydaniach Visual Basica wchodzących w skład platformy .NET Framework znacznie rozszerzono tego rodzaju mechanizmy, implementując możliwość definiowania kodu zarówno we wspomnianych strukturach danych, jak i podstawowych elementach platformy .NET. Typy znane z Visual Basica 6 zastąpiono konstrukcjami powszechnie stosowanymi w technologii .NET — strukturami (definiowanymi ze słowem kluczowym `Structure`).

Podstawową konstrukcją gromadzącą kod i dane w platformie .NET jest klasa. Klasy pod wieloma względami przypominają struktury, ale są wolne od pewnych ograniczeń, które dotyczą wyłącznie struktur. Bodaj najważniejszą różnicą dzielącą struktury od klas jest to, że w przeciwieństwie do klas (które implementują typy referencyjne), struktury implementują typy wartościowe (dziedziczące bezpośrednio po typie `System.ValueType`).

Aby zadeklarować strukturę, należy użyć wyrażenia `Structure`:

```
[Public|Private|Friend] Structure structureName
   deklaracje składowych
End Structure
```

Składowymi struktury mogą być pola, właściwości, metody, zdarzenia dzielone, typy wyliczeniowe oraz inne, zagnieżdżone struktury. Każda ze składowych musi zostać zadeklarowana z jednym z trzech modyfikatorów dostępu: `Public`, `Private` lub `Friend`.

Najprostszym i najczęściej spotykanym zastosowaniem struktur jest grupowanie wzajemnie powiązanych zmiennych (nazywanych polami). Przykładowo, w naszym programie możemy korzystać z prostej struktury definiującej podstawowe informacje o jednej osobie:

```
Structure Person
   Public Name As String
   Public Address As String
   Public City As String
   Public State As String
   Public Zip As String
   Public Age As Short
End Structure
```

Poniżej przedstawiono standardową deklarację definiującą zmienną typu `Person`:

```
Dim onePerson As Person
```

Dostęp do składowych struktury wymaga użycia standardowej składni „z kropką", która jest stosowana także w przypadku składowych klas:

```
onePerson.Name = "Beethoven"
```

Bardziej złożone struktury mogą zawierać zarówno proste pola składowe, jak i właściwości:

```
Public Structure NameAndState
   ' ----- Pola publiczne i prywatne.
   Public Name As String
   Private theState As String

   Public Function ShowAll(  ) As String
      ' ----- Metoda publiczna. Wyświetla wszystkie składowane wartości.
      If (theState = "") And (Name = "") Then
         Return "<No Name> from <Nowhere>"
      ElseIf (theState = "") Then
         Return Name & " from <Nowhere>"
      ElseIf (Name = "") Then
         Return "<No Name> from " & theState
      Else
         Return Name & " from " & theState
      End If
   End Function

   Public Property State(  ) As String
      ' ----- Właściwość publiczna. Ogranicza zbiór wartości reprezentujących stany.
      Get
         Return theState
      End Get
      Set(ByVal value As String)
         If (Len(value) = 2) Then
            theState = UCase(value)
         Else
            Throw New System.ArgumentException( _
               "Stan musi być reprezentowany przez dwa znaki.", "State")
         End If
      End Set
   End Property
End Structure
```

Egzemplarze tej struktury mogą być od tej pory tworzone i wykorzystywane dokładnie tak jak egzemplarze klas:

```
Dim onePerson As New NameAndState
onePerson.Name = "Donna"
onePerson.State = "CA"
MsgBox(onePerson.ShowAll())
```

Struktury mogą być przekazywane na wejściu funkcji w formie argumentów lub wykorzystywane w roli typów zwracanych przez funkcje. Chociaż struktury pod wieloma względami przypominają klasy, nie obsługują szeregu rozwiązań znanych z klas:

- Struktury nie mogą ani dziedziczyć po innych strukturach, ani same nie mogą być dziedziczone.
- Wszystkie konstruktory struktur muszą pobierać parametry wejściowe.
- Struktury nie mogą definiować destruktorów (metoda `Finalize` i tak nigdy nie jest wywoływana).

- Deklaracje składowych nie mogą inicjalizować ich wartości, stosować konstrukcji składniowej `As New` ani określać początkowego rozmiaru tablic.

Szczegółowe omówienie terminologii programowania można znaleźć w rozdziale 3.

Konwersja typów danych

Proces konwersji wartości jednego typu na wartość innego typu nazywamy **rzutowaniem** (ang. *casting*) lub po prostu **konwersją**. Techniki konwersji mogą być stosowane dla wartości stałych, zmiennych lub wyrażeń określonego typu. Język programowania Visual Basic oferuje szereg funkcji konwertujących, które rzutują wartości jednego typu danych do postaci ich odpowiedników innego typu danych:

```
Dim miniSize As Byte = 6
Dim superSize As Long
superSize = CLng(miniSize)   'Konwertuje wartość zmiennej typu Byte na wartość typu Long.
superSize = CLng("12")       'Konwertuje stałą typu String na wartość typu Long.
```

Rzutowania mogą mieć charakter działań rozszerzających lub zawężających. **Rzutowanie rozszerzające** (ang. *widening cast*) ma miejsce wtedy, gdy docelowy typ danych może z powodzeniem reprezentować wszystkie możliwe wartości typu źródłowego (z taką sytuacją mamy do czynienia np. wtedy, gdy konwertujemy wartość typu `Short` na wartość typu `Integer` lub wartość typu `Integer` na wartość typu `Double`). Rzutowanie rozszerzające nigdy nie powoduje utraty danych. **Rzutowanie zawężające** (ang. *narrowing cast*) polega na konwersji oryginalnego typu danych na typ, który nie może reprezentować wszystkich możliwych danych typu źródłowego. Rzutowanie zawężające może prowadzić do utraty części danych lub wręcz błędu konwersji.

Konwersje typów w Visual Basicu mogą być realizowane na dwa sposoby: jawnie (wprost) lub w ukryciu. **Konwersja niejawna** (ang. *implicit conversion*) jest realizowana przez kompilator, jeśli okoliczności na to pozwalają (i jeśli samo rzutowanie jest prawidłowe). Przykładowo, jeśli umieścimy w naszej aplikacji następującą sekwencję wyrażeń:

```
Dim smallerData As Integer = 3948
Dim largerData As Long
largerData = smallerData
```

wartość zmiennej `smallerData` automatycznie zostanie rzutowana do większego typu danych `Long` (wykorzystywanego przez zmienną `largerData`). Ten rodzaj konwersji niejawnej po części jest uzależniony od ustawienia wyrażenia `Option Strict`. Wyrażenie `Option Strict` powinno się znajdować na samym początku pliku z kodem źródłowym (przed właściwym kodem klasy):

```
Option Strict {On | Off}
```

Jeśli opcja `Option Strict` jest włączona (`On`), kompilator będzie automatycznie wykonywał tylko rzutowania rozszerzające, co oznacza, że takie rzutowania zawężające jak:

```
Dim smallerData As Integer
Dim largerData As Long = 3948
smallerData = largerData
```

spowodują wygenerowanie błędów kompilacji (nawet jeśli faktycznie konwertowane dane mogą być bez trudu reprezentowane przez typy zmiennych docelowych). Oznacza to, że należy zastosować **konwersję jawną** (ang. *explicit conversion*):

```
smallerData = CInt(largerData)
```

Ustawiając wartość `Off` w opcji `Option Strict`, zezwalamy na stosowanie techniki konwersji niejawnej nawet wtedy, gdy może to prowadzić do występowania błędów w czasie wykonywania programu.

Oprócz wspomnianej opcji `Option Strict`, Visual Basic oferuje też wyrażenie `Option Explicit`, które również powinno być stosowane na początku plików z kodem źródłowym:

```
Option Explicit {On | Off}
```

Kiedy opcja `Option Explicit` jest włączona (`On`), wszystkie zmienne muszą być deklarowane (za pomocą słowa kluczowego `Dim` lub innego, podobnego) przed użyciem. Kiedy opcja `Option Explicit` jest wyłączona (`Off`), kompilator Visual Basica automatycznie doda (w czasie kompilacji) niezbędne deklaracje dla wszystkich nazw napotkanych zmiennych, które nie zostały wcześniej zadeklarowane. (Nowe wyrażenia `Dim` nie zostaną umieszczone w naszym kodzie źródłowym — niezbędne deklaracje zostaną dodane „po cichu" w czasie kompilacji). Wyłączenie tej opcji może prowadzić do występowania w programie trudnych do zlokalizowania błędów. Więcej informacji na ten temat można znaleźć w podrozdziale „Wyrażenie Option Explicit" w rozdziale 12. Wartości domyślne obu opcji (`Option Strict` oraz `Option Explicit`) można ustawić we właściwościach projektu.

Język Visual Basic zawiera funkcje konwertujące dla wszystkich podstawowych typów danych.

Funkcja `CBool`
: Konwertuje dowolny prawidłowy łańcuch lub wyrażenie numeryczne na wartość typu `Boolean`. Kiedy wartość numeryczna jest konwertowana na wartość typu `Boolean`, wszystkie liczby różne od zera są zamieniane na `True` i tylko zero jest zamieniane na `False`.

Funkcja `CByte`
: Konwertuje dowolne wyrażenie numeryczne (którego wartość należy do przedziału obsługiwanego przez typ danych `Byte`) na wartość typu `Byte`, zaokrąglając ewentualną część ułamkową.

Funkcja `CChar`
: Konwertuje pierwszy znak łańcucha na wartość typu danych `Char`.

Funkcja `CDate`
: Konwertuje dowolną prawidłową reprezentację daty lub godziny na egzemplarz typu `Date`.

Funkcja `CDbl`
: Konwertuje dowolne wyrażenie numeryczne (którego wartość należy do przedziału obsługiwanego przez typ danych `Double`) na wartość typu `Double`.

Funkcja `CDec`
: Konwertuje dowolne wyrażenie numeryczne (którego wartość należy do przedziału obsługiwanego przez typ danych `Decimal`) na wartość typu `Decimal`.

Funkcja `CInt`
: Konwertuje dowolne wyrażenie numeryczne (którego wartość należy do przedziału obsługiwanego przez typ danych `Integer`) na wartość typu `Integer`, zaokrąglając ewentualną część ułamkową.

Funkcja `CLng`
: Konwertuje dowolne wyrażenie numeryczne (którego wartość należy do przedziału obsługiwanego przez typ danych `Long`) na wartość typu `Long`, zaokrąglając ewentualną część ułamkową.

Funkcja `CObj`
: Konwertuje dowolne wyrażenie na egzemplarz typu `Object`. Takie rozwiązanie jest uzasadnione w sytuacji, gdy chcemy, aby typ wartościowy był traktowany jak typ referencyjny.

Funkcja `CSByte`
: **Nowość w wersji 2005**. Konwertuje dowolne wyrażenie numeryczne (którego wartość należy do przedziału obsługiwanego przez typ danych `SByte`) na wartość typu `SByte`, zaokrąglając ewentualną część ułamkową.

Funkcja `CShort`
: Konwertuje dowolne wyrażenie numeryczne (którego wartość należy do przedziału obsługiwanego przez typ danych `Short`) na wartość typu `Short`, zaokrąglając ewentualną część ułamkową.

Funkcja `CSng`
: Konwertuje dowolne wyrażenie numeryczne (którego wartość należy do przedziału obsługiwanego przez typ danych `Single`) na wartość typu `Single`.

Funkcja `CStr`
: Konwertuje wyrażenie na jego reprezentację łańcuchową. Przykładowo, wartości typu `Boolean` mogą być konwertowane na dwa łańcuchy: `"True"` lub `"False"`. Daty są konwertowane na właściwe łańcuchy zgodnie z formatem zdefiniowanym w ramach ustawień regionalnych danego komputera.

Funkcja `CType`
: Oferuje uniwersalny mechanizm rzutowania, za pomocą którego obiekt lub wyrażenie dowolnego typu można przekształcać w odpowiednik innego typu. Funkcja `CType` może być stosowana dla wszystkich klas, struktur i interfejsów. Co więcej, okazuje się, że funkcja `CType` sprawdza się zarówno w przypadku podstawowych typów danych, jak i typów (w tym klas) definiowanych przez użytkownika. Poniżej przedstawiono składnię tej funkcji:

  ```
  CType(wyrażenie, nazwa-typu)
  ```

  Przykładowo, poniższe wyrażenie:

  ```
  Dim targetNumber As Integer = CType("12", Integer)
  ```

  jest odpowiednikiem następującego wyrażenia:

  ```
  Dim targetNumber As Integer = CInt("12")
  ```

  Nowość w wersji 2005. W wydaniu Visual Basica z 2005 roku dodano mechanizm przeciążania operatorów, który szczegółowo omówimy w rozdziale 5. Jednym z elementów przeciążania operatorów jest możliwość definiowania reguł konwersji funkcji `CType` dla własnych, niestandardowych klas.

Funkcja `CUInteger`
: **Nowość w wersji 2005.** Konwertuje dowolne wyrażenie numeryczne (którego wartość należy do przedziału obsługiwanego przez typ danych `UInteger`) na wartość typu `UInteger`, zaokrąglając ewentualną część ułamkową.

Funkcja `CULong`
: **Nowość w wersji 2005**. Konwertuje dowolne wyrażenie numeryczne (którego wartość należy do przedziału obsługiwanego przez typ danych `ULong`) na wartość typu `ULong`, zaokrąglając ewentualną część ułamkową.

Funkcja `CUShort`

Nowość w wersji 2005. Konwertuje dowolne wyrażenie numeryczne (którego wartość należy do przedziału obsługiwanego przez typ danych `UShort`) na wartość typu `UShort`, zaokrąglając ewentualną część ułamkową.

Zmienne

Zmienną (ang. *variable*) można zdefiniować jako konstrukcję posiadającą następujące własności:

Nazwę

Nazwa zmiennej jest wykorzystywana do jej identyfikowania w kodzie źródłowym. W języku programowania Visual Basic nazwa zmiennej musi się rozpoczynać albo od litery alfabetu (odpowiedniego znaku Unicode), albo od znaku podkreślenia poprzedzającego kolejny znak podkreślenia lub dowolny znak Unicode (literę alfabetu, cyfrę, znak formatujący lub kombinację różnych znaków). Wraz z wprowadzeniem technologii .NET firma Microsoft zaproponowała nowy zbiór standardów nazewnictwa, które należy stosować w przypadku zmiennych i innych nazywanych obiektów. Wspomniane standardy omówiliśmy już w punkcie „Konwencje nazewnicze" w rozdziale 1.

Adres

Każda zmienna ma przypisany adres pamięciowy, czyli miejsce składowania jej wartości. Zmienne stosowane w platformie .NET nie dają nam gwarancji, że reprezentowane dane będą stale składowane pod tymi samymi adresami, zatem same adresy zmiennych nie powinny być rejestrowane ani wykorzystywane.

Typ danych

Typ danych zmiennej określa zakres możliwych wartości, które mogą być przez tę zmienną reprezentowane.

Wartość

Wartość zmiennej jest jej właściwą zawartością składowaną pod odpowiednim adresem pamięciowym. Wartość zmiennej określa się czasem mianem **r-wartości** (ang. *r-value*), ponieważ zawsze występuje po prawej stronie operatora przypisania. Przykładowo, poniższe wyrażenie:

```
Dim targetValue As Integer = 5
```

można odczytać w następujący sposób: „Zapisz **wartość** `5` w pamięci, pod adresem reprezentowanym przez zmienną `targetValue`". Ponieważ nazwy zmiennej `targetValue` użyto na lewo od operatora przypisania, zmienna (lub jej lokalizacja w pamięci) bywa nazywana **l-wartością** (ang. *l-value*).

Zakres

Zakres zmiennej określa miejsca programu, w których dana zmienna będzie widoczna z perspektywy kodu źródłowego. Zakres zmiennych omówimy bardziej szczegółowo w dalszej części tego rozdziału.

Czas życia

Od czasu życia zmiennej zależy to, kiedy i jak długo dana zmienna będzie istniała. Okazuje się, że zmienna może, ale nie musi być widoczna (należeć do bieżącego zakresu) w całym czasie życia. Także czas życia zmiennych zostanie szczegółowo przeanalizowany w dalszej części tego rozdziału.

Deklaracja zmiennej

Deklaracja zmiennej (ang. *variable declaration*) ma na celu związanie jej nazwy z określonym typem danych. W przypadku zmiennych nieobiektowych (typów wartościowych) deklaracja jest równoznaczna z utworzeniem egzemplarza danej zmiennej. Przykładowo, deklaracja w postaci:

```
Dim createMeNow As Integer
```

tworzy zmienną typu `Integer` nazwaną `createMeNow`. Powyższa deklaracja jest równoważna poniższemu wyrażeniu:

```
Dim createMeNow As Integer = New Integer
```

lub nawet wyrażeniu w postaci:

```
Dim createMeNow As New Integer
```

w którym położono nacisk na sam fakt tworzenia nowego egzemplarza obiektu zmiennej.

Za pomocą pojedynczych wyrażeń można deklarować wiele zmiennych. Ogólnie, mimo że typ każdej z deklarowanych zmiennych może być inny, każdorazowe stosowanie nazw typów nie jest bezwzględnie wymagane. Jeśli w deklaracji zmiennej zrezygnujemy z jawnego określenia typu, wówczas zostanie zastosowany typ kolejnej zmiennej, dla której w sposób bezpośredni zadeklarowano właściwy typ. Przykładowo, poniższy wiersz kodu:

```
Dim first As Long, second, third As Integer, fourth As String
```

deklaruje między innymi zmienne `second` i `third` typu `Integer` (w Visual Basicu 6 `second` byłaby zmienną typu `Variant`).

Visual Basic oferuje możliwość inicjalizacji zmiennych w wierszach ich deklarowania. (Przypisywana wartość bywa nazywana **wartością inicjalizującą** — ang. *initializer*). Przykładowo, poniższe wyrażenie:

```
Dim alwaysInitialized As Integer = 5
```

deklaruje i tworzy zmienną typu `Integer`, po czym przypisuje jej wartość początkową równą `5`. Okazuje się, że pojedyncze wyrażenia mogą zawierać także wiele operacji przypisania:

```
Dim first As Integer = 6, second As Integer = 9
```

Warto pamiętać, że w przypadku stosowania wartości inicjalizujących dla każdej ze zmiennych należy wprost określić typ danych.

Zmienne obiektowe (zmienne typów referencyjnych) deklarujemy dokładnie tak samo jak ich odpowiedniki ze świata podstawowych typów danych:

```
Dim newHire As Employee
```

Przedstawiona deklaracja nie tworzy jednak zmiennej obiektowej — po wykonaniu tego wyrażenia zmienna `newHire` będzie miała wartość `Nothing`. Utworzenie obiektu wymaga jawnego wywołania konstruktora odpowiedniej klasy na następujące sposoby:

```
Dim newHire As New Employee
```

lub:

```
Dim newHire As Employee = New Employee
```

lub nawet:

```
Dim newHire As Employee
newHire = New Employee
```

Zakres, czas życia i poziom dostępu do zmiennych

Każda zmienna ma swój **zakres** (ang. *scope*), który określa, w którym miejscu (lub miejscach) programu jej identyfikator jest właściwie rozpoznawany — gdzie w kodzie źródłowym dana zmienna jest **widoczna**.

Zmienne lokalne: zakresy obejmujące blok kodu lub procedurę

Wszystkie zmienne deklarowane wewnątrz funkcji, procedur bądź właściwości nazywamy **zmiennymi lokalnymi** (ang. *local variables*). Tego rodzaju zmienne mogą być wykorzystywane wyłącznie w ramach odpowiednich funkcji, procedur i właściwości — kiedy ich wykonywanie się zakończy, wszelkie zmienne lokalne związane z tymi konstrukcjami przestają istnieć (oczywiście jeśli nie zostały przekazane do innej zmiennej charakteryzującej się szerszym zakresem).

Ogólnie, zmienne lokalne charakteryzują się **zakresem na poziomie procedury** — są dostępne z poziomu wszystkich wierszy kodu wchodzących w skład odpowiedniej procedury. Deklaracje zmiennych lokalnych z reguły występują na samym początku procedur lub funkcji (w kodzie wykonywanym bezpośrednio po wywołaniu tych konstrukcji):

```
Public Sub DoTheWork(  )
   Dim localInt As Integer
   Dim localEmp As New Employee
```

Bloki kodu są zbiorami wyrażeń zawartymi w ramach wyrażeń warunkowych `If`, w pętlach `For`, pętlach `With` i innych podobnych konstrukcjach języka programowania, dla których stosuje się odrębne wyrażenia otwierające i zamykające. Wszystkie wyrażenia zdefiniowane pomiędzy wyrażeniami otwierającymi (`If`, `ElseIf`, `For`, `With` itp.) i zamykającymi (`End If`, `Next`, `End With` itp.) są częścią odpowiednich bloków kodu. Co więcej, zmienna zdefiniowana wewnątrz bloku kodu charakteryzuje się **zakresem na poziomie bloku** i jako taka jest widoczna tylko w ramach tego bloku. Ponieważ bloki kodu mogą być zagnieżdżane, zmienne lokalne definiowane na poziomie tych bloków mogą występować dowolnie głęboko w hierarchii tego rodzaju zagnieżdżeń.

```
Public Sub DoTheWork(ByVal fromWhen As Date, ByVal howMuch As Decimal)
   If (fromWhen < Today) Then
  '----- Ta zmienna jest dostępna tylko w skrajnie zewnętrznym bloku If,
  '     który dodatkowo zawiera skrajnie wewnętrzny blok If. Zmienna nie
  '     jest dostępna poza skrajnie zewnętrznym blokiem If.
  Dim simpleCalculation As Integer

  If (howMuch > 0@) Then
     '----- Ta zmienna jest dostępna tylko w skrajnie wewnętrznym
     '     bloku If.
     Dim complexCalculation As Integer
  End If
  End If
End Sub
```

Zmienne dostępne na poziomie bloków w ogóle nie są widoczne spoza bloków, w których zostały zdefiniowane. Przeanalizujmy teraz następujący blok kodu:

```
If (origValue <> 0) Then
   Dim inverseValue As Decimal
   inverseValue = 1 / origValue
End If
'----- Poniższe wyrażenie zostanie odrzucone przez kompilator.
MsgBox(CStr(inverseValue))
```

W powyższym kodzie zmienna `inverseValue` nie będzie rozpoznawana poza blokiem kodu, w którym została zdefiniowana, zatem ostatni wiersz przedstawionego kodu spowoduje błąd kompilacji.

Wszystkie zmienne lokalne, niezależnie od tego, czy deklarujemy ja na poziomie bloków kodu, czy na poziomie procedur, charakteryzują się czasem życia równym czasowi życia swoich procedur (lub funkcji). Oznacza to, że zmienne deklarowane na poziomie bloków kodu zachowują swoje wartości przez cały czas życia procedury, która ten blok kodu zawiera (także w czasie wykonywania kodu spoza tego bloku). Przykładowo, w poniższym kodzie:

```
Dim counter As Integer
For counter = 1 To 5
   If (ProcessData(counter) = True) Then
      Dim soFar1 As Integer
      Dim soFar2 As Integer = 0
      soFar1 += 1
      soFar2 += 1
      MsgBox("Status na tym etapie: " & soFar1 & ", " & soFar2)
   End If
Next counter
```

zmienna `soFar1` zachowuje swoją wartość z poprzedniego wykonania bloku `If`, w którym została zadeklarowana (z poprzedniej iteracji pętli `For`). W chwili wyświetlania pierwszego komunikatu wspomniana zmienna będzie zawierała wartość `1`, w czasie wyświetlania drugiego komunikatu będzie zawierała wartość `2` itd. Ponieważ zmienną `soFar2` zadeklarowano z wartością inicjalizującą, wartość tej zmiennej będzie ponownie ustawiana (w tym przypadku zerowana) w każdej kolejnej iteracji pętli `For`. Oznacza to, że wszystkie wyświetlane komunikaty będą zawierały wartość `1` na drugiej pozycji.

Procedura może wykorzystywać zmienne przekazywane na jej wejściu za pośrednictwem argumentów. Zakres tego rodzaju zmiennych zawsze ogranicza się do poziomu danej procedury.

Czas życia zmiennych lokalnych może wykraczać poza czas wykonywania procedury, w której te zmienne zadeklarowano. Przykładowo, zmienne **statyczne** (mimo lokalnego zakresu) „żyją" przez cały czas istnienia klasy lub modułu, do którego należą. Tego rodzaju zmienne należy deklarować ze słowem kluczowym `Static` (zamiast ze standardowym słowem kluczowym `Dim`):

```
Static longLasting As Integer = 0
```

Wartość inicjalizująca zmiennej statycznej jest stosowana w momencie tworzenia egzemplarza odpowiedniej klasy lub modułu (nie każdorazowo w chwili napotkania odpowiedniej deklaracji). Kiedy wchodzimy do procedury zawierającej jakąś zmienną statyczną, wspomniana zmienna będzie zawierała dokładnie tę samą wartość, którą zawierała podczas poprzedniego wykonywania danej procedury. Zmienne statyczne nie mogą być stosowane w procedurach składowych struktur.

Zakres i poziomy dostępu w ramach modułów

Wszystkie zmienne deklarowane wewnątrz klas (a także struktur i modułów), ale poza jakimikolwiek procedurami składowymi, charakteryzują się **zakresem na poziomie typu** i jako takie są dostępne z poziomu wszystkich procedur i funkcji wchodzących w skład tej samej klasy (lub struktury bądź modułu). Warto jednak pamiętać o możliwości prostego rozszerzania tego zakresu (poza poziom typu) za pomocą **modyfikatorów dostępu** (ang. *access modifier*).

Dla każdej zmiennej deklarowanej na poziomie typu należy stosować słowo kluczowe modyfikatora dostępu. (Równie dobrze można się posługiwać słowem kluczowym `Dim`, jednak wówczas poziom dostępu będzie uzależniony od typów modułów, co może uczynić nasz kod nieczytelnym). Modyfikatory dostępu przydzielają dostęp na poszczególnych poziomach stopniowo — modyfikator `Poblic` jest w tym względzie najmniej restrykcyjny (patrz tabela 4.1).

Tabela 4.1. Modyfikatory dostępu

Modyfikator dostępu	Opis
`Public`	Zmienne publiczne są dostępne z poziomu dowolnego kodu, który ma dostęp do egzemplarza klasy bądź struktury lub do typu zawierającego te zmienne. Jeśli jakaś klasa zawiera zmienną publiczną i jeśli egzemplarz tej klasy jest dostępny z poziomu odrębnego projektu, aplikacji lub komponentu, także ta zmienna będzie dostępna z poziomu kodu klienckiego.
`Protected`	Zmienne chronione są dostępne zarówno w ramach klasy, w której zostały zadeklarowane, jak i z poziomu wszelkich klas potomnych względem tej klasy (ale nie spoza łańcucha klas związanych relacją dziedziczenia). Zmienne chronione mogą być deklarowane wyłącznie w klasach (nie można ich stosować w strukturach ani modułach).
`Friend`	Zmienne zaprzyjaźnione są dostępne z dowolnego miejsca w tym samym podzespole, ale nigdy spoza podzespołu — egzemplarze klasy zawierającej zmienne zaprzyjaźnione ukrywają te zmienne przed kodem zewnętrznym nawet wtedy, gdy same są przez ten kod wykorzystywane. Zmienne zaprzyjaźnione mogą być stosowane w klasach, strukturach i modułach.
`Protected Friend`	Stosowanie kombinacji słów kluczowych `Protected` i `Friend` powoduje, że tak deklarowane zmienne mają cechy zarówno zmiennych chronionych, jak i zmiennych zaprzyjaźnionych — są dostępne w ramach klasy i klas potomnych oraz w ramach tego samego podzespołu (ale nigdy poza tym podzespołem). Chronione zmienne zaprzyjaźnione mogą być stosowane wyłącznie w klasach (nie można ich stosować w strukturach ani modułach).
`Private`	Zmienne prywatne są dostępne z dowolnego punktu tej samej klasy, struktury lub modułu, ale nigdy poza swoją macierzystą konstrukcją. Zmienne prywatne są ukrywane nawet przed składowymi klas potomnych.

Czas życia zmiennych deklarowanych na poziomie typów obejmuje cały czas życia egzemplarza klasy, struktury lub modułu, do którego należą. Zmienne można deklarować jako dzielone (ze słowem kluczowym `Shared`) i — tym samym — umożliwić istnienie odpowiednich wartości bez konieczności tworzenia egzemplarza klasy, struktury lub modułu. Warto jednak pamiętać, że czas życia zmiennych dzielonych rozciąga się na cały okres funkcjonowania aplikacji. Wszystkie zmienne deklarowane w ciele modułu domyślnie są składowymi dzielonymi.

Stałe

Stałe (ang. *constants*) są w istocie zmiennymi dostępnymi tylko do odczytu. Raz ustawionej w kodzie programu wartości zmiennej (w czasie kompilacji) nie można zmieniać. Stałe są definiowane na poziomie lokalnym lub poziomie modułu za pomocą słowa kluczowego `Const`:

```
modyfikatorDostępu Const nazwa As typ = wartość
```

gdzie *`modyfikatorDostępu`* jest jednym z opisanych w poprzednim podrozdziale modyfikatorów dostępu. (Modyfikatory dostępu nie są stosowane w przypadku stałych deklarowanych wewnątrz procedur). Kiedy opcja `Option Strinct` jest włączona (ma wartość `On`), wszystkie stałe muszą być deklarowane z konkretnym typem.

Typy wyliczeniowe

Typ wyliczeniowy (ang. *enumeration*) jest grupą wzajemnie powiązanych stałych całkowitoliczbowych. Wszystkie składowe wyliczenia muszą być nie tylko egzemplarzami tego samego typu danych, ale też typu całkowitoliczbowego (`Byte`, `Integer`, `Long` lub `Short`, a w wersji 2005 lub nowszej również `SByte`, `UInteger`, `ULong` lub `UShort`). Składowe typu wyliczeniowego są elementami dzielonymi i dostępnymi tylko do odczytu przez cały czas życia aplikacji.

```
Public Enum VehicleType As Integer
   bicycle = 2
   tricycle = 3
   passengerCar = 4
   eighteenWheeler = 18
End Enum
```

Składowe typu wyliczeniowego można stosować w kodzie dokładnie tak jak stałe i zmienne:

```
Dim whatIDrive As VehicleType
whatIDrive = VehicleType.passengerCar
```

Typy wyliczeniowe mogą być deklarowane wyłącznie na poziomie przestrzeni nazw lub modułu (w żadnym razie nie można ich deklarować wewnątrz procedur).

Tablice

Tablica w wielu językach programowania (włącznie z językiem Visual Basic) jest podstawową strukturą danych. Tablice umożliwiają składowanie kolekcji egzemplarzy podobnych typów danych lub obiektów. Każdy **element** znajduje się na pozycji oznaczonej unikatowym numerem. W Visual Basicu indeksy tablic rozpoczynają się od zera (zero jest **dolną granicą**) i kończą na zdefiniowanej **górnej granicy**.

Poniższe przykłady ilustrują rozmaite sposoby deklarowania tablic jednowymiarowych:

```
' Konstruktor niejawny: Brak rozmiaru początkowego i wartości inicjalizujących.
Dim days() As Integer

' Konstruktor jawny: Brak rozmiaru początkowego i wartości inicjalizujących.
Dim days() As Integer = New Integer(  ) {}

' Konstruktor niejawny: Obecność rozmiaru początkowego, brak wartości inicjalizujących.
Dim days(6) As Integer

' Konstruktor jawny: Obecność rozmiaru początkowego, brak wartości inicjalizujących.
Dim days() As Integer = New Integer(6) {}

' Konstruktor niejawny: Rozmiar początkowy wynikający z obecności wartości inicjalizujących.
Dim days() As Integer = {1, 2, 3, 4, 5, 6, 7}

' Konstruktor jawny: Rozmiar początkowy i wartości inicjalizujące.
Dim days() As Integer = New Integer(6) {1, 2, 3, 4, 5, 6, 7}
```

Deklaracja tablicy może:

- wywołać (w sposób jawny lub niejawny) konstruktor tablicy;
- określić początkowy rozmiar wszystkich wymiarów tablicy lub pozostawić wszystkie rozmiary nieokreślone;
- zainicjalizować elementy tablicy lub pozostawić ich wartości nieokreślone.

W Visual Basicu 6 programista mógł określać zarówno dolne, jak i górne granice dla wszystkich wymiarów tablicy. W języku Visual Basic .NET dolną granicą (najniższym indeksem) wszystkich tablic jest zero. Przykładowo, wyrażenie w postaci:

```
Dim myArray(5) As Integer
```

deklaruje tablicę złożoną z sześciu elementów, gdzie indeks pierwszego będzie równy `0`, a indeks ostatniego będzie równy `5`.

Tablice mogą się składać z wielu **wymiarów** (ang. *dimensions*). Przykładowo, poniższy wiersz deklaruje i inicjalizuje tablicę dwuwymiarową:

```
Dim rectArray(,) As Integer = {{1, 2, 3}, {4, 5, 6}}
```

Poniższy fragment kodu wyświetla zawartość tej tablicy:

```
Debug.Write(rectArray(0, 0))
Debug.Write(rectArray(0, 1))
Debug.WriteLine(rectArray(0, 2))
Debug.Write(rectArray(1, 0))
Debug.Write(rectArray(1, 1))
Debug.WriteLine(rectArray(1, 2))

'----- Poniżej przedstawiono dane wyjściowe:
123
456
```

Górną granicę każdego z wymiarów tablicy można zmodyfikować za pomocą wyrażenia `ReDim`, którego ogólną składnię przedstawiono poniżej:

```
ReDim [Preserve] nazwaTablicy(nowaGórnaGranica)
```

Kwalifikator `Preserve` powoduje, że wszystkie wartości, które istnieją w momencie wykonywania wyrażenia `ReDim`, zostaną zachowane w tablicy; w razie braku tego kwalifikatora cała tablica zostanie wyczyszczona. Jeśli zdecydujemy się użyć tego kwalifikatora, powinniśmy pamiętać, że modyfikacja górnej granicy może dotyczyć tylko ostatniego wymiaru tablicy. Liczba samych wymiarów tablicy nigdy nie może być zmieniana.

Dolną i górną granicę danego wymiaru tablicy można określić odpowiednio za pomocą funkcji `LBound` oraz `UBound`:

```
Dim smallArray(5) As Integer
MsgBox(UBound(smallArray))  'Wyświetla liczbę 5.
```

Ponieważ dolną granicą wszystkich wymiarów tablicy jest indeks zerowy, funkcja `LBound` zawsze będzie zwracała właśnie zero.

Kolekcje

Visual Basic oferuje programiście obiekt tablicy asocjacyjnej nazwany **kolekcją** (ang. *collection*). Mimo wielu istotnych podobieństw do standardowych tablic, włącznie z występowaniem elementów w określonym porządku, warto pamiętać o tym, co odróżnia kolekcję od tablic: jej elementy mają postać par klucz-wartość. Elementy kolekcji można identyfikować zarówno według pozycji, jak i według klucza, lub stosując technikę iteracyjnego przeszukiwania kolejnych pozycji.

Szczególnie przydatna w codziennej pracy z kolekcjami okazuje się piątka składowych klasy `Collection`:

Metoda `Add`
: Dodaje element do kolekcji. Poza samymi danymi można określić opcjonalny klucz, za pośrednictwem którego będzie się można odwoływać do nowego elementu.

Metoda `Clear`
: Usuwa wszystkie elementy z kolekcji.

Właściwość `Count`
: Zwraca liczbę elementów składowanych w kolekcji.

Właściwość `Item`
: Zwraca pojedynczy element składowany w kolekcji i identyfikowany albo według indeksu (pozycji w ramach kolekcji), albo według klucza (jeśli został określony w chwili dodawania tego elementu do kolekcji).

Metoda `Remove`
: Usuwa z kolekcji element na podstawie przekazanego indeksu bądź klucza.

Poniższy fragment kodu definiuje kolekcję reprezentującą nazwy stanów identyfikowane za pomocą popularnych skrótów (w tym przypadku skróty występują w roli kluczy):

```
Dim states As New Collection
states.Add("New York", "NY")
states.Add("Michigan", "MI")
```

Elementy tej kolekcji można przeszukiwać iteracyjnie za pomocą prostej konstrukcji `For Each...Next`:

```
Dim oneState As String
For Each oneState In states
   MsgBox(oneState)
Next oneState
```

Podobnie jak w przypadku tablic, dostęp do elementów składowych kolekcji można uzyskiwać za pośrednictwem odpowiednich wartości indeksu. Dolna granica indeksu kolekcji zawsze wynosi jeden.

Nowość w wersji 2005. W wersji języka Visual Basic wydanej w roku 2005 wprowadzono typy uniwersalne, które umożliwiają wiązanie egzemplarzy kolekcji (i innych klas) z konkretnymi typami danych w czasie wykonywania programu. Więcej informacji na ten temat można znaleźć w rozdziale 10.

Parametry i argumenty

Chociaż procedury są autonomicznymi blokami kodu, ich prawidłowe funkcjonowanie często wymaga interakcji z danymi spoza samych procedur. Dane zewnętrzne można przekazywać do procedur za pośrednictwem tzw. **listy parametrów**. Listę parametrów definiujemy w wierszu deklaracji procedury (w nawiasie, bezpośrednio za nazwą samej procedury):

```
Public Function RepeatString(ByVal origText As String, _
      ByVal howManyTimes As Integer) As String
   ' ----- Zwraca łańcuch wielokrotnie skonkatenowany z samym sobą.
   Dim counter As Integer
   RepeatString = ""
   For counter = 1 To howManyTimes
      RepeatString &= origText
   Next counter
End Function
```

Funkcja `RepeatString` otrzymuje na wejściu dwa parametry: `origText` oraz `howManyTimes`. Dla każdego parametru określono zarówno typ danych, jak i metodę przekazywania. Metoda przekazywania parametru to albo `ByVal` (przez wartość), albo `ByRef` (przez referencję). W technologii .NET domyślnym sposobem przekazywania parametrów jest przekazywanie przez wartość.

Podczas wywoływania procedury pobierającej parametry przekazujemy na jej wejściu wartości, które na poziomie kodu wywołującego z reguły nazywa się **argumentami**. Przykładowo, poniższe wyrażenie dołącza dwa argumenty do wywołania funkcji `RepeatString`: łańcuch `"abc"` oraz liczbę całkowitą `5`:

```
targetString = RepeatString("abc", 5)
```

Ponieważ w technologii .NET klasy mogą zawierać metody przeciążone, argumenty przekazywane przez nas na wejściu procedury muszą odpowiadać sygnaturze dokładnie jednej z przeciążonych wersji wywoływanej metody. Bardziej szczegółowe omówienie techniki przeciążania można znaleźć w rozdziale 3.

Przekazywanie argumentów

Wszystkie argumenty są przekazywane albo przez wartość, albo przez referencję (w zależności od tego, czy przed nazwą odpowiedniego parametru użyjemy słowa kluczowego `ByVal`, czy `ByRef`). Kiedy dane są przekazywane przez wartość, docelowa procedura otrzymuje kopię wyrażenia lub zmiennej źródłowej. W samej procedurze parametr jest traktowany dokładnie tak jak zmienna lokalna — może być dowolnie wykorzystywany i modyfikowany, ale przestaje istnieć w chwili zakończenia wykonywania procedury. Żadna ze zmian wartości parametrów przekazanych przez wartość (zadeklarowanych ze słowem kluczowym `ByVal`) w ramach procedury nie będzie miała wpływu na wartość zmiennej źródłowej. Opisana zasada jest szczególnie łatwa do weryfikacji w przypadku typów wartościowych. Przykładowo, przeanalizujmy teraz następujący fragment kodu:

```
Public Sub ParentRoutine()
   Dim sourceValue As Integer = 5
   ChildRoutine(sourceValue)
   MsgBox(sourceValue)   '--> Wyświetla 5
End Sub

Public Sub ChildRoutine(ByVal incoming As Integer)
   incoming = 10
End Sub
```

Chociaż wartość zmiennej `sourceValue` przekazanej do procedury `ChildRoutine` (za pośrednictwem parametru `incoming`) została przez tę procedurę zmodyfikowana, odpowiednia zmiana nie została propagowana do kodu procedury wywołującej `ParentRoutine`, ponieważ parametr `incoming` reprezentował kopię wartości zmiennej `sourceValue`.

Z drugiej strony, obiekty (egzemplarze typów referencyjnych) przekazywane do procedur przez wartość mogą być przez te procedury modyfikowane. Precyzyjnie mówiąc, na poziomie procedury można modyfikować **składowe** przekazanego obiektu, nie sam obiekt. W przypadku obiektów przekazywanych przez wartość w praktyce następuje przekazanie adresu pamięciowego, stąd zmiany dokonywane przez procedurę na danym obszarze pamięci znajdują odzwierciedlenie w oryginalnym obiekcie. Warto jednak pamiętać, że obiektu przekazanego przez wartość nie można na poziomie procedury zastąpić innym egzemplarzem tej samej klasy (nasze możliwości obejmują wyłącznie modyfikowanie składowych):

```
Public Class DataClass
   Public DataMember As Integer
End Class

Public Class CodeClass
   Public Sub ParentRoutine()
      Dim sourceValue As New DataClass
      sourceValue.DataMember = 5
      ChildRoutine(sourceValue)
      MsgBox(sourceValue.DataMember)    '--> Wyświetla 10
   End Sub

   Public Sub ChildRoutine(ByVal incoming As DataClass)
      '----- Poniższy wiersz zmienia „rzeczywistą" składową.
      incoming.DataMember = 10

      '----- Poniższe wiersze nie mają wpływu na zmienną sourceValue.
      incoming = New DataClass
      incoming.DataMember = 15
   End Sub
End Class
```

Przekazywanie przez referencję (za pośrednictwem parametru zadeklarowanego ze słowem kluczowym `ByRef`) argumentu typu wartościowego na wejściu procedury zawsze oznacza przekazanie adresu pamięciowego odpowiedniej wartości — oznacza to, że ewentualne zmiany wprowadzone przez procedurę docelową będą miały bezpośredni wpływ na wartość odpowiedniej zmiennej źródłowej. (Ten mechanizm jest stosowany pod warunkiem, że wartość źródłowa jest składowana w zmiennej; stałe i wyrażenia obliczane w samym wywołaniu z natury rzeczy nie mogą być modyfikowane). Przykładowo, spróbujmy porównać poniższy kod ze słowem kluczowym `ByRef` z przedstawionym przed momentem odpowiednikiem zawierającym parametr przekazywany przez wartość:

```
Public Sub ParentRoutine()
   Dim sourceValue As Integer = 5
   ChildRoutine(sourceValue)
   MsgBox(sourceValue)   ' --> Wyświetla 10
End Sub

Public Sub ChildRoutine(ByRef incoming As Integer)
   incoming = 10
End Sub
```

Jak widać, zastąpienie słowa kluczowego `ByVal` słowem `ByRef` miało istotny wpływ na funkcjonowanie tego fragmentu kodu. W przypadku typów referencyjnych różnica nie jest tak widoczna, chyba że spróbujemy w procedurze docelowej całkowicie zastąpić oryginalny obiekt. Przekazywanie przez referencję daje taką możliwość! Wynika to z faktu, że słowo kluczowe `ByRef` wymusza przekazywanie do procedury **adresu adresu pamięciowego**. Modyfikując ten adres pamięciowy, możemy w prosty sposób zastąpić adres reprezentowany przez zmienną źródłową. W niektórych językach programowania opisywany schemat jest nazywany **podwójnym wskaźnikiem** (ang. *double pointer*). Wszelkie wątpliwości związane z tym mechanizmem powinien wyjaśnić poniższy przykład. Warto porównać ten fragment kodu z podobnym fragmentem, w którym zastosowano słowo kluczowe `ByVal`:

```
Public Class DataClass
   Public DataMember As Integer
End Class

Public Class CodeClass
   Public Sub ParentRoutine(  )
```

```
        Dim sourceValue As New DataClass
        sourceValue.DataMember = 5
        ChildRoutine(sourceValue)
        MsgBox(sourceValue.DataMember)    '--> Wyświetla 15
    End Sub

    Public Sub ChildRoutine(ByRef incoming As DataClass)
        '----- Poniższy wiersz zmienia „rzeczywistą" składową.
        incoming.DataMember = 10

        '----- Poniższe wiersze całkowicie zastępują egzemplarz wskazywany
        '      przez zmienną obiektową sourceValue.
        incoming = New DataClass
        incoming.DataMember = 15
    End Sub
End Class
```

Stosowanie parametrów przekazywanych przez referencję łącznie z referencyjnymi typami danych umożliwia procedurom docelowym zastępowanie oryginalnych obiektów zupełnie nowymi egzemplarzami właściwych klas.

Argumenty opcjonalne

Język Visual Basic obsługuje parametry opcjonalne za pośrednictwem słowa kluczowego `Optional`:

```
Sub Calculate(Optional ByVal silent As Boolean = False)
```

W przypadku wszystkich parametrów opcjonalnych ma zastosowanie następująca para reguł:

- Każda deklaracja argumentu opcjonalnego musi obejmować wartość domyślną, która dodatkowo musi mieć postać wyrażenia stałego (w żadnym razie nie może być zmienną). Wartość domyślna jest wykorzystywana tylko wtedy, gdy kod wywołujący nie dostarcza argumentu dla parametru opcjonalnego.
- Każdy parametr zadeklarowany po parametrze opcjonalnym także musi być opcjonalny. Oznacza to, że ewentualne parametry obowiązkowe (nieopcjonalne) muszą się znajdować na liście parametrów przed parametrami opcjonalnymi.

W wersjach Visual Basica sprzed wprowadzenia technologii .NET istniała możliwość rezygnacji z wartości domyślnej i — jeśli typem parametru był `Variant` — stosowania funkcji `IsMissing` do określania, czy odpowiednia wartość została przekazana. Opisane rozwiązanie nie jest obsługiwane w platformie .NET, zatem dla wszystkich parametrów opcjonalnych koniecznie musimy definiować wartości domyślne (na wypadek braku odpowiednich argumentów).

Tablice parametrów

W normalnych warunkach definicja procedury określa stałą liczbę parametrów. Programiści Visual Basica mają jednak do dyspozycji słowo kluczowe `ParamArray` (od ang. *parameter array* — tablica parametrów), za pośrednictwem którego można rozszerzać listę parametrów poza stały zbiór elementów. Każde wywołanie tak zdefiniowanej procedury może zawierać inną liczbę parametrów (oczywiście oprócz puli parametrów wymaganych).

Przeanalizujmy przykład funkcji wyznaczającej średnią ocenę z egzaminów, których liczba nie jest znana (liczba ocen wejściowych w każdym wywołaniu może być inna):

```
Public Function AverageScore(ByVal ParamArray scores(  ) _
      As Single) As Single
   '----- Oblicza średnią ocenę dla dowolnej liczby egzaminów.
   Dim counter As Integer

   AverageScore = 0
   For counter = 0 To UBound(scores)
      AverageScore += scores(counter)
   Next counter
   AverageScore /= UBound(scores) + 1
End Function
```

Wywołania funkcji `AverageScore` mogą teraz zawierać zróżnicowaną liczbę argumentów:

```
MsgBox(AverageScore(1, 2, 3, 4, 5))  '--> Wyświetla 3
MsgBox(AverageScore(1, 2, 3))        '--> Wyświetla 2
```

Poniżej przedstawiono reguły, które mają zastosowanie w przypadku parametrów deklarowanych ze słowem kluczowym `ParamArray`:

- Procedura może otrzymywać na wejściu tylko jedną tablicę parametrów i ta tablica musi być ostatnim elementem na liście parametrów.
- Tablica parametrów musi być przekazywana przez wartość, a jej deklaracja na liście parametrów procedury zawsze musi być opatrywana słowem kluczowym `ByVal`.
- Tablica parametrów musi być strukturą jednowymiarową. Jeśli nie zadeklarujemy jej typu, kompilator automatycznie założy, że tablica składa się z elementów typu `System.Object`.

Tablica parametrów zawsze jest parametrem opcjonalnym. Jej domyślną wartością jest pusta tablica jednowymiarowa właściwego typu danych.