VB .NET. Almanach

Autorzy: Steve Roman, Ron Petrusha, Paul Lomax
Tłumaczenie: Dorota Bednarz,
Krzysztof Jurczyk, Dariusz Małyszko
ISBN: 83-7197-730-1
Tytuł oryginału: VB .NET Language In A Nutshell
Format: B5, stron: 754

W dziesięć lat po powstaniu języka Visual Basic firma Microsoft wprowadza na rynek platformę .NET z całkowicie poprawioną i przebudowaną wersją tego języka, opatrzoną nazwą Visual Basic .NET. Zdaniem niektórych jest to całkiem nowy język programowania. Visual Basic jest teraz w pełnym tego słowa znaczeniu językiem zorientowanym obiektowo – z długo oczekiwanym dziedziczeniem klas i innymi elementami charakteryzującymi programowanie obiektowe.

W większości książek poświęconych Visual Basicowi zakłada się, że czytelnik jest całkowitym nowicjuszem w dziedzinie programowania i dlatego są one w dużej części poświęcone wprowadzeniu go w takie pojęcia, jak zmienne, łańcuchy i instrukcje. Niniejszy almanach jest zupełnie innym rodzajem książki. Stanowi szczegółowe, profesjonalne źródło informacji o języku VB .NET, do którego można się odwołać, by odświeżyć informacje na temat konkretnego elementu języka czy parametru. Książka będzie doskonałą pomocą podczas programowania, kiedy zaistnieje potrzeba przejrzenia reguł dotyczących stosowania konkretnego elementu składowego języka lub wtedy, gdy należy sprawdzić czy nie przeoczono jakiegoś istotnego szczegółu związanego z konkretnym elementem języka.

W książce VB .NET. Almanach omówiono m.in.:

- Podstawowe typy danych języka Visual Basic oraz sposób ich wykorzystania, a także typy danych .NET
- Programowanie obiektowe w VB .NET
- Nowe elementy składowe .NET Framework, mające największy wpływ na sposób programowania w VB .NET, bibliotekę klas .NET Framework
- Delegacje, zdarzenia i obsługę błędów w VB .NET
- Wszystkie funkcje, instrukcje, dyrektywy, obiekty i elementy składowe obiektów tworzące język VB .NET
- Pułapki czyhające na programistę VB .NET i wiele przydatnych „tricków” programistycznych

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

Spis treści

3

Wprowadzenie do programowania obiektowego

Ten rozdział jest krótkim i zwięzłym wprowadzeniem do programowania obiektowego. Ponieważ nie jest to książka o programowaniu obiektowym, skupimy się na tych zagadnieniach, które są ważne podczas programowania w VB .NET.

Dlaczego programowanie obiektowe?

Począwszy od wersji 4 Visual Basic umożliwia stosowanie szeregu technik programowania obiektowego. Jednak niejednokrotnie prezentowano pogląd, że dotychczasowe wersje języka Visual Basic nie są „prawdziwym" obiektowym językiem. Dopiero w VB .NET zmiany wprowadzone w dziedzinie obiektowości są naprawdę zauważalne. Bez względu na prezentowane w tej kwestii stanowisko wydaje się oczywiste, że VB .NET jest obiektowym językiem programowania w pełnym tego słowa znaczeniu.

Można w tym miejscu powiedzieć: „Nie chcę używać technik programowania obiektowego w moich programach.". W przypadku VB 6 było to jeszcze możliwe. Jednak w VB .NET struktura .NET Framework — szczególnie biblioteka klas `Base Class Library` — jak również dokumentacja jest całkowicie zorientowana obiektowo. Z tego powodu nie można dłużej unikać możliwości poznania podstaw programowania obiektowego nawet wtedy, gdy zdecydujemy się nie używać tych technik we własnych aplikacjach.

Podstawy programowania obiektowego

W literaturze często podaje się, że u podstaw programowania obiektowego leżą cztery główne pojęcia:

- hermetyzacja;
- abstrakcja;
- dziedziczenie;
- polimorfizm.

Każde z powyższych pojęć w charakterystyczny dla siebie sposób odgrywa znaczącą rolę w programowaniu w VB .NET. Hermetyzacja oraz abstrakcja są „teoretycznymi" pojęciami stanowiącymi podstawę programowania obiektowego. Dziedziczenie i polimorfizm stanowią pojęcia bezpośrednio stosowane podczas programowania w VB .NET.

Abstrakcja

Pojęcie *abstrakcja* oznacza po prostu przedstawienie danego elementu — encji — zawierające jedynie te jego aspekty, które są ważne w konkretnej sytuacji. Przypuśćmy, że chcemy utworzyć komponent odpowiedzialny za przechowywanie informacji o pracownikach przedsiębiorstwa. W tym celu rozpoczniemy od utworzenia listy pozycji istotnych dla naszej encji (pracownika przedsiębiorstwa). Niektóre z tych pozycji to:

- imię i nazwisko;
- adres;
- numer identyfikacyjny pracownika;
- pobory;
- zwiększenie poborów;
- zmniejszenie poborów.

Ważne jest, że dołączyliśmy nie tylko *właściwości* encji (tj. pracowników), takie jak imię i nazwisko, ale również *akcje*, które możemy wykonać na tych encjach, jak na przykład zwiększenie lub zmniejszenie poborów. Wymienione czynności lub działania nazywane są również metodami, operacjami lub zachowaniami. W tej książce będziemy używać terminu *metody*, który jest powszechnie stosowany w VB .NET.

Oczywiście nie będziemy tworzyć właściwości IQ określającej iloraz inteligencji pracownika, ponieważ nie jest to stosowne (by nie wspomnieć o dyskryminacyjnym charakterze takiego postępowania). Nie będziemy również dołączać właściwości „kolor włosów." Wprawdzie ta cecha wchodzi w skład encji, jednak nie jest w tym przypadku istotna.

Podsumowując — utworzyliśmy *abstrakcyjne* pojęcie pracownika, które zawiera jedynie te właściwości i metody, które nas interesują. Po utworzeniu takiego abstrakcyjnego modelu można przystąpić do *hermetyzacji* jego właściwości i metod w konkretnym komponencie.

Hermetyzacja

Hermetyzacja oznacza zawieranie właściwości i metod danego abstrakcyjnego modelu i *udostępnianie* na zewnątrz jedynie tych z nich, które są absolutnie konieczne. Każda właściwość i metoda modelu abstrakcyjnego nazywana jest jego *elementem*. Zbiór *udostępnianych* na zewnątrz elementów składowych modelu abstrakcyjnego (lub komponentu zawierającego model abstrakcyjny) określa się zbiorowym terminem *interfejsu publicznego* (lub po prostu *interfejsu*).

Hermetyzacja spełnia trzy główne zadania:

- umożliwia ochronę właściwości i metod przed dostępem z zewnątrz;
- umożliwia kontrolę poprawności wprowadzanych danych w interfejsie publicznym (na przykład sprawdzenie tego, czy nie zostaje przypisana zarobkom pracownika ujemna liczba);
- zwalnia użytkownika od konieczności wnikania w szczegóły implementacyjne właściwości i metod.

Weźmy jako przykład typ danych `Integer`, do którego dostęp jest starannie kontrolowany przez VB. Z pewnością wiadomo, że liczba całkowita jest przechowywana w pamięci komputera w postaci *binarnego ciągu* zer i jedynek. W VB liczby całkowite reprezentowane są w formie *dopełnienia do dwu*, dzięki czemu możliwe jest przechowywanie liczb ujemnych i dodatnich.

W celu uproszczenia rozważań weźmy pod uwagę 8-bitowe liczby całkowite. Liczba całkowita 8-bitowa ma postać $a_7a_6a_5a_4a_3a_2a_1a_0$, gdzie każdy element a o danym indeksie jest zerem lub jedynką. Można podać następującą reprezentację graficzną takiej liczby:

Rysunek 3.1. 8-bitowa liczba binarna

W liczbie, która zostanie zapisana w postaci dopełnienia do dwu, bit znajdujący się najbardziej na lewo — a_7 (nazywany również najbardziej znaczącym bitem) — jest *bitem znaku*. Jeżeli bit znaku zawiera wartość 1, to liczba jest ujemna. W przeciwnym razie, gdy bit znaku zawiera 0, liczba jest dodatnia.

Przy zamianie liczby $a_7a_6a_5a_4a_3a_2a_1a_0$ zapisanej w postaci dopełnienia do dwu na postać dziesiętną stosowany jest następujący wzór:

$$\text{postać dziesiętna} = -128a_7 + 64a_6 + 32a_5 + 16a_4 + 8a_3 + 4a_2 + 2a_1 + a_0$$

Utworzenie liczby o przeciwnym znaku do danej liczby, która jest zapisana w postaci dopełnienia do dwu, polega na zmianie wartości każdego bitu na wartość przeciwną (tzn. każde 0 zamieniamy na 1, a każde 1 na 0), po czym do powstałej liczby dodaje się 1.

W tym miejscu można powiedzieć: „Jako programista nie muszę zaprzątać sobie głowy takimi szczegółami. Wystarczy, że napiszę:

```
x = -16
y = -x
```

a kompilator niech wybierze odpowiednią reprezentację liczby i wykona wymagane operacje.”.

Właśnie o to chodzi w hermetyzacji. Szczegóły interpretacji przez komputer (i kompilator) liczb całkowitych ze znakiem oraz implementacja ich właściwości i operacji na nich wykonywanych są hermetyzowane, czyli zamknięte w samym typie całkowitoliczbowym. W ten sposób powyższe informacje są przed użytkownikami tego typu ukryte. Mamy dostęp jedynie do tych właściwości i operacji, które są potrzebne do posługiwania się liczbami całkowitymi. Udostępniane na zewnątrz właściwości i metody tworzą publiczny interfejs dla typu `Integer`.

Ponadto hermetyzacja chroni przed popełnianiem błędów. Powróćmy jeszcze na moment do przedstawionego powyżej przykładu — jeżeli musielibyśmy sami zmienić znak liczby tworząc dopełnienia do dwu i na końcu dodając 1, moglibyśmy zapomnieć wykonać którąś z tych operacji. Hermetyzowany typ danych sam sprawuje automatycznie nad tym kontrolę.

Hermetyzacja posiada jeszcze jedną ważną cechę. Kod napisany z wykorzystaniem udostępnianego na zewnątrz interfejsu pozostaje aktualny — nawet po zmianie wewnętrznych mechanizmów implementacji typu `Integer` — tak długo, jak długo ten interfejs nie ulega zmianie. Jeżeli przeniesiemy teraz nasz kod do komputera, który przechowuje liczby całkowite w postaci dopełnienia do jednego, wtedy *wewnętrzna* procedura, która implementuje operację zmiany znaku liczby całkowitej, będzie musiała się zmienić. Z punktu widzenia programisty nic się jednak nie zmienia. Poniższy fragment kodu:

```
x = -16
y = -x
```

jest nadal poprawny.

Interfejsy

W VB hermetyzacja realizowana jest poprzez tworzenie komponentów. Można utworzyć komponent hermetyzujący omawiany wcześniej abstrakcyjny modelu pracownika.

W VB .NET realizację metod interfejsu stanowią *funkcje*. Natomiast każda z właściwości, jak zobaczymy w dalszej części tego rozdziału, jest implementowana jako *prywatna* zmienna z dwiema towarzyszącymi publicznymi funkcjami. Zmienna prywatna przechowuje wartość właściwości. Pierwsza z funkcji publicznych służy do pobierania wartości właściwości, podczas gdy druga wykorzystywana jest do jej ustawiania. Wymienione dwie funkcje określa się czasami mianem *metod udostępniających* właściwości. Zbiór udostępnianych na zewnątrz funkcji (zwykłych metod oraz metod udostępniających) tworzy *interfejs* modelu abstrakcyjnego.

Komponent może hermetyzować i udostępniać na zewnątrz więcej niż jeden model abstrakcyjny (a zatem więcej niż jeden interfejs). Bardziej realistycznym przykładem może być komponent wzorujący się na pracownikach przedsiębiorstwa, który posiada interfejs `IIdentification` (pierwsza litera „I" jest skrótem od słowa interfejs) służący do celów identyfikacji. Wspomniany interfejs mógłby mieć następujące właściwości: imię i nazwisko, numer ubezpieczenia, numer prawa jazdy, wiek, znaki szczególne itd. Poza tym interfejsem, komponent mógłby również zawierać interfejs zwany `IEducation` opisujący wykształcenie pracownika. Ten drugi interfejs implementowałby takie właściwości jak: poziom wykształcenia, tytuły, ukończone szkoły itp.

Interfejs każdego modelu abstrakcyjnego udostępnianego przez komponent określa się również jako interfejs *komponentu*. W ten sposób komponent `Employee` (pracownik) implementuje przynajmniej dwa interfejsy: `IIdentification` i `IEducation`. Należy pamiętać, że termin interfejs jest również używany do określenia zbioru *wszystkich* udostępnianych na zewnątrz właściwości i metod komponentu (w tym przypadku komponent ma tylko jeden interfejs).

Wracając do naszego abstrakcyjnego modelu opisującego pracownika — jego interfejs mógłby składać się z funkcji zaprezentowanych w tabeli 3.1. Podany interfejs jest oczywiście bardzo uproszczony, ale w zupełności wystarcza do zilustrowania omawianych pojęć.

Tabela 3.1. Elementy składowe interfejsu Employee (pracownik)

Typ	Nazwa
Właściwość	`FullName: GetFullName(), SetFullName()`
Właściwość	`Address: GetAdress(), SetAddres()`
Właściwość	`EmployeeID: GetEmployeeID(), SetEmployeeID()`
Właściwość	`Salary: GetSalary(), SetSalary()`
Metoda	`IncSalary()`
Metoda	`DecSalary()`

Chociaż używanie terminu *interfejs* w znaczeniu zbioru funkcji jest powszechne, to wiąże się jednak z pewnym problemem. Mianowicie chodzi o to, że podanie *nazw* wszystkich funkcji interfejsu (w sposób przedstawiony w tabeli) nie dostarcza pełnej informacji potrzebnej do wywołania tych funkcji. Bardziej użyteczną definicją interfejsu jest podanie zbioru sygnatur publicznych funkcji komponentu.

W celu wyjaśnienia tego zagadnienia przeanalizujemy jedno z najważniejszych rozgraniczeń w programowaniu obiektowym — rozróżnienie między *deklaracją*, a *implementacją* funkcji.

Utwórzmy następującą funkcję sortującą:

```
Function Sort(a() As Integer, iSize As Integer) As Boolean
  For i = 1 to iSize
    For j = i + 1 to isize
      If a(j) < a(i) Then swap a (i), a(j)
```

```
    Next j
  Next i
  Sort = True
End Function
```

Pierwszy wiersz kodu:

```
Function Sort(a()  As Integer, iSize As Integer) As Boolean
```

jest *deklaracją funkcji*. Deklaracja funkcji zawiera informację o liczbie i typie pobieranych parametrów oraz o typie zwracanej przez funkcję wartości. Treść funkcji:

```
For i = 1 to iSize
  For j = i + 1 to iSize
    If a(j) < a(i) Then swap a (i), a(j)
  Next j
Next i
Sort = True
```

jest *implementacją* funkcji. Opisuje, *jak* funkcja wykonuje postawione przed nią zadania.

Należy zauważyć, że jest możliwa zmiana sposobu implementacji funkcji bez zmiany deklaracji funkcji. W rzeczywistości podana implementacja funkcji sortuje tablicę *a* za pomocą prostego algorytmu sortowania przez selekcję, ale możemy zmienić ten algorytm na dowolnie wybrany, inny algorytm sortowania (sortowanie bąbelkowe, sortowanie przez wstawianie, sortowanie szybkie — *quick sort* itp.)

Teraz przyjrzyjmy się programowi (klientowi), który ma wykorzystać funkcję `Sort`. Program wywołujący musi znać jedynie deklarację funkcji `Sort`, by móc ją wywołać. Nie powinien (a prawdopodobnie nie chce) znać szczegółów implementacyjnych. W ten sposób to deklaracja funkcji — a nie jej implementacja — tworzy interfejs funkcji

Sygnatura funkcji jest nazwą funkcji i typem zwracanej przez nią wartości wraz z nazwami i typami jej parametrów podanymi w prawidłowej kolejności. Deklaracja funkcji jest po prostu przejrzystym sposobem opisania sygnatury funkcji. Zgodnie z konwencją przyjętą przez Microsoft wartość zwracana przez funkcję nie jest częścią sygnatury funkcji. Wedle tej konwencji sygnatura jest tzw. *sygnaturą argumentów*. Przyczyny przyjęcia takiej konwencji staną się bardziej zrozumiałe w dalszej części rozdziału, kiedy przejdziemy do omawiania przeciążania nazw. Jednak byłoby lepiej (jak zwykle), gdyby terminologia stosowana przez Microsoft była bardziej starannie przemyślana.

Według tej specyficznej definicji interfejsu interfejs naszego komponentu `Employee` (pracownik) mógłby wyglądać następująco (przedstawiono jedynie jego część):

```
Function GetFullName(iEmpID As Long) As String
Sub SetFullName(lEmpID As Long, sName As String)
        …
Sub IncSalary(sngPercent As Single)
Sub decSalary(sngPercent As Single)
```

Klasy i obiekty

Klasa jest komponentem definiującym i implementującym jeden lub więcej interfejsów. Ściśle mówiąc — klasa nie musi implementować wszystkich składowych interfejsu (omówione to zostanie podczas opisu składowych abstrakcyjnych klasy). Wyrażając to inaczej można powiedzieć, że klasa łączy dane, funkcje i typy w jeden nowy typ. Microsoft używa pojęcia *typ* również w odniesieniu do klas.

Moduły klas VB .NET

W Visual Studio.NET moduł klas VB wstawiany jest do projektu po wybraniu pozycji *Add Class* z menu *Projekt*. W ten sposób dołączony zostaje nowy moduł zawierający kod:

```
Public Class ClassName

End Class
```

Visual Studio przechowuje każdą klasę w oddzielnym pliku, jednak nie jest to konieczne. Konstrukcja `Class ...End Class` zaznacza początek i koniec definicji klasy. W ten sposób w pojedynczym pliku źródłowym może znajdować się więcej niż jedna klasa, jak też może zostać umieszczonych jeden lub więcej modułów (ograniczonych konstrukcją `Module...End Module`).

Klasa `CPerson` zdefiniowana w następnym podrozdziale jest przykładem wykorzystania modułu klas.

Składowe klasy

W VB .NET moduły klas mogą zawierać przedstawione niżej rodzaje elementów.

Dane
: Znajdują się tutaj zmienne (również określane jako *pola*) oraz stałe.

Zdarzenia
: Zdarzenia są procedurami wywoływanymi automatycznie przez Common Language Runtime w odpowiedzi na niektóre zachodzące akcje (takie jak na przykład tworzenie obiektu, naciśnięcie przycisku, zmiana danych lub wyjście obiektu z zasięgu).

Funkcje i procedury
: Funkcje i procedury nazywane są również *metodami*. Konstruktor klasy jest specjalnym rodzajem metody. Konstruktory są szczegółowo omawiane w dalszej części tego rozdziału.

Właściwości
: Właściwość jest implementowana jako prywatna zmienna wraz z dwiema specjalnymi funkcjami udostępniającymi. Składnia tych specjalnych funkcji jest omówiona w dalszej części tego rozdziału w podrozdziale „Właściwości."

Typy
Składowa klasy może być również inną klasą (w tym przypadku określana jest jako klasa zagnieżdżona).

Poniższa klasa `CPerson` zawiera kilka rodzajów składowych:

```
Public Class CPerson
' -----------------
' Dane
' -----------------
' Zmienne składowe
  Private msName As String
  Private miAge As Integer

' Stała
  Public Const MAXAGE As Short = 120

' Zdarzenie
  Public Event Testing()

' -----------------
' Funkcje
' -----------------

' Metody
  Public Sub Test()
    RaiseEvent Testing ()
  End Sub

  Property Age() As Integer
    Get
      Age = miAge
    End Get
    Set(ByVal Value As Integer)
      ' Kontrola poprawności
      If Value < 0 Then
        MsgBox("Wiek nie może być liczbą ujemną")
      Else
        miAge = Value
      End If
    End Set
  End Property

' Właściwość
  Property Name() As String
    ' Metody udostępniające właściwości
    Get
      Name = msName
    End Get
    Set(ByVal Value As String)
      msName = Value
    End Set
  End Property

  Sub Dispose()
    ' Zwolnienie zasobów
  End Sub
End Class
```

Interfejs publiczny klasy VB .NET

Podczas omawiania pojęć związanych z programowaniem obiektowym stwierdziliśmy, że udostępniane na zewnątrz składowe komponentu tworzą jego *publiczny interfejs* (lub po prostu *interfejs*). Ponadto w VB .NET każda składowa modułu klasy ma określony *typ dostępu*, którym może być `Public`, `Private`, `Friend`, `Protected` lub `Protected Friend`. Typy dostępu zostały szczegółowo omówione w dalszej części tego rozdziału. W tym miejscu wystarczy powiedzieć, że moduł klasy w VB .NET może mieć składowe typu `Public`, `Private`, `Friend`, `Protected` oraz `Protected Friend`.

W ten sposób powstaje pewna dwuznaczność przy definiowaniu pojęcia interfejsu publicznego klasy w VB .NET. Samo pojęcie mogłoby wskazywać, że należy uważać każdą składową udostępnianą na zewnątrz jako część publicznego interfejsu klasy. W tym przypadku pod pojęciem interfejsu publicznego klasy rozumielibyśmy oprócz składowych `Public` również składowe `Protected`, `Friend` oraz `Protected Friend`. Z drugiej strony — można by argumentować, że składowe interfejsu publicznego muszą być udostępniane na zewnątrz projektu zawierającego klasę, do którego omawiany interfejs należy. Przy takim zdefiniowaniu pojęcia interfejsem publicznym są jedynie jego składowe `Public`. Na szczęście nie musimy podawać dokładnej definicji interfejsu publicznego klasy w VB .NET, jeśli pamiętamy, że ten termin może być różnie definiowany.

Obiekty

Klasa przedstawia jedynie opis właściwości oraz metod, a tym samym nie jest jednostką posiadającą samodzielny byt (z wyjątkiem składowych współużytkowanych, które zostaną omówione w dalszej części książki.) W celu wywołania metod i wykorzystania właściwości należy utworzyć *instancję* klasy, oficjalnie nazywaną *obiektem*. Tworzenie instancji klasy określane jest jako *konkretyzacja* lub *ukonkretnienie* klasy.

Istnieją trzy sposoby utworzenia obiektu danej klasy w VB .NET. Jeden z nich polega na zadeklarowaniu zmiennej obiektowej reprezentującej daną klasę:

```
Dim APerson As CPerson
```

a następnie utworzeniu obiektu za pomocą słowa kluczowego `New`:

```
APerson = New CPerson()
```

Możemy obie czynności wykonać w jednym wierszu kodu:

```
Dim APerson As New CPerson()
```

lub:

```
Dim APerson As CPerson = New CPerson()
```

Pierwszy sposób zapisu uważany jest za skróconą formę drugiego zapisu.

Właściwości

Właściwości są składowymi klasy, które mogą być zaimplementowane na dwa różne sposoby. W najprostszej postaci właściwość jest jedynie publiczną zmienną:

```
Public Class CPerson
  Public Age As Integer
End Class
```

W przypadku takiej implementacji właściwości `Age` problemem jest brak hermetyzacji. Dostęp do obiektu klasy `CPerson` daje możliwość ustawienia właściwości `Age` na dowolną wartość typu `Integer` (w tym również na wartość ujemną, która jest nieprawidłową reprezentacją wieku). Nie ma tutaj możliwości kontrolowania poprawności wprowadzanych danych (a ponadto powyższa implementacja właściwości nie umożliwia jej włączenia do interfejsu publicznego klasy w myśl podanej przez nas definicji tego terminu).

„Poprawnym" obiektowym sposobem implementacji właściwości jest utworzenie zmiennej typu `Private` wraz z dwiema funkcjami. Zmienna typu `Private` przechowuje wartość właściwości. Natomiast dwie funkcje składowe, które są zwane *metodami udostępniającymi*, służą do pobierania i ustawiania wartości właściwości. W ten sposób dane są hermetyzowane, a dostęp do właściwości może zostać ograniczony za pomocą funkcji udostępniających, które mogą kontrolować poprawność wprowadzanych danych. Poniższy fragment kodu implementuje właściwość `Age`:

```
Private miAge As Integer
Property Age() As Integer
  Get
    Age = miAge
  End Get
  Set(ByVal Value As Integer)
    ' Kontrola poprawności wprowadzanych danych
    If Value < 0 Then
      MsgBox("Wiek musi być liczbą dodatnią")
    Else
      miAge = Value
    End If
  End Set
End Property
```

Jak wynika z powyższego przykładu, Visual Basic ma specjalną składnię służącą do definiowania metod udostępniających właściwości. Natychmiast po wprowadzeniu w edytorze Visual Studio .NET poniższego wiersza kodu:

```
Property Age() As Integer
```

tworzony jest przez IDE następujący wzorzec:

```
Property Age() As Integer
  Get

  End Get
  Set(ByVal Value As Integer)

  End Set
End Property
```

Należy zauważyć, że parametr `Value` umożliwia dostęp do wartości, która ma zostać wprowadzona. W poniższym fragmencie kodu:

```
Dim cp As New CPerson()
cp.Age = 20
```

wartość 20 zostaje przekazana do metody `Set` właściwości `Age` jako argument `Value`.

Składowe obiektowe i współużytkowane

Elementy składowe klasy można podzielić na dwie grupy.

Składowe obiektowe
: Dostępne jedynie poprzez instancję, czyli obiekt danej klasy. Innymi słowy — składowe obiektowe „należą" do konkretnego obiektu, a nie do klasy jako całości.

Składowe (statyczne) współużytkowane
: Dostęp do składowych współużytkowanych jest możliwy bez tworzenia obiektu danej klasy. Te składowe są współużytkowane między wszystkimi obiektami danej klasy. Mówiąc ściślej — są niezależne od jakiegokolwiek obiektu klasy. Składowe współużytkowane „należą" do klasy jako całości, a nie do jej poszczególnych obiektów (czyli instancji).

Dostęp do składowych obiektowych wymaga podania przed nazwą składowej nazwy obiektu. Tak jak w poniższym fragmencie kodu:

```
Dim APerson As New CPerson()
APerson.Age = 50
```

Dostęp do składowych współużytkowanych wymaga podania nazwy klasy. Jako przykład omówimy klasę `String` w przestrzeni nazw `System` biblioteki klas .NET `Base` posiadającą metodę współużytkowaną `Compare`, która porównuje dwa łańcuchy. Składnia metody `Compare`:

```
Public Shared Function Compare(String, String) As Integer
```

Metoda `Compare` w przypadku równych łańcuchów zwraca 0. W przeciwnym razie zwraca wartość –1, jeżeli pierwszy łańcuch jest mniejszy od drugiego (jest wcześniejszy w porządku alfabetycznym) i 1, gdy pierwszy łańcuch jest większy od drugiego. Ponieważ `Compare` jest metodą współużytkowaną możliwy jest zapis:

```
Dim s As String = "steve"
Dim t As String = "donna"
MsgBox(String.Compare(s, t))                ' Wyświetla  1
```

Należy zauważyć, że nazwa metody `Compare` poprzedzona jest nazwą klasy `String`.

Składowe współużytkowane są pomocne podczas kontroli danych niezależnych od wszystkich obiektów danej klasy. Załóżmy, że chcemy kontrolować liczbę istniejących obiektów klasy `CPerson`. Napiszmy następujący podprogram:

```
' Deklaracja zmiennej współużytkowanej przechowującej liczbę obiektów klasy
Private Shared miInstanceCount As Integer

' Zwiększenie wartości miInstanceCount w konstruktorze
' (Poniższy kod należy dodać również do pozostałych
' konstruktorów, o ile takowe zostały zadeklarowane)
Sub New()
  miInstanceCount += 1
End Sub

' Utworzenie funkcji zwracającej liczbę obiektów
Shared Function GetInstanceCount() As Integer
  Return miInstanceCount
End Function

' Zmniejszenie wartości miInstanceCount w destruktorze
Overrides Protected Sub Finalize()
  miInstanceCount -= 1
  MyBase.Finalize
End Sub
```

Teraz za pomocą poniższego kodu mamy dostęp do zmiennej współużytkowanej:

```
Dim steve As New CPerson()
MsgBox(CPerson.GetInstanceCount)        ' Wyświetla 1
Dim donna As New CPerson()
MsgBox(CPerson.GetInstanceCount)        ' Wyświetla 2
```

Konstruktory klas

Podczas tworzenia obiektu danej klasy kompilator wywołuje specjalną funkcję zwaną *konstruktorem klasy* lub *konstruktorem obiektu*. Konstruktory mogą służyć do inicjowania obiektu, gdy zachodzi taka potrzeba (zastępują one zdarzenie `Class_Initialize` stosowane we wcześniejszych wersjach VB).

Konstruktory można definiować w module klasy. Jeżeli nie zdefiniujemy żadnego konstruktora, to VB użyje konstruktora domyślnego. W poniższym fragmencie kodu:

```
Dim APerson As CPerson = New CPerson()
```

wywoływany jest domyślny konstruktor klasy `CPerson`, ponieważ nie zadeklarowaliśmy własnego konstruktora.

Utworzenie własnego konstruktora polega na zdefiniowaniu procedury o nazwie *New* wewnątrz modułu klasy. Przypuśćmy, że chcemy podczas tworzenia obiektu klasy `CPerson` nadać jego właściwości `Name` konkretną wartość. Wtedy powinniśmy umieścić następujący kod w klasie `CPerson`:

```
' Konstruktor użytkownika
Sub New(ByVal sName As String)
  Me.Name = sName
End Sub
```

Teraz można utworzyć obiekt klasy `CPerson` i ustawić jego nazwę:

```
Dim APerson As CPerson = New CPerson("fred")
```

lub:

```
Dim APerson As New CPerson("fred")
```

Należy zauważyć, że VB .NET umożliwia przeciążanie (to zagadnienie omówiono w dalszej części tego rozdziału), więc możliwe jest zdefiniowanie wielu konstruktorów w tej samej klasie pod warunkiem, że każdy z nich ma unikatową sygnaturę argumentów. W takim przypadku można wywołać dowolny z nich, podając odpowiednią liczbę oraz odpowiedni dla tego konstruktora typ argumentów.

Po zadeklarowaniu jednego lub większej ilości konstruktorów użytkownika nie jest możliwe wywołanie konstruktora domyślnego (tzn. bez parametrów) za pomocą następującej instrukcji:

```
Dim APerson As New CPerson()
```

W tym przypadku należy jawnie zadeklarować bezparametrowy konstruktor w module klasy:

```
' Konstruktor domyślny
Sub New()
        …
End Sub
```

Finalize, Dispose i oczyszczanie pamięci

W VB 6 programista ma możliwość zaimplementowania zdarzenia `Class_Terminate` w celu wykonania określonych czynności przed zniszczeniem obiektu. Jeżeli na przykład obiekt otworzył plik i przechowuje jego uchwyt, ważne jest, by zamknąć ten plik przed jego zniszczeniem.

W VB .NET nie istnieje zdarzenie `Class_Terminate` przez co inny jest mechanizm obsługi tego typu sytuacji. W celu zrozumienia omawianych zagadnień należy najpierw omówić proces oczyszczania pamięci, zwany inaczej „odśmiecaniem".

Kiedy program odpowiedzialny za oczyszczanie pamięci stwierdzi, że obiekt nie jest już potrzebny (ma to na przykład miejsce, gdy wykonywany program nie zawiera już referencji do tego obiektu), wtedy automatycznie wywoływana jest specjalna *metoda destruktora* pod nazwą `Finalize`. Ważne jest jednak, by zrozumieć, że w przeciwieństwie do zdarzenia `Class_Terminate` nie ma teraz możliwości określenia dokładnego czasu wywołania przez program oczyszczający pamięć metody `Finalize`. Możemy być jedynie pewni, że zostanie ona wywołana jakiś czas po zwolnieniu ostatniego odwołania do obiektu. Opóźnienie wynika z faktu, że .NET Framework używa systemu pod nazwą *odśmiecanie ze śledzeniem odwołań*, który zwalnia *okresowo* nieużywane zasoby.

Metoda `Finalize` jest metodą `Protected`. Oznacza to, że może być wywoływana jedynie z klasy bazowej oraz jej klas pochodnych. Nie ma możliwości wywołania metody `Finalize` spoza klasy (klasa nie powinna praktycznie nigdy wywoływać swojej metody

`Finalize` bezpośrednio, ponieważ destruktor `Finalize` wywoływany jest automatycznie przez program oczyszczający pamięć). Jeżeli utworzymy metodę `Finalize` danej klasy, to powinniśmy również jawnie w jej treści wywołać metodę `Finalize` jej klasy bazowej. W ten sposób składnia i format metody `Finalize` wygląda następująco:

```
Overrides Protected Sub Finalize()
  ' Zaplanowane czynności
  MyBase.Finalize
End Sub
```

Korzyści wynikające z tego typu oczyszczania pamięci polegają na automatyzacji tego procesu oraz zapewnieniu tego, że niewykorzystane zasoby zostaną na pewno uwolnione bez jakiejkolwiek ingerencji ze strony programisty. Natomiast wadą jest brak możliwości bezpośredniego zainicjowania oczyszczania pamięci przez aplikację, przez co niektóre zasoby mogą pozostać w użyciu dłużej, niż to jest konieczne. Mówiąc prosto z mostu: nie możemy zniszczyć obiektu na żądanie.

Należy wziąć pod uwagę, że nie wszystkie zasoby są zarządzane przez Common Language Runtime. Niektóre zasoby, takie jak na przykład uchwyty okien i połączenia z bazami danych, nie podlegają automatycznemu oczyszczaniu. Dlatego w celu ich zwolnienia należy dołączyć odpowiedni kod w metodzie `Finalize`. Ten sposób nie umożliwia jednak zwalniania zasobów na żądanie. Do tego celu służy drugi destruktor o nazwie `Dispose`, który jest zdefiniowany w bibliotece klas `Base`. Destruktor `Dispose` ma następującą składnię:

```
Class classname
  Implements IDisposable
  Public Sub Dispose() Implements IDisposable.Dispose
    ' Zaplanowane czynności (np. zwalnianie zasobów)
    ' Wywołanie metod Dispose obiektów potomnych, jeżeli to jest konieczne
  End Sub

  ' Pozostałe składowe klasy
End Class
```

Należy zauważyć, że klasy wykorzystujące ten rodzaj destruktora muszą implementować interfejs `IDisposable` — z tego powodu konieczna jest instrukcja `Implements` pokazana powyżej. Interfejs `IDisposable` ma tylko jedną składową, a jest nią metoda `Dispose`.

Konieczne jest poinformowanie użytkowników klasy, że muszą oni wywoływać tę metodę *jawnie* w celu zwolnienia zasobów (technicznym terminem jest tutaj określenie *ręczne podejście*!).

Dziedziczenie

Najlepszym sposobem opisania mechanizmu dziedziczenia zastosowanego w VB .NET będzie podanie następującego przykładu.

Klasy wykorzystywane przez aplikację często są powiązane między sobą. Weźmy jako przykład dane odnoszące się do pracowników. Wspólne dane wszystkich pracowników

reprezentowane są przez obiekty `Employee` (pracownik) klasy `CEmployee` — można tutaj wymienić imię, nazwisko, nazwisko, adres, pobory itd.

Dodatkowe zarobki kierownictwa przedsiębiorstwa będą oczywiście inne niż analogiczne zarobki, powiedzmy, osoby pracującej na stanowisku sekretarki. Z tego względu rozsądnie byłoby zdefiniować dodatkowe klasy `CExecutive` i `CSecretary`, każdą ze swoimi własnymi właściwościami i metodami. Z drugiej strony — kierownik, to także pracownik i nie ma powodu definiowania dwu różnych właściwości `Name` w tym przypadku. Takie postępowanie jest nieefektywne i prowadzi do marnowania zasobów.

W tym właśnie celu stosuje się dziedziczenie. Najpierw zdefiniujemy klasę `CEmployee`, która implementuje właściwość `Salary` i metodę `IncSalary`:

```
' Klasa Employee
Public Class CEmployee
  ' Właściwość Salary umożliwia zapis / odczyt
  Private mdecSalary() As Decimal
  Property Salary() As Decimal
    Get
      Salary = mdecSalary
    End Get
    Set(ByVal Value as Decimal)
      mdecSalary = Value
    End Set
  End Property
  Public Overridable Sub IncSalary(ByVal sngPercent As Single)
    mdecSalary = mdecSalary * (1 + CDec(sngPercent))
  End Sub
End Class
```

Następnie definiujemy klasę `CExecutive`:

```
' Klasa CExecutive
Public Class CExecutive
  Inherits CEmployee
  ' Oblicz wzrost zarobków uwzgledniajacy 5% dodatku na samochód
  Overrides Sub IncSalary(ByVal sngPercent As Single)
    Me.Salary = Me.Salary * CDec(1.05 + sngPercent)
  End Sub
End Class
```

Należy zwrócić uwagę na dwie kwestie. Po pierwsze instrukcja:

```
Inherits CEmployee
```

określa, że klasa `CExecutive` *dziedziczy* składowe klasy `CEmployee`. Mówiąc inaczej — obiekt klasy `CExecutive` jest również obiektem klasy `CEmployee`. Zatem jeżeli utworzymy obiekt klasy `CExecutive`:

```
Dim ceo As New CExecutive
```

to mamy dostęp do właściwości `Salary`:

```
ceo.Salary = 1000000
```

Po drugie — słowo kluczowe `Overrides` w metodzie `IncSalary` oznacza, że metoda `IncSalary` klasy `CExecutive` jest wywoływana zamiast metody zaimplementowanej w `CEmployee`. Zatem kod:

```
ceo.IncSalary
```

zwiększa pobory obiektu `ceo` klasy `CExecutive` uzględniając dodatek na samochód. W deklaracji metody `IncSalary` klasy `CEmployee` znajduje się słowo kluczowe `Overridable` oznaczające, że klasy dziedziczące od klasy podstawowej mogą przesłaniać odnośną metodę klasy bazowej.

Następnie definiujemy nową klasę `CSecretary`, która też dziedziczy z klasy `CEmployee`, ale implementuje inny rodzaj dodatku do pensji:

```
' Klasa CSecretary
Public Class CSecretary
  Inherits CEmployee
  ' Sekretarka otrzymuje 2% dodatek za nadgodziny
  Overrides Sub IncSalary(ByVal sngPercent As Single)
    Me.Salary = Me.Salary * CDec(1.02 + sngPercent)
  End Sub
End Class
```

Napiszemy teraz kod wykorzystujący powyższe klasy:

```
' Utworzenie nowych obiektów
Dim ThePresident As New CExecutive()
Dim MySecretary As New CSecretary()
' Ustalenie zarobków
ThePresident.Salary = 1000000
MySecretary.Salary = 30000
' Wyświetl zarobki przewodniczącego i sekretarki bez oraz z dodatkiem
Debug.WriteLine("Prezes przed: " & CStr(ThePresident.Salary))
ThePresident.IncSalary(0.4)
Debug.WriteLine("Prezes po: " & CStr(ThePresident.Salary))

Debug.WriteLine("Sekretarka przed: " & CStr(MySecretary.Salary))
MySecretary.IncSalary(0.4)
Debug.WriteLine("Sekretarka po: " & CStr(MySecretary.Salary))
```

Wynik wykonania programu:

```
Prezes przed: 1000000
Prezes po: 1450000
Sekretarka przed: 30000
Sekretarka po: 42600
```

Dziedziczenie jest stosunkowo prostym pojęciem. W dokumentacji Microsoftu znajduje się taki jego opis:

> *Jeżeli Klasa B dziedziczy od Klasy A, to każdy obiekt Klasy B jest również obiektem Klasy A i zawiera publiczne metody i właściwości (tzn. interfejs publiczny) Klasy A. W tym przypadku Klasa A nazywana jest klasą bazową, a Klasa B klasą potomną. Klasa potomna może przesłaniać składowe klasy podstawowej zgodnie ze swoimi potrzebami.*

W poprzednim przykładzie zauważyliśmy, że słowem kluczowym definiującym dziedziczenie jest `Inherits`.

Pozwolenie na dziedziczenie

Istnieją dwa słowa kluczowe, używane w deklaracji klasy bazowej, które określają możliwość dziedziczenia z tej klasy.

`NotInheritable`
Użycie tego słowa kluczowego w deklaracji klasy:

```
Public NotInheritable Class InterfaceExample
```

powoduje, że klasa nie może być klasą bazową.

`MustInherit`
Użycie tego słowa kluczowego w deklaracji klasy:

```
Public MustInherit Class InterfaceExample
```

sprawia, że obiekty tej klasy nie mogą być tworzone bezpośrednio. Mogą być natomiast tworzone obiekty klas potomnych. Innymi słowy — klasy `MustInherit` mogą być klasami bazowymi i *tylko* klasami bazowymi.

Przesłanianie

Istnieje kilka słów kluczowych określających to, czy (i w jaki sposób) klasa potomna może przesłaniać implementację klasy bazowej. Omawiane słowa kluczowe używane są w deklaracji odnośnej składowej, a nie w deklaracji klasy.

`Overridable`
Zezwala na przesłanianie, jednak nie wymaga, by składowa była przesłaniana. Domyślnym ustawieniem dla składowej `Public` jest `NotOverridable`:

```
Public Overridable Sub IncSalary()
```

`NotOverridable`
Zabrania przesłaniania danej składowej. Domyślne ustawienie dla składowych `Public` klasy.

`MustOverride`
Składowa musi zostać przesłonięta. W przypadku użycia tego słowa kluczowego nie podajemy w deklaracji składowej implementacji i nie używamy słów kluczowych `End Sub` i `End Function`:

```
Public MustOverride Sub IncSalary()
```

Jeżeli klasa zawiera składową ze słowem kluczowym `MustOverride`, to musi zostać zadeklarowana jako klasa `MustInherit`.

`Overrides`
W przeciwieństwie do poprzednich modyfikatorów ten modyfikator stosowany jest w przypadku składowych klas potomnych i wskazuje, że modyfikowana składowa przesłania składową klasy bazowej:

```
Overrides Sub IncSalary()
```

Reguły dziedziczenia

W wielu językach obiektowych, takich jak na przykład w C++, klasa może dziedziczyć bezpośrednio z więcej niż jednej klasy bazowej. Określane jest to *dziedziczeniem wielokrotnym*. W VB .NET nie jest możliwe dziedziczenie wielokrotne. W ten sposób klasa może bezpośrednio dziedziczyć najwyżej z jednej klasy. Poniższy fragment kodu nie jest więc poprawny:

```
' Klasa Executive
Public Class CExecutive                ' NIEPOPRAWNE
  Inherits CEmployee
  Inherits CWorker
  …
End Class
```

Natomiast Klasa C może dziedziczyć z Klasy B, która z kolei może dziedziczyć z Klasy A. W ten sposób możliwe jest utworzenie hierarchii dziedziczenia. Klasa może również implementować wiele interfejsów za pomocą słowa kluczowego `Interface`. Powyższe zagadnienie jest omawiane w dalszej części tego rozdziału.

MyBase, MyClass i Me

Słowo kluczowe `MyBase` umożliwia dostęp do klasy bazowej z klasy potomnej. W celu wywołania składowej klasy bazowej z klasy potomnej należy użyć następującej składni:

```
MyBase.MemberName
```

gdzie w miejscu *MemberName* podaje się nazwę składowej klasy, do której mamy się odwołać. W ten sposób nie powstaje żadna dwuznaczność w przypadku, gdy klasa potomna ma również składową o tej samej nazwie.

Słowo kluczowe `MyBase` może zostać użyte do wywołania konstruktora klasy bazowej przy tworzeniu obiektu klasy potomnej:

```
MyBase.New(…)
```

Nie można za pomocą `MyBase` wywoływać składowych klasy o dostępie `Private`.

W przypadku użycia słowa kluczowego `MyBase` wyszukiwana jest składowa znajdująca się najbliżej w drzewie dziedziczenia. Zatem jeżeli klasa C dziedziczy od klasy B, która dziedziczy od klasy A, to wywołanie procedury `AProc` w klasie C:

```
MyBase.AProc
```

powoduje, że najpierw przeszukiwana jest klasa B w celu znalezienia procedury o nazwie *AProc*. Jeżeli w tej klasie nie zostanie znaleziona poszukiwana procedura, to VB .NET przeszuka klasę A, szukając odpowiedniej procedury (pod pojęciem *odpowiedniej* procedury rozumiemy procedurę o takiej samej nazwie i sygnaturze argumentów).

Słowo kluczowe `MyClass` umożliwia dostęp do klasy, w której zostało użyte. `MyClass` jest podobne do słowa kluczowego `Me`. Jedyna różnica występuje przy wywoływaniu metod. W celu zilustrowania tego zagadnienia utworzymy klasę `Class1` i klasę pochodną o nazwie `Class1Derived`. Obie klasy posiadają metodę `IncSalary`:

```
Public Class Class1
  Public Overridable Function IncSalary(ByVal sSalary As Single) _
                                        As Single
    IncSalary = sSalary * CSng(1.1)
  End Function
  Public Sub ShowSalary(ByVal sSalary As Single)
    MsgBox(Me.IncSalary(sSalary))
    MsgBox(MyClass.IncSalary(sSalary))
  End Sub
End Class

Public Class Class1Derived
  Inherits Class1
  Public Overrides Function IncSalary(ByVal sSalary As Single) _
                                      As Single
    IncSalary = sSalary * CSng(1.2)
  End Function
End Class
```

Przeanalizujmy teraz poniższy fragment kodu:

```
Dim c1 As New Class1()
Dim c2 As New Class1Derived()

Dim clvar As Class1

clvar = c1
clvar.ShowSalary(10000)               ' Wyświetla 11000, 11000

clvar = c2
clvar.ShowSalary(10000)               ' Wyświetla 12000, 11000
```

Pierwsze wywołanie metody `IncSalary` wykorzystuje zmienną typu `Class1`, która jest referencją do obiektu typu `Class1`. W tym przypadku oba poniższe wywołania:

```
Me.IncSalary
MyClass.IncSalary
```

zwracają taką samą wartość, ponieważ wywołują metodę `IncSalary` w klasie bazowej `Class1`.

Jednak za drugim razem zmienna typu `Class1` zawiera referencję do obiektu klasy potomnej — `Class1Derived`. Teraz `Me` odnosi się do obiektu typu `Class1Derived`, podczas gdy `MyClass` nadal wskazuje klasę bazową `Class1`, w której słowo kluczowe `MyClass` występuje. W ten sposób:

```
Me.IncSalary
```

zwraca 12000, natomiast:

```
MyClass.IncSalary
```

zwraca 11000.

Interfejsy, abstrakcyjne składowe i klasy

W trakcie wcześniejszych rozważań wspomnieliśmy już, że klasa może implementować wszystkie składowe interfejsu, część lub może nie implementować żadnej składowej interfejsów, które deklaruje. Składowa interfejsu, która nie zawiera swojej implementacji określana jest jako *składowa abstrakcyjna*. Zadaniem składowej abstrakcyjnej jest podanie sygnatury składowej (*wzorca*), która może zostać zaimplementowana przez jedną lub więcej *klas potomnych* w różny sposób w każdej z nich.

Wyjaśnimy powyższe zagadnienie na przykładzie. Przypomnijmy, że klasa `CEmployee` deklaruje i implementuje metodę `IncSalary` zwiększającą pobory pracownika. Ponadto dodajmy, że klasy potomne `CExecutive` i `CSecretary` przesłaniają implementację metody `IncSalary` klasy bazowej `CEmployee`.

Przypuśćmy, że utworzyliśmy bardziej rozbudowany model pracowników, w którym każdemu stanowisku przyporządkowano oddzielną klasę pochodną. Każda z tych klas przesłania implementację metody `IncSalary` klasy bazowej `CEmployee`. W tym przypadku implementacja metody `IncSalary` klasy bazowej nie musi być nigdy wywoływana. Po co więc podawać implementację metody, która nigdy nie będzie wykorzystana?

Zamiast tego, możemy po prostu utworzyć metodę `IncSalary` z pustym „ciałem”:

```
' Klasa Pracownik
Public Class CEmployee
  …
  Public Overridable Sub IncSalary(ByVal sngPercent As Single)
  End Sub
End Sub
```

Jeżeli wszystkie klasy pochodne *muszą* implementować metodę `IncSalary`, wtedy należy użyć słowa kluczowego `MustOverride`:

```
' Klasa pracownik
Public MustInherit Class CEmployee
  …
  Public MustOverride Sub IncSalary(ByVal sngPercent As Single)

End Class
```

Jak już stwierdziliśmy, nie ma skojarzonej instrukcji `End Sub` ze słowem kluczowym `MustOverride`. Należy również pamiętać, że słowo kluczowe `MustOverride` wymaga, by klasa, w której zostało użyte, zadeklarowana została ze słowem kluczowego `MustInherit`. W ten sposób stwierdzamy, że nie wolno bezpośrednio tworzyć obiektów klasy `CEmployee`.

W powyższych przykładach składowa `IncSalary` klasy bazowej `CEmployee` jest składową abstrakcyjną.

Klasa z przynajmniej jedną składową abstrakcyjną nazywana jest *klasą abstrakcyjną*. Zatem zdefiniowana powyżej klasa `CEmployee` jest klasą abstrakcyjną. Zastosowana termino-

logia wynika z faktu, że nie jest możliwe utworzenie obiektu klasy abstrakcyjnej, ponieważ przynajmniej jedna z metod obiektu nie posiadałaby implementacji.

W pewnych sytuacjach może nam być potrzebna klasa, której wszystkie składowe są abstrakcyjne. Innymi słowy — jest to klasa, która jedynie *definiuje* interfejs. Taką klasę można określić mianem klasy *czysto abstrakcyjnej*, chociaż nie jest to ogólnie przyjęte sformułowanie.

Jako przykład weźmy klasę określającą kształt zwaną `CShape`, której celem jest przedstawienie ogólnych właściwości i operacji, które mogą być wykonywane na figurach geometrycznych (elipsach, prostokątach, trapezoidach itp.). Każdy kształt czy figura wymaga metody `Draw` do narysowania samej siebie. Implementacja takiej metody będzie różna dla różnych rodzajów figur — na przykład inaczej rysujemy okręgi, a inaczej prostokąty. W identyczny sposób będziemy musieli jednak dołączyć metody, takie jak `Rotate` (obrót), `Translate` (przesunięcie) i `Reflect` (odbicie). Każda z tych metod będzie posiadała inną implementację (w zależności od reprezentowanej figury czy kształtu).

Zatem możemy zdefiniować klasę `CShape` na dwa sposoby:

```
Public Class Class2

  Public Overridable Sub Draw()
  End Sub

  Public Overridabe Sub Rotate(ByVal sngDegrees As Single)
  End Sub

  Public Overridabe Sub Translate(ByVal x As Integer, _
                                  ByVal y As Integer)
  End Sub

  Public Overridable Sub Reflect(ByVal iSlope As Integer, _
                                 ByVal iIntercept As Integer)
  End Sub
End Class
```

lub:

```
Public MustInherit Class CShape

  Public MustOverride Sub Draw()
  Public MustOverride Sub Rotate(ByVal sngDegresss As Single)
  Public MustOverride Sub Rotate(ByVal x As Integer, _
                                 ByVal y As Integer)
  Public MustOverride Sub Reflect(ByVal iSlope As Integer, _
                                  ByVal iIntercept As Integer)
End Class
```

Teraz możemy zdefiniować klasy potomne, takie jak `CRectangle`, `CEllipse`, `CPolygon` itp. Każda z tych klas będzie implementować (lub musi implementować w drugim przypadku) składowe klasy bazowej `CShape`. Nie omawiamy tutaj jednak szczegółów takiej implementacji, gdyż nie stanowi to głównego tematu naszych rozważań.

Interfejsy raz jeszcze

Powiedzieliśmy, że interfejsy mogą być zdefiniowane w modułach klas. W VB .NET możliwy jest również inny sposób definiowania interfejsu za pomocą słowa kluczowego `Interface`. Poniżej zdefiniowano interfejs `IShape`:

```
Public Interface IShape
  Sub Draw()
  Sub Rotate(ByVal sngDegrees As Single)
  Sub Translate(ByVal x As Integer, ByVal y As Integer)
  Sub Reflect(ByVal iSlope As Integer, _
              ByVal iIntercept As Integer)
End Interface
```

W przypadku zastosowania słowa kluczowego `Interface` nie można *implementować* składowych wewnątrz modułu, w którym zdefiniowano interfejs. Można natomiast zaimplementować interfejs za pomocą zwykłego modułu klasy. Ważne jest wykorzystanie w tej sytuacji instrukcji `Implements` (która była dostępna również w VB 6, jednak mogła odnosić się jedynie do interfejsów zewnętrznych):

```
Public Class CRectangle

' Implementacja interfejsu IShape
  Implements IShape

  Public Overridable Sub Draw() Implements IShape.Draw
    ' kod metody
  End Sub

  Public Overridable Sub Spin(ByVal sngDegrees As Single) Implements
IShape.Rotate
    ' kod metody
  End Sub

  ...

End Class
```

Należy zauważyć, że słowo kluczowe `Implements` występuje dodatkowo w każdej funkcji, która implementuje składową interfejsu. Zastosowanie tego słowa kluczowego umożliwia nadanie funkcji implementującej dowolnej nazwy. Funkcja implementująca nie musi mieć takiej samej nazwy jak metoda (metoda `Spin` implementuje metodę `Rotate` interfejsu `IShape`). Jednak nadawanie tej samej nazwy obu funkcjom jest niewątpliwie bardziej czytelne (a przez to prezentuje lepszy styl programowania.)

Główną korzyścią wynikającą ze stosowania słowa kluczowego `Implements` do definiowania interfejsu jest fakt, że pojedyncza klasa może w ten sposób implementować wiele interfejsów. Dzieje się tak, pomimo że VB .NET nie zezwala na dziedziczenie w obrębie jednej klasy z wielu klas bazowych. Z drugiej strony — stosowanie słowa kluczowego `Interface` wiąże się z brakiem możliwości podania implementacji w module definiującym ten interfejs. W ten sposób *wszystkie* składowe muszą zostać zaimplementowane w *każdej* klasie implementującej dany interfejs. Oznacza to powtarzanie tego samego kodu w przypadku, gdy składowa interfejsu ma taką samą implementację w więcej niż jednej implementującej klasie.

Polimorfizm i przeciążanie

Na szczęście nie musimy wnikać w szczegóły leżące u podstaw mechanizmu polimorfizmu i przeciążania z całą ich zawiłością i dwuznacznością. Niektórzy informatycy na przykład mówią, że przeciążanie jest formą polimorfizmu, inni natomiast twierdzą, że nie jest tak. Omówimy jedynie te zagadnienia, które wiążą się bezpośrednio z .NET Framework.

Przeciążanie

Terminu *przeciążanie* używamy w odniesieniu do takich elementów języka, które mogą zostać użyte w więcej niż jeden sposób. Nazwy operatorów są często przeciążane. Znak plus (+) na przykład może określać dodawanie liczb całkowitych, dodawanie liczb typu `Single`, dodawanie liczb `Double` czy łączenie łańcuchów. W ten sposób symbol plus (+) jest przeciążany. Ma to swoje dobre strony, gdyż w przeciwnym przypadku musielibyśmy używać oddzielnych symboli do dodawania liczb całkowitych, liczb typu `Single` czy typu `Double`.

Nazwy funkcji są również przeciążane. Funkcja *Abs* zwracająca wartość bezwzględną liczby może pobierać jako parametr liczbę całkowitą, liczbę typu `Single` lub typu `Double`. Ponieważ nazwa *Abs* reprezentuje klika różnych funkcji, to znaczy to, że jest nazwą przeciążoną. Rzeczywiście tak jest — przeglądając dokumentację składowej `Abs` klasy `Math` (w systemowej przestrzeni nazw biblioteki klas `Base`) znajdujemy następujące deklaracje funkcji o nazwie `Abs`:

```
Overloads Public Shared Function Abs(Decimal) As Decimal
Overloads Public Shared Function Abs(Double) As Double
Overloads Public Shared Function Abs(Short) As Short
Overloads Public Shared Function Abs(Integer) As Integer
Overloads Public Shared Function Abs(Long) As Long
Overloads Public Shared Function Abs(SByte) As SByte
Overloads Public Shared Function Abs(Single) As Single
```

Słowo kluczowe `Overloads` oznacza, że funkcja jest przeciążona.

Nazwa funkcji jest przeciążona, jeżeli dwie zdefiniowane funkcje mają tę samą nazwę, ale różne *sygnatury argumentów*. Weźmy jako przykład funkcję zwracającą bieżące saldo rachunku. Rachunek może być identyfikowany przez nazwę osoby albo przez numer rachunku. W ten sposób możemy zdefiniować dwie funkcje i obydwie nazwać *GetBalance*:

```
Overloads Function GetBalance(sCustName As String) As Decimal
Overloads Function GetBalance(sAccountNumber As Long) As Decimal
```

Należy zauważyć, że przeciążanie funkcji w VB .NET możliwe jest dlatego, że sygnatury argumentów dwu funkcji są różne. W ten sposób nie ma dwuznaczności. Wywołania funkcji:

```
getBalance("John Smith")
getBalance(123456)
```

są interpretowane przez kompilator bez trudności na podstawie typu danych argumentu. Ten rodzaj przeciążania jest często określany mianem przeciążania funkcji *GetBalance*. Przedstawione powyżej funkcje są jednak dwiema różnymi funkcjami, bardziej poprawne jest więc stwierdzenie, że przeciążana jest *nazwa* funkcji. Przeciążanie jest powszechnie stosowaną techniką programistyczną, której zastosowanie wykracza daleko poza programowanie obiektowe.

Polimorfizm

Termin *polimorfizm* oznacza występowanie w wielu różnych postaciach. W .NET Framework polimorfizm jest ściśle związany z dziedziczeniem. Ponownie wróćmy do przykładu klasy opisującej pracownika przedsiębiorstwa. Funkcja *IncSalary* została zdefiniowana w trzech klasach: klasie bazowej `CEmployee` oraz jej klasach potomnych `CExecutive` i `CSecretary`. W ten sposób funkcja *IncSalary* występuje w trzech postaciach. To jest właśnie polimorfizm w stylu VB .NET.

Osoby zainteresowane omawianym zagadnieniem należy poinformować, że wielu informatyków nie uznaje przedstawionego powyżej mechanizmu za polimorfizm. W tym miejscu powiedzieliby, że funkcja *IncSalary* przyjmuje tylko jedną postać. Różne są jedynie implementacje, a nie funkcja. Opisaną sytuację określiliby mianem przeciążenia funkcji *IncSalary*. Problemem występującym w tym przypadku jest zamieszanie w sposobie, w jaki Microsoft oraz pozostałe osoby używają terminów przeciążenie i polimorfizm — dlatego należy być ostrożnym podczas czytania dokumentacji.

Zasięg i dostęp w module klasy

Pojęcie zasięgu w module klasy jest znacznie ważniejsze niż w zwykłych modułach. Nie ma większych różnic w przypadku rozpatrywania zmiennych lokalnych (o zasięgu ograniczonym do bloku lub procedury) — tak samo w module klasy mamy zmienne o zasięgu ograniczonym do bloku oraz zmienne ograniczone do procedury.

Natomiast zmiennym zadeklarowanym części `Declarations` modułu klasy może być przypisany jeden z następujących modyfikatorów dostępu:

- `Public`;
- `Private`;
- `Friend`;
- `Protected`;
- `Protected Friend`.

W przypadku zwykłych modułów dozwolone są jedynie `Public`, `Private` i `Friend`.

Sam moduł klasy może być zadeklarowany z jednym z trzech modyfikatorów dostępu: `Public`, `Private` lub `Friend` (`Protected` nie jest dozwolony). Podanie w deklaracji modułu klasy jednego z tych modyfikatorów dostępu po prostu ogranicza dostęp do wszystkich składowych modułu do tego poziomu dostępu. Podany zasięg może zostać dalej ograniczony dla konkretnej składowej poprzez umieszczenie modyfikatora dostępu w deklaracji tej składowej. W ten sposób w przypadku dostępu do klasy typu `Friend` żadna z jej składowych nie może mieć dostępu typu `Public` (innymi słowy — dostęp typu `Public` zostaje przesłonięty dostępem typu `Friend`).

Natomiast wszystkie cztery modyfikatory dostępu mają wpływ na składowe modułu klasy — to znaczy na deklaracje zmiennych, stałych, wyliczeń oraz procedur wewnątrz modułu klasy.

Problem wynika z powodu istnienia trzech typów dostępu do składowej klasy o różnym zasięgu. W celu wyjaśnienia zdefiniujmy następujące składowe — przedstawione deklaracje nie są być może standardowe, ale z pewnością dobrze opisują zagadnienie. Przeanalizujmy następującą deklarację zmiennej w części `Declarations` modułu klas o nazwie `Class1`:

```
AccessModifier classvariable As Integer
```

Dostęp do tej zmiennej jest możliwy w przedstawiony niżej sposób.

Bezpośredni dostęp
Określa dostęp do składowej bez żadnych dodatkowych określeń:

```
classvariable = 100
```

Podczas dostępu do zmiennej w bezpośredni sposób (tzn. bez dodatkowych określeń), zasięg zmiennej może obejmować:

- klasę deklarującą;
- klasę deklarującą i jej klasy pochodne (jedynie wewnątrz projektu, w którym ma miejsce deklaracja);
- klasę deklarującą i jej klasy pochodne (w każdym projekcie zawierającym odwołanie do projektu, w którym ma miejsce deklaracja).

Dostęp klasowy/obiektowy
Określa dostęp do składowej za pomocą określenia w postaci nazwy klasy lub nazwy obiektu tej klasy.

Stwierdziliśmy już, że w przypadku zadeklarowania zmiennej za pomocą słowa kluczowego `Shared`, jest ona wspólna dla wszystkich obiektów klasy. Wyrażając się ściślej można powiedzieć, że taka składowa istnieje niezależnie od jakiegokolwiek obiektu klasy. W tym przypadku dostęp do składowej (wewnątrz zasięgu składowej) jest możliwy przez podanie określenia w postaci nazwy klasy:

```
Class1.classvariable = 100
```

Należy zauważyć, że dostęp do takiej składowej jest możliwy również poprzez podanie nazwy obiektu. Rezultat jest taki sam jak w przypadku podania nazwy klasy — istnieje jedynie jedna kopia składowej.

Jeżeli składowa jest zadeklarowana bez użycia słowa `Shared`, wtedy dostęp możliwy jest przez podanie nazwy istniejącego obiektu:

```
Dim c As new Class1
c.classvariable = 100
```

Zasięg klasowy/obiektowy może obejmować:

- deklarującą klasę;
- deklarujący projekt;
- deklarujący projekt i te komponenty zewnętrzne, które mają odwołanie do deklarującego projektu.

Tabela 3.2 opisuje różne modyfikatory dostępu.

Tabela 3.2. Modyfikatory dostępu w module klasy

	Zasięg przy dostępie bezpośrednim	Zasięg klasy/obiektu
`Private`	Klasa deklarująca	Klasa deklarująca
`Protected`	Wszystkie pochodne klasy	Klasa deklarująca
`Friend`	Klasy pochodne w projekcie	Deklarowany projekt
`Protected Friend`	Wszystkie pochodne klasy	Deklarowany projekt
`Public`	Wszystkie pochodne klasy	Wszystkie projekty

Nie było możliwe niezależne przedstawienie wpływu modyfikatorów `Friend` i `Protected`. Lepszym rozwiązaniem byłoby zapewne oddzielne przedstawienie modyfikatorów dostępu przy bezpośrednim i klasowym lub obiektowym dostępie, zamiast powyższego przeplatania pojęć w tabeli 3.2. No cóż...