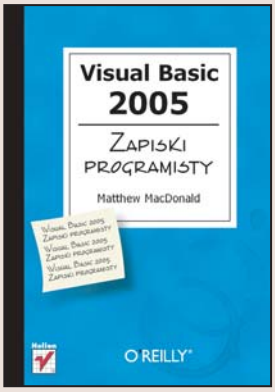
Visual Basic 2005. Zapiski programisty

Autor: Matthew MacDonald
Tłumaczenie: Grzegorz Werner
ISBN: 83-246-0087-6
Tytuł oryginału: Visual Basic 2005: A Developers Notebook
Format: B5, stron: 304
Przykłady na ftp: 1986 kB



Visual Basic 2005 nie jest tak rewolucyjnym produktem, jak Visual Basic .NET. Opracowując wersję 2005, twórcy języka skoncentrowali się na usunięciu błędów i usterek oraz zwiększeniu komfortu pracy programisty. Narzędzia i kontrolki, w które wyposażono zarówno najnowszą wersję Visual Basica, jak i środowisko programistyczne Visual Studio 2005, pozwalają znacznie przyspieszyć pisanie kodu. Jednocześnie został zachowany dostęp do wszystkich możliwości platformy .NET.

Książka „Visual Basic 2005. Zapiski programisty" to zbiór notatek spisanych przez programistów analizujących tę wersję języka. Zawiera ćwiczenia ilustrujące nowe funkcje Visual Basica 2005, platformy .NET Framework 2.0 i środowiska programistycznego Visual Studio 2005. Programiści korzystający z wcześniejszych wersji tych narzędzi szybko opanują nowe funkcje, takie jak definiowanie klas generycznych czy korzystanie z obiektów My. Godne odnotowania jest także znaczne przyspieszenie i udoskonalenie technologii ASP.NET.

- Edycja kodu w Visual Studio 2005
- Tworzenie dokumentacji w formacie XML
- Korzystanie z obiektów My
- Definiowanie klas generycznych
- Tworzenie aplikacji dla środowiska Windows oraz aplikacji WWW
- Projektowanie formularzy
- Komunikacja z bazami danych
- Wdrażanie aplikacji za pomocą technologii ClickOnce

Dzięki tej książce najnowsza wersja Visual Basica odsłania swoje tajemnice.

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

Spis treści

ROZDZIAŁ 2

Język Visual Basic

W tym rozdziale:

- ✓ Wykonywanie popularnych zadań za pomocą obiektów My
- ✓ Uzyskiwanie informacji o aplikacji
- ✓ Używanie ściśle typizowanych zasobów
- ✓ Używanie ściśle typizowanych ustawień konfiguracyjnych
- ✓ Tworzenie klas generycznych
- ✓ Używanie wartości pustych w prostych typach danych
- ✓ Używanie operatorów w połączeniu z własnymi obiektami
- ✓ Dzielenie klasy na wiele plików
- ✓ Rozszerzanie przestrzeni nazw My
- ✓ Przechodzenie do następnej iteracji pętli
- ✓ Automatyczne usuwanie obiektów

Kiedy pojawiła się pierwsza wersja Visual Basica .NET, lojalni programiści VB byli zaszokowani drastycznymi zmianami w ich ulubionym języku. Ni stąd, ni zowąd proste czynności, takie jak tworzenie instancji obiektu i deklarowanie struktury, wymagały nowej składni, a podstawowe typy, na przykład tablice, zostały przekształcone w coś zupełnie innego. Na szczęście w Visual Basicu 2005 nie ma takich niespodzianek. Zmiany w najnowszych wersjach języka upraszczają programowanie,

ale **nie sprawiają**, że istniejący kod staje się przestarzały. Wiele spośród tych zmian to funkcje przeniesione z języka C# (na przykład przeciążanie operatorów), a inne to zupełnie nowe składniki wbudowane w najnowszą wersję wspólnego środowiska uruchomieniowego (na przykład klasy generyczne). W tym rozdziale opiszemy najprzydatniejsze zmiany w języku VB.

Wykonywanie popularnych zadań za pomocą obiektów My

Nowe obiekty `My` zapewniają łatwy dostęp do popularnych funkcji, które niełatwo jest znaleźć w obszernej bibliotece klas .NET. Zasadniczo obiekty `My` udostępniają różnorodne elementy, od rejestru Windows do ustawień bieżącego połączenia sieciowego. Co najlepsze, hierarchia obiektów `My` jest zorganizowana według zastosowań i łatwo się po niej poruszać przy użyciu IntelliSense.

Przedzieranie się przez obszerną bibliotekę klas .NET w poszukiwaniu potrzebnej funkcji bywa męczące. Dzięki nowym obiektom `My` można szybko znaleźć najprzydatniejsze funkcje oferowane przez .NET.

Jak to zrobić?

Istnieje siedem podstawowych obiektów `My`. Spośród nich trzy kluczowe obiekty centralizują zestaw funkcji .NET Framework i dostarczają informacji o komputerze. Są to:

`My.Computer`

Ten obiekt zawiera informacje o komputerze, w tym o jego połączeniu sieciowym, stanie myszy i klawiatury, bieżącym katalogu, drukarce, ekranie oraz zegarze. Można też użyć go jako punktu wyjścia do odtwarzania dźwięków, wyszukiwania plików, dostępu do rejestru albo korzystania ze schowka Windows.

`My.Application`

Ten obiekt zawiera informacje o bieżącej aplikacji i jej kontekście, w tym o podzespole i jego wersji, kulturze, a także o argumentach wiersza polecenia użytych podczas uruchamiania aplikacji. Można go również użyć do rejestrowania zdarzeń występujących w aplikacji.

`My.User`

Ten obiekt zawiera informacje o bieżącym użytkowniku. Można użyć go do sprawdzenia konta użytkownika Windows i ustalenia, do jakiej grupy należy użytkownik.

Obok tych trzech obiektów istnieją kolejne dwa, które zapewniają dostęp do **instancji domyślnych**. Instancje domyślne to obiekty, które .NET tworzy automatycznie dla pewnych typów klas zdefiniowanych w aplikacji. Są to:

`My.Forms`

Ten obiekt zapewnia instancję domyślną każdego formularza Windows w aplikacji. Można użyć go do obsługi komunikacji między formularzami bez śledzenia referencji do formularzy w oddzielnej klasie.

`My.WebServices`

Ten obiekt zapewnia instancję domyślną klasy pośredniczącej dla każdej usługi WWW. Jeśli na przykład projekt używa dwóch referencji WWW, poprzez ten obiekt można uzyskać dostęp do gotowej klasy pośredniczącej dla każdej z nich.

Kolejne dwa obiekty `My` zapewniają łatwy dostęp do ustawień konfiguracyjnych i zasobów:

`My.Settings`

Ten obiekt umożliwia pobranie ustawień z pliku konfiguracyjnego XML aplikacji.

`My.Resources`

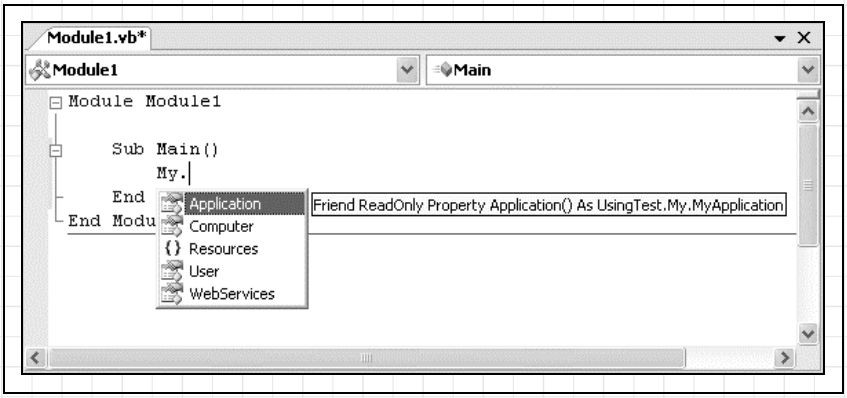
Ten obiekt umożliwia pobieranie **zasobów** — bloków danych binarnych lub tekstowych, które są kompilowane z podzespołem aplikacji. Zasobów zwykle używa się do przechowywania wielojęzycznych łańcuchów, obrazów i plików dźwiękowych.

OSTRZEŻENIE

Warto zaznaczyć, że działanie obiektów `My` zależy od typu projektu. Na przykład w aplikacjach WWW lub aplikacjach konsoli nie można korzystać z kolekcji `My.Forms`.

Niektóre spośród klas `My` są zdefiniowane w przestrzeni nazw `Microsoft.VisualBasic.MyServices`, a inne, na przykład klasy używane w obiektach `My.Settings` i `My.Resources`, są tworzone dynamicznie przez Visual Studio 2005 podczas modyfikowania ustawień aplikacji i dodawania zasobów do bieżącego projektu.

Aby wypróbować obiekt `My`, można skorzystać z funkcji IntelliSense. Wystarczy wpisać słowo `My` oraz kropkę i obejrzeć dostępne obiekty (rysunek 2.1). Następnie można wybrać jeden z nich i wpisać kolejną kropkę, aby przejść na niższy poziom hierarchii.



Rysunek 2.1. Przeglądanie obiektów My

W celu wypróbowania prostego kodu, który wyświetla kilka podstawowych informacji z wykorzystaniem obiektu `My`, utwórzmy nowy projekt typu *Console Application*. Następnie dodajmy poniższy kod do procedury `Main()`:

```
Console.WriteLine(My.Computer.Name)
Console.WriteLine(My.Computer.Clock.LocalTime)
Console.WriteLine(My.Computer.FileSystem.CurrentDirectory)
Console.WriteLine(My.User.Name)
```

Po uruchomieniu tego kodu w oknie konsoli zostanie wyświetlona nazwa komputera, bieżący czas, bieżący katalog oraz nazwa użytkownika:

```
SALESSERVER
2005-07-06 11:18:18
C:\Kod\NotatnikVB\1.07\Test\bin
MATTHEW
```

OSTRZEŻENIE

Obiekt `My` ma również „ciemną stronę". Rozwiązania, w których go zastosowano, trudniej współużytkować z innymi programistami, ponieważ nie jest on obsługiwany w językach innych niż VB (na przykład C#).

Więcej informacji

Aby dowiedzieć się więcej o obiekcie `My` i zobaczyć przykłady jego użycia, należy zajrzeć pod hasło indeksu „My Object" w pomocy MSDN. Można też wykonać inne ćwiczenia, w których wykorzystano obiekt `My`. Oto kilka przykładów:

- używanie obiektu `My.Application` do pobierania informacji o programie, takich jak bieżąca wersja oraz parametry wiersza polecenia podane podczas uruchamiania aplikacji (ćwiczenie „Uzyskiwanie informacji o aplikacji" dalej w tym rozdziale);
- wczytywanie obrazów i innych zasobów z podzespołu aplikacji za pomocą obiektu `My.Resources` (ćwiczenie „Używanie ściśle typizowanych zasobów" dalej w tym rozdziale);
- pobieranie ustawień aplikacji i użytkownika za pomocą obiektu `My.Settings` (ćwiczenie „Używanie ściśle typizowanych ustawień konfiguracyjnych" dalej w tym rozdziale);
- używanie obiektu `My.Forms` do interakcji między oknami aplikacji (ćwiczenie „Komunikacja między formularzami" w rozdziale 3.);
- manipulowanie plikami i połączeniami sieciowymi za pomocą obiektu `My.Computer` (rozdziały 5. i 6.);
- uwierzytelnianie użytkownika za pomocą obiektu `My.User` (ćwiczenie „Testowanie przynależności grupowej bieżącego użytkownika" w rozdziale 6.).

Za pomocą obiektu `My.Application` można pobrać informacje o bieżącej wersji aplikacji, jej położeniu oraz parametrach użytych do jej uruchomienia.

Uzyskiwanie informacji o aplikacji

Obiekt `My.Application` udostępnia wiele przydatnych informacji. W celu ich pobrania wystarczy odczytać właściwości obiektu.

Jak to zrobić?

Informacje zawarte w obiekcie `My.Application` przydają się w wielu różnych sytuacjach. Oto dwa przykłady:

- Chcemy ustalić dokładny numer wersji. Bywa to użyteczne, jeśli chcemy wyświetlić dynamiczne okno z informacjami o programie albo sprawdzić usługę WWW i upewnić się, że mamy najnowszą wersję podzespołu.
- Chcemy zarejestrować pewne informacje diagnostyczne. Jest to istotne, jeśli problem występuje u klienta i musimy zapisać ogólne informacje o działającej aplikacji.

Aby sprawdzić, jak to działa, można wykorzystać kod z listingu 2.1 w aplikacji konsoli. Pobiera on wszystkie informacje i wyświetla je w oknie konsoli.

Listing 2.1. Pobieranie informacji z obiektu My.Application

```
' Ustalamy parametry, z jakimi została uruchomiona aplikacja
Console.Write("Parametry wiersza polecenia: ")
For Each Arg As String In My.Application.CommandLineArgs
    Console.Write(Arg & " ")
Next
Console.WriteLine()
Console.WriteLine()

' Pobieramy informacje o podzespole, w którym znajduje się ten kod
' Informacje te pochodzą z metadanych (atrybutów w kodzie)
Console.WriteLine("Firma: " & My.Application.Info.CompanyName)
Console.WriteLine("Opis: " & My.Application.Info.Description)
Console.WriteLine("Lokalizacja: " & My.Application.Info.DirectoryPath)
Console.WriteLine("Prawa autorskie: " & My.Application.Info.Copyright)
Console.WriteLine("Znak towarowy: " & My.Application.Info.Trademark)
Console.WriteLine("Nazwa: " & My.Application.Info.AssemblyName)
Console.WriteLine("Produkt: " & My.Application.Info.ProductName)
Console.WriteLine("Tytuł: " & My.Application.Info.Title)
Console.WriteLine("Wersja: " & My.Application.Info.Version.ToString())
Console.WriteLine()
```

WSKAZÓWKA

Visual Studio 2005 zawiera okno *Quick Console*, które jest uproszczoną wersją zwykłego okna wiersza poleceń. W niektórych przypadkach okno to działa nieprawidłowo. W razie problemów z uruchamianiem przykładowej aplikacji i wyświetlaniem jej wyników należy wyłączyć okno *Quick Console*. W tym celu należy wybrać z menu *Tools* polecenie *Options*, upewnić się, że zaznaczone jest pole wyboru *Show all settings*, po czym przejść do pozycji *Debugging –> General*. Następnie należy usunąć zaznaczenie z pola *Redirect all console output to the Quick Console window*.

Przed przetestowaniem powyższego kodu warto skonfigurować środowisko tak, aby program zwrócił sensowne wyniki. Można na przykład przekazać parametry wiersza polecenia do uruchamianej aplikacji. W tym celu należy kliknąć dwukrotnie ikonę *My Project* w oknie *Solution Explorer*. Następnie należy przejść na zakładkę *Debug* i poszukać pola tekstowego *Command line parameters*. W polu tym należy wpisać parametry wiersza polecenia, na przykład `/a /b /c`.

Aby określić informacje, takie jak autor podzespołu, produkt, wersja itd., trzeba dodać specjalne atrybuty do pliku *AssemblyInfo.vb*, który zwykle nie jest widoczny w oknie *Solution Explorer*. Aby go wyświetlić, należy wybrać z menu *Project* polecenie *Show All Files*. Plik *AssemblyInfo.vb* jest umieszczony pod węzłem *My Project*. Oto typowy zestaw znaczników, które można określić w tym pliku:

```
<Assembly: AssemblyVersion("1.0.0.0")>
<Assembly: AssemblyCompany("Prosetech")>
<Assembly: AssemblyDescription("Program testujący obiekt My.Application")>
<Assembly: AssemblyCopyright("(C) Matthew MacDonald")>
<Assembly: AssemblyTrademark("(R) Prosetech")>
<Assembly: AssemblyTitle("Aplikacja testowa")>
<Assembly: AssemblyProduct("Aplikacja testowa")>
```

Nowością w VB 2005 jest możliwość dodawania informacji o aplikacji w specjalnym oknie dialogowym. Aby skorzystać z tej funkcji, należy kliknąć dwukrotnie ikonę My Project w oknie Solution Explorer, przejść na zakładkę Application i kliknąć przycisk Assembly Information.

Wszystkie te informacje zostaną osadzone w skompilowanym podzespole jako metadane.

Teraz można uruchomić testową aplikację. Oto przykładowe wyniki:

```
Parametry wiersza polecenia: /a /b /c

Firma: Prosetech
Opis: Program testujący obiekt My.Application
Lokalizacja: C:\NotatnikVB\2\InformacjeOAplikacji\bin\Debug
```

```
Prawa autorskie: (C) Matthew MacDonald
Znak towarowy: (R) Prosetech
Nazwa: InformacjeOAplikacji
Produkt: Aplikacja testowa
Tytuł: Aplikacja testowa
Wersja: 1.0.0.0
```

A co...

...z uzyskiwaniem bardziej szczegółowych informacji diagnostycznych? Obiekt `My.Computer.Info` ma dwie przydatne właściwości, które dostarczają takich informacji. Właściwość `LoadedAssemblies` to kolekcja wszystkich podzespołów, które są obecnie wczytane (i dostępne dla aplikacji). Można też zbadać ich wersje oraz informacje o wydawcy. Właściwość `StackTrace` zawiera bieżący obraz stosu i pozwala ustalić, którą część kodu obecnie wykonuje program. Jeśli na przykład metoda `Main()` wywołuje metodę `A()`, a ta wywołuje metodę `B()`, na stosie będzie widać te trzy metody — `B()`, `A()` i `Main()` — w odwrotnej kolejności.

Oto kod, który wyświetla te informacje:

```
Console.WriteLine("Wczytane podzespoły:")
For Each Assm As System.Reflection.Assembly In _
  My.Application.Info.LoadedAssemblies
    Console.WriteLine(Assm.GetName().Name)
Next
Console.WriteLine()

Console.WriteLine("Bieżący stos: " & My.Application.Info.StackTrace)
Console.WriteLine()
```

Ściśle typizowane zasoby umożliwiają osadzanie statycznych danych (na przykład obrazów) w skompilowanych podzespołach i zapewniają łatwy dostęp do tych danych z poziomu kodu.

Używanie ściśle typizowanych zasobów

Podzespoły .NET oprócz kodu mogą zawierać **zasoby** — osadzone dane binarne, na przykład obrazy i niezmienne łańcuchy. Choć środowisko .NET obsługuje zasoby już od wersji 1.0, Visual Studio nie oferowało ich zintegrowanej obsługi w czasie projektowania aplikacji. W rezultacie programiści, którzy chcieli zapisać dane obrazu w swojej aplikacji, zwykle dodawali je do kontrolek obsługujących obrazy podczas projektowania aplikacji, takich jak `PictureBox` lub `ImageList`. Kontrolki te automatycznie wstawiały dane obrazu do pliku zasobów aplikacji.

W Visual Studio 2005 znacznie łatwiej jest dodawać informacje do plików zasobu i później je aktualizować. Co najlepsze, można uzyskać dostęp do tych informacji z dowolnego punktu kodu, i to ze ścisłą kontrolą typów.

Jak to zrobić?

Aby utworzyć ściśle typizowany zasób na użytek tego ćwiczenia, należy zacząć od utworzenia nowej aplikacji Windows.

W celu dodania zasobu należy kliknąć dwukrotnie węzeł *My Project* w oknie *Solution Explorer*. Pojawi się okno projektu aplikacji, w którym można skonfigurować wiele różnorodnych ustawień programu. Następnie należy przejść na zakładkę *Resources* i wybrać żądany typ zasobów z listy rozwijanej umieszczonej w lewym górnym oknie (*Strings*, *Images*, *Audio* itd.). Widok łańcuchów przedstawia siatkę danych. Widok obrazów jest nieco inny — domyślnie pokazuje miniatury dołączonych obrazów.

Aby dodać nowy obraz, należy wybrać z listy kategorię *Images*, a następnie kliknąć strzałkę obok przycisku *Add Resource* i wybrać polecenie *Add Existing File*. Należy znaleźć plik obrazu, zaznaczyć go i kliknąć przycisk *OK*. Ci, którzy nie mają pod ręką innego obrazu, mogą wykorzystać jeden spośród plików przechowywanych w katalogu *Windows*, na przykład *winnt256.bmp* (który jest dołączany do większości wersji Windows).

Domyślnie zasób ma tę samą nazwę co plik, ale można ją zmienić po dodaniu zasobu. W tym przykładzie zmienimy nazwę obrazu na `EmbeddedGraphic` (jak pokazano na rysunku 2.2).

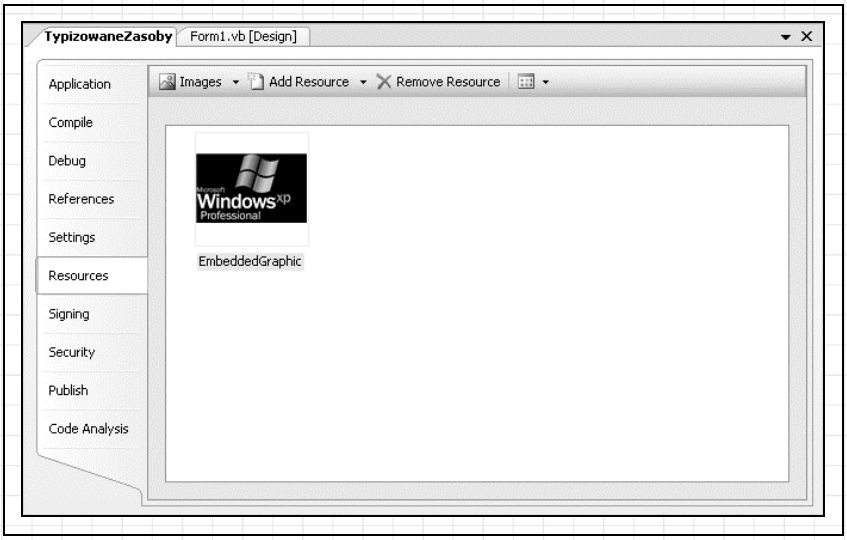
Klasa zasobów jest dodawana do węzła My Project i otrzymuje nazwę Resources.Designer.vb. Aby ją zobaczyć, trzeba wybrać z menu Project polecenie Show All Files. Oczywiście, nie należy modyfikować tego pliku ręcznie.

Korzystanie z zasobów jest bardzo łatwe. Wszystkie zasoby są kompilowane dynamicznie w ściśle typizowaną klasę zasobów, do której można uzyskać dostęp poprzez obiekt `My.Resources`. Aby wypróbować nowo dodany zasób, należy dodać kontrolkę `PictureBox` do formularza Windows (i zachować domyślną nazwę `PictureBox1`). Następnie należy dodać poniższy kod, który wyświetli obraz podczas wczytywania formularza:

```
Private Sub Form1_Load(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles MyBase.Load

    PictureBox1.Image = My.Resources.EmbeddedGraphic

End Sub
```



Rysunek 2.2. Dodawanie obrazu jako ściśle typizowanego zasobu

Po uruchomieniu programu obraz zostanie wyświetlony na formularzu. Aby upewnić się, że obraz jest pobierany z podzespołu, można skompilować aplikację, a następnie usunąć plik obrazu (kod będzie nadal działał bez żadnych problemów).

Inną zaletą zasobów jest to, że można wykorzystać te same obrazy w wielu kontrolkach na różnych formularzach bez dodawania wielu kopii obrazu do jednego pliku.

Kiedy dodajemy zasoby w taki sposób, Visual Studio kopiuje je do podkatalogu *Resources* aplikacji. Katalog ten, wraz z zawartymi w nim zasobami, można obejrzeć w oknie *Solution Explorer*. Podczas kompilowania aplikacji wszystkie zasoby są osadzane w podzespole, ale przechowywanie ich w oddzielnym katalogu ma pewną ważną zaletę — pozwala zaktualizować zasób przez zamianę pliku i ponowne skompilowanie aplikacji. Nie trzeba modyfikować kodu. Jest to niezwykle przydatne, jeśli trzeba jednocześnie zaktualizować wiele obrazów albo innych zasobów.

Zasoby można dołączać do różnych kontrolek za pomocą okna *Properties*. Na przykład po kliknięciu wielokropka (`...`) obok właściwości `Image` kontrolki `PictureBox` pojawi się okno z listą wszystkich obrazów dostępnych w zasobach aplikacji.

A co...

...z kontrolką `ImageList`? Większość programistów Windows zna tę kontrolkę, która grupuje wiele obrazów (zwykle niewielkich map bitowych) na użytek innych kontrolek, takich jak menu, paski narzędzi, drzewa i listy. Kontrolka `ImageList` nie używa typizowanych zasobów, lecz własnej metody serializacji. Choć dostęp do zawartych w niej obrazów można uzyskać zarówno podczas projektowania aplikacji, jak i metodą programistyczną, nie podlega ona kontroli typów.

Używanie ściśle typizowanych ustawień konfiguracyjnych

Tworzenie ustawień konfiguracyjnych w oknie projektu aplikacji.

Aplikacje często używają ustawień konfiguracyjnych, na przykład wskazujących położenie plików, parametry połączenia z bazą danych i preferencje użytkownika. Zamiast kodować te ustawienia „na sztywno" (albo wymyślać własny mechanizm ich przechowywania), w .NET można dodać je do pliku konfiguracyjnego specyficznego dla aplikacji. Dzięki temu można je łatwo modyfikować poprzez edycję pliku tekstowego, bez ponownego kompilowania aplikacji.

W Visual Studio 2005 jest to jeszcze łatwiejsze, ponieważ ustawienia konfiguracyjne są kompilowane w oddzielną klasę i podlegają ścisłej kontroli typów. Oznacza to, że można pobierać ustawienia przy użyciu właściwości, z wykorzystaniem funkcji IntelliSense, zamiast zdawać się na wyszukiwanie tekstowe. Co najlepsze, .NET rozszerza ten model o możliwość używania ustawień specyficznych dla użytkownika w celu śledzenia preferencji i innych informacji. W niniejszym ćwiczeniu zostaną zaprezentowane obie te techniki.

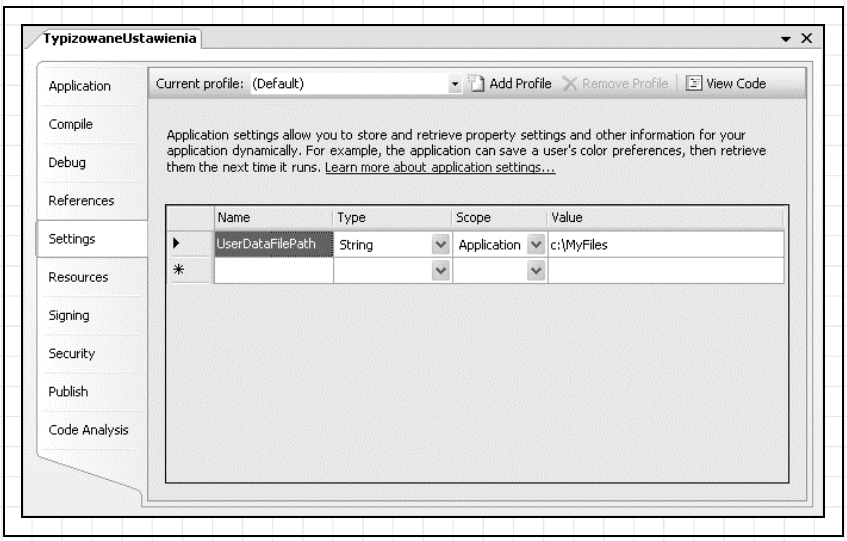
Jak to zrobić?

Każde ustawienie konfiguracyjne jest definiowane przez unikatową nazwę. W poprzednich wersjach .NET można było pobrać wartość ustawienia konfiguracyjnego poprzez wyszukanie jego nazwy w kolekcji. Jeśli jednak nazwa była nieprawidłowa, błąd można było wykryć dopiero po uruchomieniu programu i wystąpieniu wyjątku czasu wykonania.

W aplikacji WWW ustawienia konfiguracyjne są umieszczane w pliku web.config. W innych aplikacjach ustawienia są umieszczane w pliku noszącym tę samą nazwę co aplikacja i uzupełnioną rozszerzeniem .config, na przykład MojaAplikacja.exe.config.

W Visual Studio 2005 sytuacja wygląda znacznie lepiej. Aby dodać nowe ustawienie konfiguracyjne, należy kliknąć dwukrotnie węzeł *My Project* w oknie *Solution Explorer*. Pojawi się okno projektu aplikacji, w którym można skonfigurować wiele różnorodnych ustawień programu. Następnie należy przejść na zakładkę *Settings*, która pokazuje listę niestandardowych ustawień konfiguracyjnych i umożliwia zdefiniowanie nowych ustawień oraz ich wartości.

Aby dodać do aplikacji nowe ustawienie konfiguracyjne, należy wpisać jego nazwę na dole listy. Następnie należy określić typ danych, zasięg i rzeczywistą wartość ustawienia. Aby na przykład dodać ustawienie ze ścieżką do pliku, można użyć nazwy `UserDataFilePath`, typu `String`, zasięgu `Application` (niebawem powiemy więcej na ten temat) oraz wartości `c:\MyFiles`. Ustawienie to pokazano na rysunku 2.3.



Rysunek 2.3. Definiowanie ściśle typizowanego ustawienia konfiguracyjnego

Po dodaniu tego ustawienia Visual Studio .NET wstawi poniższe informacje do pliku konfiguracyjnego aplikacji:

```
<configuration>
  <applicationSettings>
    <TypizowaneUstawienia.Settings>
      <setting name="UserDataFilePath" serializeAs="String">
        <value>c:\MyFiles</value>
      </setting>
```

```
    </TypizowaneUstawienia.Settings>
  </applicationSettings>
</configuration>
```

Jednocześnie „za kulisami" Visual Studio skompiluje klasę, która zawiera informacje o niestandardowym ustawieniu konfiguracyjnym. Teraz z dowolnego miejsca w kodzie będzie można uzyskać dostęp do ustawienia według jego nazwy za pośrednictwem obiektu `My.Settings`. Oto kod, który pobiera wartość ustawienia o nazwie `UserDataFilePath`:

```
Dim path As String
path = My.Settings.UserDataFilePath
```

W .NET 2.0 ustawienia konfiguracyjne nie muszą być łańcuchami. Można używać innych serializowalnych typów danych, w tym liczb całkowitych i dziesiętnych, dat i czasów (wystarczy wybrać odpowiedni typ z listy rozwijanej *Type*). Wartości te są przekształcane w tekst podczas zapisu w pliku konfiguracyjnym, ale można je pobrać za pośrednictwem obiektu `My.Settings` w odpowiednim formacie, bez analizy składniowej!

Klasa ustawień konfiguracyjnych jest dodawana do węzła My Project i nosi nazwę Settings.Designer. Aby ją obejrzeć, należy wybrać z menu Project polecenie Show All Files.

A co...

...z aktualizowaniem ustawień? W przykładzie `UserFileDataPath` wykorzystano ustawienie **o zasięgu aplikacji**, które można odczytywać w czasie działania programu, ale nie można go modyfikować. Jeśli konieczna jest aktualizacja ustawienia o zasięgu aplikacji, trzeba ręcznie zmodyfikować plik konfiguracyjny (albo użyć listy ustawień w Visual Studio).

Alternatywą jest utworzenie ustawień **o zasięgu użytkownika**. W tym celu wystarczy wybrać pozycję `User` z listy rozwijanej *Scope* na liście ustawień. Jeśli ustawienie ma zasięg użytkownika, wartość ustawiona w Visual Studio jest zapisywana jako wartość domyślna w pliku konfiguracyjnym, w katalogu aplikacji. Jednakże zmiana tych ustawień powoduje utworzenie nowego pliku *user.config* i zapisanie go w katalogu specyficznym dla bieżącego użytkownika (o nazwie *c:\Documents and Settings\(NazwaUżytkownika)\Ustawienia lokalne\Dane aplikacji\(Nazwa ➥Firmy)\(UnikatowyKatalog)\(WersjaAplikacji)*).

W razie użycia ustawień specyficznych dla użytkownika trzeba pamiętać o wywołaniu metody `My.Settings.Save()` w celu zapisania zmian. W przeciwnym razie zmiany będą obowiązywać tylko do momentu za-

mknięcia aplikacji. Zwykle metodę `My.Settings.Save()` wywołuje się pod koniec działania aplikacji.

Aby wypróbować ustawienia o zasięgu użytkownika, należy zmienić zasięg ustawienia `UserDataFilePath` na *User*. Następnie należy utworzyć formularz z polem tekstowym (o nazwie `txtFilePath`) i dwoma przyciskami: jednym do pobierania ustawienia (`cmdRefresh`), a drugim do aktualizowania go (`cmdUpdate`). Oto odpowiednie procedury obsługi zdarzeń:

```
Private Sub cmdRefresh_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles cmdRefresh.Click
    txtFilePath.Text = My.Settings.UserDataFilePath
End Sub

Private Sub cmdUpdate_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles cmdUpdate.Click
    My.Settings.UserDataFilePath = txtFilePath.Text
End Sub
```

Aby zmiany obowiązywały podczas następnego uruchomienia aplikacji, konieczne jest utworzenie lub zaktualizowanie pliku *user.config* podczas zamykania formularza:

```
Private Sub Form1_FormClosed(ByVal sender As Object, _
  ByVal e As System.Windows.Forms.FormClosedEventArgs) _
  Handles Me.FormClosed
    My.Settings.Save()
End Sub
```

To już cały kod, który trzeba dodać do formularza. Teraz można uruchomić aplikację i sprawdzić, jak działa pobieranie bieżącego ustawienia i zapisywanie nowego. Następnie można poszukać pliku *user.config*, który zawiera zmienione ustawienia bieżącego użytkownika.

Tworzenie klas generycznych

Jak utworzyć klasę na tyle elastyczną, aby mogła działać z dowolnym typem obiektu, a jednocześnie w każdej sytuacji móc ograniczać przyjmowane przez nią obiekty? Rozwiązaniem są konstrukcje generyczne języka VB.

Programiści często stają przed trudnym wyborem. Z jednej strony, bardzo ważne jest tworzenie rozwiązań uniwersalnych, które można wielokrotnie wykorzystywać w różnych sytuacjach. Dlaczego na przykład mielibyśmy tworzyć klasę `CustomerCollection`, która przyjmuje tylko obiekty typu `Customer`, skoro możemy zaprojektować klasę `Collection` zapewniającą dostęp do obiektów dowolnego typu? Z drugiej strony kwestie wydajności i typizacji sprawiają, że rozwiązania uniwersalne są mniej pożądane. Jeśli na przykład użyjemy generycznej klasy .NET

Collection do przechowywania obiektów Customer, to skąd mamy wiedzieć, czy przypadkiem nie wstawimy do kolekcji obiektu innego typu, co później spowoduje trudne do wykrycia problemy?

Visual Basic 2005 i .NET 2.0 zapewniają rozwiązanie nazywane **klasami generycznymi**. Są to klasy **parametryzowane według typu**. Innymi słowy, umożliwiają one zbudowanie szablonu obsługującego dowolny typ. Podczas tworzenia instancji klasy określamy, jakiego typu chcemy używać, i od tego momentu jesteśmy „przywiązani" do wybranego typu.

Jak to zrobić?

Przykładem, który pokazuje przydatność konstrukcji generycznych, jest klasa System.Collections.ArrayList. ArrayList — uniwersalna, automatycznie skalowana kolekcja. Może ona przechowywać zwykłe obiekty .NET albo obiekty zdefiniowane przez użytkownika. Aby było to możliwe, klasa ArrayList traktuje wszystkie elementy jak bazowy typ Object.

Problem polega na tym, że nie sposób narzucić żadnych ograniczeń na działanie klasy ArrayList. Jeśli, na przykład, chcemy użyć klasy ArrayList do przechowywania kolekcji obiektów Customer, nie możemy upewnić się, że jakiś fragment kodu nie wstawi do niej łańcuchów, liczb całkowitych albo jakichś innych obiektów, co później spowoduje problemy. Z tej przyczyny programiści często tworzą własne, ściśle typizowane klasy kolekcji — w rzeczywistości biblioteka .NET jest wypełniona dziesiątkami takich klas.

Konstrukcje generyczne rozwiązują ten problem. Za pomocą słowa kluczowego Of można zadeklarować klasę, która działa z dowolnym typem danych:

```
Public Class GenericList(Of ItemType)
    ' (tutaj kod klasy)
End Class
```

W tym przypadku tworzymy nową klasę o nazwie GenericList, która może działać z obiektami dowolnego typu. Kod kliencki musi jednak określić, jakiego typu będzie używał. W kodzie klasy odwołujemy się do tego typu za pomocą słowa ItemType. Oczywiście, ItemType nie jest na-

prawdę typem — to tylko parametr zastępujący typ, który zostanie wybrany podczas tworzenia instancji obiektu GenericList.

Na listingu 2.2 pokazano kompletny kod prostej klasy listy, która kontroluje zgodność typów.

Listing 2.2. Kolekcja kontrolująca zgodność typów

```
Public Class GenericList(Of ItemType)
    Inherits CollectionBase

    Public Function Add(ByVal value As ItemType) As Integer
        Return List.Add(value)
    End Function

    Public Sub Remove(ByVal value As ItemType)
        List.Remove(value)
    End Sub

    Public ReadOnly Property Item(ByVal index As Integer) As ItemType
        Get
            ' Odpowiedni element jest pobierany z obiektu List i jawnie
            ' rzutowany na odpowiedni typ, a następnie zwracany
            Return CType(List.Item(index), ItemType)
        End Get
    End Property
End Class
```

Klasa GenericList jest nakładką na zwykłą klasę ArrayList, która jest dostępna za pośrednictwem właściwości List bazowej klasy CollectionBase. Klasa GenericList działa jednak inaczej niż ArrayList, ponieważ oferuje ściśle typizowane metody Add() i Remove(), w których wykorzystano zastępczy parametr ItemType.

Oto przykład użycia klasy GenericList do utworzenia kolekcji ArrayList, która obsługuje tylko łańcuchy:

```
' Tworzymy instancję GenericList i wybieramy typ (w tym przypadku String)
Dim List As New GenericList(Of String)

' Dodajemy dwa łańcuchy
List.Add("niebieski")
List.Add("zielony")

' Następna instrukcja nie zadziała, ponieważ dodaje do kolekcji błędny typ.
' Nie można automatycznie przekształcić identyfikatora GUID w łańcuch.
' Prawdę mówiąc, wiersz ten nie zostanie nigdy wykonany, ponieważ kompilator
' wykryje problem i odmówi zbudowania aplikacji.
List.Add(Guid.NewGuid())
```

Nie ma żadnych ograniczeń, jeśli chodzi o parametryzowanie klasy. W przykładzie GenericList jest tylko jeden parametr. Można jednak łatwo utworzyć klasę, która działa z dwoma lub trzema typami obiektów i pozwala sparametryzować każdy z nich. Aby użyć tej techniki, należy oddzielić każdy typ przecinkiem (między nawiasami na początku deklaracji klasy).

Rozważmy na przykład poniższą klasę GenericHashTable, która pozwala zdefiniować typ elementów kolekcji (ItemType), a także typ kluczy używanych do indeksowania tych elementów (KeyType):

```
Public Class GenericHashTable(Of ItemType, KeyType)
    Inherits DictionaryBase
    ' (tutaj kod klasy)
End Class
```

Inną ważną cechą konstrukcji generycznych jest możliwość nakładania **ograniczeń** na parametry. Ograniczenia zawężają zakres typów obsługiwanych przez klasę generyczną. Przypuśćmy, że chcemy utworzyć klasę obsługującą wyłącznie typy implementujące określony interfejs. W tym celu najpierw musimy zadeklarować typ lub typy akceptowane przez klasę, a następnie użyć słowa kluczowego As, aby określić klasę bazową, z której typ musi się wywodzić, albo interfejs, który musi implementować.

Oto przykład, który ogranicza elementy przechowywane w kolekcji GenericList do typów serializowalnych. Byłoby to użyteczne, gdybyśmy chcieli dodać do klasy GenericList metodę wymagającą serializacji, na przykład taką, która zapisywałaby wszystkie elementy listy w strumieniu:

```
Public Class SerializableList(Of ItemType As ISerializable)
   Inherits CollectionBase
    ' (tutaj kod klasy)
End Class
```

A oto kolekcja, która może zawierać obiekt dowolnego typu, pod warunkiem że wywodzi się on z klasy System.Windows.Forms.Control. Rezultatem jest kolekcja ograniczona do kontrolek, przypominająca tę eksponowaną przez właściwość Forms.Controls okna:

```
Public Class SerializableList(Of ItemType As Control)
   Inherits CollectionBase
    ' (tutaj kod klasy)
End Class
```

Czasem klasa generyczna musi mieć możliwość tworzenia sparametryzowanych obiektów. Na przykład w klasie `GenericList` konieczne może być tworzenie instancji elementu, który ma być dodany do kolekcji. W takim przypadku trzeba użyć ograniczenia `New`. Ograniczenie `New` zezwala tylko na typy, które mają publiczny, bezargumentowy konstruktor i nie są oznaczone jako `MustInherit`. Gwarantuje to, że kod będzie mógł tworzyć instancje typu. Oto kolekcja, która narzuca ograniczenie `New`:

```
Public Class GenericList(Of ItemType As New)
    Inherits CollectionBase
    ' (tutaj kod klasy)
End Class
```

Warto też zauważyć, że można definiować dowolnie wiele ograniczeń. W tym celu należy umieścić ich listę w nawiasie klamrowym, jak w poniższym przykładzie:

```
Public Class GenericList(Of ItemType As {ISerializable, New})
    Inherits CollectionBase
    ' (tutaj kod klasy)
End Class
```

Konstrukcje generyczne są wbudowane w środowisko CLR. Oznacza to, że są one obsługiwane we wszystkich językach .NET, łącznie z C#.

Ograniczenia są wymuszone przez kompilator, więc naruszenie ograniczenia podczas korzystania z klasy generycznej uniemożliwi skompilowanie aplikacji.

A co...

...z innymi konstrukcjami generycznymi? Typów sparametryzowanych można używać nie tylko w klasach, ale również w strukturach, interfejsach, delegacjach, a nawet metodach. Więcej informacji jest dostępnych pod hasłem „generics" w pomocy MSDN. Przykłady użycia zaawansowanych konstrukcji generycznych można znaleźć w artykule przeglądowym Microsoftu pod adresem *http://www.msdn.net/library/en-us/dnvs05/html/vb2005_generics.asp*.

Nawiasem mówiąc, twórcy .NET Framework zdawali sobie sprawę z przydatności kolekcji generycznych, więc utworzyli kilka gotowych kolekcji na użytek programistów. Znajdują się one w przestrzeni nazw `System.Collections.Generic`. Są to między innymi:

- `List` (prosta kolekcja podobna do przykładowej klasy `GenericList`);

- `Dictionary` (kolekcja typu „nazwa-wartość", w której każdy element jest indeksowany przy użyciu klucza);
- `LinkedList` (połączona lista, w której każdy element wskazuje następny);
- `Queue` (kolekcja typu „pierwszy na wejściu, pierwszy na wyjściu");
- `Stack` (kolekcja typu „ostatni na wejściu, pierwszy na wyjściu");
- `SortedList` (kolekcja typu „nazwa-wartość", która stale pozostaje posortowana).

Większość spośród tych kolekcji powiela jedną z klas w przestrzeni nazw `System.Collections`. Stare kolekcje pozostawiono w celu zachowania zgodności.

Używanie wartości pustych w prostych typach danych

Czasem konieczne jest reprezentowanie danych, które mogą być nieobecne. Umożliwiają to nowe typy danych VB .NET.

Dzięki obsłudze konstrukcji generycznych .NET Framework może zaoferować kilka nowych funkcji. Jedną z nich — generyczne, ściśle typizowane kolekcje — opisano w poprzednim ćwiczeniu, „Tworzenie klas generycznych". Konstrukcje generyczne rozwiązują również inne często spotykane problemy, w tym obsługę wartości pustych w prostych typach danych.

Jak to zrobić?

Wartość pusta (w Visual Basicu identyfikowana przez słowo kluczowe `Nothing`) jest specjalnym znacznikiem, który wskazuje, że dane są nieobecne. Większość programistów zna puste referencje do obiektów, które wskazują, że obiekt zdefiniowano, ale jeszcze go nie utworzono. Na przykład w poniższym kodzie obiekt `FileStream` zawiera pustą referencję, ponieważ nie został jeszcze utworzony za pomocą słowa kluczowego `New`:

```
Dim fs As FileStream
If fs Is Nothing
    ' Ten warunek jest zawsze spełniony, ponieważ
    ' nie utworzono jeszcze obiektu FileStream
    Console.WriteLine("Obiekt zawiera pustą referencję")
End If
```

Podstawowe typy danych, takie jak liczby całkowite i łańcuchy, **nie mogą** zawierać wartości pustych. Zmienne liczbowe są automatycznie inicjalizowane wartością 0, zmienne logiczne — wartością `False`, zmienne łańcuchowe — łańcuchem pustym (`""`). W rzeczywistości nawet jawne ustawienie zmiennej typu prostego na `Nothing` spowoduje automatyczne przekształcenie wartości pustej (w `0`, `False` lub `""`), jak pokazuje poniższy kod:

```
Dim j As Integer = Nothing
If j = 0 Then
    ' Ten warunek jest zawsze spełniony, ponieważ w przypadku liczb
      całkowitych
    ' zachodzi automatyczna konwersja między wartością Nothing a 0
    Console.WriteLine("Zmienna całkowita nie może zawierać wartości
pustej, j = " & j)
End If
```

Czasem powoduje to problemy, ponieważ nie da się odróżnić wartości zerowej od wartości, której nigdy nie podano. Wyobraźmy sobie, że piszemy kod, który pobiera z pliku tekstowego liczbę zamówień złożonych przez klienta. Później badamy tę wartość. Problem występuje wtedy, gdy jest ona równa zeru. Nie możemy stwierdzić, czy dane są prawidłowe (użytkownik nie złożył żadnych zamówień), czy może brakuje nam jakichś informacji (nie udało się pobrać wartości albo bieżący użytkownik nie jest zarejestrowanym klientem).

Dzięki obsłudze konstrukcji generycznych .NET 2.0 oferuje rozwiązanie — klasę `System.Nullable`, która może „owijać" dowolny typ danych. Podczas tworzenia instancji klasy `Nullable` określamy typ danych. Jeśli nie ustawimy wartości, instancja ta będzie zawierać pustą referencję. Możemy sprawdzić, czy tak jest w istocie, wywołując metodę `Nullable.HasType()`, i pobrać bazowy obiekt za pośrednictwem właściwości `Nullable.Value`.

Oto przykładowy kod tworzący zmienną całkowitą, która może zawierać wartość pustą:

```
Dim i As Nullable(Of Integer)
If Not i.HasValue Then
    ' Warunek jest spełniony, ponieważ zmiennej nie przypisano wartości
    Console.WriteLine("Zmienna i zawiera wartość pustą")
End If
```

```
' Przypisujemy wartość. Zauważmy, że trzeba ją przypisać bezpośrednio
' zmiennej i,
' a nie i.Value. Właściwość i.Value jest przeznaczona tylko do odczytu i zawsze
' odzwierciedla bieżący obiekt, jeśli nie jest to Nothing.
i = 100
If i.HasValue Then
    ' Warunek jest spełniony, ponieważ obecnie zmienna zawiera wartość (100)
    Console.WriteLine("Zmienna całkowita i = " & i.Value)
End If
```

A co...

...z używaniem klasy `Nullable` w połączeniu z typami referencyjnymi? Choć nie jest to konieczne (ponieważ typy referencyjne mogą zawierać pustą referencję), to ma pewne zalety. Możemy używać nieco czytelniejszej metody `HasValue()` zamiast testować wartość `Nothing`. Co najlepsze, możemy łatwo dokonać tej zmiany, ponieważ klasa `Nullable` pozwala na niejawną konwersję między `Nullable` a typem bazowym.

Więcej informacji

Aby dowiedzieć się więcej na temat klasy `Nullable` i jej implementacji, należy zajrzeć pod hasło „Nullable class" w pomocy MSDN.

Używanie operatorów w połączeniu z własnymi obiektami

Kto ma dość posługiwania się toporną składnią w rodzaju ObjA.Subtract(ObjB) w celu wykonania prostych operacji na własnych obiektach? Dzięki przeciążaniu operatorów można manipulować takimi obiektami równie łatwo jak zwykłymi liczbami.

Każdy programista VB dobrze zna operatory arytmetyczne służące do dodawania (+), odejmowania (-), dzielenia (/) i mnożenia (*). Zwykle operatory te są zarezerwowane dla liczbowych typów .NET i nie można używać ich w połączeniu z innymi obiektami. Jednakże w VB .NET 2.0 można budować obiekty, które obsługują wszystkie te operatory (a także operatory używane do operacji logicznych i niejawnej konwersji typu). Technika ta nie ma sensu w przypadku obiektów biznesowych, ale jest niezwykle przydatna do modelowania struktur matematycznych, takich jak wektory, macierze, liczby zespolone albo — jak w poniższym przykładzie — ułamki.

Jak to zrobić?

Aby przeciążyć operator w Visual Basicu 2005, trzeba utworzyć specjalną metodę operatorową w klasie (lub strukturze). Metodę tę deklaruje się przy użyciu słów kluczowych `Public Shared Operator`, po których następuje symbol operatora (na przykład +).

WSKAZÓWKA

Przeciążanie operatora to po prostu definiowanie, co robi operator, kiedy używamy go w połączeniu z konkretnym typem obiektu. Innymi słowy, kiedy przeciążamy operator + klasy `Fraction`, informujemy .NET, co należy zrobić, gdy kod dodaje do siebie dwa obiekty `Fraction`.

Oto przykładowa metoda operatorowa, która dodaje obsługę operatora dodawania (+):

```
Public Shared Operator(ObjA As MyClass, objB As MyClass) As MyClass
    ' (tutaj kody klasy)
End Operator
```

Każda metoda operatorowa przyjmuje dwa parametry, które reprezentują wartości po obu stronach operatora. W zależności od operatora i klasy kolejność może mieć znaczenie (jak w przypadku dzielenia).

Kiedy zdefiniujemy operator, kompilator VB będzie wywoływał nasz kod podczas wykonywania każdej instrukcji, która używa operatora na obiekcie danej klasy. Na przykład kompilator przekształci kod:

```
ObjC = ObjA + ObjB
```

do następującej postaci:

```
ObjC = MyClass.Operator+(ObjA, ObjB)
```

Na listingu 2.3 pokazano, jak przeciążyć operatory arytmetyczne Visual Basica na użytek klasy `Fraction`. Każdy obiekt `Fraction` składa się z dwóch części: licznika i mianownika (tzw. „góry" i „dołu" ułamka). Kod klasy `Fraction` przeciąża operatory +, -, * oraz /, umożliwiając wykonywanie obliczeń ułamkowych bez przekształcania liczb w wartości dziesiętne, a zatem bez utraty precyzji.

Listing 2.3. Przeciążanie operatorów arytmetycznych w klasie Fraction

```
Public Structure Fraction

    ' Dwie części ułamka
    Public Denominator As Integer
    Public Numerator As Integer

    Public Sub New(ByVal numerator As Integer, ByVal denominator As Integer)
        Me.Numerator = numerator
        Me.Denominator = denominator
    End Sub

    Public Shared Operator +(ByVal x As Fraction, ByVal y As Fraction) _
      As Fraction
        Return Normalize(x.Numerator * y.Denominator + _
          y.Numerator * x.Denominator, x.Denominator * y.Denominator)
    End Operator

    Public Shared Operator -(ByVal x As Fraction, ByVal y As Fraction) _
      As Fraction
        Return Normalize(x.Numerator * y.Denominator - _
          y.Numerator * x.Denominator, x.Denominator * y.Denominator)
    End Operator

    Public Shared Operator *(ByVal x As Fraction, ByVal y As Fraction) _
      As Fraction
        Return Normalize(x.Numerator * y.Numerator, _
          x.Denominator * y.Denominator)
    End Operator

    Public Shared Operator /(ByVal x As Fraction, ByVal y As Fraction) _
      As Fraction
        Return Normalize(x.Numerator * y.Denominator, _
          x.Denominator * y.Numerator)
    End Operator

    ' Redukcja ułamka
    Private Shared Function Normalize(ByVal numerator As Integer, _
      ByVal denominator As Integer) As Fraction
        If (numerator <> 0) And (denominator <> 0) Then
            ' Poprawianie znaków
            If denominator < 0 Then
                denominator *= -1
                numerator *= -1
            End If

            Dim divisor As Integer = GCD(numerator, denominator)
            numerator \= divisor
            denominator \= divisor
        End If

        Return New Fraction(numerator, denominator)
    End Function
```

```
        ' Zwracanie największego wspólnego dzielnika przy użyciu algorytmu
        ' Euklidesa
        Private Shared Function GCD(ByVal x As Integer, ByVal y As Integer) _
          As Integer
            Dim temp As Integer

            x = Math.Abs(x)
            y = Math.Abs(y)
            Do While (y <> 0)
                temp = x Mod y
                x = y
                y = temp
            Loop

            Return x
        End Function

        ' Przekształcanie ułamka w postać dziesiętną
        Public Function GetDouble() As Double
            Return CType(Me.Numerator, Double) / _
              CType(Me.Denominator, Double)
        End Function

        ' Pobieranie łańcucha reprezentującego ułamek
        Public Overrides Function ToString() As String
            Return Me.Numerator.ToString & "/" & Me.Denominator.ToString
        End Function

    End Structure
```

Aplikacja konsoli pokazana na listingu 2.4 przeprowadza szybki test klasy `Fraction`. Dzięki przeciążaniu operatorów liczba pozostaje w postaci ułamkowej i nie dochodzi do utraty precyzji.

Listing 2.4. Test klasy Fraction

```
    Module FractionTest

        Sub Main()
            Dim f1 As New Fraction(2, 3)
            Dim f2 As New Fraction(1, 4)

            Console.WriteLine("f1 = " & f1.ToString())
            Console.WriteLine("f2 = " & f2.ToString())

            Dim f3 As Fraction
            f3 = f1 + f2     ' f3 obecnie wynosi 11/12
            Console.WriteLine("f1 + f2 = " & f3.ToString())

            f3 = f1 / f2     ' f3 obecnie wynosi 8/3
            Console.WriteLine("f1 / f2 = " & f3.ToString())
```

```
        f3 = f1 - f2      ' f3 obecnie wynosi 5/12
        Console.WriteLine("f1 - f2 = " & f3.ToString())

        f3 = f1 * f2      ' f3 obecnie wynosi 1/6
        Console.WriteLine("f1 * f2 = " & f3.ToString())
        Console.ReadLine()
    End Sub

End Module
```

Po uruchomieniu tej aplikacji pojawią się następujące wyniki:

```
f1 = 2/3
f2 = 1/4
f1 + f2 = 11/12
f1 / f2 = 8/3
f1 - f2 = 5/12
f1 * f2 = 1/6
```

Parametry i wartość zwrotna metody operatorowej zwykle są tego samego typu. Można jednak utworzyć kilka wersji metody operatorowej, aby używać obiektów w wyrażeniach z różnymi typami.

A co…

…z przeciążaniem operatorów w innych typach danych? Istnieją klasy, które są naturalnymi kandydatami do przeciążania operatorów. Oto kilka przykładów:

- klasy matematyczne, które modelują wektory, macierze, liczby zespolone lub tensory;
- klasy finansowe, które zaokrąglają obliczenia do najbliższego grosza i obsługują różne typy walut;
- klasy miar, które używają jednostek nieregularnych, na przykład cali i stóp.

Więcej informacji

Aby dowiedzieć się więcej o składni przeciążania operatorów oraz o wszystkich operatorach, które można przeciążać, należy zajrzeć pod hasło indeksu „Operator procedures” w pomocy MSDN.

Niektóre klasy są tak duże, że przechowywanie ich definicji w jednym pliku jest niewygodne. Za pomocą nowego słowa kluczowego Partial można podzielić klasę na kilka oddzielnych plików.

Dzielenie klasy na wiele plików

Jeśli otworzymy klasę .NET 2.0 Windows Forms, zauważymy, że nie ma w niej automatycznie wygenerowanego kodu! Aby zrozumieć, gdzie podział się ten kod, trzeba zapoznać się z funkcją **klas częściowych**, która umożliwia dzielenie klasy na kilka części.

Jak to zrobić?

Za pomocą słowa kluczowego `Partial` można podzielić klasę na dowolną liczbę części. Wystarczy zdefiniować tę samą klasę w więcej niż jednym miejscu. Oto przykład, w którym klasę `SampleClass` zdefiniowano w dwóch częściach:

```
Partial Public Class SampleClass
    Public Sub MethodA()
        Console.WriteLine("Wywołano metodę A")
    End Sub
End Class

Partial Public Class SampleClass
    Public Sub MethodB()
        Console.WriteLine("Wywołano metodę B")
    End Sub
End Class
```

W tym przykładzie deklaracje znajdują się w tym samym pliku, jedna za drugą. Nic jednak nie stoi na przeszkodzie, aby umieścić dwie części klasy `SampleClass` w różnych plikach kodu źródłowego należących do tego samego projektu (jedyne ograniczenie polega na tym, że nie można zdefiniować dwóch części klasy w oddzielnych podzespołach albo różnych przestrzeniach nazw).

Podczas kompilowania aplikacji zawierającej powyższy kod Visual Studio wyszuka każdą część `SampleClass` i połączy ją w kompletną klasę z dwoma metodami, `MethodA()` i `MethodB()`. Można używać obu tych metod:

```
Dim Obj As New SampleClass()
Obj.MethodA()
Obj.MethodB()
```

Klasy częściowe nie pomagają w rozwiązywaniu problemów programistycznych, ale przydają się do dzielenia bardzo dużych, nieporęcznych klas. Oczywiście, sama obecność dużych klas w aplikacji może świadczyć

o tym, że programista źle rozłożył problem na czynniki, a w takim przypadku lepiej jest podzielić kod na odrębne, a nie częściowe klasy. Jednym z kluczowych zastosowań klas częściowych w .NET jest ukrywanie kodu generowanego automatycznie przez Visual Studio, który w starszych wersjach był widoczny, co przeszkadzało niektórym programistom VB.

Na przykład podczas budowania formularza .NET w Visual Basicu 2005 kod obsługi zdarzeń jest umieszczany w pliku kodu źródłowego formularza, ale kod, który tworzy poszczególne kontrolki i łączy je z procedurami obsługi zdarzeń, jest niewidoczny. Aby go zobaczyć, trzeba wybrać z menu *Project* polecenie *Show All Files*. Plik z brakującą częścią klasy pojawi się wówczas w oknie *Solution Explorer*. Jeśli formularz nosi nazwę Form1, to składa się na niego plik *Form1.vb*, który zawiera kod pisany przez programistę, oraz plik *Form1.Designer.vb*, który zawiera automatycznie wygenerowaną część.

A co…

…z używaniem słowa kluczowego `Partial` w połączeniu ze strukturami? To działa, ale nie można tworzyć częściowych interfejsów, wyliczeń ani żadnych innych konstrukcji programistycznych .NET.

Więcej informacji

Więcej informacji o klasach częściowych można znaleźć pod hasłem indeksu „Partial keyword" w pomocy MSDN.

Rozszerzanie przestrzeni nazw My

Niektórzy tak często używają obiektów My, że chcieliby dostosować je do własnych potrzeb. VB 2005 umożliwia dołączenie własnych klas do przestrzeni nazw My.

Obiekty `My` nie są zdefiniowane w jednym miejscu. Niektóre z nich pochodzą od klas zdefiniowanych w przestrzeni nazw `Microsoft.VisualBasic.MyServices`, a inne są generowane automatycznie w miarę dodawania formularzy, usług WWW, ustawień konfiguracyjnych i osadzonych zasobów do projektu aplikacji. Programista może jednak rozszerzyć przestrzeń nazw `My` o własne składniki (na przykład przydatne obliczenia oraz zadania specyficzne dla aplikacji).

Jak to zrobić?

Aby dołączyć nową klasę do hierarchii obiektów `My`, wystarczy użyć bloku `Namespace` z nazwą `My`. Na przykład poniższy kod definiuje nową klasę `BusinessFunctions`, która zawiera specyficzne dla firmy funkcje przeznaczone do generowania identyfikatorów (poprzez połączenie nazwy klienta z identyfikatorem GUID):

```
Namespace My

    Public Class BusinessFunctions
        Public Shared Function GenerateNewCustomerID( _
          ByVal name As String) As String
            Return name & "_" & Guid.NewGuid.ToString()
        End Function
    End Class

End Namespace
```

Po utworzeniu obiektu `BusinessFunctions` można używać go w aplikacji tak samo jak dowolnego innego obiektu `My`. Na przykład poniższa instrukcja wyświetla nowy identyfikator klienta:

```
Console.WriteLine(My.BusinessFunctions.GenerateNewCustomerID("matthew"))
```

Zwróćmy uwagę, że nowe klasy `My` muszą używać współdzielonych metod i właściwości, ponieważ obiekty `My` nie są konkretyzowane automatycznie. Gdybyśmy użyli zwykłych składowych instancyjnych, musielibyśmy sami tworzyć obiekt `My` i nie moglibyśmy manipulować nim przy użyciu tej samej składni. Innym rozwiązaniem jest utworzenie modułu w przestrzeni nazw `My`, ponieważ wszystkie metody i właściwości w module są współdzielone.

Składowe współdzielone to składowe, które są zawsze dostępne za pośrednictwem nazwy klasy, nawet jeśli obiekt nie został utworzony. W razie użycia zmiennej współdzielonej istnieje tylko jedna kopia tej zmiennej, globalna dla całej aplikacji.

Można również rozszerzać niektóre spośród istniejących obiektów `My` dzięki klasom częściowym. W ten sposób można na przykład dodać nowe informacje do obiektu `My.Computer` albo nowe procedury do obiektu `My.Application`. Wymaga to nieco innego podejścia. Właściwość `My.Computer` eksponuje instancję obiektu `MyComputer`. Właściwość `My.Application` eksponuje instancję obiektu `MyApplication`. Aby więc rozszerzyć którąś z tych klas, trzeba utworzyć klasę częściową o odpowiedniej nazwie i dodać do niej odpowiednie składowe. Należy również zadeklarować tę klasę przy użyciu słowa kluczowego `Friend`, aby dopasować ją do istniejącej klasy.

Oto przykład rozszerzania obiektu `My.Application` o metodę, która sprawdza, czy dostępna jest zaktualizowana wersja programu:

```
Namespace My

    Partial Friend Class MyApplication
        Public Function IsNewVersionAvailable() As Boolean
            ' Zwykle numer najnowszej wersji aplikacji byłby odczytywany
            ' z usługi WWW albo jakiegoś innego zasobu.
            ' W tym przykładzie jest zakodowany "na sztywno"
            Dim LatestVersion As New Version(1, 2, 1, 1)
            Return Application.Info.Version.CompareTo(LatestVersion)

        End Function
    End Class
```

Teraz możemy wykorzystać tę metodę:

```
If My.Application.IsNewVersionAvailable()
    Console.WriteLine("Dostępna jest nowsza wersja")
Else
    Console.WriteLine("To jest najnowsza wersja")
End If
```

A co...

...z używaniem rozszerzeń `My` w wielu aplikacjach? Klasy `My` możemy traktować dokładnie tak samo, jak inne przydatne klasy, które chcemy ponownie wykorzystać w innej aplikacji. Innymi słowy, możemy utworzyć projekt biblioteki klas, dodać do niego kilka rozszerzeń `My` i skompilować go do postaci biblioteki DLL. Następnie możemy odwoływać się do tej biblioteki DLL w innych aplikacjach.

Oczywiście, pomimo tego, co mogliby twierdzić entuzjaści Microsoftu, taki sposób rozszerzania przestrzeni nazw `My` ma dwa potencjalnie niebezpieczne aspekty:

- Trudniejsze staje się wykorzystanie komponentu w innych językach. Na przykład język C# nie obsługuje przestrzeni nazw `My`. Choć można używać niestandardowego obiektu `My` w aplikacji C#, jest to dużo mniej wygodne.
- Używanie przestrzeni nazw `My` oznacza rezygnację z jednej spośród największych zalet przestrzeni nazw — unikania konfliktów nazewniczych. Rozważmy na przykład dwie firmy, które tworzą komponenty przeznaczone do rejestrowania zdarzeń. Gdyby firmy uży-

wały zalecanego standardu nazewniczego przestrzeni nazw .NET (`NazwaFirmy.NazwaAplikacji.NazwaKlasy`), komponenty najprawdopodobniej nie miałyby takich samych w pełni kwalifikowanych nazw; jeden mógłby nazywać się `Acme.SuperLogger.Logger`, a drugi — `ComponentTech.LogMagic.Logger`. Gdyby jednak oba komponenty rozszerzały obiekt `My`, mogłyby używać takiej samej nazwy (na przykład `My.Application.Logger`). W rezultacie nie można byłoby wykorzystać ich obu w tej samej aplikacji.

Nowe słowo kluczowe Continue Visual Basica umożliwia szybkie wyjście ze skomplikowanego bloku kodu i przejście do następnej iteracji pętli.

Przechodzenie do następnej iteracji pętli

Visual Basic oferuje kilka instrukcji **kontroli przepływu**, które pozwalają sterować kolejnością wykonywania kodu. Można na przykład użyć słowa kluczowego `Return`, aby wyjść z funkcji, albo słowa `Exit`, aby zakończyć pętlę. Jednakże przed wersją VB 2005 nie było sposobu przejścia do następnej iteracji pętli.

Jak to zrobić?

Instrukcja `Continue` to jeden z tych mechanizmów językowych, które początkowo wydają się mało istotne, ale z czasem okazują się bardzo przydatne. Instrukcja `Continue` ma trzy wersje: `Continue For`, `Continue Do` i `Continue While`, których używa się z różnymi typami pętli (`For ... Next`, `Do ... Loop` oraz `While ... End While`).

Aby zrozumieć, jak działa instrukcja `Continue`, przeanalizujmy poniższy kod:

```
For i = 1 To 1000
    If i Mod 5 = 0 Then
        ' (kod zadania A)
        Continue For
    End If
    ' (kod zadania B)
Next
```

Pętla ta wykonuje się 1000 razy, stopniowo zwiększając wartość licznika `i`. Kiedy licznik `i` jest podzielny przez 5, wykonywany jest kod zadania A. Następnie instrukcja `Continue For` powoduje zwiększenie licznika i wznowienie wykonywania od początku pętli, z pominięciem kodu zadania B.

W tym przykładzie instrukcja `Continue` nie jest właściwie potrzebna, ponieważ można łatwo przepisać kod w następujący sposób:

```
For i = 1 To 1000
    If i Mod 5 = 0 Then
        ' (kod zadania A)
    Else
        ' (kod zadania B)
    End If
Next
```

Jest to jednak znacznie trudniejsze, gdy trzeba wykonać kilka różnych testów. Aby przekonać się o zaletach instrukcji `Continue`, należy rozważyć bardziej skomplikowany (i realistyczny) przykład.

Na listingu 2.5 pokazano pętlę, która przetwarza tablicę słów. Program analizuje każde słowo i ustala, czy jest ono puste, czy też składa się z liter lub cyfr. Jeśli spełniony jest jeden z tych warunków (na przykład słowo składa się z liter), trzeba przejść do następnego słowa bez wykonywania następnego testu. Aby osiągnąć to bez użycia instrukcji `Continue`, trzeba zastosować zagnieżdżone pętle, co zmniejsza czytelność kodu.

Listing 2.5. Analizowanie łańcucha bez użycia instrukcji Continue

```
' Definiujemy zdanie
Dim Sentence As String = "Końcowy wynik obliczeń to 433."

' Przekształcamy zdanie w tablicę słów
Dim Delimiters() As Char = {" ", ".", ","}
Dim Words() As String = Sentence.Split(Delimiters)

' Badamy każde słowo
For Each Word As String In Words
    ' Sprawdzamy, czy słowo jest puste
    If Word <> "" Then
        Console.Write("'" + Word + "'" & vbTab & "= ")

        ' Sprawdzamy, czy słowo składa się z liter
        Dim AllLetters As Boolean = True
        For Each Character As Char In Word
            If Not Char.IsLetter(Character) Then
                AllLetters = False
            End If
        Next
        If AllLetters Then
            Console.WriteLine("słowo")
        Else
            ' Jeśli słowo nie składa się z liter,
            ' sprawdzamy, czy składa się z cyfr
```

```
            Dim AllNumbers As Boolean = True
            For Each Character As Char In Word
                If Not Char.IsDigit(Character) Then
                    AllNumbers = False
                End If
            Next
            If AllNumbers Then
                Console.WriteLine("liczba")
            Else
                ' Jeśli słowo nie składa się z samych liter albo cyfr,
                ' zakładamy, że zawiera ich kombinację (albo inne znaki)
                Console.WriteLine("mieszane")
            End If
        End If
    End If
Next
```

Teraz rozważmy wersję z listingu 2.6, w której użyto instrukcji `Continue`, aby działanie programu było bardziej zrozumiałe.

Listing 2.6. Analizowanie łańcucha przy użyciu instrukcji Continue

```
' Definiujemy zdanie
Dim Sentence As String = "Końcowy wynik obliczeń to 433."

' Przekształcamy zdanie w tablicę słów
Dim Delimiters() As Char = {" ", ".", ","}
Dim Words() As String = Sentence.Split(Delimiters)

' Badamy każde słowo
For Each Word As String In Words
    ' Sprawdzamy, czy słowo jest puste
    If Word = "" Then Continue For
    Console.Write("'" + Word + "'" & vbTab & "= ")

    ' Sprawdzamy, czy słowo składa się z liter
    Dim AllLetters As Boolean = True
    For Each Character As Char In Word
        If Not Char.IsLetter(Character) Then
            AllLetters = False
        End If
    Next
    If AllLetters Then
        Console.WriteLine("słowo")
        Continue For
    End If

    ' Jeśli słowo nie składa się z liter,
    ' sprawdzamy, czy składa się z cyfr
    Dim AllNumbers As Boolean = True
    For Each Character As Char In Word
        If Not Char.IsDigit(Character) Then
            AllNumbers = False
        End If
```

```
        Next
        If AllNumbers Then
            Console.WriteLine("liczba")
            Continue For
        End If

        ' Jeśli słowo nie składa się z samych liter albo cyfr,
        ' zakładamy, że zawiera ich kombinację (albo inne znaki)
        Console.WriteLine("mieszane")
    Next
```

A co...

...z użyciem instrukcji `Continue` w zagnieżdżonej pętli? Jest to możliwe. Jeśli zagnieździmy pętlę `For` w pętli `Do`, będziemy mogli użyć instrukcji `Continue For`, aby przejść do następnej iteracji wewnętrznej pętli, albo instrukcji `Continue Do`, aby przejść do następnej iteracji zewnętrznej pętli. Technika ta działa także w odwrotnej sytuacji (pętla `Do` wewnątrz pętli `For`), ale nie w przypadku dwóch zagnieżdżonych pętli tego samego typu. W takiej sytuacji nie da się jednoznacznie odwołać do zewnętrznej pętli, więc instrukcja `Continue` zawsze dotyczy pętli wewnętrznej.

Więcej informacji

Szczegółowe informacje o instrukcji `Continue` można znaleźć pod hasłem indeksu „continue statement" w pomocy MSDN.

Automatyczne usuwanie obiektów

Jak uniknąć pozostawiania w pamięci obiektów, które zajmują zasoby aż do momentu, w którym wyszuka je „odśmiecacz"? Dzięki instrukcji Using można upewnić się, że obiekty zostaną usunięte we właściwym momencie.

W środowisku .NET trzeba zwracać szczególną uwagę na to, aby obiekty używające niezarządzanych zasobów (na przykład uchwytów plików, połączeń z bazami danych i kontekstów graficznych) zwalniały je natychmiast, kiedy będzie to możliwe. Obiekty tego typu powinny implementować interfejs `IDisposable` i udostępniać metodę `Dispose()`, którą można wywołać w celu natychmiastowego zwolnienia zasobów.

Jedyny problem z tą techniką polega na tym, że trzeba pamiętać o wywołaniu metody `Dispose()` (albo innej metody wywołującej `Dispose()`, na przykład `Close()`). VB 2005 oferuje nowe zabezpieczenie, które gwarantuje wywołanie metody `Dispose()` — instrukcję `Using`.

Jak to zrobić?

Instrukcji `Using` używa się w strukturze bloku. W pierwszym wierszu bloku `Using` należy określić usuwalny obiekt. Często w tym samym momencie tworzy się obiekt za pomocą słowa kluczowego `New`. Następnie w bloku `Using` należy napisać kod, który używa usuwalnego obiektu. Oto krótki fragment kodu, który tworzy nowy plik i zapisuje w nim dane:

```
Using NewFile As New System.IO.StreamWriter("c:\MojPlik.txt")
    NewFile.WriteLine("To jest wiersz 1")
    NewFile.WriteLine("To jest wiersz 2")
End Using

' Plik jest zamykany automatycznie
' Obiekt NewFile w tym miejscu jest już niedostępny
```

W tym przykładzie natychmiast po opuszczeniu bloku `Using` wywoływana jest metoda `Dispose()` obiektu `NewFile`, co powoduje zwolnienie uchwytu pliku.

A co…

…z błędami występującymi w bloku `Using`? Na szczęście .NET zwalnia zasób bez względu na sposób wyjścia z bloku `Using`, nawet w przypadku nieobsłużonego wyjątku.

Instrukcji `Using` można używać w połączeniu ze wszystkimi usuwalnymi obiektami, na przykład:

- plikami (w tym `FileStream`, `StreamReader` i `StreamWriter`);
- połączeniami z bazą danych (w tym `SqlConnection`, `OracleConnection` i `OleDbConnection`);
- połączeniami sieciowymi (w tym `TcpClient`, `UdpClient`, `NetworkStream`, `FtpWebResponse`, `HttpWebResponse`);
- grafiką (w tym `Image`, `Bitmap`, `Metafile`, `Graphics`).

Więcej informacji

Szczegółowe informacje o instrukcji `Using` można znaleźć pod hasłem indeksu „Using block" w pomocy MSDN.

Ochrona właściwości przy użyciu różnych poziomów dostępu

W przeszłości nie dało się utworzyć właściwości, która mogła być odczytywana przez każdego, ale aktualizowana tylko przez aplikację. VB 2005 wreszcie poluzowuje reguły i oferuje większą elastyczność.

Większość właściwości zawiera **procedurę** `Get` (która umożliwia pobieranie wartości właściwości) oraz **procedurę** `Set` (która umożliwia określenie nowej wartości właściwości). W poprzednich wersjach Visual Basica poziom dostępu do obu procedur musiał być jednakowy. W VB 2005 można zabezpieczyć właściwość, przypisując procedurze `Set` niższy poziom dostępu niż procedurze `Get`.

Jak to zrobić?

VB rozróżnia trzy poziomy dostępu. W kolejności od najmniej do najbardziej ograniczonego są to:

- `Public` (element dostępny dla wszystkich klas we wszystkich podzespołach);
- `Friend` (element dostępny dla wszystkich klas w bieżącym podzespole);
- `Private` (element dostępny tylko dla kodu w tej samej klasie).

Wyobraźmy sobie, że tworzymy komponent DLL, który będzie używany przez inną aplikację. Zawiera on właściwość `Status`, która będzie odczytywana przez aplikację kliencką, więc deklarujemy ją jako publiczną:

```
Public Class ComponentClass

    Private _Status As Integer
    Public Property Status() As Integer
        Get
            Return _Status
        End Get
        Set(ByVal value As Integer)
            _Status = value
        End Set
    End Property

End Class
```

Problem w tym, że poziom dostępu przypisany właściwości `Status` pozwala klientowi na jej zmianę, co nie ma sensu. Właściwość `Status` mogłaby być przeznaczona tylko do odczytu (w tym celu należałoby pominąć procedurę `Set`), ale wtedy nie mogłyby jej zmieniać inne klasy, które stanowią część aplikacji i znajdują się w tym samym podzespole.

Rozwiązaniem jest przypisanie procedurze `Set` poziomu dostępu `Friend`. Oto, jak powinien wyglądać kod (jedyną zmianę wyróżniono pogrubioną czcionką):

```
Public Property Status() As Integer
    Get
         Return _Status
    End Get
    Friend Set(ByVal value As Integer)
         _Status = value
    End Set
End Property
```

A co...

...z właściwościami przeznaczonymi tylko do odczytu lub tylko do zapisu? Różne poziomy dostępu nie pomogą, jeśli potrzebna jest właściwość przeznaczona tylko do odczytu (na przykład wartość obliczana) albo tylko do zapisu (na przykład hasło, które po ustawieniu nie powinno pozostawać dostępne). W celu utworzenia właściwości przeznaczonej tylko do odczytu należy dodać słowo kluczowe `ReadOnly` do deklaracji właściwości (tuż po słowie kluczowym określającym poziom dostępu) i usunąć procedurę `Set`. W celu utworzenia właściwości przeznaczonej tylko do zapisu należy usunąć procedurę `Get` i dodać słowo kluczowe `WriteOnly`. Te słowa kluczowe nie są niczym nowym — były dostępne już w Visual Basicu .NET 1.0.

Nowe operatory logiczne umożliwiają łączenie wielu warunków i pisanie bardziej zwartego kodu.

Testowanie kolejnych części wyrażenia warunkowego

W poprzednich wersjach Visual Basica istniały dwa operatory logiczne: `And` i `Or`. W Visual Basicu 2005 wprowadzono dwa dodatkowe operatory: `AndAlso` oraz `OrElse`. Działają one tak samo jak `And` i `Or`, ale pozwalają oszacować tylko jedną część długiego wyrażenia warunkowego.

Jak to zrobić?

W wielu programach trzeba oszacować kilka warunków z rzędu. Często najpierw sprawdzamy, czy obiekt nie jest pusty, a następnie badamy jedną z jego właściwości.

W tym celu musimy używać zagnieżdżonych bloków `If`, jak w poniższym przykładzie:

```
If Not MyObject Is Nothing Then
    If MyObject.Value > 10 Then
        ' (robimy coś)
    End If
End If
```

Byłoby miło połączyć te dwa warunki w jednym wierszu:

```
If Not MyObject Is Nothing And MyObject.Value > 10 Then
    ' (robimy coś)
End If
```

Niestety, to nie zadziała, ponieważ VB zawsze testuje oba warunki. Innymi słowy, nawet jeśli `MyObject` jest równy `Nothing`, VB sprawdzi drugi warunek i spróbuje pobrać właściwość `MyObject.Value`, co spowoduje wyjątek `NullReferenceException`.

W Visual Basicu 2005 rozwiązano ten problem dzięki słowom kluczowym `AndAlso` i `OrElse`. Kiedy używamy tych operatorów, Visual Basic nie testuje drugiego warunku, jeśli pierwszy nie jest spełniony. Oto poprawiony kod:

```
If Not MyObject Is Nothing AndAlso MyObject.Value > 10 Then
    ' (robimy coś)
End If
```

A co...

...z innymi ulepszeniami języka? W tym rozdziale przedstawiono najważniejsze innowacje w języku VB. Warto jednak wspomnieć o kilku innych nowościach, które nie zostały tu opisane:

- Słowo kluczowe `IsNot` pozwala uprościć niezgrabną składnię. Dzięki niemu można zastąpić konstrukcję `If Not x Is Nothing` równoważną instrukcją `If x IsNot Nothing`.
- Za pomocą funkcji `TryCast()` można nieco przyspieszyć rzutowanie typów. Działa ona tak jak funkcje `CType()` lub `DirectCast()`, z jednym wyjątkiem — jeśli obiektu nie można przekształcić w żądany typ, zwracana jest pusta referencja. Zamiast więc sprawdzać typ obiektu, a potem go rzutować, można od razu użyć funkcji `TryCast()`, a następnie sprawdzić, czy zwróciła ona rzeczywistą referencję do obiektu.

- Zmienne całkowite bez znaku umożliwiają przechowywanie wartości, które nie mogą być ujemne. Ograniczenie to oszczędza pamięć, pozwalając na przechowywanie większych liczb. Liczby bez znaku zawsze były obsługiwane przez .NET Framework, ale obecnie VB 2005 zawiera odpowiednie słowa kluczowe (`UInteger`, `ULong` i `UShort`).