Microsoft Visual Studio 2005. Księga eksperta

Autor: Lars Powers, Mike Snell
Tłumaczenie: Tomasz Walczak, Maria Chaniewska
ISBN: 83-246-0837-0
Tytuł oryginału: Microsoft Visual Studio 2005 Unleashed
Format: B5, stron: około 848
oprawa twarda

Zwiększ swoją produktywność dzięki Visual Studio 2005 oraz Team System

- Kompletny przegląd środowiska
- Nowości w Visual Studio 2005
- Praca zespołowa w Team System

Microsoft nie zaprzestaje wysiłków, doskonaląc narzędzia do tworzenia programów dla platformy .NET. Visual Studio 2005 to następny krok w tym kierunku. Nowe właściwości pozwalają jeszcze bardziej zwiększyć wydajność programistów, poprawić komfort ich pracy i zautomatyzować wykonywanie żmudnych zadań, a wersja Team System umożliwia współpracę całych zespołów projektowych, włączając w to menedżerów projektu, architektów, programistów, testerów czy administratorów. Wielość możliwości i funkcji środowiska Visual Studio 2005 sprawia, że nawet doświadczonym programistom przyda się wszechstronne źródło wiedzy.

Książka „Microsoft Visual Studio 2005. Księga eksperta" to podręcznik, w którym wyczerpująco omówiono to rozbudowane środowisko. Dzięki niemu poznasz elementy interfejsu użytkownika potrzebne do utworzenia aplikacji od początku do końca - od przygotowywania projektu, poprzez pisanie i modyfikowanie kodu, aż po diagnozowanie i testowanie programu. Nauczysz się korzystać z nowych funkcji Visual Studio 2005 umożliwiających między innymi automatyzację testów, wydajną refaktoryzację czy wiązanie danych bez konieczności pisania kodu, co przyczyni się do wzrostu Twej produktywności. Dowiesz się także, jak używać narzędzi wersji Team System pozwalających na sprawną współpracę wielu osób pracujących przy tworzeniu oprogramowania.

- Przegląd okien projektowych i edytorów
- Praca z projektami i rozwiązaniami
- Diagnozowanie kodu
- Obsługa baz danych
- Refaktoryzacja kodu
- Pisanie kreatorów, makr i dodatków
- Tworzenie aplikacji sieciowych w ASP.NET
- Pisanie i konsumowanie usług sieciowych
- Automatyzacja testów
- Zarządzanie zmianami i kontrola wersji
- Współpraca zespołowa
- Społeczność programistów .NET

Ta książka to nieoceniona pomoc dla wszystkich użytkowników Visual Studio 2005.

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

Spis treści

Rozdział 3.
Rozszerzenia platformy i języków .NET w wersji 2005

Większa część niniejszej książki dotyczy głównie narzędzi IDE Visual Studio służących poprawie produktywności. Uważamy jednak, że warto zapoznać się także z nowinkami w językach .NET i samej platformie. Te elementy (IDE, języki i platforma) są udostępniane przez Microsoft w jednym pakiecie. Z tego powodu wszelkie opisy dotyczące IDE byłyby niepełne bez wzmianki o pozostałych produktach.

W rozdziale opisano nowinki w językach Visual Basic .NET i C#. Opisujemy także niektóre z kluczowych poprawek w samej platformie. Zakładamy, że większość Czytelników ma przynajmniej podstawową wiedzę na temat albo Visual Basic, albo któregoś ze starszych języków bazujących na C, a także w miarę dobrze zna platformę .NET. Dlatego koncentrujemy się głównie na tych rozszerzeniach, dzięki którym .NET 2.0 stanowi krok naprzód w porównaniu z poprzednimi wersjami.

Rozszerzenia wspólne dla różnych języków .NET

Języki z rodziny .NET otrzymały sporo rozszerzeń w wyniku poprawek wprowadzonych we wspólnym środowisku uruchomieniowym (ang. *Common Language Runtime — CLR*). Choć niektóre z rozszerzeń są specyficzne dla poszczególnych języków, liczne istotne usprawnienia w wersji 2005 dotyczą ich obu. Dlatego przedstawiamy je wspólnie wraz z przykładowym kodem w obu tych językach. Ta grupa rozszerzeń języków .NET obejmuje następujące kluczowe usprawnienia:

- Typy ogólne.
- Typy dopuszczające wartość "null".
- Typy częściowe.
- Właściwości o mieszanym poziomie dostępu.
- Wieloznaczne przestrzenie nazw.

W następnych punktach szczegółowo opisujemy każdy z tych elementów. Przedstawiamy przykłady w językach C# i Visual Basic, ponieważ powyższe nowinki dotyczą ich obu. Rozszerzenia specyficzne dla każdego z tych języków opisane są w dalszej części rozdziału.

Typy ogólne

Typy ogólne (ang. *generics*) to bez wątpienia najważniejsza nowość w .NET 2.0, dlatego żadna książka opisująca tę platformę nie byłaby kompletna bez opisu ich działania. Typy ogólne początkowo mogą się wydawać skomplikowane, szczególnie jeśli zaczniesz analizować kod zawierający dziwne nawiasy ostre w przypadku języka C# czy słowo kluczowe `Of` w Visual Basic. Poniższe podpunkty zawierają definicję typów ogólnych, wyjaśniają ich znaczenie oraz pokazują, jak używać ich w kodzie.

Definicja typów ogólnych

Działanie *typów ogólnych* jest stosunkowo proste. Czasem trzeba utworzyć obiekt (lub zdefiniować parametr metody), jednak na etapie pisania kodu nie wiadomo, jakiego typu będzie ten obiekt. Typ powinien być *ogólny* i umożliwiać osobie korzystającej z niego określenie faktycznego typu obiektu.

Do rozwiązania tego problemu można użyć klasy `System.Object` — tak postępowali programiści używający wersji starszych od 2.0. Jednak wyobraź sobie, że chcesz uniknąć konieczności pakowania typów, sprawdzania ich w czasie wykonywania programu i jawnego rzutowania w wielu miejscach kodu. Pozwala to zrozumieć, do czego mogą służyć typy ogólne.

Zalety stosowania typów ogólnych najlepiej pokazać na przykładzie. Najprostszy to tworzenie klasy kolekcji zawierającej inne obiekty. Wyobraź sobie, że chcesz zapisać grupę obiektów. Można to zrobić, dodając je do kolekcji `ArrayList`. Jednak kompilator i środowisko uruchomieniowe mają informacje jedynie o tym, że przechowywana jest lista jakichś obiektów. Może ona zawierać obiekty typu `Order`, `Customer` lub obu tych typów (lub dowolnych innych). Jedyny sposób, aby dowiedzieć się, co przechowuje lista, to napisać kod sprawdzający typ przechowywanych obiektów.

Oczywiście można obejść ten problem i napisać własną silnie typowaną listę. Choć takie rozwiązanie jest możliwe, wymaga to żmudnego pisania niemal takiego samego kodu dla każdego typu, który ma być przechowywany w kolekcji. Jedyną istotną różnicą w kodzie jest typ obiektów, jakie lista może przechowywać. Ponadto wciąż trzeba rzutować obiekty, ponieważ używana lista zawiera obiekty typu `System.Object`.

Teraz wyobraź sobie, że możesz napisać klasę, która umożliwia użytkownikom definiowanie typu. Można napisać jedną *ogólną* klasę listy, która zamiast zawierać obiekty typu `System.Object`, będzie zawierała obiekty typu użytego w miejscu definiowania tej klasy. Umożliwia to określenie w kodzie wywołującym ogólną listę tego, czy na przykład ma ona przechowywać obiekty typu `Order` lub `Customer`. Właśnie to umożliwiają typy ogólne. Można myśleć o klasach ogólnych jako o szablonach klas.

Są dwa rodzaje elementów ogólnych: typy ogólne i metody ogólne. *Typy ogólne* to klasy, których typ jest definiowany w miejscu tworzenia egzemplarza danej klasy. *Metoda ogólna* ma przynajmniej jeden parametr o typie ogólnym. W tym przypadku w metodzie używany jest ogólny parametr, ale jego typ jest określany dopiero w momencie wywołania metody. Ponadto można zdefiniować różne ograniczenia związane z tworzeniem typów ogólnych. W następnych podpunktach przedstawimy różne związane z nimi zagadnienia.

Zalety stosowania typów ogólnych

Teraz jasno widać niektóre z zalet stosowania typów ogólnych. Bez takich typów klasy mające zarządzać różnymi typami muszą używać typu `System.Object`. Powoduje to liczne problemy. Po pierwsze, nie ma możliwości nałożenia ograniczeń ani sprawdzenia przez kompilator, co jest zapisywane w obiekcie. Istotne są wynikające z tego wnioski — nie wiadomo, co znajduje się w kolekcji, jeśli nie można ograniczyć typu zapisywanych w niej obiektów. Po drugie, używając obiektu, trzeba sprawdzić jego typ, a następnie zrzutować z powrotem na oryginalny typ. Oczywiście powoduje to spadek wydajności. Ponadto jeśli programista używa typów skalarnych i zapisuje je w obiektach `System.Object`, typy są pakowane. W momencie pobierania typu skalarnego trzeba go rozpakować. Także to wymaga zbędnego kodu i niepotrzebnie pogarsza wydajność. Typy ogólne rozwiązują wszystkie te problemy. Poniżej opisujemy, jak jest to możliwe.

W jaki sposób .NET obsługuje typy ogólne?

Kompilacja typu ogólnego, podobnie jak pozostałego kodu dla platformy .NET, powoduje utworzenie kodu w języku pośrednim Microsoftu (ang. *Microsoft Intermediate Language — MSIL*) oraz metadanych. Oczywiście w przypadku typów i metod ogólnych kompilator generuje kod MSIL definiujący sposób ich stosowania.

Kiedy kod MSIL jest wykonywany po raz pierwszy, kompilator JIT (ang. *just-in-time*) kompiluje go do kodu natywnego. Kiedy taki kompilator natrafi na typ ogólny, „wie", że w jego miejsce należy wstawić typ rzeczywisty i robi to. Ten proces nazywany jest *tworzeniem egzemplarza typu ogólnego*.

W następnych wywołaniach używany jest świeżo skompilowany typ kodu natywnego. W rzeczywistości wszystkie typy referencyjne mogą współdzielić jeden egzemplarz typu ogólnego, ponieważ w kodzie natywnym te referencje to po prostu wskaźniki na ten sam obiekt. Oczywiście jeśli w procesie tworzenia egzemplarza typu ogólnego używany jest nowy typ skalarny, środowisko uruchomieniowe „w locie" utworzy nową kopię tego typu ogólnego.

Dlatego korzystanie z typów ogólnych niesie zalety zarówno na etapie pisania kodu, jak i w czasie jego wykonywania. W czasie wykonywania cały oryginalny kod jest przekształcany na natywny, silnie typowany kod. Przyjrzyjmy się teraz, jak tworzyć typy ogólne.

Tworzenie typów ogólnych

Typy ogólne to klasy zawierające przynajmniej jeden element, którego typ należy określić w czasie tworzenia obiektu (a nie na etapie tworzenia klasy). Aby zdefiniować typ ogólny, należy najpierw zadeklarować klasę, a następnie zdefiniować dla niej parametry typu. *Parametr typu* jest przekazywany do klasy w celu zdefiniowania rzeczywistego typu dla typu ogólnego. Parametry typu działają podobnie do parametrów metod. Istotna różnica polega na tym, że zamiast wartości lub referencji do obiektu przekazywany jest typ używany w typie ogólnym.

Większość typów ogólnych służy do zarządzania kolekcjami obiektów lub listami wiązanymi. Nie jest to jednak jedyne zastosowanie tych typów. Dowolna klasa może być typem ogólnym.

Załóżmy, że programista chce napisać klasę `Fields`, która obsługuje pary nazwa-wartość, podobnie jak robią to kolekcje `Hashtable` czy `Dictionary`. Można zadeklarować tę klasę w następujący sposób:

C#:

```
public class Fields
```

VB:

```
Public Class Fields
```

Załóżmy ponadto, że ta klasa może używać różnych typów jako kluczy i wartości. Klasa ma w ogólny sposób obsługiwać wiele typów, jednak po utworzeniu jej egzemplarza należy je ograniczyć do tych, które zostały określone w procesie tworzenia tego egzemplarza. Aby dodać parametry typu do deklaracji klasy, należy użyć następującej składni:

C#:

```
public class Fields<keyType, valueType>
```

VB:

```
Public Class Fields(Of keyType, valueType)
```

W tym przypadku `keyType` i `valueType` to parametry typu, których można używać w reszcie kodu klasy zamiast typów, które zostaną określone w czasie tworzenia jej egzemplarza. Można na przykład dodać do klasy metodę `Add`, której sygnatura wygląda tak:

C#:

```
public void Add(keyType key, valueType value)
```

VB:

```
Public Sub Add(key as keyType, value as valueType)
```

Stanowi to informację dla kompilatora o tym, że typu użytego do utworzenia klasy należy także używać w tej metodzie. Aby korzystać z klasy, w kodzie trzeba najpierw utworzyć jej egzemplarz i przekazać do niego argumenty typu. *Argumenty typu* to typy przekazywane jako parametry typu, takie jak w poniższym kodzie:

C#:

```
Fields<int, Field> myFields = new Fields<int, Field>();
```

VB:

```
Dim myFields As New Fields(Of Integer, Field)
```

W tym przypadku nowy egzemplarz ogólnej klasy `Field` musi zawiera liczbę typu `int` (`Integer`) jako klucz i obiekt typu `Field` jako wartość. Wywołanie metody `Add` świeżo utworzonego obiektu `Field` wygląda teraz tak:

C#:

```
myFields.Add(1, new Field());
```

VB:

```
myFields.Add(1, New Field())
```

Próba przekazania parametru innego typu wywoła błąd kompilacji, ponieważ wymagany jest obiekt określonego typu.

Kiedy stosuje się typy ogólne, często do definiowania nazw typów używa się pojedynczych liter (szczególnie widoczne jest to w języku C#). Nierzadko można zobaczyć zapis typu <T> lub <K>. Nie trzeba ograniczać się do stosowania takich krótkich nazw. Zawsze lepiej jest używać nieco bardziej opisowego zapisu.

Tworzenie metod ogólnych

Przedstawiliśmy już parametry typów ogólnych. Te parametry typu definiują zmienne o zasięgu klasy. Oznacza to, że zmienna definiująca typ ogólny jest dostępna w całej klasie. Podobnie jak w przypadku każdej innej klasy, nie trzeba używać zmiennych o zasięgu całej klasy. Czasem wystarczy zdefiniować elementy przekazywane do danej metody. Elementy ogólne działają zawsze tak samo. Można zdefiniować je na poziomie klasy (co już pokazaliśmy) lub na poziomie metody (co pokażemy wkrótce).

Metody ogólne dobrze nadają się do obsługi często używanych funkcji narzędziowych, które wykonują operacje na różnych typach. Metody ogólne definiuje się poprzez określenie przynajmniej jednego typu ogólnego po nazwie metody. Następnie można używać tych typów ogólnych na liście parametrów metody jako zwracanego typu oraz oczywiście w ciele metody. Poniżej przedstawiona jest składnia służąca do definiowania metod ogólnych:

C#:

```
public void Save<instanceType>(instanceType type)
```

VB:

```
Public Sub Save(Of instanceType)(ByVal type As instanceType)
```

Aby wywołać tę ogólną metodę, trzeba zdefiniować w jej wywołaniu typ przekazywany do metody. Załóżmy, że metoda `Save` zdefiniowana w poprzednim fragmencie kodu

znajduje się w klasie `Field`, a programista utworzył egzemplarz tej klasy i zapisał referencję do niego w zmiennej `myField`. Poniższy kod przedstawia, jak można wywołać metodę `Save`, przekazując typ argumentu:

C#:

```
myField.Save<CustomerOrder>(new CustomerOrder());
```

VB:

```
myField.Save(Of CustomerOrder)(New CustomerOrder())
```

Warto pamiętać o kilku sprawach związanych z metodami ogólnymi. Po pierwsze, często można pominąć parametr typu w miejscu wywołania metody ogólnej. Kompilator potrafi wykryć ten typ na podstawie przekazanych parametrów, dlatego parametr typu jest opcjonalny. Jednak zwykle warto go określić, ponieważ dzięki temu kod jest bardziej czytelny, a kompilator nie musi sprawdzać typu. Po drugie, metody ogólne można deklarować jako statyczne (lub współdzielone). Po trzecie, można zdefiniować ograniczenia dla metod (i klas) ogólnych, co opisano w następnym podpunkcie.

Specyficzne typy ogólne (ograniczenia)

Przy pierwszym napotkaniu metod ogólnych łatwo myśleć o nich jako o prostych narzędziach ułatwiających przechowywanie danych. Na pierwszy rzut oka wydaje się, że cała ta technika ma poważną wadę. Najlepiej można ją przedstawić za pomocą pytań, które przychodzą do głowy: „Typy ogólne są świetne, ale co zrobić, jeśli chcemy wywołać metodę lub właściwość ogólnego obiektu, którego typ jest z definicji nieznany?”. Ta wada wydaje się ograniczać przydatność typów ogólnych. Jednak po dalszym zastanowieniu się można zauważyć, że ograniczenia typów ogólnych pozwalają rozwiązać ten problem.

Ograniczenia typów ogólnych działają tak, jak wskazuje na to ich nazwa — pozwalają zdefiniować ograniczenia dotyczące typów, których nadawca może używać do tworzenia egzemplarza klasy ogólnej lub wywoływania jednej z metod ogólnych. Ograniczenia typów ogólnych mają trzy odmiany:

- **Ograniczenia dotyczące dziedziczenia** — pozwalają określić, że typ ogólny musi implementować określone interfejsy lub dziedziczyć po danej klasie bazowej.
- **Ograniczenia dotyczące konstruktora domyślnego** — pozwalają określić, że typ ogólny musi udostępniać konstruktor bezargumentowy.
- **Ograniczenia dotyczące typu (referencyjne lub skalarne)** — pozwalają określić, że parametr typu musi być albo typem referencyjnym, albo skalarnym.

Używanie ograniczenia dotyczącego dziedziczenia umożliwia określenie jednego lub kilku interfejsów (lub typów obiektu), które można przekazywać do klasy ogólnej. To pozwala rozwiązać opisany wcześniej problem. Jeśli zdefiniowana wcześniej ogólna klasa `Fields` ma umożliwiać wywoływanie metody lub właściwości ogólnego typu `valueType` (na przykład właściwości ułatwiającej sortowanie grup obiektów typu `Fields`), można to wymusić, o ile dana metoda lub właściwość jest zdefiniowana w interfejsie lub klasie bazowej określonej w ograniczeniu. Poniższy kod pokazuje, jak zdefiniować ograniczenie dotyczące dziedziczenia:

Ograniczenie klasy w C#:

```
public class Field<keyType, valueType> where keyType : ISort
```

Ograniczenie klasy w VB:

```
Public Class Fields(Of keyType, valueType As ISort)
```

W powyższym kodzie klasa `Fields` zawiera definicje dwóch typów ogólnych, `valueType` i `keyType`, oraz ma ograniczenie związane z parametrem `keyType`. To ograniczenie polega na tym, że `keyType` musi implementować interfejs `ISort`. Umożliwia to używanie metod tego interfejsu w klasie `Fields` bez konieczności rzutowania.

Ograniczenia dotyczące dziedziczenia można definiować dla klas i metod ogólnych.

Można określić dowolną liczbę interfejsów, które typ ogólny ma implementować, ale tylko jedną klasę bazową, po której ma dziedziczyć. Oczywiście można przekazać jako typ ogólny obiekt, który dziedziczy po klasie bazowej określonej jako ograniczenie.

Jeśli programista przesłoni ogólną metodę w klasie bazowej, *nie może* dodać (ani usunąć) ograniczenia z ogólnej metody. Uwzględniane są jedynie ograniczenia zdefiniowane w klasie bazowej.

Przestrzeń nazw z kolekcjami ogólnymi

Wiesz już, jak utworzyć własną klasę ogólną. Platforma .NET udostępnia liczne klasy ogólne, których można używać w aplikacjach. Przestrzeń nazw `System.Collections.Generics` definiuje liczne ogólne klasy kolekcji, które są zaprojektowane tak, aby współpracowały z grupami obiektów określonego typu. *Kolekcja ogólna* to klasa kolekcji, która umożliwia programiście określenie typu przechowywanego w kolekcji w miejscu jej deklaracji.

Visual Studio domyślnie dodaje referencję do przestrzeni nazw `System.Collections.Generics` do wszystkich plików z kodem języków Visual Basic i C#.

Klasy ogólne zdefiniowane w tej przestrzeni nazw różnią się zastosowaniami. Klasa `List` służy do obsługi prostych list i tablic obiektów. Dostępne są także klasy `SortedList`, `LinkedList`, `Queue`, `Stack` i kilka klas `Dictionary`. Dzięki tym klasom można korzystać z wszystkich podstawowych operacji bez konieczności używania typizowanych klas kolekcji. Ponadto w tej przestrzeni nazw zdefiniowanych jest wiele interfejsów, których można używać do tworzenia własnych kolekcji ogólnych.

Typy dopuszczające wartość null

Większość programistów od czasu do czasu musi utworzyć w aplikacji zmienną i określić jej domyślną wartość, zanim jeszcze wiadomo, jaką wartość ta zmienna powinna zawierać. Wyobraź sobie, że programista utworzył klasę `Osoba` ze zmienną logiczną `JestKobieta`. Jeśli na etapie tworzenia egzemplarza nie wiadomo, jaka jest płeć osoby, trzeba wybrać domyślną wartość lub zaimplementować tę właściwość jako trójstanowe wyliczenie o wartościach `Mezczyzna`, `Kobieta` i `Nieznana`.

To ostatnie rozwiązanie może być niewygodne, szczególnie jeśli w bazie danych wartość jest przechowywana jako zmienna logiczna. Można wymyślić podobne sytuacje. Wyobraź sobie, że w programie znajduje się klasa `Test` zawierająca zmienną całkowitoliczbową `Ocena`. Jeśli nie wiadomo, jaka jest wartość tej zmiennej, jest ona inicjowana liczbą zero (0). Ta wartość oczywiście nie reprezentuje rzeczywistej oceny. Następnie trzeba uwzględnić ten fakt albo traktując zero jako magiczną liczbę, albo używając następnej zmiennej, na przykład `OcenaJestUstawiona`.

Ten problem jest tym większy, że wszystkie współczesne bazy danych uwzględniają wartości `null` (lub nieustawione). Często nie można użyć tej właściwości bez napisania dodatkowego kodu, który przekształca wartości w czasie transakcji wstawiania i pobierania danych.

Typy dopuszczające wartość `null` w .NET 2.0 mają rozwiązać ten problem. *Typ dopuszczający wartość `null`* to specjalny typ skalarny, który może mieć wartość `null`. Inaczej działają standardowe typy skalarne (`int`, `bool`, `double` i tak dalej), które po prostu nie są inicjowane w momencie deklarowania. Z kolei dzięki typom dopuszczającym wartość `null` można tworzyć liczby całkowite, wartości logiczne, liczby zmiennoprzecinkowe o podwójnej precyzji i inne, a następnie przypisać do nich wartość `null`. Nie trzeba już zgadywać, czy zmienna została ustawiona (lub używać do tego specjalnego kodu). Zwalnia to też programistę z konieczności określania wartości domyślnej. W zamian można zainicjować zmienną wartością `null` lub przypisać ją do zmiennej. Dzięki temu można pisać kod bez domyślnych założeń. Ponadto typy dopuszczające wartość `null` rozwiązują problem pobierania wartości `null` z bazy danych i umieszczania ich w niej. Przyjrzymy się teraz, jak działają te typy.

Deklarowanie typów dopuszczających wartość null

Deklarowanie typów dopuszczających wartość `null` znacznie różni się między językami C# i Visual Basic. Jednak w obu przypadkach deklarowana jest ta sama struktura typu dopuszczającego wartość `null` platformy .NET (`System.Nullable`). Ta ogólna struktura jest zdefiniowana przez typ używany w jej deklaracji. Na przykład jeśli programista definiuje liczbę całkowitą dopuszczającą wartość `null`, ta ogólna struktura zwraca wersję liczby całkowitej. Poniższe fragmenty kodu demonstrują, w jaki sposób typy dopuszczające wartość `null` są deklarowane w językach C# i VB:

Typ dopuszczający wartość `null` w C#:

```
bool? hasChildren = null;
```

Typ dopuszczający wartość `null` w VB:

```
Dim hasChildren As Nullable(Of Boolean) = Nothing
```

Warto zauważyć, że w kodzie języka C# można użyć modyfikatora typu `?` do wskazania, że typ bazowy należy traktować jako typ dopuszczający wartość `null`. Jest to po prostu skrót. Pozwala to programistom używać standardowej składni tworzenia typów i dodać znak zapytania, żeby przekształcić dany typ na dopuszczający wartość `null`. Z drugiej strony w języku Visual Basic trzeba bardziej bezpośrednio zdefiniować klasę `Nullable`, używając

składni podobnej do typów ogólnych. Można także użyć podobnej składni w języku C#, tak jak w poniższym przykładzie:

```
System.Nullable<bool> hasChildren = null;
```

Jedynie typy skalarne mogą dopuszczać wartość `null`. Dlatego nie jest poprawne tworzenie dopuszczających wartość `null` łańcuchów znaków czy egzemplarzy klas zdefiniowanych przez użytkownika. Można jednak tworzyć dopuszczające wartość `null` egzemplarze struktur, ponieważ są one typami skalarnymi.

Używanie typów dopuszczających wartość null

Ogólna struktura `System.Nullable` zawiera dwie właściwości służące tylko do odczytu: `HasValue` i `Value`. Te właściwości umożliwiają wydajne korzystanie z typów dopuszczających wartość `null`. Właściwość `HasValue` to wartość logiczna, która określa, czy dany typ dopuszczający wartość `null` ma nadaną wartość. Można użyć tej właściwości w instrukcji `If`, aby określić, czy do danej zmiennej została już przypisana wartość. Ponadto można po prostu sprawdzić, czy zmienna ma wartość `null` (jedynie w C#). Poniżej przedstawiony jest przykładowy kod:

`HasValue` w C#:

```
if (hasChildren.HasValue) {...}
```

`HasValue` w VB:

```
If hasChildren.HasValue Then
```

Sprawdzanie zmiennej w C# ze względu na wartość `null`:

```
if (hasChildren != null) {...}
```

Sprawdzanie zmiennej w VB ze względu na wartość `null`:

```
If hasChildren.Value <> Nothing Then
```

Właściwość `Value` zwraca wartość zawartą w strukturze dopuszczającej wartość `null`. Wciąż można uzyskać dostęp do wartości tej zmiennej, wywołując ją bezpośrednio (bez używania właściwości `Value`). Różnica polega na tym, że kiedy właściwość `HasValue` ma wartość `false`, próba wywołania właściwości `Value` spowoduje zgłoszenie wyjątku. Z kolei bezpośrednia próba dostępu do zmiennej w warunku (`HasValue = false`) nie powoduje tego. Dlatego ważne jest ustalenie, jak powinien działać program, a następnie należy używać tych opcji w odpowiedni sposób. Poniżej znajduje się przykład zastosowania właściwości `Value`:

Właściwość `Value` w C#:

```
System.Nullable<bool> hasChildren = null;
Console.WriteLine(hasChildren); // Nie jest zgłaszany wyjątek
if (hasChildren != null) {
  Console.WriteLine(hasChildren.Value.ToString());
}
Console.WriteLine(hasChildren.Value); // Zgłasza wyjątek InvalidOperationException
```

Właściwość `Value` w VB:

```
Dim hasChildren As Nullable(Of Boolean) = Nothing
Console.WriteLine(hasChildren) ' Nie jest zgłaszany wyjątek
If hasChildren.HasValue Then
  Console.WriteLine(hasChildren.Value.ToString())
End If
Console.WriteLine(hasChildren.Value) ' Zgłasza wyjątek InvalidOperationException
```

W powyższych fragmentach kodu bezpośrednie wywołania zmiennej `hasChildren` nie powodują zgłoszenia wyjątku. Jednak próba sprawdzania wartości właściwości `Value` w sytuacji, kiedy zmienna ma wartość `null`, spowoduje zgłoszenie przez platformę .NET wyjątku `InvalidOperationException`.

Typy (klasy) częściowe

Typy częściowe to mechanizm służący do definiowania pojedynczych klas, struktur lub interfejsów w kilku plikach. W rzeczywistości w czasie kompilacji kodu nie ma czegoś takiego jak typ częściowy. Takie typy istnieją jedynie w czasie tworzenia aplikacji. Na etapie kompilacji zawartość plików definiujących typ częściowy jest łączona w pojedynczą klasę.

Typy częściowe mają rozwiązywać dwa problemy. Po pierwsze, umożliwiają programistom rozbijanie dużych klas na kilka plików. Pozwala to różnym członkom zespołu pracować nad tą samą klasą bez konieczności korzystania z tego samego pliku (dzięki czemu unika się związanych z tym problemów z łączeniem kodu). Po drugie, pozwalają na oddzielenie kodu wygenerowanego przez narzędzia od kodu autorstwa programisty. Dzięki temu plik z kodem programisty będzie przejrzysty (ponieważ jedynie on go pielęgnuje), a na zapleczu narzędzie może generować inne elementy klasy. Programiści używający Visual Studio 2005 natychmiast zauważą to, kiedy będą korzystać z formularzy Windows, nakładek na usługi sieciowe, stron z kodem ukrytym ASP i podobnych narzędzi. Jeśli korzystałeś z tych elementów w poprzednich wersjach .NET, szybko spostrzeżesz, że w wersji 2005 w plikach nie ma wygenerowanego kodu, a klasa pisana przez programistę jest oznaczona jako `partial`.

Używanie typów częściowych

Typy częściowe zarówno w C#, jak i w Visual Basic deklaruje się za pomocą słowa kluczowego `Partial`. Można stosować to słowo kluczowe do klas, struktur i interfejsów. Musi ono być pierwszym słowem w deklaracji (przed `Class`, `Structure` czy `Interface`). Wskazanie, że typ jest częściowy, informuje kompilator o tym, że należy połączyć części tej klasy w pojedynczym pliku *.dll* lub *.exe*.

Przy definiowaniu typów częściowych trzeba stosować się do kilku prostych wskazówek. Po pierwsze, w deklaracji wszystkich typów o tej samej nazwie w tej samej przestrzeni nazw musi znajdować się słowo kluczowe `Partial`. Nie można na przykład zadeklarować klasy `Partial Public Person` w jednym pliku, a klasy `Public Person` w innym pliku tej samej przestrzeni nazw — trzeba dodać słowo kluczowe `Partial` także do drugiej deklaracji. Po drugie, trzeba pamiętać, że wszystkie modyfikatory typu częściowego są łączone w czasie kompilacji. Obejmuje to atrybuty klasy, komentarze XML i implementacje interfejsu. Na przykład jeśli programista użył atrybutu `System.SerializableAttribute` dla

typu częściowego, ten atrybut zostanie zastosowany do tego typu w czasie jego łączenia i kompilacji. Po trzecie, warto pamiętać, że wszystkie typy częściowe muszą zostać skompilowane do tego samego podzespołu (inaczej pakietu, czyli pliku *.dll* lub *.exe*; ang. *assembly*). Nie można skompilować typu częściowego obecnego w kilku podzespołach.

Właściwości z mieszanym poziomem dostępu

W poprzednich wersjach .NET można było określić poziom dostępu (publiczny, prywatny, chroniony, wewnętrzny) jedynie dla całej właściwości. Jednak często potrzebna jest właściwość umożliwiająca publiczny odczyt (`get`), ale tylko wewnętrzny zapis (`set`). Jedynym sensownym rozwiązaniem w poprzednich wersjach .NET było pominięcie metody `set` właściwości. Następnie trzeba było utworzyć nową metodę wewnętrzną służącą do ustawiania wartości tej właściwości. Kod byłby łatwiejszy do napisania i zrozumienia, gdyby możliwa była bardziej precyzyjna kontrola nad modyfikatorami dostępu w obrębie właściwości.

.NET 2.0 daje kontrolę nad modyfikatorami dostępu metod `set` i `get` właściwości. Dlatego można oznaczyć właściwość jako publiczną, ale oznaczyć metodę `set` jako prywatną lub chronioną. Poniżej znajduje się przykładowy kod:

Właściwość z mieszanym poziomem dostępu w C#:

```
private string _userId;
public string UserId {
  get { return _userId; }
  internal set { userId = value; }
}
```

Właściwość z mieszanym poziomem dostępu w VB:

```
Private _userId As String
Public Property UserId() As String
  Get
    Return _userId
  End Get
  Friend Set(ByVal value As String)
    _userId = value
  End Set
End Property
```

Wieloznaczne przestrzenie nazw

W przypadku dużych projektów mogą wystąpić konflikty przestrzeni nazw między sobą oraz z platformą .NET (przestrzeń nazw `System`). Wcześniej nie można było rozwiązać takich wieloznacznych referencji i na etapie kompilacji pojawiał się wyjątek.

.NET 2.0 umożliwia programistom zdefiniowanie własnej przestrzeni nazw `System` bez blokowania dostępu do wersji platformy .NET. Załóżmy, że programista zdefiniował przestrzeń nazw `System` i nagle utracił możliwość dostępu do jej globalnej wersji. W języku C#

należy dodać wtedy słowo kluczowe `global` wraz z kwalifikatorem aliasu przestrzeni nazw `::`, tak jak w poniższym wierszu:

```
global::System.Double myDouble;
```

W Visual Basic składnia jest podobna, ale używane słowo kluczowe to `Global`:

```
Dim myDouble As Global.System.Double
```

Ponadto można zdefiniować alias, kiedy używa się przestrzeni nazw lub importuje je. Tego aliasu można następnie używać do wskazywania typów danej przestrzeni nazw. Załóżmy, że wystąpił konflikt z przestrzenią nazw `System.IO`. Można wtedy zdefiniować alias przy jej importowaniu:

C#
```
using IoAlias = System.IO;
```

VB
```
Imports IoAlias = System.IO
```

Następnie można używać typów za pomocą aliasu. Oczywiście Visual Studio udostępnia pełną obsługę mechanizmu IntelliSense dla tych elementów. Poniżej znajduje się kod przedstawiający sposób używania zdefiniowanego wcześniej aliasu. Warto zwrócić uwagę na nową składnię języka C#, która umożliwia stosowanie operatora w postaci podwójnego dwukropka:

```
Nowa składnia języka C#
IoAlias::FileInfo file;

Stara składnia języka C#
IoAlias.FileInfo file;

VB
Dim file as IoAlias.FileInfo
```

Rozszerzenia języka Visual Basic

Doświadczeni użytkownicy języka Visual Basic ucieszą się na wieść, że w Visual Studio 2005 została przywrócona właściwość zmień i kontynuuj! Jednak jest to funkcja środowiska IDE. W rzeczywistości wiele nowych cech IDE jest specyficznych dla języka. W niniejszej książce chcemy opisać większość z nich (jeśli nie wszystkie). Rozszerzenia IDE specyficzne dla języka Visual Basic obejmują poniższe właściwości:

- Programowanie za pomocą skrótów `My`.
- Właściwość zmień i kontynuuj.
- Fragmenty kodu.
- Rozszerzenia mechanizmu IntelliSense.
- Modyfikowanie atrybutów w oknie *Properties*.

- Poprawianie błędów i ostrzeżenia.
- Asystent do obsługi wyjątków.
- Dokumentacja XML.
- Okno *Document Outline*.
- Okno projektowe dla projektów.
- Okno projektowe dla ustawień.
- Okno projektowe dla zasobów.

Dzięki powyższym rozszerzeniem (a także innym) korzystanie z języka Visual Basic jest bardzo wygodne. W następnych podpunktach skoncentrujemy się jednak na samym języku. Wskażemy nowinki specyficzne dla Visual Basic, które są niezwykle atrakcyjne w wersji 2005.

Instrukcja Continue

Nowa instrukcja `Continue` w języku VB umożliwia programistom przeskoczenie do następnej iteracji pętli. Instrukcji `Continue` można używać w pętlach `Do`, `For` i `While`. Jeśli potrzebne jest skrócenie pętli i natychmiastowe przejście do następnej iteracji, wystarczy użyć instrukcji `Continue For/Do/While`, tak jak w poniższym kodzie:

```
Sub ProcessCustomers(ByVal customers() As Customer)
  Dim i As Integer
  For i = 0 To customers.GetUpperBound(0)
    If customers(i).HasTransactions = False Then Continue For
    ProcessCustomer(customers(i))
  Next
End Sub
```

Typy bez znaku

Programiści języka Visual Basic mogą teraz używać typów całkowitoliczbowych bez znaku (`UShort`, `UInteger` i `ULong`). Ponadto najnowsza wersja Visual Basic udostępnia dodatkowy typ ze znakiem, `SByte`. Te nowe typy umożliwiają programistom łatwiejsze wywoływanie funkcji interfejsu API systemu Windows, które często zwracają typy bez znaku. Jednak takie typy bez znaku nie są obsługiwane przez specyfikację wspólnego języka (ang. *Common Language Specification — CLS*), dlatego jeśli programista napisze kod bazujący na tych nowych typach, kod zgodny z CLS może z nim nie współpracować.

Operator IsNot

Nowy operator `IsNot` języka Visual Basic umożliwia programistom sprawdzenie, czy dwa obiekty różnią się od siebie. Oczywiście można to było zrobić także w poprzednich wersjach, łącząc instrukcje `Not` i `Is`, na przykład `If Not myCustomer Is Nothing`. Jednak teraz programiści mogą korzystać z prostszej składni operatora `IsNot`, takiej jak w poniższym

wierszu kodu. Warto zauważyć, że działanie tego wiersza jest takie samo, jak przestawionej wcześniej instrukcji `Not...Is`.

```
If cust IsNot Nothing Then
```

Blok Using

Programiści języka Visual Basic, którzy używali przez pewien czas języka C#, bez wątpienia ucieszą się z możliwości definiowania zasięgu obiektów za pomocą bloku `Using`. Dzięki temu blokowi programiści języka C# mogą zagwarantować zwolnienie zasobów w momencie zakończenia wykonywania danego bloku przez aplikację. Obecnie ta właściwość została dodana także do języka Visual Basic. Załóżmy, że programista chce nawiązać połączenie z bazą danych. Teraz może to zrobić w bloku `Using`. Dzięki temu gdy program z jakiegoś powodu zakończy wykonywanie tego bloku, obiekt zdefiniowany w instrukcji `Using` (obiekt połączenia SQL) zostanie usunięty w odpowiedni sposób. Poniższy kod ilustruje używanie tej nowej właściwości:

```
Using cnn As New System.Data.SqlClient.SqlConnection(cnnStr)
  ' Tu kod używający połączenia z SQL
End Using
```

Dostęp do formularzy podobny jak w Visual Basic 6

Programiści znający wersję 6. języka Visual Basic (jeszcze sprzed platformy .NET) przypomną sobie możliwość bezpośredniego dostępu do właściwości i metod formularza poprzez jego nazwę. W poprzednich wersjach .NET programiści musieli utworzyć egzemplarz formularza, aby uzyskać dostęp do jego właściwości. W Visual Basic 8 ponownie możliwy jest dostęp do składowych formularza poprzez jego nazwę.

Jawne zerowe dolne ograniczenie w tablicach

W starszych wersjach języka Visual Basic (sprzed .NET) programiści mogli określić górne i dolne ograniczenie tablicy, używając słowa kluczowego `To`. Można było na przykład zdefiniować tablicę rozpoczynającą się od 1, a kończącą na 10. Dzięki temu kod, w którym używane były tablice, był bardzo czytelny. Jednak pojawienie się platformy .NET i specyfikacji CLS spowodowało, że dolne ograniczenie we wszystkich tablicach musiało być równe zero (0). Oznacza to, że każda tablica platformy .NET musi rozpoczynać się od elementu zerowego. W Visual Basic 8 nie uległo to zmianie, znów można jednak określić, że tablica rozpoczyna się od 0, a kończy na górnym ograniczeniu, co sprzyja czytelności kodu. Dlatego można definiować tablice w poniższy sposób, jednak dolne ograniczenie musi równać się zero (0):

```
Dim myIntArray(0 To 9) As Integer
```

Przeciążanie operatorów

Jeśli często piszesz biblioteki, w końcu natrafisz na klasę, w której będziesz musiał zdefiniować działanie operatorów dodawania (+), odejmowania (-), mnożenia (*), większości (>) i innych. Jeśli na przykład trzeba obliczyć wynik dodawania dwóch wersji klasy za pomocą operatora +, potrzebna jest definicja jego działania dla tej klasy. W poprzednich wersjach języka Visual Basic nie można było utworzyć takiej definicji, jednak Visual Basic 8 umożliwia to. Jest to tak zwane *przeciążanie operatorów*.

Do definiowania nowego operatora służy słowo kluczowe `Operator` (używane zamiast `Sub` lub `Function`), po którym następuje symbol przeciążanego operatora (+, &, *, <> i tak dalej). Można następnie dodać ciało tej „funkcji" jak każdej innej. Może ona przyjmować parametry i zwracać wartość. Jeśli programista chce umożliwić dodawanie do siebie dwóch obiektów, może zdefiniować operator + przyjmujący te obiekty jako parametry i zwracający trzeci obiekt jako wynik. Ilustruje to poniższy kod:

```
Public Operator +(ByVal obj1 As MyObject, ByVal obj2 As MyObject) As MyObject
  ' Obliczanie wartości nowego obiektu i zwracanie go
End Operator
```

Niestandardowe zdarzenia

Programiści języka Visual Basic mają teraz kontrolę nad rejestrowaniem delegatów, ponieważ mogą używać zdarzeń zdefiniowanych przez użytkownika. Nowa wersja Visual Basic obejmuje słowo kluczowe `Custom`, którego można używać przy definiowaniu zdarzeń. Po użyciu tego słowa kluczowego do zadeklarowania zdarzenia trzeba zdefiniować akcesory dla metod `AddHandler`, `RemoveHandler` i `RaiseEvent`. Te akcesory przesłaniają domyślne działanie zdarzenia kodem napisanym przez programistę. Jest to przydatne w sytuacjach, kiedy wszystkie zdarzenia mają być uruchamiane asynchronicznie lub potrzebna jest pełna kontrola nad tymi operacjami.

Rozszerzenia języka C#

W wersji 2005 język C# został rozszerzony o nowe możliwości. Opisaliśmy już kilka wspólnych nowinek, takich jak typy ogólne i dopuszczające wartość `null`. Ponadto programiści języka C# mają dostęp do nowych właściwości środowiska IDE. Niektóre z tych funkcji to:

- Fragmenty kodu.
- Refaktoryzacja.
- Rozszerzenie mechanizmu IntelliSense.
- Kreatory kodu.
- Właściwości projektów.

W niniejszej książce opiszemy te właściwości. Jednak w tym miejscu skoncentrujemy się na specyficznych rozszerzeniach języka C# w wersji 2005.

Więcej informacji dotyczących języka C# i omawianych tu zagadnień znajduje się w udostępnianym przez Microsoft dokumencie „C# Language Specification 2.0". Ten dokument można pobrać ze strony *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnvs05/html/cs3spec.asp*. Pod tym samym adresem znajduje się kompletny przegląd języka C# oraz zestaw samouczków. Ponadto gotowa jest już specyfikacja języka C# 3.0, która jest dostępna do przeglądu i recenzji.

Metody anonimowe

Pojęcie *metody anonimowe* może przy pierwszym napotkaniu wydawać się nieco skomplikowane. Są to jednak po prostu nienazwane bloki kodu (nie metody) przekazywane bezpośrednio do delegata. Po pierwsze, ta właściwość jest dostępna tylko w języku C#. Po drugie, jest przydatna tylko wtedy, kiedy nie są potrzebne wszystkie możliwości, jakie dają delegaty — nie trzeba używać wielu jednostek nasłuchujących ani nie jest potrzebna możliwość kontroli tego, jakie jednostki nasłuchują (poprzez ich dodawanie i usuwanie).

Metody anonimowe najłatwiej jest zrozumieć, porównując je ze standardowym sposobem używania delegatów. Aby użyć delegatów w poprzednich wersjach języka C#, trzeba było napisać metodę wywoływaną przez tego delegata, a następnie powiązać ją z delegatem.

Przyjrzyjmy się na przykład sposobowi łączenia kodu ze zdarzeniem `Click` przycisku (`System.Windows.Forms.Button`). Po pierwsze, zdarzenie `Click` to `System.EventHandler` (lub delegat). Trzeba upewnić się, że kod metody jest powiązany z delegatem. Załóżmy, że kod znajduje się w metodzie, która wygląda następująco:

```
private void button1_Click(object sender, EventArgs e) {
  label1.Text = "textBox.Text";
}
```

Następnie należy powiązać metodę z delegatem. Oczywiście Visual Studio wykonuje na zapleczu wszystkie potrzebne operacje. Można jednak samodzielnie napisać służący do tego kod. Ponadto Visual Studio obsługuje jedynie wiązanie delegatów kontrolek interfejsu użytkownika (zdarzeń) z metodami. Programista odpowiada za wiązanie wszystkich pozostałych delegatów — zarówno tych niestandardowych, jak i będących częścią platformy. Poniższy kod pokazuje, w jaki sposób Visual Studio wiąże metodę `button1_Click` ze zdarzeniem `Click` klasy `Button`:

```
this.button1.Click += new System.EventHandler(this.button1_Click);
```

Jak widać, trzeba napisać metodę z kodem oraz powiązać ją z delegatem. Przyjrzyjmy się teraz, jakie możliwości dają metody anonimowe. Jak już wspomnieliśmy, można przekazać kod bezpośrednio do delegata. Dlatego można dodać poniższy wiersz kodu do konstruktora formularza (po wywołaniu metody `InitializeComponents`):

```
this.button1.Click += delegate {
  label1.Text = "Do widzenia";
};
```

Jak widać w powyższym kodzie, używanie metod anonimowych wiąże się z używaniem słowa kluczowego `delegate`. Oczywiście delegaty mogą przyjmować parametry, dlatego po słowie `delegate` można umieścić opcjonalną listę parametrów (nie ma jej w powyższym kodzie). Na końcu znajduje się lista instrukcji (blok kodu). To ten kod, ograniczony nawiasami klamrowymi, jest przekazywany do delegata anonimowo (bez nazwy metody).

Metoda anonimowa ma dostęp do zmiennych znajdujących się w zasięgu w miejscu tworzenia takiej metody. Te zmienne to tak zwane *zmienne zewnętrzne* metody anonimowej (dla odróżnienia od *zmiennych wewnętrznych*, które są definiowane wewnątrz metody). Jeśli zmienna zewnętrzna jest używana w metodzie anonimowej, zostaje przez nią *przechwycona*. Czas życia takiej przechwyconej zmiennej jest zależny od *usunięcia* przez mechanizm przywracania pamięci delegata, a nie samej metody.

Ten punkt stanowi jedynie krótkie wprowadzenie do metod anonimowych. Umożliwiają one pisanie dość złożonego kodu (który może być mało czytelny). Jak łatwo się domyślić, przekazywanie wierszy kodu jako parametrów wymaga starannego namysłu, jeśli chce się uniknąć problemów.

Klasy statyczne

Nowa wersja platformy .NET zapewnia obsługę klas statycznych. *Klasa statyczna* to taka, w której każda składowa jest zadeklarowana jako składowa statyczna (w przeciwieństwie do składowych egzemplarza). Użytkownicy tej klasy nie muszą tworzyć jej egzemplarzy. Platforma gwarantuje nawet, że nie można utworzyć egzemplarza klasy statycznej. Platforma .NET udostępnia wiele takich klas, jednak język Visual Basic na razie ich nie obsługuje. Robi to jednak język C#. Dlatego to tu przedstawiamy przykładowy kod bazujący na klasach statycznych.

W języku Visual Basic można utworzyć coś podobnego do klas statycznych, tworząc klasę o prywatnym konstruktorze. Ponadto należy oznaczyć wszystkie składowe tej klasy jako `Shared`. Wadą takiego rozwiązania jest brak obsługi tego mechanizmu przez kompilator czy brak możliwości używania konstruktora statycznego.

Definiowanie klas statycznych

Klasy statyczne tworzy się poprzez dodanie słowa kluczowego `Static` do deklaracji klasy, tak jak w poniższym wierszu kodu:

```
static class ProjectProperties { ...
```

Jak już wspomnieliśmy, nie można tworzyć egzemplarzy klasy, która została zadeklarowana jako statyczna. Trzeba jednak pamiętać o zadeklarowaniu wszystkich jej składowych jako statycznych (kompilator nie zakłada, że są właśnie takie). Jednak kompilator może sprawdzić w klasie statycznej, czy przypadkowo nie znalazła się w niej składowa egzemplarza. Próba umieszczenia zmiennej niestatycznej w klasie statycznej wywoła błąd kompilacji. Dotyczy to zarówno składowych publicznych, jak i prywatnych. Listing 3.1 przedstawia prostą klasę statyczną i jej składowe.

Listing 3.1. *Klasa statyczna*

```
namespace StaticClasses {
  static class ProjectProperties {
    static string _projectName;
    static ProjectProperties() {
      _projectName = "JakiśNowyProjekt";
    }
    public static string Name {
      get { return _projectName; }
    }
    public static DateTime GetDueDate() {
      // Pobiera datę planowanego oddania projektu
      DateTime dueDate = DateTime.Now.AddDays(10);
      return dueDate;
    }
  }
}
```

Klasy statyczne są automatycznie oznaczane jako zamknięte. Oznacza to, że nie można dziedziczyć po klasach statycznych.

Konstruktory i klasy statyczne

Nie można utworzyć konstruktora dla klasy statycznej. Jednak jeśli potrzebna jest funkcja podobna do konstruktora (na przykład do ustawiania początkowych wartości), można napisać tak zwany *konstruktor statyczny*. Listing 3.1 przedstawia przykładowy konstruktor statyczny o nazwie `ProjectProperties`. Warto zauważyć, że inicjuje on wartość składowej statycznej `_projectName`.

Środowisko CLR platformy .NET wczytuje klasy statyczne automatycznie w czasie wczytywania zawierających je przestrzeni nazw. Ponadto w momencie wywołania składowej statycznej środowisko CLR automatycznie wywołuje konstruktor statyczny. W celu wywołania tego specjalnego konstruktora nie trzeba tworzyć egzemplarza klasy statycznej (nie jest to nawet możliwe).

Używanie dwóch wersji tego samego podzespołu

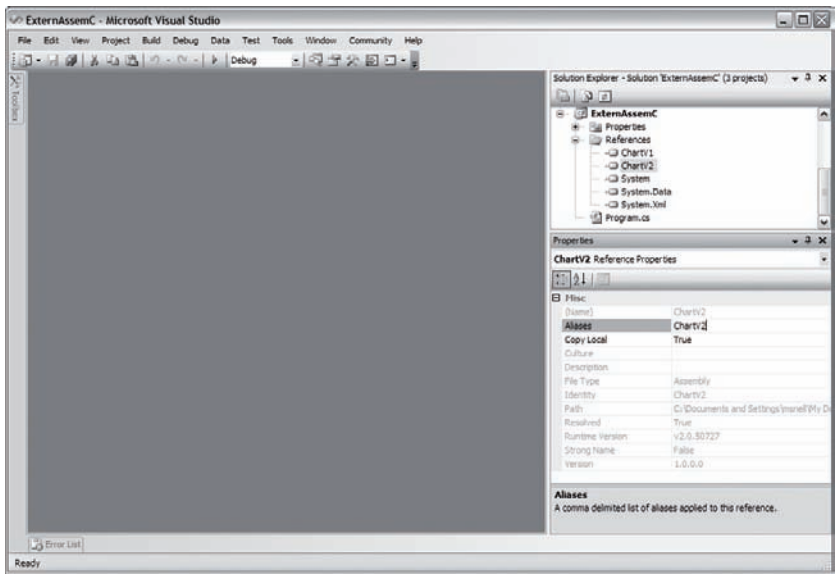
Programiści czasem potrzebują właściwości starszej wersji komponentu, a jednocześnie chcą używać jego najnowszej wersji. Często jest to wynikiem zmian wprowadzanych w komponentach bez uwzględnienia zgodności wstecz. W takich przypadkach możliwości są ograniczone do całkowitego przerzucenia się na nowy komponent lub pozostania przy starszej wersji. Język C# umożliwia trzecie rozwiązanie — używanie obu wersji za pomocą zewnętrznego aliasu podzespołu.

Podstawowym problemem związanym z korzystaniem z wielu wersji tego samego podzespołu jest rozwiązywanie konfliktów między nazwami składowych znajdujących się w tej samej przestrzeni nazw. Załóżmy, że programista używa podzespołu generującego wykresy. Odpowiedzialna jest za to klasa `Chart` działająca w przestrzeni nazw `Charting`. Kiedy zostanie udostępniona następna wersja podzespołu, programista będzie chciał za-

chować cały istniejący kod, jednak w nowym kodzie będzie chciał używać najnowszej wersji. W języku C# można to zrobić, wykonując kilka operacji.

Po pierwsze, trzeba zdefiniować alias nowego podzespołu. W tym celu należy użyć okna *Properties* dla wybranej referencji. Rysunek 3.1 pokazuje, jak ustawić taki alias. Warto zauważyć, że ustawiany jest alias dla drugiej wersji podzespołu (`ChartV2`). Gwarantuje to, że wywołania do klasy `Charting.Chart` wciąż będą kierowane do pierwszej wersji (`ChartV1`).

Rysunek 3.1. *Definiowanie aliasu*



Następnie w pliku z kodem, który ma używać nowej wersji podzespołu, trzeba zdefiniować zewnętrzny alias. W tym celu należy przejść do początku pliku (przed instrukcje `Using`) i wpisać instrukcję rozpoczynającą się od słowa kluczowego `extern`. Poniższy wiersz pokazuje, jaki kod należy umieścić na początku pliku, aby używać drugiej wersji komponentu `Charting`:

```
extern alias ChartV2;
```

Do używania składowych nowej wersji podzespołu służy operator `::` (podobnie jak w przypadku innych, podobnych aliasów języka C#), tak jak w poniższym wierszu kodu:

```
ChartV2::Charting.Chart.GenerateChart();
```

Podzespoły zaprzyjaźnione

Język C# 2.0 umożliwia połączenie podzespołów ze względu na dostęp do wewnętrznych mechanizmów. Oznacza to, że można zdefiniować wewnętrzne składowe, ale udostępnić je zewnętrznym podzespołom. Ta możliwość jest przydatna, jeśli zamierzasz rozbić podzespół na kilka plików fizycznych, ale wciąż chcesz, aby ich elementy były dostępne we wszystkich podzespołach.

Podzespoły zaprzyjaźnione *nie umożliwiają* dostępu do składowych prywatnych.

Do oznaczania podzespołów udostępniających swe wewnętrzne składowe zaprzyjaźnionym podzespołom służy nowy atrybut, `InternalsVisibleToAttribute`. Ten atrybut jest stosowany na poziomie podzespołu. Do tego atrybutu należy przekazać nazwę oraz znacznik klucza publicznego zewnętrznego podzespołu. Kompilator będzie traktował te podzespoły jako zaprzyjaźnione. Podzespół zawierający atrybut `InternalsVisibleToAttribute` udostępnia swe wewnętrzne składowe drugiemu podzespołowi (ale już nie na odwrót). Można uzyskać ten sam efekt, używając opcji kompilatora ustawianych w wierszu poleceń.

Podzespoły zaprzyjaźnione, jak to często bywa, mają pewne wady. Zdefiniowanie podzespołu jako zaprzyjaźnionego z innym podzespołem powoduje ich ścisłe powiązanie. Takie podzespoły muszą współwystępować, aby były przydatne. Oznacza to, że nie stanowią już odrębnych jednostek. Może to spowodować nieład, a takimi podzespołami trudniej jest zarządzać. Często lepiej jest nie korzystać z tej właściwości dopóty, dopóki nie zajdzie istotna potrzeba.

Rozszerzenia platformy .NET 2.0

Ponieważ w platformie .NET wprowadzono tak wiele nowinek, nie warto rozpoczynać ich opisu w tym miejscu. Oczywiście postaramy się przedstawić je w następnych rozdziałach. Chcemy jednak wyróżnić kilka kluczowych rozszerzeń, które sprawiają, że nowa wersja platformy .NET to poważny krok naprzód. Poniżej przedstawiamy niektóre z tych nowości:

- **Obsługa systemów 64-bitowych** — teraz można kompilować aplikacje .NET przeznaczone dla 64-bitowych wersji systemów operacyjnych. Obejmuje to obsługę systemów natywnych oraz zgodność z emulatorem WOW64 (umożliwia uruchamianie 32-bitowych aplikacji w systemach 64-bitowych).
- **Obsługa ACL** — programiści używający .NET mogą teraz korzystać z listy kontroli dostępu (ang. *Access Control List — ACL*) do zarządzania z poziomu kodu uprawnieniami do zasobów. W przestrzeni nazw `IO` (i innych) znajdują się nowe klasy, które pomagają użytkownikiem nadawać uprawnienia do plików i wykonywać inne operacje.
- **Strumienie uwierzytelnione** — nowa klasa `NegotiateStream` umożliwia bezpieczne (zaszyfrowane przy użyciu SSL) uwierzytelnianie między klientem a serwerem (jednostką nasłuchującą) w czasie przesyłania informacji w sieci. Dzięki tej klasie można bezpiecznie przesyłać dane uwierzytelniające klienta poprzez personifikację lub delegację. Ponadto nowa klasa `SslStream` umożliwia szyfrowanie danych w czasie ich przesyłania.
- **Data Protection API (DPAPI)** — w .NET 2.0 zmieniła się obsługa interfejsu DPAPI. Obecnie obejmuje ona możliwość szyfrowania po stronie serwera hasła i łańcuchów znaków połączenia. Programiści mogli używać tych opcji także w poprzednich wersji .NET, pobierając specjalną nakładkę. W wersji 2.0 dostęp do tego interfejsu jest wbudowany w platformę.

- **Wykrywanie zmian w sieci** — aplikacja może teraz otrzymywać powiadomienia o utracie połączenia z siecią. Dzięki klasie `NetworkChange` programiści wiedzą, kiedy komputer, na którym działa aplikacja, utracił bezprzewodowe połączenie lub zmienił adres IP.
- **Obsługa protokołu FTP** — przestrzeń nazw `System.Net` udostępnia obecnie klasy służące do obsługi protokołu FTP. Programiści mogą używać klas `WebRequest`, `WebResponse` i `WebClient` do przesyłania oraz pobierania plików poprzez ten protokół.
- **Rozszerzenia dotyczące globalizacji** — nowa wersja platformy umożliwia programistom definiowanie własnych niestandardowych ustawień lokalnych. Daje to niezwykłą elastyczność w zakresie obsługi w aplikacjach informacji dotyczących kultury. Ponadto platforma .NET 2.0 zapewnia ulepszoną obsługę Unicode.
- **Większa kontrola nad pamięcią podręczną** — programiści mogą teraz używać przestrzeni nazw `System.Net.Cache` do programowej kontroli pamięci podręcznej.
- **Obsługa szeregowych urządzeń wejścia-wyjścia** — w przestrzeni nazw `System.IO` znajduje się nowa klasa `SerialPort`. Ta klasa umożliwia programistom używanie urządzeń, które łączą się z portami szeregowymi komputera.
- **Rozszerzona obsługa protokołu SMTP** — przestrzeń nazw `System.Net.Mail` umożliwia programistom przesyłanie listów elektronicznych za pomocą serwera SMTP.
- **Transakcje** — programiści używający .NET mogą teraz korzystać z nowej przestrzeni nazw `System.Transactions`, która umożliwia klasom utworzonym dla platformy .NET łatwe uczestniczenie w transakcjach rozproszonych. Do obsługi transakcji służy koordynator transakcji rozproszonych Microsoftu (ang. *Microsoft Distributed Transaction Coordinator — MSDTC*).

Nowe właściwości w podstawowych technologiach

ADO.NET, ASP.NET i formularze Windows zostały rozbudowane w wersji 2005. Każda z tych technologii mogłaby być tematem odrębnej książki. Na przykład ADO.NET obsługuje obecnie typy definiowane przez użytkownika (ang. *User-Defined Type — UDT*) i asynchroniczne operacje w bazie danych. Zarówno ASP, jak i formularze Windows udostępniają wiele nowych kontrolek (oraz rozbudowane wersje starych). W obu tych technologiach możliwe jest wiązanie danych bez konieczności pisania kodu. Radzimy dokładnie zapoznać się z tymi zagadnieniami i przyjrzeć się wielu nowym właściwościom w zakresie tych technologii.

Podsumowanie

W tym rozdziale przedstawiliśmy podstawowe rozszerzenia języka .NET, które są kluczowe dla programistów, ponieważ ułatwiają szybsze pisanie lepszego kodu. Nowinki takie jak typy ogólne pomagają zapewnić bezpieczeństwo typów w kolekcjach, co pozwala zmniejszyć liczbę błędów. Typy dopuszczające wartość `null` pozwalają pisać kod bez konieczności przypisywania wartości niezainicjowanym zmiennym i obsługi „magicznych liczb". Te i podobne nowości wprowadzone w językach C# i Visual Basic przyczyniają się do ewolucji omawianego zestawu narzędzi i zwiększają produktywność programistów.

Na zakończenie rozdziału pokrótce opisaliśmy niektóre z nowych elementów platformy .NET. Dostępnych jest mnóstwo nowinek. Platforma staje się tak rozbudowana, że programiści (i książki) często są wyspecjalizowani w danym obszarze. Radzimy przejrzeć listę rozszerzeń, a następnie kontynuować eksplorację w zakresie interesującej Cię dziedziny.