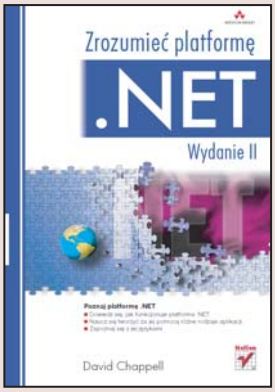
Zrozumieć platformę .NET. Wydanie II



Autor: David Chappell
Tłumaczenie: Anna Trojan
ISBN: 83-246-0755-2
Tytuł oryginału: Understanding .NET (2nd Edition)
Format: B5, stron: 312

Poznaj platformę .NET

- Dowiedz się, jak funkcjonuje platforma .NET
- Naucz się tworzyć za jej pomocą różne rodzaje aplikacji
- Zapoznaj się z jej językami

Wprowadzenie platformy programistycznej .NET okazało się przełomem w programowaniu aplikacji dla systemu Windows. Obsługiwane przez nią technologie, takie jak ADO.NET czy ASP.NET, pozwalają szybko i łatwo tworzyć różnorodne programy dla tego systemu, a także witryny oraz usługi internetowe. Zestaw elementów .NET składa się na jedną z najpotężniejszych obecnie platform programistycznych, a podstawowym narzędziem umożliwiającym korzystanie z możliwości jej najnowszej, drugiej, wersji jest Visual Studio 2005.

„Zrozumieć platformę .NET. Wydanie II" to krótkie wprowadzenie w niezwykle bogaty świat platformy .NET. Z książki tej dowiesz się, jak działa wspólne środowisko uruchomieniowe (CLR) oraz biblioteka klas .NET Framework. Poznasz możliwości Visual Studio 2005 oraz podstawowe języki platformy, takie jak C#, Visual Basic i C++. Nauczysz się tworzyć różne rodzaje programów przy użyciu podstawowych technologii platformy .NET, między innymi aplikacje webowe za pomocą ASP.NET czy bazodanowe w ADO.NET. Książka ta pozwoli Ci rozpocząć korzystanie z olbrzymich możliwości platformy .NET.

- Biblioteka klas .NET Framework
- Wspólne środowisko uruchomieniowe (CLR)
- Przegląd języków .NET
- Visual Studio 2005
- Tworzenie aplikacji webowych za pomocą ASP.NET
- Używanie ADO.NET do obsługi danych
- Programowanie rozproszone

Dzięki tej książce szybko wkroczysz w świat platformy .NET

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

Spis treści

3

Języki .NET

Zrozumienie języka opartego na CLR rozpoczyna się od zrozumienia CLR

Wspólne środowisko uruchomieniowe (CLR) zostało zaprojektowane specjalnie w celu wspierania wielu języków. Jednak ogólnie rzecz biorąc, języki zbudowane na bazie CLR zazwyczaj mają ze sobą wiele wspólnego. Definiując duży zbiór podstawowej semantyki, CLR jednocześnie definiuje znaczną część typowego języka programowania, który je wykorzystuje. Podstawową sprawą przy opanowaniu dowolnego języka opartego na CLR jest na przykład zrozumienie, w jaki sposób standardowe typy zdefiniowane przez CLR są odwzorowywane na ten język. Oczywiście, konieczne jest także nauczenie się składni języka, w tym struktur sterujących, które ten język zapewnia. Jednak wiedząc już, co oferuje CLR, łatwiej jest zrozumieć dowolny język zbudowany na bazie wspólnego środowiska uruchomieniowego.

Niniejszy rozdział opisuje C# i Visual Basic — dwa najważniejsze języki oparte na CLR. Omawia także krótko C++/CLI, czyli rozszerzenia, które pozwalają programistom C++ na pisanie kodu opartego na CLR. Celem nie jest dostarczenie wyczerpującego opisu każdej cechy tych języków — do tego potrzeba by następnych trzech książek — ale raczej naszkicowanie, jak wyglądają te języki i w jaki sposób wyrażają podstawową funkcjonalność zapewnianą przez CLR. W całym rozdziale opisano wersje tych języków dostępne w Visual Studio 2005.

Perspektywa: co z Javą dla .NET Framework?

Od samego początku Microsoft dostarczał J#, wersję języka programowania Java dla .NET Framework. Język ten implementuje składnię i zachowanie Javy na bazie CLR. Dobrze mieć taką możliwość, ale kiedy używanie J# ma jakikolwiek sens?

Istnieją dwie sytuacje, w których J# będzie dobrym wyborem. Pierwsza ma miejsce, gdy istniejący kod w Javie musi zostać przeniesiony na .NET Framework. Kod ten mógł zostać napisany za pomocą Visual J++, narzędzia sprzed ery .NET produkcji Microsoftu, bądź też za pomocą standardowych narzędzi do języka Java, takich jak Eclipse. W obu przypadkach istnienie J# ułatwia przeniesienie tego kodu do świata .NET. Należy jednak zwrócić uwagę na fakt, iż .NET Framework nie implementuje popularnych technologii Javy, takich jak Java Server Pages (JSP) czy Enterprise JavaBeans, zatem nie każdy kod w tym języku można łatwo przenieść. Mimo to typowa logika biznesowa Javy może być uruchomiona na .NET Framework bez większego nakładu pracy. Istnieje nawet narzędzie konwersji binarnej, które pozwala przekonwertować kod bajtowy Javy na MSIL, przydatne wtedy, gdy nie mamy już dostępu do kodu źródłowego aplikacji.

Druga okazja, w której wybranie J# jest dobrym rozwiązaniem, to sytuacja mająca miejsce, gdy programista Javy przenosi się na .NET Framework. Ludzie często bardzo się przywiązują do konkretnej składni, dlatego praca w znajomym języku może ułatwić przejście. Nadal jednak trudno jest powiedzieć, że tworzenie kodu .NET Framework w Javie jest dobrym pomysłem. Microsoft wyraźnie skupia się na C# oraz VB jako głównych językach do tworzenia nowych aplikacji .NET, dlatego wybranie innego języka jest zawsze nieco ryzykowne. Ponadto brak standardowych pakietów Javy w .NET oznacza, że programista przechodzący z Javy nadal nie będzie czuł się w .NET jak w domu — ciągle jeszcze będzie musiał się dużo nauczyć. Jednak biorąc pod uwagę podobieństwa pomiędzy Javą i C#, nawet najbardziej zagorzały fan Javy nie powinien być zmartwiony przesiadką na C#.

Obsługa Javy przez Microsoft ma wyraźnie zachęcać do przenoszenia kodu na platformę .NET Framework i kusić programistów tego języka, a nie pomagać programistom w tworzeniu nowego oprogramowania w Javie. Linie frontu są jasne: to .NET przeciwko światowi Javy. Bez wątpienia ma to jednak pozytywne strony. Dwa obozy z potężnymi technologiami, każdy z silną pozycją — to idealna sytuacja. Każdy z nich dostarcza innowacji, które od razu podchwytuje druga strona, dzięki czemu w efekcie końcowym na konkurencji zyskują wszyscy.

C#

C# jest zorientowanym obiektowo językiem programowania ze składnią podobną do języka C

Jak sugeruje sama nazwa, C# jest członkiem rodziny języków programowania C. W przeciwieństwie do C, C# jest jawnie zorientowany obiektowo. Z kolei przeciwieństwie do C++, który był najpopularniejszym językiem zorientowanym obiektowo z tej rodziny, C# nie jest tak bardzo skomplikowany. Zamiast tego C# został zaprojektowany jako język przystępny dla każdego, kto ma jakieś doświadczenie z C++ czy Javą.

Visual Studio 2005 implementuje wersję 2.0 języka C#

Zaprojektowany przez Microsoft C# pierwszy raz pojawił się wraz z wydaniem Visual Studio .NET w 2002 roku. W Visual Studio 2005 zaimplementowano C# 2.0, zbudowany na podstawie tej początkowej wersji sprzed trzech lat. Jak w przypadku wszystkich zagadnień omówionych w niniejszej książce, opisana tutaj wersja języka jest wersją z wydania z 2005 roku.

Microsoft dostarcza najpopularniejsze kompilatory C#, jednak nie jedyne

Najpopularniejszym narzędziem, za pomocą którego tworzy się dzisiaj kod w C#, jest Microsoft Visual Studio. Nie jest to jednak jedyny możliwy wybór. W ramach .NET Framework Microsoft dostarcza także kompilator wiersza poleceń o nazwie *csc.exe*; istnieje także open source'owy kompilator C#. Jednak biorąc pod uwagę rozbudowane wsparcie dla tworzenia aplikacji napisanych w C# i opartych na CLR w Visual Studio, trudno sobie wyobrazić, że alternatywne rozwiązania przyciągną wielu programistów.

Przykład C#

Jak większość języków programowania, C# definiuje typy danych, struktury sterujące i tak dalej. W przeciwieństwie do starszych języków, C# robi to wszystko, budując w oparciu o CLR. By zilustrować to twierdzenie, poniżej znajduje się prosty przykład C#:

```
// Przykład C#
interface IMath
{
   int Factorial(int f);
   double SquareRoot(double s);
}

class Compute : IMath
{
   public int Factorial(int f)
```

```
   {
      int i;
      int result = 1;
      for (i=2; i<=f; i++)
         result = result * i;
      return result;
   }

   public double SquareRoot(double s)
   {
      return System.Math.Sqrt(s);
   }
}

class DisplayValues
{
   static void Main()
   {
      Compute c = new Compute();
      int v;
      v = 5;
      System.Console.WriteLine(
         "{0} silnia: {1}",
         v, c.Factorial(v));
      System.Console.WriteLine(
         "Pierwiastek kwadratowy z {0}: {1:f4}",
         v, c.SquareRoot(v));
   }
}
```

Każdy program w C# zbudowany jest z jednego lub większej liczby typów

Program rozpoczyna się od komentarza, oznaczonego przez dwa prawe ukośniki, który zawiera krótki opis celów programu. Ciało programu składa się z trzech typów: interfejsu o nazwie `IMath` oraz dwóch klas — `Compute` i `DisplayValues`. Wszystkie programy w C# składają się z pewnej liczby typów, z których najbardziej zewnętrzne to klasy, interfejsy, struktury, typy wyliczeniowe lub delegaty (mogą się tu również pojawić, omówione później, przestrzenie nazw). Wszystkie metody, pola i inne składniki typów muszą przynależeć do jednego z wymienionych typów, co oznacza, że C# nie pozwala ani na globalne zmienne, ani na globalne metody.

Interfejs C# jest wyrażeniem interfejsu CTS

Interfejs `IMath`, będący w C# inkarnacją typu „interfejs" ze wspólnego systemu typów CTS, opisanego w rozdziale 2., definiuje metody `Factorial` oraz `SquareRoot`. Każda z tych metod przyjmuje jeden parametr

i zwraca wynik liczbowy. Parametry te przekazywane są za pomocą wartości, co jest domyślnym zachowaniem w C#. Oznacza to, że zmiany wartości parametru wewnątrz metody nie będą widziane przez program wywołujący po zakończeniu metody. Umieszczenie słowa kluczowego `ref` przed parametrem sprawia, że parametr jest przekazywany przez referencję, co oznacza, że zmiany wewnątrz metody zostaną odzwierciedlone w programie wywołującym.

Klasa C# jest wyrażeniem klasy CTS

Każda klasa w powyższym przykładzie jest także inkarnacją typu CTS w C#. Klasy języka C# mogą implementować jeden lub więcej interfejsów, dziedziczyć po co najwyżej jednej klasie i wykonywać inne działania zdefiniowane dla klasy CTS. Pierwsza zaprezentowana powyżej klasa, `Compute`, implementuje interfejs `IMath`, na co wskazuje dwukropek pomiędzy `Compute` a `IMath`. Zgodnie z tym klasa musi zawierać implementacje obu metod interfejsu. Ciało metody `Factorial` deklaruje parę zmiennych liczbowych (nazywanych w żargonie CTS **polami**), inicjalizuje drugą z nich jako `1`, a następnie wykonuje prostą pętlę `for` w celu obliczenia silni jej parametru (i nie przejmuje się sprawdzaniem przepełnienia, co jest bez wątpienia złą praktyką programistyczną). Druga metoda `Compute`, `SquareRoot`, jest jeszcze prostsza. Bazuje ona na bibliotece klas .NET Framework, wywołując funkcję `Sqrt` dostarczaną przez klasę `Math` z przestrzeni nazw `System`.

Wykonanie programu w C# rozpoczyna się od metody Main

Ostatni typ w powyższym przykładzie, klasa `DisplayValues`, zawiera pojedynczą metodę — `Main`. Podobnie jak C i C++, program w C# rozpoczyna wykonywanie od tej metody w dowolnym typie, w którym ona się znajduje. `Main` musi być zadeklarowana jako metoda statyczna i — choć nie jest to tutaj pokazane — może przyjmować argumenty przekazane w momencie uruchomienia programu. W powyższym przykładzie `Main` zwraca `void`, co jest stosowanym w C# sposobem powiedzenia, że metoda nie zwraca wartości. Typ `void` nie może być jednak wykorzystywany w parametrach w taki sposób, jak w C czy C++. Zamiast tego jego jedynym celem jest wskazanie, że metoda nie zwraca wartości.

W powyższym przykładzie `Main` tworzy obiekt klasy `Compute`, używając operatora `new` z C#. Kiedy program zostanie uruchomiony, `new` zostanie przetłumaczone na instrukcję języka MSIL `newobj`, opisaną w rozdziale 2. Następnie `Main` deklaruje zmienną `int` i ustawia jej wartość na `5`. Wartość ta przekazywana jest jako parametr do wywołań metod

`Factorial` i `SquareRoot`, dostarczonych przez interfejs `Compute`. Metoda `Factorial` oczekuje `int`, czyli dokładnie tego, co przekazano jej w tym wywołaniu, jednak `SquareRoot` oczekuje typu `double`. Typ `int` zostanie automatycznie przekonwertowany na `double`, ponieważ konwersja ta może zostać dokonana bez straty informacji. C# nazywa to **konwersją niejawną** (ang. *implicit conversion*), odróżniając ją w ten sposób od konwersji typów, które są jawnie oznaczone w kodzie.

Metoda WriteLine klasy Console wypisuje sformatowane dane wyjściowe do konsoli

Wyniki są wypisywane za pomocą metody `WriteLine` klasy `Console`, innej standardowej części przestrzeni nazw `System` w .NET Framework. Metoda ta wykorzystuje opakowane w nawiasy klamrowe liczby, odpowiadające zmiennym, które mają być zwrócone. Warto zwrócić uwagę na fakt, iż w drugim wywołaniu `WriteLine` po liczbie w nawiasach występuje `:f4`. Ta dyrektywa formatująca oznacza, że wartość powinna być zapisana jako liczba stałoprzecinkowa z czterema miejscami po prawej stronie przecinka. Zgodnie z tym dane wyjściowe powyższego prostego programu będą następujące:

```
5 silnia: 120
Pierwiastek kwadratowy z 5: 2.2361
```

Powyższy przykład jest nierealistycznie prosty, jednak jego celem jest pokazanie ogólnej struktury i stylu C#. W języku tym istnieje oczywiście o wiele więcej możliwości, co zaprezentowano w dalszej części rozdziału.

Typy w C#

Typy w C# są zbudowane na bazie typów CTS

Każdy typ w C# jest zbudowany na bazie analogicznego typu CTS, dostarczonego przez CLR. Tabela 3.1 prezentuje większość typów CTS wraz z ich odpowiednikami w C#. Jak wspomniano wcześniej w książce, wszystkie typy danych są zdefiniowane w przestrzeni nazw `System`. Odpowiedniki w C# pokazane w tabeli są w rzeczywistości skróconymi synonimami alternatywnych definicji. W omówionym powyżej przykładzie linia:

```
int i;
```

mogłaby zostać zastąpiona przez:

```
System.Int32 i;
```

Obie możliwości działają i obie zwrócą dokładnie te same wyniki.

Tabela 3.1. Niektóre typy CTS i ich odpowiedniki w C#

CTS	C#
Byte	byte
Char	char
Int16	short
Int32	int
Int64	long
UInt16	ushort
UInt32	uint
UInt64	ulong
Single	float
Double	double
Decimal	decimal
Boolean	bool
Class	class
Interface	interface
Delegate	delegate

Warto zauważyć, że w C# wielkość liter ma znaczenie. Zadeklarowanie zmiennej jako `Double` zamiast `double` skończy się błędem kompilatora. Dla osób przyzwyczajonych do języków wywodzących się z C wydaje się to normalne. Pozostałe osoby będą jednak musiały poświęcić nieco czasu na przyzwyczajenie się do tego.

Klasy

Tak jak klasa CTS, klasa C# może bezpośrednio dziedziczyć jedynie po jednej klasie

Klasy C# pokazują zachowanie klas CTS, używając składni wywodzącej się z języka C. Na przykład klasy CTS mogą implementować jeden lub więcej interfejsów, jednak dziedziczą bezpośrednio po co najwyżej jednej klasie. Klasa C# `Calculator`, która implementuje interfejsy `IAlgebra` oraz `ITrig` i dziedziczy po klasie `MathBasics` byłaby zadeklarowana w następujący sposób:

```
class Calculator : MathBasics, IAlgebra, ITrig { ... }
```

Warto zauważyć, że klasa bazowa, o ile taka występuje, musi pojawić się na tej liście jako pierwsza, a dopiero po niej następują nazwy interfejsów. Klasy C# mogą być także oznaczone jako zapieczętowane

lub abstrakcyjne, jak omówiono to w rozdziale 2., oraz mogą mieć przypisaną widoczność publiczną lub wewnętrzną (ang. *internal*), która jest wartością domyślną. Przekładają się one odpowiednio na zdefiniowane w CTS widoczności `public` oraz `assembly`. Po skompilowaniu wszystkie te informacje są przechowywane w metadanych klasy.

Klasa C# może zawierać pola, metody oraz właściwości

Klasa C# może zawierać pola, metody oraz właściwości — wszystkie one są zdefiniowane dla dowolnej klasy CTS. Każde z nich posiada dostępność, oznaczoną w C# przez odpowiedni modyfikator dostępu, taki jak `public` czy `private`. Zarówno pola, jak i metody były zilustrowane na powyższym prostym programie, jednak właściwości są na tyle ważne, że zasługują na poświęcenie im osobnego przykładu.

Właściwość zmusza wszystkie próby dostępu do wartości do używania metod get oraz set

Do każdego pola oznaczonego jako `public` można uzyskać bezpośredni dostęp z kodu innej klasy. Przypuśćmy jednak, że klasa, w której pole to jest zdefiniowane, potrzebuje kontrolować sposób, w jaki się to odbywa. Być może powinno się na przykład sprawdzać każde przypisanie do tego pola w odniesieniu do zdefiniowanych wcześniej limitów lub też każda próba odczytania tego pola powinna być w jakiś sposób weryfikowana. Jednym ze sposobów wykonania tego zadania byłoby oznaczenie pola jako `private`, a następnie utworzenie metod, za pomocą których pole będzie mogło być modyfikowane i odczytywane. Ponieważ jednak ten wzorzec jest tak częsty, C# zapewnia właściwości, będące łatwiejszą możliwością osiągnięcia tego samego. Poniżej znajduje się prosty przykład:

```
class PriorityValue
{
   private int pValue;
   public int Priority
   {
      get
      {
         return pValue;
      }
      set
      {
         if (value > 0 && value < 11)
            pValue = value;
      }
   }
}
class PropertyExample
```

```
{
   static void Main()
   {
      PriorityValue p = new PriorityValue();
      p.Priority = 8;
      System.Console.WriteLine("Priorytet: {0}",
         p.Priority);
   }
}
```

Klasa `PriorityValue` deklaruje prywatne pole `pValue` oraz właściwość o nazwie `Priority`. Łatwo stwierdzić, że `Priority` jest właściwością, ponieważ zawiera dwie metody dostępowe (ang. *accessors*): `get` oraz `set`. Dostęp do właściwości odbywa się poprzez kod zawarty w tych metodach dostępowych. Tutaj na przykład próba odczytania właściwości `Priority` wykonuje kod `get`, który zwraca zawartość `pValue`. Próba zmodyfikowania `Priority` wywołuje kod `set`, który uaktualnia tę właściwość tylko wtedy, gdy nowa wartość mieści się w przedziale od 1 do 10. Słowo kluczowe `value` oznacza cokolwiek, co kod wywołujący próbuje przypisać do tej właściwości.

Dostęp do właściwości odbywa się podobnie jak do pól

Dlaczego właściwości są lepsze od pisania własnych metod do otrzymania (`get`) i ustawienia (`set`) wartości pola? Ponieważ zamiast wymagania jawnych wywołań metod `set` i `get`, kod wykorzystujący właściwości widzi te właściwości tak, jakby były one polami — składnia jest taka sama. Pozwala to na uproszczenie kodu wywołującego i jednocześnie na kontrolowanie sposobu, w jaki właściwość jest odczytywana i modyfikowana. W rzeczywistości istnieją silne przesłanki za tym, by nigdy nie używać publicznych pól. Właściwości są zawsze lepszym wyborem.

Klasy dostarczają konstruktory, nadpisują metody swojego rodzica i redefiniują operatory

Klasa może implementować jeden lub więcej konstruktorów, które są metodami wywoływanymi wtedy, gdy tworzony jest obiekt klasy. Każda klasa może także dostarczać co najwyżej jeden destruktor, który w C# jest w rzeczywistości nazwą finalizatora — koncepcji przedstawionej w rozdziale 2. Jeśli klasa dziedziczy po innej klasie, potencjalnie może nadpisywać jeden lub więcej składników typów rodzica, takich jak metoda. By to uczynić, składnik musi być zadeklarowany u rodzica ze słowem kluczowym `virtual`, natomiast klasa dziecka musi oznaczyć nowy składnik słowem kluczowym `override`. Klasa może także definiować przeciążone operatory. Przeciążony operator to taki, który został zredefiniowany w celu otrzymania specjalnego znaczenia w momencie

użycia go z obiektami danej klasy. Na przykład klasa reprezentująca grupy robocze w organizacji może redefiniować operator + w taki sposób, by oznaczał on łączenie dwóch grup roboczych w jedną.

Interfejsy

Interfejs C# może dziedziczyć bezpośrednio po jednym lub wielu innych interfejsach

Interfejsy są relatywnie prostymi podmiotami, natomiast podstawowa składnia w C# definiująca interfejs została zaprezentowana we wcześniejszym przykładzie. Nie pokazano tam jednak, w jaki sposób C# wyraża dziedziczenie po wielu interfejsach, czyli sytuację, w której jeden interfejs dziedziczy po więcej niż jednym rodzicu. Gdyby na przykład interfejs `ITrig` dziedziczył po trzech interfejsach (`ISine`, `ICosine` oraz `ITangent`), mógłby być zadeklarowany w następujący sposób:

```
Interface ITrig: ISine, ICosine, ITangent { ... }
```

`ITrig` będzie zawierał wszystkie metody, właściwości i inne składniki typów zdefiniowane w jego trzech rodzicach, a także wszystko to, co sam zdefiniuje.

Struktury

Struktury w C# są nieco uproszczonymi klasami C#

CTS nie definiuje w jawny sposób typu danych o nazwie „struktura". Zamiast tego struktury w C# są oparte na klasach, zatem tak jak klasy mogą implementować interfejsy, zawierać metody, pola, właściwości i inne. W przeciwieństwie do klas, struktury są jednak typami bezpośrednimi (dziedziczą po `System.ValueType`), a nie referencyjnymi, co oznacza, że są alokowane na stosie. Warto przypomnieć, że typy bezpośrednie nie mogą uczestniczyć w dziedziczeniu, zatem — w przeciwieństwie do klas — struktury nie mogą dziedziczyć po innym typie. Nie można także zdefiniować typu, który dziedziczy po strukturze.

Poniżej znajduje się prosty przykład struktury C#:

```
struct employee
{
   string name;
   int age;
}
```

W powyższym przykładzie struktura zawiera jedynie pola, podobnie jak tradycyjne struktury w stylu języka C. Struktura może jednak być bardziej złożona. Na przykład przedstawiona wcześniej klasa `Compute`

mogłaby być przekonwertowana na strukturę, ze wszystkimi metodami i tak dalej, jedynie za pomocą zmiany słowa kluczowego `class` w jej definicji na słowo `struct`. Gdyby doszło do takiej zmiany, wykonanie programu wyglądałoby nieco inaczej, ale jego wynik byłby taki sam.

Tablice

Tak jak tablice CTS, tablice C# są typami referencyjnymi

Jak w innych językach programowania, tablice C# są uporządkowanymi grupami elementów tego samego typu. Jednak w przeciwieństwie do wielu innych języków, tablice C# są obiektami. W rzeczywistości, jak opisano to w rozdziale 2., są one typami referencyjnymi, co oznacza, że są alokowane na stercie. Poniżej znajduje się przykład definiujący jednowymiarową tablicę liczb całkowitych:

```
int[] ages;
```

Ponieważ `ages` jest obiektem, nie istnieje żaden jego obiekt, dopóki się go jawnie nie utworzy. Można to zrobić w poniższy sposób:

```
ages = new int[10];
```

co alokuje na stercie miejsce dla dziesięciu liczb całkowitych. Jak pokazuje powyższy przykład, tablica C# nie ma określonego rozmiaru, dopóki nie utworzy się obiektu tego typu tablicy. Możliwe jest także zarówno zadeklarowanie, jak i utworzenie obiektu tablicy w pojedynczym poleceniu, takim jak:

```
int[] ages = new int[10];
```

Możliwe jest deklarowanie tablic dowolnego typu, jednak w jaki dokładnie sposób tablica zostanie zaalokowana, zależy od tego, czy będzie to tablica typów bezpośrednich, czy referencyjnych. Powyższy przykład alokuje miejsce dla dziesięciu liczb całkowitych na stercie, podczas gdy:

```
string[] names = new string[10];
```

alokuje miejsce dla dziesięciu referencji do łańcuchów znaków na stercie. Tablica typów bezpośrednich, takich jak liczby całkowite, w rzeczywistości zawiera ich wartości, podczas gdy tablica typów referencyjnych, takich jak łańcuchy znaków w ostatnim przykładzie, zawiera jedynie referencje do wartości.

Tablice C# mogą być wielowymiarowe

Tablice mogą także mieć wiele wymiarów. Na przykład instrukcja:

```
int[,] points = new int[10,20];
```

tworzy dwuwymiarową tablicę liczb całkowitych. Pierwszy wymiar ma dziesięć elementów, natomiast drugi ma ich dwadzieścia. Bez względu na liczbę wymiarów w tablicy, dolną granicą każdego z nich jest zawsze zero.

Dostęp do standardowych metod i właściwości możliwy jest ze wszystkich tablic C#

Typ tablic w C# jest zbudowany na bazie obsługi tablic zapewnionej przez CLR. Jak wspomniano w poprzednim rozdziale, wszystkie oparte na CLR tablice, włącznie z tablicami C#, dziedziczą po `System.Array`. Ten typ bazowy dostarcza różne metody i właściwości, do których dostęp można uzyskać z dowolnego obiektu typu „tablica". Na przykład metoda `GetLength` może być wykorzystana do ustalenia długości elementów w danym wymiarze tablicy, podczas gdy metoda `CopyTo` może zostać użyta do skopiowania wszystkich elementów jednowymiarowej tablicy do innej jednowymiarowej tablicy.

Delegaty i zdarzenia

Przekazanie referencji do metody jako parametr jest często przydatne

Przekazanie referencji do metody jest stosunkowo często spotykanym działaniem. Przypuśćmy na przykład, że musimy przekazać jakiejś części kodu informację o tym, która metoda powinna być wywołana, kiedy nastąpi określone zdarzenie. Potrzebny jest jakiś sposób przekazania tożsamości tej funkcji wywołania zwrotnego w czasie uruchomienia programu. W C++ odbywa się to dzięki przekazaniu adresu metody, czyli wskaźnika do kodu, który należy wywołać. Jednak w bezpiecznym świecie .NET Framework przekazywanie surowych adresów nie jest dozwolone. Problem ten jednak nie znika. Bezpieczny sposób przekazywania referencji do metody jest nadal przydatny.

Delegat C# zapewnia bezpieczny sposób przekazywania referencji do metody

Jak opisano to pokrótce w rozdziale 2., CTS definiuje dla tych celów typ referencyjny o nazwie *delegat*. Delegat jest obiektem, który zawiera referencję do metody o pewnej konkretnej sygnaturze. Kiedy zostanie utworzony i zainicjalizowany, może być przekazany jako parametr do jakiejś metody i wywołany. Poniżej znajduje się prosty przykład utworzenia i wykorzystania delegata w C#:

```
delegate void SDelegate(string s);
class DelegateExample
{
   public static void Main()
   {
      SDelegate del = new SDelegate(WriteString);
      CallDelegate(del);
   }
   public static void CallDelegate(SDelegate Write)
   {
      System.Console.WriteLine("W CallDelegate");
      Write("Witaj w delegacie");
   }
   public static void WriteString(string s)
   {
      System.Console.WriteLine("W WriteString: {0}", s);
   }
}
```

Przykład rozpoczyna się od zdefiniowania `SDelegate` jako typu „delegat". Definicja ta określa, że obiekty `SDelegate` mogą zawierać referencje tylko do metod, które przyjmują pojedynczy parametr, będący łańcuchem znaków. W metodzie `Main` z przykładu zmienna `del` typu `SDelegate` zostaje zadeklarowana, a następnie zainicjalizowana w taki sposób, by zawierała referencję do metody `WriteString`. Metoda ta jest później definiowana w klasie i — zgodnie z wymaganiami — posiada pojedynczy parametr typu `string`. Metoda `Main` wywołuje następnie metodę `CallDelegate`, przekazując jej `del` jako parametr. Metoda `CallDelegate` jest zdefiniowana w taki sposób, by przyjmowała `SDelegate` jako swój parametr. Innymi słowy, to, co przekazywane jest do metody, to obiekt-delegat zawierający adres jakiejś metody. Ponieważ jest to `SDelegate`, metoda ta musi posiadać pojedynczy parametr typu `string`. Wewnątrz `SDelegate` metoda identyfikowana przez przekazany parametr jest określana jako `Write`, a po wyświetleniu prostego komunikatu `CallDelegate` wywołuje metodę `Write`. Ponieważ jednak `Write` jest w rzeczywistości delegatem, tak naprawdę wywołana jest metoda, do której delegat zawiera referencję, a więc `WriteString`. Wynikiem tego prostego przykładu jest:

```
W CallDelegate
W WriteString: Witaj w delegacie
```

Warto zwrócić uwagę na fakt, że jako pierwsza wykonywana jest metoda `CallDelegate`, a dopiero po niej następuje `WriteString`.

Delegat może być łączony z innymi delegatami

Delegaty mogą być znacznie bardziej skomplikowane, niż wskazuje na to powyższy przykład. Mogą być na przykład łączone tak, by wywołanie pojedynczego delegata wiązało się z wywołaniem dwóch lub większej liczby delegatów, które zawiera pierwszy z nich. Jednak nawet proste delegaty mogą być przydatne. Dzięki zapewnieniu bezpiecznego sposobu przekazania referencji do metody ułatwiają wykonanie tej czynności w sposób o wiele mnie ryzykowny niż w przypadku wcześniejszych języków.

.NET Framework oraz C# zapewniają obsługę zdarzeń opartą na delegatach

Jedno z bardziej popularnych zastosowań delegatów obejmuje obsługę zdarzeń. Na przykład w GUI kliknięcia myszką użytkownika, naciśnięcia klawiszy czy inne formy danych wejściowych mogą być przyjmowane jako zdarzenia; zdarzenia są także użyteczne w innych kontekstach. Ponieważ zdarzenia są tak popularne, C# oraz .NET Framework zapewniają specjalną pomoc przy wykorzystywaniu delegatów w obsłudze zdarzeń w spójny sposób. Delegat, którego używa zdarzenie, nazywany jest **programem obsługi zdarzeń** (ang. *event handler*), choć w rzeczywistości jest to normalny delegat. Platforma .NET Framework definiuje jednak dwie konwencje dla takich programów obsługi zdarzeń:

- Program obsługi zdarzeń nie zwraca wartości, co oznacza, że typem zwracanej wartości jest `void`.
- Program obsługi zdarzeń zawsze przyjmuje dwa argumenty. Pierwszy argument, identyfikujący źródło zdarzenia, jest zgodnie z konwencją nazywany `sender` i jest typu `System.Object` (w C# jest po prostu typem `object`, który jest aliasem `System.Object`). Dzięki temu odbiorca zdarzenia może łatwo odpowiedzieć do dowolnego obiektu, który zdarzenie spowodował, na przykład wywołując metodę w tym obiekcie. Drugi argument, zawierający dane, które źródło przekazuje, kiedy wywołuje program obsługi zdarzeń, jest tradycyjnie zwany `e` i jest typu `System.EventArgs` bądź też typu dziedziczącego po `System.EventArgs`.

Delegaty wykorzystywane w zdarzeniach przestrzegają pewnych konwencji

Poniżej znajduje się przykładowa deklaracja programu obsługi zdarzeń:

```
public delegate void MyEventHandler(object sender, MyEventArgs e);
```

W powyższym przykładzie typ `MyEventArgs` musi pochodzić od `System.EventArgs` i musi rozszerzać ten typ bazowy w taki sposób, by mógł on służyć do przekazywania danych zdarzenia. Dla zdarzeń, które nie generują żadnych specyficznych informacji, typem służącym

do danych przekazywanych do programu obsługi zdarzeń może być po prostu `System.EventArgs` (nawet jeśli żadne dane nie są przekazywane, konwencja dotycząca zdarzeń wymaga, by ten parametr nadal pojawiał się w wywołaniu). Ponieważ zdarzenia często nie mają żadnych specyficznych dla nich danych, przestrzeń nazw `System` obejmuje także wbudowany typ — `EventHandler`. Typ ten jest po prostu delegatem z dwoma argumentami: obiektem wraz z `System.EventArgs`.

C# dostarcza słowo kluczowe event, które służy do deklaracji zdarzeń

Kiedy odpowiedni program obsługi zdarzeń (to znaczy delegat zgodny z opisanymi powyżej konwencjami) zostanie zadeklarowany, możliwe jest definiowanie zdarzenia z użyciem tego delegata. Poniżej znajduje się przykład takiej sytuacji:

```
public event MyEventHandler MyEvent;
```

Jak pokazuje powyższy przykład, deklaracja musi składać się ze słowa kluczowego `event`, natomiast typ musi być typem delegata.

Znając już podstawy, najłatwiej będzie zrozumieć sposób działania zdarzeń dzięki przykładowi. Zaprezentowany poniżej listing zawiera trzy klasy: `EventSource`, definiującą zdarzenie, `EventSink`, otrzymującą i odpowiadającą na zdarzenie, oraz `EventMain`, tworzącą obiekty dwóch pierwszych klas, a następnie generującą zdarzenie. Poniżej znajduje się odpowiedni kod:

```
public class EventSource
{
   public event System.EventHandler EventX;
   public void RaiseEventX()
   {
      if (EventX != null)
         EventX(this, System.EventArgs.Empty);
   }
}
public class EventSink
{
   public EventSink(EventSource es)
   {
      es.EventX += new
         System.EventHandler(ReceiveEvent);
   }
   public void ReceiveEvent(object sender,
      System.EventArgs e)
```

```
    {
        System.Console.WriteLine("EventX wywołane");
    }
}
public class EventMain
{
    public static void Main()
    {
        EventSource source = new EventSource();
        EventSink sink = new EventSink(source);
        source.RaiseEventX();
    }
}
```

Zdarzenia są inicjalizowane jako null

Zdarzenie wykorzystane w powyższym przykładzie, `EventX`, jest deklarowane na początku klasy `EventSource`. Ponieważ zdarzenie to nie posiada żadnych powiązanych danych, deklaracja wykorzystuje standardową klasę .NET Framework `System.EventHandler` w miejsce deklarowania własnego programu obsługi zdarzeń. Po deklaracji następuje metoda `RaiseEventX`. Zdarzenie, które nie ma zarejestrowanych żadnych programów obsługi zdarzeń, będzie miało wartość `null`, zatem po upewnieniu się, że `EventX` nie jest `null` — to znaczy, że w rzeczywistości jest co wywoływać — metoda ta wywołuje zdarzenie. (`System.EventArgs.Empty` wskazuje, że żadne dane nie są przekazywane wraz ze zdarzeniem). Ponieważ zdarzenie jest w rzeczywistości delegatem, naprawdę wywoływana jest dowolna metoda, na którą wskazuje ten delegat. I choć powyższy przykład tego nie pokazuje, delegat może wskazywać na wiele metod, zatem przywołanie zdarzenia sprawi, że wykonane zostaną wszystkie metody zarejestrowane w delegacie.

EventSink może zarejestrować się do zdarzenia za pomocą operatora += C#

Druga klasa, `EventSink`, ilustruje jedno z podejść do rejestrowania się i przetwarzania zdarzenia. Konstruktor klasy, który jak wszystkie konstruktory ma taką samą nazwę jak sama klasa i działa zawsze, gdy tworzony jest obiekt klasy, oczekuje, że zostanie mu przekazany obiekt `EventSource`. Następnie rejestruje program obsługi zdarzeń dla `EventX` za pomocą operatora `+=`. W tym prostym przykładzie konstruktor `EventSink` rejestruje metodę `ReceiveEvent`. Metoda `ReceiveEvent` posiada standardowe argumenty wykorzystywane dla zdarzeń i kiedy jest wywołana, wypisuje prosty komunikat do konsoli. Choć powyższy przykład tego nie pokazuje, programy obsługi zdarzeń mogą również być wyrejestrowane za pomocą operatora `-=`.

Ostatnia klasa, `EventMain`, zawiera metodę `Main` z przykładu. Metoda ta najpierw tworzy obiekt `EventSource`, a następnie obiekt `EventSink`, przekazując mu właśnie utworzony obiekt `EventSource`. To zmusza konstruktor `EventSink` do działania i zarejestrowania metody `ReceiveEvent` z `EventX` w `EventSource`. Wynikiem jest wywołanie `ReceiveEvent` i program wypisuje:

```
EventX wywołane
```

Bezpośrednia obsługa zdarzeń sprawia, że łatwiej jest wykorzystywać ten paradygmat

W interesie prostoty powyższy przykład nie przestrzega wszystkich konwencji związanych ze zdarzeniami. Nadal jednak przykład ten ilustruje podstawy tego, w jaki sposób w C# i niektórych konwencjach .NET Framework ulepszono delegaty, dzięki czemu możliwa jest bezpośrednia obsługa zdarzeń.

Typy generyczne

Wyobraźmy sobie, że chcielibyśmy utworzyć klasę, która może działać z różnymi typami danych. Być może aplikacja powinna działać z informacjami w parach, przetwarzając dwie dane tego samego typu. Jednym podejściem do wykonania tego byłoby zdefiniowanie różnych klas dla każdego rodzaju pary: jednej klasy dla pary liczb całkowitych, kolejnej dla pary łańcuchów znaków i tak dalej. Bardziej ogólne podejście opierałoby się na utworzeniu klasy `Pair`, która przechowuje dwie wartości typu `System.Object`. Ponieważ każdy typ .NET dziedziczy po `System.Object`, obiekt tej klasy mógłby przechowywać liczby całkowite, łańcuchy znaków czy cokolwiek innego. Jednak `System.Object` może być wszystkim, zatem nic nie zapobiegłoby sytuacji, w której obiekt klasy `Pair` przechowywałby jedną liczbę całkowitą i jeden łańcuch znaków zamiast pary wartości tego samego typu. Z tego i innych powodów bezpośrednia praca z typami `System.Object` nie jest szczególnie atrakcyjnym rozwiązaniem.

To, czego naprawdę nam potrzeba, to sposób utworzenia obiektów typu `Pair`, który pozwoliłby na określenie w momencie tworzenia, jakie dokładnie informacje będą zawarte w `Pair`, a następnie wymuszałby tę specyfikację. By odpowiedzieć na to wyzwanie, C# w wersji 2.0 z Visual Studio 2005 posiada obsługę **typów generycznych** (ang. *generic types*), popularnie zwanych *generics*. Kiedy definiowany jest typ generyczny, jeden lub więcej typów, które wykorzystuje, zostaje nieokreślonych. Prawdziwe typy, które powinny być użyte, są określane dopiero wtedy, gdy tworzony jest obiekt typu generycznego. Typy te mogą być różne dla różnych obiektów tego samego typu generycznego.

Na przykład poniżej znajduje się prosta ilustracja definiowania i używania klasy `Pair`:

```
class Pair<T>
{
   T element1, element2;

   public void SetPair(T first, T second)
   {
      element1 = first;
      element2 = second;
   }
   public T GetFirst()
   {
      return element1;
   }

   public T GetSecond()
   {
      return element2;
   }
}

class GenericsExample
{
   static void Main()
   {
      Pair<int> i = new Pair<int>();
      i.SetPair(42,48);
      System.Console.WriteLine("Para liczb całkowitych: {0} {1}",
         i.GetFirst(), i.GetSecond());

      Pair<string> s = new Pair<string>();
      s.SetPair("Carpe", "Diem");
      System.Console.WriteLine(
         "Para łańcuchów znaków: {0} {1}",
         s.GetFirst(), s.GetSecond());
   }
}
```

Definicja klasy `Pair` wykorzystuje `T`, który w pierwszym wystąpieniu został opakowany w nawiasy ostre, co reprezentuje typ informacji, jakie obiekt tego typu będzie zawierał. Pola i metody klasy działają z `T` tak samo, jakby był to dowolny inny typ, używając go w parametrach oraz zwracanych wartościach. Czym jednak tak naprawdę jest `T` — liczbą całkowitą, łańcuchem znaków czy jeszcze czymś innym — ustala się dopiero w momencie, gdy deklarowany jest obiekt `Pair`.

Co nowego w C# 2.0?

Typy generyczne to prawdopodobnie najważniejszy dodatek w wersji 2.0, jednak warto także wspomnieć o kilku innych nowych aspektach tego języka. Wśród nowości znajdują się poniższe cechy:

- **Typy częściowe (ang. *partial types*)** — dzięki wykorzystaniu nowego terminu `partial` definicja klasy, interfejsu czy struktury może teraz rozciągać się na dwa lub więcej plików źródłowych. Często spotykanym przykładem jest sytuacja, w której narzędzie takie jak Visual Studio 2005 generuje kod, do którego programista dodaje ciąg dalszy. Posiadając typy częściowe, programista nie musi bezpośrednio modyfikować kodu utworzonego przez narzędzie. Zamiast tego narzędzie oraz programista mogą utworzyć typy częściowe, które zostaną ze sobą połączone w końcową definicję. Ważnym przykładem zastosowania klas częściowych jest ASP.NET, opisany w rozdziale 5.
- **Typy dopuszczające wartość null (ang. *nullable types*)** — czasami przydatna jest możliwość ustawienia wartości na stan niezdefiniowany, często określany jako `null`. Najczęściej spotyka się tę sytuację w pracy z relacyjnymi bazami danych, gdzie `null` bywa poprawną wartością. W celu zaimplementowania tego pomysłu C# w wersji 2.0 pozwala na zadeklarowanie obiektu dowolnego typu bezpośredniego ze znakiem zapytania następującym po nazwie, jak w poniższym przykładzie:

  ```
  int? x;
  ```

  Zmienna zadeklarowana w ten sposób może przyjmować dowolną ze zwykłych wartości. Jednak w przeciwieństwie do zwykłej zmiennej typu `int`, może także zostać ustawiona na `null` w sposób jawny.
- **Metody anonimowe (ang. *anonymous methods*)** — jednym ze sposobów przekazania kodu jako parametru jest jawne zadeklarowanie tego kodu wewnątrz delegata. W C# 2.0 możliwe jest także przekazanie kodu bezpośrednio jako parametru — nie istnieje wymaganie, że kod musi być opakowany w osobno zadeklarowanym delegacie. W rezultacie przekazany kod zachowuje się jak metoda, jednak ponieważ nie ma nazwy, określany jest mianem metody anonimowej.

Jak pokazuje przykład metody `Main`, tworzenie obiektu typu generycznego wymaga dokładnego określenia, jaki typ powinien być użyty dla `T`. Tutaj pierwsza para `Pair` będzie zawierała dwie liczby całkowite, zatem przy jej tworzeniu dostarczany jest typ `int`. Ten obiekt `Pair` jest następnie ustawiany tak, by zawierał dwie liczby całkowite, a jego zawartość

jest wypisywana. Jednak drugi obiekt `Pair` będzie zawierał dwa łańcuchy znaków, a zatem przy jego tworzeniu dostarczany jest typ `string`. Tym razem obiekt `Pair` jest ustawiony tak, by zawierał dwa łańcuchy znaków, które są również wypisywane. Wynikiem wykonania powyższego przykładowego kodu jest:

```
Para liczb całkowitych: 42 48
Para łańcuchów znaków: Carpe Diem
```

Typy generyczne mogą być używane z klasami, strukturami, interfejsami, delegatami (i tym samym — zdarzeniami) oraz metodami, choć najczęściej pojawiają się w przypadku klas. Nie są odpowiednie dla każdej aplikacji, jednak przy niektórych rodzajach problemów typy generyczne mogą pomóc w stworzeniu właściwego rozwiązania.

Struktury sterujące w C#

Struktury sterujące w C# są typowe dla współczesnego języka wysokiego poziomu

C# dostarcza tradycyjny zbiór struktur sterujących dla współczesnego języka. Wśród najczęściej używanych struktur sterujących znajduje się instrukcja `if`, która wygląda następująco:

```
if (x > y)
   p = true;
else
   p = false;
```

Warto zwrócić uwagę na fakt, iż warunek dla `if` musi być wartością typu `bool`. W przeciwieństwie do języków C oraz C++, warunek nie może być liczbą całkowitą.

C# posiada również instrukcję `switch`. Poniżej znajduje się przykład jej wykorzystania:

```
switch (x)
{
   case 1:
      y = 100;
      break;
   case 2:
      y = 200;
      break;
   default:
      y = 300;
      break;
}
```

W zależności od wartości `x`, `y` będzie ustawiony na `100`, `200` lub `300`. Instrukcja `break` sprawia, że sterowanie przechodzi do instrukcji następującej po kodzie `switch`. W przeciwieństwie do C i C++, takie (lub podobne) instrukcje są w C# obowiązkowe, nawet dla przypadku domyślnego. Pominięcie ich spowoduje błąd kompilatora.

C# zawiera pętle while, do/while, for oraz foreach

C# zawiera także różne rodzaje pętli. W pętli `while` warunek musi być obliczany dla `bool`, a nie liczby całkowitej — jest to kolejna cecha odróżniająca C# od języków C i C++. Istnieje także kombinacja `do/while`, która umieszcza test na końcu, a nie na początku, oraz pętla `for`, zilustrowana wcześniej przykładem. Wreszcie, C# zawiera także instrukcję `foreach`, która pozwala na iterację przez wszystkie elementy w danej wartości typu **zbiorowego** (ang. *collection type*). Istnieją różne sposoby kwalifikowania typów do typów zbiorowych, z czego najprostszym z nich jest implementowanie standardowego interfejsu `System.IEnumerable`. Popularnym przykładem typu zbiorowego jest tablica, stąd jednym z zastosowań pętli `foreach` jest badanie lub przetwarzanie każdego elementu tablicy.

C# zawiera także instrukcję `goto`, powodującą przejście do określonego i oznaczonego punktu w programie, oraz instrukcję `continue`, rozpoczynającą nową iterację poprzez natychmiastowy powrót do góry dowolnej pętli, w której jest umieszczona. Ogólnie rzecz biorąc, struktury sterujące w tym relatywnie nowym języku nie są niczym nowym i większość z nich będzie wyglądała znajomo dla każdego, kto zna inny język wysokiego poziomu.

Inne cechy C#

Podstawy języka programowania leżą w jego typach i strukturach sterujących. Jednak w C# istnieje o wiele więcej interesujących możliwości — zbyt wiele, by przedstawić je szczegółowo w tak krótkim opisie. Niniejszy podrozdział prezentuje krótki przegląd niektórych bardziej interesujących dodatkowych aspektów tego języka.

Praca z przestrzeniami nazw

Instrukcja using w C# ułatwia odniesienia do zawartości przestrzeni nazw

Ponieważ bazowe biblioteki klas są tak podstawowe, przestrzenie nazw są kluczową częścią programowania w .NET Framework. Jednym ze sposobów wywołania metody z bibliotek klas jest podanie jej pełnej nazwy kwalifikowanej. W zaprezentowanym wcześniej przykładzie metoda `WriteLine` została wywołana za pomocą następującego kodu:

```
System.Console.WriteLine(...);
```

By uniknąć konieczności wprowadzania powtarzających się fragmentów kodu, C# dostarcza dyrektywę `using`. Pozwala to na odnoszenie się do zawartości przestrzeni nazw za pomocą krótszych nazw. Na przykład często rozpoczyna się każdy program w C# następującą linią:

```
using System;
```

Gdyby w przywołanym wyżej przykładzie zastosowano powyższy zapis, metoda `WriteLine` mogłaby zostać wywołana za pomocą skróconego zapisu:

```
Console.WriteLine(...);
```

Program może także zawierać kilka dyrektyw `using`, o ile jest to konieczne; przykłady takich sytuacji zostaną pokazane w dalszej części książki.

Dzięki użyciu słowa kluczowego `namespace` możliwe jest także zdefiniowanie własnych przestrzeni nazw bezpośrednio w C#. Każda przestrzeń nazw zawiera jeden lub więcej typów bądź też nawet inne przestrzenie nazw. Do typów z takiej przestrzeni nazw można odnieść się albo za pomocą pełnych, kwalifikowanych nazw, albo poprzez odpowiednie dyrektywy `using`, w taki sam sposób, w jaki odbywa się to w przypadku zewnętrznie zdefiniowanych przestrzeni nazw.

Obsługa wyjątków

Wyjątki pozwalają na spójny sposób radzenia sobie z błędami we wszystkich językach opartych na CLR

Błędy są nieodłączną częścią życia programisty. W .NET Framework błędy pojawiające się w czasie uruchomienia dzięki wyjątkom obsługiwane są w spójny sposób. Tak jak w innych przypadkach, C# dostarcza składni do pracy z wyjątkami, jednak podstawowe mechanizmy są osadzone w samym CLR. To pozwala nie tylko zapewnić spójne podejście do obsługi błędów dla wszystkich programistów C#, ale oznacza także, że wszystkie języki oparte na CLR będą traktowały ten potencjalnie zawiły obszar w ten sam sposób. Błędy mogą nawet być przekazywane przez granice między językami, o ile języki te zbudowane są na bazie CLR.

Wyjątek może być zgłoszony, kiedy wystąpi błąd

Wyjątek jest obiektem, który reprezentuje jakieś niezwykłe zdarzenie, na przykład błąd. Platforma .NET Framework definiuje duży zbiór wyjątków; możliwe jest również tworzenie własnych. Wyjątek jest zgłaszany automatycznie w czasie uruchomienia, gdy pojawia się błąd. Co się będzie na przykład działo w poniższym fragmencie kodu:

```
x = y/z;
```

jeśli z jest równe zero? CLR zgłosi wówczas wyjątek `System.DivideByZeroException`. Jeśli nie używa się żadnej obsługi wyjątków, program zostanie zakończony.

Wyjątki mogą być obsługiwane za pomocą bloków try/catch

C# umożliwia jednak przechwytywanie wyjątków za pomocą bloków `try`/`catch`. Powyższy kod można zmienić tak, by wyglądał następująco:

```
try
{
   x = y/z;
}
catch
{
   System.Console.WriteLine("Wyjątek przechwycony");
}
```

Kod wewnątrz nawiasów klamrowych instrukcji `try` będzie teraz monitorowany na okoliczność wystąpienia wyjątków. Jeśli żaden wyjątek nie wystąpi, instrukcja `catch` zostanie pominięta, a program będzie kontynuowany. Jeśli zostanie zgłoszony wyjątek, wykonany zostanie kod z instrukcji `catch`, co w tym przypadku oznacza wypisanie ostrzeżenia, natomiast wykonanie będzie kontynuowane wraz z kolejną instrukcją, następującą po `catch`.

Różne wyjątki mogą być obsługiwane w różny sposób

Możliwe jest także posiadanie różnych instrukcji `catch` dla różnych wyjątków i dokładne sprawdzenie, który wyjątek wystąpił. Poniżej znajduje się kolejny przykład:

```
try
{
   x = y/z;
}
catch (System.DivideByZeroException)
{
   System.Console.WriteLine("z jest zerem");
}
catch (System.Exception e)
{
   System.Console.WriteLine("Wyjątek: {0}", e.Message);
}
```

W powyższym przypadku, gdy nie wystąpi żaden wyjątek, do `x` zostanie przypisana wartość `y` podzielonego przez `z`, a kod znajdujący się w obu instrukcjach `catch` zostanie pominięty. Jeśli jednak `z` będzie zerem, wykonana zostanie pierwsza instrukcja `catch`, która wydrukuje

odpowiedni komunikat. Wykonanie pominie wtedy kolejną instrukcję `catch` i przejdzie do kodu, który następuje po bloku `try/catch`. Jeśli wystąpi jakikolwiek inny wyjątek, wykonana zostanie druga instrukcja `catch`. Instrukcja ta deklaruje obiekt `e` typu `System.Exception`, a następnie odwołuje się do właściwości `Message` tego obiektu w celu uzyskania możliwego do wydrukowania łańcucha znaków, wskazującego, jaki wyjątek wystąpił.

Możliwe jest definiowanie własnych wyjątków

Skoro języki oparte na CLR, takie jak C#, w spójny sposób wykorzystują wyjątki do radzenia sobie z błędami, dlaczego nie zdefiniować własnych wyjątków do obsługi błędów? Można to uczynić dzięki zdefiniowaniu klasy dziedziczącej po `System.Exception`, a następnie wykorzystaniu instrukcji `throw` w celu zgłoszenia własnego wyjątku. Takie wyjątki mogą być przechwytywane w blokach `try/catch`, tak samo jak wyjątki zdefiniowane przez system.

Choć nie jest to tutaj pokazane, możliwe jest także zakończenie bloku `try/catch` instrukcją `finally`. Kod w tej instrukcji zostaje wykonany bez względu na wystąpienie wyjątku. Opcja ta jest przydatna, gdy potrzebny jest jakiś rodzaj końcowego uporządkowania bez względu na to, co się dzieje.

Używanie atrybutów

Program w C# może zawierać atrybuty

Po skompilowaniu każdy typ C# posiada powiązane z nim metadane, przechowywane w tym samym pliku. Większość metadanych opisuje sam typ. Jednak, jak pokazano to w poprzednim rozdziale, metadane mogą także zawierać atrybuty określane dla tego typu. Biorąc pod uwagę fakt, iż CLR zapewnia sposób przechowywania atrybutów, C# musi posiadać jakąś metodę definiowania atrybutów oraz ich wartości. Jak opisano to w dalszej części książki, atrybuty są szeroko wykorzystywane przez bibliotekę klas .NET Framework. Mogą być stosowane do klas, interfejsów, struktur, metod, pól, parametrów i innych. Możliwe jest nawet określenie atrybutów, które będą stosowane do całego pakietu.

Załóżmy na przykład, że zaprezentowana wcześniej metoda `Factorial` została zadeklarowana wraz z odnoszącym się do niej atrybutem `WebMethod`. Zakładając, że zastosuje się odpowiednie dyrektywy `using` w celu zidentyfikowania właściwej przestrzeni nazw dla tego atrybutu, deklaracja w C# mogłaby wyglądać następująco:

```
[WebMethod] public int Factorial(int f) {...}
```

Atrybut ten jest wykorzystywany przez ASP.NET — część biblioteki klas .NET Framework — do wskazania, że metoda ta powinna być udostępniona jako usługa sieciowa możliwa do wywołania przez SOAP (więcej na temat wykorzystywania tego atrybutu znajduje się w rozdziale 5.). Podobnie, załączenie atrybutu:

```
[assembly:AssemblyCompanyAttribute("QwickBank")]
```

w pliku C# ustawi wartość atrybutu używanego w całym pakiecie, przechowywanego w jego manifeście i zawierającego nazwę firmy, która utworzyła pakiet. Przykład ten pokazuje także sposób użycia parametrów w atrybutach, pozwalający użytkownikowi na określenie konkretnych wartości atrybutu.

Możliwe jest również zdefiniowanie własnych atrybutów

Programiści mogą także definiować swoje własne atrybuty. Być może przyda się na przykład zdefiniowanie atrybutu, który będzie mógł być wykorzystywany do identyfikacji daty, kiedy dany typ C# był modyfikowany. By to uczynić, można zdefiniować klasę, która dziedziczy po `System.Attribute`, a następnie zdefiniować informacje, które klasa ma zawierać, takie jak data. Nowy atrybut można następnie zastosować do typów w programie i tym samym automatycznie otrzymać odpowiednie informacje w metadanych tych typów. Po utworzeniu własnych atrybutów można je odczytywać za pomocą metody `GetCustomAttributes`, zdefiniowanej w klasie `Attribute`, części przestrzeni nazw `System.Reflection` w bibliotece klas .NET Framework. Jednak bez względu na to, czy atrybuty są standardowe, czy też własne, są one często wykorzystywaną cechą oprogramowania opartego na CLR.

Pisanie niebezpiecznego kodu

Programiści C# zazwyczaj polegają na czyszczeniu pamięci przez CLR

C# zazwyczaj polega na CLR w kwestii zarządzania pamięcią. Kiedy na przykład obiekt typu referencyjnego nie jest więcej wykorzystywany, czyszczenie pamięci z CLR zwolni pamięć zajmowaną przez ten obiekt. Jak opisano w rozdziale 2., proces czyszczenia pamięci ponownie ustawia aktualnie używane elementy na zarządzanej stercie, zagęszczając je, by zyskać więcej wolnego miejsca.

Wskaźniki i czyszczenie pamięci nie lubią się ze sobą

Co by się stało, gdyby wykorzystać w tym środowisku tradycyjne wskaźniki C/C++? Wskaźnik zawiera bezpośredni adres w pamięci, zatem wskaźnik do zarządzanej sterty musiałby się odnosić do konkretnej lokalizacji w pamięci sterty. Kiedy proces czyszczenia pamięci poprzestawia zawartość sterty, by zwolnić więcej miejsca, to, na co wskazywał wskaźnik, może się zmienić. Mieszanie wskaźników i czyszczenia pamięci na ślepo jest najprostszą receptą na katastrofę.

C# pozwala na tworzenie niebezpiecznego kodu, który wykorzystuje wskaźniki

Jednak takie mieszanie jest czasem konieczne. Przypuśćmy na przykład, że należy wywołać istniejący kod, który nie jest oparty na CLR, taki jak na przykład kod systemu operacyjnego, a wywołanie zawiera strukturę z osadzonymi wskaźnikami. Lub być może jakaś część aplikacji jest tak istotna dla jej działania, że nie można polegać na tym, by to proces czyszczenia pamięci zarządzał pamięcią. W takich sytuacjach C# umożliwia wykorzystywanie wskaźników w tak zwanym **kodzie niebezpiecznym** (ang. *unsafe code*).

Kod niebezpieczny może wykorzystywać wskaźniki, ze wszystkimi zaletami i wadami tego rozwiązania. By taką „niebezpieczną" działalność jak najlepiej zabezpieczyć, C# wymaga jednak, by cały kod ją wykonujący został oznaczony słowem kluczowym `unsafe`. Wewnątrz niebezpiecznej metody można użyć instrukcji `fixed` w celu zablokowania jednej lub większej liczby wartości typu referencyjnego w odpowiednim miejscu na zarządzanej stercie (czasami jest to nazywane **przypinaniem wartości** — ang. *pinning a value*). Poniżej zamieszczono prosty przykład takiego działania:

```
class Risky
{
   unsafe public void PrintChars()
   {
      char[] charList = new char[2];
      charList[0] = 'A';
      charList[1] = 'B';

      System.Console.WriteLine("{0} {1}", charList[0], charList[1]);
      fixed (char* f = charList)
      {
         charList[0] = *(f+1);
      }
      System.Console.WriteLine("{0} {1}", charList[0], charList[1]);
   }
}

class DisplayValues
{
   static void Main()
   {
      Risky r = new Risky();
      r.PrintChars();
   }
}
```

Metoda `PrintChars` w klasie `Risky` jest oznaczona słowem kluczowym `unsafe`. Metoda ta deklaruje niewielką tablicę ze znakami o nazwie `charList`, a następnie ustawia dwa elementy tej tablicy jako odpowiednio `A` i `B`. Pierwsze wywołanie `WriteLine` zwraca:

```
A B
```

co jest dokładnie tym, czego należało oczekiwać. Instrukcja `fixed` deklaruje następnie wskaźnik znaku `f` i inicjalizuje go w taki sposób, by zawierał on adres tablicy `charList`. Wewnątrz ciała instrukcji `fixed` do pierwszego elementu tablicy przypisuje się wartość z adresu `f+1` (gwiazdka przed wyrażeniem oznacza „zwróć to, co znajduje się pod tym adresem"). Kiedy `WriteLine` jest ponownie wywołany, zwraca:

```
B B
```

Wartość znajdująca się o jedno miejsce dalej od początku tablicy, czyli znak `B`, został przypisany do pierwszej pozycji w tablicy.

Kod niebezpieczny ma swoje ograniczenia

Powyższy przykład nie wykonuje oczywiście niczego przydatnego. Jego celem jest jedynie pokazanie, że C# pozwala na deklarowanie wskaźników, wykonywanie arytmetyki wskaźników i inne, pod warunkiem że te instrukcje znajdują się wewnątrz obszaru wyraźnie oznaczonego jako `unsafe`. Twórcy języka naprawdę chcą, by autor niebezpiecznego kodu wykonywał te czynności w sposób świadomy, dlatego kompilowanie niebezpiecznego kodu wymaga jawnego ustawienia opcji `unsafe` dla kompilatora C#. Kod niebezpieczny nie może też być weryfikowany pod kątem bezpieczeństwa typologicznego, co oznacza, że nie mogą być wykorzystywane wbudowane w CLR możliwości dotyczące bezpieczeństwa dostępu do kodu, opisane w rozdziale 2. Kod niebezpieczny może być uruchamiany jedynie w środowisku o pełnym zaufaniu, co czyni go generalnie niedostępnym dla oprogramowania pobieranego z Internetu. Nadal jednak występują sytuacje, w których kod niebezpieczny jest właściwym rozwiązaniem trudnego problemu.

Dyrektywy preprocesora

W przeciwieństwie do języków C i C++, C# nie posiada preprocesora. Zamiast tego kompilator ma wbudowaną obsługę najbardziej przydatnych cech preprocesora. Na przykład dyrektywy preprocesora w C# obejmują `#define` — termin znany programistom C++. Dyrektywa ta nie może jednak być wykorzystywana do definiowania dowolnego zastępującego łańcucha znaków dla słowa — nie można definiować makr.

Perspektywa: czy C# jest tylko kopią Javy?

C# z pewnością jest bardzo podobny do języka Java. Biorąc pod uwagę dodatkowe podobieństwo pomiędzy CLR a wirtualną maszyną Javy, trudno uwierzyć, że Microsoft nie zainspirował się sukcesem Javy. Łącząc składnię w stylu języka C z obiektami w bardziej przystępny sposób niż w języku C++, twórcy Javy znaleźli złoty środek dla dużej grupy programistów. Przed nadejściem .NET widziałem wiele projektów, w których zdecydowano się na wybór środowiska Javy, ponieważ ani Visual Basic 6, ani C++ nie uznano za język nadający się dla tworzenia oprogramowania komercyjnego na wielką skalę.

Nadejście C# i wersji VB opartej na .NET zdecydowanie polepszyło pozycję technologii Microsoftu przeciwko obozowi Javy. Jakość języka programowania nie jest już problemem. Jednak ponownie pojawia się pytania: czy C# nie jest jak Java?

W wielu aspektach odpowiedź brzmi: tak. Podstawowa semantyka CLR jest bardzo podobna do Javy. Głębokie zorientowanie obiektowe, bezpośrednia obsługa interfejsów, pozwalanie na wielokrotne dziedziczenie interfejsów, ale pojedyncze dziedziczenie implementacji — wszystkie te cechy są podobne do Javy. C# posiada jednak także możliwości, których nie było w Javie. Na przykład wbudowana obsługa właściwości, oparta na obsłudze właściwości z CLR, odzwierciedla wpływ VB na twórców C#. Atrybuty, kolejna cecha oparta na CLR, zapewniają poziom elastyczności niedostępny w oryginalnej Javie, podobnie jak możliwość pisania niebezpiecznego kodu. C# jest wyrażeniem semantyki CLR w składni wywodzącej się z języka C. Ponieważ semantyka ta jest tak podobna do Javy, C# również jest bardzo podobny do Javy. Jednak nie jest to ten sam język.

Czy C# jest lepszym językiem niż Java? Nie da się odpowiedzieć na to pytanie w sposób obiektywny, a nawet gdyby się dało — nie miałoby to znaczenia. Wybranie platformy programistycznej wyłącznie w oparciu o język programowania jest jak kupowanie samochodu, dlatego że podoba się nam jego radio. Można tak zrobić, jednak o wiele lepszym wyjściem będzie rozważenie całego pakietu.

Gdyby Sun pozwolił Microsoftowi na niewielkie zmodyfikowanie Javy, C# mógłby dzisiaj nie istnieć. Ze zrozumiałych względów Sun oparł się zakusom Microsoftu, by dopasować Javę do świata Windows. Efektem tego są dwa dość podobne języki, z których każdy ma inne docelowe środowisko programistyczne. Konkurencja jest dobra i oba języki mają przed sobą długą przyszłość.

Zamiast tego dyrektywa `#define` jest wykorzystywana jedynie do definiowania symbolu. Symbol może następnie zostać użyty wraz z dyrektywą `#if`, by zapewnić kompilację warunkową. Na przykład w następującym fragmencie kodu:

```
#define DEBUG
#if DEBUG
    // kod skompilowany, jeśli DEBUG jest zdefiniowany
#else
    // kod skompilowany, jeśli DEBUG nie jest zdefiniowany
#endif
```

`DEBUG` jest zdefiniowany, a zatem kompilator C# przetworzyłby kod zawarty pomiędzy dyrektywami `#if` oraz `#else`. Gdyby `DEBUG` był niezdefiniowany, co można osiągnąć za pomocą dyrektywy preprocesora `#undef`, kompilator przetworzyłby kod znajdujący się pomiędzy dyrektywami `#else` oraz `#endif`.

C# jest atrakcyjnym językiem. Łączy czysty i spójny projekt z nowoczesnym zbiorem możliwości. Wprowadzenie nowej technologii programistycznej jest trudne — świat jest zasypany szczątkami języków programowania, które nie odniosły sukcesu — jednak w przypadku C# Microsoft wyraźnie odniósł sukces. Jako jeden z dwóch najczęściej używanych języków .NET, C# znajduje się obecnie w głównym nurcie tworzenia oprogramowania.

Visual Basic

Przed premierą .NET Visual Basic 6 był zdecydowanie najpopularniejszym językiem programowania w świecie Windows. Pierwsza wersja VB oparta na .NET, zwana Visual Basic .NET (VB .NET), przyniosła ogromne zmiany do tego szeroko stosowanego narzędzia. Wersja obsługiwana przez Visual Studio 2005, oficjalnie zwana Visual Basic 2005, zbudowana jest na tej podstawie. Nie stanowi tak wielkiej zmiany, jaką było przejście pomiędzy VB 6 z VB .NET, jednak również zawiera kilka interesujących nowości.

Z wyjątkiem składni, C# i VB są bardzo podobne

Tak samo jak C#, Visual Basic zbudowany jest na bazie wspólnego środowiska uruchomieniowego (CLR), zatem znaczna część tego języka jest w istocie definiowana przez CLR. W rzeczywistości z wyjątkiem składni, C# i VB są w dużej mierze tym samym językiem. Ponieważ oba zawdzięczają tak wiele CLR i bibliotece klas .NET Framework, ich funkcjonalność jest bardzo podobna.

Obecnie tylko Microsoft dostarcza kompilatory VB

VB może być kompilowany za pomocą Visual Studio 2005 lub *vbc.exe* — kompilatora wiersza poleceń dostarczanego wraz z .NET Framework. Jednak w przeciwieństwie do C#, Microsoft nie zgłosił VB do żadnego organu standaryzacyjnego. Dlatego też — dopóki świat open source albo inna firma nie stworzą alternatywnej wersji — narzędzia Microsoftu w najbliższej przyszłości będą jedynym możliwym wyborem do pracy z tym językiem.

Przykład Visual Basic

Najszybszym sposobem poznania VB jest przyjrzenie się prostemu przykładowi. Zaprezentowany poniżej implementuje tę samą funkcjonalność co C# w przykładzie pokazanym we wcześniejszej części niniejszego rozdziału. Jak łatwo zauważyć, różnice pomiędzy tymi przykładami są raczej kosmetyczne.

```
' Przykład VB
Module DisplayValues

Interface IMath
   Function Factorial(ByVal F As Integer) _
      As Integer
   Function SquareRoot(ByVal S As Double) _
      As Double
End Interface

Class Compute
   Implements IMath

   Function Factorial(ByVal F As Integer) _
      As Integer Implements IMath.Factorial
      Dim I As Integer
      Dim Result As Integer = 1

      For I = 2 To F
         Result = Result * I
      Next
      Return Result
   End Function

   Function SquareRoot(ByVal S As Double) _
      As Double Implements IMath.SquareRoot
      Return System.Math.Sqrt(S)
   End Function
End Class
```

```
Sub Main()
   Dim C As Compute = New Compute()
   Dim V As Integer
   V = 5
   System.Console.WriteLine( _
      "{0} silnia: {1}", _
      V, C.Factorial(V))
   System.Console.WriteLine( _
      "Pierwiastek kwadratowy z {0}: {1:f4}", _
      V, C.SquareRoot(V))
End Sub

End Module
```

Module jest pojemnikiem dla innych typów VB

Przykład rozpoczyna się od komentarza oznaczonego pojedynczym apostrofem. Po komentarzu następuje obiekt typu `Module`, który zawiera cały kod niniejszego przykładu. `Module` jest typem referencyjnym, jednak tworzenie obiektów tego typu jest niedozwolone. Zamiast tego jego głównym celem jest bycie pojemnikiem na grupy klas, interfejsów i innych typów VB. W tym przypadku moduł zawiera interfejs, klasę oraz procedurę `Sub Main`. Moduł może również zawierać definicje metod, deklaracje zmiennych i inne elementy, które mogą być używane w całym module.

Tak jak w C#, w VB parametry domyślnie przekazuje się przez wartości

Interfejs modułu nazwany jest `IMath` i — tak jak w przykładzie w C# — definiuje on metody (w żargonie VB — funkcje) `Factorial` oraz `SquareRoot`. Każda z nich przyjmuje pojedynczy parametr i każda jest zdefiniowana w taki sposób, by można było je przekazywać przez wartości, co oznacza, że kopia parametru jest wykonywana w ramach funkcji. Końcowy znak podkreślenia jest znakiem kontynuacji linii, oznaczającym, że następna linia powinna być traktowana w taki sposób, jakby nie było pomiędzy nimi złamania wiersza. Przekazywanie przez wartość jest domyślne, zatem przykład ten działałby tak samo bez wskazań `ByVal`[1].

Klasa VB jest wyrażeniem klasy CTS

Klasa `Compute`, która jest wyrażeniem klasy CTS w VB, implementuje interfejs `IMath`. W przeciwieństwie do C#, każda z funkcji w tej klasie musi jawnie identyfikować metodę interfejsu, który implementuje. Oprócz tego funkcje są takie same jak we wcześniejszym przykładzie w C#; jedyną różnicą jest użycie składni w stylu VB. W szczególności

[1] W VB 6 domyślne było przekazywanie przez referencję, co pokazuje, jak bardzo zmienił się VB, by sprostać semantyce CLR leżącej u jego podstaw.

warto zwrócić uwagę na to, że wywołanie `System.Math.Sqrt` ma identyczną formę do tego z przykładu w C#. C#, VB i inne języki zbudowane na bazie CLR otrzymują dostęp do usług z biblioteki klas .NET Framework w bardzo podobny sposób.

Perspektywa: C# czy VB?

Przed pojawieniem się .NET wybór języka dla programistów zorientowanych na produkty firmy Microsoft był prosty. Jeśli było się prawdziwym programistą, nieskończenie dumnym ze swojej wiedzy technicznej, wybierało się C++ wraz ze wszystkimi jego cierniami. Alternatywnie, jeśli było się bardziej zainteresowanym wykonaniem konkretnego zadania niż zaawansowaną technologią i jeśli zadanie to nie było zbyt skomplikowane lub też na zbyt niskim poziomie, wybierało się VB 6. Oczywiście, za taki wybór było się poddanym surowej krytyce ze strony programistów C++ ze względu na brak językowego savoir vivre'u, jednak kod napisany w VB 6 miał za to o wiele mniej niezrozumiałych błędów.

Ten podział skończył się wraz z nadejściem .NET. C# i VB są prawie tym samym językiem. Z wyjątkiem relatywnie rzadko spotykanych aspektów, takich jak pisanie niebezpiecznego kodu, mają takie same możliwości. Microsoft może to zmienić w przyszłości, czyniąc zbiory możliwości obu języków znacznie odmiennymi. Jednak zanim to się stanie (o ile w ogóle się stanie), najważniejszym kryterium wyboru pozostaje osobista preferencja, czyli inaczej mówiąc — składnia.

Programiści bardzo przywiązują się do wyglądu używanego języka. Osoby zorientowane na język C kochają nawiasy klamrowe, podczas gdy programiści VB czują się jak u siebie w domu z instrukcjami `Dim`. Od czasu premiery .NET w 2002 roku oba języki stały się popularne i oba mają swoich zagorzałych wielbicieli. Także Microsoft z reguły traktuje je równo i nawet dokumentacja do .NET Framework jest stosunkowo wyważona, prezentując przykłady w obu językach. Żaden z nich nie zniknie w najbliższym czasie, zatem oba języki są bezpiecznym wyborem zarówno dla programistów, jak i dla organizacji, które im płacą.

Bez względu na to sądzę jednak, że każdy programista znający C# może (i powinien) poznać VB przynajmniej w stopniu umożliwiającym czytanie go — i odwrotnie. Podstawowa semantyka jest niemal identyczna, a w końcu w tym zazwyczaj leży największa trudność w nauczeniu się języka. W rzeczywistości, by zilustrować równość obu języków, przykłady w kolejnych rozdziałach niniejszej książki przedstawione są na zmianę w jednym bądź drugim. W świecie .NET nie powinno się myśleć o sobie jak o programiście VB czy C#. Bez względu na wybór języka programowania, zawsze będzie się programistą .NET Framework.

Wykonanie rozpoczyna się w procedurze Sub Main

Powyższy prosty przykład kończy się procedurą `Sub Main`, która jest analogiczna do metody `Main` w C#. Tutaj aplikacja rozpoczyna swoje wykonywanie. W powyższym przykładzie `Sub Main` tworzy obiekt klasy `Compute` za pomocą operatora VB `New` (który w efekcie końcowym zostanie przetłumaczony na instrukcję MSIL `newobj`). Następnie deklaruje zmienną `Integer` i ustawia jej wartość na `5`.

Tak samo jak w przykładzie w C#, wyniki tego prostego programu są wypisywane za pomocą metody `WriteLine` klasy `Console`. Ponieważ metoda ta jest częścią biblioteki klas .NET Framework, a nie któregoś konkretnego języka, wygląda to dokładnie tak samo jak w przykładzie w C#. Nie powinno zatem być zaskoczeniem, że dane wyjściowe tego prostego programu wyglądają tak samo jak poprzednio, czyli następująco:

```
5 silnia: 120
Pierwiastek kwadratowy z 5: 2.2361
```

Podobieństwo VB do VB 6 zarówno pomaga, jak i przeszkadza w nauczeniu się tego nowego języka

Dla kogoś, kto zna VB 6, Visual Basic w wersji .NET będzie wyglądał znajomo. Dla kogoś, kto zna C#, ta wersja VB będzie zachowywała się w dużej mierze znajomo, ponieważ zbudowana jest na tej samej podstawie. Jednak VB zaimplementowany w Visual Studio 2005 nie jest tym samym co VB 6 czy C#. Podobieństwa mogą być bardzo przydatne przy uczeniu się tego nowego języka, jednak mogą być także zdradliwe.

Typy w Visual Basic

Podobnie jak w C#, typy zdefiniowane przez VB są zbudowane na bazie typów CTS, dostarczanych przez CLR. Tabela 3.2 pokazuje większość z nich wraz z ich ekwiwalentami w Visual Basic.

Wielkość liter nie jest dla VB istotna

W przeciwieństwie do C#, dla Visual Basic wielkość liter nie ma znaczenia. Istnieją jednak dość silnie zakorzenione konwencje, które pokazano na wcześniejszym przykładzie. Dla osób, które trafiły do .NET z VB 6, brak znaczenia wielkości liter będzie się wydawał całkowicie naturalny. To jeden z przykładów, które uzasadniają istnienie zarówno VB, jak i C#, ponieważ im więcej nowe środowisko ma wspólnego ze starym, tym łatwiej ludzie się do niego przyzwyczają.

Tabela 3.2. Niektóre typy CTS wraz z ich odpowiednikami w VB

CTS	VB
Byte	Byte
Char	Char
Int16	Short
Int32	Integer
Int64	Long
UInt16	UShort
UInt32	UInteger
UInt64	ULong
Single	Single
Double	Double
Decimal	Decimal
Boolean	Boolean
Class	Class
Interface	Interface
Delegate	Delegate

Klasy

Tak jak klasa CTS, klasa VB może bezpośrednio dziedziczyć jedynie po jednej klasie

Klasy VB udostępniają zachowanie klasy CTS za pomocą składni w stylu VB. Tym samym klasy VB mogą implementować jeden lub więcej interfejsów, jednak dziedziczyć mogą po co najwyżej jednej klasie. W VB klasa `Calculator` implementująca interfejsy `IAlgebra` oraz `ITrig` i dziedzicząca po klasie `MathBasics` wygląda następująco:

```
Class Calculator
   Inherits MathBasics
   Implements IAlgebra
   Implements ITrig
...
End Class
```

Warto zwrócić uwagę na fakt, iż — podobnie jak w C# — klasa bazowa musi poprzedzać interfejsy. Należy także zauważyć, że dowolna klasa, po której dziedziczy powyższa klasa, może być napisana w VB, C# lub nawet innym języku opartym na CLR. Dopóki język ten przestrzega reguł podanych w specyfikacji CLS z CLR, dziedziczenie pomiędzy

językami jest proste. Ponadto jeśli klasa dziedziczy po innej klasie, potencjalnie może nadpisywać jeden lub więcej składników typu swojego rodzica, takich jak metoda. Jest to dozwolone tylko wtedy, gdy składnik ten jest zadeklarowany ze słowem kluczowym `Overridable`, analogicznie do słowa kluczowego `virtual` w C#.

VB obsługuje przeciążanie operatorów

Klasy VB mogą być oznaczone jako `NonInheritable` lub `MustInherit`, co oznacza to samo co odpowiednio `sealed` i `abstract` w rozumieniu CTS i C#. Do klas VB można także przypisać rozmaite dostępności, takie jak `Public` i `Friend`, które w większości odwzorowują się na widoczności zdefiniowane przez CTS. Klasa VB może zawierać zmienne, metody, właściwości, zdarzenia i inne, zgodnie z definicjami z CTS. Każda z nich może mieć określony modyfikator dostępności, taki jak `Public`, `Private` oraz `Friend`. Klasa może również zawierać jeden lub więcej konstruktorów, które wywołuje się za każdym razem, gdy tworzony jest obiekt tej klasy. VB, tak jak C#, obsługuje przeciążanie operatorów, co jest nowością w wersji 2005.

Klasy VB mogą także posiadać właściwości. Poniżej znajduje się właściwość pokazana wcześniej w C# — tym razem zaprezentowana w VB:

```
Module PropertyExample
   Class PriorityValue
      Private m_Value As Integer
      Public Property Priority() As Integer
         Get
            Return m_Value
         End Get
         Set(ByVal Value As Integer)
            If (Value > 0 And Value < 11) Then
               m_Value = Value
            End If
         End Set
      End Property
   End Class

   Sub Main()
      Dim P As PriorityValue = New PriorityValue()
      P.Priority = 8
      System.Console.WriteLine("Priorytet: {0}", _
        P.Priority)
   End Sub
End Module
```

Perspektywa: czy dziedziczenie jest naprawdę przydatne?

Dziedziczenie jest istotną częścią technologii obiektowych. Przed .NET Visual Basic właściwie nie obsługiwał dziedziczenia, zatem (co jest zupełnie zrozumiałe) nie był uznawany za język zorientowany obiektowo. Dziedziczenie jest obecnie obsługiwane w VB, ponieważ język ten jest oparty na CLR, co oznacza, że jest on teraz prawdziwie zorientowany obiektowo.

Czy jest to jednak dobrą zmianą? Microsoft z pewnością mógł dodać dziedziczenie do VB wiele lat temu, jednak nie zdecydował się na to. Zawsze, gdy pytałem w Microsofcie o powody takiej decyzji, otrzymywałem dwie odpowiedzi. Po pierwsze, dziedziczenie może być trudne do zrozumienia i poprawnego stosowania. W hierarchii klas o wielu poziomach, w której niektóre metody są nadpisywane, a inne przeciążane, dokładne zrozumienie, co się dzieje, nie zawsze jest proste. Biorąc pod uwagę fakt, iż główną grupą docelową dla VB nie byli programiści z formalnym wykształceniem informatycznym, utrzymanie prostoty tego języka miało sens.

Drugą kwestią poruszaną w odpowiedzi na pytanie, dlaczego VB nie obsługuje dziedziczenia, był fakt, iż w wielu kontekstach dziedziczenie się nie sprawdza. Ta argumentacja najczęściej dotyczyła COM — technologii, w której nie ma bezpośredniej obsługi implementowania dziedziczenia. Dziedziczenie ściśle łączy klasę dziecka z rodzicem, co oznacza, że zmiany w rodzicu mogą mieć katastrofalne skutki dla dziecka. Problem „wrażliwej" klasy bazowej jest szczególnie istotny, kiedy klasy rodzica oraz dziecka są napisane i utrzymywane przez całkowicie odrębne organizacje lub gdy źródło rodzica nie jest dostępne dla twórcy dziecka. W zorientowanym na komponenty świecie COM taki argument jest jak najbardziej uzasadniony.

Dlaczego więc Microsoft zmienił zdanie, jeśli chodzi o dziedziczenie? Dziedziczenie nadal może być problematyczne, jeśli zmiany w klasie rodzica nie zostaną zakomunikowane wszystkim programistom, którzy są uzależnieni od danej klasy, co może być dość skomplikowane. Argumentów Microsoftu nie można uznać za niewłaściwe. Jednak triumf technologii obiektowych jest całkowity: obiekty są wszędzie! Stworzenie nowych języków w całkowicie nowym środowisku, czyli stworzenie .NET Framework i obecnej wersji Visual Studio bez pełnej obsługi dziedziczenia, przykleiłoby do ich autorów metkę zacofania. Zalety dziedziczenia, w szczególności te związane z dostarczeniem wielkiego zbioru nadających się do ponownego użycia klas, takiego jak biblioteki klas .NET Framework, są ogromne. Świat poszedł do przodu i dziedziczenie jest teraz niezbędne.

Tak jak w przykładzie w C#, właściwość bazuje na wartości prywatnej w ramach klasy, która ma zawierać informacje o niej. Również metody `Get` i `Set` właściwości wyglądają podobnie do tych z wcześniejszego przykładu, uwzględniając zmiany składni wymagane przez VB. Dostęp do właściwości wygląda tak samo jak dostęp do publicznego pola w klasie, z przewagą polegającą na tym, że zarówno odczytywanie, jak i wypisywanie jej wartości bazuje na kodzie zdefiniowanym przez programistę.

Interfejsy

Tak jak interfejs CTS, interfejs VB może dziedziczyć bezpośrednio po jednym lub większej liczbie interfejsów

Interfejsy zgodne z definicją CTS są stosunkowo prostą koncepcją. VB dostarcza składnię, za pomocą której wyrażane jest to, co definiuje CTS. Oprócz zachowania, które omówiono wcześniej, interfejsy CTS mogą dziedziczyć po jednym lub wielu innych interfejsach. Na przykład w VB zdefiniowanie interfejsu `ITrig`, który dziedziczy po trzech interfejsach: `ISine`, `ICosine` oraz `ITangent`, wyglądałoby następująco:

```
Interface ITrig
   Inherits ISine
   Inherits ICosine
   Inherits ITangent
...
End Interface
```

Struktury

Struktury VB mogą zawierać pola, metody i tak dalej

Struktury w VB są bardzo podobne do struktur w C#. Tak jak klasa, struktura może zawierać pola, składniki i właściwości, implementować interfejsy i tak dalej. Tak jak struktura w C#, struktura w VB jest typem bezpośrednim, co oznacza, że nie może dziedziczyć po innym typie ani też inny typ nie może dziedziczyć po niej. Prosta struktura dla pracownika mogłaby być zdefiniowana w VB w następujący sposób:

```
Structure Employee
   Public Name As String
   Public Age As Integer
End Structure
```

By uprościć powyższy przykład, struktura ta zawiera jedynie składniki danych. Jak jednak opisano to wcześniej, struktury w VB są w rzeczywistości prawie tak potężne jak klasy.

Tablice

W przeciwieństwie do VB 6, indeksy tablic w VB rozpoczynają się od zera

Tak jak tablice w C# i innych językach opartych na CLR, tablice w VB są typami referencyjnymi, które dziedziczą po standardowej klasie `System.Array`. Zgodnie z tym dowolna tablica w VB może wykorzystywać wszystkie metody i właściwości, które udostępnia ta klasa. Tablice w VB wyglądają podobnie do tablic z wcześniejszych wersji Visual Basic. Największą różnicą jest prawdopodobnie fakt, iż pierwszym elementem tablicy VB jest teraz element zerowy, podczas gdy w wersjach tego języka sprzed ery .NET pierwszy był element numer jeden. Liczba elementów w tablicy może zatem być o jeden większa od liczby podanej w jej deklaracji. Na przykład poniższa instrukcja deklaruje tablicę z jedenastoma liczbami całkowitymi:

```
Dim Ages(10) as Integer
```

W przeciwieństwie do C#, nie ma potrzeby jawnego tworzenia obiektu tablicy za pomocą `New`. Możliwe jest także zadeklarowanie tablicy bez określonego rozmiaru i późniejsze użycie instrukcji `ReDim` do określenia jej wielkości. Na przykład poniższy kod:

```
Dim Ages() As Integer
ReDim Ages(10)
```

zwraca tablicę jedenastu liczb całkowitych, tak samo jak we wcześniejszym przykładzie. Warto zauważyć, że indeks dla obu tablic rozciąga się od 0 do 10, a nie od 1 do 10.

VB pozwala także na tablice wielowymiarowe. Na przykład instrukcja:

```
Dim Points(10,20) As Integer
```

tworzy dwuwymiarową tablicę liczb całkowitych z odpowiednio 11 i 21 elementami. I znów oba wymiary rozpoczynają się od zera, co oznacza, że indeksy rozciągają się od 0 do 10 dla pierwszego wymiaru i od 0 do 20 dla drugiego.

Delegaty i zdarzenia

Pomysł przekazywania jawnej referencji do procedury bądź funkcji, a następnie wywoływania tej procedury lub funkcji nie był czymś, do czego typowy programista VB 6 byłby przyzwyczajony. Jednak CLR zapewnia obsługę delegatów, która na to pozwala. Dlaczego nie udostępnić tego zachowania w dzisiejszym VB? I co ważniejsze, dlaczego nie ułatwić używania zdarzeń?

VB pozwala na tworzenie i używanie delegatów

Twórcy VB wybrali obydwa wyjścia, pozwalając programistom na łatwe tworzenie wywołań zwrotnych i pozostałego kodu zorientowanego na zdarzenia. Poniżej znajduje się przykład — analogiczny do pokazanego wcześniej dla C# — tworzenia i używania delegata w VB:

```
Module DelegatesExample

   Delegate Sub SDelegate(ByVal S As String)
   Sub CallDelegate(ByVal Write As SDelegate)
      System.Console.WriteLine("W CallDelegate")
      Write("Witaj w delegacie")
   End Sub

   Sub WriteString(ByVal S As String)
      System.Console.WriteLine( _
         "W WriteString: {0}", S)
   End Sub

   Sub Main()
      Dim Del As New SDelegate( _
         AddressOf WriteString)
         CallDelegate(Del)
   End Sub

End Module
```

Choć napisany w VB, powyższy kod działa dokładnie tak samo jak przykład w C#, zaprezentowany we wcześniejszej części niniejszego rozdziału. Podobnie jak tamten przykład, i ten rozpoczyna się od zdefiniowania `SDelegate` jako typ `Delegate`. Tak jak wcześniej, obiekty `SDelegate` mogą zawierać referencje jedynie do metod, które przyjmują pojedynczy łańcuch znaków jako parametr. W metodzie `Sub Main` z przykładu zmienna `Del` typu `SDelegate` jest deklarowana, a następnie inicjalizowana tak, by zawierała referencję do procedury `WriteString` (procedura VB jest metodą, która — w przeciwieństwie do funkcji — nie zwraca żadnego wyniku). Osiągnięcie tego wymaga wykorzystania słowa kluczowego VB `AddressOf` przed nazwą procedury. `Sub Main` wywołuje następnie `CallDelegate`, przekazując `Del` jako parametr.

`CallDelegate` posiada parametr `SDelegate` zwany `Write`. Kiedy wywoływany jest `Write`, metoda w delegacie, który został przekazany do `CallDelegate`, jest w rzeczywistości wywoływana. W powyższym przykładzie metodą tą jest `WriteString`, zatem następnie wykonywany jest

kod znajdujący się wewnątrz procedury `WriteString`. Wynik tego prostego przykładu jest dokładnie taki sam jak dla wersji w C#, zaprezentowanej wcześniej:

```
W CallDelegate
W WriteString: Witaj w delegacie
```

Delegaty są kolejnym przykładem nowych cech, które VB zyskał dzięki powstaniu na bazie CLR. Choć opanowanie tego języka z pewnością wymaga wiele nauki ze strony programistów, nagrodą jest pokaźny zbiór nowych możliwości.

Zdarzenia VB bazują na delegatach

Jednym z pomysłów, które nie są dla VB nowe, jest bezpośrednia obsługa zdarzeń. Jednak w przeciwieństwie do wersji Visual Basic sprzed .NET, obecnie zdarzenia są zbudowane na bazie delegatów. Nadal jednak używanie zdarzeń w VB może być stosunkowo proste, łatwiejsze nawet od używania ich w C#. Poniżej znajduje się przykład pokazany wcześniej w C#, a obecnie przeniesiony na VB:

```
Module EventsExample
   Public Class EventSource
      Public Event EventX()
      Sub RaiseEventX()
         RaiseEvent EventX()
      End Sub
   End Class

   Public Class EventSink
      Private WithEvents Source As EventSource
      Public Sub New(ByVal Es As EventSource)
         Me.Source = Es
      End Sub
      Public Sub ReceiveEvent() _
         Handles Source.EventX
         System.Console.WriteLine("EventX wywołane")
      End Sub
   End Class

   Sub Main()
      Dim Source As EventSource = New EventSource()
      Dim Sink As EventSink = New EventSink(Source)
      Source.RaiseEventX()
   End Sub
End Module
```

Zdarzenia mogą być deklarowane za pomocą słowa kluczowego Event

Tak jak we wcześniejszym przykładzie, kod ten zawiera klasę `EventSource`, klasę `EventSink` oraz metodę `Main`, która tworzy i wykorzystuje obiekt każdej z klas. Jak jednak pokazuje powyższa ilustracja, możliwe jest wykorzystywanie zdarzeń w VB bez jawnej pracy z typami delegatów. Zamiast tego zdarzenie może być zadeklarowane z użyciem słowa kluczowego `Event`, tak jak dzieje się to w pierwszym wierszu klasy `EventSource`. Nie istnieje konieczność posiadania referencji ani do delegata zdefiniowanego w systemie, takiego jak `System.EventHandler`, ani też do własnego delegata (choć oczywiście można to zrobić). Zgłaszanie zdarzenia niekoniecznie wymaga też jawnego stosowania się do konwencji argumentów wykorzystywanej w C#. Zamiast tego, jak pokazano w metodzie `RaiseEventX` klasy `EventSource`, można wykorzystać słowo kluczowe `RaiseEvent`. Kompilator VB wypełnia pozostałe wymagania.

Obsługa zdarzeń w VB może być łatwiejsza niż w C#

Sposób dołączania obsługi zdarzeń do zdarzeń jest w VB także w pewien sposób prostszy niż w C#. W klasie `EventSink` z powyższego przykładu słowo kluczowe `WithEvents` oznacza, że pole `Source` może wywoływać zdarzenia. Definicja metody, która obsługuje zdarzenie, może wykorzystywać słowo kluczowe `Handles` w celu oznaczenia, które zdarzenia powinna otrzymać dana metoda. Dokładnie to wykonywane jest przez metodę `ReceiveEvent` klasy `EventSink`. I choć, jak w przykładzie w C#, kod ten dołącza źródło zdarzenia do odbiorcy w konstruktorze `EventSink` (metody `New`), to szczegóły różnią się od siebie. Tutaj do pola `Source` w klasie `EventSink` przypisywany jest dowolny obiekt klasy `EventSource`, który jest przekazywany w momencie utworzenia `EventSink`. Wreszcie, metoda `Main` robi to samo co wcześniej: tworzy obiekty obu klas, a następnie wywołuje metodę, która w efekcie końcowym wywoła zdarzenie. Tak jak poprzednio wynikiem programu będzie:

```
EventX wywołane
```

Istnieją także inne sposoby pracy ze zdarzeniami w VB. Możliwe jest na przykład jawne deklarowanie zdarzeń za pomocą delegatów, tak jak w C#, oraz dołączanie programów obsługi zdarzeń za pomocą słowa kluczowego `AddHandler`. Bez względu na te różnice, zdarzenia (i delegaty, na których bazują) są ważną częścią programowania aplikacji, wykorzystywaną przez Windows Forms, ASP.NET oraz inne fundamentalne części .NET Framework.

Perspektywa: czy VB stał się zbyt trudny?

Być może. Zmiany spotkały się z dużą falą krytyki i z pewnością niektórzy programiści VB 6 zostali w tyle. Kiedyś Microsoft kierował C++ i VB do odrębnych docelowych grup programistów, jednak .NET Framework w dużej mierze zatarł te różnice. Pod względem funkcjonalności C# i VB są prawie identyczne.

Platforma .NET Framework jest pod pewnymi względami zdecydowanie prostsza niż środowisko Windows DNA, które zastąpiła. Jednak .NET Framework jest też trudniejsza dla pewnych klas programistów, w szczególności tych z brakiem formalnego wykształcenia w dziedzinie informatyki. Jednym z powodów sukcesu Microsoftu na rynku programowania była dostępność VB. Osoby, które tworzą narzędzia do programowania, często zapominają, że prawie zawsze same są o wiele lepszymi programistami niż osoby, które będą z tych narzędzi korzystać. W rezultacie powstają narzędzia, których chcieliby używać oni sami — potężne programy, zbyt skomplikowane dla ich potencjalnych klientów.

Oryginalni twórcy VB nie popełnili tego błędu. Bez względu na wyrazy potępienia ze strony programistów C++, z którymi spotykał się ten język oraz jego użytkownicy, Microsoft w sposób jednoznaczny skupiał się na docelowej grupie programistów i poziomie ich umiejętności. Było to dobrą decyzją i w pewnym momencie VB był najczęściej wykorzystywanym językiem programowania na świecie.

Jednak wielu programistów pragnęło więcej. Wersje VB oparte na .NET dały im więcej, jednak wymagały także od **wszystkich** programistów VB zwiększenia poziomu ich wiedzy technicznej. Umiejętności potrzebne do zbudowania opartego na GUI klienta dwuwarstwowej aplikacji (czyli oryginalnego celu VB) są wręcz nieporównywalne z umiejętnościami wymaganymi do zbudowania dzisiejszych skalowalnych, wielowarstwowych i dostępnych z sieci rozwiązań. Biorąc pod uwagę ten fakt, być może nie ma już miejsca dla pierwotnych odbiorców, do których Microsoft kierował kiedyś VB, a którzy poziomem umiejętności nie odbiegali za bardzo od zaawansowanych użytkowników. VB ze swoim całkowitym zorientowaniem obiektowym i ogromnym zbiorem bardziej zaawansowanych możliwości jest dla nich zdecydowanie zbyt skomplikowany.

Jednak tworzenie nowoczesnych aplikacji za pomocą starego VB efektywnie stawało się coraz trudniejsze. Będąc pomiędzy młotem a kowadłem, Microsoft zdecydował się uczynić ten popularny język zarówno potężniejszym, jak i bardziej skomplikowanym. Niektórzy programiści są z tego powodu bardzo szczęśliwi, inni nie. Nawet gdy ma się rozległe zasoby Microsoftu, nie zawsze da się zadowolić wszystkich.

Typy generyczne

Tak jak w C#, wydanie 2005 Visual Basic dodaje obsługę typów generycznych. Poniżej znajduje się zaprezentowany wcześniej przykład z `Pair`, tym razem wyrażony w VB:

```
Module GenericsExample
   Class pair(Of t)
      Dim element1, element2 As t
      Sub SetPair(ByVal first As t, ByVal second As t)
         element1 = first
         element2 = second
      End Sub

      Function GetFirst() As t
         Return element1
      End Function

      Function GetSecond() As t
         Return element2
      End Function
   End Class

   Sub Main()
      Dim i As New pair(Of Integer)
      i.SetPair(42, 48)
      System.Console.WriteLine( _
         "Para liczb całkowitych: {0} {1}", _
         i.GetFirst(), i.GetSecond())

      Dim s As New pair(Of String)
      s.SetPair("Carpe", "Diem")
      System.Console.WriteLine( _
         "Para łańcuchów znaków: {0} {1}", _
         s.GetFirst(), s.GetSecond())
   End Sub
End Module
```

Składnia różni się od zaprezentowanej wcześniej wersji w C#, co najbardziej oczywiste — sposobem definiowania klasy generycznej:

```
Class pair(Of t)
```

zamiast, jak w C#:

```
class Pair<T>
```

Pomijając jednak takie powierzchowne różnice, typy generyczne funkcjonują w VB tak samo jak w C#.

Tak jak w C#, VB w wersji 2005 obsługuje także typy częściowe, w tym klasy częściowe i inne. Jednak w przeciwieństwie do C# w VB nie ma typów dopuszczających wartość `null` ani metod anonimowych. Tak jak ich oryginalne inkarnacje, wersje 2005 C# i VB są funkcjonalnie prawie identyczne z drobnymi różnicami, takimi jak ta.

Perspektywa: czy typy generyczne są coś warte?

Prawdopodobnie nie ma lepszej ilustracji do dalekiej drogi, jaką język VB przebył od swych skromnych początków aż do dodania obsługi typów generycznych. Typy generyczne są podobne do szablonów z C++, czyli cechy często cytowanej jako przykład nadmiernego i niepotrzebnego skomplikowania tego języka. Czy zatem typy generyczne przynależą do VB?

Jedną z odpowiedzi jest uświadomienie sobie, że typy generyczne są opcjonalne. Programiści tworzący nowe aplikacje mogą unikać wykorzystywania typów generycznych, jeśli koncepcja ta jest dla nich myląca. Problem z takim myśleniem polega na tym, że sam Microsoft zaczął wykorzystywać typy generyczne w nowych interfejsach programistycznych, które udostępnia. Biorąc to pod uwagę, programiści VB być może będą zmuszeni do zapoznania się z tą koncepcją, bez względu na to, czy się im to podoba, czy nie.

Inny punkt widzenia podkreśla, że typy generyczne nie są wcale takie trudne. Kiedy już przyzwyczai się do tego pomysłu, może on w rzeczywistości pomóc w uproszczeniu kodu i uczynieniu go mniej podatnym na błędy. Z pewnością jest to prawda dla pewnej części programistów, jednak dla wielu innych tak nie jest. W szczególności dla programistów bardziej skupiających się na rozwiązywaniu problemów biznesowych niż na technicznych szczegółach — czyli dla tradycyjnej społeczności VB — subtelności typów generycznych mogą być krokiem za daleko.

Bez względu na to, jak będzie naprawdę, dodanie obsługi typów generycznych do VB jasno pokazuje, że bez względu na nazwę tego języka, tradycyjna prostota związana z językiem Basic odeszła na zawsze.

Struktury sterujące w Visual Basic

Struktury sterujące VB będą dla większości programistów wyglądały znajomo

Choć CLR mówi wiele o tym, jak powinny wyglądać typy w językach opartych na .NET Framework, nie mówi prawie nic o tym, w jaki sposób powinny wyglądać struktury sterujące języka. Dlatego adaptacja VB do CLR wymagała zmian w typach VB, jednak struktury sterujące tego języka pozostały zupełnie standardowe. Na przykład instrukcja `If` wygląda następująco:

```
If (X > Y) Then
   P = True
Else
   P = False
End If
```

podczas gdy instrukcja `Select Case`, analogiczna do instrukcji `switch` z C#, która została pokazana wcześniej, wygląda tak:

```
Select Case X
   Case 1
      Y = 100
   Case 2
      Y = 200
   Case Else
      Y = 300
End Select
```

Tak jak w przykładzie w C#, różne wartości `x` spowodują ustawienie `y` na `100`, `200` lub `300`. Choć nie jest to tutaj pokazane, warunki `Case` mogą także określać zakres, a nie tylko pojedynczą wartość.

VB zawiera pętlę While, pętlę Do, pętlę For...Next oraz pętlę For Each

Instrukcje pętli dostępne w VB obejmują pętlę `While`, która kończy się, kiedy określony warunek `Boolean` nie jest już prawdziwy, pętlę `Do`, która pozwala na wykonywanie pętli, dopóki warunek jest prawdziwy lub też dopóki jakiś warunek nie stanie się prawdziwy, oraz pętlę `For...Next`, która została zaprezentowana we wcześniejszym przykładzie tego podrozdziału. Tak jak C#, VB również zawiera instrukcję `For Each`, która pozwala na iterację przez wszystkie elementy wartości typu zbiorowego.

VB posiada także instrukcję `GoTo`, pozwalającą na przejście do oznaczonego punktu w programie, oraz instrukcję `Continue`, rozpoczynającą następną iterację poprzez powrót do góry pętli, w której jest zawarta (nowość w wersji 2005 tego języka). Innowacje w .NET Framework nie

skupiają się na strukturach sterujących języka (w rzeczywistości trudno jest przypomnieć sobie ostatnią innowację w tej dziedzinie), zatem VB nie oferuje w tym zakresie zbyt wiele nowości.

Inne cechy Visual Basic

VB udostępnia większość możliwości CLR

CLR dostarcza wiele innych możliwości, jak zaprezentowano to w znajdującym się wcześniej opisie C#. Z drobnymi wyjątkami twórcy Visual Basic zdecydowali się udostępnić te możliwości programistom pracującym w najnowszej wersji VB. Niniejszy podrozdział pokazuje, w jaki sposób VB obsługuje niektóre bardziej zaawansowane cechy.

Praca z przestrzeniami nazw

Instrukcja Imports z VB pozwala na łatwiejszy dostęp do zawartości przestrzeni nazw

Tak jak w C#, przestrzenie nazw są ważną częścią pisania aplikacji w VB. Jak pokazano wcześniej w przykładzie w VB, dostęp do biblioteki klas .NET Framework wygląda w VB tak samo jak w C#. Ponieważ wszędzie wykorzystywany jest CTS, metody, parametry, zwracane wartości i inne są definiowane w ten sam sposób. Jednak sposób wskazywania na to, które przestrzenie nazw będzie wykorzystywał program, jest w VB nieco inny od tego niż w C#. Najczęściej używane przestrzenie nazw mogą być dla modułu identyfikowane za pomocą instrukcji `Imports`. Na przykład poprzedzenie modułu instrukcją:

```
Imports System
```

pozwoli na wywoływanie metody `System.Console.WriteLine` za pomocą samego:

```
Console.WriteLine(...)
```

Instrukcja `Imports` w VB jest analogiczna do dyrektywy `using` z C#. Obie pozwalają programistom na oszczędzenie konieczności pisania kodu. Tak jak C#, również VB pozwala na definiowanie i używanie własnych przestrzeni nazw.

Obsługa wyjątków

Jedną z większych zalet CLR jest zapewnienie wspólnego sposobu obsługi wyjątków we wszystkich językach .NET Framework. Wspólne podejście pozwala na znalezienie błędu na przykład w procedurze C#, a następnie poradzenie sobie z tym błędem w kodzie napisanym w VB. Dokładna składnia tych języków służąca do pracy z wyjątkami jest różna, jednak samo zachowanie, określone przez CLR, jest takie samo.

Tak samo jak C#, Visual Basic wykorzystuje `Try` i `Catch` w celu zapewnienia obsługi wyjątków. Poniżej znajduje się przykład radzenia sobie z wyjątkiem, który został zgłoszony po próbie dzielenia przez zero:

Tak jak w C#, bloki try/catch w VB są wykorzystywane do obsługi wyjątków

```
Try
   X = Y/Z
Catch
   System.Console.WriteLine("Wyjątek przechwycony")
End Try
```

Dowolny kod znajdujący się pomiędzy `Try` i `Catch` jest monitorowany pod kątem wystąpienia wyjątków. Jeśli wyjątek się nie pojawia, wykonanie pomija warunek `Catch` i kontynuuje wykonanie kodu, znajdującego się po `End Try`. Jeśli wyjątek pojawia się, kod w warunku `Catch` jest wykonywany, a wykonywanie jest kontynuowane w kodzie następującym po `End Try`.

Tak samo jak w C#, możliwe jest tworzenie różnych warunków `Catch` dla różnych wyjątków. Warunek `Catch` może także zawierać klauzulę `When` z warunkiem `Boolean`. W takim przypadku wyjątek zostanie przechwycony tylko wtedy, gdy warunek ten będzie spełniony. Tak jak C#, VB pozwala na definiowanie własnych wyjątków i następnie zgłaszanie ich za pomocą instrukcji `Throw`. VB posiada również instrukcję `Finally`. Podobnie do C#, kod w bloku `Finally` jest wykonywany bez względu na wystąpienie wyjątku.

VB oferuje prawie te same opcje obsługi wyjątków co C#

Używanie atrybutów

Kod napisany w VB jest kompilowany do MSIL, zatem musi posiadać metadane. Ponieważ ma metadane, ma także atrybuty. Projektanci języka dostarczyli składnię w stylu VB służącą do określania atrybutów, jednak wynik jest taki sam jak dla każdego języka CLR: dodatkowe informacje są umieszczane w metadanych jakiegoś pakietu. By powtórzyć raz jeszcze przykład zamieszczony wcześniej w niniejszym rozdziale, przypuśćmy, że metoda `Factorial` pokazana w przykładzie VB została zadeklarowana z dołączonym do niej atrybutem `WebMethod`. Atrybut ten instruuje .NET Framework, by udostępnił tę metodę usługom sieciowym możliwym do wywołania przez SOAP, tak jak opisano to w rozdziale 7. Zakładając, że odpowiednie instrukcje `Imports` znajdowały się na miejscu i pomogły zidentyfikować właściwą przestrzeń nazw dla tego atrybutu, deklaracja w VB mogłaby wyglądać następująco:

Program w VB może zawierać atrybuty

```
<WebMethod()> Public Function Factorial(ByVal F _
As Integer) As Integer Implements IMath.Factorial
```

Atrybut ten wykorzystywany jest przez ASP.NET do oznaczenia, że metoda ta powinna być udostępniona jako usługa sieciowa, możliwa do wywołania z SOAP. Podobnie, załączenie atrybutu:

```
<assembly:AssemblyCompanyAttribute("QwickBank")>
```

w pliku VB ustawi wartość atrybutu przechowywaną w manifeście pakietu, która identyfikuje QwickBank jako twórcę tego pakietu. Programiści VB mogą także tworzyć własne atrybuty poprzez definiowanie klas dziedziczących z `System.Attribute` i następnie automatyczne skopiowanie wszystkich informacji zdefiniowanych dla tych atrybutów do metadanych. Tak jak w C# i każdym innym języku opartym na CLR, własne atrybuty można odczytywać za pomocą metody `GetCustomAttributes`, zdefiniowanej przez klasę `Attribute` w przestrzeni nazw `System.Reflection`.

Atrybuty są tylko jednym z wielu przykładów niesamowitego podobieństwa pomiędzy VB a C#. Wybór zastosowanego języka jest w dużej mierze decyzją opartą na estetyce.

Przestrzeń nazw My

W wersji VB z 2005 roku istnieje interesujący dodatek, który nie jest częścią C# — przestrzeń nazw `My`. Celem jest zbliżenie VB do jego korzeni poprzez ułatwienie programistom wykonywania często spotykanych, ale potencjalnie skomplikowanych rzeczy. W tym celu przestrzeń nazw `My` zawiera pewną liczbę obiektów, które upraszczają życie programistom VB. Niektóre z tych obiektów to:

- **`My.Application`** — obiekt ten pozwala programistom na łatwiejszy dostęp do informacji o bieżącej aplikacji. Na przykład właściwość `My.Application.CommandLineArgs` pozwala programiście VB na dostęp do dowolnych argumentów dostarczonych przez wiersz poleceń, kiedy aplikacja była wywoływana, natomiast metoda `My.Application.ChangeCulture` pozwala na modyfikowanie kultury (na przykład z angielskiej na francuską), wykorzystywanej do formatowania dat i innych ustawień.
- **`My.User`** — obiekt ten pozwala na dostęp do właściwości bieżącego użytkownika aplikacji. Na przykład właściwość `My.User.Name` zwraca nazwę bieżącego użytkownika, podczas gdy metoda `IsInRole` może być wykorzystywana w celu ustalenia, czy użytkownikowi przypisano pewną rolę, na przykład administratora.

- **`My.Computer`** — obiekt ten zapewnia dostęp do różnych aspektów maszyny, na której działa bieżąca aplikacja. `My.Computer` zawiera zbiór właściwości, które zwracają inne obiekty dla różnych rodzajów dostępu. Niektóre przykłady obejmują `My.Computer.Audio` do odtwarzania plików *.wav*, `My.Computer.Clock` do dostępu do bieżącego czasu, `My.Computer.FileSystem` do pracy z plikami i katalogami, `My.Computer.Network` do wgrywania i pobierania danych oraz `My.Computer.Registry` do dostępu do rejestru lokalnej maszyny.
- **`My.Settings`** — obiekt ten pozwala na pracę z ustawieniami aplikacji, takimi jak łączenie się z bazą danych lub preferencje użytkownika.

Nie ma rozsądnego uzasadnienia, dlaczego klasy przestrzeni nazw `My` nie mogły być udostępnione programistom pracującym w C# czy innych językach opartych na CLR. Biorąc jednak pod uwagę historyczną orientację VB w kierunku mniej technicznych programistów, nie powinno być niespodzianką, że taki upraszczający zbiór klas pojawił się najpierw właśnie tu.

Perspektywa: po co udostępniać wszystkie te języki?

Microsoft twierdzi, że ponad 20 języków zostało przeniesionych na CLR. Wraz z językami dostarczanymi przez sam Microsoft, programiści .NET mają dużo opcji do wyboru. Jednak biorąc pod uwagę centralną rolę CLR w definiowaniu tych języków, często mają one ze sobą wiele wspólnego. Jaka jest tak naprawdę korzyść z posiadania wielu języków opartych na CLR?

Dla Microsoftu istnieją dwie kluczowe zalety tej sytuacji. Po pierwsze, populacja programistów dla Windows przed .NET była podzielona na dwa główne obozy: C++ oraz Visual Basic. Microsoft potrzebował wykonać krok do przodu z obiema grupami programistów, którzy są przywiązani do swojego języka. Choć semantyka CLR (i języków na nim opartych, takich jak C# i VB) różni się zarówno od C++, jak i od VB 6, podstawowy wygląd nowych języków będzie jednak znajomy. Gdyby Microsoft zdecydował się na dostarczenie na przykład tylko C#, można się założyć, że programiści przywiązani do VB 6 mieliby duże opory przed przejściem na .NET. Z kolei dostarczenie wyłącznie języka opartego na CLR wywodzącego się z VB 6 nie uszczęśliwiłoby programistów C++. Osoby piszące kod przywiązują się do najdziwniejszych rzeczy (takich jak na przykład nawiasy klamrowe), zatem dostarczenie zarówno C#, jak i wersji VB opartej na CLR było dobrym sposobem pomocy obecnemu środowisku programistów dla Windows w wykonaniu kroku naprzód.

Drugą zaletą wynikającą z zapewnienia wielu języków jest fakt, iż daje to .NET Framework coś, czego nie posiada konkurencja. Jednym z zarzutów czynionych światowi Javy jest to, że wymaga on od wszystkich programistów wykorzystywania tego samego języka. Wielojęzykowa natura .NET Framework oferuje większy wybór i tym samym daje Microsoftowi coś, dzięki czemu może odróżnić się od konkurentów.

W rzeczywistości jednak istnieją także prawdziwe zalety wynikające z posiadania wyłącznie jednego języka. Po co dodatkowe skomplikowanie, takie jak różne składnie służące do wyrażania tego samego zachowania, jeśli nie wynika z tego prawdziwa korzyść? Tradycyjne podejście Javy: „jeden język zawsze i wszędzie" charakteryzuje się cnotą prostoty. Nawet w świecie .NET lepiej jest, gdy organizacje unikają projektów wielojęzykowych, o ile tylko jest to możliwe. Prawdą jest, że kod napisany w różnych językach opartych na CLR może ze sobą współpracować bez większych problemów oraz że programiści znający C# nie powinni mieć trudności z rozumieniem VB (i odwrotnie). Nadal jednak posiadanie dwóch (lub więcej) odrębnych grup programistów używających różnych języków komplikuje zarówno sam projekt, jak i jego późniejsze utrzymanie. Warto tego uniknąć, jeśli tylko jest to możliwe.

Dotychczas różnorodny zbiór języków, które są oficjalnie dostępne na .NET Framework, nie miał zbyt dużego znaczenia. Ze względu na wsparcie ze strony Microsoftu, najbardziej widoczne w Visual Studio, C# i VB są bez wątpienia najczęstszymi wyborami, jeśli chodzi o tworzenie nowych aplikacji opartych na CLR. Inne języki mogą być interesujące dla uniwersytetów, jednak dla zawodowych programistów Visual Studio i obsługiwane przez to narzędzie języki przeważają.

C++

C++ był zbyt popularny, by twórcy .NET Framework mogli go zignorować

C# był całkowicie nowym językiem stworzonym specjalnie na potrzeby .NET Framework. Wersja VB oparta na .NET była mniej więcej tym samym, choć nazwa i styl składniowy tego języka zostały zapożyczone z VB 6. Jednak C++ istniał na długo przed .NET i był w użyciu od wielu lat. Biorąc to pod uwagę, Microsoft zdecydował, że choć dostarczenie jakiegoś sposobu tworzenia w C++ oprogramowania opartego na CLR jest konieczne, to tak samo niezbędne jest zapewnienie utrzymania zgodności z istniejącym językiem. W przeciwieństwie do VB, Microsoft wiedział, że zmuszenie wszystkich do używania wersji C++ opartej wyłącznie na CLR nie jest dobrym pomysłem. Dlatego też Visual Studio 2005, tak jak jego poprzednicy, nadal obsługuje standardowy C++.

Semantyka C++ różni się od tej z CLR

Jednak odwzorowanie C++ na CLR wiązało się ze sporymi wyzwaniami. Co najważniejsze, oryginalna semantyka C++ nie odpowiada dokładnie semantyce CLR. Mają ze sobą wiele wspólnego — na przykład obie są zorientowane obiektowo — jednak istnieje także wiele różnic. Na przykład C++ obsługuje dziedziczenie wielokrotne, czyli sytuację, w której klasa dziedziczy jednocześnie z dwóch lub więcej rodziców, natomiast w CLR takiej możliwości nie ma.

W przeciwieństwie do VB, Microsoft nie może jednostronnie zmieniać C++, tak by pasował on do CLR

VB 6 także znacznie różnił się od CLR, jednak Microsoft jest właścicielem VB. Firma ta może wprowadzać do niego dowolne zmiany, zatem wcielenie VB oparte na .NET zostało zaprojektowane w taki sposób, by pasować do CLR. Microsoft nie posiada jednak C++. Jednostronna zmiana tego języka w taki sposób, by pasował on do CLR, spotkałaby się z falą protestów. Z drugiej strony brak możliwości tworzenia aplikacji opartych na .NET Framework w C++ unieszczęśliwiłby wielu programistów. Jakie jest zatem rozwiązanie tego problemu?

Początkowo Microsoft zdefiniował zbiór Managed Extensions for C++

Pierwszą odpowiedzią Microsoftu było stworzenie zbioru rozszerzeń podstawowego języka C++. Oficjalnie znany pod nazwą ***Managed Extensions for C++***, dialekt ten nazywany jest też po prostu ***Managed C++*** (czyli zarządzanym C++). C++ nie jest językiem łatwym do opanowania, a Managed C++ dodatkowo go komplikuje. Pomimo tego Managed C++ był używany przez wiele organizacji do tworzenia aplikacji .NET.

Wydanie C++ z 2005 roku dodaje bezpośrednie rozszerzenia języka służące do tworzenia kodu zarządzanego

W wydaniu 2005 dla Visual Studio Microsoft udostępnił inny sposób tworzenia zarządzanego kodu w C++. Choć oryginalne rozszerzenia nadal są obsługiwane, są one obecnie zdezaktualizowane. Zamiast tego sam język C++ został zmodyfikowany, dodano do niego nowe słowa kluczowe i inne elementy służące do tworzenia aplikacji, które będą działały na CLR. Zaprojektowany specjalnie w celu tworzenia kodu zarządzanego, dialekt ten znany jest jako ***C++/CLI***. Rozszerzenia te przestrzegają ścieżki standaryzacyjnej oryginalnie zdefiniowanej dla CLI, a celem jest potencjalne udostępnienie dialektu C++/CLI dla środowisk innych od Microsoftu. Niniejszy podrozdział zawiera krótkie wprowadzenie do C++/CLI oraz dialektu Managed C++.

Perspektywa: C++ czy C#?

C++ ma legiony zagorzałych zwolenników. Dlaczego by tak miało nie być? C++ jest potężnym, elastycznym narzędziem do budowania wszelkiego rodzaju aplikacji. Jest także skomplikowany, co oznacza, że nauczenie się wykorzystywania tych możliwości wymaga znacznego wysiłku. Każdy, kto poświęcił tyle czasu na osiągnięcie mistrzostwa w C++, najprawdopodobniej nie będzie uszczęśliwiony na myśl o porzuceniu tego języka.

Jednak dla zupełnie nowych aplikacji zbudowanych od podstaw na bazie .NET Framework język C++ powinien najprawdopodobniej zostać porzucony. Dla programisty C++ opanowanie C# nie jest trudne. W rzeczywistości nauczenie się C# powinno być łatwiejsze niż używanie C++/CLI bądź Managed C++ do pisania aplikacji opartych na .NET Framework. Jak zasugerowano już w krótkim wprowadzeniu w niniejszym rozdziale, te rozszerzenia języka jeszcze bardziej komplikują i tak już skomplikowany język. Dla nowych aplikacji C# będzie prawdopodobnie lepszym wyborem.

Jednak dla celów rozszerzania istniejących aplikacji w C++ za pomocą kodu zarządzanego C++/CLI będzie dobrym wyborem. Również jeśli planuje się przeniesienie istniejącego programu w C++, by działał na .NET Framework, C++/CLI będzie właściwym wyjściem, gdyż zaoszczędzi to konieczności przepisywania na nowo dużych partii kodu. Choć C++ nie jest tak często wykorzystywany w świecie .NET Framework jak C# i VB, język ten jest jednak ważną częścią arsenału językowego .NET.

C++/CLI

Przed spojrzeniem na przykład C++/CLI warto krótko opisać niektóre z rozszerzeń tego języka. By uczynić pisanie kodu opartego na CLR tak naturalnym, jak to tylko możliwe, Microsoft zdecydował się na dodanie pewnych słów kluczowych do tego języka. By uniknąć błędów w istniejącym kodzie C++, słowa kluczowe są używane zgodnie z dwoma interesującymi podejściami:

- **Kontekstowe słowa kluczowe** (ang. *contextual keywords*) mają znaczenie tylko w specyficznym kontekście. Na przykład słowo `sealed` w deklaracji określa, że żaden typ nie może po danym typie dziedziczyć — tak samo jak w C#. To słowo kluczowe ma jednak takie znaczenie tylko wtedy, gdy pojawia się w kontekście deklaracji. Pozwala to istniejącym programom, które wykorzystują identyfikator `sealed` w inny sposób, na przykład w nazwie zmiennej, na pracę bez zmian.

- **Rozdzielone słowa kluczowe** (ang. *spaced keywords*) to pary terminów, które są traktowane jako jedna całość. Na przykład interfejs C++/CLI jest definiowany za pomocą rozdzielonych słów kluczowych `interface class`. Tak jak w przypadku kontekstowego słowa kluczowego, identyfikator `interface` ma specjalne znaczenie tylko w tym kontekście, zatem istniejący kod, który wykorzystuje go w inny sposób, nie będzie zawierał błędów.

Pamiętając o tych dwóch kwestiach, możliwe jest teraz zrozumienie przykładu.

Przykład C++/CLI

Poniżej znajduje się prosty program pokazany wcześniej w C# i VB, a tym razem wyrażony w C++/CLI. Semantyka jest w zasadzie taka sama jak poprzednio. Zmieniła się jedynie składnia, w której semantyka ta została wyrażona.

```
// Przykład C++/CLI

interface class IMath
{
   int Factorial(int f);
   double SquareRoot(double s);
};

ref class Compute : public IMath
{
   public: virtual int Factorial(int f)
   {
      int i;
      int result = 1;
      for (i=2; i<=f; i++)
         result = result * i;
      return result;
   };

   public: virtual double SquareRoot(double s)
   {
      return System::Math::Sqrt(s);
   }
};

void main(void)
```

```
{
   Compute ^c = gcnew Compute;
   int v;
   v = 5;
   System::Console::WriteLine(
      "{0} silnia: {1}",
      v, c->Factorial(v));
   System::Console::WriteLine(
      "Pierwiastek kwadratowy z {0}: {1:f4}",
      v, c->SquareRoot(v));
}
```

C++/CLI przypomina C#

Pierwszą kwestią, na którą warto zwrócić uwagę, jest podobieństwo tego przykładu do wersji w C#. Większość podstawowej składni oraz wiele operatorów jest takich samych. Występują także różnice, które rozpoczynają się do instrukcji `#define`, potrzebnej do tworzenia kodu zarządzanego w C++. Po niej, tak jak poprzednio, następuje definicja interfejsu `IMath`. Tym razem jednak wykorzystane są słowa kluczowe `interface class`, opisane powyżej. Wynikiem jest inkarnacja interfejsu zdefiniowanego w CTS w C++.

Klasa CTS jest definiowana za pomocą ref class

Następnie pojawia się klasa `Compute`, która implementuje interfejs `IMath`. Klasa ta jest poprzedzona słowem kluczowym C++/CLI `ref class`, oznaczającym, że jest to klasa referencyjna CTS, której czas życia jest zarządzany przez CLR za pomocą czyszczenia pamięci. Sama klasa różni się nieco, jeśli chodzi o składnię, od przykładu w C#, ponieważ C++ nie wyraża wszystkiego dokładnie w ten sam sposób. Tym niemniej jest ona bardzo podobna.

Obiekty typów referencyjnych są tworzone za pomocą gcnew

Przykład kończy się standardową funkcją C++: `main`. Tak jak w poprzednich przykładach, tworzy ona obiekt klasy `Compute`, a następnie wywołuje jego dwie metody — wszystko to za pomocą standardowej składni C++. Najbardziej widoczną różnicą pomiędzy tymi dwoma przykładami (i pomiędzy standardowym C++) jest użycie słowa kluczowego `gcnew`. Słowo to oznacza, że tworzony jest obiekt klasy CTS (to znaczy klasa, która będzie poddana procesowi czyszczenia pamięci, inaczej *garbage-collected class* — stąd „gc" w `gcnew`). Innymi słowy, klasa `Compute` jest tworzona na stercie zarządzanej przez CLR, a nie na wbudowanym stosie, utrzymywanym przez C++. Obiekty, które powstały za pomocą standardowego operatora C++ `new`, są — tak jak zawsze — tworzone na wbudowanym stosie.

Inną różnicą jest pojawienie się w deklaracji klasy `Compute` symbolu `^`, po angielsku popularnie zwanego ***caret*** lub ***hat***, a po polsku daszkiem. Standardowy C++ do wskazania referencji używa tradycyjnej gwiazdki. By można jednak było od razu zauważyć, że w grę wchodzi typ referencyjny CTS, w C++/CLI wprowadzono pomysł **uchwytu** (ang. *handle*). Uchwyt taki jak zadeklarowany powyżej, identyfikowany przez ten nowy symbol, może czasami być wykorzystywany w sposób podobny do zwykłych wskaźników C++, jakie pokazano w wywołaniach `Factorial` i `SquareRoot` w dalszej części programu. Ponieważ jednak uchwyt jest tak naprawdę referencją do obiektu, który będzie później poddany czyszczeniu pamięci na stercie zarządzanej przez CLR, w rzeczywistości różni się on od zwykłego wskaźnika C++. Nowa składania podkreśla tę różnicę. I jak można by oczekiwać, wynik dla powyższego przykładu będzie taki sam jak poprzednio: będzie to silnia oraz pierwiastek kwadratowy z pięciu.

Referencje do klasy CTS odbywają się za pomocą uchwytów

Typy w C++/CLI

C++/CLI pozwala na pełny dostęp do .NET Framework, w tym także do typów zdefiniowanych przez CLR. Należy zauważyć, że kod zarządzany oraz niezarządzany, a także klasy zdefiniowane z `ref` i bez niego mogą być definiowane w tym samym pliku i mogą współistnieć w jednym działającym procesie. Jednak jedynie klasy zarządzane są poddawane czyszczeniu pamięci; klasy niezarządzane muszą być jawnie zwalniane, tak jak się to zwykle odbywa w C++. Tabela 3.3 pokazuje niektóre najważniejsze typy CTS oraz ich odpowiedniki w C++/CLI.

Kod zarządzany i kod niezarządzany w C++ mogą współegzystować w procesie

Inne cechy C++/CLI

Ponieważ C++/CLI w pełni obsługuje CLR, istnieje w nim o wiele więcej możliwości. Właściwości mogą na przykład być definiowane za pomocą słowa kluczowego `property`, natomiast delegaty są tworzone za pomocą słowa kluczowego `delegate`. C++/CLI obsługuje zarówno typy generyczne zdefiniowane w CLR, jak również ich kuzynów, czyli standardowe szablony C++. Referencje do przestrzeni nazw wykonuje się za pomocą instrukcji `using namespace`, jak poniżej:

C++/CLI pozwala na pełny dostęp do wszystkiego, co dostarcza CLR

```
using namespace System;
```

Obsługa wyjątków odbywa się z wykorzystaniem bloków `try`/`catch`. Mogą być też tworzone własne wyjątki, dziedziczące po `System::Exception`. Atrybuty mogą być także osadzane w kodzie za pomocą składni podobnej do używanej w C#.

Tabela 3.3. Niektóre typy CTS oraz ich odpowiedniki w C++/CLI

CTS	C++/CLI
Byte	unsigned char
Char	wchar_t
Int16	short, signed short
Int32	int, signed int, long, signed long
Int64	__int64, signed__int64
UInt16	unsigned short
UInt32	unsigned int, unsigned long
UInt64	unsigned__int64
Single	float
Double	double, long double
Decimal	Decimal
Boolean	bool
Class	ref class, ref struct
Interface	interface class
Delegate	delegate

C++ jest jedynym językiem w Visual Studio 2005, który może być kompilowany bezpośrednio do rdzennego kodu

Za wyjątkiem C++ wszystkie pozostałe języki z Visual Studio są kompilowane do MSIL i do uruchomienia potrzebują .NET Framework. Ponieważ wszystkie klasy C++/CLI są kompilowane do MSIL, język ten może oczywiście być wykorzystywany do generowania kodu opartego na .NET Framework. Jednak C++ jest wyjątkowy pośród języków opartych na .NET Framework, ponieważ możliwe jest również kompilowanie go bezpośrednio do plików binarnych dla danej maszyny. Przy budowaniu aplikacji dla Windows, które nie wymagają CLR, C++ jest dobrym rozwiązaniem.

Managed C++

Używanie Managed C++ jest obecnie niezalecane

Visual Studio .NET, oryginalne narzędzie Microsoftu do tworzenia aplikacji .NET, wprowadziło Managed C++ w celu umożliwienia tworzenia oprogramowania opartego na CLR w języku C++. Od premiery Visual Studio 2005 używanie Managed C++ jest odradzane. Nadal jednak wiele osób pisze (a jeszcze częściej rozszerza) aplikacje w C++ za pomocą tej oryginalnej próby połączenia C++ i CLR. Biorąc pod

uwagę ten fakt, warto rzucić okiem na ten obecnie zdezaktualizowany dialekt. Interesujące będzie także porównanie go z jego następcą, C++/CLI.

Tak jak C++/CLI, Managed C++ również definiuje kilka nowych słów kluczowych

Przed spojrzeniem na przykład w Managed C++ warto jednak opisać niektóre rozszerzenia znajdujące się w tym języku. Tak jak w przypadku C++/CLI, do języka tego dodano kilka słów kluczowych, które pozwalają na dostęp do usług CLR. Wszystkie słowa kluczowe rozpoczynają się od dwóch znaków podkreślenia (__), zgodnie z konwencją zdefiniowaną w standardzie ANSI dla rozszerzeń C++. Wśród najważniejszych rozszerzeń znajdują się następujące:

- **__gc** — oznacza typ CTS, który poddany będzie procesowi czyszczenia pamięci, czyli typ referencyjny CTS.
- **__value** — oznacza typ CTS, który nie będzie poddany procesowi czyszczenia pamięci, czyli typ bezpośredni CTS.
- **__interface** — używany jest w celu definiowania typu interfejsu CTS.
- **__box** — operacja, która konwertuje typ bezpośredni CTS na typ referencyjny. W przeciwieństwie do C#, VB i C++/CLI, Managed C++ nie wykonuje operacji pakowania i odpakowywania w sposób niejawny. Zamiast tego programiści muszą jawnie oznaczyć miejsca, w których takie konwersje powinny wystąpić.
- **__unbox** — operacja, która konwertuje zapakowany typ bezpośredni CTS z powrotem na oryginalną postać.

Jak w poprzednich podrozdziałach, nadszedł czas na przykład.

Przykład Managed C++

Poniżej znajduje się standardowy przykład, tym razem w Managed C++:

```
// Przykład Managed C++
#using <mscorlib.dll>

__gc __interface IMath
{
   int Factorial(int f);
   double SquareRoot(double s);
};

__gc class Compute : public IMath
```

```
{
   public: int Factorial(int f)
   {
      int i;
      int result = 1;
      for (i=2; i<=f; i++)
         result = result * i;
      return result;
   };
   public: double SquareRoot(double s)
   {
      return System::Math::Sqrt(s);
   }
};

void main(void)
{
   Compute *c = new Compute;
   int v;
   v = 5;
   System::Console::WriteLine(
      "{0} silnia: {1}",
      __box(v), __box(c->Factorial(v)));
   System::Console::WriteLine(
      "Pierwiastek kwadratowy z {0}: {1:f4}",
      __box(v), __box(c->SquareRoot(v)));
}
```

Managed C++ przypomina C++/CLI i C#

Nie jest niespodzianką, że przykład ten wygląda podobnie do wersji pokazanych wcześniej w C# oraz C++/CLI. Różnice są jednak interesujące i rozpoczynają się od instrukcji `#include` oraz `#using`, niezbędnych do stworzenia kodu w Managed C++. Ponownie definiowany jest interfejs `IMath`, jednak tym razem za pomocą słowa kluczowego `__interface`, poprzedzonego słowem kluczowym `__gc`. Kombinacja ta ma takie samo znaczenie co `interface class` w C++/CLI. Klasa `Compute` jest również deklarowana ze słowem kluczowym `_gc`, co jest innym sposobem wyrażenia tego samego, co w C++/CLI robi się za pomocą `ref`.

Managed C++ wymaga jawnego pakowania

Przykład ten kończy się standardową funkcją `main` C++. Tak jak wcześniej, tworzy ona obiekt klasy `Compute`, a następnie wywołuje jego dwie metody; wszystko to za pomocą standardowej składni C++. Jedyną istotną różnicą jest wywołanie `WriteLine`. Ponieważ metoda ta oczekuje parametrów referencyjnych, operator `__box` musi zostać użyty w celu poprawnego przekazania parametrów liczbowych. Pako-

wanie tego parametru pojawiło się także w C# i VB, jednak było wykonane automatycznie. Ponieważ C++ nie był oryginalnie zaprojektowany dla CLR, programista Managed C++ musi jawnie wywołać tę operację.

Typy w Managed C++

Managed C++ jest funkcjonalnie podobny do C++/CLI

Tak jak C++/CLI, Managed C++ pozwala na pełny dostęp do .NET Framework oraz na definiowanie kodu zarządzanego i niezarządzanego w jednym pliku. Oba dialekty C++ w pewnym sensie dostarczają jedynie innych sposobów na wyrażenie tej samej semantyki. Tabela 3.4 prezentuje niektóre główne typy CLR wraz z ich odpowiednikami w Managed C++.

Tabela 3.4. Niektóre typy CLR oraz ich odpowiedniki w Managed C++

CLR	Managed C++
`Byte`	`unsigned char`
`Char`	`wchar_t`
`Int16`	`short`
`Int32`	`int, long`
`Int64`	`__int64`
`UInt16`	`unsigned short`
`UInt32`	`unsigned int, unsigned long`
`UInt64`	`unsigned __int64`
`Single`	`float`
`Double`	`double`
`Decimal`	`Decimal`
`Boolean`	`bool`
`Class`	`__gc class`
`Interface`	`__gc __interface`
`Delegate`	`__delegate`

Inne cechy Managed C++

Managed C++ umożliwia także pełny dostęp do cech CLR

Tak jak C++/CLI, Managed C++ pozwala na pełny dostęp do CLR. Delegaty mogą być tworzone za pomocą słowa kluczowego `__delegate`, referencje do przestrzeni nazw odbywają się za pomocą `using namespace`, tak samo jak w C++/CLI; mogą także być wykorzystywane

Perspektywa: czy C++ jest wymierającym językiem?

C++ był narzędziem pracy dla zawodowych programistów przez większość lat 90. ubiegłego wieku. Był wykorzystany do napisania Lotus Notes, większości aplikacji biznesowych i nawet części Windows. Czy jednak w świecie oferującym C#, nowoczesną wersję VB i Javę nadal jest miejsce dla C++? Czy jego przydatność się wyczerpała?

Z całą pewnością nie. C#, VB i Java są lepsze od C++ dla wielu rodzajów zastosowań, w tym wielu takich, w których tradycyjnie wykorzystywano C++. Jednak wszystkie trzy języki operują w środowisku maszyny wirtualnej. Ma to wiele korzyści, ale wiąże się z tym pewna cena — jest nią wydajność i rozmiar. Niektóre kategorie oprogramowania, takie jak pewne aplikacje czasu rzeczywistego czy kod poziomu systemu, nie mogą sobie na to pozwolić.

Mimo to skończyły się czasy, w których C++ był domyślnym wyborem dla budowania szerokiej gamy nowych aplikacji. W świecie Microsoftu domyślnie wybiera się teraz C# i VB, natomiast Java dominuje w innych kręgach. Jednak w przypadkach gdy żadne z tych rozwiązań nie jest właściwe — a takie przypadki nadal istnieją — C++ będzie nadal dominował. Jego rola z pewnością się skurczyła, jednak C++ nie zniknie.

wyjątki i atrybuty. Managed C++ nie jest słabym narzędziem, które zostało zdezaktualizowane ze względu na małe możliwości. Było raczej tak, że osoby, które kontrolują tę technologię w firmie Microsoft, uznały, że ich pierwsza próba odwzorowania C++ na CLR nie była wystarczająco dobra, zatem by iść z duchem postępu, nowy kod zarządzany C++ powinien być tworzony za pomocą C++/CLI.

Wniosek

.NET Framework przynosi nowe podejście do projektowania języków programowania

Języki programowania są fascynującym tematem. Obecnie wydaje się, że istnieje szeroka zgoda co do fundamentalnych cech, jakie powinien mieć współczesny język programowania ogólnego przeznaczenia, a także jego zachowania. Nie ma jednak zgody co do wyglądu takiego języka programowania, ponieważ każdemu odpowiada jego własna ulubiona składnia. Dostarczając wspólną implementację podstaw i pozwalając następnie na różne sposoby wyrażania tych podstaw, .NET Framework przyniósł zupełnie nowe podejście do projektowania języków. Nawet bez wsparcia Microsoftu byłby to atrakcyjny model tworzenia środowiska programistycznego. W połączeniu ze wsparciem ze strony największego producenta oprogramowania na świecie .NET Framework ułatwił życie wielu, wielu programistów.