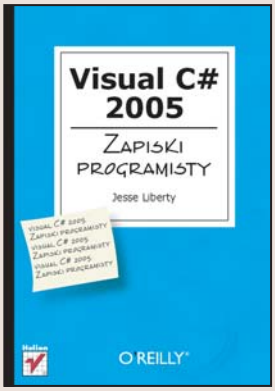
Visual C# 2005. Zapiski programisty



Autor: Jesse Liberty
Tłumaczenie: Przemysław Szeremiota
ISBN: 83-246-0249-6
Tytuł oryginału: Visual C# 2005: A Developers Notebook
Format: B5, stron: 280
Przykłady na ftp: 1162 kB

Odkryj nowe możliwości platformy .NET 2005

Visual C# 2005 to najnowsza wersja języka programowania uważanego przez wielu programistów za najlepszy język służący do tworzenia aplikacji dla platformy .NET. W połączeniu z nową biblioteką klas .NET i nowymi możliwościami środowiska Visual Studio 2005 druga edycja języka C# stała się jeszcze doskonalsza. Pisanie programów wymaga znacznie mniejszych nakładów pracy, a nowe elementy umożliwiają realizację większej ilości zadań programistycznych.

Aby poznać nowe możliwości języka C#, sięgnij po książkę „Visual C# 2005. Zapiski programisty". W tej wzorowanej na zeszytach laboratoryjnych publikacji znajdziesz notatki programistów, którzy jako pierwsi zetknęli się z tą technologią. Nie ma w niej teoretycznych wywodów, diagramów i niepotrzebnych informacji. Wykonując 50 ćwiczeń demonstrujących poszczególne aspekty tworzenia aplikacji, poznasz prostotę stosowania nowych elementów i mechanizmów i przekonasz się, jak wiele udogodnień wnosi do pracy programisty Visual C# 2005.

- Stosowanie klas generycznych
- Korzystanie z metod anonimowych
- Refaktoryzacja kodu źródłowego
- Tworzenie interfejsów użytkownika i formularzy
- Mechanizmy szybkiej instalacji aplikacji
- Zabezpieczanie aplikacji WWW
- Personalizacja stron WWW z użyciem motywów i szablonów
- Połączenia z bazą danych

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

Spis treści

ROZDZIAŁ 1.

C# 2.0

W tym rozdziale:

- ✓ Tworzenie typowanych list za pomocą kolekcji generycznych
- ✓ Własne kolekcje generyczne
- ✓ Implementacja interfejsów kolekcji
- ✓ Stosowanie iteratorów generycznych
- ✓ Implementacja `GetEnumerator` dla złożonych struktur danych
- ✓ Upraszczanie kodu — metody anonimowe
- ✓ Ukrywanie kodu — typy częściowe
- ✓ Tworzenie klas statycznych
- ✓ Wyrażanie wartości pustych typami nullable
- ✓ Odwołania do obiektów z globalnej przestrzeni nazw
- ✓ Ograniczanie dostępu do właściwości
- ✓ Elastyczność delegacji z kowariancją i kontrawariancją

Pierwszy rozdział będzie prezentował nowe cechy języka C# w wersji 2.0 — mowa będzie o typach generycznych, iteratorach, metodach anonimowych, typach częściowych, klasach statycznych, typach wyróżniających wartości puste („nullable") oraz ograniczaniu dostępu do właściwości; omówienie obejmie też kowariancję i kontrawariancję delegacji.

Najbardziej chyba wyczekiwaną cechą języka C# w wersji 2.0 są jednak typy generyczne, dające możliwość szybkiego tworzenia kolekcji. I od nich właśnie zaczniemy.

Tworzenie typowanych list za pomocą kolekcji generycznych

Bezpieczeństwo typowania to klucz do tworzenia kodu prostego w konserwacji. Język ze ścisłą kontrolą typów potrafi (skutecznie) wyszukiwać błędy typowania w czasie kompilacji, a nie w czasie wykonania (kiedy program zostanie już oddany do użytku klientom!). Największą słabością C# 1.x była nieobecność typów generycznych, które pozwalałyby na deklarowanie kolekcji uogólnionych (choćby stosów bądź list) mogących przechowywać elementy dowolnego typu przy zachowaniu możliwości kontrolowania typu w czasie kompilacji.

W wersji 1.x niemal wszystkie kolekcje były deklarowane jako kolekcje przechowujące obiekty typu `System.Object`; ponieważ zaś *wszelkie* klasy wywodzą się z `System.Object`, kolekcje takie mogły przechowywać elementy dosłownie dowolnych typów. Elastyczność ta osiągana była kosztem bezpieczeństwa typowania.

Załóżmy, że w C# 1.x zajdzie potrzeba utworzenia listy obiektów `Employee`. W tym celu można by wykorzystać klasę `ArrayList` pozwalającą na przechowywanie obiektów typu `System.Object`. Dodanie nowych obiektów `Employee` do takiej kolekcji nie stanowi żadnego problemu — przecież typ `Employee` jest pochodną `System.Object`. Problem pojawia się dopiero przy próbie wyłuskania obiektu `Employee` z kolekcji `ArrayList` — operacja taka zwraca referencję do typu `Object`, którą trzeba dopiero rzutować na typ `Employee`:

```
Employee theEmployee = (Employee) myArrayList[1];
```

Sęk nie w tym, że trzeba dokonywać rzutowania, ale w tym, że w kolekcji `ArrayList` można równie łatwo umieścić obiekt dowolnego innego typu, na przykład ciąg znaków. I wszystko będzie w porządku dopóty, dopóki program będzie się do niego odwoływał jak do ciągu. Ale jeśli kolekcja ciągów zostanie omyłkowo przekazana do metody oczekującej przekazania

kolekcji obiektów `Employee`, próba rzutowania obiektu `String` na typ `Employee` sprowokuje wyjątek.

Wreszcie kolekcje platformy .Net 1.x utrudniały przechowywanie wartości typów prostych (ang. *value type*). Wartości typów prostych musiały być przed wstawieniem do kolekcji pakowane do obiektów, a po wyłuskaniu z kolekcji — rozpakowywane z obiektów.

W .NET 2.0 wszystkie te problemy zostały wyeliminowane przez udostępnienie nowej biblioteki kolekcji generycznych, zagnieżdżonej w przestrzeni nazw `System.Collections.Generic`. Otóż *kolekcja generyczna* to po prostu taka kolekcja, która pozwala na ujęcie w deklaracji typów elementów przechowywanych. Dzięki temu kompilator znający deklarację będzie zezwalał na wstawianie do kolekcji jedynie obiektów właściwego typu. Kolekcje generyczne definiuje się z użyciem specjalnej składni; w nawiasach ostrych wymienia się nazwę typu, który musi zostać zdefiniowany przy deklarowaniu egzemplarza kolekcji.

Nie trzeba już rzutować obiektów wyłuskiwanych z kolekcji generycznej, a sam kod korzystający z takich kolekcji jest bezpieczniejszy, bo podlega statycznemu (realizowanemu w czasie kompilacji) typowaniu; łatwiej poddaje się konserwacji i prościej się go stosuje.

Kolekcje generyczne czynią kod bezpieczniejszym, upraszczają jego konserwację i stosowanie.

Jak to zrobić?

Aby oswoić się z nowymi typami generycznymi w .NET 2.0, najlepiej spróbować samemu utworzyć typowaną klasę kolekcji (`List`) przechowującą elementy typu `Employee` („pracownik"). Ćwiczenie należy rozpocząć od uruchomienia środowiska Visual Studio 2005, utworzenia nowej aplikacji konsolowej języka C# i opatrzenia jej nazwą `CreateATypeSafeList`. Kod generowany w ramach szkieletu aplikacji przez kreator Visual Studio 2005 należy zastąpić kodem z listingu 1.1.

WSKAZÓWKA

Korzystanie z typów generycznych wymaga wciągnięcia do programu przestrzeni nazw `System.Collections.Generic`. W Visual Studio 2005 jest to domyślne zachowanie dla wszystkich tworzonych projektów.

Listing 1.1. Tworzenie typowanej listy

```
using System;
using System.Collections.Generic;

namespace CreateATypeSafeList
{
// klasa obiektów przechowywanych na liście
public class Employee
{
    private int empID;

    // konstruktor
    public Employee(int empID)
    {
        this.empID = empID;
    }

    // przesłonięcie metody ToString, tak
    // aby wyświetlała identyfikator obiektu
    public override string ToString()
    {
        return empID.ToString();
    }
}  // koniec klasy

// Klasa testująca
public class Program
{
    // punkt wejściowy
    static void Main()
    {
        // Deklaracja listy typowanej (obiektów typu Employee)
        List<Employee> empList = new List<Employee>();

        // Deklaracja drugiej listy typowanej (wartości całkowitych)
        List<int> intList = new List<int>();

        // wypełnienie list
        for (int i = 0; i < 5; i++)
        {
            empList.Add(new Employee(I + 100));
            intList.Add(i * 5);
            // empList.Add(i * 5);       // patrz punkt "A co…"
        }

        // wypisanie elementów listy wartości całkowitych
        foreach (int i in intList)
        {
            Console.Write("{0} ", i.ToString());
        }

        Console.Write("\n");
```

```
            // wypisanie identyfikatorów obiektów Employee
            foreach (Employee employee in empList)
            {
                Console.Write("{0} ", employee.ToString());
            }

            Console.Write("\n");
        }
    }
    }
```

Wynik:

```
0 5 10 15 20
100 101 102 103 104
```

WSKAZÓWKA

Kod źródłowy poszczególnych ćwiczeń można pobrać z serwerów wydawnictwa Helion, spod adresu *ftp://ftp.helion.pl/przyklady/vc25za.zip*. Kod publikowany jest w postaci spakowanego archiwum. Rozpakowanie archiwum zaowocuje utworzeniem szeregu katalogów odpowiadających poszczególnym rozdziałom, a w nich podkatalogów o nazwach zgodnych z nazwami poszczególnych projektów. Na przykład kodu z listingu 1.1 należy szukać w katalogu *r1\CreateATypeSafeList*.

Jak to działa?

Kod z listingu 1.1 utworzył dwie klasy: klasę `Employee` (pracownik) — klasę obiektów przechowywanych w kolekcji, oraz klasę `Program` tworzoną przez kreator Visual Studio 2005. Do tego w programie wykorzystana została klasa `List` z biblioteki klas .NET Framework Class Library.

Klasa `Employee` zawiera pojedyncze prywatne pole (`empID`), konstruktor i metodę przesłaniającą metodę `ToString` i wyświetlającą ciąg zawierający wartość pola `empID`.

W klasie `Program` tworzony jest egzemplarz klasy `List` mający przechowywać obiekty klasy `Employee`. Typem `empList` jest więc „kolekcja `List` obiektów `Employee`". Stąd deklaracja kolekcji:

```
List<Employee> empList
```

W definicji `List<T>` element `T` jest symbolem reprezentującym przyszły, właściwy typ listy.

Deklaracja `empList` tworzy (jak zwykle) referencję obiektu tworzonego na stercie słowem kluczowym `new`. Słowo kluczowe `new` należy uzupełnić wywołaniem konstruktora, co wygląda następująco:

```
new List<Employee>()
```

Takie wywołanie tworzy na stercie egzemplarz „kolekcji `List` obiektów `Employee`"; cała instrukcja oznacza zaś utworzenie `empList` i przypisanie do niego referencji nowego obiektu sterty:

```
List<Employee> empList = new List<Employee>();
```

WSKAZÓWKA

Całość działa dokładnie tak, jak w instrukcji:

```
Dog milo = new Dog();
```

tworzącej egzemplarz klasy `Dog` na stercie i przypisującej go do referencji do typu `Dog` o nazwie `milo`.

W kolejnej instrukcji następuje utworzenie drugiej kolekcji; tym razem jest to „kolekcja `List` wartości całkowitych":

```
List<int> intList = new List<int>();
```

Od tego momentu można rozpocząć wypełnianie listy wartości całkowitych wartościami całkowitymi, a listy elementów typu `Employee` obiektami klasy `Employee`. Po wypełnieniu list można w pętlach `foreach` przejrzeć listy, wyłuskać poszczególne elementy i wypisać ich wartości na konsoli:

```
foreach (Employee employee in empList)
{
    Console.Write("{0} ", employee.ToString());
}
```

A co...

... kiedy do listy obiektów `Employee` dodana zostanie wartość całkowita?

Cóż, trzeba spróbować. Wystarczy usunąć znacznik komentarza z prezentowanego poniżej wiersza z listingu 1.1 i spróbować ponownie skompilować program:

```
empList.Add(i * 5);
```

Kompilator powinien zgłosić parę błędów:

```
   Error   1      The best overloaded method match for
'System.Collections.
Generic.List<ListCollection.Employee>.Add(ListCollection.Employee)' has
some
invalid arguments
   Error   2      Argument '1': cannot convert from 'int' to
'ListCollection.Employee'
```

Komunikaty opisujące te dwa błędy pozwalają stwierdzić, że nie można dodawać elementu typu `int` do kolekcji obiektów typu `Employee`, bo pomiędzy tymi typami nie da się przeprowadzić niejawnej konwersji, nie zachodzi też relacja zawierania się jednego typu w drugim.

Co ważniejsze, kolizję typów wykrywa się już na etapie kompilacji, a nie dopiero w czasie wykonania — a wiadomo, że błędy czasu wykonania mają tendencję do ujawniania się nie na stanowiskach testowych, a na biurkach klientów!

W kolekcjach typowanych można umieszczać elementy typów pochodnych wobec typu deklarowanego. Kolekcja elementów typu Employee może więc przechowywać również elementy typu Manager, o ile Manager jest typem pochodnym Employee.

... z innymi kolekcjami generycznymi; są jeszcze jakieś?

Owszem, w platformie .NET 2.0 dostępne są też inne kolekcje typowane, jak choćby `Stack` (stos) czy `Queue` (kolejka); do tego dochodzi interfejs `ICollection`.

Kolekcje te stosuje się tak samo, jak `List<T>`. Aby na przykład utworzyć stos obiektów typu `Employee`, należy w definicji klasy `Stack` zastąpić `T` (`Stack<T>`) nazwą typu `Employee`:

```
Stack<Employee> employeeStack = new Stack<Employee>();
```

Więcej informacji

Kompletu informacji o klasach generycznych w .NET 2.0 należy szukać w pomocy MSDN, pod hasłem „Commonly Used Collection Types"; warto też zapoznać się z artykułem publikowanym w witrynie ONDotnet.com (O'Reilly) pod adresem *http://www.ondotnet.com/pub/a/dotnet/2004/05/17/liberty.html*.

WSKAZÓWKA

Artykuły i publikacje, na które powołuję się w kolejnych ćwiczeniach, są wymienione na stronie WWW książki pod adresem *http://www.LibertyAssociates.com* (odnośnik *Books*, a następnie pozycja *C# 2005: A Developer's Notebook*).

Następne ćwiczenie będzie ilustrowało sposób tworzenia własnej typowanej kolekcji, uzupełniającej zbiór kolekcji udostępnianych w bibliotece klas platformy .NET.

Własna kolekcja generyczna

Platforma .NET 2.0 udostępnia programistom szereg klas kolekcji generycznych implementujących typowane listy, stosy, kolejki, słowniki i tym podobne. Tak bogaty zestaw zaspokaja zdecydowaną większość potrzeb programistów w tym zakresie. Załóżmy jednak, że stoimy w obliczu konieczności utworzenia własnej klasy kolekcji generycznej, uwzględniającej wiedzę z dziedziny danego problemu czy przejawiającej specjalne cechy (na przykład optymalizującej rozmieszczenie elementów celem przyspieszenia odwołań). Na szczęście i sam język, i platforma udostępniają narzędzia i mechanizmy tworzenia własnych klas kolekcji generycznych.

Od czasu do czasu przyjdzie Ci zdefiniować własną klasę kolekcji generycznych.

Jak to zrobić?

Najprostszym sposobem powołania do życia własnej klasy kolekcji generycznej jest utworzenie danej kolekcji (np. kolekcji wartości całkowitych) i potem zastąpienie nazwy typu `int` uogólnionym symbolem typu, na przykład `T`.

Mianowicie:

```
private int data;
```

ma przyjąć postać:

```
private T data;        // T to generyczny parametr typowy
```

Generyczny parametr typowy (tutaj `T`) jest ściśle definiowany dopiero w momencie tworzenia klasy kolekcji przez parametr typu występujący między nawiasami kątowymi (`< >`):

```
public class Node<T>
```

Tak definiuje się nowy typ — „węzeł `Node` typu `T`"; jeśli w definicji obiektu miejsce `T` zajmie `int`, dla środowiska wykonawczego obiekt ten będzie typu „węzeł `Node` typu `int`" (analogicznie tworzy się węzły dowolnego innego typu).

WSKAZÓWKA

Wielu programistów stosuje wygodny w zapisie symbol typu `T`, ale Microsoft zaleca stosowanie dłuższych, bardziej opisowych symboli (na przykład `Node<DocumentType>`).

Listing 1.2 prezentuje przykładowy kod tworzący listę węzłów typu `T` i następnie powołujący do życia dwa egzemplarze takich list dla różnych właściwych obiektów węzłów.

Listing 1.2. Własna kolekcja generyczna

```
using System;

namespace GenericLinkedList
{
  public class Pilgrim
  {
      private string name;
      public Pilgrim(string name)
      {
          this.name = name;
      }
      public override string ToString()
      {
          return this.name;
      }
  }
  public class Node<T>
  {
      // pola składowych
      private T data;
      private Node<T> next = null;

      // konstruktor
      public Node(T data)
      {
          this.data = data;
      }

      // właściwości
      public T Data { get { return this.data; } }

      public Node<T> Next
      {
          get { return this.next; }
      }

      // metody
      public void Append(Node<T> newNode)
```

```
        {
            if (this.next == null)
            {
                this.next = newNode;
            }
            else
            {
                next.Append(newNode);
            }
        }
        public override string ToString()
        {
            string output = data.ToString();

            if (next != null)
            {
                output += ", " + next.ToString();
            }

            return output;
        }
    }  // koniec klasy


    public class LinkedList<T>
    {
        // pola składowe
        private Node<T> headNode = null;

        // właściwości

        // indekser
        public T this[int index]
        {
            get
            {
                int ctr = 0;
                Node<T> node = headNode;

                while (node != null && ctr <= index)
                {
                    if (ctr == index)
                    {
                        return node.Data;
                    }
                    else
                    {
                        node = node.Next;
                    }

                    ++ctr;
                }  // koniec while
                throw new ArgumentOutOfRangeException();
```

```
        }  // koniec get
    }  // koniec indeksera


    // konstruktor
    public LinkedList()
    {
    }

    // metody
    public void Add(T data)
    {
        if (headNode == null)
        {
            headNode = new Node<T>(data);
        }
        else
        {
            headNode.Append(new Node<T>(data));
        }
    }
    public override string ToString()
    {
        if (this.headNode != null)
        {
            return this.headNode.ToString();
        }
        else
        {
            return string.Empty;
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
        LinkedList<int> myLinkedList = new LinkedList<int>();
        for (int i = 0; i < 10; i++)
        {
            myLinkedList.Add(i);
        }


        Console.WriteLine("Liczby: " + myLinkedList);
        LinkedList<Pilgrim> pilgrims = new LinkedList<Pilgrim>();
        pilgrims.Add(new Pilgrim("Rycerz"));
        pilgrims.Add(new Pilgrim("Młynarz"));
        pilgrims.Add(new Pilgrim("Szeryf"));
        pilgrims.Add(new Pilgrim("Kucharz"));

        Console.WriteLine("Pielgrzymi: " + pilgrims);
        Console.WriteLine("Czwarta liczba to " + myLinkedList[3]);
```

```
            Pilgrim d = pilgrims[1];
            Console.WriteLine("Drugi pielgrzym to " + d);
        }
    }
}
```

Wynik:

```
Liczby: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Pielgrzymi: Rycerz, Młynarz, Szeryf, Kucharz
Czwarta liczba to 3
Drugi pielgrzym to Młynarz
```

Jak to działa?

Otóż właśnie powstała *generyczna* kolekcja — lista. Kolekcja podlegająca typowaniu — można deklarować jej egzemplarze dla dowolnych typów. Najłatwiej skonstruować listę tego rodzaju, bazując na implementacji zwykłej (nie generycznej) listy dla konkretnego typu danych. Ten konkretny przykład przy definiowaniu listy definiuje listę generyczną, której czoło (węzeł czołowy) jest inicjalizowany wartością pustą:

```
public class LinkedList<T>
{
    private Node<T> headNode = null;
  ...
}
```

Tworzenie kolekcji z typami generycznymi jest doprawdy bardzo proste. Bo cóż prostszego, niż zaimplementować kolekcję dla dowolnie wybranego typu, a następnie zastąpić nazwę typu parametrem typu generycznego <T>.

Kiedy do listy dodawane są elementy, tworzony jest obiekt nowego węzła; w obliczu braku węzła czołowego ów nowy węzeł pełni rolę czoła listy; jeśli czoło listy jest już zajęte, nowy węzeł jest dołączany do listy metodą `Append` węzła czołowego.

Przy dodawaniu nowego węzła do listy następuje przegląd węzłów składających się na listę; szukany jest węzeł ostatni, czyli taki, który we wskaźniku następnego węzła (pole `next`) przechowuje wartość pustą. Po jego odnalezieniu do pola `next` przypisywana jest referencja nowego węzła.

Lista `LinkedList` została celowo zadeklarowana z identycznym parametrem typu jak `Node`. Obie stosują w roli symbolu generycznego parametru typu tę samą literę (`T`), dzięki czemu kompilator „wie", że typ podstawiany w miejsce `T` w definicji `LinkedList` ma również zastępować `T` w definicji `Node`. To w zasadzie oczywiste: przecież węzłami listy wartości całkowitych powinny być węzły wartości całkowitych.

A co…

… ze stosowaniem typów generycznych w innych strukturach? Czy to możliwe?

Owszem, typy generyczne można stosować w strukturach, interfejsach, delegacjach, a nawet w metodach.

Więcej informacji

Szczegółowych informacji o tworzeniu własnych klas z udziałem typów generycznych należy szukać w pomocy MSDN pod hasłem „Topic: Generics", ewentualnie we wspomnianym już artykule z witryny ONDotnet.com (O'Reilly), *http://www.ondotnet.com/pub/a/dotnet/2004/05/17/liberty.html*. Warto też zapoznać się z projektem mającym na celu gromadzenie i udostępnianie bibliotek klas dla platformy .NET, o którym więcej na stronie Wintellect (*http://www.wintellect.com/powercollections/*).

Implementacja interfejsów kolekcji

W platformie .NET 2.0 poza generycznymi klasami kolekcji znajduje się również zestaw generycznych interfejsów pozwalających na tworzenie typowanych kolekcji cechujących się pełnym zakresem funkcji wcześniejszych, niegenerycznych klas kolekcji z biblioteki platformy .NET 1.x. Interfejsy te zostały zebrane w przestrzeni nazw `System.Collections.Generic`, do której trafiło również sporo interfejsów pomocniczych, w tym `IComparable<T>` służący do porównywania dwóch obiektów typu `T` (niekoniecznie przechowywanych w kolekcji).

Wszystko to pozwala na proste utworzenie kolekcji w postaci uporządkowanej listy elementów — wystarczy dla każdego typu `T` danych przechowywanego w elementach listy zaimplementować interfejs `IComparable<T>` i uczynić obiekt `Node` odpowiedzialnym za wstawianie nowych węzłów (obiektów `Node`) w odpowiednie miejsca listy — tak aby zachować uporządkowanie węzłów.

Jak to zrobić?

Interfejs `IComparable` jest już zaimplementowany dla typu `Integer`; w prosty sposób można taką implementację utworzyć również dla klasy `Pilgrim`. Trzeba jedynie zmodyfikować definicję klasy `Pilgrim`, tak aby sygnalizowała implementację interfejsu `IComparable<T>`:

```
public class Pilgrim: IComparable<Pilgrim>
```

Rzecz jasna trzeba też koniecznie zaimplementować wymagane metody interfejsu: `CompareTo` i `Equals`. Interfejs jest typowany, więc obiekty przekazywane do tych metod będą typu `Pilgrim`:

Zakres typów właściwych dla typu generycznego można ograniczać.

```
public int CompareTo(Pilgrim rhs)
public bool Equals(Pilgrim rhs)
```

Teraz wystarczy już zmienić logikę kodu odpowiedzialnego za dodawanie nowych węzłów do listy. Tym razem zamiast szukać ostatniego węzła listy, należałoby szukać odpowiedniego miejsca pomiędzy istniejącymi węzłami — takiego, żeby po wstawieniu nowego węzła zachowane zostało uporządkowanie listy; miejsce to wyszukuje się na bazie implementacji `CompareTo`.

Przede wszystkim typ przechowywany w węźle musi implementować interfejs `IComparable`. Osiąga się to przy użyciu słowa kluczowego `where`:

```
public class Node<T> : IComparable<Node<T>> where T:IComparable<T>
```

Powyższy wiersz kodu deklaruje klasę węzła `Node` typu `T` implementującą interfejs `IComparable` (dla węzłów `Node` typu `T`) i ograniczoną do przechowywania w węzłach takich typów danych `T`, które implementują interfejs `IComparable`. Jeśli w węźle spróbujemy umieścić obiekt innego typu niż implementujący `IComparable`, próba kompilacji kodu sprowokuje błąd.

Przy samym wstawianiu węzła do listy trzeba rozważyć i obsłużyć przypadek szczególny, kiedy to nowy węzeł będzie „mniejszy" od węzła stanowiącego czoło listy. Widać to na listingu 1.3 (różnice w stosunku do poprzedniej implementacji kolekcji zostały wyróżnione pogrubieniem).

Listing 1.3. Implementacja interfejsów generycznych

```
using System;
using System.Collections.Generic;
```

```
namespace ImplementingGenericInterfaces
{
  public class Pilgrim : IComparable<Pilgrim>
  {
      private string name;
      public Pilgrim(string name)
      {
          this.name = name;
      }
      public override string ToString()
      {
          return this.name;
      }

      // implementacja interfejsu
      public int CompareTo(Pilgrim rhs)
      {
          return this.name.CompareTo(rhs.name);
      }
      public bool Equals(Pilgrim rhs)
      {
          return this.name == rhs.name;
      }
  }


  // węzeł musi implementować interfejs IComparable dla węzłów Node typu T
  // węzeł może przechowywać jedynie te typy T, które implementują
     IComparable
  // (warunek określany słowem kluczowym where).
  public class Node<T> : IComparable<Node<T>> where T:IComparable<T>
  {
      // pola składowe
      private T data;
      private Node<T> next = null;
      private Node<T> prev = null;

      // konstruktor
      public Node(T data)
      {
          this.data = data;
      }

      // właściwości
      public T Data { get { return this.data; } }

      public Node<T> Next
      {
          get { return this.next; }
      }

      public int CompareTo(Node<T> rhs)
      {
```

```
        // działa z racji ograniczenia (where T:IComparable<T>)
        return data.CompareTo(rhs.data);
    }

    public bool Equals(Node<T> rhs)
    {
        return this.data.Equals(rhs.data);
    }

    // metody
    public Node<T> Add(Node<T> newNode)
    {
        if (this.CompareTo(newNode) > 0) // wstawienie przed węzeł
                                         // bieżący
        {
            newNode.next = this;  // wskaźnik next ustawiany na węzeł
                                  // bieżący

            // jeśli istnieje węzeł poprzedni, powinien od tego momentu
            // wskazywać polem next nowy węzeł
            if (this.prev != null)
            {
                this.prev.next = newNode;
                newNode.prev = this.prev;
            }

            // wskaźnik poprzednika węzła bieżącego ma wskazywać nowy
            // węzeł
            this.prev = newNode;

            // zwrócenie referencji nowego węzła, jeśli stał się nowym
            // czołem listy
            return newNode;
        }
        else            // wstawienie za węzeł bieżący
        {
            // jeśli bieżący nie jest ostatnim, całą operację przejmuje
            // następny
            if (this.next != null)
            {
                this.next.Add(newNode);
            }

            // brak następnego węzła – nowy węzeł trzeba skojarzyć
            // z polem next bieżącego;
            // a w polu prev nowego wstawić referencję do bieżącego
            else
            {
                this.next = newNode;
                newNode.prev = this;
            }

            return this;
```

```
        }
    }

    public override string ToString()
    {
        string output = data.ToString();

        if (next != null)
        {
            output += ", " + next.ToString();
        }

        return output;
    }
}       // koniec klasy


public class SortedLinkedList<T> where T : IComparable<T>
{
    // pola składowych
    private Node<T>  headNode = null;


    // właściwości

    // indekser
    public T this[int index]
    {
        get
        {
            int ctr = 0;
            Node<T> node = headNode;

            while (node != null && ctr <= index)
            {
                if (ctr == index)
                {
                    return node.Data;
                }
                else
                {
                    node = node.Next;
                }

                ++ctr;
            }  // koniec while
            throw new ArgumentOutOfRangeException();
        }  // koniec get
    }  // koniec indeksera


    // konstruktor
    public SortedLinkedList()
```

```
        {
        }

        // metody
        public void Add(T data)
        {
            if (headNode == null)
            {
                headNode = new Node<T>(data);
            }
            else
            {
                headNode = headNode.Add(new Node<T>(data));
            }
        }
        public override string ToString()
        {
            if (this.headNode != null)
            {
                return this.headNode.ToString();
            }
            else
            {
                return string.Empty;
            }
        }
    }

    class Program
    {
        // punkt wejścia
        static void Main(string[] args)
        {
            SortedLinkedList<int> mySortedLinkedList = new
            SortedLinkedList<int>();
            Random rand = new Random();
            Console.Write("Wypełnianie: ");

            for (int i = 0; i < 10; i++)
            {
                int nextInt = rand.Next(10);
                Console.Write("{0}  ", nextInt);
                mySortedLinkedList.Add(nextInt);
            }

            SortedLinkedList<Pilgrim> pilgrims = new
            SortedLinkedList<Pilgrim>();
            pilgrims.Add(new Pilgrim("Rycerz"));
            pilgrims.Add(new Pilgrim("Młynarz"));
            pilgrims.Add(new Pilgrim("Szeryf"));
            pilgrims.Add(new Pilgrim("Kucharz"));
            pilgrims.Add(new Pilgrim("Adwokat"));
```

```
            Console.WriteLine("\nPobieranie kolekcji...");

            DisplayList<int>("Liczby", mySortedLinkedList);
            DisplayList<Pilgrim>("Pielgrzymi", pilgrims);
            //Console.WriteLine("Liczby: " + mySortedLinkedList);
            //Console.WriteLine("Pielgrzymi: " + pilgrims);

            Console.WriteLine("Czwarta liczba to " + mySortedLinkedList[3]);
            Pilgrim d = pilgrims[2];
            Console.WriteLine("Trzeci pielgrzym to " + d);

//              foreach (Pilgrim p in pilgrims)
//              {
//                  Console.WriteLine("Zawód pielgrzyma to " +
                    p.ToString());
//              }
        }    // koniec metody Main

        private static void DisplayList<T>(string intro, SortedLinkedList<T>
        theList)
            where T : IComparable<T>
        {
            Console.WriteLine(intro + ": " + theList);
        }

    }  // koniec klasy
}  // koniec przestrzeni nazw
```

Wynik:

```
Wypełnianie: 2  8  2  5  1  7  2  8  5  5
Pobieranie kolekcji...
Liczby: 1, 2, 2, 2, 5, 5, 5, 7, 8, 8
Pielgrzymi: Adwokat, Kucharz, Młynarz, Rycerz, Szeryf
Czwarta liczba to 2
Trzeci pielgrzym to Młynarz
```

Jak to działa?

Klasa `Pilgrim` została uzupełniona o implementację interfejsu generycznego `IComparable`. Sama lista nie zmieniła się ani na jotę, ale już klasa węzłów listy (`Node`) przeszła poważną metamorfozę, dzięki której wstawianie węzłów do listy odbywa się z zachowaniem ich wzajemnego uporządkowania.

Po pierwsze, klasa `Node` została oznaczona jako klasa implementująca interfejs `IComparable` i ograniczona do przechowywania obiektów takich typów, które również implementują ów interfejs:

```
public class Node<T> : IComparable<Node<T>> where T:IComparable<T>
```

Po drugie, w węźle obok pola z referencją do następnego węzła pojawiło się pole referencji do węzła poprzedniego (czyniąc listę listą dwukierunkową):

```
private Node<T> next = null;
private Node<T> prev = null;
```

Klasa `Node` musi teraz implementować metody `CompareTo` i `Equals`. Są to proste metody, bo ich działanie sprowadza się do oddelegowania porównania do analogicznych metod obiektu przechowywanego — a wiadomo, że obiekty te również implementują interfejs `IComparable`:

```
public int CompareTo(Node<T> rhs)
{
    // działa z racji ograniczenia (where T:IComparable<T>)
    return data.CompareTo(rhs.data);
}
```

A co...

... z wymaganiem implementacji interfejsu `IComparable`? Dlaczego musiała go implementować klasa `Pilgrim` i `Node`, a już sama klasa kolekcji (tu `SortedLinkedList`) nie?

Aby to wyjaśnić, trzeba przypomnieć, że i `Pilgrim`, i `Node` to obiekty danych podlegające operacjom porównania; sama lista jako ogólniejsza struktura nie jest zaś nigdy porównywana z innymi listami. Uporządkowanie węzłów listy odbywa się przez ustalanie ich kolejności na bazie porównań; nigdzie nie zachodzi zaś porównanie dwóch list i sprawdzanie, która z nich jest „większa".

... z przekazywaniem typów generycznych do metod? Czy to możliwe?

Tak, przekazywanie typów generycznych do metod jest dozwolone, pod warunkiem że chodzi o metody generyczne. Przykładem może być poniższy kod zaczerpnięty z listingu 1.3, wyświetlający zawartość listy liczb i listy pielgrzymów:

```
Console.WriteLine("Liczby: " + myLinkedList);
Console.WriteLine("Pielgrzymi: " + pilgrims);
```

Nic nie stoi na przeszkodzie, aby utworzyć metodę przyjmującą za pośrednictwem argumentu taką listę i wyświetlającą jej elementy (albo w dowolny inny sposób manipulującą tą listą):

```
private static void DisplayList<T>(string intro, LinkedList<T> theList)
    where T : IComparable<T>
{
    Console.WriteLine(intro + ": " + theList);
}
```

W wywołaniu takiej metody należy uściślić typy:

```
DisplayList<int>("Liczby", myLinkedList);
DisplayList<Pilgrim>("Pielgrzymi", pilgrims);
```

WSKAZÓWKA

Kompilator ma możliwość wnioskowania o typie metody na podstawie typów argumentów, więc poprzednie dwa wywołania można zapisać również tak:

```
DisplayList("Liczby", myLinkedList);
DisplayList("Pielgrzymi", pilgrims);
```

Więcej informacji

Zawartość przestrzeni nazw `Generic` jest wyczerpująco omawiana w dokumentacji MSDN — wystarczy ją przeszukać pod kątem hasła „Systems.Collections.Generic". Polecam też artykuł traktujący o typach generycznych, publikowany w serwisie ONDotnet.com (O'Reilly) (*http://www.ondotnet.com/pub/a/dotnet/2004/05/17/liberty.html*).

Stosowanie iteratorów generycznych

W poprzednio prezentowanych przykładach nie dało się przeglądać listy `Pilgrims` w pętli `foreach`. Gdyby w programie z listingu 1.3 umieścić poniższy kod:

```
foreach (Pilgrim p in pilgrims)
{
    Console.WriteLine("Zawód pielgrzyma to " + p.ToString());
}
```

próba kompilacji programu doprowadziłaby do zgłoszenia następującego błędu:

```
Error    1    foreach statement cannot operate on variables of type
'ImplementingGenericInterfaces.LinkedList <ImplementingGenericInterfaces.
Pilgrim>' because 'ImplementingGenericInterfaces.LinkedList
```

Dodawanie iteratorów pozwala klientom przeglądać kolekcje w pętlach foreach.

```
<ImplementingGenericInterfaces.Pilgrim>' does not contain a public
definition for 'GetEnumerator'
```

W poprzednich wersjach C# implementowanie metody `GetEnumerator` było dość uciążliwe i skomplikowane; w C# 2.0 cały zabieg został znacznie uproszczony.

Jak to zrobić?

Aby uprościć sobie tworzenie iteratorów, należałoby przede wszystkim uprościć klasy `Pilgrim` i `LinkedList`. Klasa kolekcji `LinkedList` powinna też zaniechać stosowania węzłów i przechowywać elementy listy w tablicy o stałym rozmiarze (taka tablica to najprostszy możliwy typowany kontener-kolekcja). Kolekcja zostanie więc listą jedynie z nazwy! To pozwoli się jednak skupić na implementacji interfejsu `IEnumerable`, prezentowanej na listingu 1.4.

Listing 1.4. Implementacja interfejsu IEnumerable (wersja uproszczona)

```
#region Using directives

using System;
using System.Collections.Generic;
using System.Text;

#endregion

namespace SimplifiedEnumerator
{
  // uproszczona wersja klasy Pilgrim
  public class Pilgrim
  {
      private string name;
      public Pilgrim(string name)
      {
          this.name = name;
      }
      public override string ToString()
      {
          return this.name;
      }

  }

  //  uproszczona wersja klasy listy
  class NotReallyALinkedList<T> : IEnumerable<T>
  {
```

```
        // wszystkie elementy listy są przechowywane w tablicy
        // o stałym rozmiarze
        T[] myArray;

        // konstruktor przyjmuje tablicę i umieszcza jej elementy we własnej
        // tablicy
        public NotReallyALinkedList(T[] members)
        {
            myArray = members;
        }

        // implementacja głównej metody interfejsu IEnumerable
        IEnumerator<T> IEnumerable<T>.GetEnumerator()
        {
            foreach (T t in this.myArray)
            {
                yield return t;
            }
        }

        // wymagana implementacja również dla wersji niegenerycznej
        System.Collections.IEnumerator System.Collections.IEnumerable.
        GetEnumerator()
        {
           throw new NotImplementedException();
        }
    }


    class Program
    {
        static void Main(string[] args)
        {
            // ręczne tworzenie tablicy obiektów klasy Pilgrim
            Pilgrim[] pilgrims = new Pilgrim[5];
            pilgrims[0] = new Pilgrim("Rycerz");
            pilgrims[1] = new Pilgrim("Młynarz");
            pilgrims[2] = new Pilgrim("Szeryf");
            pilgrims[3] = new Pilgrim("Kucharz");
            pilgrims[4] = new Pilgrim("Adwokat");

            // utworzenie listy, przekazanie tablicy elementów
            NotReallyALinkedList<Pilgrim> pilgrimCollection =
                new NotReallyALinkedList<Pilgrim>(pilgrims);

            // przeglądanie elementów listy
            foreach (Pilgrim p in pilgrimCollection)
            {
                Console.WriteLine(p);
            }
        }
    }
}
```

Wynik:

```
Rycerz
Młynarz
Szeryf
Kucharz
Adwokat
```

Jak to działa?

W tym przykładzie lista została znacznie uproszczona — elementy listy zamiast w przydzielanych dynamicznie węzłach lądują w statycznej tablicy (co w zasadzie dyskwalifikuje tę implementację jako listę). Ponieważ owa pseudo-lista daje się jednak przeglądać z użyciem iteratorów, przechowywana w niej kolekcja obiektów `Pilgrim` daje się przeglądać w pętli `foreach`.

W obliczu zapisu:

```
foreach (Pilgrim p in pilgrimCollection)
```

kompilator języka C# wywołuje na rzecz obiektu kolekcji metodę `GetEnumerator`. Rozwinięcie takiej pętli jest wewnętrznie implementowane mniej więcej tak:

Każde zastosowanie pętli foreach jest przez kompilator tłumaczone na wywołanie metody GetEnumerator.

```
Enumerator e = pilgrimCollection.GetEnumerator();
while (e.MoveNext())
{
    Pilgrim p = e.Current;
}
```

Jak już powiedziano, w C# 2.0 nie trzeba się zajmować implementacją metody `MoveNext()` czy właściwością `Current`. Trzeba jedynie zastosować nowe słowo kluczowe C# — `yield`:

WSKAZÓWKA

Słowo kluczowe `yield` można stosować jedynie w blokach iteracji. Słowo to albo odnosi się do wartości obiektu enumeratora, albo sygnalizuje koniec iteracji:

```
yield return wyrażenie;
yield break;
```

Gdyby wejść do wnętrza pętli `foreach` za pomocą debugera, okazałoby się, że za każdym razem następuje tam wywołanie na rzecz obiektu kolekcji

metody `GetEnumerator`; we wnętrzu tej metody następuje zaś wielokrotne zwracanie wartości do pętli `foreach` za pośrednictwem kolejnych wystąpień słowa kluczowego `yield`[1].

A co…

… z implementacją metody `GetEnumerator` dla bardziej złożonych struktur danych, na przykład dla prawdziwej listy?

To właśnie będzie przedmiotem następnego ćwiczenia.

Więcej informacji

Poruszone zagadnienie jest wyczerpująco omawiane w obszernym artykule z biblioteki MSDN, zatytułowanym „Iterators (C#)".

Implementacja GetEnumerator dla złożonych struktur danych

Aby dać pierwotnej liście `LinkedList` możliwość przeglądania elementów za pośrednictwem iteratora, trzeba zaimplementować interfejs `IEnumerable<T>` dla samej klasy listy (`LinkedList`) i klasy węzłów listy (`Node`).

```
public class LinkedList<T> : IEnumerable<T>
public class Node<T> : IComparable<Node<T>>, IEnumerable<Node<T>>
```

Jak to zrobić?

Przy okazji poprzedniego ćwiczenia okazało się, że interfejs `IEnumerable` wymaga zasadniczo implementacji tylko jednej metody — `GetEnumerator`. Implementację tę dla bardziej złożonej struktury danych, jaką jest choćby lista, prezentuje listing 1.5 (zmiany względem kodu z listingu 1.3 zostały wyróżnione pogrubieniem).

[1] Słowo `yield` działa jak instrukcja `return` uzupełniona o pamięć stanu — służy do iteracyjnego zwracania kolejnych elementów sekwencji — *przyp. tłum.*

Listing 1.5. Implementacja iteratora dla klasy listy

```
using System;
using System.Collections.Generic;

namespace GenericEnumeration
{
  public class Pilgrim : IComparable<Pilgrim>
  {
      private string name;
      public Pilgrim(string name)
      {
          this.name = name;
      }
      public override string ToString()
      {
          return this.name;
      }

      // implementacja interfejsu IComparable
      public int CompareTo(Pilgrim rhs)
      {
          return this.name.CompareTo(rhs.name);
      }
      public bool Equals(Pilgrim rhs)
      {
          return this.name == rhs.name;
      }
  }


  // węzeł musi być implementować interfejs IComparable dla węzłów Node
  // typu T teraz dodatkowo implementuje interfejs IEnumerable do użytku
  // w pętlach foreach
  public class Node<T> : IComparable<Node<T>>, IEnumerable<Node<T>> where
  T:IComparable<T>
  {
      // pola składowe
      private T data;
      private Node<T> next = null;
      private Node<T> prev = null;

      // konstruktor
      public Node(T data)
      {
          this.data = data;
      }

      // właściwości
      public T Data { get { return this.data; } }

      public Node<T> Next
      {
          get { return this.next; }
```

```
}

public int CompareTo(Node<T> rhs)
{
    return data.CompareTo(rhs.data);
}
public bool Equals(Node<T> rhs)
{
    return this.data.Equals(rhs.data);
}

// metody
public Node<T> Add(Node<T> newNode)
{
    if (this.CompareTo(newNode) > 0) // wstawienie przed węzeł
                                     // bieżący
    {
        newNode.next = this;  // wskaźnik next ustawiany na węzeł

        // bieżący jeśli istnieje węzeł poprzedni, powinien od tego
        // momentu wskazywać polem next nowy węzeł
        if (this.prev != null)
        {
            this.prev.next = newNode;
            newNode.prev = this.prev;
        }

        // wskaźnik poprzednika węzła bieżącego ma wskazywać nowy
        // węzeł
        this.prev = newNode;

        // zwrócenie referencji nowego węzła, jeśli stał się nowym
        // czołem listy
        return newNode;
    }
    else            // wstawienie za węzeł bieżący
    {
        // jeśli bieżący nie jest ostatnim, całą operację przejmuje
        // następny
        if (this.next != null)
        {
            this.next.Add(newNode);
        }

        // brak następnego węzła – nowy węzeł trzeba skojarzyć
        // z polem next bieżącego;
        // a w polu prev nowego wstawić referencję do bieżącego
        else
        {
            this.next = newNode;
            newNode.prev = this;
        }
```

```
            return this;
        }
    }

    public override string ToString()
    {
        string output = data.ToString();

        if (next != null)
        {
            output += ", " + next.ToString();
        }

        return output;
    }


    // Metody wymagane przez IEnumerable
    IEnumerator<Node<T>> IEnumerable<Node<T>>.GetEnumerator()
    {

        Node<T> nextNode = this;

        // przeglądanie wszystkich węzłów listy,
        // zwracanie (yield) kolejnych węzłów
        do
        {
            Node<T> returnNode = nextNode;
            nextNode = nextNode.next;
            yield return returnNode;
        } while (nextNode != null);
    }

    System.Collections.IEnumerator System.Collections.IEnumerable.
    GetEnumerator()
    {
       throw new NotImplementedException();
    }


} // koniec klasy


// implementacja IEnumerable pozwalająca na stosowanie
// klasy LinkedList w pętlach foreach
public class LinkedList<T> : IEnumerable<T> where T : IComparable<T>
{
    // pola składowych
    private Node<T> headNode = null;


    // właściwości
    // indekser
```

```
public T this[int index]
{
    get
    {
        int ctr = 0;
        Node<T> node = headNode;

        while (node != null && ctr <= index)
        {
            if (ctr == index)
            {
                return node.Data;
            }
            else
            {
                node = node.Next;
            }

            ++ctr;
        }  // koniec while
        throw new ArgumentOutOfRangeException();
    }  // koniec get
}  // koniec indeksera

// konstruktor
public LinkedList()
{
}

// metody
public void Add(T data)
{
    if (headNode == null)
    {
        headNode = new Node<T>(data);
    }
    else
    {
        headNode = headNode.Add(new Node<T>(data));
    }
}
public override string ToString()
{
    if (this.headNode != null)
    {
        return this.headNode.ToString();
    }
    else
    {
        return string.Empty;
    }
}
```

```
        // Implementacja wymaganej metody IEnumerable
        // przeglądająca węzły (również implementujące ten interfejs)
        // i zwracająca (yield) dane zwrócone z węzła
        IEnumerator<T> IEnumerable<T>.GetEnumerator()
        {
            foreach (Node<T> node in this.headNode)
            {
                yield return node.Data;
            }
        }

        System.Collections.IEnumerator System.Collections.IEnumerable.
        GetEnumerator()
        {
           throw new NotImplementedException();
        }

    }


    class Program
    {
        private static void DisplayList<T>(string intro, LinkedList<T>
        theList) where T : IComparable<T>
        {
            Console.WriteLine(intro + ": " + theList);
        }

        // punkt wejścia
        static void Main(string[] args)
        {
            LinkedList<Pilgrim> pilgrims = new LinkedList<Pilgrim>();
            pilgrims.Add(new Pilgrim("Rycerz"));
            pilgrims.Add(new Pilgrim("Młynarz"));
            pilgrims.Add(new Pilgrim("Szeryf"));
            pilgrims.Add(new Pilgrim("Kucharz"));
            pilgrims.Add(new Pilgrim("Adwokat"));

            DisplayList<Pilgrim>("Pilgrims", pilgrims);

            Console.WriteLine("Przeglądanie listy pielgrzymów...");

            // teraz lista daje się przeglądać, więc można ją
            // zastosować w pętli foreach
            foreach (Pilgrim p in pilgrims)
            {
                Console.WriteLine("Zawód pielgrzyma to " + p.ToString());
            }
        }
    }
}
```

Wynik:

```
Pielgrzymi: Adwokat, Kucharz, Młynarz, Rycerz, Szeryf
Przeglądanie listy pielgrzymów...
Zawód pielgrzyma to Adwokat
Zawód pielgrzyma to Kucharz
Zawód pielgrzyma to Młynarz
Zawód pielgrzyma to Rycerz
Zawód pielgrzyma to Szeryf
```

Jak to działa?

Lista implementuje teraz enumerator; implementacja polega na zainicjowaniu pętli `foreach` dla czołowego węzła listy (klasa węzła również implementuje interfejs `IEnumerable`). Implementacja zwraca obiekt danych zwrócony przez węzeł:

```
IEnumerator<T> IEnumerable<T>.GetEnumerator()
{
      foreach (Node<T> node in this.headNode)
      {
          yield return node.Data;
      }
}
```

Odpowiedzialność za realizację iteracji spada tym samym na klasę `Node`, która we własnej implementacji metody `GetEnumerator` również posługuje się słowem kluczowym `yield`.

W miejsce instrukcji yield kompilator automatycznie generuje zagnieżdżoną implementację IEnumerator. Zapamiętuje tam stan iteracji; programista musi jedynie wskazać wartości do zwrócenia w kolejnych krokach iteracji.

```
IEnumerator<Node<T>> IEnumerable<Node<T>>.GetEnumerator()
{
    Node<T> nextNode = this;
    do
    {
        Node<T> returnNode = nextNode;
        nextNode = nextNode.next;
        yield return returnNode;
    } while (nextNode != null);
}
```

Obiekt `nextNode` jest inicjalizowany referencją do węzła bieżącego, po czym następuje rozpoczęcie pętli `do...while`. Pętla taka zostanie wykonana przynajmniej jednokrotnie. W pętli następuje przepisanie wartości `nextNode` do `returnNode` i próba odwołania się do następnego węzła listy (wskazywanego polem `next`). Następna instrukcja pętli zwraca do wywołującego (za pomocą słowa `yield`) węzeł zapamiętany przed chwilą w `returnNode`. Kiedy w następnym kroku iteracji nastąpi powrót do pętli, zostanie ona

wznowiona od tego miejsca; całość będzie powtarzana dopóty, dopóki pole `next` któregoś z kolejnych węzłów nie okaże się puste, a tym samym węzeł będzie ostatnim węzłem listy.

A co...

... znaczy występujące w implementacji `LinkedList` żądanie przejrzenia (`foreach`) elementów `Node<T>` w `headNode`? Przecież `headNode` to nie lista, a jeden z jej węzłów (konkretnie węzeł czołowy)?

Otóż `headNode` to faktycznie czołowy węzeł listy. Ponieważ jednak klasa `Node` implementuje interfejs `IEnumerable`, dla potrzeb iteracji węzeł zachowuje się jak kolekcja. Choć brzmi to niedorzecznie, jest całkiem uzasadnione, bo węzeł w istocie przejawia pewne cechy kolekcji, w tym przynajmniej sensie, że potrafi wskazać następny element kolekcji (następny węzeł listy). Całość można by przeprojektować tak, żeby węzły nie były tak „sprytne", za to sama lista była „sprytniejsza" — wtedy zadanie realizacji iteracji spoczywałoby w całości na liście i ta nie delegowałaby zadania do węzłów.

Więcej informacji

O interfejsie `IEnumerable<T>` można się sporo dowiedzieć z plików pomocy MSDN dla hasła „Topic: IEnumerable<T>".

Upraszczanie kodu — metody anonimowe

Metody anonimowe pozwalają na definiowanie nienazwanych bloków kodu rozwijanych w miejscu wywołania. Z metod anonimowych można korzystać wszędzie tam, gdzie dozwolone są delegacje. Za ich pośrednictwem można na przykład znakomicie uprościć rejestrowanie procedur obsługi zdarzeń.

Jak to zrobić?

Zastosowania metod anonimowych najlepiej zilustrować przykładem:

1. Utworzyć w Visual Studio .NET 2005 nową aplikację okienkową i nadać jej nazwę `AnonymousMethods`.

2. Przeciągnąć na domyślny formularz okna dwie kontrolki: etykietę oraz przycisk (nie warto zajmować się przydzielaniem im specjalnych nazw).

3. Dwukrotnie kliknąć przycisk lewym klawiszem myszy. Wyświetlone zostanie okno edytora, w którym należy umieścić poniższy kod:

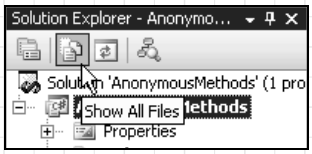
Metody anonimowe pozwalają na stosowanie bloków kodu w roli parametrów.

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

4. Uruchomić aplikację. Kliknięcie przycisku powinno zmieniać treść etykiety na `Do widzenia!`.

Wszystko świetnie. Ale jest tu pewien ukrywany przed programistą narzut. Otóż powyższy kod wymaga rejestrowania delegacji (wyręcza nas w tym stosowny kreator Visual Studio 2005), a obsługa kliknięcia przycisku wymaga zdefiniowania nowej metody. Całość można zaś uprościć stosując metody anonimowe.

Aby sprawdzić, jak faktycznie rejestrowana jest metoda obsługi zdarzenia kliknięcia przycisku, należy kliknąć w IDE przycisk *Show All Files* (prezentowany na rysunku 1.1).



Rysunek 1.1. Przycisk Show All Files

Teraz należałoby otworzyć plik *Form1.Designer.cs* i odszukać w nim delegację `button1.Click`:

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

Nie powinno się ręcznie modyfikować tego kodu, ale można wyeliminować ten wiersz inaczej — wracając do formularza i klikając w oknie właściwości (*Properties*) ikonę błyskawicy, wywołującą procedury obsługi zdarzeń. Tam można usunąć procedurę obsługi zarejestrowaną dla zdarzenia `Click`.

Po powrocie do kodu *Form1.Designer.cs* okaże się, że procedura obsługi zdarzenia `button1.Click` nie jest w ogóle zarejestrowana!

Teraz należy otworzyć do edycji plik *Form1.cs* i dodać do konstruktora (za wywołaniem `InitializeComponent()`) poniższy wiersz:

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

Dzięki temu można już pozbyć się dodatkowej metody procedury obsługi zdarzenia — można ją usunąć albo oznaczyć jako komentarz:

```
// private void button1_Click(object sender, EventArgs e)
// {
//     label1.Text = "Do widzenia!";
// }
```

Działanie metody anonimowej można sprawdzić, ponownie uruchamiając aplikację. Powinna zachowywać się dokładnie tak, jak poprzednio.

Jak widać, zamiast rejestrować delegację wywołującą metodę obsługi zdarzenia, można wskazać *metodę anonimową* — nienazwany, rozwijany w miejscu wywołania blok kodu.

A co...

... z innymi zastosowaniami metod anonimowych? Czy można je stosować we własnym kodzie?

Żaden problem. Metody anonimowe można stosować nie tylko przy inicjalizowaniu delegacji, ale i *wszędzie* tam, gdzie dozwolone jest użycie delegacji — we wszystkich tych miejscach można przekazać nienazwany blok kodu.

... jeśli w takim bloku kodu nastąpi odwołanie do zmiennej lokalnej?

Dobre pytanie. To dość myląca sytuacja i łatwo tu o pomyłkę, zwłaszcza kiedy nie jest się w pełni świadomym konsekwencji takich odwołań. Otóż C# pozwala na wciąganie zmiennych lokalnych do zasięgu anonimowego bloku kodu; odwołania do nich są wykonywane w momencie wykonania owego bloku kodu. Może to prowokować rozmaite efekty uboczne — choćby podtrzymywanie przy życiu obiektów, którymi inaczej już dawno zaopiekowałby się mechanizm zbierania nieużytków.

... z usuwaniem procedury obsługi dla zdarzenia, dodanej za pomocą delegacji anonimowej; da się to zrobić?

Jeśli procedura obsługi zdarzenia została określona delegacją anonimową, nie można jej usunąć; dlatego delegacje anonimowe powinno się stosować jedynie dla tych procedur obsługi, które mają być trwale skojarzone z danymi zdarzeniami.

Ponadto *można* stosować delegacje anonimowe również w innych dziedzinach, choćby przy implementowaniu metody `List.Find` przyjmującej delegację opisującą kryteria wyszukiwania.

Więcej informacji

W zasobach MSDN można znaleźć świetny artykuł traktujący o metodach anonimowych. Mowa o artykule „Create Elegant Code with Anonymous Methods, Iterators and Partial Classes" autorstwa Juvala Lowy'ego. Warto też zapoznać się z artykułem z serwisu ONDotnet.com (O'Reilly), publikowanym pod adresem *http://www.ondotnet.com/pub/a/dotnet/2004/04/05/csharpwhidbeypt1.html*.

Ukrywanie kodu — typy częściowe

Słowo kluczowe partial pozwala na podział definicji klasy na wiele plików.

W poprzednich wersjach C# całość definicji klasy musiała być umieszczana w pojedynczym pliku. Teraz dzięki słowu kluczowemu `partial` można dzielić klasę na części przechowywane w większej liczbie plików. Możliwość ta jest cenna z dwóch względów:

- W zespole programistycznym można przeprowadzić podział polegający na przypisaniu różnych programistów do prac nad różnymi częściami klasy.
- Visual Studio 2005 może w ten sposób oddzielać kod generowany automatycznie od kodu własnego programisty.

Jak to zrobić?

Praktyczne zastosowanie typów częściowych można zilustrować na bazie poprzedniego przykładu (AnonymousMethods). Spójrzmy na deklarację klasy w pliku *Form1.cs*:

```
partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
        this.button1.Click += delegate { label1.Text = "Do widzenia!"; };

    }

//    private void button1_Click(object sender, EventArgs e)
//    {
//        label1.Text = "Do widzenia!";
//    }
}
```

Słowo kluczowe `partial` sygnalizuje, że kod zamieszczony w tym pliku niekoniecznie reprezentuje całość definicji klasy. Co zresztą zgadza się z naszą wiedzą, bo przecież w poprzednim podrozdziale zaglądaliśmy do drugiego pliku *Form1.Designer.cs* zawierającego resztę definicji klasy:

```
namespace AnonymousMethods
{
  partial class Form1
  {
      /// <summary>
      /// Required designer variable.
      /// </summary>
      private System.ComponentModel.IContainer components = null;

      /// <summary>
      /// Clean up any resources being used.
      /// </summary>
      protected override void Dispose(bool disposing)
      {
          if (disposing && (components != null))
          {
              components.Dispose();
          }
          base.Dispose(disposing);
      }

      #region Windows Form Designer generated code
      ...
      #endregion
```

```
        private System.Windows.Forms.Label label1;
        private System.Windows.Forms.Button button1;
    }
}
```

Kompletną definicję klasy `Form1` dają dopiero te dwa pliki wzięte razem; podział klasy pozwala na wyodrębnienie kodu tworzonego przez programistę i kodu generowanego automatycznie przez różne mechanizmy środowiska programistycznego. Czyni to projekt przejrzystszym i prostszym.

Stosując klasy częściowe, trzeba mieć świadomość kilku aspektów:

- Wszystkie częściowe definicje typów muszą zawierać słowo kluczowe `partial` i muszą należeć do tej samej przestrzeni nazw oraz tego samego modułu i podzespołu.
- Modyfikator `partial` może występować jedynie przed słowami kluczowymi `class`, `inerface` i `struct`.
- We wszystkich definicjach częściowych należy uzgodnić modyfikatory dostępu do składowych (`public`, `private` itd.).

A co…

… ze stosowaniem klas częściowych we własnych projektach?

Microsoft sugeruje, że klasy częściowe mogą przydać się programistom pracującym w zespołach — mogą wtedy podzielić się pracą nad klasami. Wciąż jednak za wcześnie, aby stwierdzić, czy taka praktyka się przyjmie; osobiście uważam, że każda klasa tak rozbudowana, aby jej rozmiar uzasadniał podział pracy, powinna po prostu zostać podzielona na mniejsze klasy. Na razie więc głównym zastosowaniem typów częściowych jest upychanie po kątach rozmaitych tworów generowanych przez kreatory środowiska programistycznego Visual Studio 2005.

Więcej informacji

Dobry artykuł o typach częściowych można znaleźć w archiwach witryny Developer.com. Publikowany jest pod adresem *http://www.developer.com/net/net/article.php/2232061*.

Tworzenie klas statycznych

W C# 2.0 klasę można zadeklarować jako statyczną, sygnalizując, że ma ona służyć jedynie w roli zasobnika zestawu statycznych metod narzędziowych.

W nowej wersji języka C# jako statyczne można deklarować nie tylko metody, ale również całe klasy.

Zadaniem klasy statycznej jest udostępnianie zestawu statycznych metod pomocniczych ujętych w zasięgu nazwy klasy — jak w klasie `Convert` z biblioteki Framework Class Library.

Jak to zrobić?

Utworzenie klasy statycznej polega na poprzedzeniu nazwy klasy słowem kluczowym `static` i upewnieniu się, że sama definicja klasy spełnia podane wyżej kryterium co do metod. Trzeba też pamiętać o dodatkowych ograniczeniach nakładanych na klasy statyczne:

- Klasy statyczne mogą zawierać wyłącznie statyczne składowe.
- Nie wolno tworzyć egzemplarza klasy statycznej.
- Wszystkie klasy statyczne są klasami finalnymi (bezpłodnymi) — nie można wyprowadzać z nich klas pochodnych.

Oprócz tego klasa statyczna nie może zawierać konstruktora. Właściwe zastosowanie klasy statycznej ilustruje kod z listingu 1.6.

Listing 1.6. Stosowanie klas statycznych

```
#region Using directives

using System;

#endregion

namespace StaticClass
{

  public static class CupConversions
  {
      public static int CupToOz(int cups)
      {
          return cups * 8; // szklanka to 8 uncji płynu
      }
      public static double CupToPint(double cups)
      {
          return cups * 0.5;  // szklanka to pół pinty
```

```
        }

        public static double CupToMil(double cups)
        {
            return cups * 237; // 237 mililitrów to jedna szklanka
        }

        public static double CupToPeck(double cups)
        {
            return cups / 32; // 8 kwart to 1 peck
        }

        public static double CupToBushel(double cups)
        {
            return cups / 128; // 4 pecki to 1 buszel
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Nie każdy wie, że " +
                 "szklanka płynu da się przeliczyć na: ");
            Console.WriteLine(CupConversions.CupToOz(1) + " uncji");
            Console.WriteLine(CupConversions.CupToPint(1) + " pint");
            Console.WriteLine(CupConversions.CupToMil(1) + " mililitrów");
            Console.WriteLine(CupConversions.CupToPeck(1) + " pecków");
            Console.WriteLine(CupConversions.CupToBushel(1) + " buszli");
        }
    }
}
```

Wynik:

```
Nie każdy wie, że szklanka płynu da się przeliczyć na:
8 uncji
0.5 pint
237 mililitrów
0.03125 pecków
0.0078125 buszli
```

Główna metoda klasy `Program` wywołuje statyczne metody klasy `CupConversions`. Ponieważ klasa ta istnieje tylko jako zasobnik metod narzędziowych (pomocniczych), a obiekt klasy nie jest wcale potrzebny, klasa `CupConversion` mogła zostać zdefiniowana jako statyczna.

A co...

... z polami i właściwościami? Czy klasa statyczna może mieć takie składowe?

Owszem, może, ale wszelkie składowe (metody, pola i właściwości) powinny być również statyczne.

Więcej informacji

Klasy statyczne zostały omówione między innymi w znakomitym artykule Erica Gunnersona. Artykuł jest dostępny w zasobach MSDN pod adresem *http://blogs.msdn.com/ericgu/archive/2004/04/13/112274.aspx*.

Wyrażanie wartości pustych typami nullable

Typy nullable pozwalają na wyróżnienie wartości pustych również w takich typach prostych, jak typy całkowitoliczbowe czy typy logiczne.

Nowe typy, tzw. typy nullable, to takie typy proste, którym można przypisywać wartości puste i wartości te da się potem odróżnić od wartości z właściwej dziedziny typu. Możliwość ta okazuje się niezwykle użyteczna, zwłaszcza przy pracy z bazami danych, kiedy to zwracana wartość pola może być wartością pustą; bez możliwości przepisania takiego pola do typu nullable nie można by stwierdzić, czy pole reprezentuje wartość pustą, czy może zero (a to różnica!); nie można by też wyrazić wartości logicznej, która nie reprezentuje jeszcze ani „prawdy", ani „fałszu".

Jak to zrobić?

Typ mogący przyjmować wartości puste deklaruje się następująco:

```
System.Nullable<T> zmienna
```

A w obrębie zasięgu typu czy metody generycznej można stosować zapis:

```
T? zmienna
```

Dwie zmienne przechowujące wartości całkowite z wyróżnioną wartością pustą można więc zadeklarować tak:

```
System.Nullable<int> myNullableInt;
int? myOtherNullableInt;
```

Wartość pustą zmiennej typu nullable wykrywa się dwojako. Można zastosować konstrukcję taką:

```
if (myNullableInt.HasValue)
```

albo taką:

```
if (myNullableInt != null)
```

Obie zwrócą `true`, jeśli zmienna `myNullableInt` będzie zawierała jakąś wartość, bądź `false`, kiedy zmienna będzie pusta. Zastosowanie typów nullable ilustruje listing 1.7.

Listing 1.7. Typy nullable

```
using System;

namespace NullableTypes
{
  public class Dog
  {
      private int age;
      public Dog(int age)
      {
          this.age = age;
      }
  }

  class Program
  {
      static void Main(string[] args)
      {
          int? myNullableInt = 25;
          double? myNullableDouble = 3.14159;
          bool? myNullableBool = null; // ani tak, ani nie

          // string? myNullableString = "Ahoj"; // niedozwolone
          // Dog? myNullableDog = new Dog(3);  // niedozwolone

          if (myNullableInt.HasValue)
          {
              Console.WriteLine("myNullableInt to " +
              myNullableInt.Value);
          }
          else
          {
              Console.WriteLine("myNullableInt ma wartość pustą!");
          }

          if (myNullableDouble != null)
          {
              Console.WriteLine("myNullableDouble: " + myNullableDouble);
          }
          else
          {
              Console.WriteLine("myNullableDouble ma wartość pustą!");
          }
```

```
            if ( myNullableBool != null )
            {
                Console.WriteLine("myNullableBool: " + myNullableBool);
            }
            else
            {
                Console.WriteLine("myNullableBool ma wartość pustą!");
            }


            myNullableInt = null;     // przypisanie wartości pustej
                                      // do zmiennej całkowitej
            // int a = myNullableInt; // błąd kompilacji

            int b;
            try
            {
                b = (int)myNullableInt;  // sprowokuje wyjątek,
                                         // kiedy x będzie puste
                Console.WriteLine("b: " + b);
            }
            catch (System.Exception e)
            {
                Console.WriteLine("Wyjątek! " + e.Message);
            }

            int c = myNullableInt ?? -1;  // przypisze -1, kiedy x będzie
                                          // puste

            Console.WriteLine("c: {0}", c);

            // ostrożnie z założeniami jeśli którykolwiek z operandów
            // będzie pusty, wszelkie porównania dadzą wynik false!
            if (myNullableInt >= c)
            {
                Console.WriteLine("myNullableInt jest większe (równe) od c");
            }
            else
            {
                Console.WriteLine("Czy myNullableInt jest mniejsze od c?");
            }

        }
    }
}
```

Wynik:

```
myNullableInt to 25
myNullableDouble: 3.14159
myNullableBool ma wartość pustą!
```

```
Wyjątek! Nullable object must have a value.
C: -1
Czy myNullableInt jest mniejsze od c?
```

Jak to działa?

Skupmy się na metodzie `Main`. Następuje tu utworzenie pięciu zmiennych typów prostych, z wyróżnieniem wartości pustych:

```
int? myNullableInt = 25;
double? myNullableDouble = 3.14159;
bool? myNullableBool = null; // ani tak, ani nie

// string? myNullableString = "Ahoj";
// Dog? myNullableDog = new Dog(3);
```

Pierwsze trzy deklaracje są jak najbardziej poprawne, nie da się jednak utworzyć ciągu mającego cechę nullable ani nadać tej cechy klasie (typowi definiowanemu przez użytkownika); aby kod dał się skompilować, trzeba było oznaczyć go jako komentarz.

Wyjątkiem są struktury, które — choć są typami definiowanymi przez użytkownika — mogą być wykorzystywane jako typy nullable.

Dalej następuje sprawdzenie wartości każdej ze zmiennych, a konkretnie — sprawdzenie, czy zmiennym tym zostały nadane właściwe, niepuste wartości (co powoduje ustawienie właściwości `HasValue` na `true`). Jeśli tak, można te wartości wypisać (albo wprost, albo odwołując się do właściwości `Value`).

Potem do zmiennej `myNullableInt` przypisywana jest wartość pusta (`null`):

```
myNullableInt = null;
```

W następnym wierszu miała nastąpić próba zadeklarowania zmiennej typu `int` i przypisanie do niej wartości `myNullableInt`, ale okazuje się to niemożliwe. Takie przypisanie nie jest dozwolone, bo nie istnieje możliwość niejawnej konwersji typu `int` nullable do zwykłego typu `int`. Trzeba dokonać jawnego rzutowania:

```
b = (int)myNullableInt;
```

Takie przypisanie da się już skompilować, ale w czasie wykonania należy spodziewać się wyjątku, kiedy `myNullableInt` będzie miało wartość pustą (stąd też ujęcie tej instrukcji w bloku chronionym `try`).

Jeśli którykolwiek z operandów ma wartość pustą, operatory relacji dadzą w wyniku false!

Drugi sposób przypisania wartości typu nullable `int` do zmiennej zwykłego typu `int` to udostępnienie wartości domyślnej, wykorzystywanej w przypadku, kiedy to wartość nullable `int` będzie akurat pusta:

```
int c = myNullableInt ?? -1;
```

Powyższy wiersz mówi: „zainicjalizuj `c` wartością `myNullableInt`; chyba że `myNullableInt` ma wartość pustą — wtedy zainicjalizuj `c` wartością `-1`".

Trzeba też pamiętać, że operatory porównań (>, <, <= i tak dalej) zwracają wartość `false`, jeśli którykolwiek z operandów ma wartość pustą. Wynikowi porównania można więc zaufać tylko wtedy, kiedy da ono `true`:

```
if (myNullableInt >= c)
{
    Console.WriteLine("myNullableInt jest większe od (równe) c");
}
```

OSTRZEŻENIE

Wyjątkiem jest operator `==`, który da wartość `true` również wtedy, gdy oba operandy będą puste.

Jeśli na wyjściu wypisany zostanie komunikat „myNullableInt jest większe od (równe) c", wiadomo na pewno, że ani `c`, ani `myNullableInt` nie miało wartości pustej oraz dodatkowo wartość `myNullableInt` jest większa od wartości `c` (albo są one równe). Jeśli jednak porównanie daje wynik `false`, nie można jednoznacznie stwierdzić relacji pomiędzy wartościami:

```
else
{
    Console.WriteLine("Czy myNullableInt jest mniejsze od c?");
}
```

Klauzula `else` może zostać uruchomiona, kiedy okaże się, że albo `myNullableInt` ma wartość mniejszą od `c`, albo `myNullableInt` bądź `c` jest puste.

A co...

... z pustymi wartościami logicznymi? Jak wypadają ich porównania i jak je odnieść do trójwartościowych typów logicznych charakterystycznych dla SQL?

Język C# udostępnia dwa nowe operatory:

```
bool? operator &(bool? x, bool? y)
bool? operator |(bool? x, bool? y)
```

Działanie tych operatorów definiuje tabela prawdy z tabeli 1.1.

Tabela 1.1. Tabela prawdy operatorów logicznych dla typów logicznych z wyróżnioną wartością pustą

x	y	x & y	x | y
prawda	prawda	prawda	prawda
prawda	fałsz	fałsz	prawda
prawda	pusta	pusta	prawda
fałsz	prawda	fałsz	prawda
fałsz	fałsz	fałsz	fałsz
fałsz	pusta	fałsz	pusta
pusta	prawda	pusta	prawda
pusta	fałsz	fałsz	pusta
pusta	pusta	pusta	pusta

Więcej informacji

Świetny artykuł o typach nullable można znaleźć w czeluściach Visual C# Developer Center (*http://msdn.microsoft.com/vcsharp/2005/overview/language/nullabletypes/*).

Odwołania do obiektów z globalnej przestrzeni nazw

Kwalifikator globalnej przestrzeni nazw pozwala na odwoływanie się do identyfikatora z (domyślnie) globalnej przestrzeni nazw; normalnie odwołania są ograniczane do zestawu identyfikatorów z lokalnej przestrzeni nazw, a identyfikatory z tej przestrzeni przesłaniają nazwy definiowane globalnie.

Tak jak w poprzednim wydaniu języka C# do deklarowania zasięgu widoczności nazw (tzw. przestrzeni nazw) służy słowo kluczowe `namespace`. Stosowanie przestrzeni nazw pozwala lepiej organizować kod i zapobiega ewentualnym kolizjom nazw (na przykład próbie zdefiniowania dwóch klas o identycznej nazwie) — przydatność podziału przestrzeni nazw ujawnia się zwłaszcza przy korzystaniu z komponentów zewnętrznych, tworzonych przez osoby trzecie.

Wszelkie obiekty, których nie definiuje się jawnie w którejś z przestrzeni nazw, lądują w globalnej przestrzeni nazw. Obiekty globalnej przestrzeni

nazw są dostępne dla obiektów wszystkich pozostałych (węższych) przestrzeni nazw. W przypadku kolizji nazw potrzebny jest jednak sposób sygnalizowania, że dane odwołanie dotyczy nie lokalnej, a właśnie globalnej przestrzeni nazw.

Jak to zrobić?

Odwołanie do obiektu z globalnej przestrzeni nazw należy zasygnalizować kwalifikatorem zasięgu `global::`, jak na listingu 1.8.

Listing 1.8. Stosowanie globalnej przestrzeni nazw

```
using System;

namespace GlobalNameSpace
{
  class Program
  {
      // utworzenie zagnieżdżonej klasy System udostępniającej zestaw
      // narzędzi interakcji z obsługiwanym przez program systemem;
      // nazwa System koliduje z nazwą przestrzeni nazw System
      public class System
      {
      }

      static void Main(string[] args)
      {

          // znacznik sygnalizujący uruchomienie aplikacji konsoli;
          // koliduje z nazwą Console z przestrzeni nazw System
          bool Console = true;

          int x = 5;

          // Console.WriteLine(x); // odmowa kompilacji – kolizja
                                   // z lokalnym Console
          // System.Console.WriteLine(x); // kolizja z lokalnym System

          global::System.Console.WriteLine(x); // działa
          global::System.Console.WriteLine(Console);
      }
  }
}
```

Wynik:

```
5
True
```

Jak to działa?

Tworzenie zagnieżdżonej w klasie `Program` klasy o nazwie `System` i deklarowanie w metodzie `Main` zmiennej lokalnej o nazwie `Console` to przykład cokolwiek sztuczny. Tym niemniej takie deklaracje blokują w lokalnej przestrzeni nazw dostęp do globalnych identyfikatorów `System` i `Console`, co uniemożliwia kompilację wywołań:

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

Aby zasygnalizować, że chodzi o odwołania do identyfikatora `System` w globalnej przestrzeni nazw, należy zastosować kwalifikator globalnej przestrzeni nazw:

```
global::System.Console.WriteLine(x);
```

Warto też zauważyć, że w ostatnim wierszu kodu w odwołaniu do globalnych identyfikatorów `System` i `Console` stosowany jest kwalifikator globalnej przestrzeni nazw, a niekwalifikowane odwołanie do `Console` dotyczy lokalnej zmiennej metody:

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

A co...

... z innymi zastosowaniami operatora zasięgu (`::`)?

Operator `::` służy jako kwalifikator aliasu przestrzeni nazw. Występuje zawsze pomiędzy dwoma identyfikatorami:

```
identyfikator1::identyfikator2
```

Jeśli `identyfikator1` reprezentuje globalną przestrzeń nazw, operator zasięgu służy do wyciągnięcia identyfikatora `identyfikator2` z tejże przestrzeni globalnej. Ale jeśli `identyfikator1` będzie dowolną przestrzenią nazw inną od przestrzeni globalnej, operator zawęzi poszukiwania `identyfikator2` do zasięgu `identyfikator1`.

Więcej informacji

Kwalifikator globalnej przestrzeni nazw wspominany jest w artykule „Create Elegant Code with Anonymous Methods, Iterators and Partial Classes"

autorstwa Juvala Lowy'ego, publikowanym pod adresem *http://msdn.microsoft.com/msdnmag/issues/04/05/c20/* (MSDN).

Ograniczanie dostępu do właściwości

Wreszcie można ograniczać dostępność akcesorów właściwości.

Nowa specyfikacja języka C# pozwala na ograniczanie poziomu dostępności metod-akcesorów ustawiających i odczytujących właściwości. Służą do tego stosowne modyfikatory dostępu. Zwykle dostęp ogranicza się jedynie do akcesorów ustawiających właściwości (`set`); akcesory odczytujące są zazwyczaj udostępniane publicznie.

Jak to zrobić?

Ograniczenie dostępu do akcesora właściwości polega na opatrzeniu deklaracji tego akcesora stosownym modyfikatorem dostępu, jak na listingu 1.9.

Listing 1.9. Ograniczanie dostępu do akcesora właściwości

```
#region Using directives

using System;
using System.Collections.Generic;
using System.Text;

#endregion

namespace LimitPropertyAccess
{
  public class Employee
  {
      private string name;
      public Employee(string name)
      {
          this.name = name;
      }
      public string Name
      {
          get { return name; }
          protected set { name = value; }
      }
      public virtual void ChangeName(string name)
      {
          // tu operacje aktualizujące rekordy
          Name = name; // odwołanie do prywatnego akcesora
      }
  }
  class Program
```

```
    {
        static void Main(string[] args)
        {
            Employee joe = new Employee("Joe");
            // inne operacje
            string whatName = joe.Name; // działa
            // joe.Name = "Bob"; // odmowa kompilacji
            joe.ChangeName("Bob"); // działa
            Console.WriteLine("imię joe'a: {0}", joe.Name);
        }
    }
}
```

Wynik:

```
imię joe'a: Bob
```

Jak to działa?

Projekt klasy `Employee` (pracownik) sygnalizuje, że ciąg imienia pracownika ma być prywatny. Ale programista przewidział, że kiedyś przyjdzie mu wstawiać dane o pracownikach do bazy danych, więc udostępnił imię za pośrednictwem właściwości `Name`.

Pozostałe klasy programu powinny mieć możliwość odwoływania się do `Name`, ale jedynie w odwołaniach niemodyfikujących. Zmiana wartości pola może się odbywać jedynie za pośrednictwem jawnego wywołania metody `ChangeName`. Metoda została oznaczona jako wirtualna — w przyszłych klasach pochodnych zmiana imienia pracownika będzie się pewnie wiązała z dodatkowymi operacjami.

Zachodzi tu potrzeba udostępnienia akcesora `set`, ale tylko metodom klasy `Employee` i metodom jej klas pochodnych. Ograniczenie takie można wyegzekwować modyfikatorem dostępu dla akcesora `set`:

```
protected set { name = value; }
```

A co…

… z ograniczeniami odnośnie stosowania modyfikatorów dostępu?

Otóż modyfikatorów tych nie można stosować wobec interfejsów i jawnych implementacji składowych interfejsów. Modyfikatory dostępu można stosować jedynie wtedy, kiedy właściwość obejmuje oba akcesory (`get` i `set`); można nimi przy tym opatrywać tylko jeden z akcesorów.

Poza tym modyfikator musi ograniczać, a nie rozszerzać dostępność. Nie można więc zadeklarować właściwości ze słowem `protected`, a potem za pomocą modyfikatora oznaczyć akcesor `get` jako dostępny publicznie.

Więcej informacji

O właściwościach i modyfikatorach dostępu do właściwości traktuje artykuł MSDN publikowany pod adresem *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/csref/html/vclrfPropertiesPG.asp*.

Elastyczność delegacji z kowariancją i kontrawariancją

Kowariancja pozwala na stosowanie z delegacjami metod zwracających wartości typu będącego pochodną (bezpośrednią lub pośrednią) typu zwracanego określonego w definicji delegacji.

Nowa specyfikacja języka C# zezwala na określanie metod delegacji z typem wartości zwracanej, będącej pochodną (bezpośrednią bądź pośrednią) typu zwracanego określonego w definicji delegacji; operacja taka nosi miano *kowariancji*. Chodzi o to, że jeśli definicja delegacji zakłada zwracanie wartości typu `Mammal` (ssak), to delegację tę można zastosować do metody zwracającej wartość typu `Dog` (pies), o ile `Dog` jest pochodną `Mammal`, a także do metody zwracającej wartość typu `Retriever`, o ile `Retriever` to pochodna typu `Dog`, a ten jest pochodną `Mammal`.

Kontrawariancja pozwala na stosowanie z delegacjami metod przyjmujących parametry typu będącego typem bazowym (bezpośrednim lub pośrednim) typu parametru określonego w definicji delegacji.

Analogicznie dozwolone jest przekazywanie sygnatury metody delegacji, w której typy parametrów są pochodnymi typów parametrów zdefiniowanych w delegacji. To z kolei określa się mianem *kontrawariancji*. Chodzi o to, że jeśli definicja delegacji wymaga podania metody przyjmującej parametr typu `Dog`, to można ją użyć z metodą przyjmującą parametr typu `Mammal` (znów pod warunkiem że `Dog` to pochodna `Mammal`).

Jak to zrobić?

Kowariancja i kontrawariancja zwiększają elastyczność w zakresie doboru metod dla delegacji. Zastosowanie kowariancji i kontrawariancji ilustruje listing 1.10.

Listing 1.10. Stosowanie kowariancji i kontrawariancji

```
#region Using directives

using System;
using System.Collections.Generic;
using System.Text;

#endregion

namespace CoAndContraVariance
{

  class Mammal
  {
      public virtual Mammal ReturnsMammal()
      {
          Console.WriteLine("Ssak");
          return this;
      }
  }

  class Dog : Mammal
  {

      public Dog ReturnsDog()
      {
          Console.WriteLine("Pies");
          return this;
      }

  }

  class Program
  {

      public delegate Mammal theCoVariantDelegate();
      public delegate void theContraVariantDelegate(Dog theDog);


      private static void MyMethodThatTakesAMammal(Mammal theMammal)
      {
          Console.WriteLine("W metodzie akceptującej ssaki");
      }

      private static void MyMethodThatTakesADog(Dog theDog)
      {
          Console.WriteLine("W metodzie akceptującej psy");
      }


      static void Main(string[] args)
      {
          Mammal m = new Mammal();
```

```
            Dog d = new Dog();

            theCoVariantDelegate myCoVariantDelegate =
                new theCoVariantDelegate(m.ReturnsMammal);
            myCoVariantDelegate();

            myCoVariantDelegate =
                new theCoVariantDelegate(d.ReturnsDog);
            myCoVariantDelegate();

            theContraVariantDelegate myContraVariantDelegate =
                new theContraVariantDelegate(MyMethodThatTakesADog);
            myContraVariantDelegate(d);

             myContraVariantDelegate =
                new theContraVariantDelegate(MyMethodThatTakesAMammal);
            myContraVariantDelegate(d);
        }
    }
}
```

Wynik:

```
Ssak
Pies
W metodzie akceptującej psy
W metodzie akceptującej ssaki
```

Jak to działa?

Klasa `Program` z listingu 1.10 deklaruje dwie delegacje. Pierwsza z nich obejmuje metody bezparametrowe i zwracające wartości typu `Mammal`:

Z wcześniejszych definicji z listingu 1.10 wynika, że Dog to pochodna Mammal.

```
public delegate Mammal theCoVariantDelegate();
```

W metodzie `run` deklarowane są egzemplarze klas `Mammal` i `Dog` (po jednym z każdej klasy):

```
Mammal m = new Mammal();
Dog d = new Dog();
```

Potem można już przystąpić do utworzenia pierwszego egzemplarza `theCoVariantDelegate`:

```
theCoVariantDelegate myCoVariantDelegate =
    new theCoVariantDelegate(m.ReturnsMammal);
```

Wszystko pasuje do sygnatury delegacji (`m.ReturnsMammal()` to metoda nieprzyjmująca żadnych argumentów i zwracająca wartość typu `Mammal`), można więc przystąpić do wywołania metody za pośrednictwem delegacji:

```
myCoVariantDelegate();
```

Stosując kowariancję, można objąć tą samą delegacją również inną metodę:

```
myCoVariantDelegate =
    new theCoVariantDelegate(d.ReturnsDog);
```

Tym razem metoda realizująca delegację to metoda zwracająca wartość typu `Dog` (`d.ReturnsDog()`), a nie `Mammal`; tak się jednak składa, że typ `Dog` jest pochodną typu `Mammal`:

```
public Dog ReturnsDog()
{
    Console.WriteLine("Pies");
    return this;
}
```

Tak działa kowariancja. Kontrawariancję ilustruje druga z delegacji, której sygnatura zakłada podawanie metod niezwracających żadnych wartości, za to przyjmujących za pośrednictwem argumentu wywołania wartość typu `Dog`:

```
public delegate void theContraVariantDelegate(Dog theDog);
```

Pierwszy egzemplarz delegacji jest konstruowany na bazie metody pasującej do sygnatury deklaracji: metoda zwraca typ `void` i deklaruje parametr typu `Dog`:

```
theContraVariantDelegate myContraVariantDelegate =
    new theContraVariantDelegate(MyMethodThatTakesADog);
```

Metodę tę można oczywiście swobodnie wywoływać przez delegację. W drugim zastosowaniu delegacji została jednak wykorzystana metoda, która co prawda przyjmuje jeden parametr, ale nie typu `Dog`, a typu `Mammal`:

Zauważ, że kontrawariancja pozwala na przekazywanie obiektu klasy bazowej tam, gdzie oczekiwany jest obiekt klasy pochodnej.

```
myContraVariantDelegate =
    new theContraVariantDelegate(MyMethodThatTakesAMammal);
```

Sygnatura metody `MyMethodThatTakesAMammal` zakłada przyjmowanie za pośrednictwem jedynego parametru wywołania obiektów typu `Mammal` (ssaki), nie typu `Dog`:

```
private static void MyMethodThatTakesAMammal(Mammal theMammal)
{
    Console.WriteLine("W metodzie akceptującej ssaki");
}
```

Całość działa, bo `Dog` jest pochodną `Mammal`, a kontrawariancja pozwala na podstawianie nadtypu (typu bazowego) w miejsce typu właściwego.

A co…

… z tą kontrawariancją? Wiadomo, że dzięki kowariancji da się zwrócić obiekt typu Dog (bo `Mammal` zawiera się w `Dog`), ale skąd możliwość i jaki sens podstawienia odwrotnego? Czy tam, gdzie oczekiwany jest pewien typ, nie powinno się przekazywać tego właśnie typu, ewentualnie typu pochodnego?

Dr Jonathan Bruce Postel (1913 – 1998), współtwórca standardów internetowych.

Otóż kontrawariancja jest spójna z regułą Postela, mówiącą, żeby. Klient musi się upewnić, czy to, co przekazał do metody, będzie z tą metodą działało, ale twórca implementacji metody powinien podchodzić do tego liberalnie i akceptować wartości typu `Dog` niezależnie od formy, w jakiej są przekazywane — nawet jeśli występują w postaci referencji do obiektu typu bazowego.

… gdyby odwrócić zastosowanie kowariancji i zwracać typ bazowy tam, gdzie oczekiwany jest typ pochodny? Czy to dozwolone?

Nie, kowariancja działa tylko w jedną stronę. Można zwrócić typ pochodny tam, gdzie oczekiwany jest typ bazowy.

… gdyby odwrócić kontrawariancję i przekazać typ pochodny tam, gdzie oczekiwany jest typ bazowy?

To też nie jest dozwolone. Kontrawariancja też jest jednokierunkowa. Można jedynie przekazywać typ bazowy tam, gdzie oczekiwany jest typ pochodny.

Więcej informacji

Dodatkowych informacji należałoby szukać w archiwach licznych grup dyskusyjnych, gdzie toczyły się zażarte spory o zalety i wady języków obiektowych obsługujących kowariancję i kontrawariancję. Szczegółowego omówienie stosowania kowariancji i kontrawariancji w programach pisanych w języku C# należy szukać w dokumentacji MSDN.