Dodatek C.
Kowariancja i kontrawariancja typów parametrycznych

Jacek Matulewski
Fragment książki „Visual Studio 2017. Tworzenie aplikacji Windows w języku C#”, Helion 2018

Jedną z nowością jezyka C#, która nadal pozostaje stosunkowo mało znana, jest wariancja typów parametrycznych. Akurat w przypadku projektów z tej książki nie pojawiła się okazja, żeby tę technikę wykorzystać, ale i tak uważam, że warto ją poznać. Aby to umożliwić, zaprezentuję prosty przykład, który zilustruje to zagadnienie. Na początku utwórzmy projekt aplikacji konsolowej *Console App (.NET Framework)*, a następnie w pliku *Program.cs* zdefiniujmy interfejs deklarujący cztery metody, wśród nich takie, które przyjmują jako argument obiekty typu określonego w parametrze, jak również takie, które zwracają taki obiekt jako wartość. Interfejs ten jest widoczny na listingu C.1. Zdefiniujmy również prostą klasę, która implementuje ten interfejs, a w metodzie `Program.Main` umieśćmy dwie instrukcje, także widoczne na listingu C.1. W pierwszym tworzymy instancję klasy parametryzowanej typem `object` i zapisujemy ją do referencji typu interfejsu także parametryzowanego typem `object`. W drugim zamiast referencji typu interfejsu używamy referencji typu samej klasy. Nie ma w tym nic nietypowego i oczywiście wszystko zadziała prawidłowo.

Listing C.1. Definicja typów inwariantnych

```
using System.Collections.Generic;

using static System.Console;

namespace WariancjaTypówParametrycznych
{
    interface ITypInwariantny<Typ>
    {
        void Nic();
        void PrzyjmijObiekt(Typ typ);
        Typ ZwróćObiekt();
        Typ PrzyjmijIZwróćObiekt(Typ typ);
    }

    class TypInwariantny<T> : ITypInwariantny<T>
    {
        public void Nic() { }
        public void PrzyjmijObiekt(T obiekt)
        {
            WriteLine(obiekt.ToString());
```

```
        }
        public T ZwróćObiekt()
        {
            return default(T);
        }
        public T PrzyjmijIZwróćObiekt(T obiekt)
        {
            WriteLine(obiekt.ToString());
            return obiekt;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            ITypInwariantny<object> o11 = new TypInwariantny<object>();
            TypInwariantny<object> o12 = new TypInwariantny<object>();
        }
    }
}
```

Podobny rezultat miałaby próba utworzenia obiektów parametryzowanych np. typem `string`:

```
ITypInwariantny<string> o13 = new TypInwariantny<string>();
TypInwariantny<string> o14 = new TypInwariantny<string>();
```

Problem zacznie się wtedy, gdy spróbujemy w parametrze referencji użyć typu bazowego dla tego, którego używamy w tworzonym obiekcie:

```
ITypInwariantny<object> o15 = new TypInwariantny<string>(); //błąd kompilacji
TypInwariantny<object> o16 = new TypInwariantny<string>(); //błąd kompilacji
```

Takie przypisania nie są dozwolone i w obu przypadkach otrzymamy błąd kompilacji. Pewnie nie będzie zaskoczeniem, że rzutowanie „w drugą stronę" także się nie powiedzie, tj. otrzymamy błąd kompilacji również, gdy spróbujemy skompilować następujące polecenia:

```
ITypInwariantny<string> o17 = new TypInwariantny<object>(); //błąd kompilacji
TypInwariantny<string> o18 = new TypInwariantny<object>(); //błąd kompilacji
```

To jednak nie znaczy, że nie możemy nic zrobić. Jeżeli w interfejsie i klasie nie ma metod zwracających obiekty, których typami jest typ parametru (listing C.2), to możemy użyć słowa kluczowego `in` na oznaczenie, że typy te są kontrawariantne i możliwe jest w nich pobieranie obiektów typu parametru, ale już nie ich zwracanie.

Listing C.2. Typ kontrawariantny

```
interface ITypKontrawariantny<in Typ> //można bezpiecznie tylko przypisywać (in)
{
    void Nic();
    void PrzyjmijObiekt(Typ obiekt);
}

class TypKontrawariantny<T> : ITypKontrawariantny<T>
{
    public void Nic() { }
```

```
    public void PrzyjmijObiekt(T obiekt)
    {
        WriteLine(obiekt.ToString());
    }
}
```

W przypadku takiego typu, oprócz rzutowań bez zmiany typu parametru, możliwe staje się także rzutowanie, w którym do referencji typu bazowego parametryzowanej parametrem potomnym przypisany jest obiekt typu potomnego parametryzowany typem bazowym parametru. Szkoda czasu na analizowanie poprzedniego zdania, lepiej spojrzeć na wyróżniony przykład rzutowania z listingu C.3, aby to od razu zrozumieć.

Listing C.3. Możliwe rzutowania w przypadku typu kontrawariantnego

```
ITypKontrawariantny<object> o21 = new TypKontrawariantny<object>();
TypKontrawariantny<object> o22 = new TypKontrawariantny<object>();
ITypKontrawariantny<string> o23 = new TypKontrawariantny<string>();
TypKontrawariantny<string> o24 = new TypKontrawariantny<string>();
//ITypKontrawariantny<object> o25 = new TypKontrawariantny<string>(); //błąd kompilacji
//TypKontrawariantny<object> o26 = new TypKontrawariantny<string>(); //błąd kompilacji
ITypKontrawariantny<string> o27 = new TypKontrawariantny<object>();
//TypKontrawariantny<string> o28 = new TypKontrawariantny<object>(); //błąd kompilacji
```

Spróbujmy zrozumieć, dlaczego taki układ jest bezpieczny. Załóżmy, że referencja jest typu określonego przez interfejs `ITypKontrawariantny` z parametrem typu `string`. To oznacza, że możemy wywołać metodę `PrzyjmijObiekt` z argumentem będącym łańcuchem:

```
o27.PrzyjmijObiekt("łańcuch");
```

Jednak definicja tej metody nic o łańcuchach nie wie — musi być zdefiniowana tak, żeby działała dla dowolnego typu, więc wywołanie jej nie grozi żadnym niebezpieczeństwem. Inaczej byłoby, gdybyśmy mieli metodę, która zwraca wartość typu określonego przez parametr. Wartość ta musiałaby być tworzona w metodzie, która nie wie, jakiego typu tak naprawdę zwraca wartość (kod tej metody nie wie, jakiego typu będzie referencja), co oznacza rzutowanie typu `object` na typ `string`, które nie musi być poprawne (tu nie chodzi o pudełkowanie). Dlatego zwracanie obiektu typu określonego przez parametr w typach kontrawariantnych jest zabronione.

Można także zdefiniować typ kowariantny, tj. taki, w którym obiekty typu parametru mogą być jedynie zwracane przez metody, ale nie pobierane w ich argumentach. W takim przypadku parametr oznacza się słowem kluczowym `out` (listing C.4).

Listing C.4. Typ kowariantny

```
interface ITypKowariantny<out Typ> //można bezpiecznie tylko odbierać wartości (out)
{
    void Nic();
    Typ ZwróćObiekt();
}

class TypKowariantny<T> : ITypKowariantny<T>
{
    public void Nic() { }
    public T ZwróćObiekt()
    {
        return default(T);
    }
}
```

Tu zachodzi sytuacja dokładnie odwrotna do analizowanej przed chwilą, dlatego w tym przypadku możliwe jest rzutowanie na typ bazowy, który jest parametryzowany bazowym typem parametru (listing C.5). Posługiwanie

się referencją może bowiem doprowadzić jedynie do zwrócenia typu potomnego, na którego przyjęcie kod z typem bazowym jest przygotowany.

Listing C.5. Możliwe użycia typu kowariantnego

```
ITypKowariantny<object> o31 = new TypKowariantny<object>();
TypKowariantny<object> o32 = new TypKowariantny<object>();
ITypKowariantny<string> o33 = new TypKowariantny<string>();
TypKowariantny<string> o34 = new TypKowariantny<string>();
ITypKowariantny<object> o35 = new TypKowariantny<string>();
//TypKowariantny<object> o36 = new TypKowariantny<string>(); //błąd kompilacji
//ITypKowariantny<string> o37 = new TypKowariantny<object>(); //błąd kompilacji
//TypKowariantny<string> o38 = new TypKowariantny<object>(); //błąd kompilacji
```

Łatwo się przekonać, jaka jest wariancja popularnych typów parametrycznych zdefiniowanych na platformie .NET. Dla przykładu tablice są kowariantne, tzn. możliwa jest instrukcja typu:

```
object[] tablica = new string[5];
```

Tablicę zawsze można także zrzutować na interfejs `IEnumerable` parametryzowany typem bazowym względem typu parametru w obiekcie, tj.:

```
IEnumerable<object> _tablica = tablica;
```

I rzeczywiście, gdy zajrzymy do deklaracji interfejsu `IEnumerable<>`, znajdziemy tam słowo kluczowe `out`:

```
public interface IEnumerable<out T> : IEnumerable ...
```

Z kolei parametryczne typy delegacji mają określoną wariancję, która odpowiada temu, czy przyjmują, czy zwracają obiekty. Akcje `Action<>` są kontrawariantne:

```
Action<object> a1 = (object o) => { WriteLine(o.ToString()); };
Action<string> a3 = (string s) => { WriteLine(s); };
//Action<object> a5 = a3; //błąd kompilacji
Action<string> a7 = a1;
```

a funkcje `Func<>` kowariantne względem zwracanego typu:

```
Func<object> f1 = () => { return new object(); };
Func<string> f3 = () => { return "łańcuch"; };
Func<object> f5 = f3;
//Func<string> f7 = f1; //błąd kompilacji
```

ale jednocześnie kontrawariantne względem parametrów określających typy argumentów, tzn.:

```
public delegate TResult Func<in T, out TResult>(T arg);
```