Rozdział 4. Projektowanie zorientowane obiektowo

Jacek Matulewski

Programista C# może realizować swoje zadania, korzystając wyłącznie z gotowych komponentów dostarczonych z platformą .NET. Ponadto cały swój kod może umieszczać w tworzonych automatycznie metodach zdarzeniowych. Jednym słowem, może dowolnie długo unikać samodzielnego definiowania własnych klas. Jednak skuteczność, a konkretnie szybkość tworzenia kodu i mniejsza ilość błędów, wymaga od niego przejścia do nowego paradygmatu — projektowania zorientowanego obiektowo. Dobry programista nie obejdzie się bez tworzenia własnych klas. Dzięki nim możliwe jest „zamknięcie" implementowanych funkcjonalności w eleganckim pudełku — klasie. Klasa ukrywa szczegóły implementacji, udostępniając jedynie interfejs pozwalający na kontrolę jej działania. Takie wyraźne ustrukturalizowanie kodu bardzo pomaga również w jego testowaniu.

Zacznijmy jednak od elementarza. **Klasa** to projekt typu określający zawartość tworzonych na bazie tego typu **obiektów**, czyli **instancji klasy**. W klasie można zdefiniować **pola** przechowujące dane, **metody**, które mogą manipulować danymi, oraz **własności**, które są wygodnym mariażem jednych i drugich. Poza polami, metodami i własnościami można w jej obrębie zdefiniować także indeksatory, delegacje i zdarzenia.

W C# mamy do dyspozycji dwa „metatypy": referencyjny i wartościowy (rozdział 3.), zatem pierwszą decyzją, jaką musimy podjąć w procesie projektowania nowego typu, jest wybór między strukturą a klasą. W przypadku nowych typów liczb wybór jest oczywisty — zalety typu wartościowego przemawiają za wyborem struktury. Podobnie oczywista jest sytuacja, gdy tworzymy nową kontrolkę — wówczas musimy definiować klasę. W pozostałych przypadkach, aby ocenić, którego „metatypu" użyć, powinniśmy się zastanowić, jak wiele pamięci będzie on zajmował i jak często będziemy go kopiować lub przekazywać do metod. W przypadku dużej klasy możemy do metody przekazać referencję, co uchroni przed tworzeniem kopii i niepotrzebnym zajmowaniem pamięci. Z kolei używanie struktur i związanego z nimi prostego modelu zarządzania pamięcią zmniejsza obciążenie odśmiecacza, co wpływa na wydajność całej platformy .NET. Należy także wziąć pod uwagę to, że tworzenie i usuwanie obiektów na stosie jest szybsze niż na stercie. A z tego wynika, że struktur należy użyć, jeżeli obiekty projektowanego typu będą niewielkie i wykorzystywane raczej w małych zakresach, gdzie mogą być często tworzone i zaraz usuwane. Natomiast klasy powinny być używane raczej do implementacji typów związanych z wykorzystaniem większych obszarów pamięci (np. przez tablice) czy takich, których inicjacja wiąże się z czasochłonnymi czynnościami. Wówczas przekazywanie adresu obiektu zamiast jego klonowania jest lepszym rozwiązaniem.

Przykład struktury (Ulamek)

Jako przykład definiowania typów przedstawię strukturę `Ulamek`, która będzie implementacją liczb wymiernych. Jak wspomniałem, wybór struktury w przypadku implementacji liczb jest — na szczęście — naturalny i nie musimy się zastanawiać, czy nie lepiej byłoby użyć w tym celu klasy.

Przygotowanie projektu

Zaczynamy od utworzenia projektu aplikacji i w jego obrębie pliku dla struktury, którą będziemy rozwijać.

1. W środowisku Visual Studio naciśnij kombinację klawiszy *Ctrl+Shift+N*.
2. W oknie *New Project* wybierz typ projektu *Windows Forms Application*, ustal jego nazwę na *UlamekDemo* i kliknij *OK*.
3. Po pojawieniu się podglądu formy umieść na nim przycisk `Button`.
4. Do tak przygotowanego projektu dodaj teraz plik, w którym będziesz definiować swoją strukturę. Naciśnij kombinację klawiszy *Ctrl+Shift+A*.
5. W oknie *Add New Item — UlamekDemo* zaznacz ikonę *Class*, w polu *Name* podaj nazwę pliku *Ulamek.cs* i kliknij *Add*.

Dodany plik pojawi się w podoknie *Solution Explorer*, a umieszczony w nim bardzo prosty kod klasy `Ulamek` zobaczymy w edytorze kodu (listing 4.1).

Listing 4.1. Utworzona przez Visual C# klasa pozbawiona jest nawet konstruktora

```
using System;
using System.Linq;
using System.Collections.Generic;
using System.Text;


namespace UlamekDemo
{
      class Ulamek
      {
      }
}
```

Jednak postanowiliśmy zdefiniować strukturę, a nie klasę. Wobec tego pierwszą czynnością będzie zmiana słowa kluczowego `class` na `struct`. W tym celu modyfikujemy kod zgodnie ze wzorem z listingu 4.2. Następnie uzupełniamy powyższy szkielet definicją dwóch pól odpowiadających licznikowi i mianownikowi liczby wymiernej.

Listing 4.2. Transformacja klasy w strukturę wymaga zmiany jednego słowa kluczowego

```
struct Ulamek
{
   private int licznik, mianownik;
}
```

Pola `licznik` i `mianownik` są prywatne (modyfikator `private`). Oznacza to, że tylko metody zdefiniowane w strukturze `Ulamek`, tj. w tej samej strukturze, w której zdefiniowane są te pola, będą mogły odczytać lub zmienić ich wartość, spoza struktury `Ulamek` będą niedostępne. Elementy składowe, które będziemy chcieli udostępniać, powinny być publiczne (modyfikator `public`). Publiczny powinien być szczególnie konstruktor.

Konstruktor i statyczne obiekty składowe

W strukturach nie można umieścić polecenia inicjującego pole w miejscu jego deklaracji. Inicjowane jest ono automatycznie wartością domyślną (zerem). Do samodzielnego określenia wartości pola można wykorzystać konstruktor, ale w przypadku struktur i tu napotkamy na ograniczenie, programista nie może bowiem zdefiniować samodzielnie konstruktora domyślnego (bezargumentowego). Ten jest tworzony automatycznie przez kompilator. Można natomiast zdefiniować konstruktor zawierający argumenty. Aby to zrobić, do definicji struktury dodajemy dwuargumentowy konstruktor i korzystające z niego statyczne obiekty składowe. W tym celu uzupełniamy strukturę zgodnie z listingiem 4.3.

Listing 4.3. Konstruktor i obiekty stałe

```
struct Ulamek
{
   private int licznik, mianownik;

   public Ulamek(int licznik, int mianownik = 1)
   {
      if (mianownik == 0) throw new Exception("Mianownik musi być różny od zera");
      this.licznik = licznik;
      this.mianownik = mianownik;
   }

   public static Ulamek Zero = new Ulamek(0);
   public static Ulamek Jeden = new Ulamek(1);
   public static Ulamek Polowa = new Ulamek(1, 2);
   public static Ulamek Cwierc = new Ulamek(1, 4);
}
```

Użytkownik naszej struktury może korzystać z konstruktora, aby nadać polom odpowiednie wartości. Nasz konstruktor dba przy tym, aby nie powstał obiekt `Ulamek`, którego mianownik miałby wartość równą zero. Należy jednak pamiętać, że — pomimo naszych wysiłków — taki obiekt powstanie, jeżeli użyjemy konstruktora domyślnego! Obecność nowego konstruktora pozwala na zdefiniowanie wewnątrz struktury stałych statycznych, które będą ułatwiać jej inicjację[1]. Ponieważ są one statyczne, nawet w strukturze można je zainicjować w miejscu deklaracji.

Zwróćmy uwagę, że drugi argument konstruktora ma domyślną wartość równą 1. Dzięki temu obiekt możemy tworzyć, podając jedynie wartość licznika. Wówczas ułamek przyjmuje wartość równą liczbie całkowitej z mianownikiem równym jedności. Korzystanie z wartości domyślnej argumentu jest nowością w C# 4.0.

Oprócz statycznych pól struktury, można również definiować statyczne metody (w ich sygnaturze musi znaleźć się modyfikator `static`). Są one wywoływane na rzecz struktury, a nie jej instancji i wobec tego nie mogą w żaden sposób zależeć od stanu obiektu, tj. od wartości pól niestatycznych (w naszym przypadku od pól `licznik` i `mianownik`). Możemy np. zdefiniować metodę statyczną wyświetlającą informacje o strukturze (listing 4.4).

Listing 4.4. Statyczna metoda struktury Ulamek

```
public static string Info()
{
   return "Struktura Ulamek, (c) Jacek Matulewski 2010";
}
```

Pierwsze testy

Przetestujmy działanie struktury, tworząc jej instancję w aplikacji testującej.

1. Przejdź na zakładkę *Form1.cs [Design]*.
2. Kliknij dwukrotnie przycisk widoczny na podglądzie formy. Dzięki temu powstaje domyślna metoda zdarzeniowa dla tego przycisku (związana ze zdarzeniem kliknięcia przycisku `Click`).
3. W niej umieść polecenia z listingu 4.5.

Listing 4.5. Pierwsze testy naszej implementacji ułamka

```
private void button1_Click(object sender, EventArgs e)
```

[1] W identyczny sposób zdefiniowane są np. stałe identyfikujące kolory w strukturze `Color`.

```
{
    Ulamek u2 = new Ulamek(2,1);
    Ulamek u0 = Ulamek.Zero;
    Ulamek uP = Ulamek.Polowa;
    MessageBox.Show(uP.ToString());

    MessageBox.Show(Ulamek.Info());
}
```

Kod kompiluje się i nie zwraca błędów podczas uruchamiania testów (po kliknięciu przycisku). To dobrze! Niestety, pierwszy wyświetlany komunikat zamiast wartości ułamka pokazuje jedynie nazwę typu, a więc `UlamekDemo.Ulamek`. To oznacza jednak tylko tyle, że wywołana na rzecz obiektu ułamka metoda `ToString` wymaga nadpisania. Przy tej okazji przygotujemy także dodatkową metodę służącą do konwersji ułamka na liczbę rzeczywistą.

Konwersje na łańcuch (metoda ToString) i na typ double

Zdefiniujmy metody `ToString` i `ToDouble` służące do konwersji struktury `Ulamek` na łańcuch (`string`) oraz liczbę typu `double`. W tym celu definicję struktury `Ulamek` (z pliku *Ulamek.cs*) uzupełniamy o definicje dwóch metod publicznych widocznych na listingu 4.6.

Listing 4.6. Nadpisana metoda ToString oraz nowa metoda ToDouble

```
public override string ToString()
{
    return licznik.ToString() + "/" + mianownik.ToString();
}


public double ToDouble()
{
    return licznik / (double)mianownik;
}
```

Teraz po uruchomieniu aplikacji i kliknięciu przycisku zobaczymy wartość przechowywaną przez obiekt ułamka.

Zauważmy, że przy definiowaniu metody `ToString`, a dokładnie po wpisaniu słowa kluczowego `override` i spacji, automatycznie pojawiła się lista szablonów. Jeśli wybraliśmy z niej `ToString`, edytor uzupełnił kod o typ `string` zwracany przez tę metodę i wewnątrz niej umieścił wywołanie metody `ToString` z klasy bazowej. Tę linię należy — oczywiście — zastąpić linią widoczną na listingu 4.6.

W przypadku konwersji do typu rzeczywistego moglibyśmy pójść dalej, implementując interfejs `IConvertible` (o tym, co znaczy „implementacja interfejsu” — już za chwilę), który zmusiłby nas do zdefiniowania metod konwersji do wszystkich typów liczbowych platformy .NET. Nie warto jednak tego robić w przypadku projektowania pierwszego w życiu własnego typu.

Metoda upraszczająca ułamek

Kolejną metodą, w jaką warto wyposażyć strukturę `Ulamek`, jest metoda upraszczająca licznik i mianownik poprzez znalezienie ich największego wspólnego dzielnika. Do definicji struktury dodajemy wobec tego publiczną metodę widoczną na listingu 4.7.

Listing 4.7. Definicja metody upraszczającej licznik i mianownik

```
public void Uprosc()
{
    //NWD
```

```
        int mniejsza=Math.Min(Math.Abs(licznik),Math.Abs(mianownik));
        for (int i = mniejsza; i > 0; i--)
            if ((licznik%i==0) && (mianownik%i==0))
            {
                licznik/=i;
                mianownik/=i;
            }

        //znaki
        if (licznik*mianownik<0)
        {
            licznik=-Math.Abs(licznik);
            mianownik=Math.Abs(mianownik);
        }
        else
        {
            licznik=Math.Abs(licznik);
            mianownik=Math.Abs(mianownik);
        }
    }
```

Sprawdźmy jej działanie w metodzie `button1_Click` z pliku *Form1.cs* (listing 4.8).

Listing 4.8. Metoda button1_Click z wywołaniem metody Uprosc

```
    private void button1_Click(object sender, EventArgs e)
    {
        Ulamek u = new Ulamek(4,-2);
        u.Uprosc();
        MessageBox.Show(u.ToString());
    }
```

Własności

Warto zauważyć, że dotychczas struktura `Ulamek` nie posiada żadnych metod pozwalających na modyfikacje pól `licznik` i `mianownik`. Mógłby to być stan zamierzony, ponieważ w strukturach często ogranicza się możliwość inicjacji jedynie do momentu, w którym obiekt powstaje. My jednak zdefiniujemy własności `Licznik` i `Mianownik`, które pozwolą na manipulacje tymi składnikami. Ich definicję pokazuję na listingu 4.9. Kod znajdujący się w sekcji `get` wykonywany jest w trakcie odczytu, a w sekcji `set` — w czasie przypisania wartości do własności. W tym względzie własności przypominają metody dostępowe (tzw. akcesory). A przecież sposób ich użycia jest identyczny ze sposobem korzystania z pól. Własności łączą więc wygodę pól z elastycznością akcesorów — warto się nimi posługiwać.

Listing 4.9. Własności zajmują zwykle sporo linii, więc warto umieścić je w bloku

```
    #region Wlasnosci
    public int Licznik
    {
        get //odczyt
        {
            return licznik;
        }
```

```
        set //zapis
        {
            licznik = value;
        }
    }

    public int Mianownik
    {
        get //odczyt
        {
            return mianownik;
        }
        set //zapis
        {
            if (value == 0) throw new Exception("Mianownik musi być różny od zera");
            mianownik = value;
        }
    }
    #endregion
```

Jak widać, własność `Licznik` w obecnej postaci nie przynosi wiele pożytku. Równie dobrze można by zmienić zakres pola `licznik` na publiczny. Inaczej jest w przypadku własności `Mianownik`. Fakt, że podczas przypisania wartości do własności można wykonać dowolny kod, umożliwia np. sprawdzenie, czy przypisywaną wartością nie jest zero. Jeżeli jest — możemy zareagować, zgłaszając wyjątek.

Oczywiście, własności nie są nowym pomysłem. Obecne są we wszystkich językach, które związane były z narzędziami projektowania wizualnego, a więc współpracującymi z środowiskami wyposażonymi w okno własności lub jego odpowiednik. To właśnie publiczne własności komponentów możemy modyfikować za pomocą odpowiedników podokna *Properties*. Tak samo jest także w Visual Studio.

Aby przetestować działanie własności, wykorzystajmy je do inicjacji obiektu (podrozdział „Nowa forma inicjacji obiektów i tablic" z poprzedniego rozdziału):

```
Ulamek u = new Ulamek { Licznik = 1, Mianownik = 2 };
MessageBox.Show(u.ToString());
```

Zastąpi to konstrukcję:

```
Ulamek u = new Ulamek();
u.Licznik = 1;
u.Mianownik = 2;
```

lub

```
Ulamek k = new Ulamek(1, 2);
```

Jak wspomniałem w poprzednim rozdziale, ta nowa forma inicjacji w pewnym sensie zwalnia z obowiązku definiowania konstruktorów inicjujących stan tworzonych obiektów. Zamiast konstruktorów wystarczy zdefiniować publiczne pola lub własności.

Operatory arytmetyczne

Język C# pozwala na przeciążanie operatorów dla definiowanych typów. Szczególnie wygodne jest to w przypadkach takich jak nasz, tzn. gdy implementujemy typ, w którym mają sens operacje arytmetyczne i dla którego operatory zachowują swój naturalny sens. Unikamy wtedy największej groźby związanej z przeciążaniem operatorów — nadawania im znaczenia niezgodnego z intuicją. Użycie operatorów skraca kod i sprawia, że jest znacznie bardziej przejrzysty, a dzięki temu bardziej odporny na błędy, ale musimy zachować zdrowy rozsądek.

W strukturze `Ulamek` zdefiniujemy operatory wykonujące operacje arytmetyczne na ułamkach. Posłużmy się do tego wzorem widocznym na listingu 4.10.

Listing 4.10. Definicje operatorów +, –, * i /

```
#region Operatory
//operatory arytmetyczne
public static Ulamek operator -(Ulamek u)
{
   return new Ulamek(-u.Licznik,u.Mianownik);
}

public static Ulamek operator +(Ulamek u1, Ulamek u2)
{
   Ulamek wynik=new
Ulamek(u1.Licznik*u2.Mianownik+u2.Licznik*u1.Mianownik,u1.Mianownik*u2.Mianownik);
   wynik.Uprosc();
   return wynik;
}

public static Ulamek operator -(Ulamek u1, Ulamek u2)
{
   Ulamek wynik=new Ulamek(u1.Licznik*u2.Mianownik-
u2.Licznik*u1.Mianownik,u1.Mianownik*u2.Mianownik);
   wynik.Uprosc();
   return wynik;
}

public static Ulamek operator *(Ulamek u1, Ulamek u2)
{
   Ulamek wynik=new Ulamek(u1.Licznik*u2.Licznik,u1.Mianownik*u2.Mianownik);
   wynik.Uprosc();
   return wynik;
}

public static Ulamek operator /(Ulamek u1, Ulamek u2)
{
   Ulamek wynik=new Ulamek(u1.Licznik*u2.Mianownik,u1.Mianownik*u2.Licznik);
   wynik.Uprosc();
   return wynik;
}
#endregion
```

Operatory zostały zdefiniowane jako statyczne elementy składowe struktury. Wobec tego do wykonywania operacji nie jest angażowany bieżący stan obiektu; obiekty, na podstawie których operator oblicza nową wartość, przekazywane są przez argumenty operatora. Nie musimy, a nawet nie możemy, definiować operatorów `+=`, `-=`, `*=` i `/=`. Kompilator buduje je sam na podstawie operatorów arytmetycznych i operatora przypisania.

Aby przetestować zdefiniowane przed chwilą operatory, przechodzimy na zakładkę *Form1.cs* i w metodzie testującej umieszczamy polecenia widoczne na listingu 4.11.

Listing 4.11. Testowanie operatorów arytmetycznych

```
private void button1_Click(object sender, EventArgs e)
{
   Ulamek a=Ulamek.Polowa;
   Ulamek b=Ulamek.Cwierc;

   MessageBox.Show((a+b).ToString());
   MessageBox.Show((a-b).ToString());
   MessageBox.Show((a*b).ToString());
   MessageBox.Show((a/b).ToString());
}
```

Operatory porównania oraz metody Equals i GetHashCode

Możemy również zdefiniować operatory pozwalające na porównywanie dwóch instancji struktury `Ulamek`. Zdefiniujemy operatory `==`, `!=`, `<`, `>`, `<=`, `>=` oraz metody `Equals` i `GetHashCode`. W tym celu wracamy do edycji pliku *Ulamek.cs* i do bloku kodu zawierającego zdefiniowane w poprzednim ćwiczeniu operatory dodajemy definicję kolejnych operatorów oraz metod widocznych na listingu 4.12.

Listing 4.12. Definicja operatorów ==, !=, <, <=, >, >= oraz metod Equals i GetHashCode

```
//operatory logiczne
public static bool operator ==(Ulamek u1, Ulamek u2)
{
   return (u1.ToDouble()==u2.ToDouble());
}

public static bool operator !=(Ulamek u1, Ulamek u2)
{
   return !(u1==u2);
}

public override bool Equals(object obj)
{
   if (!(obj is Ulamek)) return false;
   Ulamek u=(Ulamek)obj;
   return (this==u);
}

public override int GetHashCode()
{
   return licznik^mianownik;
}

public static bool operator >(Ulamek u1, Ulamek u2)
{
   return (u1.ToDouble()>u2.ToDouble());
}

public static bool operator >=(Ulamek u1, Ulamek u2)
```

```
{
    return (u1.ToDouble()>=u2.ToDouble());
}

public static bool operator <(Ulamek u1, Ulamek u2)
{
    return (u1.ToDouble()<u2.ToDouble());
}

public static bool operator <=(Ulamek u1, Ulamek u2)
{
    return (u1.ToDouble()<=u2.ToDouble());
}
```

Niektóre operatory muszą być definiowane jednocześnie. Operator `==` nie może być zdefiniowany bez operatora `!=` oraz metod `Equals` i `GetHashCode`, operator `<` bez operatora `>` itd. W przypadku struktur sens operatora `==` sprowadza się w zasadzie do działania metody `Equals`. Natomiast w klasie operator `==` powinien sprawdzać, czy referencje są sobie równe, a więc czy przechowują ten sam adres i, w konsekwencji, czy wskazują na ten sam obiekt. Natomiast do porównywania stanu obiektów powinna wówczas służyć metoda `Equals`.

W przypadku definiowania metody `Equals` struktury `Ulamek` porównującej stany dwóch obiektów konieczne jest podjęcie decyzji, w jaki sposób porównywać dwa ułamki. Czy przez równość ułamków rozumiemy dosłowną równość obiektów? W takim przypadku `Equals` powinna porównywać własności `Licznik` i `Mianownik` i jeżeli są takie same, to wtedy (i tylko wtedy) uznamy, że obiekty są identyczne, a `Equals` zwróci wartość `true`. A może należy brać pod uwagę interpretację tej struktury jako liczby wymiernej i sprawdzać, czy wartości ułamków utworzonych z liczników i mianowników porównywanych obiektów są sobie równe? W pierwszym przypadku porównanie ½ i $^{2}/_{4}$ da wynik negatywny, a w drugim — pozytywny. Powyżej zaimplementowaliśmy drugą opcję.

Operatory konwersji

Kolejna rodzina operatorów to operatory konwersji. Przygotujemy operator konwersji do typu `double`, który będzie mógł być użyty jedynie jawnie, oraz operator konwersji z typu `int` do typu `Ulamek`, który będzie mógł być używany niejawnie.

1. Wcześniej zdefiniowałeś metodę `ToDouble`, zatem określenie operatora konwersji na typ `double` nie powinno być problemem (listing 4.13).

Listing 4.13. Definicja operatora jawnej konwersji na typ double

```
public static explicit operator double(Ulamek u)
{
    return u.ToDouble();
}
```

2. Teraz zdefiniuj operator konwersji z typu `int` na `Ulamek` (listing 4.14).

Listing 4.14. Definicja operatora konwersji z typu int na Ulamek

```
public static implicit operator Ulamek(int n)
{
    return new Ulamek(n);
}
```

3. I na koniec przetestuj nowe operatory w metodzie `button1_Click` (zakładka *Form1.cs*, listing 4.15).

Listing 4.15. Test operatorów konwersji

```
private void button1_Click(object sender, EventArgs e)
```

```
{
   double r=(double)Ulamek.Polowa;
   MessageBox.Show(r.ToString());
   Ulamek c=1;
   MessageBox.Show(c.ToString());
}
```

Zdefiniowaliśmy dwa operatory konwersji: z typu `Ulamek` na typ `double` oraz z typu `int` na `Ulamek`. W przypadku liczb wymiernych konwersja na liczbę rzeczywistą jest zawsze możliwa, ale wiąże się z utratą informacji ($^1/_3$ to nie do końca to samo, co 0,3333333333333), dlatego nie zezwalamy na jej wykonywanie niejawne (co gwarantuje modyfikator `explicit`), aby programista był świadomy stosowanego przybliżenia. Inaczej wygląda sprawa z konwersją liczby całkowitej na wymierną. Ta nie pociąga za sobą żadnego ryzyka, więc nie musi być jawna. Odpowiada za to modyfikator `implicit`.

Ta druga konwersja bardzo ułatwi korzystanie z operatorów i funkcji matematycznych. Za jej pomocą możemy zainicjować ułamek liczbą całkowitą (np. `Ulamek u=1;`) lub dodać ułamek do liczby całkowitej (możliwe są zarówno operacje `u+2`, jak i `2+u`) bez definiowania odpowiedniego operatora.

Implementacja interfejsu (na przykładzie IComparable)

Nic nie stoi na przeszkodzie, aby instancje struktury `Ulamek` były umieszczane w tablicach. Możemy np. zadeklarować dziesięcioelementową tablicę, co zostało pokazane na listingu 4.16.

Listing 4.16. Przykład definiowania tablicy zawierającej ułamki

```
Ulamek[] tablica = new Ulamek[10];
for (int i = 0; i < tablica.Length; i++) tablica[i] = new Ulamek(1, i+1);

string s = "";
foreach (Ulamek u in tablica) s+=u.ToString()+"="+u.ToDouble().ToString()+"\n";
MessageBox.Show(s);
```

Jednak próba ich posortowania poleceniem `Array.Sort(tablica);` skończy się niepowodzeniem. Zgłoszony zostanie wyjątek `InvalidOperationException`. Metoda ta wymaga bowiem od sortowanych elementów, aby implementowały poznany już wcześniej interfejs `IComparable<Ulamek>`, który wymusza na implementującej go klasie lub strukturze obecność metody `CompareTo`. Argumentem tej metody jest obiekt, do którego porównywana jest bieżąca instancja struktury.

Metoda `CompareTo` powinna zwracać wartość całkowitą, która jest mniejsza od zera, jeżeli obiekt porównywany jest większy od bieżącego (od obiektu wskazywanego przez `this`), równa zero, jeżeli oba obiekty są równe, i większa od zera, jeżeli to bieżący obiekt jest większy. My użyjemy liczb –1, 0 i 1.

1. Do sygnatury struktury `Ulamek` dodaj implementowany interfejs. Nowa sygnatura powinna wyglądać następująco:

   ```
   struct Ulamek : IComparable<Ulamek>
   ```

2. W obrębie struktury `Ulamek` zdefiniuj metodę `CompareTo` widoczną na listingu 4.17.

Listing 4.17. Metoda CompareTo wymagana przez interfejs IComparable<Ulamek>

```
public int CompareTo(Ulamek u)
{
   double roznica = this.ToDouble() - u.ToDouble();
   if (roznica != 0) roznica /= Math.Abs(roznica);
   return (int)(roznica);
}
```

3. Teraz możesz posortować tablicę z listingu 4.16. W tym celu w metodzie związanej ze zdarzeniem `Click` nowego przycisku umieść polecenia widoczne na listingu 4.18.

Listing 4.18. Sortowanie tablicy ułamków

```
private void button1_Click(object sender, EventArgs e)
{
   Ulamek[] tablica = new Ulamek[10];
   for (int i = 0; i < tablica.Length; i++) tablica[i] = new Ulamek(1, i+1);

   string s = "Przed sortowaniem:\n";
   foreach (Ulamek u in tablica) s+=u.ToString()+"="+u.ToDouble().ToString()+"\n";
   MessageBox.Show(s);

   try
   {
      Array.Sort(tablica);
   }
   catch (Exception exc)
   {
      MessageBox.Show(exc.Message);
   }

   s = "Po sortowaniu:\n";
   foreach (Ulamek u in tablica) s+=u.ToString()+"="+u.ToDouble().ToString()+"\n";
   MessageBox.Show(s);
}
```

Na tym zakończymy implementację ułamka. Warto zwrócić uwagę, że zbudowaliśmy strukturę, która pozwala na swobodne korzystanie z ułamków do wykonywania obliczeń. Wbrew temu, co może się wydawać, nie jest to tylko wprawka programistyczna. Ułamki, w odróżnieniu od liczb rzeczywistych, mogą przechowywać wartości ze znacznie większą precyzją, co może być kluczowe w niektórych zagadnieniach.

Definiowanie typów parametrycznych

Typy parametryczne są w oficjalnej nomenklaturze języka C# nazywane typami ogólnymi (ang. *generic types*)[2]. Pozwalają na definiowanie klas i struktur bazujących na typie, który wskazywany jest przez programistę dopiero w momencie tworzenia ich instancji. W ten sposób można przygotować struktury danych, które mogą być używane do przechowywania obiektów różnego typu. W pewnym zakresie typy parametryczne są więc zarządzaną wersją szablonów z biblioteki STL języka C/C++.

Typy parametrów mogą być zawężone do pewnych grup, np. do klasy, struktur, typów posiadających konstruktor domyślny lub implementujących wskazany interfejs. Niestety, nie ma sposobu, aby parametry ograniczyć do liczb (lub jeszcze lepiej do typów, w których zdefiniowane są operatory arytmetyczne). Typy „podstawowe" C# nie implementują bowiem wspólnego interfejsu. W konsekwencji nie ma prostego sposobu, aby zdefiniować typ parametryczny, w którym używane są operatory arytmetyczne. To oznacza też, że nie ma prostego sposobu, aby sensownie sparametryzować zdefiniowaną wyżej strukturę `Ulamek`. I choć można ten

[2] Warto zauważyć, że typy ogólne są powszechnym trendem w językach zarządzanych. Pojawiły się również w Javie 5.0.

problem obejść[3], w tej książce powinien znaleźć się raczej jakiś charakterystyczny przykład użycia typów ogólnych. A ich typowym zastosowaniem są kolekcje.

Definiowanie typów ogólnych

Przygotujmy zatem typ o nazwie `Para` zawierający dwa pola typu określonego przez parametr `T`: `pierwszy` i `drugi`:

1. Utwórz nowy projekt i dodaj do niego plik klasy (menu *Project*, *Add Class…*) o nazwie *Para.cs* (klasa zostanie automatycznie nazwana `Para`).
2. W pliku *Para.cs* rozbuduj definicję klasy `Para` według wzoru zamieszczonego na listingu 4.19.

Listing 4.19. Typ ogólny Para

```
using System;
using System.Linq;
using System.Collections.Generic;
using System.Text;

namespace Pary
{
    class Para<T>
    {
        private T pierwszy = default(T), drugi = default(T);

        public Para(T pierwszy, T drugi)
        {
            this.pierwszy = pierwszy;
            this.drugi = drugi;
        }

        public override string ToString()
        {
            return pierwszy.ToString() + "\t" + drugi.ToString();
        }
    }
}
```

3. Przejdź do widoku projektowania formy i na jej podglądzie umieść przycisk `Button`. Kliknij go dwukrotnie, aby utworzyć domyślną metodę zdarzeniową, i umieść w niej polecenia z listingu 4.20.

Listing 4.20. Tworzymy tablice Para<int>, Para<double> i Para<string>

```
private void button1_Click(object sender, EventArgs e)
{
   Random r=new Random();

   //int
   Para<int>[] pi=new Para<int>[10];
```

[3] Różne podejścia do rozwiązania problemu obliczeń w typach parametrycznych przedstawione są w artykule *Using Generics for Calculations* Rüdigera Klaehna opublikowanym w serwisie *www.codeproject.com*. Warto również zaznajomić się z artykułem Keitha Farmera pt. *Operator Overloading with Generics* z tego samego serwisu.

```
        for (int i = 0; i < pi.Length; i++)
           pi[i]=new Para<int>(r.Next(10),r.Next(10));
        string si = "";
        foreach (Para<int> para in pi) si += para.ToString() + "\n";
        MessageBox.Show("Para<int>\n" + si);

        //double
        Para<double>[] pd=new Para<double>[10];
        for (int i = 0; i < pi.Length; i++)
           pd[i] = new Para<double>(r.NextDouble(), r.NextDouble());
        string sd = "";
        foreach (Para<double> para in pd) sd += para.ToString() + "\n";
        MessageBox.Show("Para<double>\n" + sd);

        //string
        Para<string>[] ps = new Para<string>[12];
        ps[0] = new Para<string>("Bester", "Alfred");
        ps[1] = new Para<string>("Dick", "Philip");
        ps[2] = new Para<string>("Tolkien", "John");
        ps[3] = new Para<string>("Lem", "Stanisław");
        ps[4] = new Para<string>("Anderson", "Poul");
        ps[5] = new Para<string>("Pohl", "Frederik");
        ps[6] = new Para<string>("Le Guin", "Ursula");
        ps[7] = new Para<string>("Card", "Orson");
        ps[8] = new Para<string>("Gibson", "William");
        ps[9] = new Para<string>("Asimov", "Isaac");
        ps[10] = new Para<string>("Niven", "Larry");
        ps[11] = new Para<string>("Herbert", "Frank");
        string ss = "";
        foreach (Para<string> para in ps) ss += para.ToString() + "\n";
        MessageBox.Show("Para<string>\n" + ss);
    }
```

Nowy parametryzowany typ `Para<T>` zawiera dwa prywatne pola typu określonego przez parametr `T`, o nazwach `pierwszy` i `drugi`. Klasa `Para<T>` zawiera również konstruktor, który jest jedynym sposobem inicjacji pól[4], oraz metodę `ToString`[5], nadpisującą metodę odziedziczoną z niejawnej klasy bazowej. W przypadku struktur jest nią `System.ValueType`, która z kolei dziedziczy po `System.Object` (alias `object`).

Słowo kluczowe `default` użyte do inicjacji pól klasy `Para`, tych, które są typu określonego przez parametr, zwraca wartość domyślną dla typu `T` podanego w jego argumencie, czyli `0` dla `int`, `0.0` dla `double` i łańcuch pusty dla `string`.

[4] W przypadku klas, a `Para` jest klasą, zdefiniowanie własnego konstruktora powoduje, że kompilator nie tworzy konstruktora domyślnego.

[5] W klasach po zdefiniowaniu dowolnego konstruktora konstruktor bezargumentowy (domyślny) nie jest automatycznie tworzony. Jak pamiętamy, inaczej jest w przypadku struktur, w których taki konstruktor powstaje zawsze.

Określanie warunków, jakie mają spełniać parametry

W tej chwili parametr `T` klasy `Para<T>` może być dowolnym typem .NET. W przykładzie z listingu 4.20 używamy typów `int`, `double` i `string`, ale mogłyby to być równie dobrze typy `Button` lub `Ulamek`. Możemy więc jedynie założyć, że parametr posiada metody z klasy `object` — ostatecznej klasy bazowej wszystkich typów. To wystarczy do ich gromadzenia, kopiowania, przenoszenia i prezentacji użytkownikowi (klasa `object` posiada metodę `ToString`), czyli wszystkiego, co jest potrzebne, aby instancje klasy `Para<T>` mogły być elementami kolekcji. Kolekcja taka nie mogłaby być jednak sortowana. Nie wiemy, jak porównywać obiekty typu `Para<T>`. Do tego konieczne jest zaimplementowanie interfejsu `IComparable<Para<T>>` i zdefiniowanie metody `CompareTo` porównującej dwie pary. Jak mają być porównywane? Przyjmijmy następujący przepis: najpierw porównujemy pierwsze pola obu obiektów i dopiero jeżeli są one równe, o kolejności decyduje drugi element. Dzięki sparametryzowaniu klasy pola będą mogły być zarówno liczbami, jak i łańcuchami. Od typu mającego pełnić rolę parametru wymagać musimy jedynie tego, aby sam implementował interfejs `IComparable<T>`, a więc aby można było porównywać nawzajem elementy typu `T`.

Jak wspomniałem, C# umożliwia nałożenie na parametr żądania, aby implementował wskazany interfejs lub interfejsy. My zażądamy, aby implementował interfejs `IComparable<T>`. W tym celu zmieniamy sygnaturę klasy `Para<T>`, uzupełniając ją o kod wyróżniony w listingu 4.21.

Listing 4.21. Żądamy, aby typ T zawierał metodę CompareTo

```
private class Para<T> where T : IComparable<T>
{
   private T pierwszy=default(T), drugi=default(T);

   public Para(T pierwszy,T drugi)
   {
      this.pierwszy=pierwszy;
      this.drugi=drugi;
   }

   public override string ToString()
   {
      return pierwszy.ToString() + "\t" + drugi.ToString();
   }
}
```

Implementacja interfejsów przez typ ogólny

Żądanie, aby obiekty typu `T` można było ze sobą porównywać, to dopiero pierwszy krok do możliwości porównywania całych par. Jeżeli chcemy sortować elementy tablic `pi`, `pd` i `ps` zdefiniowanych w listingu 4.20, czyli elementy typu `Para<int>`, `Para<double>` i `Para<string>`, to także typ `Para<T>` musi implementować interfejs `IComparable<Para<T>>`. Zaimplementujmy zatem w klasie `Para<T>` interfejs `IComparable<Para<T>>`. W tym celu:

1. Jeszcze raz zmień sygnaturę klasy `Para<T>`, dodając do niej implementowany interfejs `IComparable<Para<T>>`. W konsekwencji do klasy dodasz także metodę `CompareTo` widoczną na listingu 4.22.

Listing 4.22. Także klasa Para implementuje interfejs IComparable

```
private class Para<T> : IComparable<Para<T>> where T : IComparable<T>
{
   private T pierwszy=default(T), drugi=default(T);

   public Para(T pierwszy,T drugi)
```

```
    {
        this.pierwszy=pierwszy;
        this.drugi=drugi;
    }

    public override string ToString()
    {
        return pierwszy.ToString() + "\t" + drugi.ToString();
    }

    public int CompareTo(Para<T> innaPara)
    {
        int wartosc = pierwszy.CompareTo(innaPara.pierwszy);
        if (wartosc != 0) return wartosc;
            else return drugi.CompareTo(innaPara.drugi);
    }
}
```

2. Następnie przejdź do edycji pliku *Form1.cs* i uzupełnij metodę testującą o możliwe teraz polecenia sortowania i prezentację posortowanych tablic (listing 4.23).

Listing 4.23. Do sortowania korzystamy ze statycznej metody Array.Sort

```
private void button1_Click(object sender, EventArgs e)
{
    Random r=new Random();

    //int
    Para<int>[] pi=new Para<int>[10];
    for (int i = 0; i < pi.Length; i++)
        pi[i]=new Para<int>(r.Next(10),r.Next(10));
    string si = "";
    foreach (Para<int> para in pi) si += para.ToString() + "\n";
    MessageBox.Show("Para<int>\n" + si);
    Array.Sort(pi);
    si = "";
    foreach (Para<int> para in pi) si += para.ToString() + "\n";
    MessageBox.Show("Para<int> po sortowaniu\n" + si);

    //double
    Para<double>[] pd=new Para<double>[10];
    for (int i = 0; i < pi.Length; i++)
    pd[i] = new Para<double>(r.NextDouble(), r.NextDouble());
    string sd = "";
    foreach (Para<double> para in pd) sd += para.ToString() + "\n";
    MessageBox.Show("Para<double>\n" + sd);
    Array.Sort(pd);
    sd = "";
    foreach (Para<double> para in pd) sd += para.ToString() + "\n";
```

```
        MessageBox.Show("Para<double> po sortowaniu\n" + sd);

        //string
        Para<string>[] ps = new Para<string>[12];
        ps[0] = new Para<string>("Bester", "Alfred");
        ps[1] = new Para<string>("Dick", "Philip");
        ps[2] = new Para<string>("Tolkien", "John");
        ps[3] = new Para<string>("Lem", "Stanisław");
        ps[4] = new Para<string>("Anderson", "Poul");
        ps[5] = new Para<string>("Pohl", "Frederik");
        ps[6] = new Para<string>("Le Guin", "Ursula");
        ps[7] = new Para<string>("Card", "Orson");
        ps[8] = new Para<string>("Gibson", "William");
        ps[9] = new Para<string>("Asimov", "Isaac");
        ps[10] = new Para<string>("Niven", "Larry");
        ps[11] = new Para<string>("Herbert", "Frank");
        string ss = "";
        foreach (Para<string> para in ps) ss += para.ToString() + "\n";
        MessageBox.Show("Para<string>\n" + ss);
        Array.Sort(ps);
        ss = "";
        foreach (Para<string> para in ps) ss += para.ToString() + "\n";
        MessageBox.Show("Para<string> po sortowaniu\n" + ss);
    }
```

Definiowanie aliasów

Zdefiniujmy aliasy dla `Para<int>`, `Para<double>` i `Para<string>`. Zmieniamy nazwę przestrzeni nazw w pliku *Para.cs* na `Para`. Następnie w sekcji `using` pliku *Form1.cs* dodajemy linie widoczne na listingu 4.24.

Listing 4.24. Zmiany w sekcji using

```
using Para;
using ParaInt = Para.Para<int>;
using ParaDouble = Para.Para<double>;
using ParaString = Para.Para<string>;
```

Mieliśmy tylko zdefiniować alias do specjalizacji naszej sparametryzowanej klasy, a w powyższym ćwiczeniu dominują czynności związane ze zmianą lokalizacji owej klasy. Niestety, było to konieczne. Tworzenie aliasu może bowiem mieć miejsce w sekcji poleceń `using`, czyli przed deklaracją przestrzeni nazw. Ale już w tej linii kompilator musi znać definicję klasy `Para<T>`. Jedyny sposób, aby pogodzić oba wymagania, to umieścić klasę w osobnym pliku i w osobnej przestrzeni nazw, a następnie dodać tę przestrzeń do zbioru zadeklarowanych nazw przed zdefiniowaniem aliasu.

Po tych wszystkich czynnościach możemy jednak korzystać z typów `ParaInt`, `ParaDouble` i `ParaString` w taki sam sposób jak z `Para<int>`, `Para<double>` i `Para<string>`.

Typy ogólne z wieloma parametrami

Na koniec zauważmy, że nie jest konieczne, aby oba pola klasy `Para` (tj. `pierwszy` i `drugi`) były tego samego typu. Rozdzielmy je zatem, definiując szablon, w którym skorzystamy z dwóch parametrów typów `T` i `C` określających niezależnie typ obu przechowywanych wartości.

1. Możesz albo zdefiniować drugą klasę `Para` o innych parametrach (jest to w C# dopuszczalne), albo zmodyfikować istniejącą już klasę `Para` z pliku *Para.cs* (listing 4.25).

Listing 4.25. Wytłuszczone zostały zmiany względem oryginalnej klasy Para

```
public class Para<T,C> : IComparable<Para<T,C>>
      where T : IComparable<T>
      where C : IComparable<C>
{
    private T pierwszy = default(T);
    private C drugi = default(C);

    public Para(T pierwszy, C drugi)
    {
        this.pierwszy = pierwszy;
        this.drugi = drugi;
    }

    public override string ToString()
    {
        return pierwszy.ToString() + "\t" + drugi.ToString();
    }

    public int CompareTo(Para<T,C> innaPara)
    {
        int wartosc = pierwszy.CompareTo(innaPara.pierwszy);
        if (wartosc != 0) return wartosc;
        else return drugi.CompareTo(innaPara.drugi);
    }
}
```

2. W pliku *Form1.cs* przedefiniuj aliasy dla par zawierających wyłącznie elementy typu `int`, `double` i `string` oraz dodaj przykład aliasu dla typu mieszanego (listing 4.26).

Listing 4.26. Nowe aliasy

```
using Para;
using ParaInt = Para.Para<int,int>;
using ParaDouble = Para.Para<double,double>;
using ParaString = Para.Para<string,string>;
using ParaIntDouble = Para.Para<int,double>;
```

3. Zdefiniuj nową zawartość metody testującej `button1_Click` (listing 4.27).

Listing 4.27. Testujemy pary mieszane

```
private void button1_Click(object sender, EventArgs e)
{
   Random r=new Random();

   //int-double
   ParaIntDouble[] p = new ParaIntDouble[10];
   for (int i = 0; i < p.Length; i++)
      p[i] = new ParaIntDouble(r.Next(10), r.NextDouble());
```

```
        string s = "";
        foreach (ParaIntDouble para in p) s += para.ToString() + "\n";
        MessageBox.Show("Para<int,double>\n" + s);
        Array.Sort(p);
        s = "";
        foreach (ParaIntDouble para in p) s += para.ToString() + "\n";
        MessageBox.Show("Para<int,double> po sortowaniu\n" + s);
    }
```

Rozszerzenia

Czasem zdarza się sytuacja, w której chcielibyśmy jakiś typ (klasę bądź strukturę) rozszerzyć o nową metodę, jednak nie jest to możliwe, ponieważ etap edycji tej klasy został już zamknięty lub po prostu nie mamy jej kodu (np. jest to jedna z klas platformy .NET). W takiej sytuacji, jeżeli problem dotyczy platformy .NET 3.5 lub nowszej, możemy zdefiniować metodę rozszerzającą. Metody rozszerzające (ang. *extension methods*) lub po prostu rozszerzenia są definiowane jako metody statyczne w statycznych klasach, ale mogą być wywoływane tak jak metody składowe innych klas. Dla przykładu „dodajmy" do klasy `String` metodę zmieniającą apostrof w znak *hash*. W tym celu musimy zdefiniować statyczną klasę niezagnieżdżoną w innej klasie (zdefiniowaną bezpośrednio w przestrzeni nazw) i umieścić w niej definicję statycznej metody, której argument jest typu, jaki rozszerzamy, i poprzedzony jest słowem kluczowym `this` (listing 4.28) Trochę to zawiłe, ale wystarczy spojrzeć na przykład, aby wszystko się wyjaśniło.

Listing 4.28. Rozszerzanie klasy String o metodę UsunApostrof

```
static class Rozszerzenia //poniższa klasa nie może być zagnieżdżona w innej
{
    public static string UsunApostrof(this String argument)
    {
        return argument.Replace('\'', '#');
    }
}
```

Od tej chwili możemy metodę `UsunApostrof` wywołać na rzecz dowolnego łańcucha (instancji klasy `String`), np. `MessageBox.Show("SQL Inje'c'tion".UsunApostrof());`. Rozszerzenie to widoczne jest również w listach podpowiedzi generowanych przez mechanizm IntelliSense.

Metoda rozszerzona może mieć więcej argumentów, ale tylko jeden (i to tylko pierwszy) ze słowem kluczowym `this`. Kolejne argumenty przy jej wywoływaniu będą używane jako normalne argumenty metody. Zdefiniujmy dla przykładu rozszerzenie, w którym jako argumentu możemy użyć znaku, jakim w łańcuchu zastąpimy apostrofy (listing 4.29).

Listing 4.29. Poniższa metoda powinna znaleźć się w klasie Rozszerzenia

```
public static string UsunApostrof(this String argument, char zamiennik)
{
    return argument.Replace('\'', zamiennik);
}
```

Możemy jej użyć następująco: `MessageBox.Show("SQL Inje'c'tion".UsunApostrof('#'));`.

Często rozszerzenia pobierają jako argument delegacje do metod (robi tak większość rozszerzeń zdefiniowanych na potrzeby LINQ). Wówczas jako ich argumentów możemy używać wyrażeń lambda opisanych w rozdziale 3. W naszym przypadku rozszerzenie `UsunApostrof` mogłoby potrzebować metody precyzującej, w jaki sposób ma zastępować znaki w łańcuchu. Na początek zdefiniujmy typ delegacji:

```
delegate char DZmieniacz(char znak);
```

Teraz możemy do klasy `Rozszerzenia` dodać kolejną wersję metody `UsunApostrof` (listing 4.30).

Listing 4.30. Rozszerzenie z delegacją jako argumentem

```
public static string UsunApostrof(this String argument, DZmieniacz zmieniacz)
{
   string wynik = "";
   foreach(char znak in argument.ToCharArray())
   wynik += zmieniacz(znak);
   return wynik;
}
```

Wywołanie tak zdefiniowanej metody aż prosi się o wykorzystanie wyrażenia lambda: `MessageBox.Show("SQL Inje'c'tion".UsunApostrof(c=>(c=='\'')?'#':c));`.

Aby sprawę uprościć, możemy deklarację delegacji usunąć, a w sygnaturze nowej metody `UsunApostrof` wykorzystać typ parametryczny `Func`. Jego pierwszym parametrem może być typ argumentu, a drugim typ zwracany przez metodę. Delegacji `DZmieniacz` odpowiada więc konkretyzacja `Func<char,char>`. Dzięki temu sygnatura metody z listingu 4.30 może być zastąpiona przez:

```
public static string UsunApostrof(this String argument, Func<char,char> zmieniacz)
```

O rozszerzeniach należy powiedzieć trzy rzeczy. Pierwsza: metodę rozszerzającą można zdefiniować także dla typu `object` (listing 4.31). A ponieważ rozszerzenia są przejmowane przez klasy potomne, dodalibyśmy w ten sposób metodę do wszystkich typów .NET. Druga: metoda nie musi zwracać wartości — może tylko wykonywać jakąś czynność, np. pokazywać komunikat (druga metoda w listingu 4.31). I trzecia: operatory LINQ, do których omówienia przejdziemy w części trzeciej, zostały zdefiniowane właśnie jako metody rozszerzające.

Listing 4.31. Metody dodane do wszystkich typów

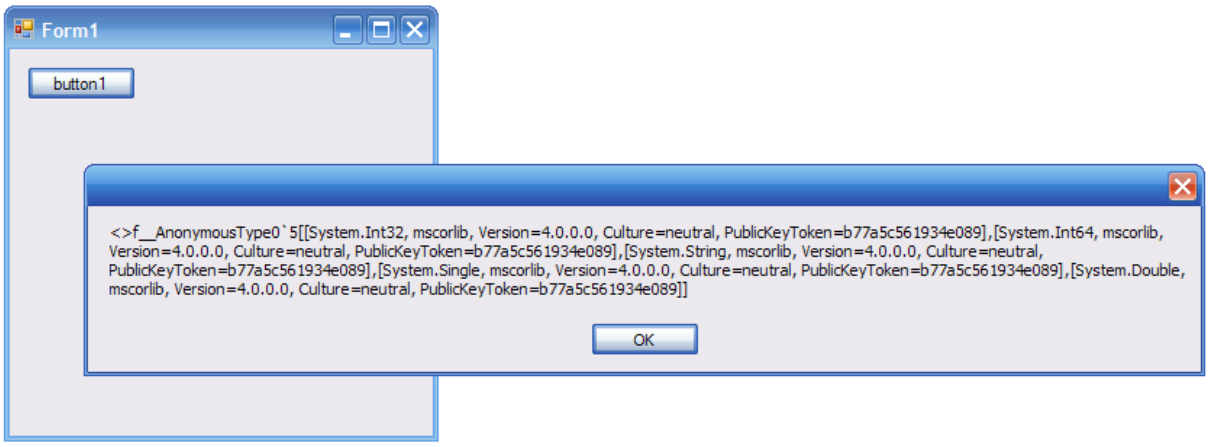
```
public static string TypJakoLancuch(this object argument)
{
   return argument.GetType().FullName;
}

public static void ShowMessage(this object argument)
{
   MessageBox.Show(argument.ToString());
}
```

Typy anonimowe

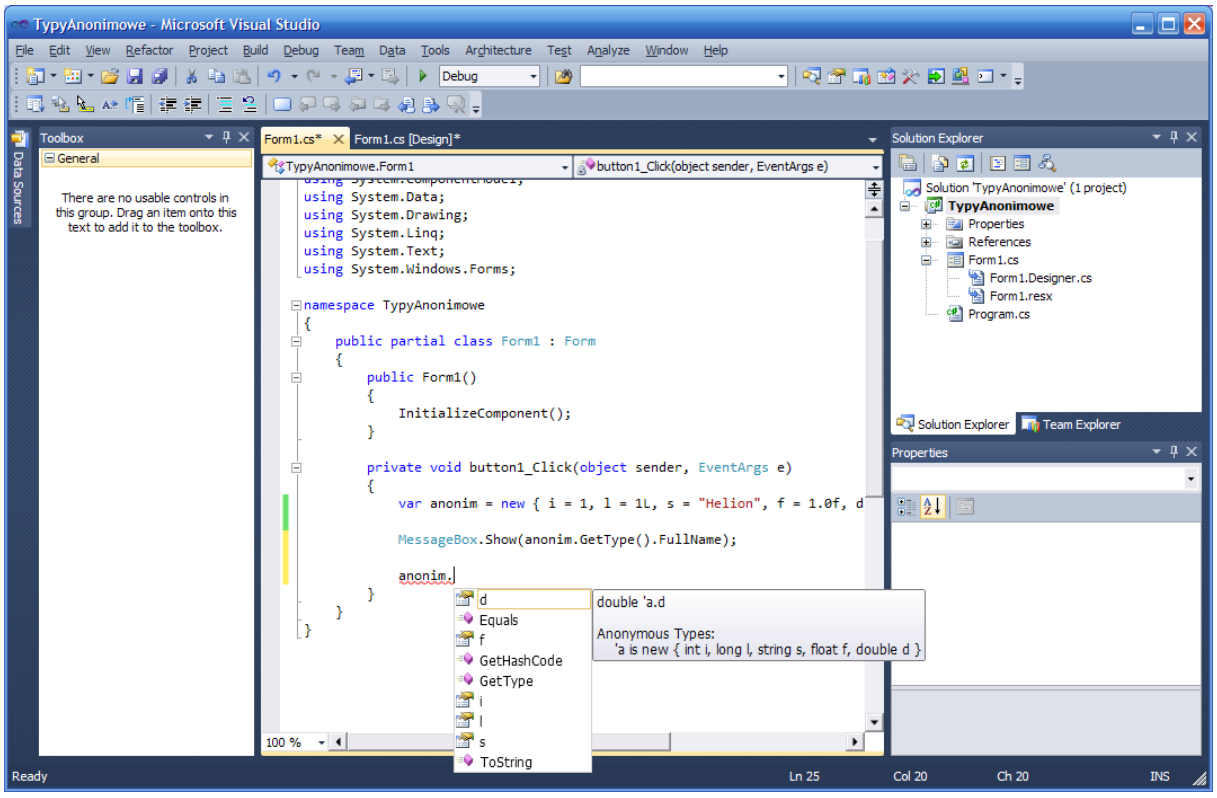
Warto również wspomnieć o jeszcze jednej „innowacji” wprowadzonej w wersji 3.0 języka C#. Umożliwia ona tworzenie obiektów bez definiowania ich typu (klasy lub struktury). Możliwa jest konstrukcja typu:

```
var anonim=new {i=1, l=1L, s="Helion", f=1.0f, d=1.0};
```

W efekcie powstaje typ o pięciu polach publicznych tylko do odczytu, o nazwach `i`, `l`, `s`, `f` i `d`. Typ tworzonego obiektu jest dość złożony (na rysunku 4.1 pokazuję typ zwracany przez polecenie `anonim.GetType().FullName`), ale na szczęście, wyposażeni jesteśmy w słowo kluczowe `var`, które pozwala go nie deklarować. Nowe składowe są poprawnie rozpoznawane przez mechanizm IntelliSense (rysunek 4.2) — nie ma więc kłopotu z korzystaniem z tak utworzonego obiektu. Nowy sposób tworzenia obiektu jest szczególnie wygodny przy zwracaniu danych w zapytaniach LINQ, o czym również przekonamy się w trzeciej części.



Rysunek 4.1. Po kompilacji struktura typu anonimowego jest już znana



Rysunek 4.2. Typy anonimowe są poprawnie rozpoznawane przez narzędzia uzupełniania kodu