

Wydanie V



C# 6.0

Kompletny przewodnik dla praktyków

„Witajcie w owocu współpracy najlepszych autorów, o których można marzyć w świecie książek o języku C# (i nie tylko!)”

*— z przedmowy Madsa Torgersena,
menedżera programu prac nad językiem
C# w Microsoftzie*

MARK MICHAELIS
ERIC LIPPERT

IntelliTect

Helion

Tytuł oryginału: Essential C# 6.0 (5th Edition)

Tłumaczenie: Tomasz Walczak

ISBN: 978-83-283-2518-0

Authorized translation from the English language edition, entitled:
ESSENTIAL C# 6.0, Fifth Edition; ISBN 0134141040; by Mark Michaelis; and Eric Lippert;
published by Pearson Education, Inc. publishing as Addison-Wesley Professional.
Copyright © 2016 Pearson Education, Inc.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

Polish language edition published by HELION SA. Copyright © 2016.

Microsoft, Windows, Visual Basic, Visual C#, and Visual C++ are either registered trademarks or trademarks of Microsoft Corporation in the U.S.A. and/or other countries/regions.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Wydawnictwo HELION

ul. Kościuszki 1c, 44-100 GLIWICE

tel. 32 231 22 19, 32 230 98 63

e-mail: helion@helion.pl

WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

<http://helion.pl/user/opinie/ch6kpp>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:

<ftp://ftp.helion.pl/przyklady/ch6kpp.zip>

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)



Spis treści

Spis rysunków 11
Spis tabel 13
Przedmowa 15
Wprowadzenie 17
Podziękowania 27
O autorach 29

1. Wprowadzenie do języka C# 31

Witaj, świecie 32
Podstawy składni języka C# 33
Dane wejściowe i wyjściowe w konsoli 44

2. Typy danych 59

Podstawowe typy liczbowe 59
Inne podstawowe typy 67
Wartości null i void 78
Kategorie typów 81
Modyfikator umożliwiający stosowanie wartości null 83
Konwersje typów danych 84
Tablice 89

3. Operatory i przepływ sterowania 105

Operatory 106
Zarządzanie przepływem sterowania 119
Bloki kodu ({}), 124
Bloki kodu, zasięgi i przestrzenie deklaracji 126
Wyrażenia logiczne 127
Operatory bitowe (<<, >>, |, &, ^, ~) 135
Instrukcje związane z przepływem sterowania — ciąg dalszy 140
Instrukcje skoku 150
Dyrektywy preprocesora języka C# 156

4. Metody i parametry 165

- Wywoływanie metody 166
- Deklarowanie metody 172
- Dyrektywa using 176
- Zwracane wartości i parametry metody Main() 181
- Zaawansowane parametry metod 183
- Rekurencja 190
- Przeciążanie metod 193
- Parametry opcjonalne 195
- Podstawowa obsługa błędów z wykorzystaniem wyjątków 199

5. Klasy 213

- Deklarowanie klasy i tworzenie jej instancji 216
- Pola instancji 218
- Metody instancji 221
- Stosowanie słowa kluczowego this 222
- Modyfikatory dostępu 228
- Właściwości 230
- Konstruktory 244
- Składowe statyczne 253
- Metody rozszerzające 262
- Hermetyzacja danych 263
- Klasy zagnieżdżone 266
- Klasy częściowe 268

6. Dziedziczenie 273

- Tworzenie klas pochodnych 274
- Przesłanianie składowych z klas bazowych 284
- Klasy abstrakcyjne 294
- Wszystkie klasy są pochodne od System.Object 299
- Sprawdzanie typu za pomocą operatora is 301
- Konwersja z wykorzystaniem operatora as 301

7. Interfejsy 303

- Wprowadzenie do interfejsów 304
- Polimorfizm oparty na interfejsach 305
- Implementacja interfejsu 309
- Przekształcanie między klasą z implementacją i interfejsami 314
- Dziedziczenie interfejsów 315
- Dziedziczenie po wielu interfejsach 317
- Metody rozszerzające i interfejsy 317
- Implementowanie wielodziedziczenia za pomocą interfejsów 319
- Zarządzanie wersjami 321
- Interfejsy a klasy 323
- Interfejsy a atrybuty 324

8. Typy bezpośrednie 327

- Struktury 331
- Opakowywanie 336
- Wyliczenia 343

9. Dobrze uformowane typy 355

- Przesłanianie składowych z klasy object 355
- Przeciążanie operatorów 365
- Wskazywanie innych podzespołów 373
- Definiowanie przestrzeni nazw 377
- Komentarze XML-owe 381
- Odzyskiwanie pamięci 385
- Porządkowanie zasobów 387
- Leniwe inicjowanie 394

10. Obsługa wyjątków 397

- Wiele typów wyjątków 397
- Przechwytywanie wyjątków 400
- Ogólny blok catch 403
- Wskazówki związane z obsługą wyjątków 405
- Definiowanie niestandardowych wyjątków 407
- Ponowne zgłaszanie opakowanego wyjątku 411

11. Typy generyczne 415

- Język C# bez typów generycznych 416
- Wprowadzenie do typów generycznych 420
- Ograniczenia 430
- Metody generyczne 442
- Kowariancja i kontrawariancja 446
- Wewnętrzne mechanizmy typów generycznych 452

12. Delegaty i wyrażenia lambda 457

- Wprowadzenie do delegatów 458
- Wyrażenia lambda 466
- Metody anonimowe 471
- Delegaty ogólnego przeznaczenia — System.Func i System.Action 473

13. Zdarzenia 489

- Implementacja wzorca „obserwator” za pomocą delegatów typu multicast 490
- Zdarzenia 503

14. Interfejsy kolekcji ze standardowymi operatorami kwerend 513

- Typy anonimowe i zmienne lokalne o niejawnie określonym typie 514
- Inicjatory kolekcji 519
- Interfejs IEnumerable<T> sprawia, że klasa staje się kolekcją 522
- Standardowe operatory kwerend 527

- 15. Technologia LINQ i wyrażenia z kwerendami 557**
 - Wprowadzenie do wyrażen z kwerendami 558
 - Wyrażenia z kwerendą to tylko wywołania metod 573
- 16. Tworzenie niestandardowych kolekcji 577**
 - Inne interfejsy implementowane w kolekcjach 578
 - Podstawowe klasy kolekcji 580
 - Udostępnianie indeksera 594
 - Zwracanie wartości null lub pustej kolekcji 598
 - Iteratory 598
- 17. Refleksja, atrybuty i programowanie dynamiczne 613**
 - Mechanizm refleksji 614
 - Operator nameof 623
 - Atrybuty 624
 - Programowanie z wykorzystaniem obiektów dynamicznych 644
- 18. Wielowątkowość 655**
 - Podstawy wielowątkowości 657
 - Używanie klasy System.Threading 663
 - Zadania asynchroniczne 670
 - Anulowanie zadania 686
 - Wzorzec obsługi asynchroniczności za pomocą zadań 692
 - Równoległe wykonywanie iteracji pętli 713
 - Równoległe wykonywanie kwerend LINQ 721
- 19. Synchronizowanie wątków 727**
 - Po co stosować synchronizację? 728
 - Zegary 752
- 20. Współdziałanie między platformami i niezabezpieczony kod 755**
 - Mechanizm P/Invoke 756
 - Wskaźniki i adresy 766
 - Wykonywanie niezabezpieczonego kodu za pomocą delegata 775
 - Używanie bibliotek Windows Runtime w języku C# 776
- 21. Standard CLI 781**
 - Definiowanie standardu CLI 782
 - Implementacje standardu CLI 783
 - Kompilacja kodu w języku C# na kod maszynowy 784
 - Środowisko uruchomieniowe 786
 - Domeny aplikacji 790
 - Podzespoły, manifesty i moduły 790
 - Język Common Intermediate Language 792
 - Common Type System 793
 - Common Language Specification 794
 - Base Class Library 794
 - Metadane 794

A Pobieranie i instalowanie kompilatora języka C# oraz platformy CLI 799

Platforma .NET dla systemu Windows 799

Platforma .NET w systemach OS X i Linux 801

B Kod źródłowy programu do gry w kółko i krzyżyk 803**C Wielowątkowość bez biblioteki TPL i przed wersją C# 6.0 809**

Wzorzec APM 810

Asynchroniczne wywoływanie delegatów 821

Wzorzec EAP — asynchroniczność oparta na zdarzeniach 824

Wzorzec wykorzystujący roboczy wątek działający w tle 827

Kierowanie wywołań do interfejsu użytkownika w systemie Windows 830

**D Zegary przed wprowadzeniem w wersji C# 5.0 słów kluczowych
async i await 835***Skorowidz 841*

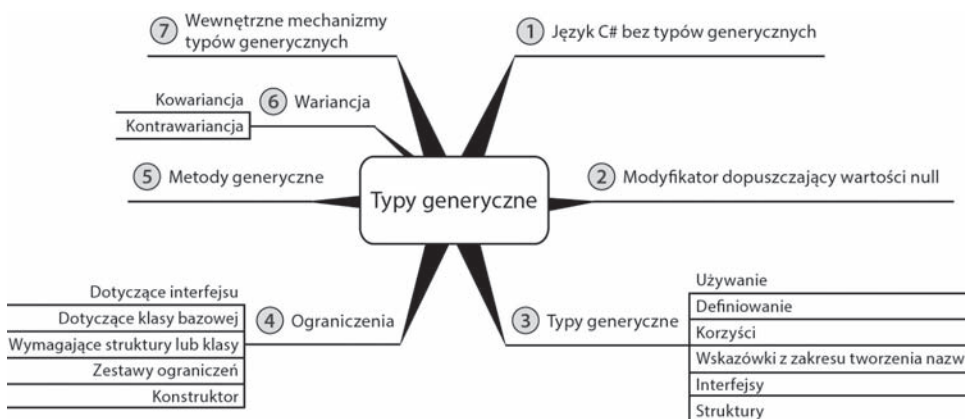
11

Typy generyczne

Początek
2.0

GDY ZACZNIESZ TWORZYĆ BARDZIEJ złożone projekty, będziesz potrzebował lepszego sposobu na ponowne wykorzystywanie i dostosowywanie istniejącego oprogramowania. Aby ułatwić wielokrotne wykorzystanie kodu (a zwłaszcza algorytmów), w języku C# udostępniono mechanizm **typów generycznych**. Podobnie jak metody są bardziej wartościowe, ponieważ mogą przyjmować argumenty, tak typy i metody przyjmujące argumenty określające typ dają dodatkowe możliwości.

Typy generyczne w języku C# są składniowo podobne do typów generycznych z Javy i szablonów z języka C++. We wszystkich trzech wymienionych językach wspomniane mechanizmy umożliwiają jednokrotne zaimplementowanie algorytmów i wzorców. Nie są potrzebne odrębne implementacje dla każdego typu, dla którego dany algorytm lub wzorzec działa. Jednak typy generyczne w języku C# znacznie różnią się od typów generycznych z Javy i szablonów z języka C++, jeśli chodzi o szczegóły implementacji oraz wpływ tych mechanizmów na system typów. Typy generyczne zostały dodane do środowiska uruchomieniowego i języka C# w wersji 2.0.



Język C# bez typów generycznych

W ramach omawiania typów generycznych najpierw przeanalizujemy klasę, w której takie typy nie są używane. Ta klasa, `System.Collections.Stack`, reprezentuje stos, czyli kolekcję obiektów, w której ostatni element dodawany do kolekcji jest pierwszym elementem z niej pobieranym (jest to kolekcja typu „ostatni na wejściu, pierwszy na wyjściu”; ang. *last in, first out* — **LIFO**). Dwie główne metody klasy `Stack`, czyli `Push()` i `Pop()`, dodają elementy do stosu i usuwają je z niego. Deklaracje tych metod z klasy `Stack` znajdują się na listingu 11.1.

Listing 11.1. Sygnatury metod klasy `System.Collections.Stack`

```
public class Stack
{
    public virtual object Pop() { ... }
    public virtual void Push(object obj) { ... }
    // ...
}
```

W programach stos często służy do umożliwiania wielokrotnego cofania operacji. Na przykład na listingu 11.2 kod używa klasy `System.Collections.Stack` do wycofywania operacji w programie symulującym działanie znikopisu.

Listing 11.2. Obsługa wycofywania operacji w programie symulującym działanie znikopisu

```
using System;
using System.Collections;

class Program
{
    // ...

    public void Sketch()
    {
        Stack path = new Stack();
        Cell currentPosition;
        ConsoleKeyInfo key; // Typ dodany w wersji C# 2.0.

        do
        {
            // Wymazywanie w kierunku określonym przez
            // strzałki wciśnięte przez użytkownika.
            key = Move();

            switch (key.Key)
            {
                case ConsoleKey.Z:
                    // Wymazanie ostatnio narysowanego elementu.
                    if (path.Count >= 1)
                    {
                        currentPosition = (Cell)path.Pop();
                        Console.SetCursorPosition(
                            currentPosition.X, currentPosition.Y);
                        Undo();
                    }
                }
            }
        }
    }
}
```

2.0

```

    }
    break;

    case ConsoleKey.DownArrow:
    case ConsoleKey.UpArrow:
    case ConsoleKey.LeftArrow:
    case ConsoleKey.RightArrow:
        // SaveState()
        currentPosition = new Cell(
            Console.CursorLeft, Console.CursorTop);
        path.Push(currentPosition);
        break;

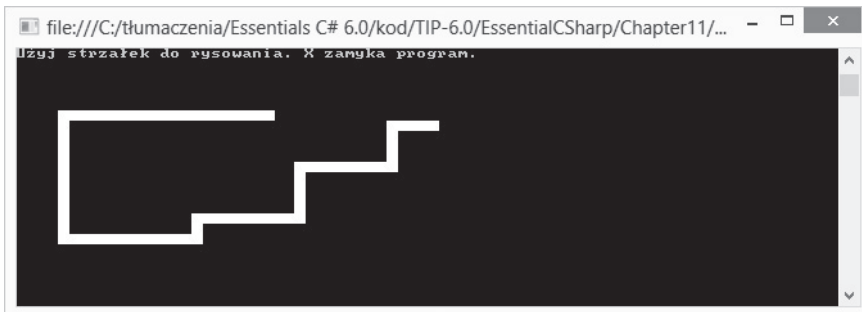
    default:
        Console.Beep(); // Dodane w wersji C# 2.0.
        break;
    }
}
while (key.Key != ConsoleKey.X); // Klawisz X pozwala zamknąć program.
}
}

public struct Cell
{
    // W wersjach starszych niż C# 6.0 należy użyć pola tylko do odczytu.
    public int X { get; }
    public int Y { get; }
    public Cell(int x, int y)
    {
        X = x;
        Y = y;
    }
}

```

Efekt uruchomienia kodu z listingu 11.2 znajdziesz w danych wyjściowych 11.1

DANE WYJŚCIOWE 11.1.



W zmiennej path typu System.Collections.Stack program zachowuje wcześniejsze ruchy pędzla, przekazując element niestandardowego typu Cell do metody Stack.Push() (w wyrażeniu path.Push(currentPosition)). Jeśli użytkownik wpisze literę Z (lub wybierze kombinację Ctrl+Z), poprzedni ruch pędzla zostaje anulowany. Anulowanie odbywa się przez

zdjęcie poprzedniego ruchu pędzla ze stosu za pomocą metody `Pop()`, przeniesienie pozycji kursora na wcześniejszą pozycję i wywołanie metody `Undo()`.

Choć ten kod działa, klasa `System.Collections.Stack` ma ważną wadę. Na listingu 11.1 pokazano, że klasa `Stack` przechowuje wartości typu `object`. Ponieważ każdy obiekt w środowisku CLR jest typu pochodnego od klasy `object`, klasa `Stack` nie sprawdza, czy elementy umieszczane w kolekcji są tego samego i odpowiedniego typu. Na przykład zamiast przekazywać zmienną `currentPosition`, możesz przekazać łańcuch znaków zawierający współrzędne `X` i `Y` połączone kropką. Kompilator musi zezwalać na zapis wartości niespójnych typów danych, ponieważ klasa `Stack` przyjmuje dowolny obiekt pochodny od klasy `object`. Specyficzny typ obiektu nie ma tu znaczenia.

2.0

Ponadto po pobraniu (za pomocą metody `Pop()`) danych ze stosu należy zrzutować zwróconą wartość na typ `Cell`. Jeśli jednak typ wartości zwróconej przez metodę `Pop()` jest różny od `Cell`, kod zgłosi wyjątek. Rzutowanie opóźnia sprawdzanie typu do czasu wykonywania programu, przez co program jest bardziej narażony na błędy. Podstawowy problem z tworzeniem (bez używania typów generycznych) klas, które mają obsługiwać różne typy danych, polega na tym, że typy te muszą działać dla wspólnej klasy bazowej lub wspólnego interfejsu. Zwykle tą wspólną klasą jest klasa `object`.

Używanie w klasach wykorzystujących klasę `object` typów bezpośrednich, na przykład struktur lub liczb całkowitych, dodatkowo nasila problem. Jeśli do metody `Stack.Push()` przekażesz wartość typu bezpośredniego, środowisko uruchomieniowe automatycznie opakuje tę wartość. W trakcie pobierania wartości typu bezpośredniego trzeba jawnie wypakować dane i zrzutować referencję do obiektu typu `object` (pobraną za pomocą metody `Pop()`) na typ bezpośredni. Rzutowanie typu referencyjnego na klasę bazową lub interfejs nie ma dużego wpływu na wydajność kodu, jednak operacja opakowywania typu bezpośredniego wiąże się z większymi kosztami, ponieważ trzeba przydzielić pamięć, skopiować wartość, a później odzyskać pamięć.

C# to język ułatwiający zachowanie bezpieczeństwa ze względu na typ. Język ten zaprojektowano w taki sposób, by wiele błędów związanych z typami (takich jak przypisanie liczby całkowitej do zmiennej typu `string`) było wykrywanych na etapie kompilacji. Problem polega na tym, że klasa `Stack` nie jest tak bezpieczna ze względu na typ, jak można tego oczekiwać po programach w języku C#. Aby zmodyfikować tę klasę i wymusić, by elementy stosu były określonego typu (jednak bez stosowania typów generycznych), należy utworzyć wyspecjalizowaną wersję klasy, przedstawioną na listingu 11.3.

Listing 11.3. Definicja wyspecjalizowanej wersji klasy `Stack`

```
public class CellStack
{
    public virtual Cell Pop();
    public virtual void Push(Cell cell);
    // ...
}
```

Ponieważ klasa `CellStack` może przechowywać tylko obiekty typu `Cell`, to rozwiązanie wymaga dodania niestandardowej implementacji metod potrzebnych do obsługi stosu, co

nie jest wygodne. Utworzenie bezpiecznego ze względu na typ stosu liczb całkowitych wymaga następnej niestandardowej implementacji, a każda z nich jest bardzo podobna do wszystkich pozostałych. To skutkuje powstaniem dużej ilości powtarzającego się nadmiarowego kodu.

2.0

■ ZAGADNIENIE DLA POCZĄTKUJĄCYCH

Inny przykład — typy bezpośrednie z możliwą wartością null

W rozdziale 2. opisano możliwość zadeklarowania zmiennych, które mogą zawierać wartość null. Wymaga to użycia modyfikatora ? w deklaracji zmiennej typu bezpośredniego. Ta możliwość pojawiła się w wersji C# 2.0, ponieważ potrzebne były do tego typy generyczne. Przed ich wprowadzeniem programiści mieli do wyboru dwa rozwiązania.

Pierwsze z nich polegało na zadeklarowaniu typów danych z obsługą wartości null. Potrzebny był jeden taki typ dla każdego typu bezpośredniego, który miał przyjmować wartości null. Kilka takich typów pokazano na listingu 11.4.

Listing 11.4. Deklarowanie wersji różnych typów bezpośrednich z dodaną obsługą wartości null

```

struct NullableInt
{
    /// <summary>
    /// Udostępnia wartość, jeśli metoda HasValue zwraca true.
    /// </summary>
    public int Value{ get; private set; }

    /// <summary>
    /// Określa, czy wartość jest dostępna, czy jest równa null.
    /// </summary>
    public bool HasValue{ get; private set; }

    // ...
}

struct NullableGuid
{
    /// <summary>
    /// Udostępnia wartość, jeśli metoda HasValue zwraca true.
    /// </summary>
    public Guid Value{ get; private set; }

    /// <summary>
    /// Określa, czy wartość jest dostępna, czy jest równa null.
    /// </summary>
    public bool HasValue{ get; private set; }

    ...
}
...

```

Na listingu 11.4 pokazano możliwe implementacje typów NullableInt i NullableGuid. Jeśli w programie potrzebne są dodatkowe typy bezpośrednie z obsługą wartości null, trzeba utworzyć nową strukturę z właściwościami działającymi dla odpowiedniego typu. Każdą poprawkę

2.0

w implementacji (na przykład dodanie zdefiniowanej przez użytkownika konwersji niejawnej z danego typu na jego odpowiednik obsługujący wartość null) wymaga wtedy zmodyfikowania deklaracji wszystkich typów.

Druga strategia implementowania typu z obsługą wartości null bez typów generycznych polega na utworzeniu jednego typu z właściwością `Value` typu `object`. To rozwiązanie pokazano na listingu 11.5.

Listing 11.5. Deklarowanie typu z obsługą wartości null, zawierającego właściwość `Value` typu `object`

```
struct Nullable
{
    /// <summary>
    /// Udostępnia wartość, jeśli metoda HasValue zwraca true.
    /// </summary>
    public object Value{ get; private set; }

    /// <summary>
    /// Określa, czy wartość jest dostępna, czy jest równa null.
    /// </summary>
    public bool HasValue{ get; private set; }

    ...
}
```

Choć ta technika wymaga utworzenia tylko jednej implementacji typu z obsługą wartości null, środowisko uruchomieniowe zawsze opakowuje wtedy typy bezpośrednie, gdy ustalana jest wartość właściwości `Value`. Ponadto pobieranie wartości z tej właściwości wymaga rzutowania, którego wynik w czasie wykonywania programu może się okazać nieprawidłowy.

Żadne z tych rozwiązań nie jest atrakcyjne. Aby wyeliminować ten problem, w wersji C# 2.0 dodano typy generyczne. Typy z obsługą wartości null mają teraz postać typu generycznego `Nullable<T>`.

Wprowadzenie do typów generycznych

Typy generyczne zapewniają mechanizm tworzenia struktur danych, które można przekształcić na wyspecjalizowaną wersję w celu obsługi konkretnych typów. Programiści definiują **typy parametryzowane** w taki sposób, by dla każdej zmiennej określonego typu generycznego używany był ten sam wewnętrzny algorytm. Jednak typy danych i sygnatury metod mogą się zmieniać w zależności od podanego argumentu określającego typ.

Aby ułatwić programistom naukę, projektanci języka C# zdecydowali się na zastosowanie składni pozornie podobnej do składni szablonów z języka C++. W C# składnia tworzenia klas i struktur generycznych wymaga użycia nawiasów ostrych do deklarowania parametrów w deklaracji typu i do podawania argumentów, gdy typ jest używany.

Używanie klasy generycznej

Na listingu 11.6 pokazano, jak w klasie generycznej podać argument określający typ. W kodzie zmienna `path` jest tworzona jako stos obiektów typu `Cell`. W tym celu typ `Cell` jest podawany w nawiasie ostrym zarówno w wyrażeniu tworzącym obiekt, jak i w deklaracji zmiennej. Oznacza to, że gdy deklarujesz zmienną (tu jest to zmienna `path`) typu generycznego, C# wymaga, by podać argument określający typ używany przez dany typ generyczny. Na listingu 11.6 pokazano ten proces na przykładzie nowej generycznej klasy `Stack`.

Listing 11.6. Implementowanie wycofywania operacji za pomocą generycznej klasy `Stack`

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

class Program
{
    // ...

    public void Sketch()
    {
        Stack<Cell> path;           // Deklaracja zmiennej typu generycznego.
        path = new Stack<Cell>(); // Tworzenie obiektu typu generycznego.
        Cell currentPosition;
        ConsoleKeyInfo key;

        do
        {
            // Rysowanie kreski w kierunku określonym przez
            // strzałkę wciśniętą przez użytkownika.
            key = Move();

            switch (key.Key)
            {
                case ConsoleKey.Z:
                    // Cofnięcie poprzedniego ruchu pędzla.
                    if (path.Count >= 1)
                    {
                        // Rzutowanie nie jest potrzebne.
                        currentPosition = path.Pop();
                        Console.SetCursorPosition(
                            currentPosition.X, currentPosition.Y);
                        Undo();
                    }
                    break;

                case ConsoleKey.DownArrow:
                case ConsoleKey.UpArrow:
                case ConsoleKey.LeftArrow:
                case ConsoleKey.RightArrow:
                    // SaveState()
                    currentPosition = new Cell(
                        Console.CursorLeft, Console.CursorTop);
                    // W wywołaniu Push() można używać tylko zmiennych typu Cell.
                    path.Push(currentPosition);
                    break;
            }
        }
    }
}
```

2.0

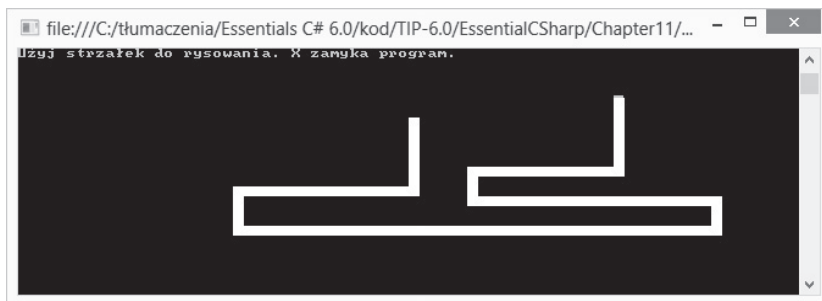
```

        default:
            Console.Beep(); // Metoda dodana w wersji C# 2.0.
            break;
    }
} while (key.Key != ConsoleKey.X); // Klavisz X pozwala zamknąć aplikację.
}
}

```

Wynik działania kodu z listingu 11.6 pokazano w danych wyjściowych 11.2.

DANE WYJŚCIOWE 11.2.



2.0

Na listingu 11.6 deklarowana jest zmienna `path` inicjowana nowym obiektem klasy `System.Collections.Generic.Stack<Cell>`. W nawiasie ostrym podano, że typ danych elementów stosu to `Cell`. Dlatego każdy obiekt dodawany do zmiennej `path` i z niej pobierany jest typu `Cell`. Nie trzeba więc rzutować wartości zwracanej przez metodę `path.Pop()` ani samodzielnie zapewniać, że tylko obiekty typu `Cell` są dodawane za pomocą metody `Push()` do zmiennej `path`.

Definiowanie prostej klasy generycznej

Typy generyczne umożliwiają tworzenie algorytmów i wzorców oraz ponowne wykorzystanie napisanego kodu dla innych typów danych. Na listingu 11.7 tworzona jest klasa `Stack<T>`, podobna do klasy `System.Collections.Generic.Stack<T>` użytej na listingu 11.6. **Parametr określający typ** (`T`) należy podać w nawiasie ostrym po nazwie klasy. Później do generycznego typu `Stack<T>` można przekazać jeden argument określający typ, podstawiany wszędzie tam, gdzie w klasie występuje `T`. Dzięki temu stos może przechowywać elementy dowolnego podanego typu. Nie wymaga to duplikowania kodu ani konwersji elementów na typ `object`. Parametr `T` (określający typ) to symbol zastępczy, który należy zastąpić argumentem określającym typ. Na listingu 11.7 parametr określający typ jest używany w wewnętrznej tablicy `Items`, w parametrze metody `Push()` i w wartości zwracanej przez metodę `Pop()`.

Listing 11.7. Deklarowanie generycznej klasy `Stack<T>`

```

public class Stack<T>
{
    // W wersjach starszych niż C# 6.0 należy zastosować pole tylko do odczytu.
    private T[] InternalItems { get; }
}

```



```

public void Push(T data)
{
    ...
}

public T Pop()
{
    ...
}
}

```

Zalety typów generycznych

2.0

Stosowanie klas generycznych zamiast ich standardowych odpowiedników (na przykład użytej wcześniej klasy `System.Collections.Generic.Stack<T>` zamiast jej pierwowzoru `System.Collections.Stack`) daje kilka korzyści.

1. Typy generyczne pozwalają zwiększyć bezpieczeństwo ze względu na typ. Uniemożliwiają stosowanie typów innych niż typy jawnie określone dla składowych parametryzowanej klasy. Na listingu 11.7 reprezentująca stos parametryzowana klasa `Stack<Cell>` pozwala stosować tylko typ `Cell`. Na przykład instrukcja `path.Push("garbage")` spowoduje błąd kompilacji informujący, że nie istnieje wersja przeciążonej metody `System.Collections.Generic.Stack<T>.Push(T)` działająca na łańcuchach znaków, ponieważ łańcucha nie można przekszttałcić na typ `Cell`.
2. Sprawdzanie typów na etapie kompilacji zmniejsza prawdopodobieństwo wystąpienia wyjątków typu `InvalidCastException` w czasie wykonywania programu.
3. Używanie typów bezpośrednich w składowych klasy generycznej nie powoduje opakowywania wartości tych typów w typ `object`. Na przykład metody `path.Pop()` i `path.Push()` nie wymagają opakowania elementu w momencie dodawania go i wypakowywania w trakcie usuwania.
4. Typy generyczne w języku C# zmniejszają ilość kodu. Pozwalają zachować korzyści, jakie dają specyficzne wersje klasy, ale nie powodują analogicznych kosztów. Nie trzeba na przykład definiować nowej klasy `CellStack`.
5. Wydajność kodu rośnie, ponieważ nie jest potrzebne rzutowanie z typu `object`. Eliminuje to operację sprawdzania typu. Inną przyczyną wzrostu wydajności jest to, że nie trzeba opakowywać wartości typów bezpośrednich.
6. Typy generyczne zmniejszają ilość zajmowanej pamięci, ponieważ nie trzeba opakowywać wartości. Dzięki temu program zużywa mniej pamięci na stercie.
7. Kod staje się bardziej czytelny, ponieważ jest w nim mniej operacji sprawdzania typów przy rzutowaniu i mniej implementacji specyficznych typów.
8. Edytory wspomagające pisanie kodu za pomocą jednej z odmian mechanizmu *IntelliSense* bezpośrednio obsługują wartości zwracane przez klasy generyczne. Nie trzeba rzutować zwracanych danych, aby używać mechanizmu *IntelliSense*.

Istotą typów generycznych jest umożliwienie implementowania wzorców i wielokrotne wykorzystywanie tych implementacji wszędzie tam, gdzie dany wzorec jest potrzebny. Wzorce opisują problemy, które często występują w kodzie. Szablony zapewniają jedno rozwiązanie dla tych powtarzających się wzorców.

2.0

Wskazówki związane z tworzeniem nazw parametrów określających typy

Podobnie jak nazwy parametrów formalnych metod, tak i nazwy parametrów określających typ powinny być jak najbardziej opisowe. Ponadto aby podkreślić, że dany parametr określa typ, nazwę takiego parametru należy poprzedzić literą T. Na przykład w definicji klasy `Entity` `ICollection<TEntity>` nazwa parametru określającego typ to `TEntity`.

Z opisowych nazw można zrezygnować w jednej sytuacji — wtedy, gdy nie dodają żadnej wartości. Na przykład użycie samej litery T w nazwie klasy `Stack<T>` jest odpowiednie, ponieważ informacja, że T to parametr określający typ, jest wystarczająco opisowa. Stos działa dla obiektów dowolnego typu.

W następnym podrozdziale zapoznasz się z ograniczeniami. Dobrą praktyką jest używanie nazw opisujących ograniczenia. Na przykład jeśli jako parametr trzeba podać typ z implementacją interfejsu `IComponent`, możesz nazwać ten parametr `TComponent`.

Wskazówki

STOSUJ opisowe nazwy parametrów określających typ i poprzedzaj te nazwy literą T.

ROZWAŻ podanie ograniczenia w nazwie parametru określającego typ.

Generyczne interfejsy i struktury

C# obsługuje stosowanie typów generycznych w różnych konstrukcjach języka, w tym w interfejsach i strukturach. Składnia jest tu identyczna jak dla klas. Aby zadeklarować interfejs z parametrem określającym typ, umieść ten parametr w nawiasie ostrym bezpośrednio po nazwie interfejsu. Pokazano to w przykładowym interfejsie `IPair<T>` na listingu 11.8.

Listing 11.8. Deklarowanie generycznego interfejsu

```
interface IPair<T>
{
    T First { get; set; }
    T Second { get; set; }
}
```

2.0

Ten interfejs reprezentuje parę podobnych obiektów (na przykład współrzędnych punktu, biologicznych rodziców danej osoby lub węzłów w drzewie binarnym). Oba elementy w parze są tego samego typu.

Aby zaimplementować ten interfejs, należy zastosować taką samą składnię jak w klasach niegenerycznych. Zauważ, że dozwolone (i często spotykane) jest użycie argumentu określającego typ z jednego typu generycznego także w innym typie. Taka sytuacja ma miejsce na listingu 11.9. Argument określający typ dla interfejsu jest jednocześnie argumentem określającym typ w strukturze. W tym przykładzie zamiast klasy zastosowano właśnie strukturę, co jest dowodem na to, że C# umożliwia tworzenie niestandardowych generycznych typów bezpośrednich.

Listing 11.9. Implementowanie generycznego interfejsu

```
public struct Pair<T>: IPair<T>
{
    public T First { get; set; }
    public T Second { get; set; }
}
```

Obsługa generycznych interfejsów jest ważna zwłaszcza w klasach reprezentujących kolekcje. To właśnie w takich klasach najczęściej używa się typów generycznych. Przed wprowadzeniem takich typów programiści musieli posługiwać się zestawem interfejsów z przestrzeni nazw `System.Collections`. Te interfejsy (podobnie jak klasy z ich implementacjami) działały tylko dla typu `object`, dlatego dostęp do elementów z takich klas zawsze wymagał rzutowania. Dzięki zastosowaniu generycznych interfejsów bezpiecznych ze względu na typ można uniknąć rzutowania.

■ ZAGADNIENIE DLA ZAAWANSOWANYCH

Wielokrotne implementowanie jednego interfejsu w tej samej klasie

Dwie deklaracje tego samego generycznego interfejsu są uznawane za różne typy. Dlatego ten sam generyczny interfejs można wielokrotnie zaimplementować w jednej klasie lub strukturze. Przyjrzyj się przykładowi z listingu 11.10.

Listing 11.10. Wielokrotne implementowanie interfejsu w jednej klasie

```
public interface IContainer<T>
{
    ICollection<T> Items { get; set; }
    {
        get;
        set;
    }
}

public class Person: IContainer<Address>,
    IContainer<Phone>, IContainer<Email>
{
    ICollection<Address> IContainer<Address>.Items
    {
        get{...}
        set{...}
    }
}
```

```

ICollection<Phone> IContainer<Phone>.Items
{
    get{...}
    set{...}
}
ICollection<Email> IContainer<Email>.Items
{
    get{...}
    set{...}
}
}

```

W tym przykładzie właściwość `Items` pojawia się wielokrotnie w jawnie zaimplementowanych interfejsach z różnymi parametrami określającymi typ. Bez typów generycznych to rozwiązanie byłoby niemożliwe. Kompilator umożliwiłby jawne zaimplementowanie tylko jednej właściwości `IContainer.Items`.

Jednak technikę implementowania wielu wersji „tego samego” interfejsu wiele osób uznaje za złą praktykę, ponieważ może utrudniać zrozumienie kodu (zwłaszcza gdy interfejs pozwala na konwersje kowariantne lub kontrawariantne). Ponadto klasę `Person` można uznać za źle zaprojektowaną. Normalnie nie uważamy osoby za „coś, co może udostępniać zestaw adresów e-mail”. Jeśli czujesz pokusę zaimplementowania w klasie trzech wersji tego samego interfejsu, pomyśl, czy nie lepiej będzie zamiast tego zaimplementować trzech właściwości, na przykład `EmailAddresses`, `PhoneNumbers` i `MailingAddresses`, z których każda zwraca odpowiednią implementację generycznego interfejsu.

Wskazówka

UNIKAJ implementowania wielu wersji tego samego generycznego interfejsu w jednym typie.

2.0

Definiowanie konstruktora i finalizatora

Zaskoczeniem może się okazać to, że konstruktory (i finalizator) klasy lub struktury generycznej nie wymagają parametrów określających typ. Oznacza to, że zapis `Pair<T>(){...}` nie jest konieczny. W klasie `Pair` na listingu 11.11 konstruktor jest zadeklarowany z sygnaturą `public Pair(T first, T second)`.

Listing 11.11. Deklarowanie konstruktora typu generycznego

```

public struct Pair<T>: IPair<T>
{
    public Pair(T first, T second)
    {
        First = first;
        Second = second;
    }
}

```

```
public T First { get; set; }
public T Second { get; set; }
}
```

Określanie wartości domyślnej

Na listingu 11.11 występuje konstruktor, który przyjmuje początkowe wartości właściwości `First` i `Second` oraz przypisuje je do pól `_First` i `_Second`. Ponieważ typ `Pair<T>` to struktura, konstruktor musi inicjować wszystkie jej pola. Prowadzi to jednak do problemu. Pomyśl o konstruktorze z typu `Pair<T>`, który w trakcie tworzenia obiektu inicjuje tylko jeden element z pary.

Zdefiniowanie takiego konstruktora, pokazanego na listingu 11.12, prowadzi do błędu kompilacji, ponieważ pole `_Second` po zakończeniu pracy konstruktora wciąż nie jest zainicjowane. Zainicjowanie pola `_Second` sprawia trudność, ponieważ typ danych `T` nie jest znany. Jeśli używany jest typ referencyjny, można ustawić wartość `null`, jednak to rozwiązanie nie zadziała, jeżeli `T` to typ bezpośredni, który nie obsługuje wartości `null`.

Listing 11.12. Jeśli nie wszystkie pola zostaną zainicjowane, wystąpi błąd kompilacji

```
public struct Pair<T>: IPair<T>
{
    // BŁĄD: Do pola 'Pair<T>._second' trzeba przypisać
    // wartość przed wyjściem sterowania poza konstruktor.
    // public Pair(T first)
    // {
    //     First = first;
    // }
    // ...
}
```

2.0

Aby umożliwić rozwiązanie tego problemu, w języku C# udostępniono operator `default`, opisany wcześniej w rozdziale 8., gdzie wyjaśniono, że wartość domyślną typu `int` można podać za pomocą wyrażenia `default(int)`. Gdy używany jest typ `T` (potrzebny w polu `_Second`), można podać wartość `default(T)`. Tę technikę zastosowano na listingu 11.13.

Listing 11.13. Inicjowanie pola za pomocą operatora `default`

```
public struct Pair<T>: IPair<T>
{
    public Pair(T first)
    {
        First = first;
        Second = default(T);
    }
    // ...
}
```

Operator `default` pozwala podać wartość domyślną dowolnego typu (także podanego za pomocą parametru).

Wiele parametrów określających typ

W typach generycznych można zadeklarować dowolną liczbę parametrów określających typ. W przedstawionym na początku typie `Pair<T>` występował tylko jeden taki parametr. Aby umożliwić zapis niejednorodnej pary obiektów, na przykład pary nazwa-wartość, możesz utworzyć nową wersję tego typu, obejmującą dwa parametry określające typ. To rozwiązanie pokazano na listingu 11.14.

Listing 11.14. Deklarowanie typu generycznego z kilkoma parametrami określającymi typ

```
interface IPair<TFirst, TSecond>
{
    TFirst First { get; set; }
    TSecond Second { get; set; }
}

public struct Pair<TFirst, TSecond>: IPair<TFirst, TSecond>
{
    public Pair(TFirst first, TSecond second)
    {
        First = first;
        Second = second;
    }

    public TFirst First { get; set; }
    public TSecond Second { get; set; }
}
```

2.0

Gdy używasz klasy `Pair<TFirst, TSecond>`, powinieneś podać oba parametry określające typ w nawiasie ostrym w deklaracji i przy inicjowaniu obiektu. Następnie należy stosować właściwe typy dla parametrów metod w ich wywołaniach. To podejście pokazano na listingu 11.15.

Listing 11.15. Używanie typu z kilkoma parametrami określającymi typ

```
Pair<int, string> historicalEvent =
    new Pair<int, string>(1914,
        "Shackleton wyrusza na Biegun Północny na statku Endurance");
Console.WriteLine("{0}: {1}",
    historicalEvent.First, historicalEvent.Second);
```

Liczba parametrów określających typ (czyli **arność**) jednoznacznie odróżnia daną klasę od innych klas o tej samej nazwie. Można więc w jednej przestrzeni nazw zdefiniować klasy `Pair<T>` i `Pair<TFirst, TSecond>`, ponieważ mają różną arność. Ponadto z powodu podobnego działania typy generyczne różniące się tylko arnością należy umieszczać w tych samych plikach z kodem w języku C#.

Wskazówka

UMIESZCZAJ w jednym pliku klasy generyczne różniące się tylko liczbą parametrów określających typ.

Różna arność

W wersji C# 4.0 zespół odpowiedzialny za środowisko CLR zdefiniował dziewięć nowych typów generycznych. Wszystkie te typy reprezentują krotki i noszą nazwę `Tuple`. W typach `Tuple`, podobnie jak w typach `Pair<...>`, można wielokrotnie wykorzystać tę samą nazwę, ponieważ typy te różnią się arnością (każda wersja ma inną liczbę parametrów określających typ). Te typy pokazano na listingu 11.16.

Początek
4.0

2.0

Listing 11.16. Wykorzystanie arności do przeciążenia definicji typu

```
public class Tuple { ... }
public class Tuple<T1>:
    IStructuralEquatable, IStructuralComparable, IComparable {...}
public class Tuple<T1, T2>: ... {...}
public class Tuple<T1, T2, T3>: ... {...}
public class Tuple<T1, T2, T3, T4>: ... {...}
public class Tuple<T1, T2, T3, T4, T5>: ... {...}
public class Tuple<T1, T2, T3, T4, T5, T6>: ... {...}
public class Tuple<T1, T2, T3, T4, T5, T6, T7>: ... {...}
public class Tuple<T1, T2, T3, T4, T5, T6, T7, TRest>: ... {...}
```

Zestaw klas `Tuple<...>` zaprojektowano w tym samym celu co typy `Pair<T>` i `Pair<TFirst, TSecond>`, jednak klasy `Tuple` umożliwiają obsługę do siedmiu argumentów określających typ. W ostatniej wersji klasy `Tuple` z listingu 11.16 parametr `TRest` można wykorzystać do zapisania następnego obiektu typu `Tuple`. Dlatego liczba elementów w krotce jest potencjalnie nieskończona.

Inną ciekawą składową rodziny klas `Tuple` jest niegeneryczna klasa `Tuple`. Ta klasa ma osiem statycznych metod fabrycznych, które służą do tworzenia obiektów różnych generycznych typów `Tuple`. Choć każdy typ generyczny umożliwia bezpośrednie tworzenie obiektów za pomocą konstruktora, metody fabryczne z klasy `Tuple` automatycznie wykrywają typy argumentów. Różnicę między bezpośrednimi wywołaniami a używaniem metod fabrycznych pokazano na listingu 11.17.

Listing 11.17. Używanie metod fabrycznych `Create()` z klasy `Tuple`

```
Tuple<string, Contact> keyValuePair;
keyValuePair =
    Tuple.Create(
        "555-55-5555", new Contact("Inigo Montoya"));
keyValuePair =
    new Tuple<string, Contact>(
        "555-55-5555", new Contact("Inigo Montoya"));
```

Gdy liczba elementów w obiektach typu `Tuple` jest duża, podawanie wszystkich argumentów określających typ jest kłopotliwe. Łatwiej zastosować wówczas metody fabryczne `Create()`.

Na podstawie tego, że w bibliotece platformy zadeklarowanych jest osiem różnych generycznych typów `Tuple`, można się domyślić, iż w systemie plików środowiska CLR nie są obsługiwane typy generyczne o różnej liczbie parametrów. Metody mogą przyjmować dowolną liczbę argumentów za pomocą tablic z parametrami, nie istnieje jednak analogiczna technika dla typów generycznych. Każdy typ generyczny musi mieć ściśle określoną arność.

Koniec
4.0

Zagnieżdżone typy generyczne

Parametry określające typ w typie generycznym są automatycznie kaskadowo przekazywane w dół do typów zagnieżdżonych. Na przykład jeśli w danym typie zadeklarowany jest określający typ parametr `T`, wszystkie typy zagnieżdżone też będą generyczne, a parametr `T` również będzie w nich dostępny. Jeżeli typ zagnieżdżony ma własny określający typ parametr `T`, spowoduje on ukrycie parametru z nadrzędnego typu. Wtedy wszystkie referencje do parametru `T` w typie zagnieżdżonym będą dotyczyły parametru właśnie z tego typu. Na szczęście ponowne użycie w typie zagnieżdżonym określającego typ parametru o wykorzystanej już nazwie powoduje, że kompilator wyświetla ostrzeżenie. Zapobiega to przypadkowemu użyciu parametrów o tej samej nazwie (zobacz listing 11.18).

Listing 11.18. Zagnieżdżone typy generyczne

```
class Container<T, U>
{
    // Klasy zagnieżdżone dziedziczą parametry określające typ.
    // Ponowne wykorzystanie nazwy takiego parametru
    // prowadzi do zgłoszenia ostrzeżenia.
    class Nested<U>
    {
        void Method(T param0, U param1)
        {
        }
    }
}
```

Określające typ parametry z typu nadrzędnego są dostępne w typie zagnieżdżonym w ten sam sposób jak składowe typu nadrzędnego. Reguła jest prosta — parametr określający typ jest dostępny wszędzie wewnątrz typu, w jakim go zadeklarowano.

Wskazówka

UNIKAJ ukrywania określającego typ parametru z typu nadrzędnego przez tworzenie parametru o identycznej nazwie w typie zagnieżdżonym.

Ograniczenia

Typy generyczne umożliwiają definiowanie ograniczeń dotyczących parametrów określających typ. Te ograniczenia gwarantują, że typy podane jako argumenty będą zgodne z wymaganymi regułami. Przyjrzyj się przykładowej klasie `BinaryTree<T>` z listingu 11.19.

Listing 11.19. Deklaracja klasy `BinaryTree<T>` bez ograniczeń

```
public class BinaryTree<T>
{
    public BinaryTree ( T item)
    {
        Item = item;
    }
}
```



```

}

public T Item { get; set; }
public Pair<BinaryTree<T>> SubItems { get; set; }
}

```

Ciekawostką jest to, że w klasie `BinaryTree<T>` wewnętrznie używany jest typ `Pair<T>`. Jest to dopuszczalne, ponieważ `Pair<T>` to zwykły inny typ.

Załóżmy, że chcesz, by drzewo sortowało wartości w obiekcie typu `Pair<T>` przypisywanym do właściwości `SubItems`. Aby posortować dane, akcesor `set` właściwości `SubItems` używa metody `CompareTo()` z podanego klucza. Ilustruje to listing 11.20.

Listing 11.20. Do działania interfejsu potrzebny jest parametr określający typ

```

public class BinaryTree<T>
{
    public T Item { get; set; }
    public Pair<BinaryTree<T>> SubItems
    {
        get{ return _SubItems; }
        set
        {
            IComparable<T> first;
            // BŁĄD: nie można przeprowadzić niejawną konwersji.
            first = value.First; // Konieczne jest jawne rzutowanie.

            if (first.CompareTo(value.Second) < 0)
            {
                // Wartość właściwości First jest mniejsza niż właściwości Second.
                // ...
            }
            else
            {
                // Wartości właściwości First i Second są takie same
                // lub wartość właściwości Second jest mniejsza.
                // ...
            }
            _SubItems = value;
        }
    }
    private Pair<BinaryTree<T>> _SubItems;
}

```

2.0

W trakcie kompilacji określający typ parametr `T` jest generyczny i nie obowiązują dla niego ograniczenia. Gdy kod wygląda tak jak na listingu 11.20, kompilator przyjmuje, że typ `T` zawiera jedynie składowe odziedziczone po typie bazowym `object`. Można tak przyjąć, ponieważ `object` jest klasą bazową wszystkich typów. Dlatego dla obiektów typu `T` można wywoływać tylko takie metody jak `ToString()`. W efekcie kompilator wyświetla błąd kompilacji, ponieważ w typie `object` nie zdefiniowano metody `CompareTo()`.

By uzyskać dostęp do metody `CompareTo()`, parametr `T` można rzutować na interfejs `IComparable<T>`. Ilustruje to listing 11.21.

Listing 11.21. Parametr określający typ musi być zgodny z interfejsem; w przeciwnym razie wystąpi wyjątek

```
public class BinaryTree<T>
{
    public T Item { get; set; }
    public Pair<BinaryTree<T>> SubItems
    {
        get{ return _SubItems; }
        set
        {
            IComparable<T> first;
            first = (IComparable<T>)value.First.Item;

            if (first.CompareTo(value.Second.Item) < 0)
            {
                // Wartość właściwości First jest mniejsza niż właściwości Second.
                ...
            }
            else
            {
                // Wartość właściwości Second jest mniejsza lub równa względem właściwości First.
                ...
            }
            _SubItems = value;
        }
    }
    private Pair<BinaryTree<T>> _SubItems;
}
```

Niestety, jeśli teraz zadeklarujesz zmienną klasy `BinaryTree<Typ>`, a podany typ nie zawiera implementacji interfejsu `IComparable<Typ>`, wystąpi błąd czasu wykonania (`InvalidCastException`). To sprawia, że główny powód stosowania typów generycznych — poprawa bezpieczeństwa ze względu na typ — staje się nieaktualny.

2.0

Aby w przypadku gdy podany typ nie zawiera implementacji interfejsu, uniknąć wspomnianego wyjątku i zamiast niego otrzymać błąd kompilacji, można podać dostępną w języku C# opcjonalną listę **ograniczeń** dla każdego określającego typ parametru zadeklarowanego w typie generycznym. Ograniczenie opisuje cechy, jakich dany typ generyczny wymaga od typów podawanych w parametrach. Do deklarowania ograniczeń służy słowo kluczowe `where`, po którym podawana jest para parametr-wymaganie. Parametry muszą być parametrami danego typu generycznego, a wymagania dotyczą klas lub interfejsów, na jakie możliwe musi być przekształcenie typu podanego w parametrze, wymagają obecności konstruktora domyślnego lub określają, że konieczny jest typ referencyjny bądź bezpośredni.

Ograniczenia dotyczące interfejsu

Aby zapewnić właściwy porządek węzłów w drzewie binarnym, można wykorzystać metodę `CompareTo()` z klasy `BinaryTree`. Najlepszym rozwiązaniem jest dodanie ograniczenia dotyczącego określającego typ parametru `T`. Podany typ powinien implementować interfejs `IComparable<T>`. Składnię służącą do deklarowania takiego ograniczenia przedstawiono na listingu 11.22.

Listing 11.22. Deklarowanie ograniczenia dotyczącego interfejsu

```

public class BinaryTree<T>
{
    where T: System.IComparable<T>
    {
        public T Item { get; set; }
        public Pair<BinaryTree<T>> SubItems
        {
            get{ return _SubItems; }
            set
            {
                IComparable<T> first;
                // Zauważ, że teraz można pominąć rzutowanie.
                first = value.First.Item;

                if (first.CompareTo(value.Second.Item) < 0)
                {
                    // Wartość właściwości First jest mniejsza niż wartość właściwości Second.
                    ...
                }
                else
                {
                    // Wartość właściwości Second jest mniejsza lub równa względem właściwości First.
                    ...
                }
                _SubItems = value;
            }
        }
        private Pair<BinaryTree<T>> _SubItems;
    }
}

```

2.0

Po dodaniu na listingu 11.22 ograniczenia dotyczącego interfejsu kompilator za każdym razem, gdy używasz klasy `BinaryTree<T>`, sprawdza, czy podany typ zawiera implementację odpowiedniej wersji interfejsu `IComparable<T>`. Ponadto nie trzeba teraz jawnie rzutować zmiennej na interfejs `IComparable<T>` przed wywołaniem metody `CompareTo()`. Rzutowanie nie jest potrzebne nawet do uzyskania dostępu do składowych z jawnie podawanym interfejsem, gdzie w innych kontekstach brak rzutowania powoduje ukrycie danej składowej. Gdy wywołujesz metodę obiektu typu podanego w parametrze typu generycznego, kompilator sprawdza, czy dana metoda pasuje do którejś z metod dowolnego interfejsu zadeklarowanego w ograniczeniach.

Jeśli teraz spróbujesz utworzyć zmienną typu `BinaryTree<T>`, podając w parametrze typ `System.Text.StringBuilder`, wystąpi błąd kompilacji, ponieważ typ `StringBuilder` nie zawiera implementacji interfejsu `IComparable<StringBuilder>`. Wyświetlany jest wtedy komunikat podobny do tekstu z danych wyjściowych 11.3.

DANE WYJŚCIOWE 11.3.

```

error CS0311: The type 'System.Text.StringBuilder' cannot be used as type parameter 'T' in the generic type or method 'BinaryTree<T>'. There is no implicit reference conversion from 'System.Text.StringBuilder' to 'System.IComparable<System.Text.StringBuilder>'.

```

Aby zażądać implementacji danego interfejsu, należy zadeklarować **ograniczenie dotyczące interfejsu**. Dzięki takiemu ograniczeniu nie trzeba nawet rzutować wartości, by wywołać składowe z jawnie podawanym interfejsem.

Ograniczenia dotyczące klasy

Czasem przydatne jest ograniczenie polegające na tym, że argument ma określać typ, który można przekształcić na wskazaną klasę. W tym celu należy zastosować **ograniczenie dotyczące klasy**, pokazane na listingu 11.23.

Listing 11.23. Deklarowanie ograniczenia dotyczącego klasy

```
public class EntityDictionary<TKey, TValue>
    : System.Collections.Generic.Dictionary<TKey, TValue>
    where TValue : EntityBase
{
    ...
}
```

2.0

Na listingu 11.23 w klasie `EntityDictionary<TKey, TValue>` wymagane jest, by wszystkie typy podawane jako parametr `TValue` umożliwiały niejawną konwersję na klasę `EntityBase`. Dzięki temu wymogowi w implementacji typu generycznego możliwe jest używanie składowych klasy `EntityBase` w wartościach typu `TValue`. Jest tak, ponieważ ograniczenie gwarantuje, że wszystkie typy podane jako argument można niejawnie przekształcić na klasę `EntityBase`.

Składnia służąca do dodawania ograniczenia dotyczącego klasy jest taka sama jak dla ograniczenia dotyczącego interfejsu. Ważne jest jednak to, że ograniczenia dotyczące klasy trzeba podawać przed ograniczeniami dotyczącymi interfejsu (podobnie jak w deklaracji klasy klasę bazową podaje się przed listą implementowanych interfejsów). Jednak — inaczej niż w przypadku ograniczeń dotyczących interfejsu — nie jest możliwe dodanie kilku ograniczeń dotyczących klasy. Wynika to z tego, że klasa nie może dziedziczyć po kilku niepowiązanych ze sobą klasach. W ograniczeniu dotyczącym klasy nie można też podawać klas zamkniętych i typów innych niż klasy. C# nie zezwala na przykład na dodanie ograniczenia dotyczącego typu `string` lub `System.Nullable<T>`, ponieważ wtedy jako argument typu generycznego można podać wyłącznie jeden typ. Trudno wówczas mówić, że typ naprawdę jest „generyczny”. Jeśli jako argument określający typ można podać tylko jeden typ, nie ma sensu stosować do tego parametru. Wystarczy bezpośrednio podać potrzebny typ.

Ponadto w ograniczeniu dotyczącym klas nie można podawać niektórych specjalnych typów. Szczegółowe informacje na ten temat znajdziesz w ZAGADNIENIU DLA ZAAWANSOWANYCH „Wymogi związane z ograniczeniami” w dalszej części rozdziału.

Ograniczenia wymagające struktury lub klasy (struct i class)

Innym przydatnym ograniczeniem w typach generycznych jest możliwość zażądania, by typ podany w argumencie był typem bezpośrednim bez obsługi wartości `null` lub typem referencyjnym. Język C# udostępnia w tym celu specjalną składnię, która działa dla typów bezpo-

średnich i referencyjnych. Zamiast określać klasę, po której `T` ma dziedziczyć, można podać słowo kluczowe `struct` lub `class`. Ilustruje to listing 11.24.

Listing 11.24. Dodawanie wymogu, by jako parametr określający typ podawano typ bezpośredni

```
public struct Nullable<T> :
    IFormattable, IComparable,
    IComparable<Nullable<T>>, INullable
where T : struct
{
    // ...
}
```

Zauważ, że ograniczenie `class` nie wymaga, by jako argument określający typ podano klasę; wymaganie dotyczy typów referencyjnych, dlatego jego nazwa jest myląca. Typ podany jako parametr z ograniczeniem `class` może być dowolną klasą, interfejsem, delegatem lub typem tablicowym.

Ponieważ ograniczenie dotyczące klasy wymaga podania konkretnej klasy, użycie ograniczenia `struct` wyklucza zastosowanie ograniczenia dotyczącego klasy. Nie można więc łączyć ograniczenia `struct` z ograniczeniem dotyczącym konkretnej klasy.

Ograniczenie `struct` ma pewną cechę — uniemożliwia podawanie typów bezpośrednich z obsługą wartości `null`. Z czego to wynika? Typy bezpośrednie z obsługą wartości `null` są implementowane za pomocą typu generycznego `Nullable<T>`, w którym do `T` stosowane jest ograniczenie `struct`. Gdyby typ bezpośredni z obsługą wartości `null` był zgodny z omawianym ograniczeniem, możliwe byłoby zdefiniowanie bezsensownego typu `Nullable<Nullable<int>>`. Typ `int` z dwukrotnie dodaną obsługą wartości `null` jest na tyle nieintuicyjny, że trudno określić jego znaczenie. Z podobnych powodów niedozwolony jest też skrótowy zapis `int??`.

Zestawy ograniczeń

Dla parametru określającego typ można ustawić dowolną liczbę ograniczeń dotyczących interfejsu, ale tylko jedno ograniczenie dotyczące klasy (podobnie w klasie można zaimplementować dowolną liczbę interfejsów, ale dziedziczyć po tylko jednej innej klasie). Każde nowe ograniczenie jest deklarowane na rozdzielonej przecinkami liście, która znajduje się po nazwie parametru typu generycznego i dwukropku. Jeśli występuje więcej niż jeden parametr określający typ, słowo kluczowe `where` należy umieścić przed każdym takim parametrem, do którego dodawane są ograniczenia. Na listingu 11.25 w generycznej klasie `EntityDictionary` zadeklarowane są dwa parametry określające typ — `TKey` i `TValue`. Parametr `TKey` ma dwa ograniczenia dotyczące interfejsu, a do parametru `TValue` dodano jedno ograniczenie dotyczące klasy.

Listing 11.25. Ustawianie wielu ograniczeń

```
public class EntityDictionary<TKey, TValue>
    : Dictionary<TKey, TValue>
where TKey : IComparable<TKey>, IFormattable
where TValue : BaseEntity
```

```
{
  ...
}
```

W tym kodzie ustawianych jest kilka ograniczeń parametru TKey i dodatkowe ograniczenie parametru TValue. Gdy dodawanych jest wiele ograniczeń jednego parametru określającego typ, wszystkie one muszą być spełnione. Na przykład jeśli jako argument TKey podano typ C, typ C musi zawierać implementację interfejsów IComparable<C> oraz IFormattable.

Zauważ, że między klauzulami where nie ma przecinka.

Ograniczenia dotyczące konstruktora

W pewnych sytuacjach w klasie generycznej potrzebny jest obiekt typu podanego jako argument tej klasy. Na listingu 11.26 metoda MakeValue() klasy EntityDictionary<TKey, TValue> musi tworzyć obiekt typu podanego jako parametr TValue.

Listing 11.26. Ograniczenie wymagające dostępności konstruktora domyślnego

```
public class EntityBase<TKey>
{
    public TKey Key { get; set; }
}

public class EntityDictionary<TKey, TValue> :
    Dictionary<TKey, TValue>
    where TKey: IComparable<TKey>, IFormattable
    where TValue : EntityBase<TKey>, new()
{
    // ...

    public TValue MakeValue(TKey key)
    {
        TValue newEntity = new TValue();
        newEntity.Key = key;
        Add(newEntity.Key, newEntity);
        return newEntity;
    }

    // ...
}
```

Ponieważ nie wszystkie obiekty mają publiczne konstruktory domyślne, kompilator nie pozwala na wywołanie konstruktora domyślnego typu podanego jako parametr, jeśli nie ustawiono odpowiedniego ograniczenia. Aby wyeliminować tę regułę kompilatora, należy dodać słowo new() po wszystkich pozostałych ograniczeniach. To słowo jest **ograniczeniem dotyczącym konstruktora**. Wskutek jego dodania typ podany jako parametr musi udostępniać publiczny konstruktor domyślny. Dodane ograniczenie może dotyczyć tylko konstruktora domyślnego. Nie da się utworzyć ograniczenia zapewniającego, że podany typ udostępni konstruktor przyjmujący parametry formalne.

Ograniczenia dotyczące dziedziczenia

Ani parametry typu generycznego, ani ich ograniczenia nie są dziedziczone w klasach pochodnych. Wynika to z tego, że parametry typu generycznego nie są jego składowymi. Pamiętaj, że dziedziczenie klas polega na tym, iż w klasie pochodnej znajdują się wszystkie składowe klasy bazowej. Często stosuje się technikę polegającą na tworzeniu nowych typów generycznych pochodnych od innych typów generycznych. W takiej sytuacji parametry określające typ w pochodnym typie generycznym są używane jako parametry określające typ w generycznej klasie bazowej. Dlatego w klasie pochodnej te parametry muszą mieć takie same (lub mocniejsze) ograniczenia jak w klasie bazowej. Czujesz się zagubiony? Przyjrzyj się listingowi 11.27.

Listing 11.27. Jawnie podawane „odziedziczone” ograniczenia

```
class EntityBase<T> where T : IComparable<T>
{
    // ...
}

// BŁĄD:
// Możliwa musi być konwersja typu 'U' na typ
// 'System.IComparable<U>', aby można było podać 'U' jako
// parametr 'T' w generycznym typie lub w generycznej metodzie.
// class Entity<U> : EntityBase<U>
// {
//     // ...
// }
```

Na listingu 11.27 klasa `EntityBase<T>` wymaga, by podany jako argument typ `U` (używany jako parametr `T` w wyniku deklaracji klasy bazowej `EntityBase<U>`) zawierał implementację interfejsu `IComparable<U>`. Dlatego w klasie `Entity<U>` trzeba zastosować to samo ograniczenie do `U`. W przeciwnym razie wystąpi błąd kompilacji. Ten wzorzec zwiększa świadomość programisty i uwidacznia ograniczenia z klasy bazowej w klasie pochodnej. Pozwala to uniknąć niejasności, które mogą wystąpić, gdy programista używa klasy pochodnej i odkrywa ograniczenie, ale nie rozumie, z czego ono wynika.

Na razie nie omówiono w książce metod generycznych. Zapoznasz się z nimi w dalszej części rozdziału. Zapamiętaj tylko, że także metody mogą być generyczne i można w nich dodawać ograniczenia parametrów określających typ. Jak interpretowane są te ograniczenia, gdy wirtualna metoda generyczna jest dziedziczona lub przesłaniana? Inaczej niż w przypadku ograniczeń parametrów określających typ w klasie generycznej, ograniczenia w nowych wersjach wirtualnych metod generycznych (i w składowych z jawnie podawanym interfejsem) są dziedziczone niejawnie i nie można ich ponownie zadeklarować (zobacz listing 11.28).

Listing 11.28. Powtórne dodawanie odziedziczonych ograniczeń składowych wirtualnych jest niedozwolone

```
class EntityBase
{
    public virtual void Method<T>(T t)
        where T : IComparable<T>
```

```

{
    // ...
}
}

```

```

class Order : EntityBase
{
    public override void Method<T>(T t)
    // Nie można powtórnie dodawać ograniczeń w
    // nowych wersjach przesłanianych składowych.
    // where T : IComparable<T>
    {
        // ...
    }
}

```

W klasie pochodnej od klasy generycznej parametr określający typ można dodatkowo ograniczyć. Wystarczy obok (wymaganych) ograniczeń z klasy bazowej podać dodatkowe ograniczenia. Jednak nowa wersja przesłanianej wirtualnej metody generycznej musi być w pełni zgodna z ograniczeniami zdefiniowanymi w wersji metody z klasy bazowej. Dodatkowe ograniczenia mogą naruszać polimorfizm, dlatego nie są dozwolone. W nowej wersji przesłanianej metody niejawnie obowiązują ograniczenia parametru określającego typ z wersji z klasy bazowej.

■ ZAGADNIENIE DLA ZAAWANSOWANYCH

Wymogi związane z ograniczeniami

W stosunku do ograniczeń obowiązują wymogi chroniące przed powstawaniem bezsensownego kodu. Na przykład nie można łączyć ograniczenia dotyczącego klasy z ograniczeniami `struct` i `class`. Ponadto nie można utworzyć ograniczenia wymagającego użycia typu pochodnego od jednego z typów specjalnych, takich jak `object`, typy tablicowe, `System.ValueType`, `System.Enum` (i typy wyliczeniowe), `System.Delegate` lub `System.MulticastDelegate`.

W niektórych sytuacjach przydatne byłoby wprowadzenie dodatkowych reguł związanych z ograniczeniami. W przedstawionych dalej podrozdziałach znajdziesz przykłady niedozwolonych ograniczeń.

2.0

Ograniczenia dotyczące operatorów są niedozwolone

Nie można utworzyć ograniczenia parametru określającego typ, które wymagałoby implementacji konkretnej metody lub danego operatora. Można to zrobić wyłącznie za pomocą ograniczenia dotyczącego interfejsu (w przypadku metod) lub ograniczenia dotyczącego klasy (dla metod i operatorów). Dlatego generyczna metoda `Add()` z listingu 11.29 nie zadziała.

Listing 11.29. W ograniczeniu nie można dodać wymogu dostępności operatorów

```

public abstract class MathEx<T>
{
    public static T Add(T first, T second)
    {
        // BŁĄD: Operator '+' nie może zostać
        // użyty do operandów typów 'T' i 'T'.
    }
}

```



```

    // return first + second;
}
}

```

W metodzie przyjęto, że operator + jest dostępny we wszystkich typach, które mogą zostać podane jako parametr T. Nie istnieje jednak ograniczenie, które pozwala zapobiec podaniu typu bez operatora dodawania. Dlatego występuje błąd. Niestety, nie można utworzyć ograniczenia, które wymaga dostępności operatora dodawania. Jedyne rozwiązanie to zastosowanie ograniczenia dotyczącego klasy i zażądanie klasy z implementacją operatora dodawania.

Można więc uogólnić i stwierdzić, że nie ma sposobu na ograniczenie dozwolonych typów do tych z potrzebną metodą statyczną.

Relacja LUB między ograniczeniami nie jest obsługiwana

Jeśli podasz kilka ograniczeń dotyczących interfejsu lub klasy, kompilator zawsze przyjmie, że występuje między nimi relacja I (czyli że wszystkie ograniczenia muszą być spełnione). Na przykład ograniczenie `where T : IComparable<T>, IFormattable` wymaga, by zaimplementowane były interfejsy `IComparable<T>` i `IFormattable`. Nie da się zapisać relacji LUB między ograniczeniami, dlatego kod z listingu 11.30 jest niedozwolony.

Listing 11.30. Łączenie ograniczeń za pomocą relacji LUB nie jest dozwolone

```

public class BinaryTree<T>
    // BŁĄD: relacja LUB nie jest obsługiwana.
    // where T: System.IComparable<T> || System.IFormattable
{
    ...
}

```

2.0

Dodanie obsługi tego mechanizmu uniemożliwiłoby kompilatorowi określenie na etapie kompilacji, którą metodę należy wywołać.

Ograniczenia dotyczące delegatów i wyliczeń są niedozwolone

Typy delegatów, typy tablicowe i typy wyliczeniowe nie mogą być używane w ograniczeniach, ponieważ działają jak typy zamknięte (jeśli nie wiesz, czym są typy delegatów, zajrzyj do rozdziału 12.). Typy bazowe wymienionych konstrukcji (`System.Delegate`, `System.MulticastDelegate`, `System.Array` i `System.Enum`) też nie mogą być używane jako ograniczenia. Dlatego gdy kompilator natrafi na deklarację klasy przedstawioną na listingu 11.31, zgłosi błąd.

Listing 11.31. W ograniczeniu dotyczącym dziedziczenia nie można używać typu `System.Delegate`

```

// BŁĄD: w ograniczeniu nie można używać specjalnej klasy 'System.Delegate'.
public class Publisher<T>
    where T : System.Delegate
{
    public event T Event;
    public void Publish()
    {
        if (Event != null)
        {
            Event(this, new EventArgs());
        }
    }
}

```

```

    }
}
}

```

Wszystkie typy delegatów są uznawane za klasy specjalne, których nie można podawać jako parametrów określających typ. Podanie typu delegata uniemożliwia sprawdzanie poprawności wywołań metody `Event()` na etapie kompilacji, ponieważ w typach `System.Delegate` i `System.MulticastDelegate` sygnatura zgłaszanego zdarzenia nie jest znana. Podobny zakaz dotyczy typów wyliczeniowych.

W ograniczeniu dotyczącym konstruktora można podawać tylko konstruktory domyślne

Na listingu 11.26 znajduje się ograniczenie dotyczące konstruktora, zgodnie z którym typ podany jako parametr `TValue` musi udostępniać publiczny konstruktor bezparametrowy. Nie można utworzyć ograniczenia, które wymusza podanie typu udostępniającego konstruktor przyjmujący parametry formalne. Możliwe, że chcesz pozwolić na podawanie jako parametr `TValue` wyłącznie typów zawierających konstruktor, który przyjmuje typ określany za pomocą parametru `TKey`. Nie da się jednak utworzyć takiego ograniczenia. Dlatego kod z listingu 11.32 jest nieprawidłowy.

2.0

Listing 11.32. W ograniczeniu dotyczącym konstruktora można podać wyłącznie konstruktor domyślny

```

public TValue New(TKey key)
{
    // BŁĄD: 'TValue': nie można podawać argumentów
    // w trakcie tworzenia instancji typu generycznego.
    TValue newEntity = null;
    // newEntity = new TValue(key);
    Add(newEntity.Key, newEntity);
    return newEntity;
}

```

Jednym ze sposobów na wyeliminowanie tego ograniczenia jest użycie interfejsu fabrycznego, który udostępnia metodę do tworzenia obiektów danego typu. Wtedy za tworzenie obiektów typu `EntityDictionary` odpowiada klasa fabryczna z implementacją wspomnianego interfejsu, a nie sama klasa `EntityDictionary` (zobacz listing 11.33).

Listing 11.33. Używanie interfejsu fabrycznego zamiast ograniczenia dotyczącego konstruktora

```

public class EntityBase<TKey>
{
    public EntityBase(TKey key)
    {
        Key = key;
    }
    public TKey Key { get; set; }
}

public class EntityDictionary<TKey, TValue, TFactory> :
    Dictionary<TKey, TValue>

```

```

where TKey : IComparable<TKey>, IFormattable
where TValue : EntityBase<TKey>
where TFactory : IEntityFactory<TKey, TValue>, new()
{
    ...
    public TValue New(TKey key)
    {
        TFactory factory = new TFactory();
        TValue newEntity = factory.CreateNew(key);
        Add(newEntity.Key, newEntity);
        return newEntity;
    }
    ...
}

public interface IEntityFactory<TKey, TValue>
{
    TValue CreateNew(TKey key);
}
...

```

2.0

Taka deklaracja umożliwia przekazanie nowego klucza (parametr `key`) do przyjmującej parametry metody fabrycznej tworzącej obiekt typu podanego w parametrze `TValue`. Dzięki temu nie trzeba polegać na konstruktorze domyślnym. Ponadto nie trzeba tworzyć ograniczenia dotyczącego konstruktora dla parametru `TValue`, ponieważ to obiekt typu `TFactory` odpowiada za tworzenie obiektów. W kodzie z listingu 11.33 można wprowadzić pewną modyfikację — zapisywać referencję do metody fabrycznej (na przykład z wykorzystaniem typu `Lazy<T>`, jeśli potrzebna jest obsługa wielowątkowości). Pozwoli to wielokrotnie wykorzystać metodę fabryczną, zamiast za każdym razem tworzyć zawierający ją obiekt.

Aby zadeklarować zmienną typu `EntityDictionary<TKey, TValue, TFactory>`, można utworzyć typ encji podobny do typu `Order` z listingu 11.34.

Listing 11.34. Deklarowanie typu encji używanych w typie `EntityDictionary<...>`

```

public class Order : EntityBase<Guid>
{
    public Order(Guid key) :
        base(key)
    {
        // ...
    }
}

public class OrderFactory : IEntityFactory<Guid, Order>
{
    public Order CreateNew(Guid key)
    {
        return new Order(key);
    }
}

```

Metody generyczne

Wcześniej przekonaliśmy się, że dodawanie metod do typów generycznych jest proste. W takiej metodzie można wykorzystać generyczne parametry określające typ. Zetknąłeś się już z tym rozwiązaniem w pokazanych wcześniej przykładowych klasach generycznych.

Metody generyczne (podobnie jak typy generyczne) korzystają z parametrów określających typ. Takie metody można deklarować w typach generycznych i zwykłych. Jeśli metoda jest zadeklarowana w typie generycznym, jej parametry są niezależne od tych z danego typu generycznego. Aby zadeklarować metodę generyczną, należy podać generyczne parametry określające typ w taki sam sposób jak w typach generycznych. Kod typu określającego parametr należy dodać bezpośrednio po nazwie metody, tak jak w przykładowych metodach `MathEx.Max<T>` i `MathEx.Min<T>` z listingu 11.35.

Listing 11.35. Definiowanie metod generycznych

```
public static class MathEx
{
    public static T Max<T>(T first, params T[] values)
        where T : IComparable<T>
    {
        T maximum = first;
        foreach (T item in values)
        {
            if (item.CompareTo(maximum) > 0)
            {
                maximum = item;
            }
        }
        return maximum;
    }

    public static T Min<T>(T first, params T[] values)
        where T : IComparable<T>
    {
        T minimum = first;

        foreach (T item in values)
        {
            if (item.CompareTo(minimum) < 0)
            {
                minimum = item;
            }
        }
        return minimum;
    }
}
```

W tym przykładzie metoda jest statyczna, choć język C# tego nie wymaga.

Metody generyczne, podobnie jak typy generyczne, mogą obejmować więcej niż jeden parametr określający typ. Arność (liczba parametrów określających typ) to cecha pozwalająca

odróżniać od siebie sygnatury metod. Dozwolone jest utworzenie dwóch metod o identycznych nazwach i typach parametrów formalnych, jeśli liczba parametrów określających typ w tych metodach jest różna.

Inferencja typów w metodach generycznych

2.0

W typach generycznych argumenty określające typ podawane są po nazwie typu. Podobnie w metodach generycznych argumenty określające typ należy podać po nazwie metody. Kod z wywołaniami metod `Min<T>` i `Max<T>` przedstawiono na listingu 11.36.

Listing 11.36. Jawne podawanie parametrów określających typ

```
Console.WriteLine(
    MathEx.Max<int>(7, 490));
Console.WriteLine(
    MathEx.Min<string>("R.O.U.S.", "Fireswamp"));
```

Wynik działania kodu z listingu 11.36 pokazano w danych wyjściowych 11.4.

DANE WYJŚCIOWE 11.4.

```
490
Fireswamp
```

Nie jest zaskoczeniem, że argumenty określające typ (`int` i `string`) są zgodne z typami użytymi w wywołaniach metod generycznych. Jednak podawanie argumentów określających typ nie jest konieczne, ponieważ kompilator potrafi ustalić typ na podstawie przekazanych do metody argumentów. Programista wywołania metody `Max` z listingu 11.36 chciał, by argumentem określającym typ był `int`, ponieważ oba argumenty metody są tego typu. Aby uniknąć zbędnego kodu, w wywołaniu można pominąć parametr określający typ, jeśli kompilator potrafi logicznie ustalić typ oczekiwany przez programistę. Przykład zastosowania tego mechanizmu, **inferencji typów w metodzie**, znajdziesz na listingu 11.37. Wynik działania tego kodu pokazano w danych wyjściowych 11.5.

Listing 11.37. Inferencja argumentu określającego typ na podstawie przekazanych argumentów

```
Console.WriteLine(
    MathEx.Max(7, 490)); // Brak argumentu określającego typ!
Console.WriteLine(
    MathEx.Min("R.O.U.S'", "Fireswamp"));
```

DANE WYJŚCIOWE 11.5.

```
490
Fireswamp
```

Aby inferencja typu w metodzie zakończyła się powodzeniem, typy argumentów muszą być „dopasowane” do parametrów formalnych metody generycznej w taki sposób, by dało się

2.0

ustalić argumenty określające typ. Co się jednak stanie, jeśli w wyniku inferencji wybrane zostaną sprzeczne argumenty? Na przykład jeśli programista wywoła metodę `Max<T>` za pomocą wywołania `MathEx.Max(7.0, 490)`, kompilator może na podstawie pierwszego argumentu wywnioskować, że argumentem określającym typ powinien być typ `double`, a na podstawie drugiego argumentu uznać, że należy zastosować `int`. Typy te są niezgodne ze sobą. W wersji C# 2.0 w takiej sytuacji zgłaszany jest błąd. Po zastanowieniu można zauważyć, że niezgodność da się wyeliminować, ponieważ każdą wartość typu `int` można przekształcić na typ `double`. Dlatego jako argument określający typ należy zastosować `double`. W wersjach C# 3.0 i C# 4.0 wprowadzono usprawnienia w algorytmie inferencji typów w metodach. Dzięki temu kompilator może przeprowadzać bardziej zaawansowane analizy.

W sytuacjach gdy mechanizm inferencji nie jest wystarczająco zaawansowany, by wywnioskować wartość argumentów określających typ, można rozwiązać błąd dzięki rzutowaniu argumentów. W ten sposób można poinformować kompilator o typach, które należy uwzględnić w trakcie inferencji. Inne rozwiązanie to rezygnacja z inferencji typów i jawne podanie argumentów określających typ.

Zauważ, że algorytm inferencji typów w metodzie uwzględni tylko argumenty, ich typy i typy parametrów formalnych metody generycznej. Algorytm w ogóle nie bierze pod uwagę innych czynników, które w praktyce mogłyby zostać wykorzystane w trakcie analiz. Te czynniki to na przykład typ wartości zwracanej przez metodę generyczną, typ zmiennej, do której przypisywana jest wartość zwracana przez metodę, lub ograniczenia używanych w metodzie generycznych parametrów określających typ.

Dodawanie ograniczeń

Dla parametrów określających typ w metodach generycznych można ustawić dokładnie te same ograniczenia co dla analogicznych parametrów w typach generycznych. Możesz na przykład dodać ograniczenie, zgodnie z którym typ podany w parametrze musi zawierać implementację danego interfejsu lub umożliwiać konwersję na wybraną klasę. Ograniczenia należy podawać między listą argumentów i ciałem metody, co pokazano na listingu 11.38.

Listing 11.38. Dodawanie ograniczeń w metodach generycznych

```
public class ConsoleTreeControl
{
    // Metoda generyczna Show<T>.
    public static void Show<T>(BinaryTree<T> tree, int indent)
    where T : IComparable<T>
    {
        Console.WriteLine("\n{0}{1}",
            "+ --".PadLeft(5*indent, ' '),
            tree.Item.ToString());
        if (tree.SubItems.First != null)
            Show(tree.SubItems.First, indent+1);
        if (tree.SubItems.Second != null)
            Show(tree.SubItems.Second, indent+1);
    }
}
```

W tym kodzie w metodzie `Show<T>` nie są bezpośrednio używane żadne składowe interfejsu `IComparable<T>`. Po co więc w ogóle dodawać ograniczenie? Pamiętaj, że jest ono potrzebne w klasie `BinaryTree<T>` (zobacz listing 11.39).

Listing 11.39. Klasa `BinaryTree<T>` wymaga podania typu z implementacją interfejsu `IComparable<T>`

```
public class BinaryTree<T>
    where T: System.IComparable<T>
{
    ...
}
```

Ponieważ klasa `BinaryTree<T>` wymaga wspomnianego ograniczenia parametru `T`, a w metodzie `Show<T>` używany jest argument `T` odpowiadający parametrowi z ograniczeniem, w metodzie należy zagwarantować, że ograniczenie parametru z klasy będzie spełnione także dla parametru z metody.

■ ZAGADNIENIE DLA ZAAWANSOWANYCH

Rzutowanie w metodach generycznych

Czasem należy zachować ostrożność w trakcie korzystania z typów lub metod generycznych — na przykład wtedy, gdy są używane specjalnie do przeprowadzenia rzutowania. Przyjrzyj się poniższej metodzie. Przekształca ona strumień na obiekt podanego typu:

```
public static T Deserialize<T>(
    Stream stream, IFormatter formatter)
{
    return (T)formatter.Deserialize(stream);
}
```

Obiekt `formatter` odpowiada za usuwanie danych ze strumienia i przekształcanie ich w obiekt. Wywołanie metody `Deserialize` obiektu `formatter` powoduje zwrócenie danych typu `object`. Wywołanie generycznej wersji metody `Deserialize()` wygląda tak:

```
string greeting =
    Deserialization.Deserialize<string>(stream, formatter);
```

Problem z tym kodem polega na tym, że dla jednostki wywołującej metoda `Deserialize<T>()` wydaje się bezpieczna ze względu na typ. Jednak na rzecz jednostki wywołującej przeprowadzane jest rzutowanie — tak jak w pokazanym poniżej niegenerycznym odpowiedniku przedstawionego wcześniej wywołania.

```
string greeting =
    (string)Deserialization.Deserialize(stream, formatter);
```

W czasie wykonywania programu to rzutowanie może się zakończyć niepowodzeniem. Metoda nie jest więc tak bezpieczna ze względu na typ, jak może się wydawać. Metoda `Deserialize<T>` jest generyczna wyłącznie po to, by mogła ukryć rzutowanie przed jednostką wywołującą. Jest to niebezpiecznie zwodnicze rozwiązanie. Lepszym podejściem może być utworzenie niegenerycznej wersji metody i zwracanie w niej obiektu typu `object`. Dzięki

2.0

temu w jednostce wywołującej wiadomo, że metoda nie jest bezpieczna ze względu na typ. Programiści powinni zachować ostrożność, gdy w generycznej metodzie dane są rzutowane, a nie ma ograniczenia sprawdzającego poprawność tej operacji.

Wskazówka

UNIKAJ tworzenia metod generycznych, które wywołują u autora jednostki wywołującej mylne wrażenie, że są bezpieczne ze względu na typ.

Kowariancja i kontrawariancja

Początkujący użytkownicy typów generycznych często zastanawiają się, dlaczego wyrażenia typu `List<string>` nie można przypisać na przykład do zmiennej typu `List<object>`. Skoro wartość typu `string` można przekształcić na typ `object`, lista łańcuchów znaków powinna być zgodna z listą obiektów. Jednak takie rozwiązanie nie jest ani bezpieczne ze względu na typ, ani dozwolone. Jeśli zadeklarujesz dwie zmienne tej samej klasy generycznej, ale z innymi parametrami określającymi typ, zmienne nie będą miały zgodnego typu nawet wtedy, jeśli jeden z podanych typów jest pochodny od drugiego. Takie zmienne nie są **kowariantne**.

Kowariancja to techniczne pojęcie z teorii kategorii. Opisuje ono proste zjawisko. Załóżmy, że między dwoma typami X i Y występuje specjalna relacja, polegająca na tym, że każdą wartość typu X można przekształcić na typ Y . Jeśli między typami $I<X>$ i $I<Y>$ także zawsze występuje ta sama relacja, można powiedzieć, że „ $I<T>$ jest kowariantny względem T ”. Dla prostych typów generycznych mających tylko jeden parametr określający typ ten parametr jest oczywisty, dlatego wystarczy powiedzieć „ $I<T>$ jest kowariantny”. W takiej sytuacji konwersja z $I<X>$ na $I<Y>$ jest **konwersją kowariantną**.

2.0

Obiekty generycznych klas `Pair<Contact>` i `Pair<PdaItem>` nie są zgodne ze względu na typ, choć same typy podane jako argumenty są ze sobą zgodne. Kompilator blokuje konwersję (niejawną i jawną) między typami `Pair<Contact>` i `Pair<PdaItem>`, choć `Contact` dziedziczy po `PdaItem`. Także próba konwersji obiektu typu `Pair<Contact>` na interfejs `IPair<PdaItem>` zakończy się niepowodzeniem. Przykładowy kod pokazano na listingu 11.40.

Listing 11.40. Konwersja między typami generycznymi z różnymi parametrami określającymi typ

```
// ...
// BŁĄD: nie można przeprowadzić konwersji typów.
Pair<PdaItem> pair = (Pair<PdaItem>) new Pair<Contact>();
IPair<PdaItem> duple = (IPair<PdaItem>) new Pair<Contact>();
```

Jednak dlaczego jest to niedozwolone? Dlaczego typy `List<T>` i `Pair<T>` nie są kowariantne? Na listingu 11.41 pokazano, co by się stało, gdyby język C# zapewniał nieograniczoną kowariancję.

Obiekt typu `IPair<PdaItem>` może zawierać adres, jednak w tym kodzie obiekt w rzeczywistości jest typu `Pair<Contact>`, dlatego może przechowywać tylko dane kontaktowe, a nie adresy. Tak więc nieograniczona kowariancja skutkuje naruszeniem bezpieczeństwa ze względu na typ.

Listing 11.41. Rezygnacja z kowariancji pozwala zachować jednolitość typów

```
//...
Contact contact1 = new Contact("Princess Buttercup"),
Contact contact2 = new Contact("Inigo Montoya");
Pair<Contact> contacts = new Pair<Contact>(contact1, contact2);

// Ten kod prowadzi do błędu z komunikatem, że nie można przeprowadzić konwersji.
// Wyobraź sobie jednak, że taka instrukcja jest dozwolona.
// IPair<PdaItem> pdaPair = (IPair<PdaItem>) contacts;
// Wtedy poniższy kod jest poprawny, ale nie zapewnia bezpieczeństwa ze względu na typ.
// pdaPair.First = new Address("Ulica Sezamkowa 123");
...

```

Teraz powinno być zrozumiałe, dlaczego listy łańcuchów znaków nie można użyć jako listy obiektów. Nie można wstawić liczby całkowitej do listy łańcuchów znaków, natomiast jest możliwe wstawienie takiej liczby do listy obiektów. Dlatego rzutowanie listy łańcuchów znaków na listę obiektów musi być niedozwolone i kompilator zgłasza wtedy błąd.

2.0

Umożliwianie kowariancji za pomocą modyfikatora out stosowanego do parametru określającego typ (od wersji C# 4.0)Początek
4.0

Może zauważyłeś, że oba opisane wcześniej problemy związane z nieograniczoną kowariancją wynikają z tego, że generyczna para i generyczna lista umożliwiają zapis przechowywanych w nich danych. Załóżmy, że wyeliminujesz tę możliwość, tworząc przeznaczony tylko do odczytu interfejs `IReadOnlyPair<T>`, który udostępni `T` jako typ wartości wyjściowej interfejsu (`T` może być tu typem wartości zwracanej przez metodę lub przeznaczoną tylko do odczytu właściwość). `T` nigdy nie może być wtedy typem wartości wejściowej (nie może być typem parametru formalnego ani typem właściwości z możliwością zapisu). Jeśli wprowadzisz ograniczenie dotyczące interfejsu powodujące, że `T` może być tylko typem wartości wyjściowych, opisany wcześniej problem z kowariancją nie wystąpi (zobacz listing 11.42).

Listing 11.42. Rozwiązanie z potencjalnie możliwą kowariancją

```
interface IReadOnlyPair<T>
{
    T First { get; }
    T Second { get; }
}

interface IPair<T>
{
    T First { get; set; }
    T Second { get; set; }
}

public struct Pair<T> : IPair<T>, IReadOnlyPair<T>
{
    // ...
}

```

4.0

```

class Program
{
    static void Main()
    {
        // BŁĄD: tylko teoretycznie możliwe, jeśli nie
        // dodasz modyfikatora out dla parametru określającego typ.
        Pair<Contact> contacts =
            new Pair<Contact>(
                new Contact("Princess Buttercup"),
                new Contact("Inigo Montoya") );
        IReadOnlyPair<PdaItem> pair = contacts;
        PdaItem pdaItem1 = pair.First;
        PdaItem pdaItem2 = pair.Second;
    }
}

```

2.0

Gdy ograniczysz deklarację typu generycznego w taki sposób, że dane są dostępne tylko jako dane wyjściowe z interfejsu, nie ma powodu, by kompilator blokował możliwość kowariancji. Wszystkie operacje na obiekcie typu `IReadOnlyPair<PdaItem>` powodują przekształcenie obiektów typu `Contact` (z pierwotnego obiektu typu `Pair<Contact>`) na typ bazowy `PdaItem`. Jest to w pełni poprawna konwersja. Nie może się wtedy zdarzyć, że program zapisze adres w obiekcie, który w rzeczywistości jest parą danych kontaktowych. Dzieje się tak, ponieważ użyty interfejs nie udostępnia właściwości przeznaczonych do zapisu.

Mimo to kod z listingu 11.42 także się nie skompiluje. W wersji C# 4 dodano jednak obsługę bezpiecznej kowariancji. Aby określić, że generyczny interfejs ma umożliwiać kowariancję względem jednego z parametrów określających typ, ten parametr trzeba zadeklarować z modyfikatorem `out`. Na listingu 11.43 pokazano, jak zmodyfikować deklarację interfejsu, by informowała, że należy umożliwić kowariancję.

Listing 11.43. Kowariancja dzięki użyciu modyfikatora `out` do parametru określającego typ

```

...
interface IReadOnlyPair<out T>
{
    T First { get; }
    T Second { get; }
}

```

4.0

Dodanie modyfikatora `out` do parametru określającego typ w interfejsie `IReadOnlyPair<out T>` umożliwia kompilatorowi stwierdzenie, że typ `T` rzeczywiście jest używany tylko dla danych wyjściowych — jako typ wartości zwracanych przez metodę lub przez właściwości przeznaczone tylko do odczytu. Typ ten nigdy nie jest używany dla parametrów formalnych lub w setterze właściwości. Na tej podstawie kompilator dopuszcza konwersję kowariantną z wykorzystaniem tego interfejsu. Po wprowadzeniu potrzebnej zmiany w kodzie z listingu 11.42 program można z powodzeniem skompilować i wykonać.

Stosowanie konwersji kowariantnych jest związane z wieloma ważnymi zastrzeżeniami.

- Kowariancja jest możliwa tylko w kontekście generycznych interfejsów i generycznych delegatów (zobacz rozdział 12.). Klasy i struktury generyczne nigdy nie obsługują kowariancji.
- Argumenty określające typ w źródłowym i docelowym typie generycznym muszą być typem referencyjnym (nie można używać typów bezpośrednich). To oznacza, że obiekt typu `IReadOnlyPair<string>` można kowariantnie przekształcić na obiekt typu `IReadOnlyPair<object>`, ponieważ `string` i `IReadOnlyPair<object>` to typy referencyjne. Natomiast nie jest możliwe przekształcenie obiektu typu `IReadOnlyPair<int>` na typ `IReadOnlyPair<object>`, ponieważ `int` nie jest typem referencyjnym.
- Używany interfejs lub delegat musi być zadeklarowany jako obsługujący kowariancję. Ponadto kompilator musi móc stwierdzić, że odpowiednie parametry określające typ są używane tylko dla wartości wyjściowych.

2.0

Umożliwianie kontrawariancji z użyciem modyfikatora `in` dla parametru określającego typ (od wersji C# 4.0)

Kowariancja przeprowadzana „w drugą stronę” to **kontrawariancja**. Ponownie założmy, że dwa typy, `X` i `Y`, są powiązane w taki sposób, że każdą wartość typu `X` można przekształcić na wartość typu `Y`. Jeśli typy `I<X>` i `I<Y>` zawsze spełniają tę samą relację „w drugą stronę”, czyli każdą wartość typu `I<Y>` można przekształcić na typ `I<X>`, to `I<T>` jest kontrawariantny względem `T`.

Dla większości osób kontrawariancja jest dużo trudniejsza do zrozumienia niż kowariancja. Standardowym przykładem ilustrującym kontrawariancję jest mechanizm porównań. Załóżmy, że utworzyłeś typ `Apple` pochodny od typu `Fruit`. Między tymi typami występuje specjalna relacja — każdą wartość typu `Apple` można przekształcić na typ `Fruit`.

Teraz założmy, że istnieje interfejs `ICompareThings<T>` zawierający metodę `bool FirstIsBetter<T t1, T t2>`. Ta metoda przyjmuje dwa obiekty typu `T` i zwraca wartość logiczną informującą, czy pierwszy obiekt jest lepszy od drugiego.

Co się stanie, gdy podasz argumenty określające typ? Obiekt typu `ICompareThings<Apple>` udostępnia metodę, która przyjmuje dwa obiekty typu `Apple` i je porównuje. Obiekt typu `ICompareThings<Fruit>` ma metodę, która przyjmuje dwa obiekty typu `Fruit` i je porównuje. Jednak ponieważ każdy obiekt typu `Apple` jest też typu `Fruit`, możliwe powinno być bezpieczne użycie wartości typu `ICompareThings<Fruit>` wszędzie tam, gdzie potrzebny jest obiekt `ICompareThings<Apple>`. Kierunek konwersji jest tu odwrócony, stąd nazwa kontrawariancja.

4.0

Prawdopodobnie nie jest zaskoczeniem, że bezpieczna kontrawariancja wymaga odwrotnych ograniczeń interfejsu niż kowariancja. Interfejs umożliwiający kontrawariancję z użyciem jednego z parametrów określających typ musi wykorzystywać odpowiedni parametr tylko dla wartości wejściowych, na przykład w parametrach formalnych (lub we właściwości przeznaczonej tylko do zapisu, co jednak zdarza się bardzo rzadko). Interfejs można opisać jako zgodny z kontrawariancją, dodając modyfikator `in` do parametru określającego typ. To rozwiązanie pokazano na listingu 11.44.

Listing 11.44. Kontrawariancja dzięki zastosowaniu modyfikatora `in` do parametru określającego typ

```

class Fruit {}
class Apple : Fruit {}
class Orange : Fruit {}

```

```

interface ICompareThings<in T>
{
    bool FirstIsBetter(T t1, T t2);
}

```

```

class Program
{
    class FruitComparer : ICompareThings<Fruit>
    { ... }
    static void Main()
    {
        // Dozwolone w wersji C# 4.0.
        ICompareThings<Fruit> fc = new FruitComparer();
        Apple apple1 = new Apple();
        Apple apple2 = new Apple();
        Orange orange = new Orange();
        // Obiekt typu FruitComparer może porównywać jabłka (Apple) z pomarańczami (Orange).
        bool b1 = fc.FirstIsBetter(apple1, orange);
        // a także jabłka z jabłkami.
        bool b2 = fc.FirstIsBetter(apple1, apple2);
        // Jest to dozwolone, ponieważ używany interfejs umożliwia kontrawariancję.
        ICompareThings<Apple> ac = fc;
        // W rzeczywistości obiekt jest typu FruitComparer, dlatego
        // też może porównywać dwa jabłka.
        bool b3 = ac.FirstIsBetter(apple1, apple2);
    }
}

```

Kontrawariancja (podobnie jak kowariancja) wymaga użycia modyfikatora parametru określającego typ. Tu jest to modyfikator `in`, który występuje w deklaracji określającego typ parametru interfejsu. Jest to dla kompilatora informacja, że ma sprawdzić, czy `T` nigdy nie występuje w getterze właściwości lub jako typ wartości zwracanej przez metodę. To umożliwia konwersje kontrawariantne z użyciem danego interfejsu.

Konwersje kontrawariantne podlegają analogicznym ograniczeniom co opisane wcześniej konwersje kowariantne. Są dozwolone tylko dla generycznych interfejsów i delegatów, jako parametr określający typ trzeba podać typ referencyjny, a kompilator musi mieć możliwość ustalenia, że interfejs pozwala na bezpieczne konwersje kontrawariantne.

Interfejs może obsługiwać kowariancję względem jednego parametru określającego typ i kontrawariancję względem innego parametru. W praktyce takie rozwiązanie stosuje się rzadko (wyjątkiem są delegaty). Na przykład rodzina delegatów `Func<A1, A2, ..., R>` jest kowariantna względem typu zwracanej wartości (`R`), a kontrawariantna względem pozostałych parametrów określających typ.

Zauważ, że kompilator sprawdza w kodzie źródłowym poprawność modyfikatorów parametrów ważnych ze względu na kowariancję i kontrawariancję. Przyjrzyj się interfejsowi `Pair >_INITIALIZER<in T>` na listingu 11.45.

Listing 11.45. Sprawdzanie poprawności wariancji przez kompilator

```
// BŁĄD: nieprawidłowa wariancja. Określający typ parametr "T"
// nie jest poprawny ze względu na wariancję.
interface IPairInitializer<in T>
{
    void Initialize(IPair<T> pair);
}

// Załóżmy, że przedstawiony wyżej kod jest poprawny.
// Zobacz, jakie problemy mogą wystąpić.
class FruitPairInitializer : IPairInitializer<Fruit>
{
    // Kod inicjuje parę obiektów typu Fruit
    // wartościami typów Orange i Apple.
    public void Initialize(IPair<Fruit> pair)
    {
        pair.First = new Orange();
        pair.Second = new Apple();
    }
}

// Dalej w kodzie.
var f = new FruitPairInitializer();
// Gdyby kontrawariancja była tu dozwolona, ten kod byłby poprawny:
IPairInitializer<Apple> a = f;
// Poniższy kod zapisuje obiekt typu Orange w obiekcie z parą obiektów typu Apple.
a.Initialize(new Pair<Apple>());
```

4.0

Na pozór można sądzić, że ponieważ typ `IPair<T>` jest używany tylko dla wejściowego parametru formalnego, kontrawariantny modyfikator `in` w typie `IPairInitializer` jest prawidłowy. Jednak interfejs `IPair<T>` nie może być bezpiecznie modyfikowany, dlatego nie można go tworzyć ze zmiennym argumentem określającym typ. Jak widać, to rozwiązanie nie jest bezpieczne ze względu na typ, dlatego kompilator w ogóle nie zezwala na zadeklarowanie interfejsu `IPairInitializer<T>` jako kontrawariantnego.

Obsługa niezabezpieczonej kowariancji w tablicach

Do tego miejsca kowariancja i kontrawariancja były opisywane jako cechy typów generycznych. Spośród wszystkich typów niegenerycznych najbardziej generyczne są tablice. Podobnie jak można tworzyć generyczne listy obiektów typu `T` lub generyczne pary obiektów typu `T`, tak można potraktować tablicę obiektów typu `T` jako wzorzec. Ponieważ tablice umożliwiają odczyt i zapis danych, to na podstawie wiedzy o kowariancji i kontrawariancji prawdopodobnie podejrzewasz, że tablice nie obsługują bezpiecznej kontrawariancji ani kowariancji. Zapewne sądzisz, że tablice umożliwiają bezpieczną kowariancję tylko wtedy, gdy nie pozwalają na zapis, a bezpieczna kontrawariancja jest możliwa tylko wtedy, gdy dane z tablicy nigdy nie są wczytywane (choć oba te ograniczenia są nierealistyczne).

Niestety, C# umożliwia kowariancję w tablicach, choć ta operacja nie jest bezpieczna ze względu na typ. Na przykład instrukcja `Fruit[] fruits = new Apple[10];` jest w języku C#

2.0

w pełni poprawna. Jeśli potem wykonasz wyrażenie `fruits[0] = new Orange();`, środowisko uruchomieniowe zgłosi wyjątek informujący o naruszeniu bezpieczeństwa typu. Bardzo kłopotliwe jest to, że nie zawsze można poprawnie przypisać obiekt typu `Orange` do tablicy elementów typu `Fruit`, ponieważ w rzeczywistości może to być tablica elementów typu `Apple`. Problem ten dotyczy nie tylko języka C#, ale wszystkich języków ze środowiska CLR, w których używana jest implementacja tablic ze środowiska uruchomieniowego.

Staraj się unikać niezabezpieczonej kowariancji z użyciem tablic. Każdą tablicę można przekształcić na przeznaczony tylko do odczytu (a tym samym bezpieczny ze względu na kowariancję) interfejs `IEnumerable<T>`. Dlatego wyrażenie `IEnumerable<Fruit> fruits = new Apple[10]` jest bezpieczne i dozwolone, ponieważ nie można wstawić do tej tablicy obiektu typu `Orange` (dostępny jest wyłącznie interfejs przeznaczony tylko do odczytu).

Wskazówka

UNIKAJ stosowania niezabezpieczonej kowariancji z wykorzystaniem tablic. Zamiast tego **ROZWAŻ** konwersję tablicy na przeznaczony tylko do odczytu interfejs `IEnumerable<T>`, co pozwala na bezpieczne konwersje kowariantne.

4.0

Wewnętrzne mechanizmy typów generycznych

Z poprzednich rozdziałów dowiedziałeś się o powszechności obiektów w systemie typów interfejsu CLI. Nie powinno być więc zaskoczeniem, że typy generyczne też służą do tworzenia obiektów. Określający parametr typ w klasie generycznej jest używany jako metadane, wykorzystywane przez środowisko uruchomieniowe do budowania odpowiednich klas, gdy są one potrzebne. Dlatego typy generyczne obsługują dziedziczenie, polimorfizm i hermetyzację. W typach generycznych można definiować metody, właściwości, pola, klasy, interfejsy i delegaty.

2.0

Aby było to możliwe, typy generyczne wymagają obsługi w używanym środowisku uruchomieniowym. W C# typy generyczne są mechanizmem obsługiwany zarówno przez kompilator, jak i przez platformę. Na przykład aby uniknąć opakowywania obiektów, używana jest inna implementacja typów generycznych w zależności od tego, czy jako parametr określający typ podano typ bezpośredni, czy typ referencyjny.

ZAGADNIENIE DLA ZAAWANSOWANYCH

Reprezentacja typów generycznych w kodzie CIL

Po skompilowaniu klasa generyczna tylko nieznacznie różni się od klasy niegenerycznej. W wyniku kompilacji powstają metadane i kod CIL. Kod CIL jest sparametryzowany, by umożliwić zastosowanie typu podanego przez użytkownika w określonym miejscu kodu. Załóżmy, że zadeklarowana jest prosta klasa `Stack` przedstawiona na listingu 11.46.

Po skompilowaniu tej klasy wygenerowany kod CIL jest sparametryzowany i wygląda tak jak na listingu 11.47.

Listing 11.46. Deklaracja klasy Stack<T>

```
public class Stack<T> where T : IComparable
{
    T[] items;
    // Pozostała część klasy.
}
```

Listing 11.47. Kod CIL klasy Stack<T>

```
.class private auto ansi beforefieldinit
    Stack'1<([mscorlib]System.IComparable)T>
    extends [mscorlib]System.Object
{
    ...
}
```

Pierwszym wartym uwagi fragmentem jest człon '1 pojawiający się po nazwie Stack w drugim wierszu. Podana wartość to arność, czyli liczba określających typ parametrów wymaganych w danej klasie generycznej. Dla klasy EntityDictionary<TKey, TValue> arność będzie równa 2.

W drugim wierszu wygenerowanego kodu CIL znajdują się też ograniczenia stawiane klasie. Określający typ parametr T jest powiązany z interfejsem, ponieważ ograniczenie wymaga implementacji interfejsu IComparable w danym typie.

Z dalszej analizy kodu CIL dowiesz się też, że do deklaracji tablicy items z elementami typu T zastosowano notację z wykrzyknikiem, wykorzystywaną w wersji kodu CIL z obsługą typów generycznych. Wykrzyknik oznacza obecność pierwszego określającego typ parametru danej klasy (zobacz listing 11.48).

2.0

Listing 11.48. Kod CIL z notacją z wykrzyknikiem oznaczającą obsługę typów generycznych

```
.class public auto ansi beforefieldinit
    'Stack'1'<([mscorlib]System.IComparable) T>
    extends [mscorlib]System.Object
{
    .field private !0[ ] items
    ...
}
```

Oprócz arności, parametru określającego typ w nagłówku klasy i tegoż parametru wyróżnionego wykrzyknikiem kod CIL wygenerowany dla klasy generycznej prawie się nie różni od kodu CIL klasy niegenerycznej. ■

ZAGADNIENIE DLA ZAAWANSOWANYCH**Tworzenie obiektów typów generycznych opartych na typach bezpośrednich**

Gdy tworzony jest pierwszy obiekt typu generycznego i jako parametr określający typ używany jest typ bezpośredni, środowisko uruchomieniowe tworzy wyspecjalizowany typ generyczny z podanymi parametrami umieszczonymi w odpowiednich miejscach kodu CIL. Tak więc

środowisko uruchomieniowe tworzy nowe wyspecjalizowane typy generyczne dla każdego typu bezpośredniego podanego jako parametr.

Założmy, że w kodzie zadeklarowana jest klasa `Stack` z parametrem `int`, tak jak na listingu 11.49.

Listing 11.49. Definicja klasy `Stack<int>`

```
Stack<int> stack;
```

Gdy używasz typu `Stack<int>` po raz pierwszy, środowisko uruchomieniowe generuje wyspecjalizowaną wersję klasy `Stack`, w której określający typ argument `int` jest podstawiany za parametr określający typ. Później za każdym razem, gdy kod używa typu `Stack<int>`, środowisko uruchomieniowe ponownie wykorzystuje wygenerowaną wyspecjalizowaną klasę `Stack` ↪ `<int>`. Na listingu 11.50 zadeklarowane są dwa obiekty typu `Stack<int>`. Dla obu używany jest wygenerowany już przez środowisko uruchomieniowe kod klasy `Stack<int>`.

2.0

Listing 11.50. Deklarowanie zmiennych typu `Stack<T>`

```
Stack<int> stackOne = new Stack<int>();
Stack<int> stackTwo = new Stack<int>();
```

Jeśli dalej w kodzie utworzysz nowy obiekt typu `Stack`, z innym typem bezpośrednim (na przykład typem `long` lub strukturą zdefiniowaną przez użytkownika) podstawianym za parametr określający typ, środowisko uruchomieniowe wygeneruje inną wersję typu generycznego. Zaletą wyspecjalizowanych klas generycznych opartych na typach bezpośrednich jest ich wydajność. Ponadto w kodzie można uniknąć konwersji i opakowywania, ponieważ każda wyspecjalizowana klasa generyczna „natywnie” korzysta z typu bezpośredniego. ■

■ ZAGADNIENIE DLA ZAAWANSOWANYCH

Tworzenie obiektów typów generycznych opartych na typach referencyjnych

Typy generyczne oparte na typach referencyjnych działają nieco inaczej. Gdy po raz pierwszy tworzony jest obiekt typu generycznego opartego na typie referencyjnym, środowisko uruchomieniowe tworzy wyspecjalizowany typ generyczny, w którym w kodzie CIL za parametry określające typ podstawiany jest typ `object` (a nie typ określony w argumencie). Później za każdym razem, gdy tworzony jest obiekt danego typu generycznego opartego na typie referencyjnym, środowisko uruchomieniowe ponownie wykorzystuje wcześniej wygenerowaną wersję tego typu generycznego — także wtedy, jeśli ten typ referencyjny jest inny niż wcześniej.

Założmy, że dostępne są dwa typy referencyjne — klasa `Customer` i klasa `Order`. Kod tworzy obiekt typu `EntityDictionary` z elementami typu `Customer`:

```
EntityDictionary<Guid, Customer> customers;
```

Zanim będzie można uzyskać dostęp do tej klasy, środowisko uruchomieniowe tworzy wyspecjalizowaną wersję klasy `EntityDictionary`, przy czym jako podany typ danych wykorzystuje typ `object`, a nie typ `Customer`. Założmy teraz, że następny wiersz kodu tworzy obiekt typu `EntityDictionary` oparty na innym typie referencyjnym — `Order`.


```
EntityDictionary<Guid, Order> orders =
    new EntityDictionary<Guid, Order>();
```

Inaczej niż w przypadku typów bezpośrednich, tu nie jest tworzona nowa wyspecjalizowana wersja klasy `EntityDictionary`, wykorzystująca typ `Order`. Zamiast tego tworzony jest obiekt wersji typu `EntityDictionary` opartej na typie `object` — i to ten obiekt jest przypisywany do zmiennej `orders`.

Aby móc uzyskać bezpieczeństwo ze względu na typ, dla każdej referencji typu `object` podstawionej za parametr określający typ alokowany jest w pamięci obszar potrzebny na typ `Order` i tworzony jest wskaźnik do tego obszaru.

Przyjmijmy, że natrafiłeś na wiersz kodu tworzący obiekt typu `EntityDictionary` opartej na typie `Customer`:

```
customers = new EntityDictionary<Guid, Customer>();
```

Podobnie jak wcześniej, gdy tworzono obiekt typu `EntityDictionary` opartego na typie `Order`, powstaje następny obiekt wyspecjalizowanej klasy `EntityDictionary` (z referencjami typu `object`), a wskaźniki z tego obiektu są ustawiane na typ `Customer`. Taka implementacja typów generycznych znacznie zmniejsza ilość kodu, ponieważ ogranicza do jednej liczby wyspecjalizowanych klas tworzonych przez kompilator na podstawie klas generycznych opartych na typach referencyjnych.

Choć środowisko uruchomieniowe wykorzystuje tę samą wewnętrzną definicję typu generycznego, gdy jako parametry określające typ podane są różne typy referencyjne, sytuacja wygląda inaczej, gdy jako takie parametry używane są różne typy bezpośrednie. Na przykład klasy `Dictionary<int, Customer>`, `Dictionary<Guid, Order>` i `Dictionary<long, Order>` wymagają osobnych wewnętrznych definicji typów.

Porównanie języków — typy generyczne w Javie

Implementacja typów generycznych w Javie jest w całości obsługiwana przez kompilator, a nie przez maszynę wirtualną Javy. Firma Sun zastosowała to podejście, by uniknąć konieczności dystrybucji zaktualizowanej wersji maszyny wirtualnej Javy po zastosowaniu typów generycznych.

W Javie dla typów generycznych używana jest składnia podobna jak dla szablonów z języka C++ i typów generycznych z języka C# (włącznie z parametrami określającymi typ i ograniczeniami). Jednak ponieważ typy bezpośrednie wyglądają w składni tak samo jak typy referencyjne, niezmodyfikowana maszyna wirtualna Javy nie obsługuje typów generycznych opartych na typach bezpośrednich. Dlatego typy generyczne w Javie nie dają takiego wzrostu wydajności kodu co w języku C#. Gdy kompilator Javy musi zwrócić dane, przeprowadza automatyczne rzutowanie w dół z typu podanego w ograniczeniu (jeśli istnieje) lub z typu bazowego `Object` (jeżeli nie ma ograniczenia). Ponadto kompilator Javy generuje jeden wyspecjalizowany typ na etapie kompilacji, a następnie korzysta z tego typu do tworzenia obiektów dowolnej wersji typu generycznego. Poza tym ponieważ maszyna wirtualna Javy nie ma wbudowanej obsługi typów generycznych, nie ma sposobu na sprawdzenie w czasie wykonywania programu parametru określającego typ w danym obiekcie typu generycznego. Inne zastosowania mechanizmu refleksji w typach generycznych też są mocno ograniczone.

2.0

2.0

Podsumowanie

Dodanie generycznych typów i metod w wersji C# 2.0 znacznie zmieniło sposób pisania kodu przez programistów używających języka C#. W prawie wszystkich sytuacjach, w których w wersji C# 1.0 programiści używali typu `object`, od wersji C# 2.0 typy generyczne stały się lepszym rozwiązaniem. Jeśli w obecnie rozwijanych programach w języku C# używany jest typ `object` (zwłaszcza w kolekcjach), należy się zastanowić, czy lepszym rozwiązaniem nie będzie zastosowanie typów generycznych. Większe bezpieczeństwo ze względu na typ, uzyskane dzięki możliwości rezygnacji z rzutowania, wyeliminowanie spadku wydajności związanego z opakowywaniem i zmniejszenie ilości powtarzającego się kodu, to istotne korzyści zapewniane przez typy generyczne.

W rozdziale 16. omówiono jedną z najczęściej używanych przestrzeni nazw z typami generycznymi — `System.Collections.Generic`. Jak wskazuje nazwa, ta przestrzeń nazw obejmuje prawie wyłącznie typy generyczne. Znajdziesz tam dobre przykłady ilustrujące, jak niektóre typy używające wcześniej typu `object` przekształcono w typy generyczne. Jednak zanim przejdziesz do tych zagadnień, warto przyjrzeć się wyrażeniom, które od wersji C# 3.0 znacznie usprawniły pracę z kolekcjami.

Koniec
2.0



Skorowidz

.NET Core, 801

A

adresy, 766, 769

agregacja, 282

akcesor tablicy, 95

akronimy, 796

alias, 180

przestrzeni nazw, 380

alokowanie danych, 772

analiza typów zmiennoprzecinkowych, 62

anulowanie

kooperatywne, 686

kwerendy PLINQ, 724, 725

pętle równoległej, 718

zadania, 686

API, Application Programming Interface, 50

APM, Asynchronous Programming Model, 810

argument, 167, 170

arność, 429

asynchroniczne

lambdy, 702

wywoływanie, 659

wywoływanie delegatów, 821

wywoływanie operacji, 694

zadania, 671

asynchroniczność, 670, 692

oparta na zadaniach, 697, 779

oparta na zdarzeniach, 824

atribut, 324, 624

FlagsAttribute, 351, 635

IndexerName, 596

MethodImplAttribute, 737

return, 627

STAThreadAttribute, 753

StructLayoutAttribute, 759

System.AttributeUsageAttribute, 633

System.ConditionalAttribute, 637

System.NonSerializable, 641

System.ObsoleteAttribute, 638

System.SerializableAttribute, 644

ThreadStaticAttribute, 750

atributy

niestandardowe, 627

podzespołu, 626

predefiniowane, 636

związane z serializacją, 639

automatycznie

implementowane wartości, 232

implementowane właściwości, 239

ujednolicane interfejsy, 778

B

badanie drzewa wyrażań, 484

bajt, 136

BCL, Base Class Library, 52, 794

bezpieczeństwo, 787

biblioteka

BCL, 52

PCL, 375

TPL, 694, 809, 815–819

WinRT, 776

biblioteki klas, 373, 375

bit, 136

blok kodu, 124, 126
 catch, 202, 206, 403, 405
 finally, 203
 kontrolowany, 85
 niekontrolowany, 86, 413
 try, 202
 blokada, 662, 736
 błąd nieskończonej rekurencji, 192
 błędny typ docelowy, 769
 błędy, 159, 199
 związane z tablicami, 102

C

cechy
 typów zmiennoprzecinkowych, 111
 zegarów, 836
 centralizowanie inicjowania, 251
 CIL, Common Intermediate Language, 51, 782
 CLI, Common Language Infrastructure, 781, 782
 CLR, Common Language Runtime, 98, 786
 CLS, Common Language Specification, 51, 794
 CPS, Continuation Passing Style, 814
 CPU, central processing unit, 657
 CTS, Common Type System, 51, 793

D

dane
 wejściowe, 44
 wyjściowe, 44, 55
 zarządzane, 786
 definiowanie
 finalizatora, 388
 indeksera, 595
 interfejsu, 305
 iteratora, 599
 klasy, 37
 klasy abstrakcyjnej, 295
 klasy częściowej, 269
 klasy generycznej, 422
 klasy zagnieżdżonej, 266, 270
 konstruktora, 426
 niestandardowego atrybutu, 627
 niestandardowych konwersji, 278
 operatora indeksowania, 597
 operatorów rzutowania, 278, 371
 przestrzeni nazw, 377, 378

standardu CLI, 782
 typu, 37
 typu wyczerpieniowego, 345
 właściwości, 239
 wyczerpienia, 344
 deklarowanie
 aliasu typu, 180
 funkcji, 758
 generycznego typu delegata, 506
 klasy, 216
 konstruktora, 244
 metody, 172
 metody Main, 38
 ograniczenia, 433
 parametrów formalnych, 174
 pól, 219
 pól jako zmiennych, 738
 stałej, 119
 struktury, 331
 tablicy, 91
 tablicy dwuwymiarowej, 94
 typu delegata, 461
 właściwości, 231
 wskaźników, 768
 zdarzeń, 504
 zmiennej, 41, 42
 dekrementacja, 115, 118
 delegat, 457, 460, 483, 507
 Action, 473
 AsyncCallback, 812
 Func, 473
 delegaty
 ogólnego przeznaczenia, 473
 synchroniczne, 671
 typu multicast, 489, 490, 497, 499
 dereferencja wskaźników, 772
 deserializacja, 643
 deterministyczna finalizacja, 389
 deterministyczny destruktor, 393
 diagram
 interfejsów, 320
 klasy, 524
 CancellationTokenSource, 689
 Dictionary, 587
 LinkedList<T>, 595
 List<>, 581
 Queue<T>, 594
 SortedList<>, 592
 Stack<T>, 593
 Venna, 541

długość
łańcuchów znaków, 75
tablicy, 96

dodawanie
atrybutów, 625
dyrektywy using, 178
komentarzy, 48
ograniczeń, 444
operatora, 367

dokumentacja, 383

dołączanie kodu, 156

domeny aplikacji, 790

domknięcie, 479

dostęp
do interfejsu użytkownika, 831
do metadanych, 614
do pola statycznego, 255
do pól instancji, 220
do składowych, 774
do właściwości, 333

drzewo wyrażeń, 481–483

dynamiczne
programowanie, 645
wywoływanie składowej, 617, 624

dyrektywa
#elif, 157
#else, 157
#endregion, 161
#error, 157
#if, 157
#line, 157, 160
#pragma, 157, 160
#region, 157, 161
#undef, 157
#warning, 157
extern alias, 381
using, 72, 176, 178
using static, 72, 74, 179

dyrektywy preprocesora, 156, 157

działanie iteratorów, 608

dziedziczenie, 214, 273
interfejsów, 315
klas wyjątków, 205
po jednym typie, 281
po wielu interfejsach, 317

dzielenie przez zero, 113

E

EAP, Event-based Asynchronous Pattern, 824

F

FCL, Framework Class Library, 794

FIFO, 594

filtrowanie, 530, 558, 565

finalizacja, 392
deterministyczna, 389

finalizator, 248, 388, 426

format szesnastkowy, 65

formatowanie
dwustronne, 66
kodu, 41
łańcuchów znaków, 74
złożone, 47

funkcja VirtualAllocEx(), 758

funkcje
anonimowe, 466
zewnętrzne, 756, 763

G

generowanie
błędów, 159
pliku z dokumentacją, 383
typów anonimowych, 518

generyczne interfejsy, 424

getter, 230, 241

gra w kółka i krzyżyk, 803

grafy obiektów, 482

grupowanie, 568
instrukcji, 166
wyników, 547

H

hermetyzacja, 214, 218, 228, 788
danych, 263
procesu publikacji, 504
subskrypcji, 503
typów, 375

hierarchia
generycznych interfejsów, 579
klas, 215

I

identyfikatory, 35
 iloczyn kartezjański, 545, 572
 implementacje standardu CLI, 783
 implementowanie
 generycznego interfejsu, 425
 interfejsów, 305, 309, 583, 590
 metod, 363
 operatorów, 366
 opóźnionego wykonywania, 565
 typu dynamicznego, 651
 wielodziedziczenia, 319
 wielokrotne interfejsu, 425
 zdarzeń, 511
 złączeń, 549
 zmiennych zewnętrznych, 478
 indeks, 89
 indeksy, 594
 inferencja typów, 443
 informacje o tablicach, 90
 inicjator, 251
 kolekcji, 247, 519
 obiektów, 247
 inicjowanie
 atrybutu, 629
 pola, 427
 struktur, 332
 tablicy tablic, 95
 inkrementacja, 115, 118
 instalacja xcopsy, 792
 instancja, 217
 delegata, 462, 463
 klasy, 217
 instrukcja
 asynca, 697
 awaia, 697
 await, 712
 break, 121, 151
 continue, 120, 153
 do while, 120
 dynamic, 645
 fixed, 771
 for, 120
 foreach, 120, 522, 523
 goto, 121, 154
 if, 120, 121, 124
 Join(), 544

lock, 339
 return, 175
 switch, 121, 148
 System.Console.WriteLine(), 200
 throw, 208
 TryParse(), 210
 using, 391
 while, 120
 yield, 610
 yield break, 607
 yield return, 605
 instrukcje, 39, 171
 bez średników, 39
 if/zagnieżdżone, 122
 if/else, 123
 skoku, 150
 interfejs, 303, 323, 324
 API, 764, 827
 API Win32, 760
 API WinRT, 777
 bazowy, 316
 ICollection<T>, 579
 IComparer<T>, 583
 IDictionary<TKey, 578
 IDisposable, 390
 IEnumerable, 527
 IEnumerable<T>, 514, 522, 577
 IEnumerator, 524
 IEqualityComparer<T>, 590
 IList<T>, 578
 IQueryable<T>, 554
 pochodny, 315
 TValue>, 578
 użytkownika, 830
 interfejsy
 API, 50
 kolekcji, 513
 interpolacja łańcuchów znaków, 71, 72
 iteratory, 598, 603
 rekurencyjne, 606
 iterowanie, 524, 527
 po kolekcji, 589

J

jawne rzutowanie, 84, 277
 jawnie podawany interfejs, 311, 313

język

- CIL, 51, 54, 782, 792
- UML, 215

języki źródłowe, 793

- JIT, just-in-time, 784

K

kategorie typów, 81, 327

klasa, 167, 213, 323

- BackgroundWorker, 827
- BinaryTree<T>, 430
- CancellationToken, 687, 689
- CancellationTokenSource, 689
- Dictionary, 587
- Interlocked, 739
- LinkedList<T>, 595
- LinkedListNode<T>, 595
- List<T>, 580
- MemberInfo, 621
- Monitor, 732
- Mutex, 743
- object, 355
- Queryable, 554
- Queue<T>, 594
- SortedDictionary<TKey, TValue>, 592
- Stack, 418
- Stack<T>, 453, 593, 621
- System.Collections.Stack, 416
- System.Object, 300
- System.Threading, 663
- System.Threading.Interlocked, 738
- System.Threading.Thread, 663
- System.Threading.Timer, 838
- System.Timers.Timer, 837
- WaitHandle, 744

klasy

- abstrakcyjne, 294, 324
- bazowe, 274
- częściowe, 268
- dziedziczące, 275
- generyczne, 421
- kolekcji, 580, 748
- pochodne, 275
- publikujące zdarzenia, 491
- słownika, 586
- statyczne, 260
- zagnieżdżone, 266
- zamknięte, 284

klauzula

- from, 572
- into, 571
- let, 567
- orderby, 566

kod

- natywny, 51
- niezabezpieczony, 767, 775
- zarządzany, 786
- źródłowy gry, 803

kodowanie Unicode, 68

kolejka, 594

- finalizacji, 392

kolekcja typu List<T>, 584

kolekcje, 519

- niestandardowe, 577
- posortowane, 592
- w postaci listy, 580

komentarze, 48

- jednowierszowe, 49
- XML-owe, 49, 381, 382
- z ogranicznikami, 49

kompilacja, 800

- kodu, 784
- statyczna, 650

kompilator, 783, 799

- JIT, 784

kompilowanie aplikacji, 32

komputer wirtualny, 776

komunikaty z ostrzeżeniami, 159

konfigurowanie ścieżki kompilatora, 800

konsola, 44

konstruktor, 244, 293, 426

- atrybutu, 629

konstruktory

- domyślne, 246
- statyczne, 258

kontekst synchronizacji, 708

kontekstowe słowa kluczowe, 609

kontrawariancja, 446, 449

kontrola

- konwersji, 413
- typów, 787

kontynuowanie

- kwerendy, 571
- zadania, 675

konwencje programistyczne, 506

konwersja, 347
 danych, 301, 302
 drzewa wyrażań, 482
 kontrolowana, 85
 kowariantna, 446
 niejawna, 87, 277
 niekontrolowana, 85
 typów bez rzutowania, 87
 typów danych, 84
 kowariancja, 446, 447
 kwant czasu, 659
 kwerenda
 filtrująca, 566
 LINQ, 721
 PLINQ, 724

L

lambda, 472, 702
 bezparametrowe, 469
 w postaci instrukcji, 467
 w postaci wyrażań, 469
 latencja, 655, 692, 694
 leniwe inicjowanie, 394
 LIFO, 416, 524, 593
 LINQ, Language Integrated Query, 15, 483, 533
 lista, 188
 subskrybentów, 502
 literały, 69
 liczbowe, 62
 szesnastkowe, 65
 lokalizacja danych, 770

Ł

łańcuch
 wywołań konstruktorów, 250
 znaków, 44, 69
 formatowania, 47
 jako tablice, 100
 niezmiennosc, 76
 łączenie
 interfejsów API, 764
 konstruktorów, 250
 ograniczeń, 439
 przypisania, 369
 subskrybentów z nadawcą, 491
 łączność, 108

M

manifesty, 790
 mapa myśli, 21
 maska, 138
 mechanizm
 CAS, 788
 delegatów, 463
 delegatów typu multicast, 497
 odzyskiwania pamięci, 218
 P/Invoke, 756
 refleksji, 614
 typów generycznych, 452
 właściwości, 243
 wyrażań lambda, 475
 zdarzeń, 509
 metadane, 614, 794
 metoda, 37, 165
 add, 777
 BubbleSort(), 458, 459, 460
 Console.ReadLine(), 93
 ContinueWith(), 679, 819
 Count(), 534
 Equals(), 359, 363, 364
 FindAll(), 585
 GetEnumerator(), 602
 GetHashCode(), 356, 358
 GetResponseAsync(), 699
 GetType(), 615
 GroupBy(), 547
 GroupJoin(), 548, 549
 Main(), 38, 181
 OrderBy(), 539
 Pop(), 593
 Push(), 593
 remove, 777
 Select(), 531, 722
 SelectMany(), 551
 System.Console.Read(), 45
 System.Linq.Enumerable.Select(), 532
 System.Linq.Enumerable.Where(), 530
 Task.ContinueWith(), 676, 708
 Task.Delay(), 753
 Task.Factory.StartNew(), 690
 ThenBy(), 539
 Thread.Sleep(), 666
 ToString(), 355, 356, 518
 TryParse(), 88

Wait(), 689
 Where(), 530

metody

- anonimowe, 466, 471, 814
- anonimowe bezparametrowe, 472
- asynchroniczne, 704, 705
- częściowe, 270
- dla tablic, 97, 98
- dla typu string, 72
- fabryczne, 429
- generyczne, 442, 445
- instancji, 221
- instancyjne tablice, 99
- klasy Interlocked, 739
- rozszerzające, 262, 281, 317
- statyczne, 256
- statyczne typu string, 73
- typu string, 74
- z ciałem w postaci wyrażenia, 175
- zewnętrzne, 756

miejsce wywołania, 182

model

- programowania sekwencyjnego, 17
- programowania ustrukturyzowanego, 17
- wątkowy, 658

moduły, 790

modyfikator

- async, 710
- await, 707, 710
- const, 263
- dostępu private, 229, 279
- dostępu protected, 280
- formatowania R, 66
- in, 449
- new, 288, 290
- out, 447
- override, 290
- protected internal, 377
- readonly, 264
- sealed, 292
- virtual, 284

modyfikatory dostępu, 228, 241, 376, 377

modyfikowanie

- implementacji zdarzeń, 511
- łańcuchów znaków, 773
- wartości zmiennej, 43

monada, 530

monitor, 732

MTA, Multi-threaded Apartment, 754

N

nadtyp, 273

nakładki, 765

narzędzie ILDASM, 54

nawiasy, 108

nazwa

- indeksera, 596
- metody, 170
- parametru, 424
- typu, 169

niejawna konwersja, 84

niejawne rzutowanie, 276

niejawnie określany typ, 516

niepowtarzalne elementy, 572

niezabezpieczony kod, 755

notacja

- pascalowa, 35
- szesnastkowa, 65
- wykładnicza, 64

numery wierszy, 160

O

obiekt, 217

- typu BinaryTree<string>, 606
- typu System.Type, 614
- typu Task<T>, 673

obiektowy model typów delegatów, 464

obiekty typów generycznych, 453, 454

obsługa

- asynchroniczności, 692
- błędów, 199, 499, 526, 760
- kowariancji, 451
- nieobsłużonego wyjątku, 682
- wartości null, 420
- wyjątków, 203, 397, 405, 501, 830

odpytywanie cykliczne, 673

odstęp, 40

odwijanie stosu, 183

odzworowywanie wskaźników, 766

odzyskiwanie pamięci, 385, 392, 786, 787

ogólny blok catch, 207

ograniczenia, 430

- dotyczące delegatów, 439
- dotyczące dziedziczenia, 437
- dotyczące interfejsu, 432
- dotyczące klasy, 434

ograniczenia
 dotyczące konstruktora, 436, 440
 dotyczące operatorów, 438
 wymagające klasy, 434
 wymagające struktury, 434
 ograniczniki instrukcji, 39
 określanie
 wartości, 108
 wywoływanej metody, 198
 opakowywanie, 336, 340, 342
 interfejsu API, 765
 opcje pętli równoległych, 720
 operacja atomowa, 660
 operacje arytmetyczne, 110
 operator, 105
 ?, 133, 135
 ??, 132
 AND, 130
 as, 301
 await, 712
 default, 335, 427
 is, 301
 minus, 106
 nameof, 237, 623
 negacji, 131
 new, 246, 334
 OR, 129
 plus, 106
 typeof(), 616
 XOR, 130
 operatory
 bitowe, 135, 138, 152
 dekrementacji, 115
 dodawania dla łańcuchów znaków, 109
 dopełnienia, 140
 dwuargumentowe, 107, 367
 indeksowania, 597
 inkrementacji, 115
 jednoargumentowe, 106, 370
 konwersji, 371, 372
 kwerend, 527, 552, 554
 logiczne, 129
 porównania, 366
 priorytet, 163
 przesunięcia, 137
 przypisania, 114, 140
 relacyjne, 128
 równości, 128, 359

rzutowania, 84
 używane do delegatów, 495
 warunkowe, 131, 369
 opisywanie kodu, 382
 opóźnione wykonanie, 535, 565
 wyrażeń, 562
 ostrzeżenie, 159

P

pamięć lokalna wątku, 749, 750
 parametr
 out, 242
 ref, 242, 758
 parametry, 165, 167, 170
 formalne, 174
 generyczne, 622
 metod, 183
 metody Main(), 181
 nazwane, 634
 opcjonalne, 195
 przekazywane przez referencję, 185
 przekazywane przez wartość, 183
 wyjściowe, 186
 PCL, portable class library, 375
 pętla
 do/while, 141, 142
 for, 143, 714
 for z kilkoma wyrażeniami, 145
 foreach, 145, 527, 538, 606, 715
 Parallel.For(), 721
 while, 141, 142, 524
 pętle równoległe, 718, 720, 721
 pierwszy program, 32
 platforma
 .NET, 52, 385, 799, 801
 CLI, 51, 799
 Mono, 802
 plik Comments.xml, 384
 PLINQ, 533, 657, 726
 pobieranie danych z pliku, 226
 podkradanie pracy, 716
 podtyp, 273
 podzespoły, 374, 790
 pola, 75, 219, 234
 instancji, 218, 219, 255
 statyczne, 254
 wirtualne, 239

polimorfizm, 298, 305
 operacyjny, 193
 porządkowanie
 całkowite, 584
 zasobów, 525, 526
 powiadomienia o zdarzeniu, 507, 740
 powiązanie, 219
 późne wiązanie, 795
 precyzja typów zmiennoprzecinkowych, 112
 predefiniowane atrybuty, 636
 predykat, 530, 565
 priorytety, 108
 operatorów, 163
 proces, 657
 procesor, 657
 programowanie
 dynamiczne, 644, 650
 obiektywne, 214
 równoległe, 659
 tablic, 103
 programy szeregujące zadania, 671, 708
 projekcja, 531, 560
 przechwytywanie
 błędów, 200
 wyjątków, 201, 400, 404
 zmiennych, 479, 480
 przeciążanie
 konstruktorów, 248
 metod, 193
 operatorów, 365
 przekazywanie
 danych przez referencję, 502
 listy, 188
 stanu, 814
 wyjątków, 393
 przenośne biblioteki klas, 375
 przenośność między platformami, 788
 przepełnienie
 bufora, 98
 typu całkowitoliczbowego, 85, 412
 przepływ sterowania, 105, 119, 203, 657, 700
 przesłanianie
 metody, 355, 356, 359, 364
 operatora równości, 359
 składowych, 284, 355
 właściwości, 285
 przestrzeń
 deklaracji, 126
 nazw, 168, 176, 377

przeszukiwanie kolekcji, 584
 przetwarzanie równoległe, 748
 przypisywanie wartości, 43
 do wskaźników, 769
 pula
 pamięci tymczasowej, 328
 wątków, 658, 668
 pusta kolekcja, 598

R

RCW, runtime callable wrapper, 754
 refaktoryzacja, 173, 274
 metody, 145
 referencje, 185
 do obiektów, 385
 mocne, 386
 słabe, 386
 refleksja, 614, 622, 788, 795
 rejestrowanie, 685
 rekurencja, 190
 nieskończona, 192
 relacja
 jeden do wielu, 542, 548
 LUB, 439
 wiele do wielu, 541
 reprezentacja typów generycznych, 452
 równoległe wykonywana metoda, 722
 równoległe
 kwerendy LINQ, 723
 wykonywanie iteracji, 713
 wykonywanie kwerend, 533
 wykonywanie kwerend LINQ, 721
 wykonywanie pętli, 716
 równość strukturalna, 474
 rzutowanie, 84, 276, 277, 418, 445
 jawne, 84

S

sekwencja
 ucieczki, 68
 wywołań delegatów, 498
 semafor, 747
 serializacja, 410, 639
 niestandardowa, 641
 setter, 230, 241
 składnia, 33
 iteratora, 599

składowa base, 292
 składowe

- abstrakcyjne, 295
- bez jawnie podawanego interfejsu, 312
- klasy System.Object, 300
- prywatne, 228
- statyczne, 253
- z jawnie podawanym interfejsem, 311

 słownik, 586
 słowo kluczowe, 34, 36

- async, 835
- await, 835
- class, 216
- lock, 734
- new, 92
- null, 78
- this, 222, 224, 736

 sortowanie, 459, 539, 566

- kolekcji, 582

 sprawdzanie

- poprawności, 235
- równości, 590, 591
- typu, 301

 stałe

- lokalne, 118
- publiczne, 264

 stan, 602
 standard

- CLI, 781, 783
- Unicode, 67

 standardowy operator kwerend, 527
 statyczna inicjacja, 259
 stos, 593, 772

- wywołań, 182

 stosowanie

- biblioteki TPL, 816
- refleksji, 622
- słabych referencji, 387
- wzorca, 829
- wzorca APM, 810

 struktury, 331, 424
 subskrybent, 501
 sygnatury, 812
 symbole preprocesora, 158
 synchroniczne wywołanie operacji, 692, 710
 synchronizacja, 728, 732, 737, 742

- dostępu, 731
- wątków, 727

 system VES, 782

Ś

środowisko

- uruchomieniowe, 786
- Visual Studio, 800
- zarządzane, 51

T

tablice, 89

- dwuwymiarowe, 94
- parametrów, 188, 189
- tablic, 95
- trójwymiarowe, 94

 TAP, Task-based Asynchronous Pattern, 657, 809
 technika duck typing, 527
 technologia

- COM, 753
- LINQ, 483, 533, 557, 722
- Windows Forms, 831

 tożsamość, 360

- obiektów, 359

 TPL, Task Parallel Library, 657
 TPL, Task Programming Library, 809
 tworzenie

- aliasów, 180
- delegata, 467
- identyfikatorów, 610
- instancji, 217
- instancji delegata, 462, 463
- instancji tablic, 92
- interfejsu pochodnego, 315, 322
- klas pochodnych, 274
- metody asynchronicznej, 703
- nazw parametrów, 424
- niestandardowego wyjątku, 408
- niestandardowych kolekcji, 577
- obiektów dynamicznych, 651
- obiektów typów generycznych, 454
- zarządzanych zasobów, 762

 typ

- (T), 422
- AggregateException, 717
- AutoResetEvent, 747
- CancellationToken, 688
- ComparisonHandler, 465
- decimal, 62
- dynamic, 646, 648

logiczny, bool, 67
 ManualResetEvent, 746, 747
 Pair<T>, 607
 pochodny, 273
 SafeHandle, 762
 Semaphore, 747
 string, 44, 72
 System.Text.StringBuilder, 77
 Task<T>, 674, 707
 ThreadLocal<T>., 749
 znakowy, char, 67
 typy
 anonimowe, 252, 514, 518, 561
 bazowe, 273
 bezpośrednie, 81, 184, 327, 328, 339
 całkowitoliczbowe, 60
 danych, 42, 59
 danych parametrów, 757
 docelowe, 769
 generyczne, 415, 420, 423, 455, 507
 zagnieżdżone, 430
 liczbowe, 59
 niejawne, 79
 niezarządzane, 769
 parametryzowane, 420
 referencyjne, 82, 184, 329
 wyjątków, 206, 397
 wyliczeniowe, 347
 zmiennoprzecinkowe, 61, 111
 zwracanej wartości, 174

U

ujednolicanie interfejsów, 778
 układ sekwencyjny, 759
 UML, Unified Modeling Language, 215
 Unicode, 67
 unikanie
 blokad, 742
 wypakowywania, 343
 zakleszczenia, 741
 uruchamianie aplikacji, 32
 ustalanie typów, 620
 usuwanie wątków, 667

V

VES, Virtual Execution System, 782

W

wariancja, 475
 wartości, 167
 domyślne, 427
 logiczne, 87
 void, 78
 wyliczeniowe, 344, 678, 679
 zwracane, 171
 wartość null, 78, 83, 419, 493
 wątek, 118, 495, 658
 główny, 730
 z dekrementacją, 730
 wątki robocze, 827
 wczytywanie plików, 225
 wersje, 321, 642
 języka, 52
 wewnętrzne mechanizmy zdarzeń, 509
 wiązanie dynamiczne, 649
 wielodziedziczenie, 283, 319
 wielowątkowość, 655, 809
 równoległa, 657
 Windows Forms, 831
 Windows Presentation Foundation, 833
 Windows RT, 776
 właściwości, 75, 230, 231, 234
 statyczne, 259
 tylko do odczytu, 75, 238
 tylko do zapisu, 238
 WPF, Windows Presentation Foundation, 833
 wskazywanie podzespołu, 374
 wskaźniki, 758, 766, 769
 dereferencja, 772
 przypisywanie wartości, 769
 wspinać, 716
 współdziałanie między platformami, 755
 współdzielony stan, 524
 wydajność, 659, 716, 789
 wyjątek, 199, 205, 397, 643, 830
 AggregateException, 681, 717
 TaskCanceledException, 689
 wyjątki
 nieobsłużone, 683, 684
 niestandardowe, 407
 z obsługą serializacji, 410
 wykonywanie
 długotrwałych zadań, 691
 niezabezpieczonego kodu, 775
 zarządzane, 786

wykrywanie nieobsłużonych wyjątków, 683
 wyliczenia, 343
 jako flagi, 349
 wypakowywanie, 337, 339
 wyrażenia
 lambda, 457, 466, 470
 logiczne, 127
 o stałej wartości, 118
 warunkowe wyjątku, 401
 z kwerendami, 557–563, 573
 z kwerendą filtrującą, 566
 wyszukiwanie
 atrybutów, 628
 elementów, 585
 wyścig, 661
 wyświetlanie
 danych, 46
 liczb, 65
 wywoływanie
 asynchronicznego zadania, 672
 delegatów, 492–495, 821
 funkcji zewnętrznych, 763
 inicjatorów obiektów, 248
 instrukcji using, 391
 konstruktora, 245
 metod, 166, 171, 815, 820
 obiektów, 833
 operatorów dwuargumentowych, 368
 sekwencyjne, 497
 składowych, 624
 zmiennych, 616
 wzorzec
 APM, 810, 819, 820
 CPS, 812
 EAP, 824, 825
 obserwator, 490
 obsługi asynchroniczności, 692
 TAP, 697, 820

X

XML, Extensible Markup Language, 50

Z

zadania, 658
 asynchroniczne, 670, 671
 długotrwałe, 690
 poprzedzające, 677

zagnieżdżone
 dyrektywy using, 178
 instrukcje if, 122
 przeustrzenie nazw, 177, 379
 typy generyczne, 430
 zakleszczenie, 662, 741
 zapisywanie
 na sztywno, 63
 plików, 225
 zarządzanie
 pamięcią, 661
 wątkami, 665
 wersjami, 321, 642
 zasięg, 126, 170
 zmiennych lokalnych, 127
 zdarzenia, 489, 503, 740, 824
 resetujące, 744
 zdarzenie ManualResetEventSlim, 745
 zegary, 752, 835
 zestawy ograniczeń, 435
 zgłaszanie
 błędów, 208
 opakowanego wyjątku, 411
 ponowne wyjątku, 401
 wyjątku, 209, 398, 402
 zliczanie elementów, 534
 złączenie
 wewnętrzne, 541, 544, 546
 zewnętrzne lewostronne, 541
 zewnętrzne pełne, 541
 zewnętrzne prawostronne, 541
 zmienna, 41
 var, 515
 zmienne
 lokalne, 41, 79, 514, 515
 lokalne bez synchronizacji, 732
 pętli, 143
 przechwytywane, 476
 składowe, 219
 zakresowe, 559
 zewnętrzne, 476, 478
 zwalnianie zasobów, 691, 815
 zwracanie
 wartości, 181, 731
 wartości null, 598
 wartości przez iterator, 600

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



- 1. ZAREJESTRUJ SIĘ**
- 2. PREZENTUJ KSIĄŻKI**
- 3. ZBIERAJ PROWIZJĘ**

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

GRUPA WYDAWNICZA

 **Helion SA**

Sprawdź, jak wygodnym i niezawodnym językiem jest C#!

C# jest prostym, nowoczesnym, bezpiecznym językiem programowania, który powstał na bazie języków C i C++, jednak otrzymał też najlepsze cechy takich języków jak Visual Basic, Object Pascal, Delphi czy Java. Został od podstaw zaprojektowany jako obiektowy. C# stanowi część platformy Microsoft .NET Framework. Ta dojrzała technologia pozwala na efektywne tworzenie kodu bezpiecznego, przejrzystego, wydajnego i prostego w konserwacji.

Niniejsza książka to bardzo praktyczne kompendium wiedzy o języku C#. Została oparta na podstawowej specyfikacji C# Language 6.0. Zawiera kompletne omówienie języka. Książkę pomyślano jako podręcznik, dzięki któremu szybko można rozpocząć praktyczną pracę nad projektami programistycznymi. Osoby znające C# będą mogły zapoznać się ze skomplikowanymi paradygmatami programowania, a także przejrzeć szczegółowe omówienie funkcji wprowadzonych w najnowszej wersji języka, C# 6.0, oraz w platformie .NET Framework 4.6. Ponadto każdy, kto pracuje w C#, znajdzie tu doskonale zorganizowane źródło wiedzy o tym potężnym języku.

W książce przedstawiono:

- kompletne omówienie elementów języka C#
- jasne wskazówki dotyczące implementowania niezawodnej obsługi błędów
- metody zmniejszania złożoności kodu
- zasady programowania dynamicznego z wykorzystaniem refleksji i atrybutów
- opis wykorzystania możliwości platformy .NET, w tym omówienie kolekcji i standardu CLI
- powiązania kodu C# z wykorzystywanym środowiskiem uruchomieniowym

MARK MICHAELIS jest założycielem firmy IntelliTect. Laureat nagrody Microsoft MVP, dyrektor regionalny Microsoftu. Prowadzi prelekcje na konferencjach dla programistów i jest autorem wielu książek.

ERIC LIPPERT jest programistą w zespole odpowiedzialnym za analizę języka C# w firmie Coverity/Synopsys. Wcześniej był głównym programistą w zespole pracującym nad kompilatorem języka C# w Microsoftzie i członkiem zespołu projektującego ten język. Brał też udział w projektowaniu i implementowaniu wielu innych technologii w Microsoftzie.

 Addison-Wesley

Helion

46062 numer katalogowy

księgarnia internetowa

<http://helion.pl>

zamówienia telefoniczne



0 801 339900



0 601 339900

Informatyka w najlepszym wydaniu

Sprawdź najnowsze promocje:
● <http://helion.pl/promocje>
Książki najchętniej czytane:
● <http://helion.pl/bestsellery>
Zamów informacje o nowościach:
● <http://helion.pl/nowosci>

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
<http://helion.pl>

sięgnij po WIĘCEJ



KOD KORZYŚCI

ISBN 978-83-283-2518-0



9 788328 325180

cena: 129,00 zł

THE ADDISON-WESLEY

Microsoft
Technology
Series