

WYDANIE VI



# C# 7.0

Kompletny przewodnik dla praktyków

**MARK MICHAELIS**

**ERIC LIPPERT**, redaktor techniczny

Przedmowa **MADS TORGERSEN**,

menedżer ds. prac nad językiem C#  
w firmie Microsoft



**Helion**

Tytuł oryginału: Essential C# 7.0 (6th Edition)

Tłumaczenie: Tomasz Walczak

ISBN: 978-83-283-5780-8

Authorized translation from the English language edition, entitled ESSENTIAL C# 7.0, 6th Edition; by MICHAELIS, MARK; , published by Pearson Education, Inc, publishing as Addison-Wesley Professional.

Copyright © 2018 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 S.A. Copyright © 2019.

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 Helion SA 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 Helion SA nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Helion SA

ul. Kościuszki 1c, 44-100 Gliwice

tel. 32 231 22 19, 32 230 98 63

e-mail: [helion@helion.pl](mailto: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/c7kop6>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:

<ftp://ftp.helion.pl/przyklady/c7kop6.zip>

Printed in Poland.

- Kup książkę
- Poleć książkę
- Oceń książkę

- Księgarnia internetowa
- Lubię to! » Nasza społeczność



# Spis treści

---

<i>Spis rysunków</i>	11
<i>Spis tabel</i>	13
<i>Przedmowa</i>	15
<i>Wprowadzenie</i>	17
<i>Podziękowania</i>	27
<i>O autorze</i>	29

## **1. Wprowadzenie do języka C# 31**

Witaj, świecie	32
Podstawy składni języka C#	40
Korzystanie ze zmiennych	47
Dane wejściowe i wyjściowe w konsoli	51
Wykonywanie kodu w środowisku zarządzanym i platforma CLI	57
Różne wersje platformy .NET	62
Podsumowanie	64

## **2. Typy danych 65**

Podstawowe typy liczbowe	65
Inne podstawowe typy	73
Wartości null i void	85
Konwersje typów danych	86
Podsumowanie	92

## **3. Jeszcze o typach danych 93**

Kategorie typów	93
Modyfikator umożliwiający stosowanie wartości null	96
Krotki	98
Tablice	104
Podsumowanie	118

**4. Operatory i przepływ sterowania 119**

Operatory	120
Zarządzanie przepływem sterowania	133
Bloki kodu ({})	138
Bloki kodu, zasięgi i przestrzenie deklaracji	140
Wyrażenia logiczne	142
Operatory bitowe (<<, >>,  , &, ^, ~)	150
Instrukcje związane z przepływem sterowania — ciąg dalszy	155
Instrukcje skoku	165
Dyrektywy preprocesora języka C#	170
Podsumowanie	177

**5. Metody i parametry 179**

Wywoływanie metody	180
Deklarowanie metody	185
Dyrektywa using	190
Zwracane wartości i parametry metody Main()	195
Zaawansowane parametry metod	197
Rekurencja	207
Przeciążanie metod	209
Parametry opcjonalne	212
Podstawowa obsługa błędów z wykorzystaniem wyjątków	216
Podsumowanie	227

**6. Klasy 229**

Deklarowanie klasy i tworzenie jej instancji	232
Pola instancji	235
Metody instancji	237
Stosowanie słowa kluczowego this	238
Modyfikatory dostępu	244
Właściwości	246
Konstruktory	260
Składowe statyczne	269
Metody rozszerzające	277
Hermetyzacja danych	278
Klasy zagnieżdżone	281
Klasy częściowe	283
Podsumowanie	287

**7. Dziedziczenie 289**

Tworzenie klas pochodnych	290
Przesłanie składowych z klas bazowych	300
Klasy abstrakcyjne	310
Wszystkie klasy są pochodne od System.Object	315

Sprawdzanie typu za pomocą operatora is	316
Dopasowanie do wzorca z użyciem operatora is	317
Dopasowanie do wzorca w instrukcji switch	318
Konwersja z wykorzystaniem operatora as	320
Podsumowanie	321

## 8. Interfejsy 323

Wprowadzenie do interfejsów	324
Polimorfizm oparty na interfejsach	325
Implementacja interfejsu	329
Przekształcanie między klasą z implementacją i interfejsami	334
Dziedziczenie interfejsów	335
Dziedziczenie po wielu interfejsach	337
Metody rozszerzające i interfejsy	337
Implementowanie wielodziedziczenia za pomocą interfejsów	339
Zarządzanie wersjami	341
Interfejsy a klasy	343
Interfejsy a atrybuty	344
Podsumowanie	345

## 9. Typy bezpośrednie 347

Struktury	351
Opakowywanie	356
Wyliczenia	363
Podsumowanie	373

## 10. Dobrze uformowane typy 375

Przesłanianie składowych z klasy object	375
Przeciążanie operatorów	387
Wskazywanie innych podzespółów	394
Definiowanie przestrzeni nazw	402
Komentarze XML-owe	405
Odzyskiwanie pamięci	409
Porządkowanie zasobów	411
Leniwe inicjowanie	418
Podsumowanie	420

## 11. Obsługa wyjątków 421

Wiele typów wyjątków	421
Przechwytywanie wyjątków	424
Ogólny blok catch	427
Wskazówki związane z obsługą wyjątków	429
Definiowanie niestandardowych wyjątków	433
Ponowne zgłaszanie opakowanego wyjątku	435
Podsumowanie	439

**12. Typy generyczne 441**

- Język C# bez typów generycznych 442
- Wprowadzenie do typów generycznych 446
- Ograniczenia 457
- Metody generyczne 468
- Kowariancja i kontrawariancja 472
- Wewnętrzne mechanizmy typów generycznych 479
- Podsumowanie 482

**13. Delegaty i wyrażenia lambda 483**

- Wprowadzenie do delegatów 484
- Deklarowanie typu delegata 487
- Wyrażenia lambda 494
- Metody anonimowe 499
- Podsumowanie 513

**14. Zdarzenia 515**

- Implementacja wzorca publikuj-subskrybuj za pomocą delegatów typu multicast 516
- Zdarzenia 528
- Podsumowanie 538

**15. Interfejsy kolekcji ze standardowymi operatorami kwerend 539**

- Inicjatory kolekcji 540
- Interfejs IEnumerable<T> sprawia, że klasa staje się kolekcją 542
- Standardowe operatory kwerend 547
- Typy anonimowe w technologii LINQ 576
- Podsumowanie 583

**16. Technologia LINQ i wyrażenia z kwerendami 585**

- Wprowadzenie do wyrażeń z kwerendami 586
- Wyrażenia z kwerendą to tylko wywołania metod 601
- Podsumowanie 603

**17. Tworzenie niestandardowych kolekcji 605**

- Inne interfejsy implementowane w kolekcjach 606
- Podstawowe klasy kolekcji 608
- Udostępnianie indeksera 623
- Zwracanie wartości null lub pustej kolekcji 626
- Iteratory 627
- Podsumowanie 639

**18. Refleksja, atrybuty i programowanie dynamiczne 641**

- Mechanizm refleksji 642
- Operator nameof 651
- Atrybuty 652
- Programowanie z wykorzystaniem obiektów dynamicznych 672
- Podsumowanie 682

**19. Wielowątkowość 683**

- Podstawy wielowątkowości 685
- Używanie klasy System.Threading 691
- Zadania asynchroniczne 698
- Anulowanie zadania 715
- Wzorzec obsługi asynchroniczności za pomocą zadań 720
- Równoległe wykonywanie iteracji pętli 746
- Równoległe wykonywanie kwerend LINQ 754
- Podsumowanie 759

**20. Synchronizowanie wątków 761**

- Po co stosować synchronizację? 762
- Zegary 786
- Podsumowanie 788

**21. Współdziałanie między platformami i niezabezpieczony kod 789**

- Mechanizm P/Invoke 790
- Wskaźniki i adresy 801
- Wykonywanie niezabezpieczonego kodu za pomocą delegata 809
- Podsumowanie 810

**22. Standard CLI 811**

- Definiowanie standardu CLI 811
- Implementacje standardu CLI 812
- Specyfikacja .NET Standard 815
- Biblioteka BCL 815
- Kompilacja kodu w języku C# na kod maszynowy 816
- Środowisko uruchomieniowe 818
- Podzespoły, manifesty i moduły 821
- Język Common Intermediate Language 823
- Common Type System 824
- Common Language Specification 825
- Metadane 825
- Architektura .NET Native i kompilacja AOT 826
- Podsumowanie 826

*Skorowidz* 829





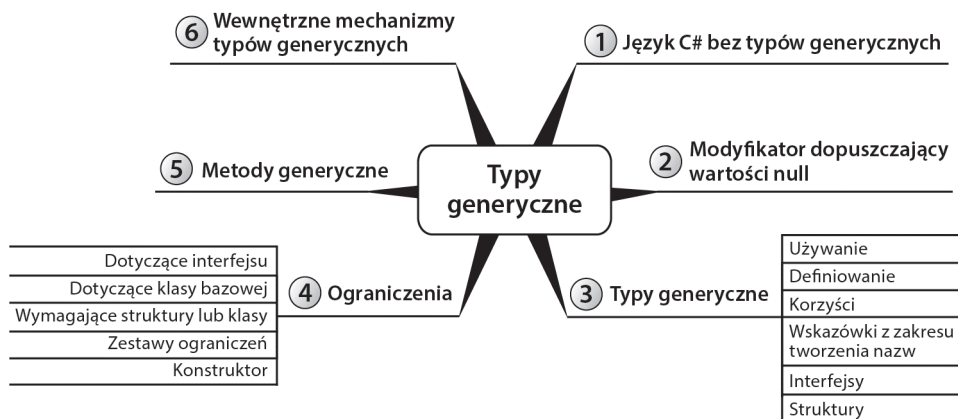
# 12

## Typy generyczne

Początek  
2.0

**G**DY 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 wzorec 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 12.1.

### Listing 12.1. Sygnatury metod klasy `System.Collections.Stack`

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

2.0

W programach stos często służy do umożliwiania wielokrotnego cofania operacji. Na przykład na listingu 12.2 kod używa klasy `System.Collections.Stack` do wycofywania operacji w programie symulującym działanie znikopisu.

### Listing 12.2. 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();
                    }
                }
            }
        }
    }
}
```

```

    }
    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.
}
}

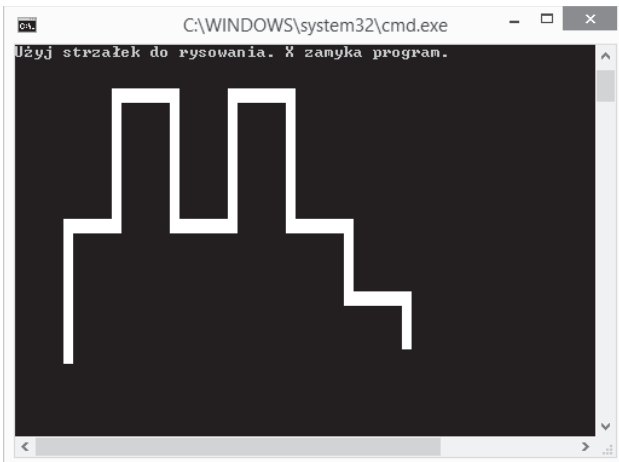
public struct Cell
{
    // W wersjach starszych niż C# 6.0 należy użyć pola tylko do odczytu.
    public int X { get; }
    public int Y { get; }
    public Cell(int x, int y)
    {
        X = x;
        Y = y;
    }
}

```

2.0

Efekt uruchomienia kodu z listingu 12.2 znajdziesz w danych wyjściowych 12.1

#### DANE WYJŚCIOWE 12.1.



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 12.1 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 12.3.

### Listing 12.3. 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.

## ■ ZAGADNIENIE DLA POCZĄTKUJĄCYCH

### Inny przykład — typy bezpośrednie z możliwą wartością null

W rozdziale 3. 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 12.4.

2.0

#### Listing 12.4. Deklarowanie wersji różnych typów bezpośrednich z dodaną obsługą wartości null

```

struct NullableInt
{
    /// <summary>
    /// Udogatnia 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>
    /// Udogatnia 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 12.4 pokazano możliwe implementacje typów `NullableInt` i `NullableGuid`. Jeśli w programie potrzebne są dodatkowe typy bezpośrednio z obsługą wartości `null`, trzeba utworzyć nową strukturę z właściwościami działającymi dla odpowiedniego typu. Każdą poprawkę 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 12.5.

### Listing 12.5. 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 bezpośrednio, 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 12.6 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 12.6 pokazano ten proces na przykładzie nowej generycznej klasy `Stack`.

### Listing 12.6. 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

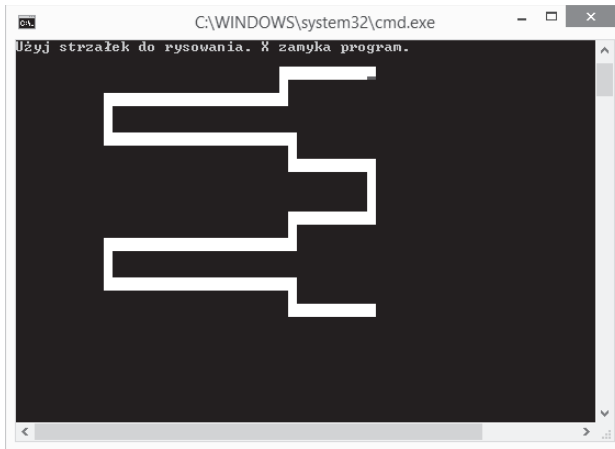
```

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

```

Wynik działania kodu z listingu 12.6 pokazano w danych wyjściowych 12.2.

#### DANE WYJŚCIOWE 12.2.



Na listingu 12.6 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 12.7 tworzona jest klasa `Stack<T>`, podobna do klasy `System.Collections.Generic.Stack<T>` użytej na listingu 12.6. **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 12.7 parametr określający typ jest używany w wewnętrznej tablicy `Items`, w parametrze metody `Push()` i w wartości zwracanej przez metodę `Pop()`.



**Listing 12.7.** 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 12.7 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 przekształcić 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.

## 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 `EntityCollection<Entity>` nazwa parametru określającego typ to `TEntity`.

2.0

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 12.8.

### Listing 12.8. Deklarowanie generycznego interfejsu

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

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 12.9. 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 12.9. Implementowanie generycznego interfejsu

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

2.0

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 przedstrzeni 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 12.10.

### Listing 12.10. Wielokrotne implementowanie interfejsu w jednej klasie

```
public interface IContainer<T>
{
    ICollection<T> Items { 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`.

2.0

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.

## 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 12.11 konstruktor jest zadeklarowany z sygnaturą `public Pair(T first, T second)`.

### Listing 12.11. 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 12.11 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 i automatycznie implementowane właściwości. 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 12.12, 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`.

2.0

### Listing 12.12. 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;
    // }

    // ...
}
```

Aby umożliwić rozwiązanie tego problemu, w języku C# udostępniono operator `default`, opisany wcześniej w rozdziale 9., 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 12.13.

### Listing 12.13. 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).

W C# 7.1 wprowadzono możliwość używania operatora `default` bez podawania parametru, jeśli możliwe jest wywnioskowanie typu danych. Na przykład przy inicjowaniu lub przypisywaniu wartości zmiennej można zastosować składnię `Pair<T> pair = default` zamiast `Pair<T> pair = default(Pair<T>)`. Ponadto jeśli metoda zwraca wartość typu `int`, można zastosować zapis `return default`, a kompilator wywnioskuje wywołanie `default(int)` na podstawie typu zwracanej wartości. Takie wnioskowanie jest możliwe także dla (opcjonalnych) parametrów domyślnych i argumentów wywołań metod.

## 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 12.14.

2.0

### Listing 12.14. 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; }
}
```

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 12.15.

### Listing 12.15. 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.

## ZAGADNIENIE DLA POCZĄTKUJĄCYCH

Początek  
7.0

### Krotki — typy o różnej arności

2.0

W rozdziale 3. opisano obsługę składni dla krotek z wersji C# 7.0. Wewnętrznie typ używany na potrzeby tej składni jest typem generycznym `System.ValueTuple`. Podobnie jak przy stosowaniu typu `Pair<...>` można wielokrotnie wykorzystać tę samą nazwę dzięki różnym arnościom (w każdej klasie używana jest inna liczba parametrów określających typ), co pokazano na listingu 12.16.

#### Listing 12.16. Przesłanianie definicji typów na podstawie arności

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

Rodzinę klas `ValueTuple<...>` zaprojektowano w tym samym celu co klasy `Pair<T>` i `Pair<TFirst, TSecond>`, przy czym klasy `ValueTuple<...>` obsługują do ośmiu argumentów określających typ. W ostatniej wersji klasy `ValueTuple` z listingu 12.16 parametr `TRest` można wykorzystać do zapisania następnego obiektu typu `ValueTuple`. Dlatego liczba elementów w krotce jest potencjalnie nieskończona. Jeśli zdefiniujesz krotki za pomocą wprowadzonej w C# 7.0 składni dla krotek, kompilator wygeneruje obiekty tego typu.

Inną ciekawą składową rodziny klas reprezentujących krotki jest niegeneryczna klasa `ValueTuple`. 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, gdy wywoływana jest metoda `Create()`. W C# 7.0 nie ma to znaczenia, ponieważ kod jest bardzo prosty: `var keyValuePair = ("555-55-5555", new Contact("Inigo Montoya"))` (przy założeniu, że nie są używane elementy nazwane). Jednak, co pokazano na listingu 12.17, stosowanie metody `Create()` w połączeniu z inferencją typów upraszcza pracę w C# 6.0.

**Listing 12.17.** Porównanie sposobów tworzenia instancji typu `System.ValueTuple`

```

#if !PRECSHARP7
(string, Contact) keyValuePair;
keyValuePair =
    "555-55-5555", new Contact("Inigo Montoya"));
#else // W wersjach starszych niż C# 7.0 należy stosować składnię System.ValueTupe<string,Contact>.
Tuple<string, Contact> keyValuePair;
keyValuePair =
    ValueTuple.Create(
        "555-55-5555", new Contact("Inigo Montoya"));
keyValuePair =
    new ValueTuple<string, Contact>(
        "555-55-5555", new Contact("Inigo Montoya"));
#endif // !PRECSHARP7

```

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

Warto zauważyć, że podobną klasę reprezentującą krotki, `System.Tuple`, dodano w C# 4.0. Stwierdzono jednak, że powszechne używanie składni dla krotek wprowadzonej w C# 7.0 i wynikająca z tego wszechobecność krotek uzasadniają utworzenie typu `System.ValueTuple`, ponieważ pozwala on zwiększyć wydajność.

Na podstawie tego, że w bibliotece platformy zadeklarowanych jest osiem różnych generycznych typów `ValueTuple`, 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. W zagadnieniu dla początkujących „Krotki — typy o różnej arności” znajdziesz przykład ilustrujący to zagadnienie.

## 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 12.18).

**Listing 12.18.** Zagnieżdżone typy generyczne

```

class Container<T, U>
{
    // Klasy zagnieżdżone dziedziczą parametry określające typ.

```

2.0

Początek  
4.0Koniec  
4.0Koniec  
7.0



```
// 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.

2.0

## 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 12.19.

### Listing 12.19. 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 12.20.

### Listing 12.20. 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 niejawnej 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 12.20, 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 12.21.

**Listing 12.21.** 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.

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, wymagając obecności konstruktora domyślnego lub określają, że konieczny jest typ referencyjny bądź bezpośredni.

2.0

## 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 12.22.

### Listing 12.22. 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 12.22 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 12.3.

### DANE WYJŚCIOWE 12.3.

```

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 12.23.

### Listing 12.23. Deklarowanie ograniczenia dotyczącego klasy

```

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

```

Na listingu 12.23 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.

2.0

## 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 bezpośrednich i referencyjnych. Zamiast określać klasę, po której `T` ma dziedziczyć, można podać słowo kluczowe `struct` lub `class`. Ilustruje to listing 12.24.

**Listing 12.24.** 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

2.0

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 12.25 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 12.25. Ustawianie wielu ograniczeń

---

```
public class EntityDictionary<TKey, TValue>
    : Dictionary<TKey, TValue>
    where TKey : IComparable<TKey>, IFormattable
    where TValue : EntityBase
{
    ...
}
```

---

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 12.26 metoda `MakeValue()` klasy `EntityDictionary<TKey, TValue>` musi tworzyć obiekt typu podanego jako parametr `TValue`.

**Listing 12.26.** 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;
    }

    // ...
}

```

2.0

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 udostępnia 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 12.27.

**Listing 12.27.** 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 12.27 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.

2.0

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 12.28).

**Listing 12.28.** 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 12.29 nie zadziała.

#### Listing 12.29. 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 12.30 jest niedozwolony.

**Listing 12.30.** 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
{
    ...
}
```

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 13.). 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 12.31, zgłosi błąd.

2.0

**Listing 12.31.** 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 12.26 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 12.32 jest nieprawidłowy.

**Listing 12.32.** 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 12.33).

2.0

**Listing 12.33.** 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);
}
...

```

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 12.33 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 12.34.

---

**Listing 12.34.** Deklarowanie typu encji używanych w typie `EntityDictionary<...>`


---

2.0

```
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 przekonałeś 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 12.35.

---

**Listing 12.35.** 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;
}
}
```

2.0

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

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 12.36.

### Listing 12.36. 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 12.36 pokazano w danych wyjściowych 12.4.

**DANE WYJŚCIOWE 12.4.**

```
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 12.36 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 12.37. Wynik działania tego kodu pokazano w danych wyjściowych 12.5.

2.0

**Listing 12.37.** 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 12.5.**

```
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 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 uwzględnia 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 12.38.

### Listing 12.38. Dodawanie ograniczeń w metodach generycznych

2.0

```
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 12.39).

### Listing 12.39. 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:

2.0

```
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 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**.

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 12.40.

2.0

---

#### Listing 12.40. 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 12.41 pokazano, co by się stało, gdyby język C# zapewniał nieograniczoną kowariancję.

---

#### Listing 12.41. 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");
...

```

---

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.

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.

## Umożliwianie kowariancji za pomocą modyfikatora `out` stosowanego do parametru określającego typ (od wersji C# 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 12.42).

### Listing 12.42. Rozwiązanie z potencjalnie możliwą kowariancją

2.0

```
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>
{
    // ...
}

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;
    }
}
```

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 12.42 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 12.43 pokazano, jak zmodyfikować deklarację interfejsu, by informowała, że należy umożliwić kowariancję.

#### Listing 12.43. Kowariancja dzięki użyciu modyfikatora `out` do parametru określającego typ

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

Dodanie modyfikatora `out` do parametru określającego typ w interfejsie `IReadOnlyPair<out T>` umożliwił 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 konwersje kowariantne z wykorzystaniem tego interfejsu. Po wprowadzeniu potrzebnej zmiany w kodzie z listingu 12.42 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 13.). 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.

## 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

4.0

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 załóżmy, 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*.

2.0

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 12.44.

#### Listing 12.44. 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 `PairInitializer<in T>` na listingu 12.45.

#### Listing 12.45. 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>());
```

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

2.0

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#` 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.

Koniec  
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.

Aby było to możliwe, typy generyczne wymagają obsługi w używanym środowisku uruchomieniowym. W C# typy generyczne są mechanizmem obsługiwanym 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.

2.0

### 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 12.46.

#### Listing 12.46. Deklaracja klasy Stack<T>

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

Po skompilowaniu tej klasy wygenerowany kod CIL jest sparametryzowany i wygląda tak jak na listingu 12.47.

#### Listing 12.47. 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 12.48).

---

**Listing 12.48.** 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
  ...
}
```

2.0

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.

Załóżmy, że w kodzie zadeklarowana jest klasa `Stack` z parametrem `int`, tak jak na listingu 12.49.

---

**Listing 12.49.** 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 12.50 zadeklarowane są dwa obiekty typu `Stack<int>`. Dla obu używany jest wygenerowany już przez środowisko uruchomieniowe kod klasy `Stack<int>`.

---

**Listing 12.50.** 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 argumentcie). 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.

Załóżmy, 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`. Załóżmy 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` opartego 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 liczbę 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.

## 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 15. 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 przejdiesz do tych zagadnień, warto przyjrzeć się wyrażeniom, które od wersji C# 3.0 znacznie usprawniły pracę z kolekcjami.



# Skorowidz

---

.NET

- Core, 814
- Native, 826
- Standard, 64, 815

## A

- adresy, 801
- agregacja, 298
- alias, 194, 659
- alokowanie danych, 806
- analiza
  - klasy, 44
  - typów zmiennoprzecinkowych, 68
- anulowanie
  - iteracji, 636
  - kooperatywne, 715
  - kwerendy PLINQ, 757
  - zadania, 715
- AOT, Ahead of Time, 826
- API, Application Programming Interface, 62
- apostrof, 75
- architektura .NET Native, 826
- argument, 181, 184
- argumenty nazwane, 214
- arność, 455
- asynchroniczna metoda
  - Main(), 769
- asynchroniczne
  - lambdy, 734
  - metody, 730

- operacje, 691
- wywoływanie operacji, 722
- zadania, 700
- żądania sieciowe, 726
- asynchroniczność
  - instrukcja async, 726
  - instrukcja await, 726
  - zadania, 726
- atrybut, 344, 652
  - FlagsAttribute, 371, 663
  - IndexerName, 625
  - MethodImplAttribute, 771
  - StructLayoutAttribute, 793
  - System.AttributeUsage
    - ↳ Attribute, 661
  - System.Conditional
    - ↳ Attribute, 664
  - System.NonSerializable, 668
  - System.ObsoleteAttribute, 666
  - System.SerializableAttribute, 672
- atrybuty
  - inicjowanie, 657
  - niestandardowe, 656
  - predefiniowane, 664
  - wyszukiwanie, 656
  - związane z serializacją, 667
- automatycznie implementowane
  - właściwości, 255

## B

- bajty, 150
- BCL, Base Class Library, 815, 827
- bezpieczeństwo, 819
  - ze względu na typ, 58
- biblioteka
  - BCL, 59, 815
  - Microsoft.Extension.CommandUtils, 400
  - Parallel Extensions, 691
  - TPL, 747
- biblioteki
  - klas, 394
  - wskazywanie, 395
- bity, 150
- blok
  - catch, 218
  - finally, 220
  - ogólny catch, 223, 427
  - try, 218
- blokada, 690, 770
- bloki kodu, 138, 140
- błąd, 794
  - kompilacji, 453
  - nieskończonej rekurencji, 209
- błędy związane z tablicami, 116

## C

centralizowanie inicjowania, 266  
 CIL, Common Intermediate Language, 57, 811, 827  
 CLI, Common Language Infrastructure, 58, 811, 827  
   definiowanie standardu, 811  
   implementacje, 813  
 CLR, 827  
 CLS, Common Language Specification, 58, 825, 827  
 COM, 787  
 CTS, Common Type System, 58, 824, 827

## D

dane  
   wejściowe, 51  
   wyjściowe, 51, 60, 307  
 definiowanie  
   indeksera, 623  
   interfejsu, 324  
   iteratora, 627  
   klasy, 43, 232, 444  
     abstrakcyjnej, 311  
     częściowej, 284  
     generycznej, 448  
     publikującej zdarzenia, 517  
     zagnieżdżonej, 281  
   metod subskrybujących, 516  
   niestandardowych  
     konwersji, 294  
     wyjątków, 433  
   operatora indeksowania, 625  
   operatorów rzutowania, 294  
   przestrzeni nazw, 402  
   składowych  
     abstrakcyjnych, 311  
   symboli preprocesora, 172  
   typu, 43  
   typu wyliczeniowego, 365  
   właściwości, 248  
   wylczenia, 364

deklarowanie  
   funkcji zewnętrznych, 790  
   generycznego interfejsu, 450  
   generycznego typu delegata, 532  
   getterów i setterów, 246  
   klasy, 43, 232  
   konstruktora, 260  
   metod, 185  
   metody Main, 44  
   parametrów formalnych, 187  
   pola instancji, 235  
   pól statycznych, 269  
   stałej, 133  
   struktury, 351  
   tablicy, 106  
   typu delegata, 487, 489  
   właściwości, 247  
   wskaźników, 803  
   zdarzeń, 530  
   zmiennej, 47, 48  
 dekonstruktory, 267  
 dekrementacja, 129, 132  
   w pętli, 129  
 delegaty, 149, 466, 483, 507  
   a drzewa wyrażeń, 510, 509  
   deklarowanie typu, 487, 489  
   operatory, 521  
   równość strukturalna, 501  
   synchroniczne, 699  
   tworzenie instancji, 490, 491  
   typ danych, 486  
   typu generycznego, 532  
   typu multicast, 515, 523  
   typy, 488  
   wewnętrzne mechanizmy, 491  
   wywoływanie, 518  
 dereferencja wskaźników, 807  
 deserializacja, 671  
 deterministyczna finalizacja, 413  
 dezassembler, 59  
 diagram  
   interfejsów, 340  
   klas, 544  
   sekwencji wywołań delegatów, 526

  sekwencyjny, 631  
   z sekwencją wywołań delegatów, 524  
 długość łańcuchów znaków, 82  
 dodawanie  
   atributów, 653  
   łańcuchów znaków, 123  
   referencji, 397  
 dokumentacja w formacie XML, 407  
 dołączanie kodu, 172  
 dopasowanie do wzorca, 317, 318  
 dosłowny literał tekstowy, 77  
 dostęp  
   do instancji klasy, 240  
   do metadanych, 642  
   do pola statycznego, 270  
   do pól instancji, 236  
   do składowych typu docelowego, 808  
   do tablicy, 105  
 dostrajanie wydajności, 749  
 Dotnet CLI, 33, 39  
 drzewo wyrażeń, 507, 509  
   badanie, 511  
   lambda, 509, 510  
 duck typing, 547  
 dynamiczne wywoływanie składowej, 644, 652  
 dyrektywa  
   #define, 171  
   #elif, 171, 172  
   #else, 171, 172  
   #endif, 172  
   #endregion, 175  
   #error, 171, 173  
   #if, 171, 174  
   #line, 171  
   #pragma, 171, 174  
   #region, 171  
   #undef, 171, 172  
   #warning, 171, 173  
   #region, 175  
   using, 80, 190  
     static, 80, 193  
     zagnieżdżanie, 192  
 dyrektywy preprocesora, 170

dziedziczenie, 230, 289, 356  
interfejsów, 335  
jawne, 316  
klas wyjątków, 221  
ograniczenia, 463  
po jednym typie, 297  
po wielu interfejsach, 337

## F

FCL, Framework Class Library, 816, 827  
FIFO, first in, first out, 622  
filtrowanie, 550, 593  
finalizacja, 415  
finalizatory, 264, 411, 452  
formatowanie  
bez wcięcia, 47  
dwustronne, 72  
łańcuchów znaków, 81  
złożone, 53  
funkcje  
anonimowe, 494, 504  
lokalne, 734  
zewnętrzne, 790, 797

## G

generowanie  
błędów i ostrzeżeń, 173  
pliku z dokumentacją, 407  
typów anonimowych, 581  
generyczne  
interfejsy, 450  
struktury, 450  
gettery, 246, 258  
grafy obiektów, 509  
grupowanie, 567  
instrukcji, 180  
wyników zwracanych, 596

## H

hermetyzacja, 230, 234, 244  
danych, 278  
procesu publikacji, 530  
subskrypcji, 529  
typów, 400

hierarchia  
generycznych interfejsów, 607  
klas, 231

## I

IDE, Intergrated Development Environment, 35  
identyfikatory, 41  
iloczyn kartezjański, 566  
implementacja  
generycznego interfejsu, 451  
interfejsu, 325, 329  
IComparer<T>, 611  
IEqualityComparer<T>, 619  
System.Runtime.Serializaton.ISerializable, 669  
metody Equals(), 383  
niestandardowej metody asynchronicznej, 736, 738  
opóźnionego wykonywania, 593  
relacji jeden do wielu, 568  
typów subskrybujących, 516  
wielodziedziczenia, 339  
wzorca publikuj-subskrybuj, 516  
zdarzeń, 537  
złączeń zewnętrznych, 570  
zmiennych zewnętrznych, 504

implementacje standardu CLI, 813  
indeks, 623  
inferencja typów, 469  
informacje o tablicach, 105  
inicjowanie  
kolekcji, 263, 540, 582  
obiektów, 262  
słownika, 541  
struktur, 352  
inkrementacja, 129, 132  
instancja  
delegata, 490  
klasy, 45  
tablicy, 107

instrukcja, 45  
async, 726  
await, 726  
break, 135, 165  
continue, 134, 167  
do while, 134  
dynamic, 673  
fixed, 805  
for, 134  
foreach, 134  
dla interfejsu IEnumerable<T>, 543  
dla tablic, 542  
goto, 135, 169  
if, 134, 135  
bez bloku kodu, 138  
if-else, 318  
Join(), 564  
lock, 359  
return, 188  
switch, 135, 162, 318  
throw, 224  
using, 413, 415  
while, 134  
yield, 639  
break, 636  
return, 631, 633, 638  
instrukcje, 45  
bez średników, 46  
opakowywania, 357  
skoku, 165  
wierszowe, 45  
interfejs, 323, 344, 356  
API Win32, 794  
ICollection<T>, 608  
IComparer<T>, 611  
IDictionary<TKey, TValue>, 606  
IDisposable, 413, 415  
IEnumerable<T>, 542, 543  
IEqualityComparer<T>, 619  
IList<T>, 606  
IQueryable<T>, 575  
System.Runtime.Serializaton.ISerializable, 669  
interfejsy  
a atrybuty, 344  
a klasy, 343

## interfejsy

- API, 62
  - bazowe, 336
  - definiowanie, 324
  - diagramy, 340
  - dziedziczenie, 335
  - generyczne, 450
  - implementowanie, 325, 329, 333
    - wielodziedziczenia, 339
    - wielokrotne, 451
  - jawnie podawane, 331
  - kolekcji, 539
  - metody rozszerzające, 337
  - ograniczenia, 459
  - pochodne, 335, 342
  - polimorfizm, 325
- interpolacja łańcuchów znaków, 52, 78
- iteracja, 545
- iteratory, 627, 630
  - definiowanie, 627
  - działanie, 636
  - rekurencyjne, 635
  - składnia, 628
  - w jednej klasie, 638
  - zwracanie wartości, 629

## J

## język

- C#, 31
- CIL, 57, 59
- Common Intermediate Language, 823
- JIT, just-in-time, 58, 816

## K

## kategorie typów, 347

## klasa, 229, 233

- BinaryTree<string>, 634
- CancellationToken, 717
- CancellationTokenSource, 717
- CommandLineInfo, 647
- Dictionary<TKey, TValue>, 614

- LinkedList<T>, 623
- List<T>, 608, 609
- MemberInfo, 648
- Monitor, 765
- object, 375, 643
- Queryable, 575
- Queue<T>, 622
- Stack, 444
- Stack<T>, 621, 649
- System.Delegate, 492
- System.Object, 315
- System.Threading, 691
- System.Threading.
  - ↳ Interlocked, 772
- System.Threading.Mutex, 776
- System.Threading.Thread, 691
- System.Type, 642
- ThreadPool, 697
- WaitHandle, 778

## klasy

- abstrakcyjne, 310, 344
- bazowe, 300
- bazowe podatne na błędy, 304
- częściowe, 283
- definiowanie, 232
- deklarowanie, 232
- deklarowanie zmiennych, 233
- dotyczące, 460
- finalizatory, 264
- generyczne, 447
  - deklarowanie konstruktora, 452
- kolekcji, 542, 608
- kolekcji przetwarzanych równoległe, 782
- konstruktory, 260
- metody instancji, 237
- modyfikatory dostępu, 257
- pochodne, 290
- poła instancji, 235
- przesłanianie składowych, 300
- publikujące zdarzenia, 517
- składowe, 234

- składowe statyczne, 269
- statyczne, 275
- tworzenie instancji, 232, 233
- właściwości, 246
- wyjątków, 221
- zagnieżdżone, 281
- zamknięte, 299

## klauzula

- from, 600
- group, 598, 599
- into, 599
- kontynuacji kwerendy, 599
- let, 595

## kod

- dyrektywy preprocesora, 172
- edytowanie, 32
- kompilowanie, 32, 38
- maszynowy, 816
- natywny, 58
- niezabezpieczony, 789, 801, 809
- tworzenie, 32
- uruchamianie, 32
- wykonywanie, 38

## kolejka, 622

## kolekcja, 263, 539

- Dictionary<TKey, TValue>, 614
- LinkedList<T>, 623
- List<T>, 608, 612
- Queue<T>, 622
- SortedDictionary<TKey, TValue>, 620
- SortedList<T>, 620
- Stack<T>, 621

## kolekcje

- interfejsy, 606
- niestandardowe, 605
- posortowane, 620
- przeszukiwanie, 612
- przetwarzane równoległe, 782
- puste, 626
- wyszukiwanie wielu elementów, 614

komentarze, 55  
 jednowierszowe, 56  
 XML-owe, 56, 405  
 z ogranicznikami, 56  
 z konstrukcjami  
 programistycznymi, 406  
 kompilacja, 38, 816  
 AOT, 826  
 kompilator JIT, 58, 771, 816  
 konsola, 51  
 konstruktory, 260, 309  
 definiowanie, 260  
 deklarowanie, 260  
 domyślne, 262, 466  
 inicjowanie atrybutu, 657  
 łączenie w łańcuch, 265  
 ograniczenia, 462  
 przeciążone, 264  
 statyczne, 273  
 typu generycznego, 452  
 wywoływanie, 261  
 kontekst synchronizacji, 740  
 kontrawariancja, 472, 475  
 kontrola  
 konwersji, 437  
 typów, 819  
 konwencje programistyczne,  
 532  
 konwersje, 320  
 drzewa wyrażeń, 508  
 między wyliczeniami  
 a łańcuchami znaków,  
 367  
 niejawne, 89, 293  
 typów bez rzutowania, 90  
 z łańcucha znaków na  
 wyliczenie, 368  
 kontrolowane, 87  
 niekontrolowane, 87  
 typów danych, 86  
 z kontrolą, 436  
 kowariancja, 472  
 niezabezpieczona, 478  
 krotka Value Tuple, 103  
 krotki, 98, 189, 455  
 w wyrażeniach, 589  
 kwery, 539  
 operatory, 572

PLINQ, 756  
 równoległe wykonywanie,  
 553, 754  
 standardowe operatory, 547

## L

lambda, 500  
 asynchroniczne, 734  
 w postaci instrukcji, 495  
 w postaci wyrażeń, 497  
 latencja, 683  
 leniwe inicjowanie, 418  
 za pomocą typów  
 generycznych, 419  
 liczby zmiennoprzecinkowe,  
 126  
 LIFO, last in, first out, 621  
 LINQ, Language Integrated  
 Query, 15, 539, 585  
 grupowanie, 567  
 opóźnione wykonanie, 555  
 równoległe wykonywanie  
 kwery, 553, 754  
 sortowanie, 559  
 typy anonimowe, 576  
 złączania, 564  
 listy, 608  
 powiązane, 623  
 literał, 76  
 typu decimal, 70  
 typu double, 69  
 literały liczbowe, 68  
 binarne, 72  
 szesnastkowe, 71

## Ł

łańcuch znaków, 50, 76  
 długość, 82  
 dodawanie, 123  
 formatowanie, 81  
 interpolacja, 78  
 jako tablica, 115  
 metody, 79, 80  
 niezmiennosc, 83  
 złączanie, 77

łączenie subskrybentów  
 z nadawcą, 517  
 łączność, 122

## M

manifesty, 821  
 maska, 153  
 mechanizm  
 odzyskiwania pamięci, 234,  
 411  
 P/Invoke, 790  
 menu Projekt, 398  
 metadane, 642, 825  
 metoda, 44  
 Assert(), 127  
 BubbleSort(), 484, 486  
 CommandLineHandler.Try  
 Parse(), 659  
 ContinueWith(), 708  
 Count(), 554  
 Equals(), 379, 383  
 FindAll(), 614  
 GetGenericArguments(),  
 650  
 GetHashCode(), 377, 378  
 GetType(), 643  
 GroupBy(), 567  
 GroupJoin(), 568, 570  
 int.TryParse(), 227  
 Main, 44  
 Main(), 195  
 OrderBy(), 559  
 Pop(), 621  
 Push(), 621  
 Select(), 551  
 SelectMany(), 571  
 System.Console.Read(), 52  
 System.Console.ReadLine()  
 , 51  
 Task.ContinueWith(), 741  
 Task.Delay(), 786  
 Task.Factory.StartNew(),  
 718, 719  
 ThenBy(), 559  
 ToString(), 376  
 TryParse(), 91, 227  
 Where(), 550

## metody

- anonymowe, 494, 499
- bezparametrowe, 500
- wewnętrzne mechanizmy, 502
- argumenty, 184
- async void, 732
- asynchroniczne
  - niestandardowe, 736, 738
  - zwracanie wartości, 730, 732
- częściowe, 285
- deklarowanie, 185
- dla tablic, 113
- dla typu string, 79
- generyczne, 468
  - dodawanie ograniczeń, 471
  - inferencja typów, 469
  - rzutowanie, 472
- instancji, 237
- instancyjne tablice, 114
- klasy Interlocked, 773
- krotki, 189
- nazwa, 184
- parametry, 184
- parametry opcjonalne, 212
- przeciążone, 209
- rozszerzające, 277, 297, 337
- statyczne, 272
- typ wartości zwracanej, 188
- typu string, 80
- wartości zwracane, 184, 528
- wirtualne, 301
- wywołania, 180, 181, 185, 215
- z ciałem w postaci
  - wrażenia, 190
- z jawnie podawanym interfejsem, 331
- zaawansowane parametry, 197
- microsoft .NET Framework, 814
- miejsce wywołania, 196
- modele zarządzania pamięcią, 690
- moduły, 821

## modyfikator

- async, 742
- await, 739, 742
- const, 278
- dostępu, 244, 402
  - internal, 400
  - private, 245, 294
  - protected, 295
  - protected internal, 401
  - public, 400
  - w getterach i setterach, 257
- formatowania R, 73
- formatu szesnastkowego, 72
- in, 475
- new, 304, 305
- out, 200, 474
- readonly, 279
- sealed, 307
- virtual, 300
- modyfikowanie
  - implementacji zdarzeń, 537
  - kolekcji, 547
- MTA, Multi-threaded Apartment, 787

## N

- nadtyp, 289
- nakładki, 799
- narzędzie
  - Dotnet CLI, 33
  - ILDASM, 59
- nawiasy, 122
- nazwa
  - indeksera, 625
  - klasy, 38
  - metody, 184
  - parametrów określających typ, 450
  - pliku, 38
  - typu, 182
- nieskończoność dodatnia, 128
- notacja
  - pascalowa, 42
  - szesnastkowa, 71
  - wykładnicza, 71
- notacja wiersz, 82

## O

- obiekty, 233
  - dynamiczne, 672, 679
  - inicjatory, 262
- obsługa
  - asynchroniczności, 720
  - błędów, 216, 794
  - serializacji, 435
  - wartości null, 445
  - wyjątków, 216, 219, 421, 429
  - wyjątków w pętłach równoległych, 749
- odpytywanie cykliczne, 701
- odstęp, 46
- odzyskiwanie pamięci, 409, 415, 818
  - w platformie .NET, 819
- ogólne bloki catch, 428
- ograniczenia
  - dotyczące
    - delegatów, 466
    - dziedziczenia, 463
    - interfejsu, 459
    - klasy, 460
    - konstruktora, 462, 466
    - wyliczeń, 466
    - wymagające struktury, 461
    - wymogi, 465
- ograniczniki instrukcji, 45
- okno dialogowe Nowy projekt, 36
- określanie wartości, 122
- opakowywanie, 356, 360
  - interfejsu API, 799
- opcja nowarn:<lista ostrzeżeń>, 174
- opcje pętli równoległych, 752
- operacja odrzucania, 102
- operacje
  - arytmetyczne, 124
  - asynchroniczne, 691
  - atomowe, 688
- operand, 120
- operator, 119, 120
  - ?, 147
  - ??, 147
  - AND, 144



- as, 320
  - await, 744
  - default, 355
  - dodawania dla łańcuchów znaków, 123
  - dopełnienia, 155
  - indeksowania, 625
  - inkrementacji, 129
    - postinkrementacji, 130
    - preinkrementacji, 131
  - is, 316, 317
  - minus, 120
  - nameof, 253, 651
  - negacji, 145
  - new, 260, 261, 354
  - OR, 144
  - plus, 124
  - rzutowania, 86, 392
  - typeof, 644, 649
  - warunkowy, 145
  - XOR, 144
  - operatory
    - bitowe, 150, 152
    - dwuargumentowe, 121, 388, 390
    - jednoargumentowe, 120, 391
    - konwersji, 392, 393
    - kwerend, 539, 547, 572
    - logiczne, 143
    - łączność, 122
    - określanie wartości, 122
    - porównania, 387
    - priorytety, 122
    - przesunięcia, 151
    - przypisania, 128
      - bitowe złożone, 154
    - relacyjne i równości, 143
    - rzutowania, 294
    - warunkowe, 391
- P**
- P/Invoke, 790
  - pakiet
    - NuGet, 396, 399
    - SDK, 32
  - pamięć, 409
    - lokalna wątków, 783
  - parametr, 181, 184
    - określający typ (T), 448
    - out, 258
    - ref, 258
    - w postaci delegata, 486
  - parametry
    - formalne, 187
    - generyczne
      - określanie, 649
    - nazwane, 662
    - opcjonalne, 212
    - przekazywane
      - przez referencję, 199, 202, 528
      - przez wartość, 197
    - ref, 792
    - wyściowe, 200
  - pętla
    - do/while, 156
    - for, 157, 747
    - foreach, 160, 547, 634, 748
      - bez interfejsu
        - IEnumerable, 547
      - dla kolekcji, 546
      - Parallel.For(), 753
      - while, 155, 156
    - pętle równoległe, 747, 748
      - anulowanie wykonywania, 751
      - obsługa wyjątków, 749
      - opcje, 752
  - pierwszy program, 32
  - platforma
    - .NET, 32, 62
    - CLI, 57
  - plik projektu, 37
  - pliki
    - wczytywanie, 241
    - z dokumentacją, 407
    - zapisywanie, 241
  - PLINQ, Parallel LINQ, 553, 685
  - pobieranie
    - danych, 578
    - niewymagalnych elementów, 601
    - niestandardowych atrybutów, 658
  - podkradanie pracy, 749
  - podtyp, 289
  - podzespoły, 821
  - pola, 250, 252
    - instancji, 235, 270
    - jako zmienne, 772
    - statyczne, 269
    - wirtualne, 255
  - polimorfizm, 210, 232, 313
    - oparty na interfejsach, 325
  - porządkowanie
    - całkowite, 612
    - zasobów, 411, 545
  - powiadomienia o zdarzeniach, 773
  - predykat, 489, 593
  - priorytety, 122
  - proces, 685
  - procesor, 685
  - programowanie
    - dynamiczne, 672, 678
    - obiektowe, 230
    - równoległe, 687
    - sekwencyjne, 17
    - ustrukturyzowane, 17
  - programy szeregujące zadania, 740
  - projekcja, 551, 588
    - na krotkę, 552
  - projekt
    - tworzenie, 37
  - projektowanie synchronizacji, 775
  - przechwytywanie
    - wyjątków, 217, 424
    - zmiennych, 505
  - przeciążanie
    - funkcji, 386
    - konstruktorów, 264
    - metod, 209
    - operatorów, 387
  - przekazywanie
    - przez referencję, 199, 202, 528
    - przez wartość, 197
    - wyjątków, 418
  - przełączanie kontekstu, 687
  - przenośność, 59, 820

przepełnienie typu  
 całkowitoliczbowego, 437  
 przepływ sterowania, 119, 133  
 przesłanianie  
 definicji typów, 455  
 metody, 376–379, 384  
 operatora równości, 380  
 składowych, 300, 375  
 właściwości, 300  
 przestrzenie  
 deklaracji, 140  
 nazw, 182, 190  
 definiowanie, 402  
 zagnieżdżanie, 403  
 przypisanie, 49, 390  
 przywracanie pamięci, 58  
 pula wątków, 686, 696

## R

RCW, runtime callable  
 wrapper, 788  
 refaktoryzacja, 187  
 klasy, 290  
 referencje, 199, 395, 409, 528  
 do pakietów, 397  
 słabe, 410  
 refleksja, 642  
 do typów generycznych,  
 649, 650  
 instrukcja dynamic, 673  
 rekurencja, 207  
 nieskończona, 209  
 relacja  
 „potrafi”, 323  
 jeden do wielu, 562, 568  
 LUB, 465  
 wiele do wielu, 561  
 reprezentacja  
 liczby, 153  
 typów generycznych, 479  
 równoległe wykonywanie  
 iteracji, 746  
 równość strukturalna, 501  
 rzutowanie, casting, 86, 292  
 jawne, 86  
 na typ bazowy, 292  
 niejawne, 292

w łańcuchu dziedziczenia,  
 293  
 w metodach generycznych,  
 472

## S

SDK, Software Development  
 Kit, 32  
 sekwencja  
 ucieczki, 74, 75  
 wywołań delegatów, 524,  
 526  
 semaforey, 781  
 separator cyfr, 70  
 serializacja, 435, 667  
 niestandardowa, 669  
 settery, 246, 258  
 składnia, 40  
 dla krotek, 99  
 składowa base, 308  
 składowe  
 abstrakcyjne, 310  
 bez jawnie podawanego  
 interfejsu, 332  
 prywatne, 244  
 statyczne, 269  
 z jawnie podawanym  
 interfejsem, 331  
 słabe referencje, 410  
 słowa kluczowe, 40–43, 597,  
 598  
 kontekstowe, 597, 598, 638  
 zwracane po kolei, 629  
 słownik, 541, 606, 614  
 sprawdzanie równości, 618  
 słowo kluczowe  
 async, 726  
 await, 726  
 class, 232, 461  
 default, 105  
 lock, 767, 770  
 new, 108  
 null, 85  
 struct, 461  
 this, 238, 240, 265, 770  
 static, 269  
 unchecked, 89  
 void, 85  
 volatile, 772  
 sortowanie, 559, 594  
 kolekcji, 610  
 specyfikacja  
 .NET Standard, 815  
 CLI, 58  
 sprawdzanie  
 poprawności, 252  
 równości, 619  
 typu, 316  
 wartości null, 521  
 stała lokalizacja danych, 805  
 stałe, 133  
 lokalne, 132  
 publiczne, 279  
 stan, 630  
 standard  
 CLI, 811  
 Unicode, 74  
 standardowe operatory  
 kwerend, 539, 547  
 statyczna kompilacja, 678  
 sterta, 94  
 stos, 621, 806  
 wywołań, 196  
 stosowanie wartości null, 96  
 struktury, 351  
 deklarowanie, 351  
 inicjowanie, 352  
 struktury generyczne, 450  
 subskrybent, 527  
 synchroniczne wywoływanie  
 operacji, 720, 743  
 synchronizacja wątków, 761  
 klasa  
 Interlocked, 772  
 Monitor, 765  
 Mutex, 776  
 projektowanie, 775  
 słowo kluczowe lock, 767  
 system  
 typów CTS, 58  
 VES, 58  
 szeregowanie zadań, 699, 740

## Ś

środowisko  
 IDE, 35  
 uruchomieniowe, 58, 812,  
 818  
 zarządzane, 58

## T

tablica parametrów, 206  
 tablice, 104  
 akcesor, 110  
 definiowanie wielkości, 108  
 deklaracja, 105, 106, 111  
 dostęp, 105  
 dwuwymiarowe, 106, 109  
 łańcuchy znaków, 115  
 metody, 113  
 metody instancyjne, 114  
 niezabezpieczona  
 kowariancja, 478  
 parametrów, 204  
 pobieranie długości, 111  
 przypisywanie, 105  
 tablic, 110, 111  
 trójwymiarowe, 109  
 tworzenie instancji, 107  
 typowe błędy, 116  
 wartość Length, 112  
 tożsamość obiektów, 379  
 TPL, Task Parallel Library, 685,  
 759  
 tryb tylko do odczytu, 202  
 tworzenie  
 aliasów, 194  
 instancji  
 delegata, 490, 491  
 klasy, 233  
 klas pochodnych, 290, 291  
 niestandardowych kolekcji,  
 605  
 obiektów  
 dynamicznych, 679  
 typów generycznych, 481  
 usuwanych, 418  
 projektu, 37

## typ

AggregateException, 749  
 AutoResetEvent, 781  
 bazowy, 289  
 CancellationToken, 717  
 CountdownEvent, 781  
 danych w postaci delegata,  
 486  
 decimal, 68  
 dynamic, 674, 676  
 float, 128  
 logiczny, 73  
 ManualResetEvent, 781  
 Pair<T>, 635  
 pochodny, 289  
 SafeHandle, 796  
 Semaphore, 781  
 SemaphoreSlim, 781  
 string, 79  
 System.Dynamic.Dynamic  
 Object, 681  
 System.Text.StringBuilder,  
 84  
 System.Threading.WaitHa  
 ndle, 778  
 System.ValueTuple<...>,  
 102  
 Task, 720  
 TaskContinuationOptions,  
 707  
 ThreadLocal<T>, 783  
 Type, 656  
 ValueTask<T>, 730  
 wyliczeniowy, 365  
 znakowy, 74  
 zwracanej wartości, 188

## typy

bezpośrednie, 198, 347,  
 354, 356, 359, 445  
 danych, 48, 65, 791  
 anonimowe, 97, 579, 582  
 bezpośrednie, 93  
 całkowitoliczbowe, 66  
 kategorie, 93  
 konwersje, 86  
 liczbowe, 65  
 niejawnie określone, 96

referencyjne, 94  
 zmiennoprzecinkowe, 67,  
 125  
 delegatów, 488  
 dobrze uformowane, 375  
 generyczne, 441  
 a delegaty, 534  
 dowolna liczba  
 parametrów, 454  
 ograniczenia, 457  
 oparte na typach  
 referencyjnych, 481  
 wewnętrzne mechanizmy,  
 479  
 zagnieżdżone, 456  
 zalety, 449  
 niezarządzane, 803  
 o różnej arności, 455  
 parametryzowane, 446  
 referencyjne, 198, 349  
 w technologii LINQ, 576  
 wyjątków, 222, 421

## U

ujemna nieskończoność, 128  
 układ sekwencyjny, 793  
 UML, Unified Modeling  
 Language, 340  
 diagram klas, 544  
 diagramy interfejsów, 340  
 Unicode, 74  
 unikanie  
 blokad, 770, 776  
 opakowywania, 362  
 wieloznaczności, 239  
 wypakowywania, 363  
 ustalanie typów, 649  
 usunięcie odstępów, 47

## V

VES, Virtual Execution System,  
 58, 827  
 Visual Studio 2017, 35, 39

## W

- wariancja, 501
- wartości wyliczeniowe, 706
- wartość
  - Length, 112
  - NaN, 127
  - null, 85, 96, 445, 519
- wątek, 521, 686, 688
  - główny, 763
  - z dekrementacją, 763
- wątki
  - pamięć lokalna, 783
  - powiadamy o zdarzeniach, 774
  - problemy, 688
  - pule, 696
  - synchronizowanie, 761
  - usuwanie, 695
  - usypianie, 694
  - zarządzanie, 693
- wczytywanie plików, 241
- wersje
  - języka C#, 63, 670
  - platformy .NET, 62
- wiązanie
  - dynamiczne, 677
  - komentarzy XML-owych, 406
- wielodziedziczenie, 298, 339
- wielowątkowość, 683
  - równoległa, 685
- właściwości, 82, 246, 247, 250
  - automatycznie implementowane, 249, 255
  - definiowanie, 248
  - deklarowanie, 247
  - dodawanie atrybutów, 653
  - jako pola wirtualne, 255
  - leniwe inicjowanie, 418
  - przesłanie, 300
  - stacyjne, 274
  - tylko do odczytu, 254, 280
  - tylko do zapisu, 254
- właściwość
  - AsyncState, 703
  - IsCompleted, 703
  - SubItems, 635
- wskazywanie
  - biblioteki, 395
  - projektu, 396
- wskazniki, 792, 801
  - deklarowanie, 803
  - dereferencja, 807
  - do funkcji, 800
  - przypisywanie wartości, 804
- wspinaczka, 749
- współdziałanie języków, 58
- współdzielony stan, 545
- wstawianie nowego wiersza, 76
- wydajność, 687, 749, 820
- wyjątek, 217
  - AggregateException, 709
  - NullReferenceException, 519
- wyjątki, 216, 421
  - nieobsłużone w wątku, 712
  - niestandardowe, 433
  - obsługa, 421
  - ogólny blok catch, 427
  - ponowne zgłaszanie, 435
  - przechwytywanie, 424
  - typy, 222, 421
  - wskazówki, 429
  - z konstruktorów, 418
  - z obsługą serializacji, 435
- wykluczanie kodu, 172
- wykonywanie kodu, 38
  - niezabezpieczonego kodu, 809
- wyliczenia, 363, 466
  - jako flagi, 369
- wypakowywanie, 357
- wrażenia
  - lambda, 419, 483, 494, 498
  - jako dane, 507
  - zmienne zewnętrzne, 503
- logiczne, 142
- z kwerendami, 585
  - filtrowanie, 593
  - krotki, 589
  - opóźnione wykonywanie, 590, 591
- pobieranie
  - niepowtarzalnych elementów, 601
  - projekcja, 588
  - sortowanie, 594
  - wywołania metod, 601
- wrażenie o stałej wartości, 132
- wyszukiwanie atrybutów, 656
- wyścig, 689
- wyświetlanie
  - apostrofu, 75
  - danych, 52
- wywołania
  - asynchroniczne, 722
  - dynamiczne, 644, 652
  - sekwencyjne, 523
  - synchroniczne, 720 743
- wywoływanie
  - asynchronicznego zadania, 700
  - delegata, 518, 519, 521
  - funkcji zewnętrznych, 797
  - instrukcji using, 415
  - konstruktora, 261
  - metod, 180, 185
  - operatorów dwuargumentowych, 389
  - pakietu NuGet, 399
  - wyrażeń lambda, 557
- wzorzec
  - obsługi asynchroniczności, 720
  - publikuj-subskrybuj, 516

## X

- Xamarin, 815
- XML, Extensible Markup Language, 57

## Z

- zabezpieczenie dostępu do kodu, 58
- zadania, 686
  - anulowanie, 715
  - asynchroniczne, 698, 699
  - długotrwałe, 719

- kontynuacja, 703
- obsługa wyjątków, 709
- powiadomienia, 708
- wzorzec obsługi
  - asynchroniczności, 720
  - zwalnianie zasobów, 720
- zagnieżdżone
  - instrukcje if, 136
  - typy generyczne, 456
- zakleszczenia, 690, 775
- zapisywanie
  - na sztywno, 69
  - plików, 241
- zarządzanie
  - przepływem sterowania, 133
  - wątkami, 693
  - wersjami, 341, 670
- zasięgi, 140, 184
- zasoby, 411, 545
- zdarzenia, 515, 528
  - deklarowanie, 530
  - modyfikowanie implementacji, 537
  - powiadomienia, 773
  - resetujące, 778
- wewnętrzne mechanizmy, 535
- zgłaszanie powiadomień, 533
- zdarzenie
  - Cooler, 516
  - Heater, 516
  - ManualResetEvent, 778
  - ManualResetEventSlim, 778
- zegary, 786
- zestawy ograniczeń, 462
- zgłaszanie
  - błędów, 224
  - powiadomień, 773
  - powiadomień o zdarzeniu, 533
  - wyjątku, 225, 422
- zgodność typów
  - wyliczeniowych, 367
- zliczanie elementów, 554
- złączanie, 561
  - łańcuchów znaków, 77
- złączenie
  - pełne zewnętrzne, 561
  - wewnętrzne, 561, 566
- zewnętrzne lewostronne, 561
- zewnętrzne prawostronne, 561
- zmienna
  - liczba parametrów, 205
  - zakresowa, 587
- zmiennne, 47
  - deklarowanie, 48
  - lokalne, 48
  - modyfikowanie wartości, 49
  - przypisywanie wartości, 49
  - lokalne bez synchronizacji dostępu, 764
  - typu anonimowego, 578
  - var, 577
  - zewnętrzne, 503
- znaki Unicode, 74
- zwracanie wartości null, 626
  - przez iterator, 629

## Ż

żądania asynchroniczne, 726



# PROGRAM PARTNERSKI

— GRUPY 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  
**Helion**

MARK MICHAELIS jest autorytetem w dziedzinie tworzenia zaawansowanego oprogramowania. Zabiera głos na prestiżowych konferencjach dla programistów. Obecnie prowadzi kolumnę *Essential.NET* w „MSDN Magazine”. Od 1996 roku posiada tytuł Microsoft MVP. W 2007 roku został dyrektorem regionalnym Microsoftu, jest też członkiem kilku zespołów oceniających projekty oprogramowania tej firmy (między innymi języka C# i technologii VSTS).

#### W tej książce między innymi:

- przewodnik po C# oraz różne paradygmaty programowania
- interfejsy, dziedziczenie, typy bezpośrednie
- obsługa wyjątków
- delegaty, technologia LINQ i mechanizm refleksji
- zarządzanie wątkami i programowanie asynchroniczne

## C#. NOWOCZESNY, ELEGANCKI, BEZPIECZNY!

C# jest jednym z lepiej dopracowanych języków programowania. Cechują go dojrzałość, prostota, nowoczesność i bezpieczeństwo. Został od podstaw zaprojektowany jako obiektowy. Stanowi integralną część platformy Microsoft .NET Framework. Jest ulubionym narzędziem profesjonalnych programistów, którym zależy na pisaniu kodu bezpiecznego, przejrzystego, wydajnego i prostego w konserwacji. W wersji 7.0 tego języka pojawiły się usprawnienia, dzięki którym programowanie stało się jeszcze bardziej efektywne i satysfakcjonujące.

Ta książka jest szóstym, zaktualizowanym i uzupełnionym wydaniem jednego z najlepszych podręczników programowania. Poza znakomitym kompendium języka C# znalazły się tu informacje o poszczególnych metodykach programowania, od sekwencyjnego aż po podstawy programowania deklaratywnego z wykorzystaniem atrybutów. Szczegółowo przedstawiono funkcje wprowadzone do wersji 7.0 języka, a także w platformie .NET Framework 4.7/.NET Core 2.0. Książka jest też wygodnym źródłem wiedzy o pewnych rzadko stosowanych konstrukcjach składniowych, specyficznych szczegółach i subtelnościach języka C#. Jasny i przejrzysty sposób prezentowania treści pozwoli na szybkie zrozumienie nawet najbardziej zawiłych zagadnień.

**Helion**  
helion.pl  
HELION SA  
ul. Kościuszki 1c  
44-100 Gliwice  
tel.: 32 230 98 63  
helion@helion.pl

Sprawdź nasze szkolenia  
SZKOLENIA  
AKADEMIA IT & BUSINESS  
WWW.SZKOLENIA.HELION.PL

KOD KORZYŚCI  
Sięgnij po więcej!



ISBN 978-83-283-5780-8



9 788328 357808

INFORMATYKA W NAJLEPSZYM WYDANIU

Cena: 129,00 zł