WYDANIE VII

C# 8.0

Kompletny przewodnik dla praktyków

MARK MICHAELIS

oraz redaktorzy techniczni:

ERIC LIPPERT

i KEVIN BOST

IntelliTect

Helion

Tytuł oryginału: Essential C# 8.0 (7th Edition)

Tłumaczenie: Tomasz Walczak

ISBN: 978-83-283-7567-3

Authorized translation from the English language edition, entitled: ESSENTIAL C# 8.0, 7th Edition by MARK MICHAELIS; published by Pearson Education, Inc, publishing as Addison-Wesley Professional. Copyright © 2021 Pearson Education, Inc.

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

Polish language edition published by Helion SA, Copyright © 2021.

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 jakiejkolwiek 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*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/c8kpp7
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/c8kpp7.zip

Printed in Poland.

- Kup książkę
- Poleć książkę
- Oceń książkę

- Księgarnia internetowa
- Lubię to! » Nasza społeczność

Spis treści

Spis rysunków *11*
Spis tabel *13*
Przedmowa *15*
Wprowadzenie *17*
O autorze *28*
Podziękowania *29*

1. Wprowadzenie do języka C# 31
Witaj, świecie 32
Podstawy składni języka C# 42
Korzystanie ze zmiennych 49
Dane wejściowe i wyjściowe w konsoli 52
Wykonywanie kodu w środowisku zarządzanym iplatforma CLI 59
Różne wersje platformy .NET 63
Podsumowanie 67

2. Typy danych 69
Podstawowe typy liczbowe 69
Inne podstawowe typy 77
Konwersje typów danych 91
Podsumowanie 97

3. Jeszcze o typach danych 99
Kategorie typów 99
Deklarowanie typów umożliwiających stosowanie wartości null 102
Zmienne lokalne z niejawnie określanym typem danych 106
Krotki 108
Tablice 114
Podsumowanie 130

Kup ksi k Pole ksi k

4. Operatory i przepływ sterowania 131

Operatory 132
Zarządzanie przepływem sterowania 145
Bloki kodu ({}) 150
Bloki kodu, zasięgi i przestrzenie deklaracji 152
Wyrażenia logiczne 154
Programowanie z użyciem wartości null 158
Operatory bitowe (<<, >>, |, &, ^, ~) 164
Instrukcje związane z przepływem sterowania — ciąg dalszy 169
Instrukcje skoku 179
Dyrektywy preprocesora języka C# 184
Podsumowanie 191

5. Metody i parametry 193

Wywoływanie metody 194
Deklarowanie metody 199
Dyrektywa using 204
Zwracane wartości i parametry metody Main() 208
Zaawansowane parametry metod 211
Rekurencja 220
Przeciążanie metod 223
Parametry opcjonalne 226
Podstawowa obsługa błędów z wykorzystaniem wyjątków 229
Podsumowanie 241

6. Klasy 243

Deklarowanie klasy i tworzenie jej instancji 246
Pola instancji 249
Metody instancji 251
Stosowanie słowa kluczowego this 252
Modyfikatory dostępu 258
Właściwości 260
Konstruktory 274
Konstruktory a właściwości typów referencyjnych niedopuszczających wartości null 283
Atrybuty dopuszczające wartość null 286
Dekonstruktory 288
Składowe statyczne 290
Metody rozszerzające 298
Hermetyzacja danych 299
Klasy zagnieżdżone 302
Klasy częściowe 304
Podsumowanie 308

Kup ksi k Pole ksi k

7. Dziedziczenie 309
Tworzenie klas pochodnych 310
Przesłanianie składowych z klas bazowych 318
Klasy abstrakcyjne 328
Wszystkie klasy są pochodne od System.Object 334
Dopasowanie do wzorca za pomocą operatora is 335
Dopasowanie do wzorca w wyrażeniu switch 340
Unikaj dopasowania do wzorca, gdy możliwy jest polimorfizm 341
Podsumowanie 343

8. Interfejsy 345
Wprowadzenie do interfejsów 346
Polimorfizm oparty na interfejsach 347
Implementacja interfejsu 351
Przekształcanie między klasą z implementacją i interfejsami 356
Dziedziczenie interfejsów 356
Dziedziczenie po wielu interfejsach 359
Metody rozszerzające i interfejsy 359
Zarządzanie wersjami 361
Metody rozszerzające a domyślne składowe interfejsu 374
Interfejsy a klasy abstrakcyjne 375
Interfejsy a atrybuty 377
Podsumowanie 377

9. Typy bezpośrednie 379
Struktury 383
Opakowywanie 388
Wyliczenia 395
Podsumowanie 405

10. Dobrze uformowane typy 407
Przesłanianie składowych z klasy object 407
Przeciążanie operatorów 418
Wskazywanie innych podzespołów 425
Hermetyzacja typów 431
Definiowanie przestrzeni nazw 433
Komentarze XML-owe 436
Odzyskiwanie pamięci 440
Porządkowanie zasobów 443
Leniwe inicjowanie 455
Podsumowanie 457

11. Obsługa wyjątków 459
Wiele typów wyjątków 459
Przechwytywanie wyjątków 462
Ponowne zgłaszanie przetwarzanego wyjątku 463
Ogólny blok catch 465
Wskazówki związane z obsługą wyjątków 466

Definiowanie niestandardowych wyjątków 469
Ponowne zgłaszanie opakowanego wyjątku 471
Podsumowanie 475

12. Typy generyczne 477
Język C# bez typów generycznych 478
Wprowadzenie do typów generycznych 482
Ograniczenia 493
Metody generyczne 507
Kowariancja i kontrawariancja 511
Wewnętrzne mechanizmy typów generycznych 517
Podsumowanie 521

13. Delegaty i wyrażenia lambda 523
Wprowadzenie do delegatów 524
Deklarowanie typu delegata 527
Wyrażenia lambda 534
Lambdy w postaci instrukcji 535
Metody anonimowe 539
Delegaty nie zapewniają równości strukturalnej 541
Zmienne zewnętrzne 543
Drzewo wyrażeń 547
Podsumowanie 553

14. Zdarzenia 555
Implementacja wzorca publikuj-subskrybuj za pomocą delegatów typu multicast 556
Zdarzenia 569
Podsumowanie 578

15. Interfejsy kolekcji ze standardowymi operatorami kwerend 579
Inicjatory kolekcji 580
Interfejs IEnumerable<T> sprawia, że klasa staje się kolekcją 582
Standardowe operatory kwerend 587
Typy anonimowe w technologii LINQ 615
Podsumowanie 622

16. Technologia LINQ i wyrażenia z kwerendami 623
Wprowadzenie do wyrażeń z kwerendami 624
Wyrażenia z kwerendą to tylko wywołania metod 639
Podsumowanie 641

17. Tworzenie niestandardowych kolekcji 643
Inne interfejsy implementowane w kolekcjach 644
Podstawowe klasy kolekcji 646
Udostępnianie indeksera 661
Zwracanie wartości null lub pustej kolekcji 664
Iteratory 665
Podsumowanie 677

Kup ksi k Pole ksi k

18. Refleksja, atrybuty i programowanie dynamiczne 679
Mechanizm refleksji 680
Operator nameof 689
Atrybuty 690
Programowanie z wykorzystaniem obiektów dynamicznych 705
Podsumowanie 714

19. Wprowadzenie do wielowątkowości 717
Podstawy wielowątkowości 719
Zadania asynchroniczne 724
Anulowanie zadania 741
Używanie przestrzeni nazw System.Threading 746
Podsumowanie 748

20. Programowanie z wykorzystaniem wzorca TAP 749
Synchroniczne wykonywanie operacji o wysokiej latencji 750
Asynchroniczne wywoływanie operacji o dużej latencji za pomocą biblioteki TPL 752
Asynchroniczność oparta na zadaniach oraz instrukcjach async i await 756
Dodanie możliwości zwracania typu ValueTask<T> w metodach asynchronicznych 761
Strumienie asynchroniczne 763
Interfejs IAsyncDisposable a deklaracje i instrukcje await using 766
Używanie technologii LINQ razem z interfejsem IAsyncEnumerable 767
Zwracanie wartości void w metodach asynchronicznych 769
Asynchroniczne lambdy i funkcje lokalne 772
Programy szeregujące zadania i kontekst synchronizacji 777
Modyfikatory async i await w programach z interfejsem użytkownika z systemu Windows 779
Podsumowanie 782

21. Równoległe iteracje 783
Równoległe wykonywanie iteracji pętli 783
Równoległe wykonywanie kwerend LINQ 791
Podsumowanie 796

22. Synchronizowanie wątków 797
Po co stosować synchronizację? 798
Zegary 822
Podsumowanie 823

23. Współdziałanie między platformami i niezabezpieczony kod 825
Mechanizm P/Invoke 826
Wskaźniki i adresy 837
Wykonywanie niezabezpieczonego kodu za pomocą delegata 845
Podsumowanie 846

Kup ksi k Pole ksi k

24. Standard CLI 847
Definiowanie standardu CLI 847
Implementacje standardu CLI 848
Specyfikacja .NET Standard 851
Biblioteka BCL 851
Kompilacja kodu w języku C# na kod maszynowy 852
Środowisko uruchomieniowe 853
Podzespoły, manifesty i moduły 857
Język Common Intermediate Language 859
Common Type System 860
Common Language Specification 861
Metadane 861
Architektura .NET Native i kompilacja AOT 862
Podsumowanie 862

Kup ksi k Pole ksi k

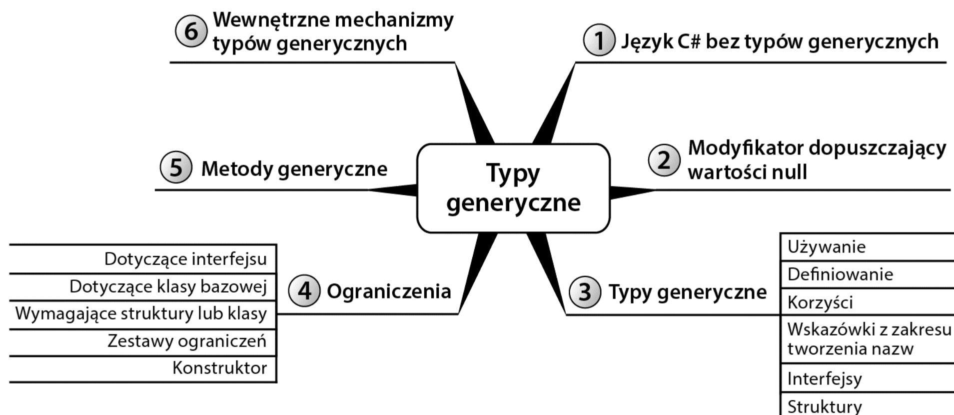
12

Typy generyczne

GDY ZACZNIESZ TWORZYĆ BARDZIEJ złożone projekty, będziesz potrzebował lepszego sposobu na ponowne wykorzystywanie i dostosowywanie istniejącego oprogramowania. Aby ułatwić wielokrotne wykorzystanie kodu (a zwłaszcza algorytmów), w języku C# udostępniono mechanizm **typów generycznych**.

Początek 2.0

Typy generyczne w języku C# są składniowo podobne do typów generycznych z Javy i szablonów z języka C++. We wszystkich trzech wymienionych językach wspomniane mechanizmy umożliwiają jednokrotne zaimplementowanie algorytmów i wzorców. Nie są potrzebne odrębne implementacje dla każdego typu, dla którego dany algorytm lub wzorzec działa. Jednak typy generyczne w języku C# znacznie różnią się od typów generycznych z Javy i szablonów z języka C++, jeśli chodzi o szczegóły implementacji oraz wpływ tych mechanizmów na system typów. Podobnie jak metody są bardziej wartościowe, ponieważ mogą przyjmować argumenty, tak typy i metody przyjmujące argumenty określające typ dają dodatkowe możliwości.



Typy generyczne zostały dodane do środowiska uruchomieniowego i języka C# w wersji 2.0.

Kup książkę Poleć książkę

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

2.0

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

W programach stos często służy do umożliwiania wielokrotnego cofania operacji. Na przykład na listingu 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.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 określanym 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;
```

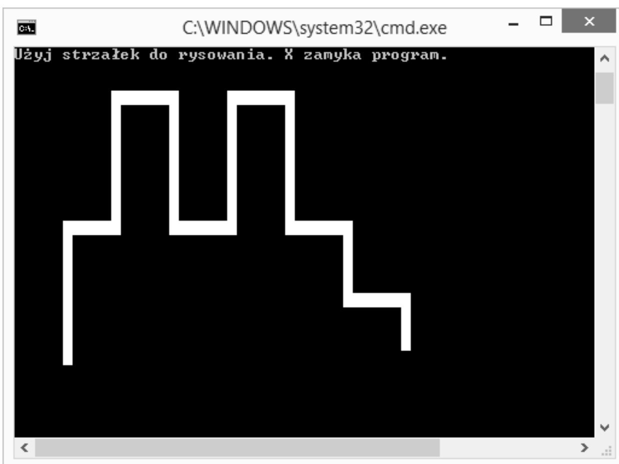
Kup książkę Poleć książkę

```
        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);
  // ...
}
```

Kup książkę Poleć książkę

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

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

  // ...
}

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

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

  ...
}
...
```

Kup ksi k Pole ksi k

Na listingu 12.4 pokazano możliwe implementacje typów `NullableInt` i `NullableGuid`. Jeśli w programie potrzebne są dodatkowe typy bezpośrednie z obsługą wartości `null`, trzeba utworzyć nową strukturę z właściwościami działającymi dla odpowiedniego typu. Każdą poprawkę 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

2.0

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

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

  ...
}
```

Choć ta technika wymaga utworzenia tylko jednej implementacji typu z obsługą wartości `null`, środowisko uruchomieniowe zawsze opakowuje wtedy typy bezpośrednie, gdy ustawiana 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.

Kup ksi k Pole ksi k

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;
```

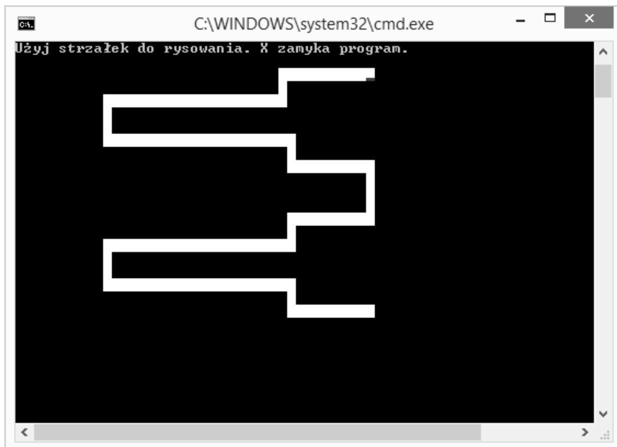
2.0

```
        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 WYJŚIOWE 12.2.

2.0



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()`.

Kup ksi k Pole ksi k

Listing 12.7. Deklarowanie generycznej klasy Stack<T>

```
public class Stack<T>
{
  public Stack(int maxSize)
  {
    InternalItems = new T[maxSize];
  }
  // 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()
  {
    ...
  }
}
```

2.0

Zalety typów generycznych

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.

Kup ksi k Pole ksi k

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 wzorzec 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

2.0

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<TEntity>` nazwa parametru określającego typ to `TEntity`.

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

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

Wskazówki

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

ROZWAŻ podanie ograniczenia w nazwie parametru określającego typ.

Generyczne interfejsy i struktury

C# obsługuje stosowanie typów generycznych w różnych konstrukcjach języka, w tym w interfejsach i strukturach. Składnia jest tu identyczna jak dla klas. Aby zadeklarować interfejs z parametrem określającym typ, umieść ten parametr w nawiasie ostrym bezpośrednio po nazwie interfejsu. Pokazano to w przykładowym interfejsie `IPair<T>` na listingu 12.8.

Listing 12.8. Deklarowanie generycznego interfejsu

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

Kup książkę Poleć książkę

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 w języku C# programiści musieli posługiwać się zestawem interfejsów z przestrzeni nazw `System.Collections`. Te interfejsy (podobnie jak klasy z ich implementacjami) działały tylko dla typu `object`, dlatego dostęp do elementów z takich klas zawsze wymagał rzutowania. Dzięki zastosowaniu generycznych interfejsów bezpiecznych ze względu na typ można uniknąć rzutowania.

ZAGADNIENIE DLA ZAAWANSOWANYCH

Wielokrotne implementowanie jednego interfejsu w tej samej klasie

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

Kup ksi k Pole ksi k

```
  {
    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; }
}
```

Kup książkę Poleć książkę

Określanie wartości domyślnej za pomocą operatora default

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 jest to typ dopuszczający wartość `null`, można użyć tej wartości. Ta technika nie zadziała jednak, jeśli `T` jest typem niedopuszczającym wartości `null`.

Listing 12.12. Jeśli nie wszystkie pola zostaną zainicjowane, wystąpi błąd kompilacji

2.0

```
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;
  // }

  // ...
}
```

Początek 7.0

Aby umożliwić rozwiązanie tego problemu, w języku C# udostępniono operator `default`. Na przykład wartość domyślną typu `int` można podać jako `default` (jeśli używana jest wersja C# 7.1 lub nowsza). Gdy używany jest typ `T` (potrzebny w polu `Second`), można podać wartość `default`. 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;
  }

  // ...
}
```

Operator `default` pozwala podać wartość domyślną dowolnego typu (dotyczy to także parametrów określających typ).

W wersjach starszych niż C# 7.1 konieczne było podawanie w operatorze `default` parametru określającego typ, na przykład `Second = default(T)`. W C# 7.1 wprowadzono możliwość używania operatora `default` bez podawania parametru, jeśli możliwe jest wywnioskowanie

Kup książkę Poleć książkę

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.

Początek 8.0

Zauważ, że wszystkie typy dopuszczające wartość null mają wartość domyślną null. To samo dotyczy typów generycznych dopuszczających wartość null (na przykład default(T?)). Ponadto null jest wartością domyślną wszystkich typów referencyjnych. Dlatego po dodaniu w wersji C# 8.0 obsługi typów referencyjnych dopuszczających wartość null użycie operatora default do typu referencyjnego niedopuszczającego tej wartości skutkuje ostrzeżeniem. Dlatego kod stosujący operator default do typów referencyjnych w C# 7.0 i starszych wersjach będzie generował ostrzeżenia, jeśli zostanie aktualizowany do wersji C# 8.0 z obsługą dopuszczania wartości null. Dlatego w wersjach starszych niż C# 8.0 (i oczywiście w aktualnych wersjach) należy unikać przypisywania default i null do typów referencyjnych, chyba że null ma być dopuszczalną wartością. Jeśli to możliwe, pozostawiaj zmienną niezainicjowaną do czasu, gdy dostępna będzie prawidłowa wartość do przypisania. W scenariuszach takich jak na listingu 12.13, gdzie w konstruktorze odpowiednia wartość zmiennej Second jest nieznana, zmienna ta będzie równa null dla typów referencyjnych i dla typów bezpośrednich dopuszczających tę wartość. Dlatego użycie operatora default (potencjalnie przypisującego null) do właściwości typu generycznego T skutkuje ostrzeżeniem. Aby odpowiednio poradzić sobie z tym ostrzeżeniem, trzeba zadeklarować właściwość Second typu T? i określić, czy T jest typu referencyjnego, czy bezpośredniego. Umożliwiają to ograniczenia dotyczące klasy lub struktury opisane w podrozdziale „Ograniczenia wymagające struktury lub klasy (struct i class)". Wszystkie te rozważania prowadzą do ogólniejszej wskazówki, zgodnie z którą nie należy stosować operatora default do typów generycznych, chyba że używane jest w nim ograniczenie dotyczące klasy lub struktury.

2.0

Koniec 8.0

Koniec 7.0

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.

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)
```

Kup ksi k Pole ksi k

```
  {
    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

2.0

```
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.

Początek 7.0

ZAGADNIENIE DLA POCZĄTKUJĄCYCH

Krotki — typy o różnej arności

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>: ... {...}
```

Kup książkę Poleć książkę

```
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.

2.0

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>.
ValueTuple<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()`.

Początek 4.0 Koniec 7.0 Koniec 4.0

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).

2.0

Listing 12.18. Zagnieżdżone typy generyczne

```
class Container<T1, T2>
{
  // Klasy zagnieżdżone dziedziczą parametry określające typ.
  // Ponowne wykorzystanie nazwy takiego parametru
  // prowadzi do zgłoszenia ostrzeżenia.
  class Nested<T2>
  {
    void Method(T1 param0, T2 param1)
    {
    }
  }
}
```

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

Wskazówka

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

Ograniczenia

Typy generyczne umożliwiają definiowanie ograniczeń dotyczących parametrów określających typ. Te ograniczenia gwarantują, że typy podane jako argumenty będą zgodne z wymaganymi regułami. Przyjrzyj się przykładowej klasie `BinaryTree<T>` z listingu 12.19.

Kup ksi k Pole ksi k

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.

2.0

Listing 12.20. Do działania interfejsu potrzebny jest parametr określający typ

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

  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;
}
```

Kup ksi k   Pole ksi k

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 zrzutować 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

2.0

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

  public T Item { get; set; }
  public Pair<BinaryTree<T>?>? SubItems
  {
    get{ return _SubItems; }
    set
    {
      switch(value)
      {
        // Obsługa null została pominięta, aby przykład był bardziej czytelny.

        // Używanie dopasowania do wzorca z wersji C# 8.0. W starszych wersjach
        // zastosuj sprawdzanie wartości null.
        case {
            First: {Item: IComparable<T> first },
            Second: {Item: T second } }:
          if (first.CompareTo(second) < 0)
          {
            // Element first jest mniejszy niż second.
          }
          else
          {
            // Element second jest mniejszy lub równy względem first.
          }
          break;
        default:
          throw new InvalidCastException(
            @$"Nie da się posortować elementów. Typ {
              typeof(T) } nie obsługuje interfejsu IComparable<T>");
      }
      _SubItems = value;
    }
  }
  private Pair<BinaryTree<T>?>? _SubItems;
}
```

Kup książkę Poleć książkę

Niestety, jeśli teraz zadeklarujesz zmienną klasy `BinaryTree<Typ>`, a podany typ nie zawiera implementacji interfejsu `IComparable<Typ>`, nie da się posortować elementów i 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, wymagają 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 BinaryTree(T item)
  {
    Item = item;
  }
  public T Item { get; set; }
  public Pair<BinaryTree<T>> SubItems
  {
    get{ return _SubItems; }
    set
    {
      switch(value)
      {
        // Obsługa wartości null została pominięta, aby zwiększyć czytelność.

        // Używanie dopasowania do wzorca z wersji C# 8.0. W starszych
        // wersjach zastosuj sprawdzanie wartości null.
        case {
            First: {Item: T first },
            Second: {Item: T second } }:
          if (first.CompareTo(second) < 0)
          {
            // Element first jest mniejszy niż second.
          }
```

```
          else
          {
            // Element second jest mniejszy lub równy względem first.
          }
          break;
        default:
          throw new InvalidCastException(
            @$"Nie da się posortować elementów. Typ {
              typeof(T) } nie obsługuje interfejsu IComparable<T>");
      }
      _SubItems = value;
    }
  }
  private Pair<BinaryTree<T>?>? _SubItems;
}
```

Choć zmiany w kodzie są niewielkie, za wykrywanie błędów odpowiada teraz kompilator, a nie środowisko uruchomieniowe. Jest to istotna różnica. 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.

2.0

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 parametru określającego typ

Czasem możesz oczekiwać, że argument określający typ da się przekształcić na konkretny typ. W tym celu można użyć **ograniczenia dotyczącego parametru określającego typ**, co ilustruje listing 12.23.

Kup ksi k Pole ksi k

Listing 12.23. Deklarowanie ograniczenia dotyczącego parametru określającego typ

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

2.0

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.

Jako ograniczeń dotyczących klasy nie można używać niektórych typów „specjalnych". Więcej na ten temat dowiesz się z zagadnienia dla zaawansowanych „Wymogi związane z ograniczeniami" w dalszej części rozdziału.

Początek 7.3

Od wersji C# 7.3 jako ograniczenie można podać `System.Enum`, aby zagwarantować, że parametr określający typ jest wyliczeniem. Jako ograniczenia nie można jednak zastosować typu `System.Array`. Jest to jednak mało istotne, ponieważ i tak zaleca się stosowanie innych typów i interfejsów kolekcji (zobacz rozdział 15.).

ZAGADNIENIE DLA ZAAWANSOWANYCH

Ograniczenia dotyczące delegatów

W C# 7.3 jako ograniczenie można też podać typ `System.Delegate` (i `System.MulticastDelegate`). Dzięki temu można dodawać delegaty (za pomocą statycznej metody `Combine()`) i je odłączać (za pomocą statycznej metody `Remove()`) w sposób bezpieczny ze względu na typ. Nie ma możliwości wywoływania delegatów dla typu generycznego z zachowaniem ścisłej kontroli typów, jednak podobny efekt można uzyskać za pomocą metody `DynamicInvoke()`. Wewnętrznie używa ona mechanizmu refleksji. Choć typ generyczny nie może bezpośrednio wywoływać

Kup książkę Poleć książkę

delegata (bez używania metody DynamicInvoke()), to można wywoływać delegaty za pomocą bezpośredniej referencji do typu T na etapie kompilacji. Możesz na przykład wywołać metodę Combine() i zrzutować wynik na oczekiwany typ za pomocą mechanizmu dopasowania do wzorca, jak ilustruje to listing 12.24.

Listing 12.24. Deklarowanie typu generycznego z ograniczeniem MulticastDelegate

```
static public object? InvokeAll<TDelegate>(
  object?[]? args, params TDelegate[] delegates)
  // Nie można zastosować ograniczenia Action lub Func.
  where TDelegate : System.MulticastDelegate
{
  switch (Delegate.Combine(delegates))
  {
    case Action action:
      action();
      return null;
    case TDelegate result:
      return result.DynamicInvoke(args);
    default:
      return null;
  };
}
```

2.0

W tym przykładzie kod próbuje zrzutować obiekt na typ Action przed wywołaniem. Jeśli kończy się to niepowodzeniem, obiekt jest rzutowany na typ TDelegate i wywoływany za pomocą metody DynamicInvoke().

Zauważ, że poza typami generycznymi typ T jest znany, dlatego można bezpośrednio wywołać obiekt po wywołaniu Combine():

```
Action? result =
  (Action?)Delegate.Combine(actions);
result?.Invoke();
```

Zwróć uwagę na komentarz z listingu 12.24. Choć jako ograniczenie można podawać typy System.Delegate i System.MulticastDelegate, nie można używać konkretnych typów delegatów takich jak Action, Func<T> i pokrewne typy.

Ograniczenie unmanaged

W C# 7.3 wprowadzone zostało ograniczenie unmanaged, które powoduje, że parametrem określającym typ może być: sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal, bool, wyliczenie, wskaźnik lub struktura, w której wszystkie pola są niezarządzane. Pozwala to między innymi używać operatora sizeof (lub stackalloc; zobacz rozdział 22.) do parametru określającego typ z ograniczeniem unmanaged.

Początek 8.0

Przed wersją C# 8.0 ograniczenie unmanaged pozwalało używać jako parametrów określających typ tylko **niegenerycznych typów strukturalnych** (czyli niegenerycznych typów bezpośrednich). W C# 8.0 nie jest to wymagane. Możesz teraz zadeklarować zmienną typu Thing<Thing<int>> nawet wtedy, gdy w typie Thing<T> dla T używane jest ograniczenie unmanaged.

Koniec 7.3

Kup książkę Poleć książkę

Ograniczenie notnull

Na listingu 12.23 używane jest też drugie ograniczenie: braku wartości null. Informuje o nim kontekstowe słowo kluczowe notnull. To ograniczenie skutkuje ostrzeżeniem, jeśli jako opatrzony ograniczeniem notnull parametr określający typ podany jest typ dopuszczający wartość null. Na przykład deklaracja EntityDictionary<string?, EntityBase> spowoduje wtedy ostrzeżenie: Obsługa wartości null w argumencie typu "string?" jest niezgodna z ograniczeniem "notnull".

Słowa kluczowego notnull nie można łączyć z ograniczeniami struct i class, które domyślnie nie dopuszczają wartości null (co jest opisane dalej).

Ograniczenia wymagające struktury lub klasy (struct i class)

2.0

Innym przydatnym ograniczeniem w typach generycznych jest możliwość zażądania, by typ podany w argumencie był typem bezpośrednim bez obsługi wartości null lub typem referencyjnym. Zamiast określać klasę, po której T ma dziedziczyć, można podać słowo kluczowe struct lub class. Ilustruje to listing 12.25.

Listing 12.25. Dodawanie wymogu, by jako parametr określający typ podawano typ bezpośredni

```
public struct Nullable<T> :
  IFormattable, IComparable,
  IComparable<Nullable<T>>, INullable
  where T : struct
{
  // ...
  public static implicit operator T?(T value) =>
    new T?(value);

  public static explicit operator T(T? value) => value!.Value;
}
```

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 też interfejsem, delegatem lub typem tablicowym.

W wersji C# 8.0 ograniczenie class domyślnie oznacza typ niedopuszczający wartości null (przy założeniu, że włączona jest obsługa typów referencyjnych dopuszczających wartość null). Użycie typu referencyjnego dopuszczającego wartość null spowoduje wtedy, że kompilator wyświetli ostrzeżenie. Aby umożliwić użycie typu referencyjnego dopuszczającego wartość null, dodaj modyfikator ? do ograniczenia class. Przypomnij sobie przedstawioną w rozdziale 10. klasę WeakReference<T>. Ponieważ mechanizm przywracania pamięci uwzględnia tylko typy referencyjne, w tej klasie ograniczenie class jest dodawane tak:

```
public sealed partial class WeakReference<T> : ISerializable
  where T : class?
{ ... }
```

To powoduje, że parametr określający typ (T) musi być typu referencyjnego; może to być typ dopuszczający wartość null.

Kup ksi k Pole ksi k

W ograniczeniu `struct` (inaczej niż w ograniczeniu `class`) nie można stosować modyfikatora ?. Zamiast tego można określić dopuszczalność wartości `null`, używając parametru. Na przykład na listingu 12.25 w operatorach jawnej i niejawnej konwersji używane są parametry `T` i `T?`, co jest informacją, czy dozwolona jest wersja typu `T` dopuszczająca wartość `null`, czy niedopuszczająca takiej wartości. Dlatego wymogi wobec parametru określającego typ są podane w deklaracji składowej, a nie jako ograniczenie typu.

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

Ograniczenie `struct` ma pewną cechę — uniemożliwia podawanie typów bezpośrednich dopuszczających wartość `null`. Z czego to wynika? Typy bezpośrednie dopuszczające wartość `null` są implementowane za pomocą typu generycznego `Nullable<T>`, w którym do `T` stosowane jest ograniczenie `struct`. Gdyby typ bezpośredni dopuszczający wartość `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??`.

2.0

Koniec 8.0

Zestawy ograniczeń

Dla parametru określającego typ można ustawić dowolną liczbę ograniczeń dotyczących interfejsu, ale tylko jedno ograniczenie dotyczące klasy (podobnie w klasie można zaimplementować dowolną liczbę interfejsów, ale dziedziczyć po tylko jednej innej klasie). Każde nowe ograniczenie jest deklarowane na rozdzielonej przecinkami liście, która znajduje się po nazwie parametru typu generycznego i dwukropku. Jeśli występuje więcej niż jeden parametr określający typ, słowo kluczowe `where` należy umieścić przed każdym takim parametrem, do którego dodawane są ograniczenia. Na listingu 12.26 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 12.26. 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 (są one łączone relacją I). 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 12.27 metoda `MakeValue()` klasy `EntityDictionary<TKey, TValue>` musi tworzyć obiekt typu podanego jako parametr `TValue`.

Listing 12.27. Ograniczenie wymagające dostępności konstruktora domyślnego

2.0

```
public class EntityBase<TKey>
  where TKey: notnull
{
  public EntityBase(TKey key)
  {
    Key = key;
  }

  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();
    {
      Key = key;
    }
    Add(newEntity.Key, newEntity);
    return newEntity;
  }

  // ...
}
```

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

Na listingu 12.27 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

Kup książkę Poleć książkę

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 12.28 jest nieprawidłowy.

Listing 12.28. 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;
}
```

2.0

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 12.29).

Listing 12.29. 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);
}
...
```

Kup ksi k Pole ksi k

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 12.29 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 12.30.

Listing 12.30. 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);
  }
}
```

Dziedziczenie ograniczeń

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 12.31.

Listing 12.31. Jawnie podawane „odziedziczone” ograniczenia

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

```
// BŁĄD:
// Możliwa musi być konwersja typu 'U' na typ
```

Kup książkę Poleć książkę

```
// '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 12.31 klasa `EntityBase<T>` wymaga, by podany jako argument typ `U` (używany jako parametr `T` w wyniku deklaracji klasy bazowej `EntityBase<U>`) zawierał implementację interfejsu `IComparable<U>`. Dlatego w klasie `Entity<U>` trzeba zastosować to samo ograniczenie do `U`. W przeciwnym razie wystąpi błąd kompilacji. Ten wzorzec zwiększa świadomość programisty i uwidacznia ograniczenia z klasy bazowej w klasie pochodnej. Pozwala to uniknąć niejasności, które mogą wystąpić, gdy programista używa klasy pochodnej i odkrywa ograniczenie, ale nie rozumie, z czego ono wynika.

Na razie nie omówiono w książce metod generycznych. Zapoznasz się z nimi w dalszej części rozdziału. Zapamiętaj tylko, że także metody mogą być generyczne i można w nich dodawać ograniczenia parametrów określających typ. Jak interpretowane są te ograniczenia, gdy wirtualna metoda generyczna jest dziedziczona lub przesłaniana? Inaczej niż w przypadku ograniczeń parametrów określających typ w klasie generycznej, ograniczenia w nowych wersjach wirtualnych metod generycznych (i w składowych z jawnie podawanym interfejsem) są dziedziczone niejawnie i nie można ich ponownie zadeklarować (zobacz listing 12.32).

2.0

Listing 12.32. 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.

Kup ksi k Pole ksi k

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 określonej klasy lub `notnull` z ograniczeniami `struct` i `class`. Ponadto nie można utworzyć ograniczenia wymagającego użycia typu pochodnego od jednego z typów specjalnych (takich jak `object`, typy tablicowe lub `System.ValueType`). Wcześniej zostało opisane, że od wersji C# 7.3 jako ograniczenie można stosować typy `System.Enum` (i typy wyliczeniowe), `System.Delegate` i `System.MulticastDelegate`. Nie można jednak stosować ograniczeń w postaci konkretnych typów delegatów, takich jak `Action`, `Func<T>` lub pokrewne typy.

W niektórych sytuacjach przydatne byłoby wprowadzenie dodatkowych reguł związanych z ograniczeniami (dotyczy to na przykład zażądania dostępności konstruktora domyślnego). W przedstawionych dalej podrozdziałach znajdziesz przykłady niedozwolonych ograniczeń.

2.0

Ograniczenia dotyczące operatorów są niedozwolone

Wszystkie typy generyczne automatycznie umożliwiają porównania za pomocą operatorów `==` i `!=` oparte na niejawnym rzutowaniu wartości na typ `object` (ponieważ wszystkie wartości są obiektami). 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 12.33 nie zadziała.

Listing 12.33. 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.

Kup książkę Poleć książkę

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 12.34 jest niedozwolony.

Listing 12.34. 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.

2.0

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)
```

Kup książkę Poleć książkę

```
    where T : IComparable<T>
  {
    T minimum = first;

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

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

2.0

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.

Kup ksi k Pole ksi k

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.

2.0

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.

Kup ksi k Pole ksi k

Listing 12.38. Dodawanie ograniczeń w metodach generycznych

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

2.0

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:

Kup ksi k Pole ksi k

```
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.

2.0

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.

Kup ksi k Pole ksi k

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

2.0

```
//...
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.

Początek 4.0

Umożliwianie kowariancji za pomocą modyfikatora out stosowanego do parametru określającego typ

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 udostępnia `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)[1].

[1] Opisany mechanizm wprowadzono w wersji C# 4.0.

Kup ksi k Pole ksi k

Listing 12.42. Rozwiązanie z potencjalnie możliwą kowariancją

```
interface IReadOnlyPair<T>
{
  T First { get; }
  T Second { get; }
}
interface IPair<T>
{
  T First { get; set; }
  T Second { get; set; }
}
public struct Pair<T> : IPair<T>, IReadOnlyPair<T>
{
  // ...
}
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 Buttercupt"),
        new Contact("Inigo Montoya") );
    IReadOnlyPair<PdaItem> pair = contacts;
    PdaItem pdaItem1 = pair.First;
    PdaItem pdaItem2 = pair.Second;
  }
}
```

2.0

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

Mimo to kod z listingu 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 IReadOnly ↪Pair<out T> umożliwia kompilatorowi stwierdzenie, że typ T rzeczywiście jest używany tylko dla danych wyjściowych — jako typ wartości zwracanych przez metodę lub przez właściwości przeznaczone tylko do odczytu. Typ ten nigdy nie jest używany dla parametrów formalnych lub w setterze właściwości. Na tej podstawie kompilator dopuszcza 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.

2.0

Umożliwianie kontrawariancji z użyciem modyfikatora in dla parametru określającego typ

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

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

Teraz załóżmy, że istnieje interfejs ICompareThings<T> zawierający metodę bool First ↪IsBetter<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*.

Kup książkę Poleć książkę

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[2].

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);
}
```

2.0

```
class Program
{
  class FruitComparer : ICompareThings<Fruit>
  { ... }
  static void Main()
  {
    // Dozwolone od 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.

[2] Opisany mechanizm jest dostępny od wersji 4.0.

Kup ksi k Pole ksi k

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

2.0

```
// 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.

Kup ksi k Pole ksi k

Obsługa niezabezpieczonej kowariancji w tablicach

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

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

2.0

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.

Kup ksi k   Pole ksi k

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
{
  private T[] _Items;
  // Pozostała część klasy.
}
```

2.0

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
  ...
}
```

Kup ksi k Pole ksi k

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.

2.0

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 argumencie).

Kup książkę Poleć książkę

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>();
```

2.0

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.

Kup ksi k Pole ksi k

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

Koniec 2.0

Kup ksi k Pole ksi k

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 światowej klasy ekspertem w dziedzinie zaawansowanego programowania w C#. Pracuje jako wykładowca na Easter Washington University. Od 1996 roku posiada tytuł Microsoft MVP. W 2007 roku został dyrektorem regionalnym Microsoftu, jest członkiem kilku zespołów oceniających projekty oprogramowania tej firmy (między innymi języka C# i technologii VSTS). Wraz z rodziną mieszka w Spokane w stanie Waszyngton.

C#. PRAKTYCZNE ROZWIĄZANIA RZECZYWISTYCH PROBLEMÓW!

C# jest jednym z najlepszych dzieł Microsoftu — cechuje go dojrzałość, prostota i nowoczesność. Został zaprojektowany jako język obiektowy i konsekwentnie jest rozwijany. Służy do tworzenia aplikacji sieciowych, mikrousług, aplikacji desktopowych, oprogramowania dla urządzeń mobilnych i internetu rzeczy. Ponadto C# jest językiem otwartym, pozwalającym na pisanie kodu bezpiecznego, przejrzystego, wydajnego i prostego w konserwacji. W wersji 8.0 pojawiły się funkcjonalności, które jeszcze bardziej usprawniają pracę programisty.

W książce między innymi:

- istotne konstrukcje w C#
- techniki programowania obiektowego w C#, w tym klasy, dziedziczenie i interfejsy
- typy generyczne, delegaty, wyrażenia lambda oraz refleksje i atrybuty
- strumienie asynchroniczne
- przetwarzanie równoległe i wielowątkowość
- współdziałanie z kodem niezarządzanym

To siódme, zaktualizowane i uzupełnione wydanie jednego z najlepszych podręczników programowania, docenianego przez programistów na każdym poziomie zaawansowania. Poza znakomitym samouczkiem języka C# znalazły się tu informacje o poszczególnych metodykach programowania. Książka zawiera także omówienie nowości w C#: typów referencyjnych dopuszczających wartość null, indeksów, przedziałów, rozbudowanego dopasowywania do wzorca, strumieni asynchronicznych i innych. Treść jest uporządkowana i przejrzysta, co nadaje podręcznikowi przystępną formę, a zawarte w nim wskazówki pomagają w ograniczeniu liczby błędów w kodzie.

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

Sprawdź nasze szkolenia!
SZKOLENIA
AKADEMIA IT & BUSINESS
HELIONSZKOLENIA.PL

KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-283-7567-3
9 788328 375673
Cena: 149,00 zł