



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: 978-83-283-7568-0

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 be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

Polish language edition published by Helion SA, Copyright © 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 kopowanie 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 właścicielami.

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_ebook

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:

<https://ftp.helion.pl/przyklady/c8kpp7.zip>

- [Poleć książkę na Facebook.com](#)
- [Kup w wersji papierowej](#)
- [Oceń książkę](#)

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

Dla mojej rodziny: Elisabeth, Benjamina, Hanny i Abigail.

*Zrezygnowaliście z męża i taty na niezliczone godziny potrzebne na pisanie tej książeczki
— często w chwilach, kiedy był najbardziej potrzebny.*

Dziękuję!

Dziękuję też moim znajomym i współpracownikom z IntelliTect.

*Jestem wdzięczny za to, że mnie zastępowaliście, gdy więcej czasu poświęcałem na pisanie
niż na wykonywanie swojej pracy, a także za pomoc przy wielu drobiazgach,
co pozwoliło ulepszyć treść książeczki i opracować procesy DevOps
zapewniające poprawne działanie tak rozbudowanego kodu bazowego.*





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 określonym typem danych 106

Krotki 108

Tablice 114

Podsumowanie 130

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. Klasы 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

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
- Wyciągania 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
Lambda 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-subskrybij 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

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

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



Spis rysunków

- Rysunek 1.1.** *Instalowanie rozszerzenia dla języka C# w edytorze Visual Studio Code* 33
- Rysunek 1.2.** *Okno dialogowe Utwórz nowy projekt* 36
- Rysunek 1.3.** *Okno dialogowe Konfiguruj nowy projekt* 37
- Rysunek 1.4.** *Okno dialogowe z plikiem Program.cs* 37
- Rysunek 3.1.** *Typy bezpośrednie przechowują dane bezpośrednio* 100
- Rysunek 3.2.** *Typy referencyjne wskazują na lokalizację na stercie* 101
- Rysunek 4.1.** *Wartości odpowiadające kolejnym pozycjom* 164
- Rysunek 4.2.** *Obliczanie wartości bajta bez znaku* 164
- Rysunek 4.3.** *Obliczanie wartości dla bajtów reprezentujących liczby ze znakiem* 165
- Rysunek 4.4.** *Liczby 12 i 7 przedstawione w postaci binarnej* 166
- Rysunek 4.5.** *Zwinięty obszar w środowisku Microsoft Visual Studio .NET* 191
- Rysunek 5.1.** *Przepływ sterowania w programie z obsługą wyjątków* 233
- Rysunek 6.1.** *Hierarchia klas* 245
- Rysunek 7.1.** *Refaktoryzacja w celu utworzenia klasy bazowej* 310
- Rysunek 8.1.** *Zastosowanie agregacji i interfejsu do rozwiązania problemu dziedziczenia po tylko jednej klasie* 361
- Rysunek 9.1.** *Typy bezpośrednie przechowują dane bezpośrednio* 380
- Rysunek 9.2.** *Typy referencyjne prowadzą do sterty* 381
- Rysunek 10.1.** *Tożsamość* 412
- Rysunek 10.2.** *Menu Projekt* 429
- Rysunek 10.3.** *Filtr Przeglądaj* 430
- Rysunek 10.4.** *Komentarze XML-owe jako podpowiedzi w środowisku IDE Visual Studio* 436
- Rysunek 13.1.** *Obiektowy model typów delegatów* 532
- Rysunek 13.2.** *Terminologia dotycząca funkcji anonimowych* 535
- Rysunek 13.3.** *Typy reprezentujące drzewo wyrażeń lambda* 549
- Rysunek 13.4.** *Typy reprezentujące drzewa wyrażeń jedno- i dwuargumentowych* 550

- Rysunek 14.1.** Diagram z sekwencją wywołań delegatów 564
Rysunek 14.2. Delegaty typu multicast połączone w łańcuch 565
Rysunek 14.3. Diagram sekwencji wywołań delegatów w sytuacji,
gdy zgłoszany jest wyjątek 567
- Rysunek 15.1.** Diagram klas przedstawiający interfejsy `IEnumerable<T>` i `IEnumerator` 584
Rysunek 15.2. Sekwencja operacji wywołujących wyrażenia lambda 597
Rysunek 15.3. Diagram Venna dotyczący kolekcji reprezentujących wynalazców i patenty 601
- Rysunek 17.1.** Hierarchia generycznych interfejsów implementowanych w kolekcjach 645
Rysunek 17.2. Diagram klas `List<>` 647
Rysunek 17.3. Diagram klasy `Dictionary` 653
Rysunek 17.4. Klasa sortowanych kolekcji 659
Rysunek 17.5. Diagram klasy `Stack<T>` 660
Rysunek 17.6. Diagram klasy `Queue<T>` 660
Rysunek 17.7. Diagramy klas `LinkedList<T>` i `LinkedListNode<T>` 661
Rysunek 17.8. Diagram sekwencyjny z instrukcją `yield return` 669
- Rysunek 18.1.** Klasa pochodna od klasy `MethodInfo` 686
- Rysunek 19.1.** Zmiany szybkości taktowania w czasie 717
Rysunek 19.2. Oś czasu w scenariuszu zakleszczenia 724
Rysunek 19.3. Diagramy klas `CancellationTokenSource` i `CancellationToken` 743
- Rysunek 20.1.** Przepływ sterowania w każdym zadaniu 760
Rysunek 20.2. `IAsyncEnumerable<T>` i powiązane interfejsy 765
- Rysunek 23.1.** Wskaźniki zawierają adresy danych 839
- Rysunek 24.1.** Kompilacja kodu w języku C# do kodu maszynowego 853
Rysunek 24.2. Podzespoły z używanymi w nich modułami i plikami 858



Spis tabel

- Tabela 1.1.** Słowa kluczowe języka C# 43
Tabela 1.2. Typy komentarzy w języku C# 57
Tabela 1.3. Najważniejsze wersje platformy .NET 63
Tabela 1.4. Wersje języka C# i platformy .NET 65
- Tabela 2.1.** Typy całkowitoliczbowe 70
Tabela 2.2. Typy zmiennoprzecinkowe 71
Tabela 2.3. Typ decimal 72
Tabela 2.4. Sekwencje ucieczki 79
Tabela 2.5. Metody statyczne typu string 83
Tabela 2.6. Metody typu string 84
- Tabela 3.1.** Przykładowy kod ilustrujący deklarowanie krotek i przypisanie do nich wartości 109
Tabela 3.2. Najważniejsze informacje o tablicach 115
Tabela 3.3. Typowe błędy związane z programowaniem tablic 129
- Tabela 4.1.** Instrukcje związane z przepływem sterowania 146
Tabela 4.2. Operatory relacyjne i równości 155
Tabela 4.3. Wartości zwracane przez operator XOR 156
Tabela 4.4. Sprawdzanie wartości null 159
Tabela 4.5. Dyrektywy preprocesora 185
Tabela 4.6. Przykładowe ostrzeżenia 186
Tabela 4.7. Priorytety operatorów 192
- Tabela 5.1.** Często używane przestrzenie nazw 196
Tabela 5.2. Często używane typy wyjątków 236
- Tabela 6.1.** Atrybuty dotyczące dopuszczania wartości null 286
- Tabela 7.1.** Dlaczego stosować modyfikator new? 322
Tabela 7.2. Składowe klasy System.Object 334
Tabela 7.3. Dopasowanie do wzorca w postaci typu, stałej i var za pomocą operatora is 336

- Tabela 8.1.** Mechanizmy do refaktoryzacji składowych domyślnych interfejsów 366
Tabela 8.2. Porównanie klas abstrakcyjnych i interfejsów 376
- Tabela 9.1.** Kod CIL z instrukcjami opakowywania 389
- Tabela 10.1.** Modyfikatory dostępu 433
- Tabela 13.1.** Uwagi i przykłady dotyczące wyrażeń lambda 538
- Tabela 15.1.** Proste standardowe operatory kwerend 613
Tabela 15.2. Funkcje agregujące z klasy System.Linq.Enumerable 614
- Tabela 19.1.** Lista dostępnych wartości wyliczeniowych typu TaskContinuationOptions 732
- Tabela 22.1.** Przykładowe wykonanie programu opisane za pomocą pseudokodu 799
Tabela 22.2. Metody klasy Interlocked związane z synchronizacją 809
Tabela 22.3. Ścieżka wykonania w kodzie z synchronizacją opartą na typie ManualResetEvent 816
Tabela 22.4. Klasy kolekcji przetwarzanych równolegle 818
- Tabela 24.1.** Implementacje standardu CLI 849
Tabela 24.2. Często używane akronimy związane z językiem C# 863



Przedmowa

WITAJCIE. MACIE PRZED sobą owoc współpracy najlepszych autorów, o których można marzyć w świecie książek o języku C# (i nie tylko!). Seria *Essential C#* Marka Michaelisa jest klasykiem już od lat, jednak gdy poznałem Marka, miała dopiero powstać.

Gdy w 2005 roku wprowadzano technologię LINQ (ang. *Language Integrated Query*), byłem nowym pracownikiem Microsoftu i musiałem się udać na konferencję PDC, gdzie LINQ został zaprezentowany. Choć nie brałem prawie żadnego udziału w pracach nad tą technologią, bardzo cieszył mnie rozgłos, jaki jej towarzyszył. Wszędzie o niej mówiono, a wydrukowane materiały na jej temat rozchodziły się jak świeże bułeczki. To był wielki dzień dla języka C# i platformy .NET, a ja świetnie się bawiłem.

Jednak w pomieszczeniach z komputerami, gdzie ludzie mogli wypróbować wersję wstępную technologii na podstawie skryptów opisujących kolejne kroki, było cicho. To tam natrafiłem na Marka. Nie muszę chyba mówić, że nie stosował się do skryptu. Przeprowadzał własne eksperymenty. Przekopywał się przez dokumentację, rozmawiał z innymi osobami i pracowicie tworzył własny obraz technologii.

Ponieważ byłem nowicjuszem w społeczności użytkowników języka C#, na tej konferencji poznałem zapewne wiele osób, z którymi potem nawiązałem dobre relacje. Jeśli jednak mam być szczerzy, nie pamiętam tego. Całą konferencję pamiętam jak przez mgłę. Jedyną osobą, którą zapamiętałem, był Mark. Dlaczego tak się stało? Gdy zapytałem go, czy podoba mu się nowa technologia, nie przyłączył się do zachwytów. Z dystansem stwierdził: *Jeszcze nie wiem. Na razie nie wyrobilem sobie opinii na jej temat.* Mark chciał przyswoić i zrozumieć cały pakiet narzędzi, dlatego nie zamierzał wcześniej pozwalać, by inni narzucali mu ocenę nowej technologii.

Tak więc zamiast usłyszeć przesłodzone zachwyty, których mogłem oczekiwąć, udało mi się odbyć szczerą i wartościową rozmowę (pierwszą z wielu, jakie przeprowadziliśmy przez lata) o szczegółach nowej technologii, konsekwencjach jej wprowadzenia i zastrzeżeniach wobec niej. Od tamtej pory Mark jest niezwykle cennym członkiem społeczności dla projektantów języków, ponieważ jest nadzwyczaj inteligentny, stara się dogłębnie zrozumieć wszystkie rozwiązania i ma fenomenalną intuicję pozwalającą mu przewidzieć, jak nowe technologie wpłyną na pracę programistów. Jednak prawdopodobnie największą zaletą Marka jest to, że jest szczerzy i nie boi się wyrażać swojego zdania. Jeśli jakieś rozwiązanie przejdzie „test Marka”, można z dużym prawdopodobieństwem podejrzewać, że się sprawdzi!

Wymienione cechy Marka sprawiają też, że jest świetnym autorem. Dociera do istoty sprawy i przekazuje informacje w uczciwy sposób, bez upiększania i z naciskiem na praktyczną wartość rozwiązań oraz rzeczywiste problemy. Mark ma wspaniały dar jasnego wyjaśniania problemów. Nikt nie pomoże Ci w zrozumieniu języka C# 8.0 jak on.

Życzę przyjemnej lektury!

— *Mads Torgersen*
Główny projektant języka C#
Microsoft



Wprowadzenie

WHISTORII INŻYNIERII oprogramowania metodyki stosowane do pisania programów komputerowych przeszły kilka zmian. W każdej nowej metodyce rozwijano poprzednią, by zwiększyć uporządkowanie kodu i zmniejszyć jego złożoność. Ta książka jest zbudowana w taki sposób, by zaprezentować Ci podobne zmiany paradymatów.

W początkowych rozdziałach książki zapoznasz się z **modelem programowania sekwenacyjnego**, w którym instrukcje są zapisywane w kolejności ich wykonywania. Problem z tym modelem polega na tym, że wraz ze wzrostem wymagań złożoność kodu rośnie wykładniczo. Aby ograniczyć złożoność, można przenieść bloki kodu do metod i zastosować w ten sposób **model programowania ustrukturyzowanego**. Pozwala on wywoływać ten sam blok kodu w różnych miejscach programu bez konieczności powielania kodu. Jednak nawet w tym modelu rosnące programy szybko mogą się stać chaotyczne i wymagać dodatkowej warstwy abstrakcji. Opisane w rozdziale 6. programowanie obiektowe było odpowiedzią na ten problem. W dalszych rozdziałach książki poznasz dodatkowe metodyki, na przykład programowanie oparte na interfejsach, technologię LINQ (i zmiany, jakie powoduje ona w interfejsie API kolekcji), a także podstawy programowania deklaratywnego z wykorzystaniem atrybutów (rozdział 18.).

Ta książka ma pełnić trzy podstawowe zadania. Oto one:

- Zapewnienie kompletnego omówienia języka C#. Ta książka ma być czymś więcej niż samouczkiem i ma dać podstawy pozwalające skutecznie rozpocząć prace nad projektami programistycznymi.
- Zapewnienie (dla osób znających już język C#) informacji o skomplikowanych paradymatach programowania i szczegółowego omówienia funkcji wprowadzonych w najnowszej wersji języka, C# 8.0, oraz w platformie .NET Framework 4.8/.NET Core 3.1.
- Pełnienie funkcji źródła wiedzy także dla osób, które potrafią biegły posługiwać się językiem C#.

Kluczem do udanego opanowania języka C# jest jak najszybsze rozpoczęcie pisania kodu. Nie odkładaj tego do momentu, w którym stanieś się „ekspertem” od teorii tego języka. Od razu rozpoczęj budowanie oprogramowania. Ponieważ jestem zwolennikiem programowania

iteracyjnego, mam nadzieję, że ta książka pozwoli nawet początkującym programistom rozpoczęć pisanie prostego kodu w języku C# już po lekturze rozdziału 2.

W tej książce pominięto niektóre zagadnienia. Nie znajdziesz tu omówienia technologii ASP.NET, Entity Framework, Xamarin, tworzenia inteligentnych klientów, programowania rozproszonego itd. Choć te tematy są powiązane z platformą .NET, zasługują na odrębne książki. Na szczęście seria *Microsoft Windows Development* wydawnictwa Addison-Wesley obejmuje wiele pozycji poświęconych tym zagadnieniom. W książce *C# 8.0. Kompletny przewodnik dla praktyków* skoncentrowano się na języku C# i typach z biblioteki Base Class. Lektura tej książki przygotuje Cię do skupienia się na obszarach opisanych w innych pozycjach ze wspomnianej serii i rozwinięcia wiedzy eksperckiej w tych dziedzinach.

Dla kogo przeznaczona jest ta książka?

Wyzwaniem w trakcie pisania tej książki było zainteresowanie nią doświadczonych programistów bez zniechęcania nowicjuszy stosowaniem takich pojęć jak *podzespół, łączenie, łańcuch, wątek* lub *fuzja*. Głównymi odbiorcami tej książki mają być doświadczeni programiści chcący dodać do arsenału znanych narzędzi nowy język — umieścić nową strzałę w swoim kołczanie. Ta książka została jednak starannie opracowana w taki sposób, by była wartościowa dla programistów o różnym poziomie zaawansowania.

- *Początkujący*. Jeśli dopiero uczysz się programować, ta książka pomoże Ci przejść drogę od początkującego programisty do dewelopera używającego języka C#. Będziesz wiedział, jak poradzić sobie z dowolnym zadaniem programistycznym z zakresu tego języka. Dzięki tej książce nie tylko poznasz składnię języka, ale też opanujesz dobre praktyki programistyczne, które przydadzą Ci się w trakcie kariery programisty.
- *Programiści stosujący model ustrukturyzowany*. Najlepszy sposób na opanowanie języka obcego polega na zanurzeniu się w środowisku, w którym jest on używany. Podobnie nauka języka programowania jest najbardziej skuteczna, jeśli zaczniesz go używać przed opanowaniem wszystkich jego zawiłości. Dlatego ta książka zaczyna się od samouczka, który jest łatwy do zrozumienia dla osób stosujących programowanie ustrukturyzowane. Do czasu zakończenia lektury rozdziału 5. osoby z tej grupy nie będą miały trudności z pisaniem prostych programów z instrukcjami związanymi z przepływem sterowania. Jednak aby dobrze opanować język C#, nie wystarczy zapamiętać składni. By przejść drogę od prostych programów do aplikacji dla przedsiębiorstw, programista języka C# musi zacząć myśleć w kategoriach obiektów i powiązań między nimi. Dlatego w rozdziale 6. w bloku „**ZAGADNIENIE DLA POCZĄTKUJĄCYCH**” omówiono klasy i programowanie obiektowe. Języki programowania strukturalnego, takie jak C, COBOL i FORTRAN, wciąż odgrywają ważną rolę, ale są stosowane coraz rzadziej. Dlatego inżynierom oprogramowania wypada opanować programowanie obiektowe. C# jest idealnym językiem do dokonania takiej zmiany, ponieważ z założenia zaprojektowano go jako język obiektowy.

- *Programiści używający języków obiektowych.* Do tej grupy należą programiści języków C++, Java, Python, TypeScript i Visual Basic. Liczne z tych osób są przyzwyczajone do posługiwania się średnikami i nawiasami klamrowymi. Krótkie omówienie kodu w rozdziale 1. pokazuje, że w swej istocie język C# jest podobny do znanych Ci już języków o stylu zbliżonym do C i C++.
- *Profesjonalni użytkownicy języka C#.* Dla osób biegłych w posługiwaniu się językiem C# ta książka stanowi wygodne źródło wiedzy na temat rzadziej stosowanych konstrukcji składniowych. Ponadto znajdziesz tu omówienie rzadko opisywanych szczegółów i subtelności języka. Co najważniejsze, książka zawiera wskazówki i wzorce pomagające w pisaniu niezawodnego i łatwego w konserwacji kodu. Ta pozycja pomaga też w nauczaniu języka C# innych osób. Oto najważniejsze usprawnienia, jakie pojawiły się od wersji 3.0 do wersji 8.0 języka C#:
 - interpolacja łańcuchów znaków (zobacz rozdział 2.),
 - zmienne o niejawnie określonym typie (zobacz rozdział 2.),
 - krotki (zobacz rozdział 3.),
 - typy referencyjne dopuszczające wartości null (zobacz rozdział 3.),
 - dopasowywanie do wzorców (zobacz rozdział 4.),
 - metody rozszerzające (zobacz rozdział 6.),
 - metody częściowe (zobacz rozdział 6.),
 - składowe domyślne w interfejsach (zobacz rozdział 8.),
 - typy anonimowe (zobacz rozdział 12.),
 - typy generyczne (zobacz rozdział 12.),
 - instrukcje i wyrażenia lambda (zobacz rozdział 13.),
 - drzewa wyrażeń (zobacz rozdział 13.),
 - standardowe operatory kwerend (zobacz rozdział 15.),
 - wyrażenia kwerend (zobacz rozdział 16.),
 - programowanie dynamiczne (zobacz rozdział 18.),
 - tworzenie programów wielowątkowych za pomocą biblioteki Task Programming i instrukcji async (zobacz rozdział 20.),
 - równoległe przetwarzanie kwerend z wykorzystaniem technologii PLINQ (zobacz rozdział 21.),
 - kolekcje współbieżne (zobacz rozdział 22.).

Te zagadnienia są opisane szczegółowo na potrzeby programistów, którzy jeszcze ich nie znają. Z zaawansowanym programowaniem w języku C# związane są też wskaźniki (ich omówienie zawiera rozdział 23.). Nawet wielu doświadczonych programistów języka C# nie rozumie w pełni tego zagadnienia.

Cechy tej książki

C# 8.0. *Kompletny przewodnik dla praktyków* to książka o języku programowania oparta na podstawowej specyfikacji C#. Aby pomóc Ci w zrozumieniu różnych konstrukcji języka C#, przedstawiono tu liczne przykłady ilustrujące poszczególne funkcje. Do każdego zagadnienia dołączone są wskazówki i najlepsze praktyki, które pomogą zapewnić, że kod się skompiluje, uniknąć pułapek i maksymalnie ułatwić konserwację kodu.

W celu ułatwienia lektury kod został sformatowany w specjalny sposób, a streszczenia rozdziałów przedstawiono w postaci map myśli.

Wskazówki dotyczące pisania kodu w języku C#

Jedną z ważnych cech książki C# 8.0. *Kompletny przewodnik dla praktyków* jest obecność wskazówek dotyczących pisania kodu w języku C#. Poniżej pokazana jest przykładowa wskazówka z rozdziału 17.

Wskazówki

UPEWNIJ SIĘ, że równe sobie obiekty mają takie same skróty.

UPEWNIJ SIĘ, że skrót obiektu nigdy się nie zmienia, gdy jest przechowywany w tablicy z haszowaniem.

UPEWNIJ SIĘ, że algorytm haszowania szybko generuje skróty o równomiernym rozkładzie.

UPEWNIJ SIĘ, że algorytm haszowania działa poprawnie dla obiektu będącego w dowolnym stanie.

Te wskazówki są bardzo ważne, ponieważ stosowanie się do nich pozwala odróżnić programistę, który tylko zna składnię, od eksperta potrafiącego pisać maksymalnie efektywny kod dostosowany do okoliczności. Taki ekspert nie tylko tworzy kod, który się kompliluje, ale przestrzega przy tym najlepszych praktyk minimalizujących liczbę błędów i ułatwiających konserwację kodu w przyszłości. We wskazówkach dotyczących pisania kodu przedstawione są niektóre z najważniejszych zasad, których Czytelnicy powinni koniecznie przestrzegać w trakcie programowania.

Przykładowy kod

Fragmenty kodu prezentowane w większości rozdziałów mogą działać w większości implementacji platformy CLI (ang. *Common Language Infrastructure*), jednak tu koncentrujemy się na implementacjach Microsoft .NET Framework i .NET Core. Biblioteki specyficzne dla platformy lub producenta są stosowane rzadko, chyba że pozwalają przedstawić ważne zagadnienia dotyczące tylko wybranych technologii (np. obsługę jednowątkowego interfejsu użytkownika w systemie Windows). Kod wymagający zgodności z wersjami 5.0, 6.0, 7.0 lub 8.0 języka C# jest oznaczony w odrębnych indeksach w końcowej części książki.

Poniżej pokazano przykładowy listing.

Listing 1.19. Komentarze w kodzie

```
class Comment Samples
{
    static void Main()
    {
        string firstName; // Zmienna do przechowywania imienia.
        string lastName; // Zmienna do przechowywania nazwiska.

        System.Console.WriteLine("Hej, ty!");

        System.Console.Write /* Bez nowego wiersza. */(
            "Podaj imię: ");
        firstName = System.Console.ReadLine();

        System.Console.Write /* Bez nowego wiersza. */(
            "Podaj nazwisko: ");
        lastName = System.Console.ReadLine();

        /* Wyświetlanie pozdrowienia w konsoli
           z wykorzystaniem formatowania złożonego. */ }

        System.Console.WriteLine("Twoje imię i nazwisko to {0} {1}.",
            firstName, lastName);
        // To jest koniec
        // listingu.
    }
}
```

Formatowanie stosowane w kodzie opisano poniżej.

- Komentarze są wyróżnione kursywą:

```
/* Wyświetlanie pozdrowienia w konsoli
   z wykorzystaniem formatowania złożonego. */
```

- Słowa kluczowe są wyróżnione pogrubieniem.

```
static void Main()
```

- Wyróżnione są fragmenty kodu, które zmieniły się w porównaniu z wcześniejszym listingiem lub ilustrują zagadnienie opisane w tekście:

```
System.Console.WriteLine(valerie);
miracleMax = "Potrzebny byłby cud.";
System.Console.Write(miracleMax);
```

Tego typu wyróżnienie może dotyczyć całego wiersza lub tylko wybranych znaków:

```
System.Console.WriteLine(
    $"Palindrom \"{palindrome}\" zawiera"
    + $" {palindrome.Length} znaków.");
```

- Niekompletne listingi są reprezentowane za pomocą wielokropka, który oznacza pominięty nieistotny kod.
// ...
- Dane wyjściowe z konsoli to informacje generowane przez kod z listingów. Takie dane są prezentowane po listingach. Dane wejściowe od użytkownika są wyróżnione **pogrubieniem**.

DANE WYJŚCIOWE 1.7

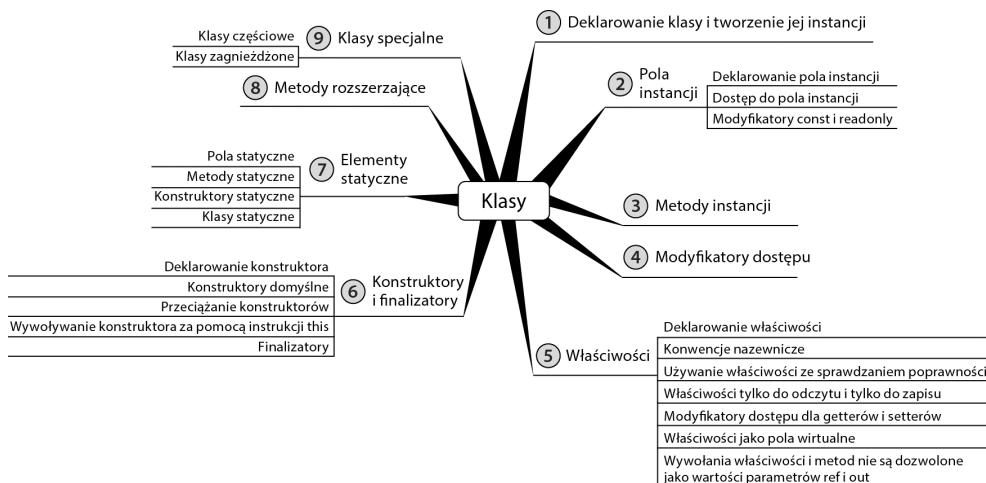
```
Hej, ty!
Podaj imię: Adam
Podaj nazwisko: Kowalski
Twoje imię i nazwisko to Adam Kowalski.
```

Choć przedstawianie kompletnych przykładów, które mógłbyś skopiować do własnych programów, byłoby wygodne, odciągałoby uwagę od nauki omawianych zagadnień. Dlatego musisz zmodyfikować przykładowy kod przed wykorzystaniem go w swoich programach. Najważniejszym pominiętym aspektem jest kod wykrywający błędy, na przykład kod do obsługi wyjątków. Ponadto w przykładowym kodzie nie są jawnie używane instrukcje `using System`. Powinieneś przyjąć, że występują one we wszystkich przykładach.

Przykładowy kod znajdziesz na stronie <http://intellitect.com/essentialcsharp>. Spolszczone wersja kodu jest dostępna na stronie poświęconej książce w witrynie wydawnictwa Helion.

Mapy myśli

We wprowadzeniu do każdego rozdziału przedstawiona jest **mapa myśli**. Jest ona streszczeniem pozwalającym szybko zapoznać się z zawartością każdego rozdziału. Poniżej przykładowa mapa myśli z rozdziału 6.



Temat przewodni każdego rozdziału jest zapisany w środku mapy myśli. Ogólne zagadnienia rozgałęzają się od środka. Mapy myśli pozwalają łatwo dostrzec przejście od ogólnych do konkretnych tematów, dzięki czemu z mniejszym prawdopodobieństwem natrafisz na bardzo szczegółowe informacje, które Cię nie interesują.

Pomocne uwagi

Specjalne bloki kodu i oznaczenia pomagają poruszać się po tekście i wyszukiwać informacje dostosowane do osób o różnym poziomie doświadczenia.

- Bloki „ZAGADNIENIE DLA POCZĄTKUJĄCYCH” zawierają definicje i wyjaśnienia skierowane do początkujących programistów.
- Bloki „ZAGADNIENIE DLA ZAAWANSOWANYCH” pozwalają doświadczonym programistom skoncentrować się na najbardziej istotnych dla nich materiałach.
- W wyróżnionych uwagach opisano najważniejsze zasady, dzięki którym Czytelnicy mogą łatwo zrozumieć ich znaczenie.
- W ramkach Porównanie języków opisane są najważniejsze różnice między językiem C# i jego poprzednikami, co jest pomocne dla użytkowników innych języków.

Struktura książki

Na ogólnym poziomie inżynieria oprogramowania polega na zarządzaniu złożonością. Książka *C# 8.0. Kompletny przewodnik dla praktyków* też jest uporządkowana z myślą o tym celu. Rozdziały od 1. do 5. zawierają wprowadzenie do programowania strukturalnego, co pozwala natychmiast rozpocząć pisanie prostego działającego kodu. Rozdziały od 6. do 10. dotyczą konstrukcji obiektowych stosowanych w języku C#. Początkujący powinni dobrze zrozumieć te rozdziały przed przejściem do bardziej zaawansowanych zagadnień omawianych w dalszej części książki. W rozdziałach od 12. do 14. opisano dodatkowe konstrukcje pomagające zmniejszyć złożoność. Znajdziesz tu omówienie standardowych wzorców stosowanych w prawie wszystkich współczesnych programach. Dalsze rozdziały dotyczą programowania dynamicznego z wykorzystaniem mechanizmu refleksji i atrybutów. Te techniki są powszechnie wykorzystywane w programowaniu wielowątkowym i do obsługi współdziałania komponentów, co opisano w kolejnych rozdziałach.

Książka kończy się rozdziałem 24. poświęconym platformie CLI. Opisano w nim język C# w kontekście platformy rozwoju aplikacji, w ramach której działa. Ten rozdział znajduje się na końcu, ponieważ dotyczy nie tylko języka C# i stanowi odejście od składni i stylu programowania wykorzystywanych w innych miejscach książki. Możesz jednak zapoznać się z tym rozdziałem w dowolnym momencie. Prawdopodobnie najlepiej będzie to zrobić bezpośrednio po lekturze rozdziału 1.

Poniżej znajdziesz opis każdego rozdziału. Na tej liście numery zapisane **pogrubioną kursywą** oznaczają rozdziały z informacjami na temat wersji 7.0 – 8.0 języka C#.

- **Rozdział 1.** „Wprowadzenie do języka C#”. Na początku przedstawiony jest program HelloWord w języku C#, a dalej znajdziesz analizę tej aplikacji. Dzięki temu powinieneś się zaznajomić z wyglądem i stylem programów w języku C#. W rozdziale opisano też szczegółowo kompilowanie i debugowanie programów. Ponadto pokrótko omówiono kontekst wykonywania programów w języku C# oraz język pośredni.
- **Rozdział 2.** „Typy danych”. Programy w trakcie działania manipulują danymi. W tym rozdziale przedstawiono proste typy danych z języka C#.
- **Rozdział 3.** „Jeszcze o typach danych”. W tym rozdziale znajdziesz opis dwóch kategorii typów — bezpośrednich i referencyjnych. Omówiono tu też zmienne lokalne z niejawnie określonym typem danych, krotki, modyfikator nullable i wprowadzone w wersji C# 8.0 typy referencyjne dopuszczające wartość null. Ten rozdział kończy się szczegółowym objaśnieniem struktury prostych tablic.
- **Rozdział 4.** „Operatory i przepływ sterowania”. Aby wykorzystać możliwości komputera w zakresie iteracji, musisz wiedzieć, jak dodawać do programów pętle i logikę warunkową. Ten rozdział zawiera też omówienie operatorów języka C#, konwersji danych i dyrektyw preprocessora.
- **Rozdział 5.** „Metody i parametry”. W tym rozdziale szczegółowo opisano metody i ich parametry. Omówiono tu przekazywanie parametrów przez wartość, przekazywanie parametrów przez referencję i zwracanie danych za pomocą parametrów. W wersji C# 4.0 dodano obsługę parametrów domyślnych, a w tym rozdziale wyjaśniono, jak wykorzystać ten mechanizm.
- **Rozdział 6.** „Klasy”. W tym rozdziale pokazano, jak za pomocą podstawowych cegiełek służących do budowania klas tworzyć w pełni funkcjonalne typy. Klasy są podstawą technologii obiektowej, ponieważ pozwalają zdefiniować szablon obiektów. W tym rozdziale opisano też wprowadzone w wersji C# 8.0 atrybuty dopuszczające wartość null.
- **Rozdział 7.** „Dziedziczenie”. Choć dziedziczenie jest dla wielu programistów jedną z podstawowych technik, w języku C# dostępne są pewne wyjątkowe konstrukcje, na przykład modyfikator new. W tym rozdziale opisano szczegółowo składnię dziedziczenia, w tym przesłanianie.
- **Rozdział 8.** „Interfejsy”. W tym rozdziale pokazano, jak wykorzystać interfejsy do definiowania przyjmującego różne wersje kontraktu opisującego interakcje między klasami. Język C# umożliwia jawnie i niejawnie implementowanie składowych interfejsu, co pozwala zapewnić dodatkowy poziom hermetyzacji, niedostępny w większości innych języków. Ponieważ w C# 8.0 wprowadzono składowe domyślne interfejsów, znajdziesz tu też nowy podrozdział poświęcony zarządzaniu wersjami interfejsu w tym wydaniu języka.

- **Rozdział 9.** „Typy bezpośrednie”. Choć częściej definiowane są typy referencyjne, czasem trzeba zastosować typ bezpośredni, działający podobnie jak typy proste wbudowane w język C#. W tym rozdziale opisano, jak definiować struktury, a także wyjaśniono związane z nimi osobliwości.
- **Rozdział 10.** „Dobrze uformowane typy”. W tym rozdziale omówiono zaawansowane definicje typów. Wyjaśniono, jak implementować operatory (na przykład operator + i operator rzutowania), a także jak ukryć grupę klas w jednej bibliotece. Ponadto pokazano tu proces definiowania przestrzeni nazw, stosowanie komentarzy w formacie XML i projektowanie klas z myślą o przywracaniu pamięci.
- **Rozdział 11.** „Obsługa wyjątków”. Ten rozdział jest rozwinięciem wprowadzenia do obsługi wyjątków przedstawionego w rozdziale 5. Tu opisano, że wyjątki mają określona hierarchię wspomagającą tworzenie wyjątków niestandardowych. W rozdziale znajdziesz też omówienie najlepszych praktyk z zakresu obsługi wyjątków.
- **Rozdział 12.** „Typy generyczne”. Typy generyczne to prawdopodobnie najważniejszy mechanizm, którego zabrakło w wersji 1.0 języka C#. W tym rozdziale dokładnie omówiono tę wprowadzoną w wersji 2.0 funkcję. W C# 4.0 dodano obsługę kowariancji i kontrawariancji. W tym rozdziale oba te mechanizmy opisano w kontekście typów generycznych.
- **Rozdział 13.** „Delegaty i wyrażenia lambda”. Delegaty są cechą odróżniającą język C# od jego poprzedników, ponieważ pozwalają definiować wzorce określające obsługę zdarzeń w kodzie. Ta technika niemal całkowicie eliminuje konieczność pisania procedur z mechanizmem odpytywania. Wyrażenia lambda to najważniejszy mechanizm, który pozwolił dodać technologię LINQ w wersji C# 3.0. W rozdziale 13. wyjaśniono, że wyrażenia lambda są oparte na delegatach i umożliwiają stosowanie bardziej eleganckiej oraz zwięzkiej składni. Ten rozdział jest wprowadzeniem do omówienia przedstawionego dalej nowego interfejsu API dla kolekcji.
- **Rozdział 14.** „Zdarzenia”. Zdarzenia (czyli delegaty oddane hermetyzacji) są podstawową konstrukcją środowiska CLR (ang. *Common Language Runtime*). W tym rozdziale opisano też metody anonimowe (jest to następna funkcja wprowadzona w wersji C# 2.0).
- **Rozdział 15.** „Interfejsy kolekcji ze standardowymi operatorami kwerend”. W tym rozdziale widoczna staje się wartość prostych, ale dających dużo możliwości zmian wprowadzonych w wersji C# 3.0. Omówiono tu metody rozszerzające nowej klasy Enumerable. Ta klasa pozwoliła utworzyć szczegółowo opisany w tym miejscu zupełnie nowy interfejs API kolekcji ze standardowymi operatorami kwerend.
- **Rozdział 16.** „Technologia LINQ i wyrażenia z kwerendami”. Stosowanie tylko standardowych operatorów kwerend prowadzi do powstawania długich instrukcji, które często są trudne do zrozumienia. Wyrażenia kwerend zapewniają alternatywną składnię, która ściśle odpowiada składni SQL-a, co opisano w tym rozdziale.

- **Rozdział 17.** „Tworzenie niestandardowych kolekcji”. W trakcie tworzenia niestandardowych interfejsów API dla obiektów biznesowych czasem trzeba zbudować także niestandardowe kolekcje. W tym rozdziale szczegółowo opisano, jak to zrobić, a także omówiono kontekstowe słowa kluczowe ułatwiające tworzenie niestandardowych kolekcji.
- **Rozdział 18.** „Refleksja, atrybuty i programowanie dynamiczne”. Programowanie obiektowe doprowadziło pod koniec lat 80. do zmiany paradygmatu tworzenia struktury programów. Z kolei atrybuty ułatwiają programowanie deklaratywne i dodawanie metadanych, co prowadzi do powstania nowego paradygmatu. W tym rozdziale opisano atrybuty i wyjaśniono, jak pobierać je za pomocą mechanizmu refleksji. Omówiono też plikowe dane wejściowe i wyjściowe obsługiwane przy użyciu platformy serializacji (będącej częścią biblioteki Base Class). W wersji C# 4.0 do języka dodano nowe słowo kluczowe — `dynamic`. Pozwala ono odroczyć sprawdzanie typów do momentu wykonywania programu, co jest istotnym rozwinięciem możliwości języka C#.
- **Rozdział 19.** „Wielowątkowość”. Większość nowych programów wymaga zastosowania wątków, co pozwala jednocześnie wykonywać długie zadania i aktywnie reagować na zdarzenia. Ponieważ programy stają się coraz bardziej złożone, trzeba stosować dodatkowe środki ostrożności, by chronić dane w dynamicznych środowiskach. Pisanie aplikacji wielowątkowych jest skomplikowane. W tym rozdziale opisano, jak pracować z wątkami, a także przedstawiono najlepsze praktyki pozwalające uniknąć problemów często występujących w aplikacjach wielowątkowych.
- **Rozdział 20.** „Wzorzec asynchroniczny oparty na zadaniach”. W tym rozdziale omówiono wzorzec asynchroniczny oparty na zadaniach wraz z powiązaną z nim składnią `async/await`. Ten wzorzec pozwala znacznie uproszczyć programowanie wielowątkowe. Opisane są tu też wprowadzone w C# 8.0 strumienie asynchroniczne.
- **Rozdział 21.** „Iteracja równoległa”. Łatwym sposobem na poprawę wydajności jest równoległe iteracyjne przetwarzanie danych za pomocą obiektu `Parallel` lub biblioteki `Parallel LINQ`.
- **Rozdział 22.** „Synchronizowanie wątków”. W tym rozdziale wykorzystano informacje z wcześniejszych rozdziałów i opisano wbudowaną obsługę wzorców wielowątkowych, która pozwala uproszczyć bezpośrednie kontrolowanie wątków w kodzie aplikacji wielowątkowych.
- **Rozdział 23.** „Współdziałyanie między platformami i niezabezpieczony kod”. Ponieważ C# to stosunkowo młody język, znacznie więcej dostępnego kodu napisano w innych językach. Aby umożliwić wykorzystanie istniejącego kodu, w języku C# dodano obsługę współdziałyania (wywoływanie niezarządzanego kodu) za pomocą mechanizm `P/Invoke`. Ponadto C# umożliwia stosowanie wskaźników i bezpośrednie manipulowanie pamięcią. Choć wykonywanie kodu obejmującego wskaźniki wymaga specjalnych uprawnień, dzięki wskaźnikom możliwe jest pełne współdziałyanie z tradycyjnymi interfejsami programowania opartymi na języku C.

- *Rozdział 24. „Standard CLI”.* Język C# to składnia zaprojektowana po to, by utworzyć najskuteczniejszy język programowania oparty na platformie Common Language Infrastructure (CLI). W tym rozdziale opisano, jak programy w języku C# są powiązane z używanym przez nie środowiskiem uruchomieniowym i jego specyfikacją.
- *Indeksy z funkcjami z wersji C# 6.0, 7.0 i 8.0.* Te indeksy to krótki przegląd funkcji dodawanych w wersjach od 6.0 do 8.0 języka C#. Indeksy mają pomóc programistom szybko zaktualizować wiedzę o języku pod kątem jego nowszych wersji.

Mam nadzieję, że ta książka będzie wartościowym źródłem, które pomoże Ci zdobyć wiedzę ekspercką z zakresu języka C#. Liczę też na to, że będziesz szukać w niej informacji o rzadziej stosowanych przez Ciebie technikach jeszcze długo po tym, jak zyskasz biegłość w tym języku.

— *Mark Michaelis*
Blog: <http://intellitect.com/mark>
Twitter: @Intellitect, @MarkMichaelis



O autorze

Mark Michaelis jest założycielem firmy IntelliTect, zajmującej się projektowaniem i tworzeniem zaawansowanego oprogramowania. Jest tam głównym architektem i trenerem. Mark kieruje swoją odnoszącą sukcesy firmą, a przy tym prowadzi na całym świecie wykłady na konferencjach poświęconych kierowaniu zespołami i technologiom. Prowadzi też prelekcje dla Microsoftu i innych klientów. Ponadto jest autorem wielu artykułów i książek, wykładowcą na Easter Washington University, założycielem grupy Spokane .NET oraz współorganizatorem dorocznej imprezy TEDx Coeur d'Alene.

Mark jest światowej klasy ekspertem od języka C#, od 2007 r. posiada tytuł Microsoft Regional Director, a od ponad 25 lat — tytuł Microsoft MVP.

Mark otrzymał tytuł licencjata z zakresu filozofii na Uniwersytecie Illinois oraz tytuł magistra w dziedzinie nauk komputerowych w Instytucie Technologii Illinois.

Gdy nie siedzi przy komputerze, zajmuje się rodziną lub gra w racquetball (odkąd w 2016 r. zaprzestał uczestniczenia w zawodach Ironman). Mark mieszka w Spokane w stanie Waszyngton z żoną, Elisabeth, i trójką dzieci, Benjaminem, Hanną i Abigail.

O redaktorach technicznych

Kevin Bost posiada tytuł Microsoft MVP oraz jest starszym architektem oprogramowania w firmie IntelliTect. Odegrał ważną rolę w opracowaniu szeregu innowacyjnych produktów takich jak System.CommandLine, Moq.AutoMocker i ShowMeTheXAML. Gdy nie pracuje, pomaga w rozwoju innym programistom za pośrednictwem serwisu YouTube (kanał youtube.keboo.dev) i poprawia popularny zestaw narzędzi Material Design in XAML (<http://materialdesigninxaml.net/>). Lubi też gry planszowe, ultimate frisbee i jazdę motocyklem.

Eric Lippert pracuje nad narzędziami dla programistów w korporacji Facebook. Wcześniej należał do zespołu projektującego język C# w Microsoftie. Gdy nie odpowiada na pytania na temat języka C# w serwisie StackOverflow i nie redaguje książek dla programistów, stara się utrzymywać w dobrym stanie swoją małą żaglówkę. Eric mieszka w Seattle w stanie Waszyngton razem ze swoją żoną, Leah.



Podziękowania

ŻADNA KSIĄŻKA NIE MOŻE zostać opublikowana bez pomocy innych, dlatego jestem niezwykle wdzięczny wielu osobom, które pomogły mi w pracach nad tą pozycją. Kolejność, w jakiej wymieniam tych ludzi, nie ma znaczenia; wyjątkiem są osoby, którym dziękuję na początku. Ponieważ jest to już siódme wydanie tej książki, możecie sobie wyobrazić, na jakie poświęcenie zdobyła się moja rodzina, aby pozwolić mi pisać przez ostatnich 14 lat (nie wspominam tu nawet o wcześniejszych pozycjach). Benjamin, Hanna i Abigail często musieli znaść to, że ich tata był rozkojarzony pracą nad książką, ale na jeszcze większe kłopoty narażona była Elisabeth, która często musiała sama zajmować się wszystkim i samodzielnie dbać o rodzinę. W trakcie wakacji w 2017 r. przeznaczałem dnie na pisanie w pokoju, podczas gdy moi bliscy zdecydowanie woleliby spędzać czas na plaży.

Pisanie C# 8.0. *Kompletny przewodnik dla praktyków* wyglądało trochę inaczej, ponieważ zamiast poświęcać życie rodzinne, poważnie zaniedbałem swoją pracę. Jestem głęboko wdzięczny, że otacza mnie zespół tak fantastycznych inżynierów oprogramowania, którzy doskonale radzili sobie bez mojej pomocy. A jakby i tego było mało, kilku z nich pomagało mi przy rozmaitych drobiazgach: od erraty, przez proces DevOps i numerowanie listingów, aż po poprawki techniczne. Specjalne podziękowania należą się Cameron Osborn, Philowi Spokasowi (który pomagał przy pisaniu fragmentów rozdziału 24.), Andresowi Scottowi i, w ostatnim czasie, Austenowi Frostadowi.

Z Kevinem Bostem pracuję w IntelliTect już od 2013 r. Mimo to Kevin nadal zaskakuje mnie wybitnymi umiejętnościami z zakresu tworzenia oprogramowania. Nie tylko posiada fenomenalną wiedzę na temat języka C#, ale jest też najwyższej klasy ekspertem od wielu technologii pomocniczych. Z tych i innych powodów tym razem poprosiłem Kevinę Bosta o to, by został oficjalnym redaktorem technicznym tej książki. Bardzo się cieszę, że to zrobił. Kevin przekazał mi uwagi i usprawnienia dotyczące materiałów znajdujących się w tekście już od początkowych wydań, o czym nikt wcześniej nie pomyślał. Ta szczegółowość w połączeniu z niestrudzonym dążeniem do doskonałości sprawiły, że C# 8.0. *Kompletny przewodnik dla praktyków* jest fundamentalną pozycją dla osób zainteresowanym językiem C#.

Oczywiście Eric Lippert też jest niesamowity. Poziom, w jakim opanował język C#, jest zdumiewający. Bardzo doceniam jego poprawki, zwłaszcza te związane z dążeniem do perfekcji w obszarze terminologii. Poprawki Erica dotyczące rozdziałów poświęconych wersji C# 3.0 były niezwykle istotne. Żałowałem tylko tego, że w trakcie prac nad drugim wydaniem książki nie poprosiłemErica o sprawdzenie wszystkich rozdziałów. Jednak obecnie moje ubolewania są już nieaktualne, ponieważ Eric skrupulatnie poprawił każdy rozdział wydania poświęconego wersji C# 4.0, a nawet został współautorem wydań dotyczących wersji C# 5.0 i C# 6.0. Jestem mu niezwykle wdzięczny za pracę redaktora technicznego książki C# 8.0. *Kompletny przewodnik dla praktyków*. Dzięki, Eric! Nikt lepiej niż Ty nie nadawałby się do tej pracy. Dzięki Tobie ta książka z dobrej stała się świetna.

Podobnie jak mało kto może się równać z Erikiem, jeśli chodzi o znajomość języka C#, tak niewiele jest osób znających się na wielowątkowości w platformie .NET równie dobrze jak Stephen Toub. Dlatego Stephen skoncentrował się na dwóch (napisanych po raz trzeci) rozdziałach dotyczących wielowątkowości i na obsłudze asynchroniczności wprowadzonej w wersji C# 5.0. Dzięki, Stephen!

Przez lata wielu redaktorów technicznych szczegółowo sprawdzało każdy rozdział, by zapewnić jego poprawność merytoryczną. Często byłem zaskoczony tym, jak subtelne błędy redaktorzy potrafili wykryć. Oto lista tych osób: Paul Bramsman, Kody Brown, Andrew Comb, Ian Davis, Doug Dechow, Gerard Frantz, Dan Haley, Thomas Heavey, Anson Horton, Brian Jones, Shane Kercheval, Angelika Langer, Neal Lundby, John Michaelis, Jason Morse, Nicholas Paldino, Jason Peterson, Jon Skeet, Michael Stokesbary, Robert Stokesbary i John Timney.

Dziękuję też wszystkim osobom z wydawnictwa Pearson/Addison-Wesley za to, że wykazali się cierpliwością w trakcie współpracy ze mną, choć czasem koncentrowałem się na wszystkim oprócz pisania książki. Dziękuję Chrisowi Zahnowi za sformatowanie tekstu i zapewnienie jego czytelności. Podziękowania dla Jill Hobbs! Podziwiłam ludzi takich jak Ty i Twoją dbałość o szczegóły oraz wiedzę z zakresu języka angielskiego. Dziękuję zespołowi produkcyjnemu odpowiedzialnemu za układ tekstu: Robowi Mauharowi i Violi Jasko. Wasze wyjątkowe umiejętności sprawiły, że moje uwagi ograniczały się do fragmentów, w których sam popełniłem błędy w tekście. Jestem wdzięczny Rachel Paul za jej pytania, gdy natrafiła na niewłaściwe fragmenty tekstu lub tytuły, a także za zarządzanie różnymi kwestiami w pracach toczących się na zapleczu. Dziękuję Malobice Chakraborty za pomoc w całym procesie prac: od propozycji napisania książki po etap produkcyjny.

1

Wprowadzenie do języka C#

JĘZYK C# MOŻNA STOSOWAĆ DO rozwoju komponentów oprogramowania i aplikacji działających w rozmaitych systemach operacyjnych (platformach), w tym: aplikacji sieciowych, mikrousług, aplikacji desktopowych oraz rozwiązań działających w urządzeniach przenośnych, konsolach do gier i internecie rzeczy. Ponadto C# jest bezpłatny. Co więcej, jest to język otwarty, dlatego możesz przeglądać jego kod, modyfikować go, rozpowszechniać i udostępniać wprowadzone poprawki. C# jest językiem opartym na mechanizmach dostępnych w poprzedzających go językach w stylu języka C (takich jak C, C++ i Java). Dzięki temu wygląda znajomo dla wielu doświadczonych programistów¹.



W tym rozdziale język C# jest przedstawiony na przykładzie tradycyjnego programu HelloWorld. Skupiono się tu na podstawach składni języka C#, w tym na definiowaniu punktu wejścia do programów w tym języku. Dzięki temu zapoznasz się ze stylem i strukturą składni tego języka oraz będziesz potrafił napisać w nim najprostsze programy. Przed omówieniem podstaw składni języka znajdziesz krótki opis zarządzanego środowiska wykonania. Z tego opisu dowiesz się, jak wykonywane są programy w języku C#. Rozdział kończy się omówieniem deklarowania zmiennych, podawania i pobierania danych w konsoli oraz dodawania komentarzy do kodu w języku C#.

¹ Pierwsze spotkanie poświęcone projektowi języka C# miało miejsce w 1998 roku.

Witaj, świecie

Najlepszy sposób na poznanie nowego języka programowania polega na pisaniu kodu. Pierwszym przykładem będzie tu klasyczny program `HelloWorld`. W tym programie zobacysz, jak wyświetlać tekst na ekranie.

Listing 1.1 przedstawia kompletny kod programu `HelloWorld`. Dalej dowiesz się, jak skompilować i uruchomić ten kod.

Listing 1.1. Program `HelloWorld` w języku C²

```
class HelloWorld
{
    static void Main()
    {
        System.Console.WriteLine("Witaj. Nazywam się Inigo Montoya.");
    }
}
```

Uwaga

W języku C# wielkość liter ma znaczenie. Błędna wielkość liter może spowodować, że kod się nie skompiluje.

Programiści mający doświadczenie w korzystaniu z języków Java, C lub C++ natychmiast dostrzegą podobieństwa. Podstawowa składnia języka C#, podobnie jak Javy, jest oparta na składni języków C i C++³. Składniowe „znaki przestankowe” (na przykład średniki i nawiasy klamrowe), mechanizmy (takie jak istotna wielkość liter) i słowa kluczowe (na przykład `class`, `public` i `void`) są znane programistom używającym wymienionych języków. Dla początkujących i użytkowników innych języków konstrukcje z języka C# szybko staną się intuicyjnie zrozumiałe.

Tworzenie, edytowanie, komplikowanie i uruchamianie kodu źródłowego w języku C#

Po napisaniu kodu w języku C# należy go skompilować i uruchomić. Możesz wybrać, której implementacji środowiska .NET chcesz używać; czasem są one nazywane **platformami .NET**. Zwykle takie implementacje mają postać **pakietu SDK** (ang. *Software Development Kit*). Taki pakiet obejmuje kompilator, silnik wykonawczy, platformę funkcji wygodnie dostępnych w środowisku uruchomieniowym (zobacz punkt „Interfejsy API” w dalszej części rozdziału)

² Jeśli dziwi Cię występujące tu imię i nazwisko Inigo Montoya, obejrzyj film *Narzeczona dla księcia*.

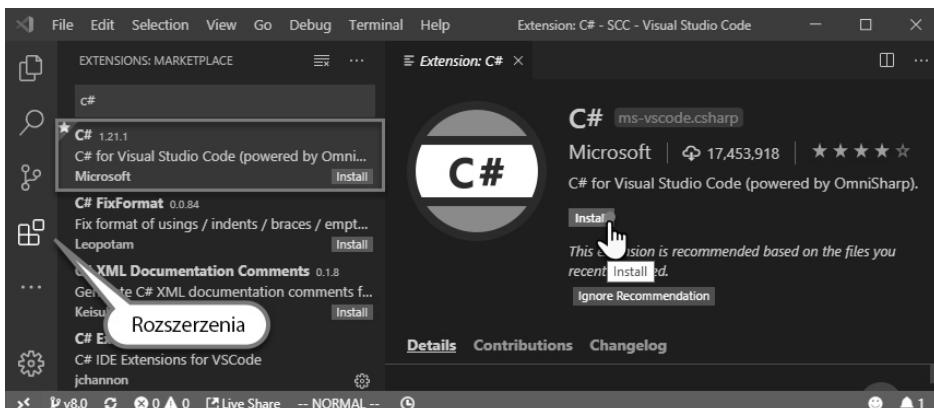
³ W trakcie tworzenia języka C# jego autorzy przeanalizowali specyfikacje języków C oraz C++ i dosłownie wykreślili funkcje, które im się nie podobały, oraz opracowali listę odpowiadających im rozwiązań. W grupie autorów języka znajdowali się też projektanci o dużym doświadczeniu w korzystaniu z innych języków.

i dodatkowe narzędzia (np. system do automatyzowania etapów budowania kodu). Ponieważ język C# jest publicznie dostępny od 2000 r., istnieje kilka różnych platform .NET do wyboru (zobacz punkt „Różne wersje platformy .NET”).

Instrukcje instalacji platformy .NET zależą od systemu operacyjnego, na który piszesz kod, i używanej wersji takiej platformy. Dlatego zachęcam do zapoznania się z instrukcjami pobierania i instalowania platformy na stronie <https://dotnet.microsoft.com/download>. Najpierw wybierz platformę .NET, a następnie pobierany pakiet na podstawie docelowego systemu operacyjnego. Choć mogłbym opisać tu więcej szczegółów, witryna służąca do pobierania platformy .NET zawiera najbardziej aktualne informacje dla każdej obsługiwanej kombinacji narzędzi.

Jeśli nie masz pewności, której platformy .NET użyć, domyślnie wybierz wersję .NET Core. Działa ona w systemach Linux, macOS i Microsoft Windows. To ta implementacja jest najintensywniej rozwijana przez zespoły odpowiedzialne za środowisko .NET. Ponadto w tej książce preferuję instrukcje z wersji .NET Core, ponieważ działa ona w różnych systemach.

Kod źródłowy można pisać na wiele sposobów, także używając najprostszych narzędzi takich jak Notatnik w systemie Windows,TextEdit w systemie Mac/macOS lub vi w systemie Linux. Jednak prawdopodobnie zechcesz korzystać z bardziej zaawansowanych rozwiązań, oferujących przynajmniej kolorowanie składni. Wystarczający będzie każdy edytor dla programistów obsługujący język C#. Jeśli jeszcze nie masz ulubionych narzędzi, zachęcam do rozważenia otwartego edytora Visual Studio Code (<https://code.visualstudio.com>). Aby zoptymalizować pracę z językiem C# w tym edytorze, warto zainstalować rozszerzenie dla tego języka pokazane na rysunku 1.1. Jeżeli korzystasz z systemów Windows lub macOS, rozważ użycie środowiska Microsoft Visual Studio 2019 (albo nowszą wersję; zobacz <https://www.visualstudio.com/vs/>). Oba te narzędzia są dostępne bezpłatnie.



Rysunek 1.1. Instalowanie rozszerzenia dla języka C# w edytorze Visual Studio Code

W dwóch kolejnych punktach przedstawione są instrukcje dotyczące obu wymienionych edytorów. W kontekście edytora Visual Studio Code oprócz komplikacji i uruchomienia programu zastosujemy narzędzie **dotnet CLI** z wiersza poleceń do utworzenia początkowego szkieletu programu w języku C#. Na potrzeby systemów Windows i macOS skupimy się na używaniu środowiska Visual Studio 2019.

Używanie narzędzia Dotnet CLI

Polecenie dotnet.ext to używany w wierszu poleceń interfejs narzędzia dotnet CLI. Za pomocą tego narzędzia można wygenerować początkowy kod programu C#, a także skompilować i uruchomić program⁴. Nazwa CLI może oznaczać technologię *Common Language Infrastructure* lub interfejs z wiersza poleceń (ang. *Command-Line Interface*). Dlatego aby uniknąć pomyłek, w tej książce poprzedzam akronim CLI nazwą dotnet, gdy chodzi o narzędzie dotnet CLI. Nazwa CLI bez członu dotnet oznacza technologię *Common Language Infrastructure*. Po zakończeniu instalacji sprawdź, czy polecenie dotnet jest dostępne w wierszu poleceń. Pozwoli to stwierdzić, czy instalacja przebiegła poprawnie.

Oto instrukcje tworzenia, komplikowania i wykonywania programu *HelloWorld* w wierszu poleceń systemów Windows, macOS i Linux:

1. Otwórz wiersz poleceń w systemie Microsoft Windows lub terminal w systemie Mac/macOS. Możesz też rozważyć użycie dostępnego w wielu systemach interfejsu PowerShell⁵.

2. Utwórz nowy katalog, w którym chcesz umieścić kod. Możesz go nazwać `./HelloWorld`, `./EssentialCSharp/HelloWorld`. W wierszu poleceń wywołaj instrukcję:

```
mkdir ./HelloWorld
```

3. Przejdź do nowego katalogu, tak aby był aktualną lokalizacją w wierszu poleceń:

```
cd ./HelloWorld
```

4. Wywołaj instrukcję `dotnet new console` w katalogu *HelloWorld*, aby wygenerować początkowy szkielet programu. Utworzonych zostanie kilka plików. Dwa najważniejsze z nich to *Program.cs* i plik projektu:

```
dotnet new console
```

5. Uruchom wygenerowany program. Spowoduje to skompilowanie i uruchomienie domyślnego pliku *Program.cs* utworzonego przez instrukcję `dotnet new console`. Zawartość pliku *Program.cs* wygląda podobnie jak kod z listingu 1.1, jednak wyświetlany jest napis „Hello, World!”.

```
dotnet run
```

Choć nie zażądzano bezpośrednio komplikacji (ani budowania) aplikacji, ten krok jest wykonywany, ponieważ zostaje wywołany pośrednio w wyniku uruchomienia instrukcji `dotnet run`.

6. Otwórz plik *Program.cs* i zmodyfikuj kod, aby wyglądał tak jak na listingu 1.1. Jeśli używasz środowiska Visual Studio Code do otwierania i edytowania pliku *Program.cs*, dostrzeżesz zalety edytora z obsługą języka C#, ponieważ kolory w kodzie będą oznaczać różne rodzaje elementów w programie. Aby otworzyć plik na potrzeby edycji, użyj następującego polecenia:

⁴ To narzędzie zostało udostępnione mniej więcej w tym samym czasie co język C# 7.0 i wyparło bezpośrednie komplikowanie kodu za pomocą kompilatora języka C# `csc.exe`.

⁵ Zobacz stronę <https://github.com/PowerShell/PowerShell>.

```
code .
```

Instrukcje z danych wyjściowych 1.1 działają w wierszach poleceń Bash i PowerShell.

7. Ponownie uruchom program:

```
dotnet run
```

W danych wyjściowych 1.1 pokazano dane wyjściowe wyświetlane po wykonaniu wcześniejszych kroków⁶.

DANE WYJŚCIOWE 1.1

```
1>
2> mkdir ./HelloWorld
3> cd ./HelloWorld/
4> dotnet new console
Pomyślnie utworzono szablon "Console Application".
```

Trwa przetwarzanie akcji po utworzeniu...

```
Trwa uruchamianie polecenia "dotnet restore" dla ...\\EssentialCSharp\\HelloWorld\\
↪HelloWorld.csproj...
  Restoring packages for ...\\EssentialCSharp\\HelloWorld\\
↪HelloWorld.csproj...
  Generating MSBuild file ...\\EssentialCSharp\\HelloWorld\\obj\\
↪HelloWorld.csproj.nuget.g.props.
  Generating MSBuild file ...\\EssentialCSharp\\HelloWorld\\obj\\
↪HelloWorld.csproj.nuget.g.targets.
  Restore completed in 184.46 ms for ...\\EssentialCSharp\\
↪HelloWorld\\HelloWorld.csproj.
```

Przywracanie powiodło się.

```
5> dotnet run
Hello World!
6> echo '
class HelloWorld
{
    static void Main()
    {
        System.Console.WriteLine("Witaj. Nazywam się Inigo Montoya.");
    }
}
' > Program.cs
7> dotnet run
Witaj. Nazywam się Inigo Montoya.
8>
```

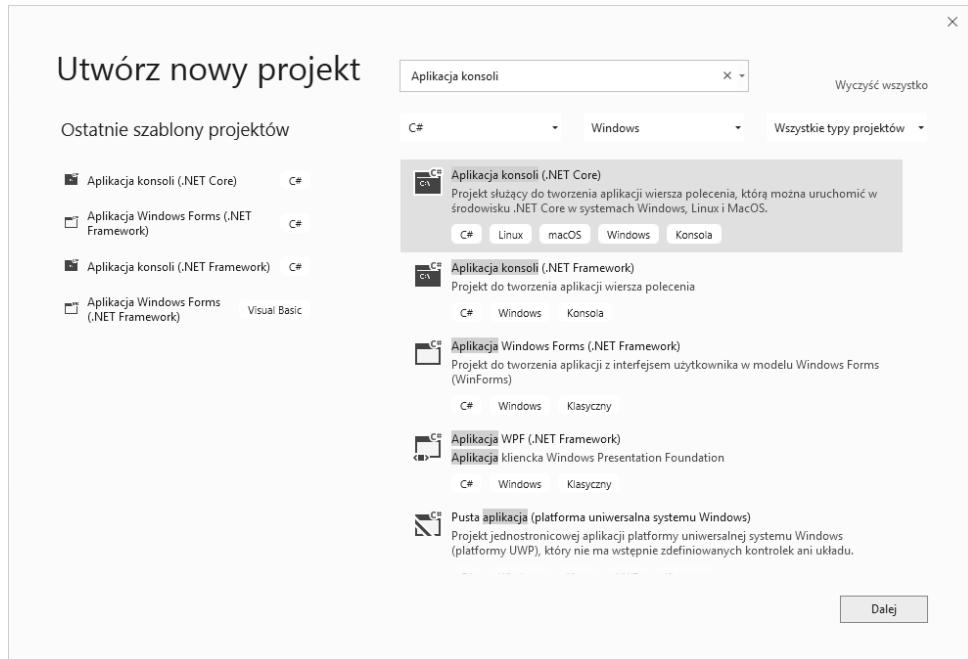
Koniec
7.0

Używanie środowiska Visual Studio 2019

W środowisku Visual Studio 2019 procedura wygląda podobnie, jednak zamiast wiersza poleceń używane jest **środowisko IDE** (ang. *Intergrated Development Environment*). Obejmuje ono menu z opcjami do wyboru; nie trzeba wtedy wykonywać wszystkich operacji w wierszu poleceń.

⁶ Pogrubienie w danych wyjściowych oznacza fragmenty wpisywane przez użytkownika.

1. Uruchom środowisko Visual Studio 2019.
2. Kliknij przycisk *Utwórz nowy projekt*. Jeśli okno startowe nie jest widoczne, możesz je wyświetlić za pomocą opcji *Plik/Uruchom okno*. Inna możliwość to bezpośrednie przejście do tworzenia projektu za pomocą opcji *Plik/Nowy projekt (Ctrl+Shift+N)*.
3. W polu *Wyszukaj (Alt+S)* wpisz *Aplikacja konsoli* i wybierz opcję *Aplikacja konsoli (.NET Core) — Visual C#* (zobacz rysunek 1.2).

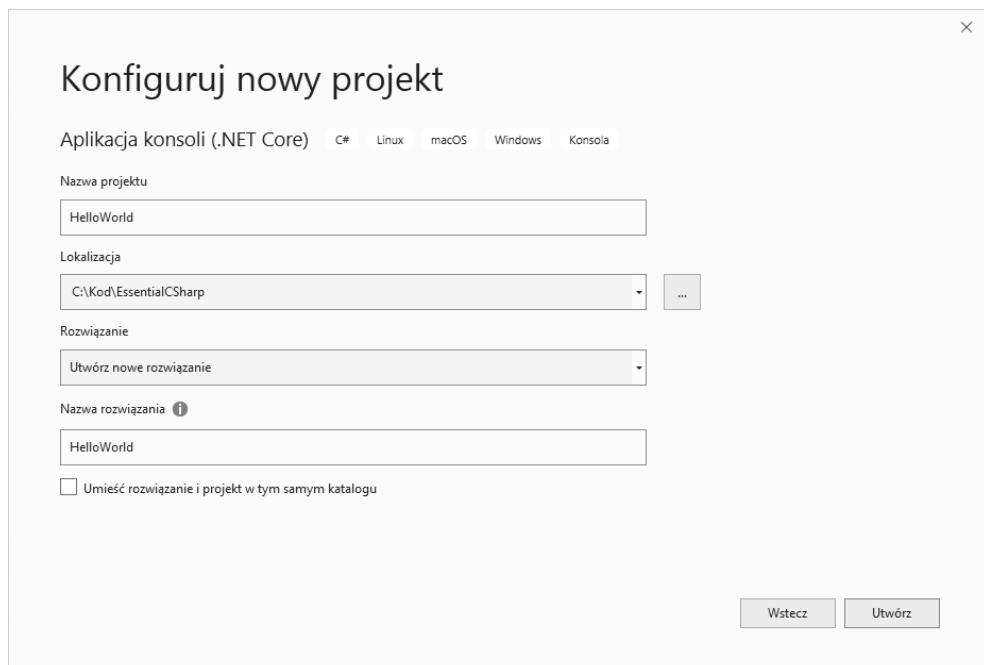


Rysunek 1.2. Okno dialogowe Utwórz nowy projekt

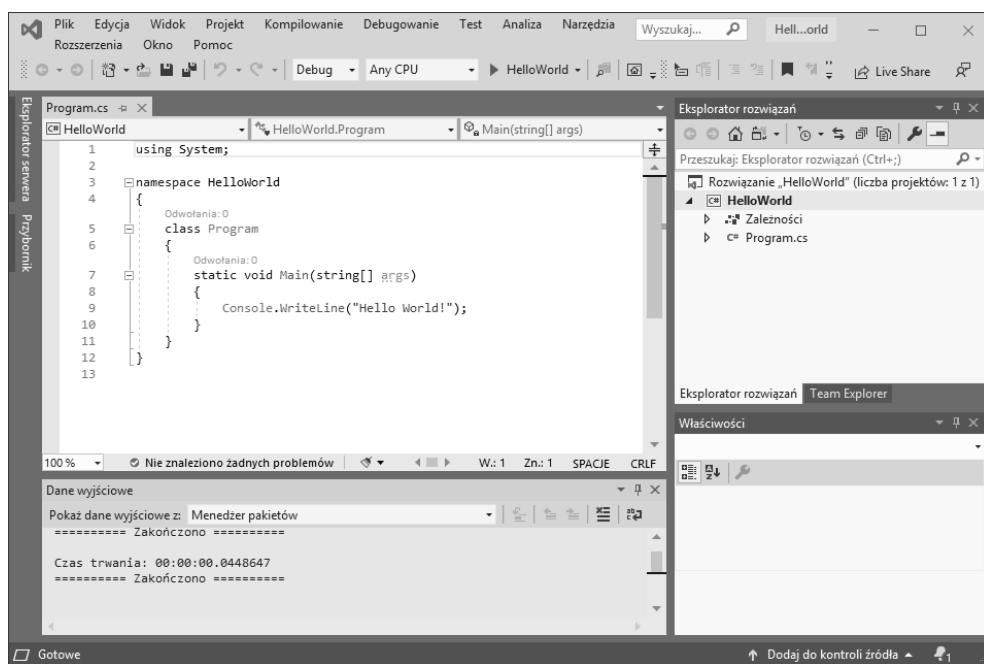
4. W polu tekstowym *Nazwa projektu* wpisz tekst *HelloWorld*, a w polu *Lokalizacja* wybierz dowolny katalog roboczy (rysunek 1.3).
5. Po utworzeniu projektu powinieneś zobaczyć plik *Program.cs* przedstawiony na rysunku 1.4.
6. Uruchom wygenerowany program za pomocą opcji *Debugowanie/Uruchom bez debugowania (Ctrl+F5)*. Wyświetlone zostanie okno polecen z tekstem widocznym w danych wyjściowych 1.2, przy czym w pierwszym wierszu pojawi się tylko tekst „Hello World!”.
7. Zmodyfikuj plik *Program.cs*, aby wyglądał tak jak na listingu 1.1.
8. Ponownie uruchom program, a zobaczysz dane wyjściowe 1.2.

DANE WYJŚCIOWE 1.2

```
> Witaj. Nazywam się Inigo Montoya.  
Press any key to continue . . .
```



Rysunek 1.3. Okno dialogowe Konfiguruj nowy projekt



Rysunek 1.4. Okno dialogowe z plikiem Program.cs

Debugowanie

Jedną z ważnych cech środowisk IDE jest obsługa debugowania. Aby wypróbować ten mechanizm, wykonaj następujące dodatkowe kroki w środowisku Visual Studio lub edytorze Visual Studio Code:

1. Umieść kursor w wierszu `System.Console.WriteLine` i kliknij w menu opcję *Debugowanie/Przełącz punkt przerwania (F9)*, aby dodać punkt przerwania w danym wierszu.
2. Wybierz opcję *Debugowanie/Rozpocznij debugowanie (F5)*, aby ponownie uruchomić aplikację — tym razem z włączonym debugowaniem.

W edytorze Visual Studio Code należy wybrać środowisko (wybierz .NET Core), na podstawie którego wygenerowane zostaną pliki `launch.json` i `tasks.json`. Następnie trzeba ponownie uruchomić debugowanie, wybierając opcję *Debug/Start Debugging (F5)*.

Zauważ, że wykonanie zatrzyma się w wierszu z ustawionym punktem przerwania. Możesz wtedy umieścić kursor na zmiennej (np. `args`), aby zobaczyć jej wartość. Możesz też przenieść wykonywanie programu z bieżącego wiersza do innego miejsca metody, przeciągając żółtą strzałkę widoczną na lewym marginesie okna z plikiem.

3. Aby kontynuować wykonywanie programu, użyj opcji *Debugowanie/Kontynuuj (F5)* lub przycisku *Kontynuuj*.

Więcej informacji na temat debugowania w środowisku Visual Studio 2019 znajdziesz na stronie <http://itl.tc/vsdebugging>.

W edytorze Visual Studio Code dane wyjściowe pojawiają się w zakładce *Debug Console* (wybierz opcję *View/Debug Console* lub kombinację *Ctrl+Shift+V*, aby zobaczyć tekst „Witaj. Nazywam się Inigo Montoya”). Więcej informacji o debugowaniu w tym edytorze zawiera strona <https://code.visualstudio.com/docs/editor/debugging>.

Tworzenie projektu

Niezależnie od tego, czy używasz narzędzi dotnet CLI, czy środowiska Visual Studio, dwóch różnych jest kilka plików. Pierwszym z nich tradycyjnie jest plik C# o nazwie `Program.cs`. Nazwa `Program` jest zwyczajowo używana jako punkt wejścia do programu konsolowego, jednak można zastosować dowolną inną nazwę. Rozszerzenie `.cs` jest standardowo stosowana dla wszystkich plików C# i kompilator domyślnie kompiluje takie pliki, tworząc gotowy program. Aby zastosować kod z listingu 1.1, otwórz plik `Program.cs` i zastąp jego zawartość tym kodem. Przed zapisaniem zaktualizowanego pliku zauważ, że jedyna funkcjonalna różnica między listkiem 1.1 a domyślnie wygenerowanym kodem dotyczy tekstu ujętego w cudzysłów. Semantyczną różnicą jest lokalizacja instrukcji `System`.

Porównanie języków — w Javie nazwy plików muszą odpowiadać nazwom klas

W Javie nazwa pliku musi odpowiadać nazwie klasy. W języku C# ta konwencja często jest przestrzegana, ale nie jest to konieczne. W C# można umieścić dwie klasy w jednym pliku, a dopuszczalne jest nawet zapisanie jednej klasy w wielu plikach (umożliwiają to **klasy częściowe**). Takie rozwiązanie stosowane jest w różnych technologiach, a technologie należy traktować jak narzędzia. Im dokładniej je poznasz, tym lepiej będziesz przygotowany do dokonywania najważniejszych architektonicznych wyborów dostosowanych do wymogów aplikacji.

Choć nie zawsze jest to konieczne, w projektach w języku C# zwykle razem z generowanym kodem źródłowym tworzony jest plik konfiguracyjny nazywany **plikiem projektu**. Zawartość pliku projektu zależy od typu aplikacji i wersji platformy .NET. Jednak przeważnie taki plik przynajmniej określa, jakie pliki mają być uwzględniane w trakcie komplikacji, jakiego typu aplikację należy zbudować (konsolową, internetową, mobilną, testową itd.), która wersja platformy .NET ma być obsługiwana, a także jakie ustawienia są potrzebne do debugowania lub uruchamiania aplikacji. Podawane są też zależności (*biblioteki*) potrzebne w kodzie. Na listingu 1.2 pokazano prosty przykładowy plik projektu utworzonej wcześniej aplikacji konsolowej na platformę .NET Core.

Listing 1.2. Przykładowy plik projektu aplikacji konsolowej dla platformy .NET Core

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp3.1</TargetFramework>
  </PropertyGroup>
</Project>
```

Na listingu 1.2 typ aplikacji jest określony jako aplikacja konsolowa (Exe) dla platformy .NET Core w wersji 3.1 (netcoreapp3.1). Wszystkie pozostałe ustawienia (np. to, które pliki C# należy skompilować) są określane na podstawie konwencji. Na przykład domyślnie komplowane są wszystkie pliki *.cs znajdujące się w tym samym katalogu (lub podkatalogu) co plik projektu.

Kompilowanie i wykonywanie kodu

Skompilowany projekt utworzony przez polecenie dotnet build to **podzespoł** o nazwie *HelloWorld.dll*⁷. Rozszerzenie .dll oznacza bibliotekęłączoną dynamicznie (ang. *Dynamic Link Library*). W platformie .NET Core wszystkie podzespoły mają rozszerzenie .dll — nawet programy konsolowe takie jak utworzony w przykładzie. Skompilowany projekt aplikacji

⁷ Warto zauważyć, że jeśli używasz platformy Microsoft .NET Framework do tworzenia programu konsolowego, skompilowany kod jest umieszczany w pliku *HelloWorld.exe*, który można uruchomić bezpośrednio (o ile na komputerze zainstalowana jest wspomniana platforma).

na platformę .NET Core domyślnie jest umieszczany w podkatalogu `./bin/Debug/netcoreapp3.1/linux-x64/publish/`. Katalog `Debug` jest używany, ponieważ domyślnie stosowana jest konfiguracja na potrzeby debugowania. Ta konfiguracja powoduje, że skompilowany projekt jest zoptymalizowany pod kątem debugowania, a nie z myślą o wydajności. Tak skompilowany projekt nie jest wykonywany samodzielnie. Do wykonywania kodu potrzebne jest wtedy środowisko CLI. W przypadku aplikacji dla platformy .NET Core wymaga to użycia procesu `dotnet.exe` jako hosta aplikacji (w systemach Linux i macOS jest to proces `dotnet`). Dlatego program jest wykonywany za pomocą polecenia `dotnet run`. Istnieje jednak sposób na wygenerowanie niezależnego pliku wykonywalnego, który obejmuje niezbędne pliki uruchomieniowe. Wtedy instalowanie środowiska uruchomieniowego `dotnet` nie jest konieczne. Więcej dowiesz się z ramki ZAGADNIENIE DLA ZAAWANSOWANYCH „Udostępnianie niezależnego pliku wykonywalnego”.

ZAGADNIENIE DLA ZAAWANSOWANYCH

Udostępnianie niezależnego pliku wykonywalnego

Można wygenerować niezależny plik wykonywalny, uruchamiany bez konieczności używania polecenia `dotnet`. W tym celu wywołaj polecenie `dotnet publish` z argumentem `--runtime(-r)`. Ten argument określa docelową platformę (system operacyjny), z którą zgodny będzie wynikowy plik. W większości systemów linuksowych możesz użyć argumentu `linux-x64` i wywołać polecenie w katalogu, w którym znajduje się plik `.csproj`.

```
dotnet publish --runtime linux-x64
```

Wykonanie tego polecenia spowoduje utworzenie katalogu `(./bin/Debug/netcoreapp3.1/linux-x64/publish/)` zawierającego wszystkie pliki potrzebne do uruchomienia programu konsolego `HelloWorld` bez konieczności wcześniejszego instalowania środowiska uruchomieniowego `dotnet`. Aby uruchomić ten program, użyj nazwy pliku wykonywalnego i podaj ścieżkę do katalogu bieżącego (jeśli nie jest nim katalog `publish`):

```
./bin/Debug/netcoreapp3.1/linux-x64/publish/HelloWord
```

W systemie Windows nazwa pliku wykonywalnego obejmuje rozszerzenie `.exe`, jednak gdy uruchamiasz program, nie musisz go podawać.

Zauważ, że otrzymany tu plik wykonywalny będzie działać tylko w systemach z rodziny `linux-x64`. Dlatego polecenie `dotnet publish` trzeba wykonać dla każdej docelowej platformy. Często używane identyfikatory innych platform to `win-x64` i `osx-x64` (kompletną listę znajdziesz na stronie <http://itl.tc/ridcatalog>).

Można też udostępnić niezależny program wykonywalny w jednym pliku (zwykle używanych jest około 200 plików). W celu uzyskania jednego pliku wykonywalnego dodaj do polecenia człon `/p:PublishSingleFile=true`:

```
dotnet publish --runtime linux-x64 -p:PublishSingleFile=true
```

Więcej informacji o argumencie `-p` zawiera rozdział 10.

Kod źródłowy do książki C# 8.0

Kod źródłowy⁸ dołączony do książki obejmuje plik rozwiązania, *EssentialCSharp.sln*, który łączy kod z wszystkich rozdziałów. Ten kod źródłowy można zbudować, uruchomić i przetestować za pomocą narzędzi Visual Studio lub dotnet CLI.

Prawdopodobnie najprostszą techniką jest umieszczenie kodu w utworzonym wcześniej w rozdziale programie *HelloWorld* i uruchomienie tego kodu. Kod źródłowy zawiera też pliki projektów z każdego rozdziału i menu, w którym można wybrać listing do uruchomienia. Te elementy są opisane w dwóch następnych podpunktach.

Używanie dotnet CLI

Aby budować i uruchamiać kod za pomocą narzędzia dotnet CLI, otwórz wiersz poleceń i przejdź do katalogu z plikiem *EssentialCSharp.sln*. Następnie wywołaj instrukcję `dotnet build`, aby skompilować wszystkie projekty.

W celu uruchomienia kodu źródłowego z konkretnego projektu przejdź do katalogu zawierającego plik projektu i wywołaj instrukcję `dotnet run`. Możesz też z poziomu dowolnego katalogu użyć instrukcji `dotnet run -p <plik_projektu>`, gdzie *plik_projektu* to ścieżka do pliku projektu, który chcesz uruchomić (np. `dotnet run -p .\src\Chapter01\Chapter01.csproj`). Po tym wywołaniu program wyświetli pytanie o to, który listing należy wykonać, a następnie uruchomi odpowiedni kod.

Dla wielu listingów dostępne są testy jednostkowe w katalogu *Chapter[??].Tests*, gdzie człon *[??]* oznacza numer rozdziału. Aby uruchomić testy, użyj polecenia `dotnet test` w katalogu z testami, które chcesz wykonać. Możesz też wywołać to samo polecenie w katalogu z plikiem *EssentialCSharp.sln*, aby wykonać wszystkie testy.

Używanie Visual Studio

Po otwarciu pliku rozwiązania użyj opcji *Kompilowanie/Skompiluj rozwiązanie*, aby skompilować kod za pomocą środowiska Visual Studio. Zanim będziesz mógł uruchomić kod źródłowy, musisz wybrać, który projekt wykonać. W tym celu wskaż projekt powiązany z danym rozdziałem jako projekt startowy. Na przykład aby uruchomić przykłady z rozdziału 1., kliknij prawym przyciskiem myszy projekt *Chapter01* i wybierz opcję *Ustaw jako projekt startowy*. Jeśli nie wskażesz właściwego rozdziału, to po podaniu numeru listingu zgłoszony zostanie wyjątek i zobaczysz komunikat z informacją o braku podanego programu.

Po wybraniu właściwego projektu możesz go uruchomić za pomocą opcji *Debugowanie/Uruchom bez debugowania*. Jeżeli chcesz debugować projekt, wybierz opcję *Debugowanie/Rozpocznij debugowanie*. Gdy program zacznie pracę, wyświetli prośbę o podanie numeru listingu (na przykład 18.33), który ma zostać uruchomiony. Jak wcześniej wspomniano, uruchomić można wyłącznie listingi z projektu ustawionego jako startowy.

⁸ Kod źródłowy do tej książki (wraz z wybranymi rozdziałami dotyczącymi starszych wersji języka C#) możesz pobrać ze strony <https://IntelliTect.com/EssentialCSharp>. Kod jest też dostępny bezpośrednio w serwisie GitHub: <https://github.com/IntelliTect/EssentialCSharp>. Spolszoną wersję znajdziesz w witrynie wydawnictwa Helion.

Wiele listingów jest powiązanych z testami jednostkowymi. Aby uruchomić określony test, otwórz projekt z testami i przejdź do testu odpowiadającego listingowi, który chcesz zbadać. Następnie kliknij prawym przyciskiem myszy metodę testową i wybierz opcję *Uruchom testy* (*Ctrl+R, T*) lub *Debuguj testy* (*Ctrl+R, Ctrl+T*).

Podstawy składni języka C#

Jeśli już z powodzeniem skompilowałeś i uruchomileś program `HelloWorld`, możesz rozpocząć analizę kodu, by poznać jego elementy. Zaczni od przyjrzenia się słowom kluczowym języka C# oraz identyfikatorom tworzonym przez programistów.

ZAGADNIENIE DLA POCZĄTKUJĄCYCH

Słowa kluczowe

Aby umożliwić kompilatorowi interpretację kodu, niektórym słowom w języku C# przypisano specjalny status i określone znaczenie. Są to **słową kluczowe**, będące częścią składni używanej przez kompilator do interpretowania wyrażeń pisanych przez programistów. W programie `HelloWorld` słowami kluczowymi są na przykład `class`, `static` i `void`.

Kompilator na podstawie słów kluczowych ustala strukturę i uporządkowanie kodu. Ponieważ dla kompilatora te słowa mają specjalne znaczenie, język C# wymaga, by były umieszczane tylko w określonych miejscach. Jeśli programista naruszy obowiązujące reguły, kompilator zgłosi błąd.

Słowa kluczowe języka C#

Słowa kluczowe języka C# są przedstawione w tabeli 1.1.

Początek
2.0

Po wersji C# 1.0 do języka C# nie dodano żadnych nowych **zarezerwowanych słów kluczowych**. Jednak w niektórych konstrukcjach wprowadzonych w późniejszych wersjach używane są **kontekstowe słowa kluczowe**, które mają specjalne znaczenie wyłącznie w określonych miejscach. Poza tymi lokalizacjami kontekstowe słowa kluczowe nie mają specjalnego znaczenia⁹. Dzięki temu większość kodu dostosowanego do wersji C# 1.0 jest zgodna także z późniejszymi standardami¹⁰.

Koniec
2.0

⁹ Na przykład na początku prac nad wersją C# 2.0 projektanci języka ustalili, że `yield` będzie słowem kluczowym. Microsoft udostępnił więc tysiącom programistów wersję alfa kompilatora języka C# 2.0 ze słowem kluczowym `yield`. Jednak ostatecznie projektanci stwierdzili, że dzięki zastosowaniu konstrukcji `yield return` zamiast samego `yield` mogą zrezygnować z dodawania słowa kluczowego `yield`, ponieważ nie ma ono specjalnego znaczenia, jeśli występuje niezależnie od słowa `return`.

¹⁰ W języku występują rzadkie i niefortunne niezgodności. Oto niektóre z nich:

- Wersja C# 2.0 wymaga implementowania interfejsu `IDisposable` z wykorzystaniem instrukcji `using`, a nie z metodą `Dispose()`.
- Niektóre rzadko stosowane wyrażenia generyczne, na przykład `F(G<A,B>(7))`, w wersji C# 1.0 oznaczają `F((G<A>, (B>7))`, natomiast w wersji C# 2.0 reprezentują wywołanie generycznej metody `G<A,B>` z argumentem 7 i wynikiem zwracanym do `F`.

Tabela 1.1. Słowa kluczowe języka C#

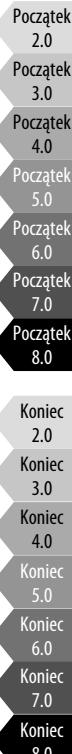
abstract	enum	long	stackalloc	static
add* (1)	equals* (3)	nameof* (6)	string	
alias* (2)	event	namespace	struct	
as	explicit	new	switch	
ascending* (3)	extern	nonnull* (8)	this	
async* (5)	false	null	throw	
await* (5)	finally	object	true	
base	fixed	on* (3)	try	
bool	float	operator	typeof	
break	for	orderby* (3)	uint	
by* (3)	foreach	out	ulong	
byte	from* (3)	override	unchecked	
case	get* (1)	params	unmanaged* (7.3)	
catch	global* (2)	partial* (2)	unsafe	
char	goto	private	ushort	
checked	group* (3)	protected	using	
class	if	public	value* (1)	
const	implicit	readonly	var* (3)	
continue	in	ref	virtual	
decimal	int	remove* (1)	void	
default	interface	return	volatile	
delegate	internal	sbyte	where* (2)	
descending* (3)	into* (3)	sealed	when* (6)	
do	is	select* (3)	while	
double	join* (3)	set* (1)	yield* (2)	
dynamic* (4)	let* (3)	short		
else	lock	sizeof		

* Kontekstowe słowa kluczowe. Numer w nawiasie (*n*) wskazuje, w której wersji języka dodano określone kontekstowe słowo kluczowe.

Identyfikatory

C#, podobnie jak inne języki, obejmuje **identyfikatory**, służące do identyfikowania elementów w kodzie pisany przez programistę. Na listingu 1.1 identyfikatorami są na przykład `HelloWorld` i `Main`. Te identyfikatory są przypisane do elementów kodu i używane później do wskazywania tych elementów. Dlatego ważne jest, by nazwy podane przez programistę były znaczące i opisowe, a nie arbitralne.

Umiejętność doboru związkowych i opisowych nazw to ważna cecha dobrego programisty, ponieważ sprawia, że gotowy kod będzie łatwy do zrozumienia i ponownego wykorzystania. Przejrzystość w połączeniu ze spójnością jest na tyle ważna, że w wytycznych dotyczących platformy .NET (<http://bit.ly/dotnetguidelines>) odradza się stosowania skrótów nazw w identyfikatorach. Ponadto zaleca się unikanie akronimów, które nie są w powszechnym użyciu. Jeśli akronim jest wystarczająco rozpowszechniony (na przykład HTML), należy się nim konsekwentnie posługiwać. Unikaj stosowania akronimu w jednym miejscu, a kompletnej nazwy w innym. Ograniczenie zmuszające programistów do umieszczania wszystkich akronimów w słowniczku jest na tyle uciążliwe, że chroni przed ich nadużywaniem. Stosuj oczywiste nazwy, nawet jeśli będą rozwlekłe. Dotyczy to zwłaszcza sytuacji, gdy pracujesz w zespole lub rozwijasz bibliotekę, która będzie używana przez inne osoby.



Istnieją dwa podstawowe formaty stosowania wielkich i małych liter w identyfikatorach. **Notacja pascalowa** (NotacjaPascalowa) została nazwana tak przez autorów platformy .NET z powodu popularności tego zapisu w języku Pascal. Polega ona na rozpoczynaniu wielką literą każdego słowa w identyfikatorze. Oto przykładowe identyfikatory w tej notacji: ComponentModel, Configuration, HttpFileCollection. Identyfikator HttpFileCollection pokazuje, że w akronimach obejmujących więcej niż dwie litery (takich jak HTTP) tylko pierwsza litera jest wielka. Drugi format, notacja wielbłąda (notacjaWielbłąda), działa podobnie, przy czym pierwsza litera całego identyfikatora jest mała. Oto przykłady: quotient, firstName, httpFileCollection, ioStream i theDreadedPirateRoberts.

Wskazówki

PRZEDKŁADAJ jasność nad zwięzłość w trakcie tworzenia identyfikatorów.

NIE stosuj skrótów w nazwach identyfikatorów.

NIE stosuj akronimów, chyba że są powszechnie używane. Wtedy korzystaj z nich konsekwentnie.

Podkreślenia są dopuszczalne, jednak zwykle w nazwach identyfikatorów nie występują podkreślenia, myślniki ani inne znaki niealfanumeryczne. Ponadto w języku C#, inaczej niż w starszych językach, nie korzysta się z notacji węgierskiej (polegającej na poprzedzaniu nazw skrótem reprezentującym typ danych). Pozwala to uniknąć modyfikowania nazw zmiennych po zmianie typu danych lub niespójności powstających w sytuacji, gdy programista używający notacji węgierskiej nie dostosuje odpowiednio przedrostka określającego typ.

Niektóre identyfikatory (jest ich niewiele), na przykład Main, mają w języku C# specjalne znaczenie.

Wskazówki

STOSUJ dwie wielkie litery dla dwuliterowych akronimów. Wyjątkiem jest pierwsze słowo w identyfikatorach w notacjiWielbłąda.

STOSUJ wielką literę tylko dla pierwszego znaku w akronimach obejmujących przynajmniej trzy litery. Wyjątkiem jest pierwsze słowo w identyfikatorach w notacjiWielbłąda.

NIE stosuj wielkich liter w akronimach na początku identyfikatorów w notacjiWielbłąda.

NIE stosuj notacji węgierskiej (nie zapisuj typu zmiennej w jej nazwie).

ZAGADNIENIE DLA ZAAWANSOWANYCH

Słowa kluczowe

Choć zdarza się to rzadko, słowa kluczowe są czasem stosowane jako identyfikatory. Wymaga to dodania przedrostka @. Możesz na przykład nazwać zmienną lokalną @return. Podobnie (choć nie jest to zgodne ze standardami stosowania wielkich liter w kodzie języka C#) można nazwać metodę @throw().

W implementacji opracowanej przez Microsoft występują cztery nieudokumentowane słowa kluczowe. Oto one: `_arglist`, `_makeref`, `_reftype` i `_refvalue`. Są one potrzebne tylko w rzadkich sytuacjach współdziałania komponentów i w praktyce można je zignorować. Zauważ, że te cztery specjalne słowa kluczowe rozpoczynają się od dwóch symboli podkreślenia. Projektanci języka C# zastrzegli sobie możliwość przekształcenia w przyszłość każdego identyfikatora, który rozpoczyna się od dwóch znaków podkreślenia, w słowo kluczowe. Dla bezpieczeństwa unikaj tworzenia tego typu identyfikatorów.

Definicja typu

Cały wykonywalny kod w języku C# pojawia się w definicjach typów. Najczęściej stosowana definicja typu rozpoczyna się od słowa kluczowego `class`. **Definicja klasy** to blok kodu, który zwykle rozpoczyna się od członu `class identyfikator { ... }`. Przykładową definicję przedstawia listing 1.3.

Listing 1.3. Prosta deklaracja klasy

```
class HelloWorld
{
    //...
}
```

Nazwa typu może być dowolna (tu jest to `HelloWorld`), jednak zgodnie ze zwyczajem należy stosować NotacjęPascalową. W tym przykładzie możliwe inne nazwy to `Greetings`, `HelloInigoMontoya`, `Hello` lub po prostu `Program`. `Program` to dobra nazwa, gdy klasa zawiera opisaną dalej metodę `Main()`.

Wskazówki

STOSUJ rzeczowniki lub frazy nominalne jako nazwy klas.

STOSUJ NotacjęPascalową we wszystkich nazwach klas.

Zwykle programy obejmują wiele typów, które zawierają po wiele metod.

Metoda Main

ZAGADNIENIE DLA POCZĄTKUJĄCYCH

Czym jest metoda

Składniowo **metoda** w języku C# to nazwany blok kodu dodany za pomocą deklaracji metody (na przykład `static void Main()`), po której zwykle następują instrukcje w nawiasach klamrowych. Metody wykonują obliczenia lub operacje. Pełnią podobną funkcję jak akapity w tekście — umożliwiają strukturyzowanie i porządkowanie kodu, dzięki czemu staje się on bardziej czytelny. Co ważniejsze, metody można wielokrotnie stosować i wywoływać w wielu

miejscach, co pozwala uniknąć powielania kodu. Deklaracja metody dodaje metodę i definiuje jej nazwę oraz dane przekazywane do metody i przez nią zwracane. Na listingu 1.4 metodą języka C# jest Main() z nawiasem { ... }.

Miejsce, w którym rozpoczyna się wykonywanie programów w języku C#, to **metoda Main**. Jej deklaracja zaczyna się od członu static void Main(). Gdy wywołasz program, wpisując instrukcję dotnet run w konsoli, program się uruchomi, znajdzie metodę Main i rozpoczęcie wykonywanie pierwszej instrukcji z listingu 1.4.

Listing 1.4. Analiza klasy HelloWorld

```
class HelloWorld
{
    static void Main() } Deklaracja metody
    {
        System.Console.WriteLine("Witaj. Nazywam się Inigo Montoya");
    } } Metoda Main
} } Instrukcja } Definicja klasy
```

Choć deklaracje metody Main mogą przyjmować różną postać, zawsze potrzebny są modyfikator static i nazwa metody, Main.

ZAGADNIENIE DLA ZAAWANSOWANYCH

Deklaracja metody Main

Język C# wymaga, aby metoda Main zwracała wartość void lub wartość typu int. Ponadto ta metoda może nie przyjmować żadnych parametrów lub pobierać jedną tablicę łańcuchów znaków. Listing 1.5 przedstawia kompletną deklarację metody Main. Parametr args to tablica łańcuchów znaków reprezentujących argumenty wprowadzone w wierszu poleceń. Jednak (inaczej niż w językach C i C++) pierwszym elementem tej tablicy nie jest nazwa programu. Aby pobrać kompletne polecenie użyte do uruchomienia programu, wykorzystaj właściwość System.Environment.CommandLine.

Listing 1.5. Metoda Main z parametrami i określonym typem zwracanej wartości

```
static int Main(string[] args)
{
    //...
}
```

Porównanie języków — w językach C++ i Java nazwa main() składa się tylko z małych liter

Wartość typu int zwracana tu przez metodę Main to kod statusu informujący o tym, czy wykonanie programu zakończyło się sukcesem. Niezerowa wartość zwykle oznacza błąd.

Ponadto w wersji C# 7.1 dodana została obsługa mechanizmu async/await w metodzie Main.

Uwaga

Inaczej niż w starszych językach opartych na C, w języku C# w nazwie metody `Main` występuje wielkie `M`. Pozwala to zachować zgodność z Notacją Pascalową stosowaną w nazwach w języku C#.

Oznaczenie metody `Main` jako statycznej (`static`) powoduje, że inne metody mogą wywoływać ją bezpośrednio poza definicją klasy. Jeśli modyfikator `static` nie jest używany, konsola uruchamiająca program przed wywołaniem metody musi wykonać dodatkowe operacje (**utworzyć instancję klasy**). W rozdziale 6. znajdziesz cały podrozdział poświęcony składowym statycznym.

Słowo `void` przed nazwą `Main()` określa, że dana metoda nie zwraca żadnych danych. Więcej na ten temat dowiesz się z rozdziału 2.

Jedną z cech języków C i C++ występującą też w C# jest umieszczanie zawartości elementu (klasy lub metody) w nawiasie klamrowym. Na przykład metoda `Main` obejmuje nawias klamrowy, w którym znajduje się jej implementacja. Tu implementacja metody to tylko jedna instrukcja.

Instrukcje i ograniczniki instrukcji

Metoda `Main` obejmuje tu jedną instrukcję, `System.Console.WriteLine()`, służącą do wyświetlania wiersza tekstu w konsoli. W języku C# standardowo do oznaczania końca instrukcji służy średnik. Instrukcja składa się z jednej lub kilku operacji wykonywanych przez kod. Typowe zastosowania instrukcji to deklarowanie zmiennych, kontrolowanie przepływu sterowania i wywoływanie metod.

Porównanie języków — instrukcje wierszowe w Visual Basicu

Niektóre języki są oparte na wierszach, co oznacza, że jeśli nie dodasz specjalnych znaków, instrukcje nie mogą rozciągać się na kilka wierszy. Do wersji Visual Basic 2010 język ten był oparty na wierszach. Aby określić, że instrukcja rozciąga się do następnego wiersza, na końcu bieżącego trzeba było dodać podkreślenie. Od wersji Visual Basic 2010 w wielu przypadkach dodawanie znaku kontynuacji instrukcji jest opcjonalne.

ZAGADNIENIE DLA ZAAWANSOWANYCH

Instrukcje bez średników

Wiele jednostek kodu w języku C# kończy się średnikiem. Jednym z przykładowych elementów, po których średnik nie jest używany, jest instrukcja `switch`. Ponieważ w instrukcji `switch` zawsze używane są nawiasy klamrowe, język C# nie wymaga dodawania średnika po tej instrukcji. Całe bloki kodu też są uznawane za instrukcje (które same składają się z innych instrukcji) i nie wymagają zakończenia średnikiem. Ponadto w niektórych sytuacjach, na przykład gdy stosowana jest deklaracja `using`, na końcu wiersza pojawia się średnik, ale nie jest on wtedy zakończeniem instrukcji.

Ponieważ przejście do nowego wiersza nie powoduje rozdzielenia instrukcji, możesz umieścić wiele instrukcji w tym samym wierszu, a kompilator języka C# uzna, że wiersz obejmuje zestaw poleceń. Na przykład na listingu 1.6 w jednym wierszu znajdują się dwie instrukcje, które powodują wyświetlenie słów *Góra* i *Dół* w dwóch odrębnych wierszach.

Listing 1.6. Kilka instrukcji w jednym wierszu

```
System.Console.WriteLine("Góra");System.Console.WriteLine("Dół");
```

Język C# umożliwia też podział instrukcji na kilka wierszy. Kompilator języka C# szuka średnika oznaczającego koniec instrukcji. Na listingu 1.7 przedstawiona wcześniej instrukcja `WriteLine()` z programu `HelloWorld` jest rozdzielona między kilka wierszy.

Listing 1.7. Podział jednej instrukcji na kilka wierszy

```
System.Console.WriteLine(  
    "Witaj. Nazywam się Inigo Montoya.");
```

■ ZAGADNIENIE DLA POCZĄTKUJĄCYCH

Czym są odstępy?

Odstęp (ang. *whitespace*) to połączenie jednego lub więcej przyległych znaków formatujących, takich jak tabulacja, spacja lub znak nowego wiersza. Usunięcie wszystkich odstępów między słowami ma oczywiście wpływ na kod, podobnie jak dodanie odstępów w łańcuchach znaków ujętych w cudzysłów.

Odstępy

Średnik umożliwia kompilatorowi języka C# ignorowanie odstępów w kodzie. Oprócz kilku wyjątków język C# pozwala programistom wstawiać odstępy w kodzie bez zmieniania jego działania. Na listingach 1.6 i 1.7 nie miało znaczenia, czy znak nowego wiersza znajdował się w instrukcji, czy pomiędzy instrukcjami. Nie miało to wpływu na plik wykonywalny wygenerowany przez kompilator.

Programiści często stosują odstępy do tworzenia wcięć zwiększających czytelność kodu. Przyjrzyj się dwóm wersjom programu `HelloWorld` przedstawionym na listingach 1.8 i 1.9. Choć te dwa przykłady wyglądają inaczej niż pierwotna wersja programu, kompilator języka C# traktuje wszystkie zapisy jako identyczne.

Listing 1.8. Formatowanie bez wcięć

```
class HelloWorld  
{  
    static void Main()  
    {  
        System.Console.WriteLine("Witaj, Inigo Montoya");  
    }  
}
```

Listing 1.9. Usunięcie odstępów

```
class HelloWorld{static void Main()  
{System.Console.WriteLine("Witaj, Inigo Montoya");}}
```

ZAGADNIENIE DLA POCZĄTKUJĄCYCH**Formatowanie kodu za pomocą odstępów**

Dodawanie wcięć w kodzie za pomocą odstępów jest ważne, ponieważ zwiększa czytelność. Gdy zaczniesz pisać programy, powinieneś stosować się do sprawdzonych standardów i konwencji, by poprawić przejrzystość kodu.

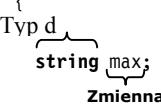
Konwencje stosowane w tej książce polegają na umieszczaniu nawiasów klamrowych w odrębnych wierszach i dodawaniu wcięć dla kodu pomiędzy takimi nawiasami. Jeśli pierwsza para nawiasów klamrowych obejmuje następną taką parę, cały kod w drugiej parze też należy wyróżnić wcięciem.

Nie jest to jednolity standard obowiązujący w języku C#, a tylko preferencja stylistyczna.

Korzystanie ze zmiennych

Po zapoznaniu się z bardzo prostym programem w języku C# pora przejść do deklarowania zmiennej lokalnej. Gdy zmienna jest zadeklarowana, można przypisać do niej wartość, zastąpić tę wartość nową wartością i wykorzystać zmienną w obliczeniach, danych wyjściowych itd. Nie można natomiast zmienić typu danych zmiennej. Na listingu 1.10 fragment string max to deklaracja zmiennej.

Listing 1.10. Deklarowanie zmiennej i przypisywanie do niej wartości

```
class miracleMax  
{  
    static void Main()  
    {  
        Typ d  
          
        string max;  
        Zmienna  
        max = "Dobrej zabawy w trakcie szтурmowania zamku!";  
        System.Console.WriteLine(max);  
    }  
}
```

ZAGADNIENIE DLA POCZĄTKUJĄCYCH**Zmienne lokalne**

Zmienna to nazwa wskazująca wartość. Ta wartość może się zmieniać. Określenie *lokalna* oznacza, że programista **zadeklarował** zmienną w metodzie.

Deklaracja zmiennej powoduje jej zdefiniowanie. Aby zadeklarować zmienną, należy:

- określić typ danych wskazywanych przez zmienną,
- określić identyfikator (nazwę) zmiennej.



Typy danych

Na listingu 1.10 zadeklarowana jest zmienna, której typ danych to `string`. Inne często używane w tym rozdziale typy danych to `int` i `char`.

- Typ `int` w języku C# służy do przechowywania 32-bitowych liczb całkowitych.
- Typ `char` to typ znakowy. Ma pojemność 16 bitów, co wystarcza do zapisania znaków Unicode (z wyjątkiem par znaków zastępczych).

W następnym rozdziale te i inne często spotykane typy danych są omówione bardziej szczegółowo.

■ ZAGADNIENIE DLA POCZĄTKUJĄCYCH

Czym jest typ danych?

Typ określany w deklaracji zmiennej to **typ danych** (lub typ obiektu). Typ danych, lub po prostu **typ**, określa obiekty o wspólnych cechach i działaniu. Na przykład *zwierzę* to typ. Określa on wszystkie obiekty (małpy, guźce, dziobaki itd.) o zwierzęcych cechach (wielokomórkowce, mogą się przemieszczać itd.). Podobnie w językach programowania typ to definicja grupy obiektów o podobnych cechach.



Deklarowanie zmiennej

Na listingu 1.10 `string max` to deklaracja zmiennej typu `string` o nazwie `max`. W jednej instrukcji można zadeklarować kilka zmiennych. W tym celu należy jednokrotnie podać typ danych i rozdzielić poszczególne identyfikatory przecinkami. Taką deklarację przedstawia listing 1.11.

Listing 1.11. Deklarowanie dwóch zmiennych w jednej instrukcji

```
string message1, message2;
```

Ponieważ instrukcje deklarujące wiele zmiennych umożliwiają programistom podanie typu danych tylko raz, wszystkie zadeklarowane tak zmienne są tego samego typu.

W języku C# nazwa zmiennej może się rozpoczynać literą lub podkreśnięciem (`_`), po czym następuje dowolna liczba liter, cyfr i (lub) podkreseń. Zwyczajowo nazwy zmiennych lokalnych są zapisywane w notacji Wielbłąda (pierwsze litery wszystkich słów oprócz pierwszego są wielkie) i nie obejmują podkreseń.

■ Wskazówka

STOSUJ notację Wielbłąda w nazwach zmiennych lokalnych.

Przypisywanie wartości do zmiennej

Po zadeklarowaniu zmiennej lokalnej trzeba przypisać do niej wartość. Dopiero potem możliwy jest odczyt wartości zmiennej. Jednym ze sposobów ustawiania wartości jest użycie **operatora =**, nazywanego **prostym operatorem przypisania**. Operatory to symbole używane do określania zadania, jakie kod ma wykonać. Listing 1.12 pokazuje, jak zastosować operatory przypisania do określenia łańcuchów znaków wskazywanych za pomocą zmiennych `miracleMax` i `valerie`.

Listing 1.12. Modyfikowanie wartości zmiennej

```
class StormingTheCastle
{
    static void Main()
    {
        string valerie;
        string miracleMax = "Udanej zabawy w trakcie szтурmowania zamku!";

        valerie = "Myślisz, że to zadziała?";

        System.Console.WriteLine(miracleMax);
        System.Console.WriteLine(valerie);

        max = "Potrzebny byłby cud.";
        System.Console.WriteLine(miracleMax);
    }
}
```

W tym listingu zwróć uwagę na to, że wartość można przypisać do zmiennej w deklaracji (tak jak dla zmiennej `miracleMax`) lub później, w odrębnej instrukcji (tak jak dla zmiennej `valerie`). Przypisaną wartość zawsze należy umieszczać po prawej stronie.

Uruchomienie skompilowanego programu powoduje wygenerowanie informacji pokazanych w danych wyjściowych 1.3.

DANE WYJŚCIOWE 1.3.

```
>dotnet run
Udanej zabawy w trakcie szтурmowania zamku!
Myślisz, że to zadziała?
Potrzebny byłby cud.
```

W tym przykładzie bezpośrednio pokazane jest polecenie `dotnet run`. W dalszych danych wyjściowych ten wiersz jest pomijany, chyba że instrukcja potrzebna do wykonania programu ma jakieś wyjątkowe cechy.

Język C# wymaga, by zmienne lokalne przed ich odczytem zostały uznane przez kompilator za zmienne z „definitywnie przypisaną wartością”. Ponadto operacja przypisania zwraca wartość. Dlatego język C# umożliwia połączenie dwóch przypisań w jednej instrukcji, co przedstawia listing 1.13.

Listing 1.13. Operacja przypisania zwraca wartość, którą można ponownie przypisać

```
class StormingTheCastle
{
    static void Main()
    {
        // ...
        string requirements, miracleMax;
        requirements = miracleMax = "Potrzebny byłby cud.";
        // ...
    }
}
```

Korzystanie ze zmiennej

Po przypisaniu można oczywiście wskazywać wartość za pomocą identyfikatora zmiennej. Dlatego gdy używasz zmiennej `miracleMax` w instrukcji `System.Console.WriteLine(miracleMax)`, program wyświetla w konsoli tekst Udanej zabawy w trakcie szturmowania zamku!, czyli wartość zmiennej `miracleMax`. Modyfikacja wartości zmiennej `miracleMax` i wywołanie tej samej instrukcji `System.Console.WriteLine(miracleMax)` spowoduje wyświetlenie nowej wartości zmiennej `miracleMax` — Potrzebny byłby cud.

■ ■ ■ ZAGADNIENIE DLA ZAAWANSOWANYCH

Wartość łańcuchów znaków jest niezmienna

Wszystkie dane typu `string`, czy to literały znakowe, czy inne wartości, są niezmienne (niemodyfikowalne). Nie można na przykład zmienić łańcucha znaków z `Mamy ciepły dzień` na `Mamy ciemny dzień`. Taka zmiana wymaga, by zmienna zaczęła wskazywać nową lokalizację w pamięci. Nie można zmodyfikować danych pierwotnie wskazywanych przez zmienną.

Dane wejściowe i wyjściowe w konsoli

W tym rozdziale zastosowano już wielokrotnie instrukcję `System.Console.WriteLine`, by wyświetlać tekst w konsoli. Oprócz mechanizmu wyświetlania danych potrzebna jest też możliwość pobierania danych wprowadzanych przez użytkowników.

Pobieranie danych wejściowych z konsoli

Jednym ze sposobów na pobieranie tekstu wprowadzonego w konsoli jest użycie metody `System.Console.ReadLine()`. Wstrzymuje ona wykonywanie programu, dzięki czemu użytkownik może wprowadzić znaki. Gdy użytkownik wcisnie klawisz `Enter` i doda w ten sposób znak nowego wiersza, program wznowi działanie. Dane wyjściowe, czyli **zwracana wartość** metody `System.Console.ReadLine()`, to wprowadzony łańcuch znaków. Przyjrzyj się teraz listingowi 1.14 i powiązanym z nim danym wyjściowym 1.4.

Listing 1.14. Korzystanie z metody System.Console.ReadLine()

```
class HeyYou
{
    static void Main()
    {
        string firstName;
        string lastName;

        System.Console.WriteLine("Hej, ty!");

        System.Console.Write("Podaj imię: ");
        firstName = System.Console.ReadLine();

        System.Console.Write("Podaj nazwisko: ");
        lastName = System.Console.ReadLine();
    }
}
```

DANE WYJŚCIOWE 1.4.

```
Hej, ty!
Podaj imię: Inigo
Podaj nazwisko: Montoya
```

Po każdej prośbie o dane program używa metody `System.Console.ReadLine()`, by pobrać tekst wprowadzony przez użytkownika i przypisać go do odpowiedniej zmiennej. Po drugim przypisaniu wartości zwrotnej przez metodę `System.Console.ReadLine()` zmienna `firstName` wskazuje wartość `Inigo`, a zmienna `lastName` prowadzi do wartości `Montoya`.

ZAGADNIENIE DLA ZAAWANSOWANYCH**Metoda System.Console.Read()**

Oprócz metody `System.Console.ReadLine()` istnieje też metoda `System.Console.Read()`. Jednak typ danych wartości zwracanej przez metodę `System.Console.Read()` to liczba całkowita. Odpowiada ona wartości wczytanego znaku lub, jeśli żaden znak nie jest dostępny, ma wartość `-1`. Aby pobrać znak, trzeba najpierw zrzutować liczbę całkowitą na typ znakowy, co ilustruje listing 1.15.

Listing 1.15. Korzystanie z metody System.Console.Read()

```
int readValue;
char character;
readValue = System.Console.Read();
character = (char) readValue;
System.Console.Write(character);
```

Metoda `System.Console.Read()` nie zwraca danych wejściowych do czasu wciśnięcia przez użytkownika klawisza `Enter`. Do tego momentu program nie zacznie przetwarzanie znaków (nawet jeśli użytkownik wpisze ich dużą liczbę).

Początek
2.0Koniec
2.0

W C# 2.0 i nowszych wersjach języka można stosować metodę `System.Console.ReadKey()`, która (w odróżnieniu od metody `System.Console.Read()`) zwraca dane wejściowe po każdym wciśnięciu klawisza. Metoda `ReadKey()` umożliwia programistom przechwytywanie operacji wciśnięcia klawisza i wykonywanie zadań takich jak sprawdzanie poprawności i przyjmowanie samych cyfr.

Wyświetlanie danych wyjściowych w konsoli

Kod z listingu 1.14 prosi użytkownika o imię i nazwisko. Używana jest przy tym metoda `System.Console.Write()`, a nie `System.Console.WriteLine()`. Zamiast dodawać znak nowego wiersza po wyświetleniu tekstu, metoda `System.Console.Write()` pozostawia kurSOR w tej samej linii. Dzięki temu tekst wprowadzany przez użytkownika znajduje się w tym samym wierszu co prośba o dane wejściowe. Dane wyjściowe z listingu 1.14 pokazują efekt użycia metody `System.Console.Write()`.

Początek
6.0

Następny krok polega na wyświetleniu w konsoli wartości pobranych za pomocą metody `System.Console.ReadLine()`. Program z listingu 1.16 wyświetla imię i nazwisko użytkownika. Jednak tym razem zamiast stosować użytą wcześniej metodę `System.Console.WriteLine()`, wprowadzono pewną modyfikację. Wykorzystano tu wprowadzoną w C# 6.0 **interpolację łańcuchów znaków**. Zwróć uwagę na znak dolara poprzedzający literał tekstowy w wywołaniu `Console.WriteLine`. Znak ten wskazuje na zastosowanie interpolacji łańcuchów znaków. Efekt przedstawiają dane wyjściowe 1.5.

Listing 1.16. Formatowanie z wykorzystaniem interpolacji łańcuchów znaków

```
class HeyYou
{
    static void Main()
    {
        string firstName;
        string lastName;

        System.Console.WriteLine("Hej, ty!");

        System.Console.Write("Podaj imię: ");
        firstName = System.Console.ReadLine();

        System.Console.Write("Podaj nazwisko: ");
        lastName = System.Console.ReadLine();

        System.Console.WriteLine(
            $"Twoje imię i nazwisko to { firstName } { lastName }.");
    }
}
```

DANE WYJŚCIOWE 1.5.

```
Hej, ty!
Podaj imię: Inigo
Podaj nazwisko: Montoya

Twoje imię i nazwisko to Inigo Montoya.
```

Zamiast wyświetlać fragment Twoje imię i nazwisko, a następnie w kolejnej instrukcji Write podawać wartość zmiennej firstName, w trzeciej instrukcji Write dodawać spację, a w jeszcze następnej zwracać wartość zmiennej lastName, na listingu 1.16 całe dane wyjściowe wyświetlono za pomocą mechanizmu interpolacji łańcuchów znaków dostępnego w C# 6.0. Kompilator za pomocą tego mechanizmu interpretuje zawartość nawiasów klamrowych w łańcuchach znaków jako obszar, w którym można osadzać kod (wyrażenia) przetwarzany przez kompilator i przekształcany na tekst. Zamiast wykonywać osobno wiele fragmentów kodu i na końcu łączyć wyniki w łańcuch znaków, za pomocą omawianego mechanizmu można osiągnąć ten sam efekt w jednym kroku. Dzięki temu kod jest bardziej zrozumiały.

W wersjach starszych niż C# 6.0 stosowano inne podejście — **formatowanie złożone**. Polegało ono na tym, że w kodzie trzeba było najpierw podać **łańcuch znaków formatowania**, aby zdefiniować format danych wyjściowych. Tę technikę przedstawia listing 1.17.

Listing 1.17. Formatowanie złożone w metodzie System.Console.WriteLine()

```
class HeyYou
{
    static void Main()
    {
        string firstName;
        string lastName;

        System.Console.WriteLine("Hej, ty!");

        System.Console.Write("Podaj imię: ");
        firstName = System.Console.ReadLine();

        System.Console.Write("Podaj nazwisko: ");
        lastName = System.Console.ReadLine();

        System.Console.WriteLine(
            "Twoje imię i nazwisko to {0} {1}.", firstName, lastName);
    }
}
```

W tym przykładzie łańcuch znaków formatowania to "Twoje imię i nazwisko to {0} {1}.". Znajdują się tu dwa indeksowane symbole zastępcze służące do wstawiania danych do łańcucha znaków. Wszystkie symbole zastępcze zawierają liczby określające pozycje argumentów występujących po łańcuchu znaków formatowania.

Zauważ, że indeksowanie rozpoczyna się od zera. Każdy wstawiany argument (**formatowany element**) jest podany po łańcuchu znaków formatowania. Kolejność argumentów wyznacza wartości ich indeksów. W tym przykładzie pierwszym argumentem po łańcuchu znaków formatowania jest zmienna firstName, dlatego jest ona powiązana z indeksem 0. Zmienna lastName odpowiada indeksowi 1.

Zwróć uwagę na to, że symbole zastępcze w łańcuchu znaków formatowania nie muszą być podawane po kolei. Na listingu 1.18 przedstawiono kolejność indeksowanych symboli zastępczych i dodano przecinek. Zmienia to sposób wyświetlania imienia i nazwiska (zobacz dane wyjściowe 1.6).

Listing 1.18. Przedstawianie indeksowanych symboli zastępczych i odpowiadających im wartości

```
System.Console.WriteLine("Twoje nazwisko i imię to {1}, {0}",
    firstName, lastName);
```

DANE WYJŚCIOWE 1.6.

```
Hej, ty!
Wprowadź imię: Inigo
Wprowadź nazwisko: Montoya
```

Twoje nazwisko i imię to Montoya, Inigo.

Symbol zastępcze w łańcuchach znaków formatowania nie muszą więc występować obok siebie. Ponadto jeden symbol zastępczy można wykorzystać w takim łańcuchu wielokrotnie. Można też pominąć symbol zastępczy odpowiadający danemu argumentowi. Nie jest jednak dozwolone używanie symboli zastępczych, dla których nie istnieje powiązany argument.

Uwaga

Ponieważ kod wykorzystujący interpolację łańcuchów znaków w stylu języka C# 6.0 jest prawie zawsze łatwiejszy do zrozumienia niż kod oparty na złożonych łańcuchach znaków, dalej w książce domyślnie stosowana jest właśnie interpolacja.

Komentarze

W tym podrozdziale zmodyfikujesz program z listingu 1.17 i dodasz do niego komentarze. Nie zmieni to sposobu wykonywania programu. Dodanie komentarzy w kodzie ułatwia zrozumienie fragmentów, które nie są intuicyjnie oczywiste. Listing 1.19 przedstawia nowy kod, a dane wyjściowe 1.7 zawierają powiązane dane wyjściowe.

Listing 1.19. Dodawanie komentarzy w kodzie

```
class CommentSamples
{
    static void Main()
    {
        Komentarz jednowierszowy
        string firstName; // Zmienna do przechowywania imienia
        string lastName; // Zmienna do przechowywania nazwiska

        System.Console.WriteLine("Hej, ty!");

        Komentarz z ogranicznikami w instrukcji
        System.Console.Write /* Bez nowego wiersza */(
            "Podaj imię: ");
        firstName = System.Console.ReadLine();

        System.Console.Write /* Bez nowego wiersza */(
            "Podaj nazwisko: ");
        lastName = System.Console.ReadLine();

        /* Wyświetlanie powitania w konsoli
           z wykorzystaniem łańcucha znaków formatowania. */
        System.Console.WriteLine("Twoje imię i nazwisko to {0} {1}.",
            firstName, lastName);
    }
}
```

```
// To koniec listingu
// z kodem tego programu
}
}
```

DANE WYJŚCIOWE 1.7.

Hej, ty!
Podaj imię: **Inigo**
Podaj nazwisko: **Montoya**

Twoje imię i nazwisko to Inigo Montoya.

Mimo wstawionych komentarzy komplilacja i wykonanie nowego programu prowadzą do wygenerowania tych samych danych wyjściowych co wcześniejsza wersja kodu.

Programiści stosują komentarze w celu opisywania i wyjaśniania rozwijanego kodu — zwłaszcza gdy sama składnia jest mało zrozumiała lub gdy implementacja algorytmu jest niespotykana. Ponieważ komentarze są ważne tylko dla programisty czytającego kod, kompi-lator ignoruje je i generuje podzespółł, w którym nie ma żadnego śladu po komentarzach będących częścią pierwotnego kodu źródłowego.

W tabeli 1.2 przedstawiono cztery różne typy komentarzy stosowane w C#. W programie z listingu 1.19 wykorzystano dwa spośród tych typów.

Tabela 1.2. Typy komentarzy w języku C#

Typ komentarza	Opis	Przykład
Komentarze z ogranicznikami	Ukośnik, po którym następuje gwiazdka (*), oznacza początek komentarza z ogranicznikami. Aby zakończyć taki komentarz, dodaj gwiazdkę i ukośnik (*). Komentarze w tej postaci mogą się rozciągać na wiele wierszy pliku z kodem lub znajdować się w jednej linii. Gwiazdki występujące na początkach wierszy, ale między ogranicznikami, służą wyłącznie do formatowania kodu.	/* Komentarz */
Komentarze jednowierszowe	Komentarze można zadeklarować za pomocą ogranicznika w postaci dwóch ukośników (//). Komplilator traktuje cały tekst od ogranicznika do końca wiersza jako komentarz. Można też umieszczać komentarze jednowierszowe jeden pod drugim. Tak wygląda na przykład ostatni komentarz na listingu 1.19.	// Komentarz
XML-owe komentarze z ogranicznikami	Komentarze rozpoczynające się od ogranicznika /** i kończące sekwencją **/ to XML-owe komentarze z ogranicznikami. Mają one te same cechy co zwykłe komentarze z ogranicznikami, przy czym komplilator zamiast ignorować takie komentarze, może umieścić je w odrębnym pliku tekstowym [†] .	/** Komentarz **/
XML-owe komentarze jednowierszowe	Jednowierszowe komentarze XML-owe rozpoczynają się od sekwencji /// i ciągną do końca wiersza. Ponadto komplilator może zapisać takie komentarze w odrębnym pliku razem z XML-owymi komentarzami z ogranicznikami.	/// Komentarz

[†] XML-owe komentarze z ogranicznikami dodano dopiero w wersji C# 2.0, jednak ich składnia jest zgodna z wersją C# 1.0.

Bardziej kompletne omówienie komentarzy XML-owych znajdziesz w rozdziale 10., gdzie dokładniej opisano różne znaczniki XML-a.

W historii programowania był okres, gdy bogate komentarze były cechą charakterystyczną zdyscyplinowanych i doświadczonych programistów. Obecnie sytuacja się zmieniła. Teraz kod, który jest czytelny nawet bez komentarzy, jest ceniony bardziej niż kod wymagający objaśniania wykonywanych operacji. Jeśli programista stwierdza, że musi dodać komentarze, by wyjaśnić działanie danego bloku kodu, powinien się decydować raczej na przekształcenie kodu na bardziej zrozumiałą postać niż na pisanie komentarzy. Dołączanie komentarzy, w których opisane jest tylko to, co można łatwo wynioskować z kodu, wyłącznie zaśmieca kod, zmniejsza jego czytelność i zwiększa prawdopodobieństwo zdezaktualizowania się komentarzy (w sytuacji gdy zmodyfikowany zostanie kod, a komentarze pozostaną niezmienione).

■ ■ ■ Wskazówki

NIE stosuj komentarzy, jeśli nie zawierają informacji nieoczywistych dla osób innych niż autor kodu.

PRZEDKŁADAJ pisanie bardziej przejrzystego kodu nad dodawanie komentarzy w celu objaśniania skomplikowanych algorytmów.

■ ■ ■ ZAGADNIENIE DLA POCZĄTKUJĄCYCH

XML

XML (ang. *Extensible Markup Language*) to prosty i dający dużą swobodę format tekstowy, często stosowany w aplikacjach sieciowych i do przekazywania danych między aplikacjami. XML jest rozszerzalny, ponieważ w dokumentach XML umieszczone są **metadane**, czyli informacje opisujące dane. Oto przykładowy plik w formacie XML.

```
<?xml version="1.0" encoding="utf-8" ?>
<body>
  <book title="C# 8.0. Praktyczny podręcznik">
    <chapters>
      <chapter title="Wprowadzenie do języka C#" />
      <chapter title="Typy danych" />
      ...
    </chapters>
  </book>
</body>
```

Plik rozpoczyna się od nagłówka określającego wersję dokumentu XML i używane w nim kodowanie znaków. Dalej znajduje się jeden główny element book. Elementy rozpoczynają się od słowa zapisanego w nawiasie ostrym, na przykład <body>. Aby zakończyć element, należy umieścić w nawiasie ostrym to samo słowo poprzedzone ukośnikiem (na przykład </body>). Oprócz elementów XML obsługuje atrybuty. Przykładowy atrybut w XML-u to title="C# 8.0. Praktyczny podręcznik". Zauważ, że metadane (book title, chapter itd.) opisujące dane (C# 8.0. Praktyczny podręcznik, Typy danych) znajdują się w samym pliku XML. To podejście może prowadzić do powstawania długich plików, jednak ma tę zaletę, że dane obejmują opis pomocny przy ich interpretacji.

Wykonywanie kodu w środowisku zarządzanym iplatforma CLI

Procesor nie potrafi bezpośrednio interpretować podzespołów. Podzespoły składają się głównie z kodu w pomocniczym języku **CIL** (ang. *Common Intermediate Language*), nazywanym też w skrócie **IL**¹¹. Kompilator języka C# przekształca plik źródłowy z kodem w języku C# na wspomniany język pośredni. Aby przekształcić kod z języka CIL na zrozumiały dla procesora **kod maszynowy**, trzeba zrobić dodatkowy krok, co zwykle ma miejsce na etapie wykonywania programu. Używany jest przy tym ważny element procesu wykonywania programów języka C# — system **VES** (ang. *Virtual Execution System*). System VES, potocznie nazywany **środowiskiem uruchomieniowym** (ang. *runtime*), kompiluje na żądanie kod CIL. Jest to proces **kompilacji JIT** (ang. *just-in-time*). Kod wykonywany w kontekście agenta takiego jak środowisko uruchomieniowe to **kod zarządzany**. Proces działania kodu pod kontrolą środowiska uruchomieniowego to **wykonywanie w środowisku zarządzanym**. Kod jest „zarządzany”, ponieważ środowisko uruchomieniowe kontroluje wiele aspektów działania programu. Środowisko steruje na przykład przydziałem pamięci, zabezpieczeniami i komplikacją JIT. Kod, który nie wymaga do pracy środowiska uruchomieniowego, to **kod natywny (kod niezarządzany)**.

Specyfikacja środowiska uruchomieniowego jest częścią większej specyfikacji **CLI** (ang. *Common Language Infrastructure*)¹². CLI to międzynarodowy standard obejmujący specyfikację wielu zagadnień. Oto wybrane z nich:

- system VES (środowisko uruchomieniowe),
- język CIL,
- system typów **CTS** (ang. *Common Type System*) ułatwiający współdziałanie między językami,
- wskazówki na temat pisania bibliotek dostępnych w językach zgodnych z CLI (te wskazówki znajdziesz w specyfikacji **CLS** — ang. *Common Language Specification*),
- metadane umożliwiające działanie wielu usług opisanych w CLI (w tym specyfikacje układu lub formatu plików podzespołów).

Wykonywanie kodu w kontekście środowiska uruchomieniowego pozwala korzystać z wielu usług i funkcji, dzięki czemu programiści nie muszą samodzielnie ich pisać. Oto przykładowe dostępne mechanizmy:

¹¹ Trzecią nazwą języka CIL jest **MSIL** (ang. *Microsoft IL*). W tej książce używane jest określenie **CIL**, ponieważ to pojęcie występuje w standardzie CLI. W rozmowach między użytkownikami języka C# najczęściej stosowana jest nazwa IL, gdyż osoby te przyjmują, że IL oznacza CIL, a nie inne rodzaje języków pośrednich.

¹² J. Miller, S. Ragsdale, *The Common Language Infrastructure Annotated Standard*, Addison-Wesley, Boston 2004.

- *Współdziałanie języków.* Możliwe jest współdziałanie kodu źródłowego napisanego w różnych językach. Jest to możliwe, ponieważ kompilatory przekształcają wszystkie języki źródłowe na ten sam język pośredni (CIL).
- *Bezpieczeństwo ze względu na typ.* Ten mechanizm sprawdza konwersję typów i gwarantuje, że przeprowadzane będą tylko konwersje między zgodnymi typami. To pomaga zapobiegać jednej z najważniejszych luk bezpieczeństwa — przepełnieniu bufora.
- *Zabezpieczenie dostępu do kodu.* Używane są certyfikaty informujące, że kod z podzespołu programisty ma uprawnienia do działania na danym komputerze.
- *Przywracanie pamięci.* Środowisko zarządza pamięcią i automatycznie zwalnia pamięć zajętą wcześniej przez środowisko uruchomieniowe.
- *Przenośność między systemami.* Potencjalnie możliwe jest wykonywanie tego samego podzespołu w różnych systemach operacyjnych. Obowiązuje przy tym pewne ograniczenie — nie można używać bibliotek zależnych od systemu. Dlatego, podobnie jak w Javie, mogą występować pewne osobliwości związane z systemami. Programista musi rozwiązać łączące się z tym problemy.
- *Biblioteka BCL.* BCL to rozbudowana baza kodu, na którym programiści mogą polegać we wszystkich wersjach platformy .NET. Dzięki temu programiści nie muszą samodzielnie pisać kodu dostępnego w bibliotece.

■ **Uwaga**

W tym podrozdziale znajdziesz krótki przegląd platformy CLI, co pozwoli Ci zaznajomić się z kontekstem wykonywania programów w języku C#. Dostępny jest tu także przegląd pojęć używanych w książce. Platformie CLI i jej znaczeniu dla programistów języka C# jest poświęcony rozdział 24. Choć kończy on książkę, nie jest zależny od wcześniejszych rozdziałów. Dlatego jeśli chcesz zapoznać się z platformą CLI, możesz w dowolnym momencie przejść do wspomnianego rozdziału.

Język CIL i narzędzie ILDASM

We wprowadzeniu do tego podrozdziału wspomniano, że kompilator języka C# przekształca kod z języka C# na kod CIL, a nie na kod maszynowy. Procesor potrafi bezpośrednio zrozumieć kod maszynowy, natomiast kod CIL przed jego wykonaniem przez procesor trzeba przekształcić. Gdy dostępny jest podzespoł, można wyświetlić kod CIL przy użyciu dezasemblera języka CIL. W ten sposób można przekształcić podzespoł na jego reprezentację w języku CIL. Do określenia dezasemblera języka CIL zwykle używana jest nazwa opracowanego przez Microsoft pliku *ILDASM* (która sama pochodzi od zwrotu *IL Disassembler*). To narzędzie dezasembliuje program lub bibliotekę klas i wyświetla kod CIL wygenerowany przez kompilator języka C#.

Dane wyjściowe dezasemblieracji podzespołu z platformy .NET są znacznie łatwiejsze do zrozumienia niż kod maszynowy. U wielu programistów może to budzić obawy, ponieważ

można łatwo zdekompilować program i zrozumieć użyte algorytmy mimo tego, że kod źródłowy nie jest bezpośrednio udostępniany. Podobnie jak w każdym programie (niezależnie od tego, czy ma on działać w platformie CLI) jedynym pewnym sposobem na uniemożliwienie dezasemblieracji jest całkowite zablokowanie dostępu do skompilowanego programu (można na przykład uruchamiać program w witrynie, zamiast instalować go na maszynach użytkowników). Jeśli jednak chcesz tylko utrudnić dostęp do kodu źródłowego, możesz skorzystać z jednego z kilku narzędzi do zaciemniania kodu. Te narzędzia przekształcają kod w języku IL w taki sposób, by działał tak samo, ale był znacznie trudniejszy do zrozumienia. Dzięki temu mało doświadczeni programiści nie uzyskają dostępu do kodu, a dekomplikacja podzespołu do zrozumiałej postaci będzie dużo trudniejsza i żmudniejsza. Jeśli program nie wymaga ścisłego zabezpieczenia algorytmów, zwykle wystarczy zastosować narzędzia do zaciemniania kodu.

ZAGADNIENIE DLA ZAAWANSOWANYCH

Dane wyjściowe w języku CIL z programu HelloWorld.exe

Polecenie stosowane do uruchomienia dezasemblera języka CIL zależy od używanej implementacji interfejsu CLI. Instrukcje dla platformy .NET Core znajdziesz na stronie <http://itl.tc/ildasm>. Listing 1.20 przedstawia kod w języku CIL wygenerowany przez narzędzie ILDASM.

Listing 1.20. Przykładowe dane wyjściowe w języku CIL

```
.assembly extern System.Runtime
{
    .publickeytoken = ( B0 3F 5F 7F 11 D5 0A 3A )
    .ver 4:2:0:1
}

.assembly extern System.Console
{
    .publickeytoken = ( B0 3F 5F 7F 11 D5 0A 3A )
    .ver 4:1:0:1
}

.assembly 'HelloWorld'
{
    .custom instance void [System.Runtime]System.Runtime.
    ↳CompilerServices.CompilationRelaxationsAttribute::ctor(int32) = ( 01 00 08
    ↳00 00 00 00 )
    .custom instance void [System.Runtime]System.Runtime.
    ↳CompilerServices.RuntimeCompatibilityAttribute::ctor() = ( 01 00 01 00 54
    ↳02 16 57 72 61 70 4E 6F 6E 45 78 63 65 70 74 69 6F 6E 54 68 72 6F 77 73 01 )
    .custom instance void [System.Runtime]System.Runtime.
    ↳Versioning.TargetFrameworkAttribute::ctor(string) = ( 01 00 18 2E 4E 45 54
    ↳43 6F 72 65 41 70 70 2C 56 65 72 73 69 6F 6E 3D 76 32 2E 30 01 00 54 0E 14
    ↳46 72 61 6D 65 77 6F 72 6B 44 69 73 70 6C 61 79 4E 61 6D 65 00 ).
    .custom instance void [System.Runtime]System.
    ↳Reflection.AssemblyCompanyAttribute::ctor(string) = ( 01 00 0A 48 65 6C 6C
    ↳6F 57 6F 72 6C 64 00 00 )
    .custom instance void [System.Runtime]System.
```

```
↳Reflection.AssemblyConfigurationAttribute::ctor(string) = ( 01 00 05 44 65
↳62 75 67 00 00 )
    .custom instance void [System.Runtime]System.
↳Reflection.AssemblyDescriptionAttribute::ctor(string) = ( 01 00 13 50 61 63
↳6B 61 67 65 20 44 65 73 63 72 69 70 74 69 6F 6E 00 00 )
    .custom instance void [System.Runtime]System.
↳Reflection.AssemblyFileVersionAttribute::ctor(string) = ( 01 00 07 31 2E 30
↳2E 30 2E 30 00 00 )
    .custom instance void [System.Runtime]System.
↳Reflection.AssemblyInformationalVersionAttribute::ctor(string) = ( 01 00 05
↳31 2E 30 2E 30 00 00 )
    .custom instance void [System.Runtime]System.
↳Reflection.AssemblyProductAttribute::ctor(string) = ( 01 00 0A 48 65 6C 6C
↳6F 57 6F 72 6C 64 00 00 )
    .custom instance void [System.Runtime]System.
↳Reflection.AssemblyTitleAttribute::ctor(string) = ( 01 00 0A 48 65 6C 6C 6F
↳57 6F 72 6C 64 00 00 )
    .hash algorithm 0x00008004
    .ver 1:0:0:0
}

.module 'HelloWorld.dll'
// MVID: {05b2d1a7-c150-4f20-bd96-c065e4f97a31}
.imagebase 0x00400000
.file alignment 0x00000200
.stackreserve 0x00100000
.subsystem 0x0003 // WindowsCui
.corflags 0x00000001 // ILOnly

.class private auto ansi beforefieldinit HelloWorld.Program extends [System.
↳Runtime]System.Object
{
    .method private hidebysig static void Main(string[] args) cil managed
    {
        .entrypoint
        // Code size 13
        .maxstack 8
        IL_0000: nop
        IL_0001: ldstr "Witaj. Nazywam się Inigo Montoya."
        IL_0006: call void [System.Console]System.Console::WriteLine(string)
        IL_000b: nop
        IL_000c: ret
    } // End of method System.Void HelloWorld.Program::Main(System.String[])
    .method public hidebysig specialname rtspecialname instance void .ctor() cil managed
    {
        // Code size 8
        .maxstack 8
        IL_0000: ldarg.0
        IL_0001: call instance void [System.Runtime]System.Object::..ctor()
        IL_0006: nop
        IL_0007: ret
    } // End of method System.Void HelloWorld.Program::ctor()
} // End of class HelloWorld.Program
```

Na początku listingu znajduje się manifest. Obejmuje on nie tylko pełną nazwę dezasemblowanego modułu (*HelloWorld.dll*), ale też listę wszystkich potrzebnych modułów i podzespołów oraz ich wersji.

Prawdopodobnie najciekawsze jest tu to, że na podstawie tego listingu można dużo łatwiej zrozumieć działanie programu niż w sytuacji, gdy trzeba odczytać i zinterpretować kod maszynowy (w asemblerze). W przedstawionym kodzie pojawia się bezpośrednie wywołanie metody `System.Console.WriteLine()`. Na listingu z kodem CIL znajduje się też wiele mniej istotnych informacji, jeśli jednak programista chce zrozumieć wewnętrzne działanie modułu w języku C# (lub dowolnego programu opartego na platformie CLI) bez dostępu do pierwotnego kodu źródłowego, może to zrobić stosunkowo łatwo (chyba że autor zastosował narzędzie do zaciemniania kodu). Dostępnych jest nawet kilka bezpłatnych narzędzi (na przykład Reflector firmy Red Gate, ILSpy, JustDecompile, dotPeek i CodeReflect), które automatycznie dekompilują kod CIL do kodu C#.

Różne wersje platformy .NET

Wcześniej pokrótko wspomniałem, że istnieje wiele wersji platformy .NET. Ich duża liczba wynika głównie z chęci udostępnienia implementacji platformy .NET w różnych systemach operacyjnych, a nawet na różnych platformach sprzętowych. W tabeli 1.3 zostały wymienione najważniejsze wersje.

Tabela 1.3. Najważniejsze wersje platformy .NET

Wersje	Opis
.NET Core	W pełni międzysystemowa i otwarta platforma .NET udostępniająca wysoce modułowy zestaw interfejsów API zarówno dla serwera, jak i dla aplikacji uruchamianych w wierszu poleceń.
Microsoft .NET Framework	Pierwsza wersja platformy .NET, powoli wypierana przez .NET Core.
Xamarin	Implementacja platformy .NET przeznaczona dla urządzeń mobilnych. Działa w systemach iOS i Android. Umożliwia tworzenie aplikacji mobilnych z użyciem jednej wersji kodu bazowego, a jednocześnie zapewnia dostęp do natywnych interfejsów API obsługiwanych systemów.
Mono	Najstarsza otwarta implementacja platformy .NET. Na jej podstawie powstały Xamarin i Unity. Na potrzeby dalszego rozwoju platformy Mono została zastąpiona wersją .NET Core.
Unity	Wielosystemowy silnik obsługi gier służący do tworzenia gier na konsole, komputery PC, urządzenia mobilne, a nawet przeglądarki. Wersja Unity jest pierwszą publiczną implementacją, która obsługuje rzeczywistość rozszerzoną i gogle Hololens Microsoftu.

Wszystkie przykłady z tej książki działają co najmniej na platformach .NET Core i Microsoft .NET Framework (chyba że napisano inaczej). Jednak ponieważ to w wersję .NET Core inwestowana jest większość prac nad przyszłością platformy .NET, przykładowy kod źródłowy dołączony do książki (dostępny na stronie <https://IntelliTect.com/EssentialCSharp> i w witrynie wydawnictwa Helion) jest domyślnie skonfigurowany pod kątem współdziałania z .NET Core.

■ Uwaga

W tej książce nazwa *platforma .NET* oznacza platformę, na podstawie której tworzone są implementacje .NET. Z kolei określenie *Microsoft .NET Framework* to konkretna implementacja platformy .NET działająca tylko w systemie Windows i *po raz pierwszy* udostępniona przez Microsoft w roku 2001.

Interfejsy API

Wszystkie metody (lub, bardziej ogólnie, składowe) występujące w określonym typie danych, na przykład w typie `System.Console`, tworzą **interfejs API** (ang. *Application Programming Interface*). Interfejs API definiuje, w jaki sposób program komunikuje się z komponentem. Zagadnienie to nie ogranicza się do jednego typu danych. Zestaw wszystkich interfejsów API dla zbioru typów danych tworzy interfejs API dla grupy komponentów. Na przykład w platformie .NET wszystkie typy (i składowe z tych typów) podzespołu tworzą interfejs API podzespołu. Podobnie dla grupy podzespołów, na przykład z platformy .NET Core lub Microsoft .NET Framework, dostępny jest wspólny większy interfejs API. Często większa grupa interfejsów API nazywana jest **platformą**; stąd nazwa *platforma .NET* oznaczająca interfejsy API udostępniane przez wszystkie podzespoły z platformy .NET Core lub Microsoft .NET Framework. Interfejs API obejmuje zestaw interfejsów i protokołów (instrukcji) umożliwiających pisanie kodu z wykorzystaniem grupy komponentów. W platformie .NET protokoły określają reguły wykonywania dostępnych w niej podzespołów.

Wersje języka C# i platformy .NET

Ponieważ cykl rozwoju platformy .NET różni się od cyklu rozwoju języka C#, poszczególne wersje platformy .NET i powiązane z nimi wersje języka C# mają różne numery. Dlatego jeśli kompilujesz kod na przykład za pomocą kompilatora z wersji C# 5.0, domyślnie będzie on używał wersji 4.6 platformy .NET. W tabeli 1.4 przedstawiono krótki przegląd wersji języka C# i platformy .NET (Microsoft .NET Framework i .NET Core).

Tabela 1.4. Wersje języka C# i platformy .NET

Wersje	Opis
C# 1.0 i Microsoft .NET Framework 1.0/1.1 (środowiska Visual Studio 2002 i 2003)	Pierwsza wersja języka C#. Język zbudowano od podstaw pod kątem programowania w platformie .NET.
C# 2.0 i Microsoft .NET Framework 2.0 (środowisko Visual Studio 2005)	Dodano typy generyczne do języka C# i biblioteki obsługujące typy generyczne do platformy Microsoft .NET Framework 2.0.
Microsoft .NET Framework 3.0	Wprowadzono dodatkowe interfejsy API do obsługi komunikacji rozproszonej (technologia WCF — ang. <i>Windows Communication Foundation</i>), tworzenia klientów z bogatą warstwą prezentacji (technologia WPF — ang. <i>Windows Presentation Foundation</i>), przepływu pracy (technologia WF — ang. <i>Windows Workflow</i>) i uwierzytelniania w sieci (technologia Cardspaces).
Język C# 3.0 i platforma Microsoft .NET Framework 3.5 (środowisko Visual Studio 2008)	Dodano obsługę technologii LINQ — znaczącego usprawnienia w interfejsach API używanych do programowania z wykorzystaniem kolekcji. W platformie Microsoft .NET Framework 3.5 wprowadzono biblioteki rozszerzające istniejące interfejsy API, tak by możliwe było działanie technologii LINQ.
Język C# 4.0 i platforma Microsoft .NET Framework 4 (środowisko Visual Studio 2010)	Dodano obsługę dynamicznego określania typów oraz wprowadzono znaczące usprawnienia w interfejsie API służącym do pisania programów wielowątkowych, które wykorzystują możliwości komputerów wieloprocesorowych i wielordzeniowych.
Język C# 5.0 i platforma Microsoft .NET Framework 4.5 (środowisko Visual Studio 2012) i integracja ze środowiskiem WinRT	Dodano obsługę asynchronicznego wywoływanie metod bez jawnego rejestrowania wywołań zwrotnych w delegatach. Ponadto w platformie dodano obsługę współdziałania ze środowiskiem WinRT (ang. <i>Windows Runtime</i>).
Język C# 6.0 i platforma Microsoft .NET Framework 4.6 oraz .NET Core 1.X (środowisko Visual Studio 2015)	Dodano interpolację łańcuchów znaków, operator ?. używany przy dostępie do składowych, filtry wyjątków, nową składnię inicjowania słowników i wiele innych mechanizmów.
Język C# 7.0 i platforma Microsoft .NET Framework 4.7 oraz .NET Core 1.1 i 2.0 (środowisko Visual Studio 2017)	Dodano krótki, dekonstruktory, dopasowywanie do wzorców, funkcje lokalne, zwracanie wartości przez referencję i inne mechanizmy.
Język C# 8.0 i platforma Microsoft .NET Framework 4.8 oraz .NET Core 3.0	Dodano typy referencyjne dopuszczające wartość null, zaawansowane dopasowanie do wzorca, deklaracje using, statyczne funkcje lokalne, struktury referencyjne z usuwaniem zasobów, przedziały i indeksy oraz strumienie asynchroniczne (przy czym Microsoft .NET Framework nie obsługuje dwóch ostatnich z wymienionych mechanizmów).

Początek
2.0
Początek
3.0
Początek
4.0
Początek
5.0
Początek
6.0
Początek
7.0
Początek
8.0

Koniec
5.0
Koniec
4.0
Koniec
3.0
Koniec
2.0

Prawdopodobnie najważniejszą funkcją platformy dodaną razem z wersją C# 6.0 była obsługa komplikacji programów dla wielu systemów. Oznacza to, że dostępna będzie nie tylko platforma Microsoft .NET Framework działająca w systemie Windows, ale też implementacja .NET Core dla systemów Linux i macOS. Choć platforma .NET Core nie udostępnia tylu funkcji co kompletna platforma Microsoft .NET Framework, obejmuje wystarczająco dużo mechanizmów, by uruchamianie kompletnych witryn napisanych w technologii ASP.NET było możliwe w systemach innych niż Windows i na serwerze innym niż IIS (ang. *Internet Information Server*). To oznacza, że ten sam kod bazowy pozwala kompilować i uruchamiać aplikacje działające w różnych systemach. Platforma .NET Core jest kompletnym pakietem SDK, który obejmuje wszystkie komponenty od .NET Compiler Platform („Roslyn”), który sam działa w systemach Linux i macOS, do środowiska uruchomieniowego .NET Core i narzędzi takich jak uruchamiane w wierszu poleceń dotnet, *dotnet.exe* (wprowadzone w podobnym czasie co język 7.0). Zarówno .NET Core, jak i dotnet CLI były stale rozwijane, a w C# 8.0 i .NET Core 3.0 stały się podstawowymi elementami środowiska .NET.

Koniec
6.0
Koniec
7.0
Koniec
8.0

.NET Standard

Ponieważ dostępnych jest tak wiele różnych implementacji platformy .NET, a także różne wersje każdej z tych implementacji, nastąpiło zróżnicowanie platform, poszczególne implementacje obsługują bowiem różne zbiorzy tylko częściowo pokrywających się interfejsów API. Dlatego pisanie kodu działającego w różnych wersjach platformy .NET stało się trudne; konieczne było zaśmiercanie kodu instrukcjami sprawdzającymi, czy dany interfejs API jest dostępny. Aby uprościć pracę, wprowadzono specyfikację .NET Standard, definiującą, które API muszą obsługiwane w poszczególnych wersjach platformy. Specyfikacje .NET Standard definiują więc, co platforma .NET musi udostępniać, aby zachować zgodność z określona wersją specyfikacji. Ponieważ jednak wiele implementacji już istnieje, podejmowanie decyzji o tym, które interfejsy API włączyć w poszczególne wersje specyfikacji, odbywało się w pewnym stopniu na podstawie istniejących implementacji i łączenia gotowych implementacji z numerami wersji specyfikacji .NET Standard.

W czasie, gdy powstaje ta książka, najnowszą wersją jest .NET Standard 2.1. Niestety, najnowsza wersja platformy Microsoft .NET Framework (4.8) nadal obsługuje wersję .NET Standard 2.0, przez co nie udostępnia przedziałów i indeksów oraz strumieni asynchronicznych z językiem C# 8.0. Pomijając to, zaletą wersji .NET Standard 2.0 jest to, że wszystkie najważniejsze wersje platformy są z nią zgodne. Dlatego specyfikacja .NET Standard 2.0 stanowi ujednolicenie rozbieżnych API ze starszych wersji platformy .NET i umożliwia komplikację kodu z użyciem każdej z najnowszych implementacji tej platformy.

Microsoft planuje, aby w przyszłości różne wersje kodu źródłowego platformy .NET zostały połączone w jeden kod bazowy wersji .NET 5.0. Ma to na celu ujednolicenie platform .NET Framework, .NET Core, Xamarin i Mono. Po takim scaleniu specyfikacja .NET Standard stanie się nieistotna. Jeśli wszystkie odmiany platformy .NET będą oparte na tym samym kodzie bazowym, wszystkie API będą automatycznie jednolite, dlatego nie będzie potrzebny standard, z którym zgodność trzeba zapewnić. Więcej informacji znajdziesz na stronie <http://itl.tc/net5unification>.

Podsumowanie

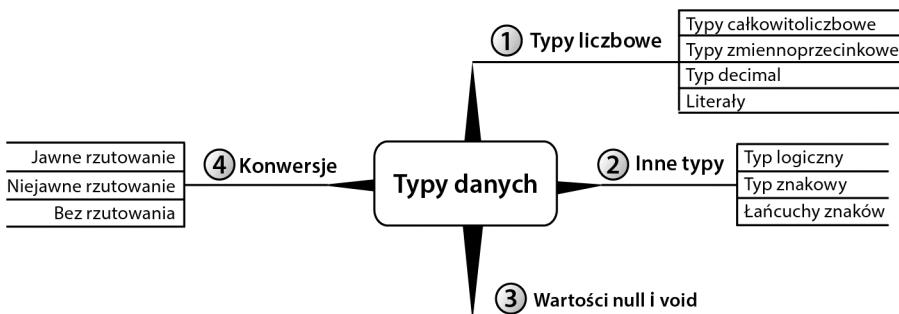
Ten rozdział to proste wprowadzenie do języka C#. Przedstawiono tu informacje pozwalające zapoznać się z podstawami składni tego języka. Z powodu podobieństwa C# do języków zbliżonych w stylu do C++ wiele z tych informacji mogło być Ci już znanych. Jednak C# i kod zarządzany mają pewne wyjątkowe cechy, takie jak komplikacja do języka CIL. Ponadto ważną (choć już nie wyjątkową) właściwością języka C# jest kompletna obsługa mechanizmów obiektowych. Nawet takie zadania jak odczyt i zapis danych w konsoli są wykonywane w sposób obiektowy. Obiektywość jest jedną z podstaw języka C#, o czym przekonasz się w trakcie lektury książki.

W następnym rozdziale opisano podstawowe typy danych będące częścią języka C#, a także pokazano, jak stosować te typy danych razem z operandami w celu tworzenia wyrażeń.

2

Typy danych

P RZEDSTAWIONY W ROZDZIALE 1. program HelloWorld pozwolił Ci wstępnie zapoznać się z językiem C#, jego strukturą, podstawowymi cechami składni i pisaniem najprostszych programów. W niniejszym rozdziale znajdziesz kontynuację omawiania podstaw języka C#. Tu przedstawiono podstawowe typy tego języka.



Do tego miejsca korzystałeś tylko z kilku wbudowanych typów danych, które nie były szczegółowo objaśnione. Język C# udostępnia tysiące typów; możesz je łączyć w nowe typy. Niektóre typy języka C# są stosunkowo proste i pełnią funkcję cegiełek do budowania pozostałych. Te podstawowe typy to **typy predefiniowane**. W języku C# do tej grupy zalicza się osiem typów całkowitoliczbowych, dwa binarne typy zmiennoprzecinkowe przeznaczone do przeprowadzania obliczeń naukowych, jeden dziesiętny typ zmiennoprzecinkowy do obliczeń finansowych, jeden typ logiczny i typ znakowy. W tym rozdziale znajdziesz analizę tych typów i szczegółowe omówienie typu `string`.

Podstawowe typy liczbowe

Dla podstawowych typów liczbowych języka C# używane są słowa kluczowe. Do tej grupy typów należą typy całkowitoliczbowe, typy zmiennoprzecinkowe i specjalny typ zmiennoprzecinkowy `decimal`, służący do przechowywania dużych liczb bez błędu reprezentacji danych.

Typy całkowitoliczbowe

W języku C# jest osiem typów całkowitoliczbowych wymienionych w tabeli 2.1. Ich różnorodność pozwala wybrać wystarczająco pojemy typ, by pomieścić wartości z oczekiwanej przedziału i nie marnować przy tym zasobów.

Tabela 2.1. Typy całkowitoliczbowe

Typ	Wielkość	Przedział (domknięty)	Nazwa w BCL	Ze znakiem?	Przyrostek
sbyte	8 bitów	Od -128 do 127	System.SByte	Tak	
byte	8 bitów	Od 0 do 255	System.Byte	Nie	
short	16 bitów	Od -32 768 do 32 767	System.Int16	Tak	
ushort	16 bitów	Od 0 do 65 535	System.UInt16	Nie	
int	32 bity	Od -2 147 483 648 do 2 147 483 647	System.Int32	Tak	
uint	32 bity	Od 0 do 4 294 967 295	System.UInt32	Nie	U lub u
long	64 bity	Od -9 223 372 036 854 775 808 do 9 223 372 036 854 775 807	System.Int64	Tak	L lub l
ulong	64 bity	Od 0 do 18 446 744 073 709 551 615	System.UInt64	Nie	UL lub ul

W tabeli 2.1 (oraz w tabelach 2.2 i 2.3) znajduje się kolumna z pełnymi nazwami poszczególnych typów. Przyrostki są opisane dalej w rozdziale. Wszystkie typy podstawowe w języku C# mają krótką i pełną nazwę. Pełna nazwa jest używana w bibliotece BCL. Ta nazwa (taka sama dla wszystkich języków powiązanych z tą biblioteką) jednoznacznie identyfikuje typ w podzespole. Ponieważ omawiane typy są podstawowe, w języku C# dostępne są też słowa kluczowe używane jako krótkie nazwy tych typów. Dla kompilatora obie nazwy dotyczą tego samego typu, dlatego generowany jest dokładnie taki sam kod. Gdy przyjrzesz się wynikowemu kodowi CIL, nie znajdziesz w nim informacji o tym, która nazwa została użyta.

Język C# obsługuje zarówno pełne nazwy z biblioteki BCL, jak i słowa kluczowe. Programiści muszą zdecydować, kiedy chcą stosować poszczególne nazwy. Zamiast przeskakiwać z jednej wersji na drugą, lepiej wybrać jeden zapis i konsekwentnie go stosować. Programiści języka C# zwykle używają słów kluczowych — na przykład `int` zamiast `System.Int32` i `string` zamiast `System.String` (lub skrótu `String`).

Wskazówki

STOSUJ słowa kluczowe z języka C# zamiast nazw BCL, gdy określasz typ danych (na przykład posługuj się nazwą `string` zamiast `String`).

PRZEDKŁADAJ w kodzie spójność nad różnorodność.

Niektóre wskazówki są niezgodne z zaleceniem zachowywania spójności. Nawet jeśli zastosujesz się do rady sugerującej używanie słów kluczowych języka C# zamiast nazw z biblioteki BCL, czasem będziesz musiał zajmować się konserwacją plików (lub bibliotek), w których używany jest odmienny styl. Wtedy lepiej zachować zgodność z pierwotnym stylem, niż wprowadzać nowy styl, który spowoduje niespójności w konwencjach stosowanych w kodzie. Jeśli jednak używany jest „styl” wynikający ze złych praktyk programistycznych, sprzyjający powstawaniu błędów i utrudniający konserwację, należy go poprawić.

Porównanie języków — typ danych short z języka C++

W językach C i C++ short to skrót od nazwy short int. W języku C# short to odrębny typ danych.

Typy zmiennoprzecinkowe (float i double)

Liczby zmiennoprzecinkowe mają różnych poziom precyzji. Binarne typy zmiennoprzecinkowe pozwalają precyzyjnie przedstawiać tylko te wartości, które są ułamkiem z potęgą liczby 2 w mianowniku. Jeśli ustawisz wartość zmiennej zmiennoprzecinkowej na 0,1, może ona zostać przedstawiona jako 0,0999999999999999, 0,10000000000000001 lub jako jeszcze inna liczba bardzo zbliżona do 0,1. Ponadto przypisanie do zmiennej bardzo dużej wartości, na przykład liczby Avogadra ($6,02 \times 10^{23}$), może prowadzić do błędu reprezentacji na poziomie 10^8 , co jest przecież tylko niewielkim ułamkiem całej liczby. Liczby zmiennoprzecinkowe są precyzyjne do określonej liczby znaczących cyfr — nie mają ustalonej wartości błędu takiej jak $\pm 0,01$. Od wersji .NET Core 3.0 typ double ma najwyższej 17 cyfr znaczących, a typ float — maksymalnie 9 cyfr znaczących (przy założeniu, że liczba nie została przekształcona z łańcucha znaków; zobacz ramkę ZAGADNIENIE DLA ZAAWANSOWANYCH „Analiza typów zmiennoprzecinkowych”)¹.

Język C# obsługuje dwa binarne typy zmiennoprzecinkowe wymienione w tabeli 2.2. Liczby binarne są tu reprezentowane jako wartości o podstawie 10, aby były czytelne dla użytkowników.

Tabela 2.2. Typy zmiennoprzecinkowe

Typ	Wielkość	Przedział (domknięty)	Nazwa w BCL	Liczba znaczących cyfr	Przyrostek
float	32 bity	Od $\pm 1,5 \times 10^{-45}$ do $\pm 3,4 \times 10^{38}$	System.Single	7	F lub f
double	64 bity	Od $\pm 5,0 \times 10^{-324}$ do $\pm 1,7 \times 10^{308}$	System.Double	15 – 16	D lub d

¹ Przed wersją .NET Core 3.0 bity (cyfry binarne) były przekształcane na 15 cyfr dziesiętnych, a reszta pozwalała określić szesnastą cyfrę dziesiętną (zobacz tabelę 2.2). Liczby z przedziału od $1,7 \times 10^{307}$ do 1×10^{308} (z pominięciem tej wartości) miały tylko 15 znaczących cyfr. Liczby z przedziału od 1×10^{308} do $1,7 \times 10^{308}$ miały 16 znaczących cyfr. Podobna liczba znaczących cyfr występowała też w wartościach typu decimal.

Początek
8.0

Koniec
8.0

ZAGADNIENIE DLA ZAAWANSOWANYCH

Analiza typów zmiennoprzecinkowych

Liczby dziesiętne z obsługiwanej przedziału i zgodne z limitem precyzji obowiązującym dla typu decimal są reprezentowane dokładnie. Jednak w reprezentacji wielu takich wartości za pomocą binarnych liczb zmiennoprzecinkowych kryje się błąd przybliżenia. Podobnie jak wartości $\frac{1}{3}$ nie można przedstawić precyzyjnie za pomocą skończonej liczby cyfr dziesiętnych, tak wartość $\frac{11}{10}$ nie może zostać zapisana precyzyjnie przy użyciu skończonej liczby cyfr dwójkowych (jej reprezentacja binarna ma postać 1,0001100110011001101...). W obu sytuacjach występuje błąd zaokrąglenia.

Liczba typu decimal jest reprezentowana jako $\pm N \cdot 10^k$, gdzie:

- N (mantysa) to dodatnia 96-bitowa liczba całkowita,
- k (wykładnik) należy do przedziału $-28 \leq k \leq 0$.

Binarna liczba zmiennoprzecinkowa to dowolna wartość w formie $\pm N \cdot 2^k$, gdzie:

- N to dodatnia 24-bitowa (typ float) lub 53-bitowa (typ double) liczba całkowita,
- k to liczba całkowita z przedziału od -149 do +104 (typ float) lub od -1074 do +970 (typ double).

Typ decimal

Język C# udostępnia typ zmiennoprzecinkowy decimal o precyzyji 128 bitów (zobacz tabelę 2.3). Ten typ jest odpowiedni do prowadzenia obliczeń finansowych.

Tabela 2.3. Typ decimal

Typ	Wielkość	Przedział (domknięty)	Nazwa w BCL	Liczba znaczących cyfr	Przyrostek
decimal	128 bitów	Od $1,0 \times 10^{-28}$ do $7,9 \times 10^{28}$	System.Decimal	28 – 29	M lub m

Typ decimal, w odróżnieniu od binarnych liczb zmiennoprzecinkowych, zachowuje precyzję wszystkich liczb dziesiętnych z obsługiwanej przedziału. Dlatego wartość 0,1 zapisała za pomocą tego typu zawsze jest równa 0,1. Jednak choć typ decimal zapewnia większą precyzję niż typy zmiennoprzecinkowe, ma mniejszy przedział. Z tego względu konwersja danych z typów zmiennoprzecinkowych na typ decimal może skutkować błędem przepelenia. Ponadto obliczenia z wykorzystaniem typu decimal są minimalnie wolniejsze (zwykle jest to niedostrzegalne).

Literał liczbowe

Literał liczbowy to reprezentacja stałej wartości w kodzie źródłowym. Jeśli na przykład chcesz wyświetlić za pomocą metody System.Console.WriteLine() wartość całkowitoliczbową 42 i wartość 1.618034 typu double, możesz wykorzystać kod z listingu 2.1.

Listing 2.1. Podawanie literałów liczbowych

```
System.Console.WriteLine(42);
System.Console.WriteLine(1.618034);
```

Dane wyjściowe 2.1 przedstawiają wyniki wykonania kodu z listingu 2.1.

DANE WYJŚCIOWE 2.1.

```
42
1.618034
```

ZAGADNIENIE DLA POCZĄTKUJĄCYCH**Zachowaj ostrożność, gdy zapisujesz wartości na sztywno**

Umieszczanie wartości bezpośrednio w kodzie źródłowym jest nazywane **zapisywaniem ich na sztywno** (ang. *hardcoding*), ponieważ zmiana wartości wymaga ponownego skompilowania kodu. Programiści muszą starannie rozważyć wybór między zapisywaniem wartości na sztywno w kodzie a pobieraniem ich z zewnętrznego źródła (na przykład z pliku konfiguracyjnego). To drugie rozwiązanie pozwala modyfikować wartości bez potrzeby rekompilacji kodu.

Gdy podasz literał z punktem dziesiętnym, kompilator domyślnie zinterpretuje wartość jako liczbę typu double. Literał bez punktu dziesiętnego zwykle domyślnie jest zapisywany za pomocą 32-bitowego typu int, chyba że wartość jest zbyt duża, by ten typ ją pomieścił. Dla większych wartości kompilator używa typu long. Ponadto kompilator języka C# umożliwia przypisywanie literałów do typów liczbowych innych niż int, pod warunkiem że wartość jest zgodna z docelowym typem danych. Dopuszczalne są na przykład przypisania short s = 42 i byte b = 77. Jednak ta składnia jest dozwolona tylko dla stałych. Instrukcja b = s jest nieprawidłowa, chyba że zastosowana zostanie dodatkowa składnia opisana w podrozdziale „Konwersje między typami danych” w dalszej części rozdziału.

Wcześniej w podrozdziale „Podstawowe typy liczbowe” wyjaśniono, że język C# obejmuje wiele różnych typów liczbowych. Na listingu 2.2 w kodzie w języku C# używany jest literał. Ponieważ liczby z punktem dziesiętnym są domyślnie zapisywane za pomocą typu double, w danych wyjściowych 2.2 widoczny jest wynik 1.61803398874989 (z brakującą końcową cyfrą 5), co odpowiada oczekiwanej dokładności typu double.

Listing 2.2. Podawanie literała typu double

```
System.Console.WriteLine(1.61803398874989);
```

DANE WYJŚCIOWE 2.2.

```
1.61803398874989
```

Aby wyświetlić podaną liczbę z pełną precyzją, trzeba jawnie zadeklarować literał jako wartość typu decimal. W tym celu należy dodać literę M lub m (zobacz listingu 2.3 i dane wyjściowe 2.3).

Listing 2.3. Podawanie literala typu decimal

```
System.Console.WriteLine(1.618033988749895M);
```

DANE WYJŚCIOWE 2.3.

```
1.618033988749895
```

Teraz dane wyjściowe z listingu 2.3 mają oczekiwany postać — 1.618033988749895. Litera d służy do ustawiania typu double. Aby zapamiętać, że dla typu decimal należy dodawać literę m, zastosuj mnemotechnikę: „m jest używane dla obliczeń na mamonie”.

Możesz też dodać inne przyrostki, aby jawnie zadeklarować literał jako wartość typu float (F) lub double (D). Dla typów całkowitoliczbowych używane są przyrostki U, L, LU i UL. Typ literalów całkowitoliczbowych jest określany w następujący sposób:

- Literaly liczbowe bez przyrostka są interpretowane jako wartość pierwszego typu, który potrafi pomieścić daną liczbę, przy czym uwzględniane są kolejno typy int, uint, long, ulong.
- Literaly liczbowe z przyrostkiem U są interpretowane jako wartość pierwszego typu, który potrafi pomieścić daną liczbę, przy czym uwzględniane są kolejno typy uint i ulong.
- Literaly liczbowe z przyrostkiem L są interpretowane jako wartość pierwszego typu, który potrafi pomieścić daną liczbę, przy czym uwzględniane są kolejno typy long i ulong.
- Jeśli w literale występuje przyrostek UL lub LU, używany jest typ ulong.

Zauważ, że w przyrostkach w literalach wielkość znaków nie ma znaczenia. Zwykle jednak preferuje się stosowanie wielkich liter, by uniknąć pomylenia małej litery l z cyfrą 1.

Wskazówka

STOSUJ w literalach przyrostki w postaci wielkich liter
(na przykład 1.618033988749895M).

Początek
7.0

Zdarza się, że liczby są długie i mało czytelne. Aby rozwiązać problem z czytelnością, w C# 7.0 dodano **separator cyfr** w postaci znaku podkreślenia (_), który można stosować w zapisie literalów liczbowych takich jak na listingu 2.4.

Listing 2.4. Stosowanie separatora cyfr

```
System.Console.WriteLine(9_814_072_356);
```

Koniec
7.0

W tym kodzie liczba jest podzielona na grupy obejmujące po trzy cyfry, jednak C# nie narzuca wielkości takich grup. Separator cyfr możesz stosować do tworzenia dowolnych grup, przy czym podkreślenia muszą się znajdować między pierwszą a ostatnią cyfrą. Możesz nawet wstawić kilka podkreseń obok siebie (bez cyfr między nimi).

W niektórych sytuacjach wygodniej jest zastosować notację wykładniczą, zamiast dodawać listę zer przed punktem dziesiętnym lub po nim (niezależnie od tego, czy stosujesz separator cyfr). Aby posłużyć się notacją wykładniczą, podaj wrostek e lub E, po nim wpisz dodatnią bądź ujemną liczbę całkowitą, a następnie uzupełnij literał przyrostkiem dla odpowiedniego typu danych. Możesz na przykład wyświetlić liczbę Avogadra z wykorzystaniem typu float, co przedstawiają listing 2.5 i dane wyjściowe 2.4.

Listing 2.5. Notacja wykładnicza

```
System.Console.WriteLine(6.023E23F);
```

DANE WYJŚCIOWE 2.4.

```
6.023E+23
```

ZAGADNIENIE DLA POCZĄTKUJĄCYCH

Notacja szesnastkowa

Zwykle używane są liczby reprezentowane za pomocą podstawy 10. Oznacza to, że każda cyfra w liczbie jest reprezentowana za pomocą jednego z dziesięciu symboli (od 0 do 9). Gdy liczba jest wyświetlana w notacji szesnastkowej, stosowane są liczby o podstawie 16. Używa się wtedy 16 symboli — od 0 do 9 i od A do F (można też posługiwać się małymi literami). Wartość 0x000A odpowiada dziesiętnej wartości 10, a liczba 0x002A to wartość dziesiętna 42 ($2 \times 16 + 10$). Reprezentowane liczby są takie same. Przejście z notacji szesnastkowej na dziesiętną lub w drugą stronę nie zmienia samej liczby, a jedynie jej reprezentację.

Każda cyfra w notacji szesnastkowej zajmuje cztery bity, tak więc za pomocą bajta można zapisać dwie cyfry szesnastkowe.

Dotychczas w omówieniu literałów uwzględniane były tylko wartości o podstawie 10. Język C# umożliwia też podawanie wartości szesnastkowych. Aby podać taką wartość, należy poprzedzić ją członem 0x, a następnie wpisać dowolne cyfry szesnastkowe. Ilustruje to listing 2.6.

Listing 2.6. Literał szesnastkowy

```
// Wyświetlanie wartości 42 za pomocą literala szesnastkowego.  
System.Console.WriteLine(0x002A);
```

W danych wyjściowych 2.5 pokazany jest wynik wykonania kodu z listingu 2.6. Zauważ, że kod wyświetla wartość 42, a nie 0x002A.

DANE WYJŚCIOWE 2.5.

```
42
```

Od wersji C# 7.0 liczby można reprezentować w postaci binarnej (zobacz listing 2.7).

Listing 2.7. Literał binarny

```
// Wartość 42 zapisana jako literał binarny.
System.Console.WriteLine(0b101010);
```

Ten zapis przypomina składnię dla liczb szesnastkowych, przy czym przedrostkiem jest tu człon 0b (można też używać wielkiej litery B). W ZAGADNIENIU DLA POCZĄTKUJĄCYCH „Bity i bajty” z rozdziału 4. znajdziesz kompletne omówienie zapisu binarnego oraz konwersji między formatami binarnym i dziesiętnym.

Koniec
7.0

Warto zauważyc, że od wersji C# 7.2 separator cyfr można umieszczać po literze x (w literałach szesnastkowych) lub po literze b (w literałach binarnych).

ZAGADNIENIE DLA ZAAWANSOWANYCH**Wyświetlanie liczb w formacie szesnastkowym**

Aby wyświetlić liczbę w formacie szesnastkowym, trzeba zastosować modyfikator formatu x lub X. Wielkość modyfikatora określa, czy w liczbach szesnastkowych będą używane wielkie, czy małe litery. Listing 2.8 przedstawia przykład zastosowania tej techniki.

Listing 2.8. Przykład użycia modyfikatora formatu szesnastkowego

```
// Wyświetla "0x2A"
System.Console.WriteLine($"0x{42:X}");
```

Wynik jest pokazany w danych wyjściowych 2.6.

DANE WYJŚCIOWE 2.6.

0x2A

Zauważ, że używany literał liczbowy (42) można podać w postaci dziesiętnej lub szesnastkowej. Efekt będzie taki sam. Ponadto by wyświetlić liczbę w formacie szesnastkowym, zastosowano modyfikator formatu oddzielony średnikiem od wyrażenia, dla którego stosowana jest interpolacja łańcuchów znaków.

ZAGADNIENIE DLA ZAAWANSOWANYCH**Formatowanie dwustronne**

Instrukcja `System.Console.WriteLine(1.618033988749895);` domyślnie wyświetla liczbę 1.61803398874989, w której brakuje ostatniej cyfry. Aby precyzyjnie wyświetlić tekstową reprezentację wartości typu double, można przekształcić ją za pomocą łańcucha znaków formatowania i modyfikatora formatowania dwustronnego (ang. *round-trip format*) — R lub r. Na przykład instrukcja `string.Format("{0:R}", 1.618033988749895)` zwraca wynik 1.6180339887498949.

Modyfikator formatowania dwustronnego zwraca łańcuch znaków, którego ponowna konwersja na liczbę zawsze daje pierwotną wartość. Listing 2.9 przedstawia zmienne liczbowe, które są uznawane za równe tylko wtedy, gdy zastosowane jest formatowanie dwustronne.

Listing 2.9. Formatowanie z wykorzystaniem modyfikatora formatowania R

```
// ...
const double number = 1.618033988749895;
double result;
string text;

text = $"{number}";
result = double.Parse(text);
System.Console.WriteLine($"{result == number}: result == number");

text = string.Format("{0:R}", number);
result = double.Parse(text);
System.Console.WriteLine($"{result == number}: result == number");

// ...
```

Uzyskany wynik przedstawiono w danych wyjściowych 2.7.

DANE WYJŚCIOWE 2.7.

```
False: result == number
True: result == number
```

W pierwszym przypisaniu wartości do zmiennej `text` nie ma modyfikatora formatowania dwustronnego. Dlatego wartość zwracana przez wyrażenie `double.Parse(text)` różni się od pierwotnej wartości ze zmiennej `number`. Natomiast po użyciu modyfikatora formatowania dwustronnego wyrażenie `double.Parse(text)` zwraca pierwotną wartość.

Krótką uwagę skierowaną do czytelników, którzy nie znają składni `==` z języków opartych na C — wyrażenie `result == number` zwraca wartość `true`, jeśli zmienna `result` jest równa zmiennej `number`. Wyrażenie `result != number` zwraca `true`, jeśli zmienne zawierają różne wartości. Operacje przypisania i operatory równości opisano w następnym rozdziale.

Inne podstawowe typy

Do tej pory omówiono liczbowe typy podstawowe. Język C# obejmuje też dodatkowe typy `bool`, `char` i `string`.

Typ logiczny (`bool`)

Innym typem bezpośrednim w języku C# jest typ logiczny `bool`, reprezentujący prawdę lub fałsz w instrukcjach warunkowych i wyrażeniach. Dozwolonymi wartościami dla tego typu są słowa kluczowe `true` i `false`. W bibliotece BCL nazwą typu `bool` jest `System.Boolean`. Aby porównać dwa łańcuchy znaków bez uwzględniania wielkości liter, wywołaj metodę `string.Compare()` z argumentem w postaci literału `true` typu `bool` (zobacz listing 2.10).

Listing 2.10. Porównywanie dwóch łańcuchów znaków bez uwzględniania wielkości liter

```
string option;  
...  
int comparison = string.Compare(option, "/Help", true);
```

Ten kod porównuje (bez uwzględniania wielkości znaków) zawartość zmiennej `option` z literałem tekstowym `/Help` i przypisuje wynik do zmiennej `comparison`.

Choć teoretycznie do przechowywania wartości logicznych wystarcza jeden bit, wartości typu `bool` zajmują jeden bajt.

Typ znakowy (char)

Typ `char` reprezentuje 16-bitowe znaki, których zestaw wartości jest oparty na kodowaniu UTF-16 ze standardu Unicode. Typ `char` zajmuje tyle samo miejsca co 16-bitowe liczby całkowite bez znaku (`ushort`), które reprezentują wartości z przedziału od 0 do 65 535. Jednak typ `char` jest w języku C# odrębnym typem i tak należy go traktować w kodzie.

Nazwa typu `char` w bibliotece BCL to `System.Char`.

ZAGADNIENIE DLA POCZĄTKUJĄCYCH

Standard Unicode

Unicode to międzynarodowy standard używany do przedstawiania znaków występujących w większości języków naturalnych. Unicode umożliwia budowanie w systemach komputerowych **lokalizowanych** aplikacji, wyświetlanego w odpowiednim języku i w sposób dostosowany do różnych kultur.

ZAGADNIENIE DLA ZAAWANSOWANYCH

16 bitów nie wystarcza do przedstawienia wszystkich znaków Unicode

Niestety, jeden 16-bitowy typ `char` nie pozwala na przedstawienie wszystkich znaków Unicode. Autorzy standardu Unicode wierzyli, że 16 bitów wystarczy, jednak wraz z dodawaniem obsługi nowych języków okazało się, że to założenie było błędne. W efekcie niektóre (rzadko stosowane) znaki Unicode składają się z par zastępczych (ang. *surrogate pairs*) obejmujących dwie wartości typu `char`.

Aby utworzyć literał typu `char`, umieść znak między apostrofami, na przykład '`'A'`'. Dzwolone są wszystkie symbole z klawiatury, w tym litery, cyfry i znaki specjalne.

Niektórych znaków nie można bezpośrednio umieszczać w kodzie źródłowym. Te symbole wymagają specjalnej obsługi. Takie znaki poprzedzane są lewym ukośnikiem (`\`), po którym następuje kod znaku specjalnego. Lewy ukośnik w połączeniu z kodem znaku specjalnego tworzy **sekwencję ucieczki**. Na przykład sekwencja `\n` reprezentuje nowy wiersz, a sekwencja `\t` — tabulację. Ponieważ lewy ukośnik oznacza początek sekwencji ucieczki, nie może reprezentować zwykłego lewego ukośnika. Dlatego by zapisać jeden lewy ukośnik, należy zastosować sekwencję `\\`.

Kod z listingu 2.11 wyświetla jeden apostrof, ponieważ właśnie ten znak jest reprezentowany przez sekwencję `\'`.

Listing 2.11. Wyświetlanie apostrofu za pomocą sekwencji ucieczki

```
class SingleQuote
{
    static void Main()
    {
        System.Console.WriteLine('\'');
    }
}
```

W tabeli 2.4 przedstawiono dostępne sekwencje ucieczki i reprezentacje używanych w nich znaków w standardzie Unicode.

Tabela 2.4. Sekwencje ucieczki

Sekwencja ucieczki	Nazwa znaku	Kodowanie w Unicode
\'	Apostrof	\u0027
\"	Cudzysłów	\u0022
\\"	Lewy ukośnik	\u005C
\0	Zero	\u0000
\a	Alert (systemowy sygnał dźwiękowy)	\u0007
\b	Klawisz <i>Backspace</i>	\u0008
\f	Znak wysunięcia strony	\u000C
\n	Znak nowego wiersza	\u000A
\r	Znak powrotu karetki	\u000D
\t	Tabulacja w poziomie	\u0009
\v	Tabulacja w pionie	\u000B
\xxxxx	Znak Unicode w zapisie szesnastkowym	\u0029
\x[n] [n] n	Znak Unicode w zapisie szesnastkowym (trzy pierwsze symbole zastępcze są tu opcjonalne); wersja zapisu \xxxx dopuszczająca zmienną długość	\u3A
\Uxxxxxxxx	Sekwencja ucieczki służąca do tworzenia par zastępczych ze znakami Unicode	\uD840DC01 (ϑ)

Z pomocą kodowania Unicode można przedstawić dowolny znak. W tym celu poprzedź wartość reprezentującą znak Unicode sekwencją `\'u`. Znaki Unicode są zapisywane w notacji szesnastkowej. Na przykład literze A odpowiada wartość szesnastkowa 0x41. Na listingu 2.12 wykorzystano znaki Unicode do przedstawienia uśmiechniętej buźki (:)), a efekt pokazany jest w danych wyjściowych 2.8.

Listing 2.12. Używanie kodowania Unicode do wyświetlania uśmiechniętej buźki

```
System.Console.WriteLine('\u003A');
System.Console.WriteLine('\u0029');
```

DANE WYJŚCIOWE 2.8.

```
:)
```

Łańcuchy znaków

Skończona sekwencja obejmująca zero lub więcej znaków to **łańcuch znaków** (ang. *string*). W języku C# do reprezentowania łańcuchów znaków służy typ `string`, którego nazwa w bibliotece BCL to `System.String`. Typ `string` ma pewne wyjątkowe cechy, które mogą się okazać nieoczekiwane dla programistów posługujących się innymi językami programowania. W rozdziale 1. opisano literały tekstowe. Dostępne są też dosłowne łańcuchy znaków, poprzedzone przedrostkiem `@`, oraz możliwa jest interpolacja łańcuchów znaków, do czego służy przedrostek `$`. Należy pamiętać, że łańcuchy znaków są niezmienne.

Literały

Aby wprowadzić literał tekstowy w kodzie, ujmij tekst w cudzysłów (""), tak jak w programie `HelloWorld`. łańcuchy znaków składają się ze znaków, dlatego w łańcuchu można umieszczać sekwencje ucieczki.

Kod z listingu 2.13 wyświetla dwa wiersze tekstu. Jednak zamiast metody `System.Console.WriteLine()` wywoływana jest metoda `System.Console.Write()` z sekwencją nowego wiersza — `\n`. Efekty są przedstawione w danych wyjściowych 2.9.

Listing 2.13. Stosowanie sekwencji `\n` do wstawiania nowego wiersza

```
class DuelOfWits
{
    static void Main()
    {
        System.Console.Write(
            "\"Twoja inteligencja jest doprawdy niezwykła.\"");
        System.Console.Write("\n\"Poczekaj, aż się rozkręczę!\"\n");
    }
}
```

DANE WYJŚCIOWE 2.9.

```
"Twoja inteligencja jest doprawdy niezwykła."
"Poczekaj, aż się rozkręczę!"
```

Sekwencja ucieczki z cudzysłowem pozwala odróżnić wyświetlany cudzysłów od cudzysłowa definiującego początek lub koniec łańcucha znaków.

W języku C# można przed łańcuchem znaków dodać symbol @, by podkreślić, że lewego ukośnika nie należy interpretować jako początku sekwencji ucieczki. W efekcie powstaje **dosłowny literał tekstowy**, w którym nie są interpretowane lewe ukośniki ani inne znaki. Gdy stosujesz symbol @, także odstępy są traktowane dosłownie. Na przykład trójkąt z listingu 2.14 pojawia się w konsoli dokładnie w takiej postaci, w jakiej został zapisany, z lewymi ukośnikami, znakami nowego wiersza i wcięciami. Efekt pokazano w danych wyjściowych 2.10.

Listing 2.14. Wyświetlanie trójkąta za pomocą dosłownego literala tekstowego

DANE WYJŚCIOWE 2.10.

Jeśli pominiesz symbol `\t`, kod się nawet nie skompiluje. Nawet jeżeli zmienisz kształt na kwadrat i wylimujesz lewe ukośniki, kodu i tak nie da się skompilować, ponieważ w łańcuchu znaków, który nie jest poprzedzony symbolem `\t`, nie można bezpośrednio umieszczać znaków nowego wiersza.

Jedyna sekwencja ucieczki przetwarzana w dosłownych łańcuchach znaków to "". Oznacza ona cudzysłów i nie kończy łańcucha znaków.

Porównanie języków — łączenie łańcuchów znaków w czasie komplikacji w języku C++

Język C#, inaczej niż język C++, nie łącza automatycznie literałów tekstowych. Nie można na przykład zapisać literalu tekstuowego w następujący sposób:

"Major Strasser został zastrzelony."

"Aresztujcie podejrzanych – tych co zawsze."

Złączanie wymaga zastosowania operatora dodawania. Jeśli jednak kompilator potrafi uzyskać wynik w trakcie komplikacji, w wynikowym kodzie CIL użyty zostanie jeden łańcuch znaków.

Jeżeli ten sam literał tekstowy występuje w podzespolu wielokrotnie, kompilator zdefiniuje łańcuch znaków tylko raz w podzespolu, a wszystkie zmienne będą prowadzić do tego samego łańcucha. Dzięki temu jeżeli taki sam literał zawierający tysiące znaków pojawi się w kodzie wiele razy, w wynikowym podzespolu potrzebne będzie miejsce na tylko jedno wystąpienie tego tekstu.

Początek
6.0

Interpolacja łańcuchów znaków

W rozdziale 1. wyjaśniono, że w łańcuchach znaków w formacie z interpolacją (wprowadzonych w C# 6.0) można umieszczać wyrażenia. Składnia interpolowanych łańcuchów znaków wymaga poprzedzenia literala tekstowego symbolem dolara. Zagnieżdżane w łańcuchu wyrażenia należy umieszczać w nawiasach klamrowych. Oto przykład:

```
System.Console.WriteLine($"Twoje imię i nazwisko to {firstName} {lastName}.");
```

W tym kodzie `firstName` i `lastName` to proste wyrażenia reprezentujące zmienne.

Początek
8.0

Zauważ, że łańcuch może być jednocześnie dosłowny i interpolowany. Aby utworzyć taki łańcuch, umieść symbol \$ przed znakiem @, tak jak w poniższym przykładzie (od wersji C# 8.0 można też użyć sekwencji @\$"..."):

Koniec
8.0

```
System.Console.WriteLine($@"Twoje imię i nazwisko to:  
{ firstName } { lastName }");
```

Ponieważ jest to dosłowny literał, tekst zostanie wyświetlony w dwóch wierszach. Możesz też rozbić kod na dwa wiersze, a jednocześnie uniknąć tworzenia dwóch linii w danych wyjściowych. W tym celu umieść znak nowego wiersza w nawiasach klamrowych:

```
System.Console.WriteLine($@"Twoje imię i nazwisko to:  
{ firstName } { lastName }");
```

Warto zauważyc, że symbol @ jest potrzebny nawet wtedy, gdy znak nowego wiersza znajduje się w nawiasie klamrowym.

ZAGADNIENIE DLA ZAAWANSOWANYCH

Wewnętrzne mechanizmy interpolacji łańcuchów znaków

Interpolacja łańcuchów znaków to krótsza wersja wywołania metody `string.Format()`. Przyjrzyj się poniższej instrukcji:

```
System.Console.WriteLine($"Twoje imię i nazwisko to {firstName} {lastName}.")
```

W języku C# można ją przekształcić na dające ten sam efekt polecenia:

```
object[] args = new object[] { firstName, lastName };  
Console.WriteLine(string.Format("Twoje imię i nazwisko to {0} {1}.", args));
```

Koniec
6.0

W tym podejściu można lokalizować tekst w podobny sposób jak w złożonych łańcuchach znaków. Ponadto chroni to przed wstrzykiwaniem kodu za pomocą łańcuchów znaków po komplikacji.

Metody dla typu string

Typ `string`, podobnie jak typ `System.Console`, udostępnia zestaw metod. Dostępne są na przykład metody służące do formatowania, łączenia i porównywania łańcuchów znaków.

Metoda `Format()` (przedstawiona w tabeli 2.5) działa w podobny sposób jak metody `Console.WriteLine()` i `Console.WriteLine()`, jednak wywołanie `string.Format()` nie wyświetla wyniku w oknie konsoli, ale zwraca dane do jednostki wywołującej. Oczywiście dzięki wprowadzeniu interpolacji łańcuchów znaków metoda `string.Format()` jest stosowana znacznie rzadziej (służy głównie do obsługi lokalizowania). Jednak na zapleczu interpolacja łańcuchów znaków jest kompilowana do kodu CIL wykorzystującego wywołanie `string.Format()`.

Tabela 2.5. Metody statyczne typu `string`

Instrukcje	Przykłady
<code>static string</code> <code>string.Format(</code> <code>string format,</code> <code>...)</code>	<code>string text, firstName, lastName;</code> <code>//...</code> <code>text = string.Format("Twoje imię i nazwisko to {0}</code> <code>{1}.",</code> <code>firstName, lastName);</code> <code>// Wyświetla</code> <code>// "Twoje imię i nazwisko to <firstName> <lastName>."</code> <code>System.Console.WriteLine(text);</code>
<code>static string</code> <code>string.Concat(</code> <code>string str0,</code> <code>string str1)</code>	<code>string text, firstName, lastName;</code> <code>//...</code> <code>text = string.Concat(firstName, lastName);</code> <code>// Wyświetla "<firstName><lastName>"; zwróć uwagę</code> <code>// na brak spacji między imieniem a nazwiskiem.</code> <code>System.Console.WriteLine(text);</code>
<code>static int</code> <code>string.Compare(</code> <code>string str0,</code> <code>string str1)</code> <code>static int</code> <code>string.Compare(</code> <code>string str0,</code> <code>string str1)</code> <code>string ignoreCase)</code>	<code>string option;</code> <code>//...</code> <code>// Porównanie z uwzględnianiem wielkości liter.</code> <code>int result = string.Compare(option, "/help");</code> <code>string option;</code> <code>//...</code> <code>// Wyświetla:</code> <code>// 0 dla równych łańcuchów znaków</code> <code>// wartość ujemna, gdy option < /help</code> <code>// wartość dodatnia, gdy option > /help</code> <code>System.Console.WriteLine(result);</code> <code>string option;</code> <code>//...</code> <code>// Porównywanie bez uwzględniania wielkości znaków</code> <code>int result = string.Compare(</code> <code>option, "/Help", true);</code> <code>// Wyświetla:</code> <code>// 0 dla równych łańcuchów znaków</code> <code>// wartość ujemna, gdy option < /help</code> <code>// wartość dodatnia, gdy option > /help</code> <code>System.Console.WriteLine(result);</code>

Wszystkie metody wymienione w tabeli 2.5 są **statyczne**. To oznacza, że w celu wywołania metody trzeba poprzedzić jej nazwę (na przykład Concat) nazwą typu zawierającego tę metodę (na przykład string). Jednak, jak pokazano dalej, niektóre metody klasy string są metodami **instancjnymi**. Takie metody należy poprzedzać nazwą zmiennej (lub inną referencją do instancji klasy), a nie nazwą typu. W tabeli 2.6 wymieniono kilka metod z tej grupy i pokazano przykłady ich stosowania.

Tabela 2.6. Metody typu string

Instrukcje	Przykłady
<code>bool StartsWith(string value)</code>	<code>string lastName //... bool isPhd = lastName.EndsWith("Ph.D."); bool isDr = lastName.StartsWith("Dr.");</code>
<code>string ToLower() string ToUpper()</code>	<code>string severity = "ostrzeżenie"; // Wyświetla zawartość zmiennej severity wielkimi literami. System.Console.WriteLine(severity.ToUpper());</code>
<code>string Trim() string Trim(...) string TrimEnd() string TrimStart()</code>	<code>// Usuwanie odstępów na początku lub końcułańcucha. username = username.Trim(); string text = "kaseta chłopaka"; // Usuwa 'k' oraz 'a' na początku i na końcułańcucha. text = text.Trim("ka".ToCharArray()); // Wyświetla: seta chłopak System.Console.WriteLine(text);</code>
<code>string Replace(string oldValue, string newValue)</code>	<code>string filename; //... // Usuwanie znaków ? z lańcucha. filename = filename.Replace("?", "");;</code>

ZAGADNIENIE DLA ZAAWANSOWANYCH

Dyrektwy using i using static

Początek
6.0

Dotychczas w książce w wywołaniach metod statycznych używany był przedrostek w postaci przestrzeni nazw, po którym następowała nazwa typu. Na przykład w instrukcji `System.Console.WriteLine` wywoływana jest metoda `WriteLine()` i choć w kontekście nie są dostępne żadne inne metody o tej nazwie, to i tak trzeba poprzedzić ją przestrzenią nazw (`System`) oraz typem (`Console`). Czasem programista chce skrócić kod i uniknąć bezpośredniego podawania pełnej nazwy. W tym celu można wykorzystać dyrektywę `using static` z wersji C# 6.0. Ilustruje to listing 2.15.

Listing 2.15. Dyrektywa using static

```
// Dyrektywy using pozwalają pominąć przestrzeń nazw
using static System.Console;
class HeyYou
{
```

```

static void Main()
{
    string firstName;
    string lastName;

    WriteLine("Hej, ty!");

    Write("Wprowadź imię: ");
    firstName = ReadLine();

    Write("Wprowadź nazwisko: ");
    lastName = ReadLine();
    WriteLine(
        $"Twoje imię i nazwisko to {firstName} {lastName}.");
}

```

Dyrektyna `using static` musi się znajdować na początku pliku². Po dodaniu tej dyrektywy, gdy używasz klasy `System.Console`, nie musisz podawać przedrostka `System.Console`. Wystarczy wpisać nazwę metody. Przy stosowaniu dyrektywy `using static` warto pamiętać, że działa ona tylko dla metod i właściwości statycznych, a nie dla składowych instancyjnych.

Podobna dyrektywa `using` umożliwia pominięcie przedrostka z przestrzenią nazw (na przykład `System`). Dyrektywa `using` (w odróżnieniu od dyrektywy `using static`) dotyczy wszystkich elementów w pliku (lub w przestrzeni nazw), gdzie występuje; nie jest ograniczona do składowych statycznych. Gdy korzystasz z dyrektywy `using`, możesz (opcjonalnie) wyeliminować wszystkie referencje do przestrzeni nazw — w instrukcjach tworzenia instancji, w wywołaniach metod statycznych, a nawet w wywołaniach operatora `nameof` dostępnego w wersji C# 6.0.

Koniec
6.0

Formatowanie łańcuchów znaków

Gdy używasz metody `string.Format()` lub posługujesz się interpolacją łańcuchów znaków z wersji C# 6.0 do tworzenia złożonych łańcuchów znaków formatowania, możesz korzystać z bogatego zestawu wzorców formatowania. Umożliwiają one wyświetlanie liczb, dat, godzin, przedziałów czasu itd. Na przykład jeśli używasz zmiennej `price` typu `decimal`, instrukcja `string.Format("{0,20:C2}", price)` lub analogiczna wersja z interpolacją, `$"{price,20:C2}"`, powodują przekształcenie wartości typu `decimal` na łańcuch znaków. Używane są przy tym domyślne reguły formatowania wartości finansowych — kod zaokrąglą liczbę do dwóch miejsc po przecinku i wyrównuje tekst do prawej w ramach 20-znakowego łańcucha. Nie ma tu miejsca na szczegółowe omawianie wszystkich możliwych łańcuchów znaków formatowania. Kompletną listę takich łańcuchów znajdziesz w opisie metody `string.Format()` w dokumentacji MSDN (<http://itl.tc/CompositeFormatting>).

Jeśli chcesz wyświetlić lewy lub prawy nawias klamrowy w interpolowanym lub formatowanym łańcuchu znaków, możesz podać podwójny nawias, aby określić, że nie jest to część wzorca. Przykładowy interpolowany łańcuch znaków `$"{{ {{ price:C2 } }} }"` generuje łańcuch znaków w postaci `"{ $1,234.56 }"`.

² Lub na początku deklaracji przestrzeni nazw.

Nowy wiersz

Gdy dodajesz nowy wiersz, potrzebne do tego znaki zależą od używanego systemu operacyjnego. W systemie Microsoft Windows znak nowego wiersza łączy znak powrotu karetki (\r) ze znakiem przesuwu (\n), natomiast w systemie UNIX stosowany jest pojedynczy znak przesuwu. Jednym ze sposobów na rozwiązywanie niezgodności między systemami jest używanie metody `System.Console.WriteLine()` do wyświetlania pustego wiersza. Inna instrukcja, która w wielu systemach jest niemal niezbędna do dodania nowego wiersza, gdy dane nie są wyświetlane w konsoli, to `System.Environment.NewLine`. Polecenia `System.Console.WriteLine("Witaj, świecie")` i `System.Console.WriteLine($"Witaj, świecie{System.Environment.NewLine}")` działają tak samo. Jednak w systemie Windows instrukcje `System.WriteLine()` i `System.Console.WriteLine(System.Environment.NewLine)` działają tak jak `System.Console.WriteLine("\r\n")`, a nie jak `System.Console.WriteLine("\n")`. Dlatego aby uwzględnić rozbieżności między systemami Windows a Linux i iOS, lepiej jest używać instrukcji `System.WriteLine()` i `System.Environment.NewLine` zamiast znaku \n.

Wskazówki

STOSUJ polecenia `System.WriteLine()` i `System.Environment.NewLine` zamiast znaku \n, jeśli chcesz uwzględnić rozbieżności w wykonywaniu kodu w systemach Windows oraz Linux i iOS.

■ ZAGADNIENIE DLA ZAAWANSOWANYCH

Właściwości w języku C#

Składowa `Length` używana w dalszym tekście nie jest metodą. Informuje o tym brak nawiasów po wywoaniu tej składowej. `Length` to właściwość typu `string`. Składnia języka C# pozwala na dostęp do właściwości w taki sposób, jakby były zmiennymi składowymi (nazywanymi w języku C# **polami**). Za działanie właściwości odpowiadają specjalne metody, settery i gettery, z których jednak korzysta się tak jak z pola.

Gdy przyjrzyisz się implementacji właściwości w kodzie CIL, zobacysz, że po komplikacji używane są dwie metody — `set_<NazwaWłaściwości>` i `get_<NazwaWłaściwości>`. Jednak żadna z tych metod nie jest bezpośrednio dostępna w kodzie w języku C# — konieczne jest stosowanie konstrukcji specyficznej dla właściwości. Więcej informacji o właściwościach znajdziesz w rozdziale 6.

Długość łańcuchów znaków

Aby określić długość łańcucha znaków, można wykorzystać składową `Length` typu `string`. Ta składowa to **właściwość tylko do odczytu**, dlatego nie można ustawić jej wartości. Nie wymaga ona żadnych parametrów. Na listingu 2.16 pokazano, jak zastosować właściwość `Length`. W danych wyjściowych 2.11 zaprezentowano efekty działania kodu.

Listing 2.16. Używanie składowej Length typu string

```
class PalindromeLength
{
    static void Main()
    {
        string palindrome;

        System.Console.Write("Wprowadź palindrom: ");
        palindrome = System.Console.ReadLine();

        System.Console.WriteLine(
            $"Liczba znaków w palindromie \"{palindrome}\" to"
            + $" {palindrome.Length}.");
    }
}
```

DANE WYJŚCIOWE 2.11.

Wprowadź palindrom: A to kanapa pana Kota
Liczba znaków w palindromie "A to kanapa pana Kota" to 21.

Długości łańcucha znaków nie można bezpośrednio ustawić. Jest ona obliczana na podstawie liczby znaków w łańcuchu. Ponadto długość nie może się zmieniać, ponieważ łańcuchy znaków są **niezmienne**.

Łańcuchy znaków są niezmienne

Ważną cechą typu `string` jest jego niezmienność. Do zmiennej typu `string` można przypisać nową wartość, nie można jednak modyfikować zawartości takich zmiennych. Nie można więc na przykład przekształcić wartości typu `string` na łańcuch składający się z samych wielkich liter. Można w prosty sposób utworzyć nowy łańcuch znaków, składający się z wielkich liter wchodzących w skład dawnego łańcucha, jednak dawny łańcuch nie jest w tym procesie modyfikowany. Przyjrzyj się przykładowemu listingowi 2.17.

Listing 2.17. Błąd — łańcuchy znaków są niezmienne

```
class Uppercase
{
    static void Main()
    {
        string text;

        System.Console.Write("Wprowadź tekst: ");
        text = System.Console.ReadLine();

        // NIEOCZEKIWANE: kod nie przekształca liter w zmiennej text na wielkie.
        text.ToUpper();

        System.Console.WriteLine(text);
    }
}
```

Dane wyjściowe 2.12 przedstawiają wyniki działania kodu z listingu 2.17.

DANE WYJŚCIOWE 2.12.

Wprowadź tekst: **To test systemu emisji audycji alarmowych.**
To test systemu emisji audycji alarmowych.

Na pozór wydaje się, że instrukcja `text.ToUpper()` powinna przekształcić znaki ze zmiennej `text` na wielkie litery. Jednak łańcuchy znaków są niezmienne, dlatego polecenie `text.ToUpper()` nie wprowadza oczekiwanych zmian. Zamiast tego instrukcja zwraca nowy łańcuch znaków, który trzeba zapisać w zmiennej lub bezpośrednio przekazać do metody `System.Console.WriteLine()`. Poprawiony kod przedstawiono na listingu 2.18, a wynik jego działania znajdziesz w danych wyjściowych 2.13.

Listing 2.18. Korzystanie z łańcuchów znaków

```
class Uppercase
{
    static void Main()
    {
        string text, uppercase;

        System.Console.Write("Wprowadź tekst: ");
        text = System.Console.ReadLine();

        // Zwraca nowy łańcuch znaków z wielkimi literami.
        uppercase = text.ToUpper();

        System.Console.WriteLine(uppercase);
    }
}
```

DANE WYJŚCIOWE 2.13.

Wprowadź tekst: **To test systemu emisji audycji alarmowych.**
TO TEST SYSTEMU EMISJI AUDYCJI ALARMOWYCH.

Jeśli zapomnisz o niezmienności łańcuchów znaków, możesz popełniać błędy podobne do tych z listingu 2.17 także przy korzystaniu z innych metod typu `string`.

Aby zmienić wartość zmiennej `text`, przypisz do niej wartość zwróconą przez metodę `ToUpper()`. Tę technikę przedstawiono poniżej:

```
text = text.ToUpper()
```

Typ `System.Text.StringBuilder`

Gdy potrzebne są znaczne modyfikacje łańcucha znaków, na przykład w trakcie tworzenia długich łańcuchów w wielu krokach, należy stosować typ `System.Text.StringBuilder` zamiast typu `string`. Typ `StringBuilder` obejmuje metody: `Append()`, `AppendFormat()`, `Insert()`, `Remove()` i `Replace()`. Niektóre z nich są dostępne także w typie `string`. Najważniejsza różnica polega na tym, że w typie `StringBuilder` te metody modyfikują dane w pierwotnym łańcuchu, zamiast zwracać nowy łańcuch znaków.

Wartości null i void

Dwa dodatkowe słowa kluczowe związane z typami to `null` i `void`. Wartość `null`, podawana za pomocą słowa kluczowego `null`, określa, że zmienna nie wskazuje żadnego prawidłowego obiektu. Słowo kluczowe `void` służy do określania braku typu lub braku jakiekolwiek wartości.

Słowo kluczowe `null`

Słowo kluczowe `null` może być też stosowane jako „literał” tekstowy. To słowo oznacza, że zmienna nie wskazuje żadnej wartości. Kod, który ustawia wartość zmiennej na `null`, jawnie przypisuje do niej wartość równą „niczemu”. Można też sprawdzić, czy zmienna ma tę wartość.

Przypisanie wartości `null` do typu referencyjnego nie jest równoważne brakowi przypisania. Zmienna o wartości `null` jest ustawiona. Zmienna bez żadnego przypisania nie jest ustawiona, dlatego próba jej użycia często spowoduje błąd komilacji.

Przypisanie do zmiennej typu `string` wartości `null` znacznie różni się też od przypisania pustego łańcucha znaków, `""`. Użycie słowa kluczowego `null` oznacza, że zmienna nie ma wartości, natomiast po przypisaniu łańcucha `""` zmienna ma wartość — pusty łańcuch znaków. Tego typu rozróżnienia bywają przydatne. Na przykład kod może interpretować zmienną `homePhone` o wartości `null` jako nieznany numer telefonu stacjonarnego, a tę samą zmienną o wartości `""` traktować jako brak numeru takiego telefonu.

ZAGADNIENIE DLA ZAAWANSOWANYCH

Początek
8.0
Początek
2.0

Modyfikator dopuszczający wartości `null`

Listing 2.19 ilustruje, jak przypisać wartość `null` do zmiennej typu `int`, dodając do deklaracji typu modyfikator dopuszczający wartość `null` (tym modyfikatorem jest znak zapytania — `?`).

Listing 2.19. Przypisywanie wartości `null` do zmiennej typu `int`

```
static void Main()
{
    int? age;
    // ...

    // Usuwanie wartości ze zmiennej age.
    age = null;

    // ...
}
```

Obsługa modyfikatora dopuszczającego wartość `null` została dodana w wersji C# 2.0. Wcześniej nie można było przypisywać wartości `null` do zmiennych omawianych do tej pory typów bezpośrednich; `null` można było przypisywać do zmiennych typów referencyjnych, na przykład do typu `string`. Więcej informacji o typach bezpośrednich i referencyjnych znajdziesz w rozdziale 3.

Ponadto przed pojawiением się wersji C# 8.0 typy referencyjne (na przykład `string`) domyślnie umożliwiał przypisywanie wartości `null`, dlatego nie można było opatrzyć typu referencyjnego modyfikatorem dopuszczającym takie wartości. Ponieważ i tak domyślnie można było przypisać wartość `null`, dodawanie tego modyfikatora było zbędne.

ZAGADNIENIE DLA ZAAWANSOWANYCH

Typy referencyjne dopuszczające wartości null

Przed wprowadzeniem wersji C# 8.0 wszystkie typy referencyjne domyślnie dopuszczały wartości `null`, dlatego nie istniała kategoria „typów referencyjnych dopuszczających wartości `null`” — wszystkie typy referencyjne tak działały. Jednak w C# 8.0 można skonfigurować obsługę wartości `null` w typach referencyjnych, aby domyślnie takie typy nie dopuszczały wartości `null` (można jednak dopuścić te wartości za pomocą modyfikatora). W ten sposób pojawiła się kategoria typów **referencyjnych dopuszczających wartości `null`** — są to typy z odpowiednim modyfikatorem. Gdy włączone jest rozróżnianie rodzajów typów referencyjnych, próba przypisania wartości `null` do zmiennej typu referencyjnego bez modyfikatora spowoduje ostrzeżenie.

Jednym typem referencyjnym opisanym do tej pory jest `string`. Jeśli skonfigurujesz obsługę modyfikatora dopuszczającego wartość `null` dla typów referencyjnych, możesz na przykład zadeklarować zmienną typu `string` w następujący sposób: `string? homeNumber = null;`.

Jednym ze sposobów na włączenie opisanego mechanizmu z C# 8.0 (i nowszych wersji) jest dodanie kodu `#nullable enable` w dowolnym wierszu przed użyciem typów referencyjnych dopuszczających wartości `null`.

„Typ” `void`

Czasem składnia języka C# wymaga określenia typu danych, ale żadne dane nie są przekazywane. Na przykład jeśli metoda nie musi zwracać żadnej wartości, w języku C# można podać `void` jako typ danych. Tę technikę zastosowano na przykład w deklaracji metody `Main` w programie `HelloWorld` (listing 1.1). Użycie słowa kluczowego `void` jako typu zwracanej wartości oznacza, że metoda nie zwraca żadnych danych. Jest to informacja dla kompilatora, by nie oczekivał wartości. Słowo kluczowe `void` nie oznacza rzeczywistego typu danych, ale określa, że żadne dane nie są zwracane.

Porównanie języków — C++

W językach C++ i C# słowo `void` ma dwa znaczenia — określa metodę, która nie zwraca żadnych danych, a także reprezentuje wskaźnik do lokalizacji z zawartością nieznanego typu. W programach w języku C++ stosunkowo często pojawiają się wskaźniki `void**`. W języku C# też można zapisywać wskaźniki do lokalizacji o wartości nieznanego typu za pomocą tej składni, ale jest to stosunkowo rzadko stosowana technika. Zwykle korzysta się z niej tylko w trakcie pisania programów współpracujących z bibliotekami z niezarządzanym kodem.

Porównanie języków — w języku Visual Basic odpowiednikiem zwracania wartości void jest zdefiniowanie podprocedury

W języku Visual Basic odpowiednikiem zwrócenia w C# wartości void jest zdefiniowanie podprocedury (Sub/End Sub) zamiast zwracającej wartość funkcji.

Konwersje typów danych

Ponieważ istnieją tysiące predefiniowanych typów w różnych wersjach platformy .NET, a w kodzie można zdefiniować dowolną liczbę dodatkowych typów, ważne jest, by w uzasadnionych sytuacjach możliwa była konwersja typów. Najczęściej wykonywaną operacją, która prowadzi do konwersji, jest **rzutowanie** (ang. *casting*).

Pomyśl o konwersji między dwoma typami liczbowymi — na przykład z typu long na typ int. Typ long może przechowywać wartości do 9 223 372 036 854 775 808, natomiast maksymalna wartość typu int to 2 147 483 647. Dlatego taka konwersja może skutkować utratą danych, gdy zmienna typu long zawiera wartość większą niż maksimum dla typu int. Każda konwersja, która może prowadzić do przycięcia wartości lub wyjątku (z powodu nieudanego przekształcenia wartości), wymaga **jawnego rzutowania**. Z kolei konwersje, które niezależnie od operandów nie skutkują przycięciem wartości ani zgłoszeniem wyjątku, można przeprowadzać w ramach **niejawnej konwersji**.

Jawne rzutowanie

W języku C# rzutowanie przeprowadza się za pomocą **operatora rzutowania**. Należy podać w nawiasie docelowy typ, na który należy przekształcić zmienną. W ten sposób programista określa, że wie, iż w trakcie jawnego rzutowania może nastąpić utrata precyzji lub danych albo zgłoszenie wyjątku. Kod z listingu 2.20 przekształca wartość typu long na typ int, przy czym programista jawnie nakazuje systemowi próbę wykonania tej operacji.

Listing 2.20. Przykład ilustrujący jawnie rzutowanie

```
long longNumber = 50918309109;
int intNumber = (int) longNumber;
Operator rzutowania
```

Z pomocą operatora rzutowania programista przekazuje kompilatorowi następującą informację: „Zaufaj mi; wiem, co robię. Jestem przekonany, że wartość zmieści się w docelowym typie”. Zastosowanie tej techniki powoduje, że kompilator dopuszcza konwersję. Jednak także przy jawniej konwersji może wystąpić błąd (wyjątek), jeśli przekształcanie danych nie zakończy się powodzeniem. Dlatego programista odpowiada za to, by zadbać o udaną konwersję danych lub udostępnić wykonywany w razie niepowodzenia niezbędny kod obsługi błędów.

ZAGADNIENIE DLA ZAAWANSOWANYCH

Konwersje kontrolowane i niekontrolowane

Język C# udostępnia specjalne słowa kluczowe służące do określania w blokach kodu, co powinno się stać, jeśli docelowy typ danych jest za mały na przypisywanie dane. Domyślnie jeśli docelowy typ danych nie wystarcza do zapisywania przypisywanych danych, wartość zostaje przycięta. Przyjrzyj się przykładowi z listingu 2.21.

Listing 2.21. Przepelenienie typu całkowitoliczbowego

```
class Program
{
    static void Main()
    {
        // int.MaxValue to 2147483647
        int n = int.MaxValue;
        n = n + 1 ;
        System.Console.WriteLine(n);
    }
}
```

Efekt przedstawiono w danych wyjściowych 2.14.

DANE WYJŚCIOWE 2.14.

-2147483648

Kod z listingu 2.21 wyświetla w konsoli wartość -2147483648. Jeśli jednak umieścisz kod w kontrolowanym bloku lub zastosujesz opcję checked przy uruchamianiu kompilatora, środowisko uruchomieniowe zgłosi wyjątek System.OverflowException. Do tworzenia kontrolowanego bloku służy słowo kluczowe checked, użyte na listingu 2.22.

Listing 2.22. Przykład ilustrujący kontrolowany blok kodu

```
class Program
{
    static void Main()
    {
        checked
        {
            // int.MaxValue jest równe 2147483647
            int n = int.MaxValue;
            n = n + 1 ;
            System.Console.WriteLine(n);
        }
    }
}
```

Efekt jest widoczny w danych wyjściowych 2.15.

DANE WYJŚCIOWE 2.15.

```
Unhandled Exception: System.OverflowException: Arithmetic operation
resulted in an overflow at Program.Main() in ...Program.cs:line 12
```

Efektem jest zgłoszenie wyjątku, jeśli w kontrolowanym bloku w czasie wykonywania programu nastąpi przypisanie prowadzące do przepełnienia zmiennej.

Kompilator języka C# udostępnia też opcję wiersza poleceń, która pozwala sprawić, by kod domyślnie był kontrolowany. Język C# obsługuje również tworzenie niekontrolowanych bloków, w których następuje przepełnienie danych zamiast zgłaszania wyjątku (zobacz listing 2.23).

Listing 2.23. Przykład ilustrujący niekontrolowany blok

```
using System;

class Program
{
    static void Main()
    {
        unchecked
        {
            // int.MaxValue jest równe 2147483647
            int n = int.MaxValue;
            n = n + 1 ;
            System.Console.WriteLine(n);
        }
    }
}
```

Efekt jest widoczny w danych wyjściowych 2.16.

DANE WYJŚCIOWE 2.16.

```
-2147483648
```

Nawet jeśli w trakcie komplikacji używana jest opcja włączająca kontrolowanie kodu, słowo kluczowe `unchecked` w przedstawionym kodzie zapobiega zgłoszeniu przez środowisko uruchomieniowe wyjątku w czasie wykonywania kodu.

Możliwe, że zastanawiasz się, dlaczego w wersji niekontrolowanej po dodaniu 1 do wartości `int.MaxValue` wynik wynosi `-2147483648`. Powodem jest mechanizm „zawijania” wartości. Binarna reprezentacja wartości `int.MaxValue` to `01111111111111111111111111111111`, gdzie pierwsza cyfra (0) oznacza liczbę dodatnią. Zwiększenie tej wartości daje następną liczbę, `10000000000000000000000000000000`. Jest to najmniejsza liczba całkowita (`int.MinValue`), a pierwsza cyfra (1) oznacza, że wartość jest ujemna. Dodanie 1 do `int.MinValue` da liczbę `10000000000000000000000000000000` (czyli `-2147483647`) itd.

W przypadku gdy jawnie zażadasz konwersji za pomocą operatora rzutowania, nie zawsze można przekształcić wartość określonego typu na dowolny inny typ. Kompilator i tak sprawdza, czy operacja konwersji jest poprawna. Nie można na przykład przekształcić wartości typu `long` na wartość typu `bool`. Taka konwersja nie jest zdefiniowana, dlatego kompilator nie zezwoli na rzutowanie.

Porównanie języków — przekształcanie liczb na wartości logiczne

Może Cię dziwić to, że w C# nie jest dozwolone rzutowanie z typu liczbowego na typ logiczny, które jest tak powszechnie w wielu innych językach. Powodem, dla którego taka konwersja nie jest dostępna w C#, jest chęć uniknięcia wieloznaczności. Na przykład czy wartość -1 odpowiada wartości true, czy false? Co ważniejsze, jak wyjaśniono w następnym rozdziale, to ograniczenie zmniejsza ryzyko użycia operatora przypisania zamiast operatora równości. Pozwala to na przykład uniknąć zapisania instrukcji if ($x=42$) {...} zamiast polecenia if ($x==42$) {...}.

Konwersja niejawnna

W innych sytuacjach, na przykład przy przekształcaniu wartości z typu int na typ long, nie dochodzi do utraty precyzji i zmiany wartości. Wtedy w kodzie wystarczy zastosować operator przypisania, a konwersja zachodzi **niejawnie**. Oznacza to, że kompilator może ustalić, iż taka konwersja się powiedzie. W kodzie z listingu 2.24 w celu przeprowadzenia konwersji z typu int na long wystarczy zastosować operator przypisania.

Listing 2.24. Brak operatora rzutowania przy niejawnym rzutowaniu

```
int intNumber = 31416;
long longNumber = intNumber;
```

Nawet gdy operator rzutowania nie jest niezbędny (ponieważ dozwolona jest niejawną konwersją), można go dodać (tak jak na listingu 2.25).

Listing 2.25. Używanie operatora rzutowania przy przeprowadzaniu niejawnnej konwersji

```
int intNumber = 31416;
long longNumber = (long) intNumber;
```

Konwersja typów bez rzutowania

W języku nie ma zdefiniowanej konwersji z łańcuchów znaków na typy liczbowe, dlatego trzeba stosować metody takie jak Parse(). Każdy typ liczbowy udostępnia funkcję Parse(), która umożliwia konwersję z łańcuchów znaków na dany typ liczbowy. Listing 2.26 ilustruje wywołanie tej funkcji.

Listing 2.26. Używanie funkcji int.Parse() do przekształcenia wartości typu string na liczbowy typ danych

```
string text = "9.11E-31";
float kgElectronMass = float.Parse(text);
```

Dostępny jest też specjalny typ służący do przekształcania wartości różnych typów. Jest to typ System.Convert. Przykładowy kod, który z niego korzysta, pokazano na listingu 2.27.

Listing 2.27. Konwersja typów za pomocą typu System.Convert

```
string middleCText = "261.626";
double middleC = System.Convert.ToDouble(middleCText);
bool boolean = System.Convert.ToBoolean(middleC);
```

Typ `System.Convert` obsługuje tylko niewielką liczbę typów i nie jest rozszerzalny. Umożliwia on konwersję między wartościami następujących typów: `bool`, `char`, `sbyte`, `short`, `int`, `long`, `ushort`, `uint`, `ulong`, `float`, `double`, `decimal`, `DateTime` i `string`.

Ponadto wszystkie typy udostępniają metodę `ToString()`, którą można wykorzystać do utworzenia tekstowej reprezentacji typu. Na listingu 2.28 pokazano, jak zastosować tę metodę. Efekt przedstawiono w danych wyjściowych 2.17.

Listing 2.28. Używanie metody `ToString()` w celu konwersji danych na typ `string`

```
bool boolean = true;
string text = boolean.ToString();
// Wyświetla "True"
System.Console.WriteLine(text);
```

DANE WYJŚCIOWE 2.17.

True

Dla większości typów metoda `ToString()` zwraca nazwę typu zamiast tekstowej reprezentacji danych. Tekstowa reprezentacja jest zwracana tylko wtedy, gdy typ ma jawnie zaimplementowaną metodę `ToString()`. Warto też zauważyć, że można pisać niestandardowe metody przeznaczone do konwersji typów. Wiele takich metod jest dostępnych dla klas w środowisku uruchomieniowym.

ZAGADNIENIE DLA ZAAWANSOWANYCH

Początek
2.0

Metoda TryParse()

Od wersji C# 2.0 (platforma .NET 2.0) wszystkie bezpośrednie typy liczbowe udostępniają statyczną metodę `TryParse()`. Działa ona bardzo podobnie do metody `Parse()`, ale zamiast zgłaszać po nieudanej konwersji wyjątek, zwraca wartość `false`, co pokazano na listingu 2.29.

Listing 2.29. Używanie metody `TryParse()`, by uniknąć wyjątku spowodowanego nieprawidłowym rzutowaniem

```
double number;
string input;

System.Console.Write("Wprowadź liczbę: ");
input = System.Console.ReadLine();
if (double.TryParse(input, out number))
{
    // Poprawna konwersja — teraz można korzystać z liczby
    // ...
}
```

```

else
{
    System.Console.WriteLine(
        "Wprowadzony tekst nie jest poprawną liczbą.");
}

```

Efekt działania kodu z listingu 2.29 przedstawiają dane wyjściowe 2.18.

DANE WYJŚCIOWE 2.18.

Wprowadź liczbę: czterdzieści-dwa
Wprowadzony tekst nie jest poprawną liczbą.

Wynikowa wartość przetworzona przez kod z wejściowej wartości typu string jest zwracana w parametrze out. Tu zwracana wartość jest typu number.

Metoda TryParse() jest dostępna nie tylko dla różnych typów liczbowych, ale też dla typów wyliczeniowych.

Warto zauważyć, że od wersji C# 7.0 nie trzeba już deklarować zmiennej przed jej użyciem jako argumentu typu out. Dzięki temu zmienną number można zadeklarować tak jak na listingu 2.30.

Listing 2.30. Używanie metody TryParse() z wewnętrzierszą deklaracją zmiennej typu out (C# 7.0)

```

// double number;
string input;

System.Console.Write("Wprowadź liczbę: ");
input = System.Console.ReadLine();
if (double.TryParse(input, out double number))
{
    System.Console.WriteLine(
        $"Wprowadzony tekst został poprawnie przekształcony na liczbę {number}.");
}
else
{
    // Uwaga: także tu zmienna number znajduje się w zasięgu (choć nie ma przypisanej wartości).
    System.Console.WriteLine(
        "Wprowadzony tekst nie jest poprawną liczbą.");
}
System.Console.WriteLine(
    $"'{number}' ma teraz wartość: { number }");

```

Warto zauważyć, że typ danych zmiennej number jest podany po modyfikatorze out, a przed deklarowaną zmienną. W efekcie zmienna number jest dostępna w obu blokach instrukcji if (dla wartości true i false), a nawet poza tą instrukcją.

Wałą różnicą między metodami Parse() i TryParse() jest to, że metoda TryParse() nie zgłasza wyjątku po niepowodzeniu. Często konwersja z typu string na typ liczbowy odbywa się po wprowadzeniu tekstu przez użytkownika. W takiej sytuacji można oczekiwać, że użytkownik poda nieprawidłowe dane, których nie da się poprawnie przetworzyć. Jeśli zastosujesz metodę TryParse() zamiast Parse(), będziesz mógł uniknąć zgłaszania wyjątków w oczekiwanych sytuacjach. Tu oczekiwana sytuacją jest wprowadzenie przez użytkownika nieprawidłowych danych. Warto unikać zgłaszania wyjątków w takich przewidywalnych scenariuszach.

Podsumowanie

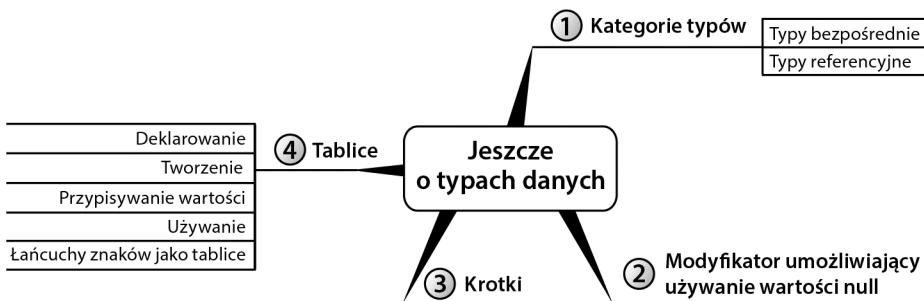
Język C# zawiera kilka konstrukcji nowych nawet dla doświadczonych programistów. W podrozdziale poświęconym typom danych opisano na przykład typ `decimal`, z którego można korzystać w trakcie wykonywania obliczeń finansowych bez anomalii występujących w typach zmiennoprzecinkowych. Ponadto w rozdziale wyjaśniono, że typ logiczny `bool` nie umożliwia bezpośredniej konwersji na typ całkowitoliczbowy (nie jest też możliwa konwersja w drugą stronę). Chroni to przed błędem użycia operatora przypisania w wyrażeniach warunkowych. Inne cechy języka C#, odróżniające go od wielu wcześniejszych języków, to modyfikator służący do tworzenia dosłownych łańcuchów znaków (@; powoduje on, że w łańcuchu ignorowany jest znak ucieczki), interpolacja łańcuchów znaków (poprawiająca czytelność kodu, ponieważ pozwala umieszczać zmienne w łańcuchach) i niezmienny charakter typu `string`.

Rozdział 3. to dalszy opis typów danych. Omówiono tam dwa rodzaje typów danych: typy bezpośrednie i typy referencyjne. Oprócz tego wyjaśniono, jak łączyć elementy danych w krotki i tablice.

3

Jeszcze o typach danych

W ROZDZIALE 2. OMÓWIŁEM wszystkie typy wbudowane języka C#, a także wspomniałem o typach referencyjnych i bezpośrednich. W tym rozdziale znajdziesz ciąg dalszy omówienia typów danych i dodatkowe objaśnienie rodzajów typów.



Oprócz tego opisane jest tu łączenie elementów danych w krotki — mechanizm wprowadzony w C# 7.0 — i grupowanie danych w zbiory nazywane **tablicami**. Najpierw jednak należy poznać typy bezpośrednie i referencyjne.

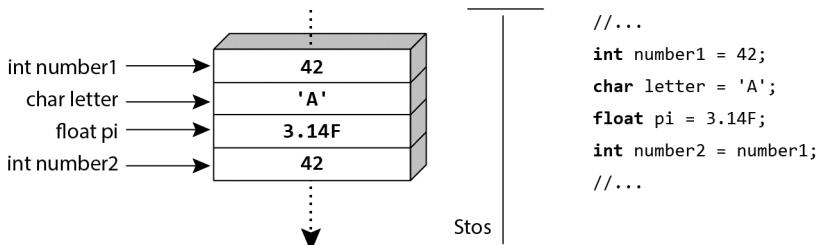
Kategorie typów

Wszystkie typy należą do jednej z dwóch kategorii — **typów bezpośrednich** (ang. *value types*) i **typów referencyjnych** (ang. *reference types*). Różnice między tymi kategoriami wynikają ze sposobu kopiowania wartości. Typy bezpośrednie są zawsze kopowane przez wartość, natomiast typy referencyjne — przez referencję.

Typy bezpośrednie

Prawie wszystkie (z wyjątkiem typu `string`) predefiniowane typy używane w książce do tego miejsca to typy bezpośrednie. Zmienne typów bezpośrednich przechowują wartość bezpośrednio. Oznacza to, że zmienna wskazuje do tej samej lokalizacji w pamięci, w której zapisana

jest wartość. Dlatego gdy do nowej zmiennej przypisywana jest wartość innej zmiennej, ta wartość zostaje skopiowana do lokalizacji nowej zmiennej. Druga zmienna tego samego typu bezpośredniego nie może wskazywać lokalizacji w pamięci, do której prowadzi już pierwsza zmienna. Zmiana wartości pierwszej zmiennej nie wpływa więc na wartość drugiej zmiennej. Ilustruje to rysunek 3.1. Na tym rysunku zmienna number1 prowadzi do określonej lokalizacji w pamięci, gdzie zapisana jest wartość 42. Po przypisaniu zmiennej number1 do number2 obie zmienne zawierają wartość 42. Jednak zmodyfikowanie wartości jednej zmiennej nie wpływa na wartość drugiej.



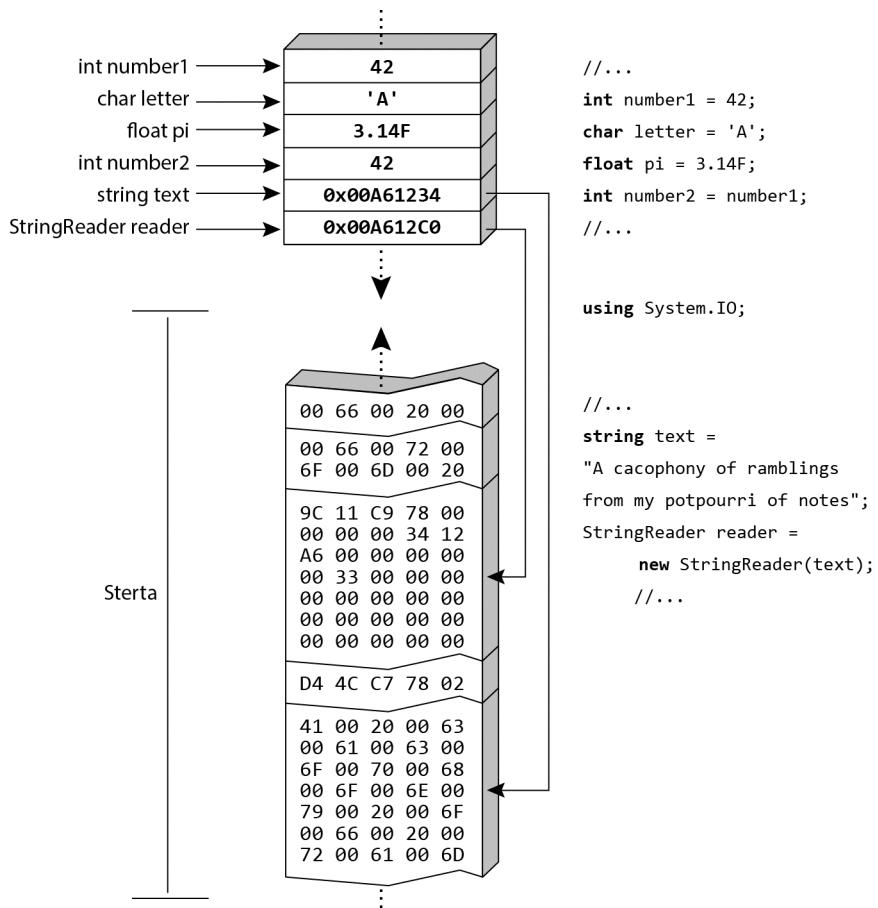
Rysunek 3.1. Typy bezpośrednie przechowują dane bezpośrednio

Podobnie przekazanie wartości typu bezpośredniego do metody takiej jak `Console.WriteLine()` spowoduje utworzenie kopii danych w pamięci. Wszelkie zmiany parametru w metodzie nie wpłyną na pierwotną wartość używaną w funkcji wywołującej. Ponieważ typy bezpośrednie wymagają tworzenia kopii w pamięci, zwykle powinny zajmować niewiele miejsca. W prawie wszystkich sytuacjach powinny zajmować nie więcej niż 16 bajtów.

Typy referencyjne

Wartość typu referencyjnego to referencja do miejsca w pamięci zawierającego dane. Typy referencyjne przechowują referencję określającą lokalizację danych. W odróżnieniu od typów bezpośrednich nie zawierają samych danych. Dlatego by uzyskać dostęp do danych, środowisko uruchomieniowe odczytuje ze zmiennej lokalizację w pamięci, a następnie przeskakuje do tej lokalizacji, gdzie zapisane są dane (ta operacja to **derefencja**). Obszar pamięci z danymi, do których prowadzą typy referencyjne, to **sterta** (zobacz rysunek 3.2).

Typy referencyjne nie wymagają tworzenia w pamięci kopii danych, co jest konieczne dla typów bezpośrednich. Dlatego kopiowanie wartości typów referencyjnych odbywa się dużo szybciej niż kopiowanie dużych wartości typów bezpośrednich. Gdy przypisujesz wartość jednej zmiennej typu referencyjnego do innej takiej zmiennej, kopiowana jest tylko referencja, a nie wskazywane przez nią dane. W praktyce referencje prawie zawsze mają wielkość odpowiadającą natywnej wielkości słowa procesora. Oznacza to, że procesory 32-bitowe kopiują referencje 32-bitowe, procesory 64-bitowe kopiują referencje 64-bitowe itd. Kopiowanie niewielkiej referencji do dużego bloku danych jest oczywiście szybsze niż kopiowanie całego bloku, co jest konieczne dla typów bezpośrednich.



Rysunek 3.2. Typy referencyjne wskazują na lokalizację na stercie

Ponieważ gdy stosuje się typy referencyjne, kopowane są referencje do danych, dwie różne zmienne mogą wskazywać te same dane. Jeśli dwie zmienne prowadzą do tego samego obiektu, zmiana pola w obiekcie przy użyciu jednej zmienniej powoduje, że efekt będzie widoczny przy dostępie do tego pola za pomocą drugiej zmiennej. Dotyczy to zarówno przypisań, jak i wywołań metod. Metoda może zmienić dane typu referencyjnego, a ta zmiana będzie widoczna po zwróceniu sterowania do jednostki wywołującej. Dlatego ważnym czynnikiem, który należy uwzględnić w trakcie podejmowania decyzji o zdefiniowaniu typu referencyjnego lub typu bezpośredniego, jest charakter obiektu. Jeśli obiekt to logicznie niezmienna wartość o stałym rozmiarze, zwykle należy zastosować typ bezpośredni, a jeżeli jest logicznie zmienny, prawdopodobnie warto utworzyć typ referencyjny.

Oprócz typu `string` i klas niestandardowych (na przykład `Program`) wszystkie typy omawiane do tej pory to typy bezpośrednie. Jednak większość typów to typy referencyjne. Choć możliwe jest definiowanie niestandardowych typów bezpośrednich, są one tworzone rzadziej niż niestandardowe typy referencyjne.

Początek
8.0
Początek
2.0

Deklarowanie typów umożliwiających stosowanie wartości null

Często trzeba jakoś przedstawić brakujące lub nieznane wartości. Na przykład gdy określasz liczbę, jaką wartość podasz, jeśli liczba jest nieznana lub nieprzypisana? Możliwe jest użycie „magicznej” wartości, na przykład -1 lub `int.MaxValue`, są to jednak poprawne liczby całkowite, dlatego niejednoznaczne jest, czy reprezentują one zwykłą wartość typu `int`, czy brak wartości. Zamiast tego lepiej przypisać wartość `null`, aby zaznaczyć, że dana wartość jest nieprawidłowa lub nie została jeszcze przypisana. Przypisywanie wartości `null` jest przydatne zwłaszcza w kontekście korzystania z baz danych. Kolumny w tabelach baz danych często dopuszczają stosowanie wartości `null`. Pobieranie wartości z takich kolumn i przypisywanie ich do powiązanych pól w kodzie w języku C# sprawia problemy, chyba że dane pole może przyjmować wartość `null`.

Za pomocą **modyfikatora dopuszczającego wartość null** można zadeklarować typ jako akceptujący lub nieakceptujący wartości `null`. W C# ten modyfikator jest dostępny dla typów bezpośrednich od wersji C# 2.0, a dla typów referencyjnych od wersji C# 8.0. Aby włączyć obsługę wartości `null`, po deklaracji typu dodaj modyfikator dopuszczający takie wartości (znak zapytania podawany bezpośrednio po nazwie typu). Na przykład zapis `int? number = null` deklaruje zmienną typu `int` dopuszczającą wartości `null` i przypisuje zmiennej tę wartość. Niestety, dopuszczanie wartości `null` związane jest z kilkoma pułapkami i wymaga specjalnego uwzględniania tego mechanizmu w kodzie.

Dereferencja referencji null

Choć mechanizm przypisywania wartości `null` do zmiennych jest bardzo wartościowy (to celowa gra słów), ma też wady. Kopiowanie lub przekazywanie wartości `null` do innych zmiennych i metod nie powoduje problemów, jednak dereferencja (wywołanie składowej) instancji równej `null` skutkuje wyjątkiem `System.NullReferenceException`. Dotyczy to na przykład wywołania `text.GetType()`, gdy `text` ma wartość `null`. Zgłoszenie wyjątku `System.NullReferenceException` przez kod produkcyjny oznacza błąd. Ten wyjątek wskazuje na to, że programista piszący kod nie pamiętał o sprawdzeniu wartości `null` przed wywołaniem. Ten problem jest potęgowany przez to, że sprawdzanie wartości `null` wymaga od programisty wiedzy o tym, iż taka wartość może wystąpić, dlatego trzeba podjąć odpowiednie kroki. To dlatego deklarowanie zmiennych dopuszczających wartość `null` wymaga zastosowania modyfikatora — zmienne nie akceptują wtedy domyślnie wartości `null` (zobacz punkt „Typy referencyjnych dopuszczające wartości null”). Gdy programista chce dopuścić przypisywanie wartości `null` do zmiennej, bierze dodatkową odpowiedzialność za unikanie dereferencji zmiennych o takiej wartości.

Ponieważ sprawdzanie wartości `null` wymaga stosowania instrukcji i operatorów, które nie zostały jeszcze omówione, opis tego zadania znajdziesz w ramce ZAGADNIENIE DLA ZAAWANSOWANYCH „Sprawdzanie wartości null”. Kompletne objaśnienia zawiera rozdział 4.

ZAGADNIENIE DLA ZAAWANSOWANYCH

Sprawdzanie wartości null

Jest wiele instrukcji i operatorów, z jakich programiści mogą korzystać do sprawdzania wartości null. Na listingu 3.1 pokazanych jest kilka przykładów. Najprostsza technika polega na użyciu instrukcji `if` i operatora `is`, co ilustruje listing 3.1.

Listing 3.1. Sprawdzanie wartości null

```
static void Main()
{
    int? number = null;
    // ...
    if (number is null)
    {
        System.Console.WriteLine(
            "'number' wymaga wartości i musi być różna od null");
    }
    else
    {
        System.Console.WriteLine(
            $"Podwojona wartość 'number': { number * 2 }.");
    }
}
```

Instrukcja `if` sprawdza tu, czy zmienna `number` ma wartość `null`, i w zależności od sytuacji wykonuje różne czynności. Choć możesz użyć operatora równości (`==`), to za pomocą przesłaniania można zmienić sposób jego działania. Dlatego lepiej jest zastosować operator `is`.

Innym przydatnym operatorem (dodanym w C# 6.0) związany z wartościami `null` jest operator `?..`. Ten operator przed dereferencją sprawdza, czy zmienna ma wartość `null`. Na przykład instrukcja `int? length = text?.Length` automatycznie zwróci `null`, jeśli zmienna `text` jest równa `null`, a w przeciwnym razie zapisze długość łańcucha znaków przechowywanego w zmiennej `text`. Zauważ, że ponieważ wartość zwracana przez wyrażenie `text?.Length` może być równa `null` (jeśli `text` to `null`), zmienna `length` musi dopuszczać wartości `null`.

Początek
6.0

Instrukcja `if` i operator `?..` są opisane szczegółowo w rozdziale 4. Operator `is` jest po krótkie przedstawiony w rozdziale 4., ale szczegółowo omówiony zostanie dopiero w rozdziale 7. w kontekście dopasowywania wzorców.

Koniec
6.0

Typy bezpośrednie dopuszczające wartość null

Zmienne typu bezpośredniego zawierają konkretne wartości, dlatego nie mogą być równe `null`, ponieważ z definicji nie mogą przechowywać referencji (w tym referencji do „niczego”). Mimo to używam zwrotu „dereferencja typu bezpośredniego”, gdy piszę o wywoływaniu składowych takich typów. Choć technicznie takie podejście nie jest w pełni poprawne, określenie „dereferencja” na wywoływanie składowych (niezależnie od tego, czy typ jest bezpośredni, czy referencyjny) stosuje się dość często¹.

¹ Typy bezpośrednie dopuszczające wartości null zostały wprowadzone w wersji C# 2.0.

ZAGADNIENIE DLA ZAAWANSOWANYCH I POCZĄTKUJĄCYCH

Dereferencja wartości null w typach bezpośrednich

W ujęciu technicznym typy bezpośrednie deklarowane z użyciem modyfikatora dopuszczającego wartości null nadal są typami bezpośredniimi. Dlatego choć działają tak, jakby były równe null, w rzeczywistości mają inną wartość. Z tego powodu dereferencja zmiennej typu bezpośredniego równej null nie skutkuje wyjątkiem `NullReferenceException`. Dla typu `Nullable<T>` zaimplementowane są składowe takie jak `HasValue`, `ToString()`, a nawet składowe związane z równością (`GetHashCode()` i `Equals()`), dlatego nie powodują one wyjątków dla wartości reprezentujących null. Dereferencja zmiennej typu bezpośredniego równej null skutkuje wyjątkiem `InvalidOperationException` (zamiast `NullReferenceException`), aby przypomnieć programistom, że powinni sprawdzać wartość przed dereferencją. Z kolei wywołanie `GetType()` dla wartości równej null prowadzi do wyjątku `NullReferenceException`. Ta niespójność wynika z tego, że `GetType()` nie jest metodą wirtualną, dlatego nie można jej przeciążyć w typie `Nullable<T>`. Pozostaje więc domyślne rozwiązanie — zgłaszanie wyjątku `NullReferenceException`.

Koniec
2.0

Typy referencyjne dopuszczające wartość null

Przed wersją C# 8.0 wszystkie typy referencyjne dopuszczały wartość null. Niestety, skutkowało to licznymi błędami, ponieważ uniknięcie wyjątku związanego z referencją null wymagało od programisty, aby zdał sobie sprawę z konieczności sprawdzenia wartości null i zastosował programowanie defensywne, by uniknąć dereferencji takich wartości. Problem jest dodatkowo nasilony przez to, że typy referencyjne domyślnie dopuszczają wartości null. Jeśli do zmiennej typu referencyjnego nie zostanie przypisana żadna wartość, zmienna domyślnie będzie równa null. Dereferencja zmiennej lokalnej typu referencyjnego bez przypisanej wartości powoduje (słuszne) zgłoszenie przez kompilator błędu "Użyto nieprzypisanej zmiennej lokalnej 'text'". Najprościej przypisać wtedy wartość null w deklaracji zmiennej, zamiast ustawać konkretną wartość odpowiednią niezależnie od dalszego przebiegu programu (zobacz listing 3.2). Programiści mogą więc łatwo wpaść w pułapkę deklarowania zmiennej i przypisywania jej wartości null, ponieważ jest to najłatwiejsze rozwiązanie problemu; takie podejście wynika z założenia (możliwe, że błędnego), iż kod przypisze nową wartość zmiennej przed jej dereferencją.

Listing 3.2. Dereferencja zmiennej bez przypisanej wartości

```
#nullable enable
static void Main()
{
    string? text;
    // ...
    // Błąd komplikacji: Użyto nieprzypisanej zmiennej lokalnej 'text'
    System.Console.WriteLine(text.length);
}
```

Podsumowując: domyślne dopuszczanie wartości null w typach referencyjnych było częstym źródłem błędów powodujących wyjątki System.NullReferenceException, a sposób działania kompilatora zachęcał programistów do stosowania niewłaściwego podejścia za miast podjęcia odpowiednich działań w celu uniknięcia pułapki.

Aby znacznie ułatwić programistom pracę, zespół rozwijający język C# wprowadził w wersji C# 8.0 **typy referencyjne dopuszczające wartość null** (co oczywiście pozwala się domyślić, że typy referencyjne mogą też nie dopuszczać takich wartości). Typy referencyjne dopuszczające wartość null upodabniają typy referencyjne do bezpośrednich pod tym względem, że w deklaracjach typów referencyjnych można teraz stosować (lub pomijać) modyfikator dopuszczający wartości null. W C# 8.0 po wyłączeniu obsługi typów referencyjnych dopuszczających wartość null zadeklarowanie zmiennej bez modyfikatora dopuszczającego wartość null oznacza, że zmienna nie powinna akceptować takich wartości.

Niestety, obsługa deklarowania typów referencyjnych z modyfikatorem dopuszczającym wartość null i domyślne traktowanie zmiennych bez tego modyfikatora jako nieakceptujących wartości null ma poważny wpływ na aktualizowany kod z wcześniejszych wersji języka C#. Ponieważ w C# 7.0 i starszych wersjach null można było przypisywać do wszystkich zmiennych typów referencyjnych (na przykład `string text = null`), to czy oznacza to, że całego takiego kodu nie da się skompilować w C# 8.0?

Na szczęście zgodność wstecz jest niezwykle istotna dla zespołu rozwijającego C#, dlatego obsługa kontroli dopuszczania wartości null nie jest domyślnie włączona. Można ją aktywować na dwa sposoby: za pomocą dyrektywy `#nullable` i właściwości projektu.

Oto jak aktywować mechanizm kontroli dopuszczania wartości null przy użyciu dyrektywy `#nullable`:

```
#nullable enable
```

Ta dyrektywa przyjmuje wartości `enable`, `disable` i `restore` (ta ostatnia przywraca ustawienia obowiązujące na poziomie projektu). Listing 3.2 pokazuje, jak włączyć kontrolę dopuszczania wartości null za pomocą dyrektywy `nullable` i ustawienia `enable`. Następnie można zadeklarować zmienną `text` za pomocą typu `string?` i nie spowoduje to ostrzeżeń ze strony kompilatora.

Inna możliwość to zastosowanie właściwości projektu. Domyślnie w pliku projektu (`*.csproj`) właściwość `Nullable` nie jest aktywna. Aby ją włączyć, dodaj właściwość projektu `Nullable` o wartości `enable`, tak jak na listingu 3.3.

Listing 3.3. Włączanie kontroli dopuszczania wartości null na poziomie projektu w pliku csproj

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp3.0</TargetFramework>
    <Nullable>enable</Nullable>
  </PropertyGroup>
</Project>
```

W całym przykładowym kodzie powiązanym z tą książką (dostępnym na stronie <https://github.com/EssentialCSharp> i w witrynie wydawnictwa Helion) właściwość `Nullable` jest włączona na poziomie projektu. Możesz też ustawić tę właściwość w wierszu polecień za pomocą narzędzia dotnet, używając argumentu `/p:`

Koniec
8.0 dotnet build /p:Nullable=enable

Wartość właściwości `Nullable` podana w wierszu polecień zastępuje ustawienia z pliku projektu.

Początek 3.0 Zmienne lokalne z niejawnie określonym typem danych

W C# 3.0 dodano kontekstowe słowo kluczowe `var`, które służy do deklarowania **zmiennych lokalnych z niejawnie określonym typem danych**. Jeśli zmienna jest inicjowana na etapie deklaracji i za pomocą wyrażenia o jednoznacznie określonym typie, w C# 3.0 i nowszych wersjach typ zmiennej może zostać wywnioskowany i nie trzeba go podawać. Ilustruje to listing 3.4.

Listing 3.4. Używanie łańcuchów znaków

```
class Uppercase
{
    static void Main()
    {
        System.Console.Write("Wprowadź tekst: ");
        var text = System.Console.ReadLine();

        // Zwarcanie nowego łańcucha znaków obejmującego wielkie litery
        var uppercase = text.ToUpper();

        System.Console.WriteLine(uppercase);
    }
}
```

Ten kod różni się od wersji z listingu 2.18 w dwóch aspektach. Po pierwsze, na listingu 3.4 zamiast bezpośrednio określać w deklaracji typ danych `string`, wykorzystano słowo `var`. Wynikowy kod CIL jest identyczny jak przy jawnym używaniu typu `string`. Jednak słowo `var` informuje kompilator, że typ danych należy ustalić na podstawie przypisywanej w deklaracji wartości (zwracanej przez metodę `System.Console.ReadLine()`).

Po drugie, zmienne `text` i `uppercase` są inicjowane w deklaracji. Pominięcie inicjacji spowodowałoby błąd w czasie komplikacji. Wcześniej wspomniano, że kompilator określa typ danych wyrażenia inicjującego zmienną i na tej podstawie deklaruje zmienną. Efekt jest podobny jak wtedy, gdy programista jawnie określa typ.

Choć używanie słowa `var` zamiast jawnego określania typu danych jest dopuszczalne, staraj się unikać tej techniki, gdy typ danych jest znany. Na przykład w deklaracjach zmiennych `text` i `uppercase` lepiej podać typ danych `string`. Nie tylko sprawia to, że kod staje się bardziej zrozumiały, ale też pozwala sprawdzić, czy typ danych zwrócony przez wyrażenie zapisane po prawej stronie jest zgodny z oczekiwaniami. Dla zmiennych deklarowanych za pomocą słowa `var` typ danych wyrażenia po prawej stronie powinien być oczywisty. Jeśli jest inaczej, pomyśl o rezygnacji z deklaracji ze słowem `var`.

Wskazówki

UNIKAJ stosowania zmiennych lokalnych z niejawnie określonym typem, chyba że typ danych przypisywanej wartości jest oczywisty.

Porównanie języków — słowa void*, Variant i var w językach C++, Visual Basic i JavaScript

Zmienna o niejawnie określonym typie nie jest odpowiednikiem zmiennych void* z języka C++, Variant z języka Visual Basic lub var z JavaScriptu. W wymienionych językach te deklaracje pozwalają przypisać do zmiennych wartość dowolnego typu. W języku C# podobnie działa zmienna zadeklarowana za pomocą typu object. Jednak zmienne var w języku C# mają typ jednoznacznie określany przez kompilator. Po ustaleniu typu nie może się on zmieniać, a sprawdzanie typu i wywołań składowych odbywa się na etapie komplikacji.

3.0

ZAGADNIENIE DLA ZAAWANSOWANYCH

Typy anonimowe

Obsługę słowa var dodano w wersji C# 3.0, by umożliwić stosowanie typów anonimowych. Są to typy danych deklarowane „w locie” w metodzie, a nie za pomocą jawnych definicji klas. Przykładowy typ anonimowy przedstawiono na listingu 3.5, a szczegółowe informacje o takich typach znajdziesz w rozdziale 15.

Listing 3.5. Zmienne lokalne o niejawnie określonym typie (w kodzie wykorzystano typy anonimowe)

```
class Program
{
    static void Main()
    {
        var patent1 =
            new { Title = "Soczewki dwuogniskowe",
                  YearOfPublication = "1784" };
        var patent2 =
            new { Title = "Fonograf",
                  YearOfPublication = "1877" };

        System.Console.WriteLine(
            $"{ patent1.Title } ({ patent1.YearOfPublication })");
        System.Console.WriteLine(
            $"{ patent2.Title } ({ patent2.YearOfPublication })");
    }
}
```

Efekt działania kodu przedstawiono w danych wyjściowych 3.1.

DANE WYJŚCIOWE 3.1.

Soczewki dwuogniskowe (1784)
Fonograf (1877)

Na listingu 3.5 pokazano przypisywanie wartości typu anonimowego do zmiennej lokalnej o niejawnie określonym typie (zapewnia to słowo kluczowe var). Ta operacja daje wartościowe możliwości w połączeniu z dostępną od wersji C# 3.0 obsługą łączenia typów danych, a także umożliwia zmniejszenie wielkości określonego typu poprzez ograniczenie liczby elementów z danymi. W C# 7.0 wprowadzono składnię tworzenia krotek, dzięki której stosowanie typów anonimowych jest zbędne.

Krotki

Czasem przydatne jest łączenie elementów danych ze sobą. Weźmy na przykład informacje na temat najbiedniejszego państwa na świecie w 2019 r. Jest to Sudan Południowy. Stolicą tego państwa jest Juba, a produkt krajowy brutto na mieszkańca wynosi 275,18 dolara. Za pomocą omówionych do tej pory technik można zapisać każdą z tych informacji w odrębnej zmiennej. Jednak takie dane nie będą ze sobą powiązane. Oznacza to, że wartość 275,18 dolara nie będzie powiązana z nazwą Sudan Południowy — chyba że przy użyciu identycznego przedrostka lub przyrostka w nazwach zmiennych. Inna możliwość to połączenie wszystkich danych w jeden łańcuch znaków. Wadą tego rozwiązania jest to, że każdy element danych musi wtedy zostać pobrany z takiego łańcucha.

W C# 7.0 dostępna jest trzecia możliwość — użycie krotek. Krotki pozwalają połączyć operacje przypisania wartości zmiennych w jedną instrukcję, taką jak ta:

```
(string country, string capital, double gdpPerCapita) =  
    ("Sudan Południowy", "Juba", 275.18);
```

Dostępne są też inne mechanizmy składniowe związane z krotkami. Znajdziesz je w tabeli 3.1.

W pierwszych czterech przykładach prawa strona reprezentuje krotkę, a po lewej znajdują się poszczególne zmienne, do których jednocześnie przypisywane są wartości za pomocą **składni dla krotek**. Składnia ta obejmuje nawias z przynajmniej dwoma elementami rozdzielonymi przecinkami. Używam tu określenia *składnia dla krotek*, ponieważ typ danych generowany przez kompilator dla wyrażenia występującego po lewej stronie nie jest technicznie krotką. Dlatego choć zaczynamy od wartości połączonych po prawej stronie jak krotka, przypisanie po lewej stronie rozbija tę krotkę na osobne elementy. W przykładzie 2. przypisanie dotyczy wcześniej zadeklarowanych zmiennych. W przykładach 1., 3. i 4. zmienne są deklarowane w wyrażeniu ze składnią dla krotek. Ponieważ deklarowane są tu tylko zmienne, ich nazwy i wielkość liter są zgodne ze wskazówkami z rozdziału 1., np.: „STOSUJ notację Wielbłąda w nazwach zmiennych lokalnych”.

Warto zauważać, że choć pośrednie ustalanie typów (z wykorzystaniem słowa var) można zastosować jednocześnie do wszystkich deklaracji zmiennych w składni dla krotek, co ilustruje przykład 4., nie można zrobić tego samego w przypadku jawnie podawanego typu (np. string). Ponieważ w krotkach każdy element może być innego typu danych, przypisanie jednego typu do wszystkich elementów może spowodować błąd, chyba że każdy element rzeczywiście jest tego samego typu danych (jednak nawet wtedy kompilator nie pozwala na jednoczesne podawanie typu wszystkich elementów).

Tabela 3.1. Przykładowy kod ilustrujący deklarowanie krotek i przypisanie do nich wartości

Przykład	Opis	Przykładowy kod
1.	Przypisywanie krotki do osobno deklarowanych zmiennych	(string country, string capital, double gdpPerCapita) = ("Sudan Południowy", "Juba", 275.18); System.Console.WriteLine(\${@"Najbiedniejszym krajem w 2019 r. był { country}, {capital} : {gdpPerCapita}"});
2.	Przypisywanie krotki do osobno zadeklarowanych wcześniej zmiennych	string country; string capital; double gdpPerCapita; (country, capital, gdpPerCapita) = ("Sudan Południowy", "Juba", 275.18); System.Console.WriteLine(\${@"Najbiedniejszym krajem w 2019 r. był { country}, {capital} : {gdpPerCapita}"});
3.	Przypisywanie krotki do osobno deklarowanych zmiennych z niejawnie określonym typem	[var] country, [var] capital, [var] gdpPerCapita) = ("Sudan Południowy", "Juba", 275.18); System.Console.WriteLine(\${@"Najbiedniejszym krajem w 2019 r. był { country}, {capital} : {gdpPerCapita}"});
4.	Przypisywanie krotki do osobno deklarowanych zmiennych z niejawnie określonym typem wszystkich zmiennych	var (country, capital, gdpPerCapita) = ("Sudan Południowy", "Juba", 275.18); System.Console.WriteLine(\${@"Najbiedniejszym krajem w 2019 r. był { country}, {capital} : {gdpPerCapita}"});
5.	Deklarowanie nazwanej krotki, przypisywanie jej wartości i dostęp do elementów krotki za pomocą nazwy	(string Name, string Capital, double GdpPerCapita) countryInfo = ("Sudan Południowy", "Juba", 275.18); System.Console.WriteLine(\${@"Najbiedniejszym krajem w 2019 r. był { countryInfo.Name}, {countryInfo.Capital} : { countryInfo.GdpPerCapita}"});

Tabela 3.1. Przykładowy kod ilustrujący deklarowanie krotek i przypisanie do nich wartości — ciąg dalszy

Przykład	Opis	Przykładowy kod
6.	Przypisywanie krotki z nazwanymi elementami do elementu o niejawnie określonym typie i dostęp do elementów krotki na podstawie nazw	<pre>var countryInfo = {Name: "Sudan Południowy", Capital: "Juba", GdpPerCapita: 275.18}; System.Console.WriteLine(\$"@'Najbiedniejszym krajem świata w 2019 r. był { countryInfo.Name}, {countryInfo.Capital}: { countryInfo.GdpPerCapita}'");</pre>
7.	Przypisywanie krotki z elementami bez nazw do jednej zmiennej o niejawnie określonym typie i dostęp do elementów krotki na podstawie numerowanych właściwości	<pre>var countryInfo = {"Sudan Południowy", "Juba", 275.18}; System.Console.WriteLine(\$"@' Najbiedniejszym krajem świata w 2019 r. był { countryInfo.Item1}, {countryInfo.Item2}: { countryInfo.Item3}'");</pre>
8.	Przypisywanie krotki z nazwanymi elementami do jednej zmiennej o niejawnie określonym typie i dostęp do elementów krotki na podstawie numerowanych właściwości	<pre>var countryInfo = {Name: "Sudan Południowy", Capital: "Juba", GdpPerCapita: 275.18}; System.Console.WriteLine(\$"@' Najbiedniejszym krajem świata w 2019 r. był { countryInfo.Item1}, {countryInfo.Item2}: { countryInfo.Item3}'");</pre>
9.	Pomijanie fragmentów krotek za pomocą znaku podkreślenia	<pre>(string name, double gdpPerCapita) countryInfo = ("Sudan Południowy", "Juba", 275.18);</pre>
10.	Nazwy elementów krotek mogą zostać wywnioskowane na podstawie nazw zmiennych i właściwości (od wersji C# 7.1)	<pre>string country = "Sudan Południowy"; string capital = "Juba"; double gdpPerCapita = 275.18; var countryInfo = (country, capital, gdpPerCapita); System.Console.WriteLine(\$"@'Najbiedniejszym krajem świata w 2019 r. był { countryInfo.country}, {countryInfo.capital}: { countryInfo.gdpPerCapita}'");</pre>

W przykładzie 5. po lewej stronie deklarowana jest krotka, a po prawej przypisywana jest taka struktura. Zauważ, że ta krotka obejmuje nazwane elementy, które można podawać, aby pobierać wartości elementów z krotki. To te nazwy pozwalają stosować składnię `countryInfo.Name`, `countryInfo.Capital` i `countryInfo.GdpPerCapita` w instrukcji `System.←Console.WriteLine`. Efektem zadeklarowania krotki po lewej stronie jest połączenie zmiennych w jedną zmienną (`countryInfo`), która daje dostęp do elementów krotki. Ta technika jest przydatna, ponieważ — co zostało opisane w rozdziale 4. — można wtedy przekazywać jedną zmienną do innych metod, a te będą miały dostęp do poszczególnych elementów krotki.

Wspomniałem już, że dla zmiennych definiowanych za pomocą składni dla krotek używana jest notacja Wielbłurga. Jednak konwencje dla nazw elementów krotek nie są dobrze zdefiniowane. Pojawiły się sugestie, aby stosować tu konwencje typowe dla nazw parametrów, gdy krotka działa jak parametr — na przykład jeśli służy do zwracania zestawu wartości, dla których przed wprowadzeniem krotek używane byłyby parametry typu `out`. Inna możliwość to stosowanie NotacjiPascalowej, tak jak dla składowych typów (właściwości, funkcji i pól publicznych, co zostało omówione w rozdziałach 5. i 6.). Ja jestem zdecydowanym zwolennikiem drugiego z tych podejść i stosowania NotacjiPascalowej, zapewniającej spójność z nazwami wszystkich identyfikatorów składowych w języku C# i platformie .NET. Jednak ponieważ konwencje z tego obszaru nie są powszechnie przyjęte, we wskazówkach używam słowa ROZWAŻ zamiast STOSUJ: „ROZWAŻ stosowanie NotacjiPascalowej dla wszystkich nazw elementów krotek”.

Wskazówki

STOSUJ notację Wielbłurga w deklaracjach zmiennych w składni dla krotek.

ROZWAŻ stosowanie NotacjiPascalowej dla wszystkich nazw elementów krotek.

W przykładzie 6. został przedstawiony ten sam mechanizm co w przykładzie 5., przy czym po prawej stronie, w wartości krotki, używane są nazwane elementy krotek, a po lewej — typ jest określany niejawnie. Nazwy elementów są zachowywane w zmiennej z niejawnie określonym typem, dlatego są dostępne w instrukcji `WriteLine`. Oczywiście grozi to tym, że elementy po lewej stronie mogłyby mieć nazwy inne niż elementy po stronie prawej. Choć kompilator języka C# zezwala na to, wyświetla ostrzeżenie z informacją, że nazwy elementów podane po prawej stronie zostaną zignorowane i że użyte zostaną nazwy zapisane po lewej.

Nawet jeśli nie podasz nazw elementów, poszczególne elementy będą dostępne w zmiennej, do której przypisano krotkę. Nazwy tych elementów to `Item1`, `Item2`, ..., co ilustruje przykład 7. Co więcej, w krotkach nazwy w formie `ItemX` zawsze są dostępne, nawet jeśli podano niestandardowe nazwy (zobacz przykład 8.). Jednak gdy używasz środowiska IDE, np. jednej z nowych wersji środowiska Visual Studio (z obsługą języka C# 7.0), właściwość `ItemX` nie jest wyświetlana na liście rozwijanej mechanizmu IntelliSense. Jest to dobre podejście, ponieważ preferowane są nazwy podane przez programistę. W przykładzie 9. pokazano, że fragmenty przypisania krotki można zablokować, używając znaku podkreślenia — jest to operacja **odrzucania**.

Możliwość wywnioskowania nazw elementów krotki, jak ilustruje to przykład 10., wprowadzono dopiero w wersji C# 7.1. Ten przykład pokazuje, że nazwę elementu krotki można wywnioskować na podstawie nazwy zmiennej, a nawet nazwy właściwości.

Krotki są prostą techniką umieszczania danych w jednym obiekcie w podobny sposób, jak wkłada się różne produkty do torby w sklepie. W odróżnieniu od opisanych dalej tablic krotki zawierają elementy o dowolnych typach danych², przy czym elementy te są określane w kodzie i nie można ich modyfikować w czasie wykonywania programu. Inną różnicą w porównaniu z tablicami jest to, że liczba elementów krotki jest trwale ustalana na etapie komplikacji. Ponadto do krotki nie można dodawać niestandardowych operacji (w tym metod rozszerzających). Jeśli chcesz powiązać dane z operacjami, wtedy lepiej jest wykorzystać programowanie obiektowe i zdefiniować klasę. Ten temat jest szczegółowo opisany w rozdziale 6.

7.0

ZAGADNIENIE DLA ZAAWANSOWANYCH

Typ System.ValueTuple<...>

Kompilator języka C# generuje kod, w którym do implementacji składni dla krotek we wszystkich instancjach krotek występujących po prawej stronie w przykładach z tabeli 3.1 używany jest zestaw generycznych typów bezpośrednich (struktur) takich jak `System.ValueTuple<T1, T2, T3>`. Ten sam zestaw generycznych typów bezpośrednich `System.ValueTuple<...>` jest używany dla typów danych występujących po lewej stronie w przykładach od 5. do ostatniego. Nie jest zaskoczeniem, że jedyne metody dostępne w typie reprezentującym krotki służą do porównań.

Niestandardowe nazwy i typy elementów nie są uwzględnione w definicji typu `System.ValueTuple<...>`. Jak więc możliwe jest, że każda taka nazwa działa jak składowa tego typu i jest dostępna jako składowa? Zaskakujące (zwłaszcza dla osób zaznajomionych z implementacją typów anonimowych) jest to, że kompilator nie generuje kodu CIL odpowiadającego niestandardowym nazwom składowych. Jednak choć w kodzie CIL nie występuje składowa o niestandardowej nazwie, to w C# wygląda na to, że taka składowa jest dostępna.

W tabeli 3.1 we wszystkich przykładach z krotkami nazwanymi możliwe jest, że nazwy są znane przez kompilator w zasięgu używania krotki, ponieważ ten zasięg odpowiada składowej, w której zadeklarowano daną krotkę. Rzeczywiście — kompilator (i środowisko IDE) wykorzystuje ten zasięg, aby umożliwić dostęp do każdego elementu za pomocą nazwy. Innymi słowy, kompilator szuka nazw elementów w deklaracji krotki i korzysta z nich, aby umożliwić pracę kodu używającego tych nazw w zasięgu krotki. To także z tego powodu nazwy `ItemX` nie są wyświetlane przez mechanizm IntelliSense środowiska IDE jako dostępne składowe krotki (środowisko IDE ignoruje te nazwy i zastępuje je bezpośrednio podanymi nazwami elementów).

Ustalanie nazw elementów na podstawie nazw występujących w określonej składowej jest zrozumiałe dla kompilatora, co się jednak dzieje, gdy krotka jest używana poza tą składową, na przykład jako parametr lub wartość zwracana przez metodę z innego podzespołu

² Technicznym ograniczeniem jest zakaz stosowania wskaźników — zagadnienie to jest opisane w rozdziale 23.

(którego kod źródłowy jest niedostępny)? W przypadku wszystkich krotek będących częścią API (czy to publicznego, czy to prywatnego) kompilator dodaje nazwy elementów (w postaci atrybutów) do metadanych składowej. Na listingu 3.6 pokazano odpowiednik kodu, jaki kompilator wygeneruje na podstawie następującej instrukcji z języka C#:

```
public (string First, string Second) ParseNames(string fullName)
```

Listing 3.6. Kod w języku C# będący odpowiednikiem kodu CIL wygenerowanego przez kompilator na podstawie krotki `ValueTuple`

```
[return: System.Runtime.CompilerServices.TupleElementNames(new string[] {"First", "Second"})]
public System.ValueTuple<string, string> ParseNames(string fullName)
{
    // ...
}
```

Powiązanym zagadnieniem jest to, że C# 7.0 nie pozwala stosować niestandardowych nazw elementów w bezpośrednio podawanym typie danych `System.ValueTuple<...>`. Dlatego jeśli zastąpisz słowo `var` w przykładzie 8. z tabeli 3.1, zobaczysz ostrzeżenia z informacją, że wszystkie nazwy elementów zostaną zignorowane.

Oto kilka dodatkowych informacji, o których warto pamiętać w kontekście typu `System.ValueTuple<...>`:

- Istnieje osiem generycznych typów `System.ValueTuple<...>`. Pierwszych siedem umożliwia tworzenie krotek zawierających do siedmiu elementów. W ósmym typie, `System.ValueTuple<T1, T2, T3, T4, T5, T6, T7, TRest>`, ostatni parametr pozwala podać dodatkową wartość typu `ValueTuple`, dzięki czemu możliwe jest dodanie n elementów. Jeśli utworzysz krotkę o ósmiu parametrach, kompilator automatycznie zastosuje typ `System.ValueTuple<T1, T2, T3, T4, T5, T6, T7, System.ValueTuple<TSub1>>`. W ramach uzupełnienia warto dodać, że typ `System.Value<T1>` istnieje, ale rzadko jest używany, ponieważ składnia dla krotek w C# wymaga podania przynajmniej dwóch elementów.
- Dostępny jest też niegeneryczny typ `System.ValueTuple`, pełniący funkcję fabryki krotek. Udostępnia on metody `Create()` odpowiadające każdej liczbie elementów w obiektach typu `ValueTuple`. Jednak ponieważ stosowanie literałów krotek w formie `var t1 = ("Inigo Montoya", 42)` jest tak łatwe, takie literaly są używane częściej niż metoda `Create()` — przynajmniej przez użytkowników wersji C# 7.0 i nowszych.
- We wszystkich praktycznych zastosowaniach programiści języka C# mogą ignorować typy `System.ValueTuple` i `System.ValueTuple<T>`.

Istnieje też inny typ krotek, wprowadzony w platformie Microsoft .NET Framework 4.5: `System.Tuple<...>`. Wówczas zakładano, że będzie to podstawowa implementacja krotek. Jednak po dodaniu składni dla krotek w C# stwierdzono, że typ bezpośredni zapewnia wyższą wydajność, dlatego wprowadzono typ `System.ValueTuple<...>`, który w praktyce zastąpił typ `System.Tuple<...>` we wszystkich zastosowaniach z wyjątkiem miejsc, gdzie konieczne jest zachowanie zgodności z istniejącymi API, które wymagają typu `System.Tuple<...>`.

Początek
6.0

Koniec
6.0

Koniec
7.0

Tablice

Jednym z aspektów deklarowania zmiennych, którego nie omówiłem w rozdziale 1., jest deklarowanie tablic. W tablicach można zapisać wiele elementów tego samego typu, używając jednej zmiennej. Dostęp do tych elementów jest możliwy za pomocą indeksu. W C# indeksy rozpoczynają się od wartości zero. Dlatego indeksowanie tablic w C#aczyna się od zera.

ZAGADNIENIE DLA POCZĄTKUJĄCYCH

Tablice

Tablice umożliwiają zadeklarowanie za pomocą jednej zmiennej kolekcji elementów tego samego typu. Każdy element tablicy jest jednoznacznie określany przy użyciu wartości całkowitoliczbowej nazywanej **indeksem**. Dostęp do pierwszego elementu w tablicy w języku C# można uzyskać za pomocą indeksu 0. Programiści powinni pamiętać, by zawsze podawać wartości indeksu mniejsze niż liczba elementów tablicy. Ponieważ tablice w C# są indeksowane od zera, indeks ostatniego elementu jest o jeden mniejszy niż łączna liczba elementów. W C# 8.0 dostępny jest operator indeksowania od końca tablicy. Na przykład indeks ^1 wskazuje ostatni element w tablicy.

Dla początkujących pomocne jest traktowanie indeksu jak przesunięcia. Pierwszy element jest przesunięty o zero pozycji od początku tablicy, drugi element jest przesunięty o jedną pozycję od początku tablicy itd.

Tablice są podstawową strukturą w prawie każdym języku programowania, dlatego niemal wszyscy programiści powinni się nauczyć nimi posługiwać. Choć w języku C# tablice są często używane i z tego względu początkujący powinni się z nimi zapoznać, obecnie w większości programów do przechowywania kolekcji danych używane są generyczne typy kolekcji zamiast tablic. Dlatego wystarczy побieżnie przejrzeć podrozdział „Deklarowanie tablic”, by zapoznać się z tworzeniem instancji tablic i przypisywaniem do nich wartości. W tabeli 3.2 opisano punkty, na które warto zwrócić uwagę. Szczegółowe omówienie generycznych kolekcji znajdziesz w rozdziale 15.

Ponadto w ostatnim podrozdziale, „Typowe błędy związane z tablicami”, znajduje się przegląd wybranych osobliwości dotyczących tablic.

Deklarowanie tablicy

W języku C# do deklarowania tablic służą nawiasy kwadratowe. Najpierw należy podać typ elementów tablicy, a następnie otwierający i zamknięty nawias kwadratowy. Potem wpisywana jest nazwa zmiennej. Na listingu 3.7 zadeklarowana jest zmienna languages, która jest tablicą łańcuchów znaków.

Tabela 3.2. Najważniejsze informacje o tablicach

Opis	Przykład
Deklaracja Zauważ, że nawiasy kwadratowe pojawiają się obok typu danych. Tablice wielowymiarowe deklarowane są za pomocą przecinków. Liczba wymiarów to liczba przecinków + 1.	<pre>string[] languages; // Jednowymiarowa int[,] cells; // Dwuwymiarowa</pre>
Przypisywanie W deklaracji słowo kluczowe new i nazwa typu danych są opcjonalne. Po deklaracji słowo kluczowe new jest wymagane, gdy tworzona jest instancja tablicy. Dla tablicy można wywołać przypisanie bez podawania literałów. Wtedy każdy element tablicy jest inicjowany domyślną wartością. Jeśli wartości nie są podane, należy określić wielkość tablicy. Nie musi być to stała. Można podać zmienną o wartości ustalanej w czasie wykonywania programu. Od wersji C# 3.0 określanie typu danych jest opcjonalne.	<pre>string[] languages = { "C#", "COBOL", "Java", "C++", "TypeScript", "Pascal", "Python", "Lisp", "JavaScript"}; languages = new string[9]; languages = new string[]{ "C#", "COBOL", "Java", "C++", "TypeScript", "Pascal", "Python", "Lisp", "JavaScript"};</pre> <p>// Inicjowanie tablic wielowymiarowych // i przypisywanie do nich wartości.</p> <pre>int[,] cells = new int[3,3]; cells = { {1, 0, 2}, {1, 2, 0}, {1, 2, 1} };</pre>
Dostęp do tablicy od początku Tablice są indeksowane od zera, dlatego pierwszy element ma indeks 0. Nawiasy kwadratowe służą do zapisywania i pobierania danych w tablicach.	<pre>string[] languages = new string[]{"C#", "COBOL", "Java", "C++", "TypeScript", "Visual Basic", "Python", "Lisp", "JavaScript"}; // Pobieranie piątego elementu tablicy // languages (TypeScript)..</pre> <pre>string language = languages[4]; // Wyświetli „TypeScript”. System.Console.WriteLine(language); // Pobieranie trzeciego elementu // od końca (Python)..</pre> <pre>language = languages[^3]; // Wyświetli „Python”. System.Console.WriteLine(language);</pre>
Dostęp do tablicy od końca Od wersji C# 8.0 można podawać indeksy tablicy liczone od końca. Na przykład indeks ^1 oznacza ostatni element tablicy, a ^3 to trzeci element od końca.	<pre>System.Console.WriteLine(\$"0^3..^0: { // Python, Lisp, JavaScript string.Join(", ", languages[^3..^0]) }"); System.Console.WriteLine(\$"0^3..: { // Python, Lisp, JavaScript string.Join(", ", languages[^3..]) }"); System.Console.WriteLine(\$"0" ..^3: { // C++, TypeScript, Visual Basic string.Join(", ", languages[3..^3]) }"; System.Console.WriteLine(\$"0" ..^6: { // C#, COBOL, Java string.Join(", ", languages[..^6]) }");</pre>
Przedziały W wersji C# 8.0 można wskazywać i pobierać tablice elementów za pomocą operatora przedziałów, który określa początkowy i końcowy element przedziału (końcowy element nie jest pobierany).	<pre>Koniec 3.0 Koniec 8.0</pre>

Listing 3.7. Deklarowanie tablicy

```
string[] languages;
```

Pierwsza część deklaracji określa typ danych elementów tablicy. Nawiązy kwadratowe w deklaracji wyznaczają jej **stopień**, czyli liczbę wymiarów. Tu używana jest tablica pierwszego stopnia. Dwa opisane elementy określają typ danych zmiennej `languages`.

Porównanie języków — deklaracje tablic w językach C++ i Java

W języku C# nawiasy kwadratowe reprezentujące tablice pojawiają się bezpośrednio po nazwie typu danych, a nie po deklaracji zmiennej. Dzięki temu wszystkie informacje o typie znajdują się w jednym miejscu, a nie przed identyfikatorem i po nim, jak ma to miejsce w językach C++ i Java.

Na listingu 3.7 zdefiniowano tablicę pierwszego stopnia. Przecinki w nawiasie kwadratowym pozwalają zdefiniować dodatkowe wymiary. Na przykład na listingu 3.8 zdefiniowana jest tablica dwuwymiarowa z polami do gry w szachy lub kółko i krzyżyk.

Listing 3.8. Deklarowanie tablicy dwuwymiarowej

```
//   |   |
// ---+---+---
//   |   |
// ---+---+---
//   |   |
int[,] cells;
```

Tablica na listingu 3.8 jest drugiego stopnia. Pierwszy wymiar może reprezentować wiersze, a drugi — kolumny. Dodatkowe wymiary są dodawane za pomocą kolejnych przecinków. Stopień tablicy jest o jeden większy od liczby przecinków. Zauważ, że liczba elementów w określonym wymiarze nie jest w deklaracji podawana. Określa się ją w momencie tworzenia instancji tablicy i przydzielania pamięci dla wszystkich elementów.

Tworzenie instancji tablic i przypisywanie do nich wartości

Po zadeklarowaniu tablicy można natychmiast podać jej wartości przy użyciu rozdzielonej przecinkami listy elementów w nawiasie klamrowym. Na listingu 3.9 deklarowana jest tablica łańcuchów znaków, do której w nawiasie klamrowym przypisywane są nazwy dziewięciu języków programowania.

Listing 3.9. Deklaracja tablicy i przypisanie do niej wartości

```
string[] languages = { "C#", "COBOL", "Java",
"C++", "TypeScript", "Visual Basic",
"Python", "Lisp", "JavaScript"};
```

Pierwszy element na przedstawionej rozdzielonej przecinkami liście staje się pierwszym elementem tablicy. Drugi element z listy zostaje drugim elementem tablicy itd. Nawiasy klamrowe to notacja służąca do definiowania literałów tablicowych.

Składnia przypisania pokazana na listingu 3.9 jest dostępna tylko wtedy, gdy w jednej instrukcji deklarujesz tablicę i przypisujesz do niej wartość. Aby przypisać wartość po zadeklarowaniu tablicy, należy zastosować słowo kluczowe new, tak jak na listingu 3.10.

Listing 3.10. Przypisywanie wartości do tablicy po deklaracji

```
string[] languages;
languages = new string[] {"C", "COBOL", "Java",
    "C++", "TypeScript", "Visual Basic",
    "Python", "Lisp", "JavaScript"};
```

Od wersji C# 3.0 określanie typu danych tablicy (tu jest to typ `string`) po słowie kluczowym `new` jest opcjonalne, jeśli kompilator może wywnioskować typ elementów tablicy na podstawie typów elementów użytych w trakcie jej inicjowania. Niezależnie od tego konieczne jest dodanie nawiasów kwadratowych.

W języku C# można też podać słowo kluczowe `new` w deklaracji. Dlatego przypisanie i deklaracja z listingu 3.11 są poprawne.

Listing 3.11. Przypisywanie wartości do tablicy w deklaracji z wykorzystaniem słowa new

```
string[] languages = new string[] {
    "C", "COBOL", "Java",
    "C++", "TypeScript", "Visual Basic",
    "Python", "Lisp", "JavaScript"};
```

Słowo kluczowe `new` jest dla środowiska uruchomieniowego informacją, że należy przydzielić pamięć dla określonego typu danych. Środowisko uruchomieniowe tworzy wtedy instancję typu danych (tu jest to tablica).

Gdy podajesz słowo kluczowe `new` w przypisaniu wartości do tablicy, możesz też określić w nawiasie kwadratowym wielkość tablicy. Tę składnię przedstawiono na listingu 3.12.

Listing 3.12. Deklaracja i przypisanie wartości z wykorzystaniem słowa kluczowego new

```
string[] languages = new string[9] {
    "C", "COBOL", "Java",
    "C++", "TypeScript", "Visual Basic",
    "Python", "Lisp", "JavaScript"};
```

Wielkość tablicy podana w instrukcji inicjującej i liczba elementów wymienionych w nawiasie klamrowym muszą się ze sobą zgadzać. Ponadto można przypisać do zmiennej tablicę bez określania jej początkowych wartości. Tę technikę zastosowano na listingu 3.13.

Listing 3.13. Przypisywanie tablicy bez określonych wartości

```
string[] languages = new string[9];
```

Początek
3.0

Koniec
3.0

Przypisywanie tablicy bez określania wartości też powoduje zainicjowanie wszystkich elementów. Środowisko uruchomieniowe używa wtedy wartości domyślnych. Oto ich opis:

- Typy referencyjne (niezależnie od tego, czy dopuszczają wartości null, czy nie, na przykład `string` i `string?`) są inicjowane wartością null.
- Typy bezpośrednie dopuszczające wartość null są inicjowane wartościami null.
- Typy liczbowe niedopuszczające wartości null są inicjowane wartością zero.
- Typ `bool` jest inicjowany wartością `false`.
- Typ `char` jest inicjowany wartością `\0`.

Typy bezpośrednie, które nie są typami prostymi, są rekurencyjnie inicjowane w wyniku zainicjowania wszystkich ich pól wartościami domyślnymi. Oznacza to, że nie trzeba przypisywać wartości do wszystkich elementów tablicy przed ich użyciem.

Ponieważ wielkość tablicy nie jest określana w deklaracji zmiennej, można ją ustawić w czasie wykonywania programu. Kod z listingu 3.14 tworzy tablicę na podstawie rozmiaru pobranego w wywołaniu metody `Console.ReadLine()`.

Listing 3.14. Definiowanie wielkości tablicy w czasie wykonywania programu

```
string[] groceryList;
System.Console.Write("Ile elementów znajduje się na liście? ");
int size = int.Parse(System.Console.ReadLine());
groceryList = new string[size];
// ...
```

Inicjowanie tablic wielowymiarowych w języku C# odbywa się podobnie jak dla tablic jednowymiarowych. Do rozdzielania poszczególnych wymiarów służy przecinek. Kod na listingu 3.15 inicjuje planszę do gry w kółko i krzyżyk, na której nie wykonano jeszcze żadnych posunięć.

Listing 3.15. Deklarowanie tablicy dwuwymiarowej

```
int[,] cells = new int[3,3];
```

Aby zainicjować planszę do gry w kółko i krzyżyk z określoną pozycją, możesz użyć kodu przedstawionego na listingu 3.16.

Listing 3.16. Inicjowanie dwuwymiarowej tablicy liczb całkowitych

```
int[,] cells = {
    {1, 0, 2},
    {1, 2, 0},
    {1, 2, 1}
};
```

Inicjowanie odbywa się zgodnie ze wzorcem, w którym występuje trzyelementowa tablica typu `int[]`. Każdy element tej tablicy ma ten sam rozmiar (tu jest on równy 3). Zauważ, że wielkość wszystkich elementów `int[]` musi być identyczna. Dlatego deklaracja z listingu 3.17 nie jest prawidłowa.

Listing 3.17. Wielowymiarowa tablica z elementami o niezgodnych rozmiarach powoduje błąd

```
// BŁĄD: wszystkie elementy muszą mieć spójną wielkość.
int[,] cells = {
    {1, 0, 2, 0},
    {1, 2, 0},
    {1, 2}
    {1}
};
```

Tworzenie reprezentacji planszy do gry w kółko i krzyżyk nie wymaga liczby całkowitej na każdej pozycji. Jedną z możliwości jest utworzenie odrębnych wirtualnych plansz dla obu graczy. Obie plansze mogą zawierać wartości typu `bool` określające pola zaznaczone przez poszczególnych graczy. Listing 3.18 tworzy planszę reprezentowaną przez tablicę trójwymiarową.

Listing 3.18. Inicjowanie tablicy trójwymiarowej

```
bool[,,] cells;
cells = new bool[2,3,3]
{
    // Posunięcia gracza nr 1      // X |   |
    { {true, false, false},     // ---+---+---
      {true, false, false},     // X |   |
      {true, false, true} },   // ---+---+---
                                // X |   | X

    // Posunięcia gracza nr 2      //   |   | O
    { {false, false, true},    // ---+---+---
      {false, true, false},    //   | O |
      {false, true, true} }    // ---+---+---
                                //   | O |
```

};

W tym przykładzie plansza jest inicjowana, a liczba elementów w każdym wymiarze jest podana jawnie. Kod określa liczbę elementów w wyrażeniu `new`, a ponadto ustawia wartości przechowywane w tablicy. Tworzona jest tablica typu `bool[, ,]`, podzielona na dwie tablice typu `bool[,]` o wymiarach 3×3 . Każda dwumiarowa tablica składa się z trzech trzyelementowych tablic typu `bool`.

Wcześniej wspomniano, że w tablicy wielowymiarowej wszystkie elementy z danego wymiaru muszą mieć tę samą wielkość. Można jednak zdefiniować także **tablicę postrzępioną** (ang. *jagged array*), czyli tablicę tablic. Składnia tworzenia takich tablic różni się nieco od składni dla tablic wielowymiarowych. Ponadto elementy w tablicach tablic nie muszą mieć takiej samej długości. Dlatego tablicę tablic można zainicjować w sposób przedstawiony na listingu 3.19.

Listing 3.19. Inicjowanie tablicy tablic

```
int[][] cells = {
    new int[]{1, 0, 2, 0},
    new int[]{1, 2, 0},
    new int[]{1, 2},
    new int[]{1}
};
```

W tablicy tablic nie stosuje się przecinka do definiowania nowych wymiarów. Zamiast tego definiowana jest tablica tablic. Na listingu 3.19 nawias [] jest umieszczony po typie danych int[], co powoduje zadeklarowanie tablicy typu int[].

Zauważ, że tablica tablic wymaga instancji tablicy (lub wartości null) dla każdej wewnętrznej tablicy. W tym przykładzie do tworzenia instancji wewnętrznych elementów tablicy tablic używane jest słowo kluczowe new. Pominięcie tego kroku spowodowałoby błąd komplikacji.

Początek
8.0

Korzystanie z tablicy

Dostęp do elementu tablicy można uzyskać za pomocą **akcesora tablicy**, czyli nawiasu kwadratowego z indeksem. Aby pobrać pierwszy element tablicy, należy podać indeks 0. Na listingu 3.20 wartość piątego elementu (o indeksie 4, ponieważ pierwszy element ma indeks 0) ze zmiennej languages jest zapisywana w zmiennej language.

Listing 3.20. Deklarowanie tablicy i dostęp do niej

```
string[] languages = new string[9]{
    "C#", "COBOL", "Java",
    "C++", "TypeScript", "Visual Basic",
    "Python", "Lisp", "JavaScript"};
// Pobieranie piątego elementu (TypeScript) z tablicy languages
string language = languages[4];
// Wyświetla "TypeScript"
Console.WriteLine(language);
// Pobieranie trzeciego elementu od końca (Python)
language = languages[^3];
// Wyświetla "Python"
Console.WriteLine(language);
```

Od wersji C# 8.0 można też pobierać elementy, podając indeksy od końca tablicy. Służy do tego **operator indeksowania od końca tablicy** — ^. Indeks ^1 oznacza ostatni element tablicy, a aby pobrać pierwszy element przykładowej tablicy, należy użyć indeksu liczonego od końca tablicy ^9 (9 to liczba elementów w danej tablicy). Na listingu 3.20 indeks ^3 pozwala przypisać do zmiennej language wartość trzeciego elementu od końca ("Python") z tablicy languages.

Ponieważ ^1 to indeks ostatniego elementu, ^0 powoduje wyjście poza koniec listy. Oczywiście taka pozycja nie istnieje w tablicy, dlatego nie można wskazać elementu za pomocą indeksu ^0. Podobnie użycie długości tablicy, 9, powoduje wskazanie pozycji poza końcem tablicy. Jako indeksów tablicy nie można też używać wartości ujemnych.

Widoczna jest niespójność w indeksowaniu tablicy od początku za pomocą dodatnich liczb całkowitych i indeksowaniu od końca za pomocą operatora ^ i liczb całkowitych (lub wyrażeń zwracających wartość całkowitoliczbową). Indeksowanie od początku wymaga podania indeksu 0 w celu wskazania pierwszego elementu, a indeksowanie od końca wymaga użycia indeksu ^1 w celu wskazania ostatniego elementu. Zespół projektowy odpowiedzialny za język C# zdecydował się indeksować tablice od zera, aby zachować spójność z językami, na których C# bazuje (C, C++ i Java). Jeśli chodzi o indeksowanie od końca, w C#

wzorowano się na Pythonie (ponieważ języki z rodziny C nie obsługiwały operatora indeksowania od końca) i użyto indeksowania od jedynki. Jednak, inaczej niż w Pythonie, twórcy C# zdecydowali się zastosować operator \wedge (zamiast ujemnych liczb całkowitych), aby uniknąć niezgodności wstecz przy używaniu operatora indeksowania do kolekcji (nie do tablic) dopuszczających wartości ujemne. Operator \wedge ma też dodatkową zaletę związaną z obsługą przedziałów, co opisane jest dalej w rozdziale. Aby zrozumieć działanie indeksów liczących od początku i od końca, zauważ, że gdy dodatnia liczba całkowita oznacza indeks liczony od końca, ostatni element to długość — 1, przedostatni element to długość — 2 itd. Liczba całkowita odejmowana od długości oznacza element „n-ty od końca” — $\wedge 1$, $\wedge 2$ itd. W tym pojęciu indeks liczony od początku plus indeks liczony od końca zawsze dają w sumie długość tablicy.

Zauważ, że jako indeksu liczonego od końca nie trzeba podawać literała całkowitoliczbowego. Można też posłużyć się wyrażeniem takim jak:

```
languages[ $\wedge$ languages.Length]
```

To wyrażenie zwróci pierwszy element.

Notacja z nawiasem kwadratowym służy też do zapisywania danych w tablicy. Na listingu 3.21 kod zmienia kolejność wartości "C++" i "Java".

Listing 3.21. Zmiana pozycji danych w tablicy

```
string[] languages = new string[] {
    "C#", "COBOL", "Java",
    "C++", "TypeScript", "Visual Basic",
    "Python", "Lisp", "JavaScript"};
// Zapisywanie wartości "C++" w zmiennej language.
string language = languages[3];
// Przypisywanie wartości "Java" na pozycji zajmowanej przez wartość "C++".
languages[3] = languages[2];
// Przypisywanie wartości zmiennej language do pozycji zajmowanej pierwotnie przez wartość "Java".
languages[2] = language;
```

W tablicach wielowymiarowych w celu wskazania elementu należy podać indeks z każdego wymiaru. Ilustruje to listing 3.22.

Listing 3.22. Inicjowanie dwuwymiarowej tablicy liczb całkowitych

```
int[,] cells = {
    {1, 0, 2},
    {0, 2, 0},
    {1, 2, 1}
};
// Wykonywanie w grze w kółko i krzyżyk ruchu dającego wygraną graczu nr 1.
cells[1,0] = 1;
```

Przypisywanie wartości do elementów tablicy tablicy odbywa się inaczej, w sposób zgodny z deklarowaniem takich tablic. Pierwszy indeks określa tablicę z tablicy tablic. Drugi indeks wskazuje element w wybranej tablicy (zobacz listing 3.23).

Listing 3.23. Deklarowanie tablicy tablic

```
int[][] cells = {
    new int[]{1, 0, 2},
    new int[]{0, 2, 0},
    new int[]{1, 2, 1}
};

cells[1][0] = 1;
// ...
```

8.0

Długość

Długość tablicy można określić w sposób przedstawiony na listingu 3.24.

Listing 3.24. Pobieranie długości tablicy

```
Console.WriteLine(
    $"Liczba języków w tablicy to { languages.Length }.");
```

Tablice mają stałą długość. Nie można jej zmienić bez ponownego utworzenia tablicy. Ponadto wyjście poza **granice** (długość) tablicy skutkuje zgłoszeniem błędu przez środowisko uruchomieniowe. Taka sytuacja może nastąpić przy próbie dostępu (w celu odczytu lub zapisu) do tablicy za pomocą indeksu, który nie prowadzi do elementu z danej tablicy. Taki błąd często występuje, gdy programista używa długości tablicy jako indeksu, co pokazano na listingu 3.25.

Listing 3.25. Próba dostępu poza granicami tablicy prowadzi do zgłoszenia wyjątku

```
string languages = new string[9];
...
// BŁĄD CZASU WYKONANIA: indeks spoza tablicy. Dla ostatniego elementu
// należy użyć indeksu 8.
languages[4] = languages[9];
```

Uwaga

Składowa Length zwraca liczbę elementów w tablicy, a nie najwyższy indeks. Dla zmiennej languages wartość składowej Length to 9, ale najwyższy indeks to 8, ponieważ taka jest odległość ostatniego elementu od początku.

Porównanie języków — błędy przepełnienia bufora w języku C++

Niezarządzany kod w języku C++ nie zawsze sprawdza, czy nastąpiło wyjście poza zakres tablicy. Powodowane przez to błędy nie tylko są trudne do wykrycia, ale mogą prowadzić do problemów z bezpieczeństwem związanych z **przepełnieniem bufora**. Natomiast środowisko CLR (ang. *Common Language Runtime*) chroni cały kod w języku C# (i w Managed C++) przed wyjściem poza zakres tablicy, co w praktyce eliminuje ryzyko przepełnienia bufora w zarządzanym kodzie.

W języku C# 8.0 ten sam problem pojawia się przy próbie dostępu do elementu `^0`. Ponieważ `^1` jest ostatnim elementem, `^0` wykracza poza koniec tablicy, na element, który nie istnieje.

Aby uniknąć wyjścia poza granice tablicy przy dostępie do ostatniego elementu, należy sprawdzać, czy długość tablicy jest większa od 0, i przy dostępie do ostatniego elementu tablicy używać zapisu `^1` (od wersji C# 8.0) lub `Length - 1` zamiast zapisanej na sztywno wartości. Aby wykorzystać wartość `Length` jako indeks, należy odjąć od niej 1, co pozwala uniknąć błędu wyjścia poza zakres tablicy (zobacz listing 3.26).

Listing 3.26. Używanie wartości `Length - 1` jako indeksu tablicy

```
string languages = new string[9];
...
languages[4] = languages[languages.Length - 1];
```

8.0

Oczywiście jeśli występuje ryzyko, że instancja tablicy nie istnieje, przed dostępem do niej należy sprawdzić wartość `null`.

Wskazówki

ROZWAŻ sprawdzanie wartości `null` przed dostępem do tablicy, zamiast przyjmować, że instancja tablicy istnieje.

ROZWAŻ sprawdzanie długości tablicy przed użyciem indeksu, zamiast zakładać, że ma ona określona wielkość.

ROZWAŻ używanie w wersjach C# 8.0 i nowszych operatora indeksowania od końca tablicy (`^`) zamiast wyrażenia `Length - 1`.

Składowa `Length` zwraca liczbę wszystkich elementów tablicy. Dlatego dla tablic wielowymiarowych, na przykład dla tablicy `bool cells[,]` o wymiarach $2 \times 3 \times 3$, zwracana jest łączna liczba elementów (tu jest to 18).

W tablicach tablic składowa `Length` zwraca liczbę elementów z nadrzędnej tablicy. Sprawdzana jest wtedy tylko zewnętrzna, nadrzędna tablica i zwracana jest liczba jej elementów. Nie ma przy tym znaczenia, co znajduje się w wewnętrznych tablicach.

Przedziały

Innym związanym z indeksowaniem mechanizmem dodanym do C# 8.0 jest obsługa pobierania wycinków tablicy do nowej tablicy. Do podawania przedziałów służy operator przedziału `..` (dwie kropki). Można je umieszczać także między indeksami (w tym indeksami licznymi od końca). Przykłady znajdziesz na listingu 3.27.

Listing 3.27. Przykłady stosowania operatora przedziału

```
string[] languages = new string[]{
    "C#", "COBOL", "Java",
    "C++", "TypeScript", "Visual Basic",
    "Python", "Lisp", "JavaScript"};
```

```
System.Console.WriteLine($"0..3: {  
    string.Join(", ", languages[0..3]) // C#, COBOL, Java  
}");  
System.Console.WriteLine($"^3..^0: {  
    string.Join(", ", languages[^3..^0]) // Python, Lisp, JavaScript  
}");  
System.Console.WriteLine($"3..^3: {  
    string.Join(", ", languages[3..^3]) // C++, TypeScript, Visual Basic  
}");  
System.Console.WriteLine($"..^6: {  
    string.Join(", ", languages[..^6]) // C#, COBOL, Java  
}");  
System.Console.WriteLine($"^6..: {  
    string.Join(", ", languages[6..]) // Python, Lisp, JavaScript  
}");  
System.Console.WriteLine($"..: {  
    // C#, COBOL, Java, C++, TypeScript, Visual Basic, Python, Lisp, JavaScript  
    string.Join(", ", languages[...])}
```

8.0

Ważną kwestią związaną z operatorem przedziału jest to, że elementy są wskazywane od początkowego (włączanego) do końcowego (pomijanego). Dlatego `0..3` na listingu 3.27 oznacza elementy od pierwszego do czwartego, ale z *pominięciem* tego ostatniego (3 reprezentuje czwarty element, ponieważ indeksowanie od początku zaczyna się od zera). Z kolei wyrażenie `^3..^0` powoduje pobranie trzech ostatnich elementów. Zapis `^0` nie powoduje tu błędu dostępu do elementu spoza końca tablicy, ponieważ element o końcowym indeksie przedziału jest pomijany.

Podawanie początkowego lub końcowego indeksu jest opcjonalne, co ilustrują przykłady od 4. do 6. na listingu 3.27. Dlatego pominiecie indeksów oznacza to samo co zapis `0..^0`.

Na zakończenie warto zauważyć, że indeksy i przedziały są w .NET i C# pełnoprawnymi typami (zobacz ZAGADNIENIE DLA ZAAWANSOWANYCH „System.Index i System.Range”). Ich zastosowania nie ograniczają się do dostępu do tablic.

ZAGADNIENIE DLA ZAAWANSOWANYCH

System.Index i System.Range

Użycie operatora indeksowania od końca oznacza bezpośrednie zastosowanie wartości typu `System.Index`. Możesz więc używać indeksów także poza nawiasami kwadratowymi — na przykład deklarując indeks i przypisując mu dosłowną wartość: `System.Index index = ^42`. Do obiektów typu `System.Index` można też przypisywać zwykłe liczby całkowite. Typ `System.Index` ma dwie właściwości: `Value` typu `int` i `IsFromEnd` typu `bool`. Ta druga informuje, czy indeks jest liczony od początku tablicy, czy od jej końca.

Typem danych używanym do podawania przedziałów jest `System.Range`. Dlatego możesz deklarować i przypisywać wartości przedziałów: `System.Range range = ..^0` lub nawet `System.Range range = ...` Ten typ ma dwie właściwości: `Start` i `End`.

Dzięki udostępnieniu tych typów w C# możliwe jest pisanie niestandardowych kolekcji obsługujących przedziały i indeksy liczone od końca. Tworzenie niestandardowych kolekcji jest opisane w rozdziale 17.

Inne metody dla tablic

Dla tablic dostępne są też dodatkowe metody przeznaczone do manipulowania elementami. Niektóre z tych metod to: Sort(), BinarySearch(), Reverse() i Clear() (zobacz listing 3.28).

Listing 3.28. Inne metody dla tablic

```
class ProgrammingLanguages
{
    static void Main()
    {
        string[] languages = new string[]{
            "C#", "COBOL", "Java",
            "C++", "TypeScript", "Visual Basic",
            "Python", "Lisp", "JavaScript"};

        System.Array.Sort(languages);

        string searchString = "COBOL";
        int index = System.Array.BinarySearch(
            languages, searchString);
        System.Console.WriteLine(
            "Język przyszłości, "
            + $"{searchString}, jest dostępny pod indeksem {index}.");

        System.Console.WriteLine();
        System.Console.WriteLine(
            $"{ "Pierwszy element",-20 }\t{ "Ostatni element",-20 }");
        System.Console.WriteLine(
            $"{ "-----",-20 }\t{ "-----",-20 }");
        System.Console.WriteLine(
            $"{ languages[0],-20 }\t{ languages[^1],-20 }");
        System.Array.Reverse(languages);
        System.Console.WriteLine(
            $"{ languages[0],-20 }\t{ languages[^1],-20 }");
        // Zauważ, że poniższa instrukcja usuwa elementów z tablicy.
        // Zamiast tego do wszystkich elementów przypisywana jest wartość domyślna.
        System.Array.Clear(languages, 0, languages.Length);
        System.Console.WriteLine(
            $"{ languages[0],-20 }\t{ languages[^1],-20 }");
        System.Console.WriteLine(
            $"Po wywołaniu Clear wielkość tablicy to: {languages.Length}");
    }
}
```

8.0

Efekt uruchomienia kodu z listingu 3.28 pokazano w danych wyjściowych 3.2.

DANE WYJŚCIOWE 3.2.

Język przyszłości, COBOL, jest dostępny pod indeksem 2.

Pierwszy element	Ostatni element
-----	-----
C#	TypeScript
TypeScript	C#

Po wywołaniu Clear wielkość tablicy to: 9

Dostęp do wymienionych metod można uzyskać za pomocą klasy `System.Array`. Metody te są zwykle intuicyjnie zrozumiałe. Warto jednak zwrócić uwagę na dwie kwestie:

- Ważne jest, by przed wywołaniem metody `BinarySearch()` posortować tablicę. Jeśli wartości nie są posortowane rosnąco, kod może zwrócić niewłaściwy indeks. Jeżeli szukany element nie istnieje, zwracana jest wartość ujemna. Użycie operatora dopełnienia, `-index`, powoduje zwrócenie indeksu pierwszej wartości większej od szukanej (jeśli ta większa wartość istnieje).
- Metoda `Clear()` nie usuwa elementów tablicy i nie powoduje ustawienia jej długości na zero. Wielkość tablicy jest stała i nie można jej zmienić. Dlatego metoda `Clear()` ustawia wszystkie elementy tablicy na ich wartość domyślną (`false`, `0` lub `null`). To wyjaśnia, dlaczego metoda `Console.WriteLine()` tworzy pusty wiersz, gdy wyświetla tablicę po wywołaniu metody `Clear()`.

Porównanie języków — zmiana liczby elementów tablicy w języku Visual Basic

Visual Basic udostępnia instrukcję `Redim`, która umożliwia zmianę liczby elementów tablicy. Choć w języku C# nie istnieje analogiczne słowo kluczowe, w platformie .NET 2.0 dostępna jest metoda, która odtwarza tablicę i kopiuje do nowej tablicy wszystkie elementy z jej pierwotnej wersji. Jest to metoda `System.Array.Resize`.

Koniec
8.0

Metody instancynie tablicy

Tablice, podobnie jak łańcuchy znaków, udostępniają metody instancynie, używane nie za pomocą typu danych, ale bezpośrednio z wykorzystaniem zmiennej. Przykładową składową instancynią jest `Length`, ponieważ dostęp do niej odbywa się za pomocą zmiennej, a nie przy użyciu klasy. Inne ważne składowe instancynie to `GetLength()`, `Rank` i `Clone()`.

Pobranie długości w wybranym wymiarze nie wymaga używania właściwości `Length`. Aby ustalić długość w danym wymiarze, należy zastosować metodę instancynią `GetLength()`. W wywołaniu tej metody trzeba wskazać wymiar, którego długość kod ma zwrócić (zobacz listing 3.29).

Listing 3.29. Pobieranie długości określonego wymiaru

```
bool[, ,] cells;
cells = new bool[2, 3, 3];
System.Console.WriteLine(cells.GetLength(0)); // Wyświetla 2
System.Console.WriteLine(cells.Rank); // Wyświetla 3
```

Efekt wykonania kodu z listingu 3.29 przedstawiają dane wyjściowe 3.3.

DANE WYJŚCIOWE 3.3.

Kod z listingu 3.29 wyświetla wartość 2, ponieważ tyle jest elementów w pierwszym wymiarze tablicy.

Można też określić stopień całej tablicy. Służy do tego składowa Rank. Na przykład instrukcja `cells.Rank` zwróci wartość 3 (zobacz listing 3.29).

Domyślnie przypisanie jednej zmiennej tablicowej do innej spowoduje skopiowanie tylko referencji do zmiennej; poszczególne elementy tablicy nie są wtedy kopiowane. Aby utworzyć zupełnie nową kopię tablicy, zastosuj metodę `Clone()`. Zwraca ona kopię tablicy. Modyfikacja elementów nowej tablicy nie wpływa na elementy pierwotnej tablicy.

Łańcuchy znaków jako tablice

Dostęp do zmiennych typu `string` odbywa się jak do tablic znaków. Na przykład aby pobrać czwarty znak łańcucha znaków `palindrome`, należy użyć wywołania `palindrome[3]`. Warto jednak zauważyc, że ponieważ łańcuchy znaków są niezmienne, nie można przypisywać do nich znaków. Dlatego język C# nie zezwala na wykonanie instrukcji `palindrome[3] = 'a'`, jeśli zmienna `palindrome` jest zadeklarowana jako łańcuch znaków. Na listingu 3.30 akcesor tablic jest używany w celu określenia, czy argument w wierszu poleceń to opcja. Opcjami są łańcuchy, których pierwszym znakiem jest kreska.

Listing 3.30. Wyszukiwanie opcji w wierszu poleceń

```
string[] args;
...
if (args[0][0] == '-')
{
    // Ten parametr jest opcją
}
```

W tym fragmencie używana jest opisana w rozdziale 4. instrukcja `if`. Przykładowy kod jest ciekawy, ponieważ akcesor tablic służy do pobrania pierwszego elementu tablicy łańcuchów znaków, `args`. Dalej używany jest drugi akcesor, który pobiera pierwszy znak łańcucha. Dlatego ten kod działa tak jak kod z listingu 3.31.

Listing 3.31. Wyszukiwanie opcji w wierszu poleceń (wersja uproszczona)

```
string[] args;
...
string arg = args[0];
if (arg[0] == '-')
{
    // Ten parametr to opcja
}
```

Możliwe jest więc pobieranie pojedynczych znaków za pomocą akcesora tablic. Ponadto można pobrać cały łańcuch jako tablicę, używając metody `ToCharArray()` łańcuchów znaków. To podejście pozwala odwrócić kolejność znaków łańcucha za pomocą metody `System.Array.Reverse()`. Ilustruje to listing 3.32, który określa, czy łańcuch znaków jest palindromem.

Listing 3.32. Odwracanie kolejności znaków w łańcuchu

```

class Palindrome
{
    static void Main()
    {
        string reverse, palindrome;
        char[] temp;

        System.Console.WriteLine("Wprowadź palindrom: ");
        palindrome = System.Console.ReadLine();

        // Usuwanie odstępów i przekształcanie liter na małe
        reverse = palindrome.Replace(" ", "");
        reverse = reverse.ToLower();

        // Przekształcanie w tablicę
        temp = reverse.ToCharArray();

        // Odwracanie kolejności elementów tablicy
        System.Array.Reverse(temp);

        // Przekształcanie tablicy z powrotem w łańcuch znaków i
        // sprawdzanie, czy odwrócony łańcuch jest identyczny z pierwotnym.
        if (reverse == new string(temp))
        {
            System.Console.WriteLine(
                $"\"{palindrome}\" jest palindromem.");
        }
        else
        {
            System.Console.WriteLine(
                $"\"{palindrome}\" NIE jest palindromem.");
        }
    }
}

```

Wyniki działania kodu z listingu 3.32 pokazano w danych wyjściowych 3.4.

DANE WYJŚCIOWE 3.4.

Wprowadź palindrom: A to kanapa pana Kota.
"A to kanapa pana Kota" jest palindromem.

W tym przykładzie używane jest słowo kluczowe new. Tu tworzy ono nowy łańcuch na podstawie tablicy z odwróconą kolejnością znaków.

Typowe błędy związane z tablicami

W tym podrozdziale przedstawiono trzy rodzaje tablic: jednowymiarowe, wielowymiarowe i tablice tablic. Z deklarowaniem i stosowaniem tablic związane są pewne reguły i osobliwości. W tabeli 3.3 wymieniono wybrane z najczęściej popełnianych błędów, co pomoże Ci utrważyć obowiązujące zasady. Zacznij od przyjrzenia się kodowi z pierwszej kolumny (bez zaglądania do dwóch pozostałych kolumn), by sprawdzić, czy rozumiesz tablice i ich składnię.

Tabela 3.3. Typowe błędy związane z programowaniem tablic

Typowe błędy	Opis błędu	Poprawiony kod
<code>int numbers[];</code>	Nawias kwadratowy służący do deklarowania tablicy należy podać po typie danych, a nie po identyfikatorze.	<code>int[] numbers;</code>
<code>int[] numbers; numbers = {42, 84, 168};</code>	Gdy wartość tablicy jest przypisywana po deklaracji, trzeba dodać słowo kluczowe new i podać typ danych.	<code>int[] numbers; numbers = new int[] {42, 84, 168};</code>
<code>int[3] numbers = {42, 84, 168};</code>	Nie można określić wielkości tablicy w deklaracji zmiennej.	<code>int[] numbers = {42, 84, 168};</code>
<code>int[] numbers = new int[];</code>	W trakcie inicjowania trzeba określić wielkość tablicy, chyba że podany jest literal tablicowy.	<code>int[] numbers = new int[3];</code>
<code>int[] numbers = new int[3]{}</code>	Wielkość tablicy jest ustalona na 3, ale w literale tablicowym nie podano żadnych elementów. Wielkość tablicy musi pasować do liczby elementów w literale.	<code>int[] numbers = new int[3] {42, 84, 168};</code>
<code>int[] numbers = new int[3]; Console.WriteLine(numbers[3]);</code>	Tablice są indeksowane od zera. Dlatego ostatni element ma indeks o jeden mniejszy niż wielkość tablicy. Zauważ, że ten błąd jest zgłoszany w czasie wykonywania programu, a nie w trakcie komilacji.	<code>int[] numbers = new int[3]; Console.WriteLine(numbers[2]);</code>
<code>int[] numbers = new int[3]; numbers[^0] = 42;</code>	Ten sam błąd co wcześniej. Trzeba użyć liczonego od końca indeksu ^1, by uzyskać dostęp do ostatniego elementu. Zapis ^0 oznacza — nieistniejący — element poza końcem tablicy. Zauważ, że ten błąd jest zgłoszany w czasie wykonywania programu, a nie w trakcie komilacji.	<code>int[] numbers = new int[3]; numbers[^1] = 42;</code>
<code>int[] numbers = new int[3]; numbers[numbers.Length] = 42;</code>	Ten sam błąd co wcześniej. Trzeba odjąć 1 od Length, by uzyskać dostęp do ostatniego elementu. Zauważ, że ten błąd jest zgłoszany w czasie wykonywania programu, a nie w trakcie komilacji.	<code>int[] numbers = new int[3]; numbers[numbers.Length - 1] = 42;</code>
<code>int[] numbers; Console.WriteLine(numbers[0]);</code>	Do elementów zmiennej numbers nie można uzyskać dostępu, ponieważ nie przypisano do niej instancji tablicy.	<code>int[] numbers = {42, 84}; Console.WriteLine(numbers[0]);</code>
<code>int[,] numbers = { {42}, {84, 42} };</code>	Wielowymiarowe tablice muszą mieć spójną strukturę.	<code>int[,] numbers = { {42, 168}, {84, 42} };</code>
<code>int[][] numbers = { {42, 84}, {84, 42} };</code>	W tablicy tablic elementami muszą być instancje tablic.	<code>int[][] numbers = { { new int[] {42, 84}, new int[] {84, 42} } };</code>

Podsumowanie

W tym rozdziale najpierw opisano dwa różne rodzaje typów: bezpośrednie i referencyjne. Są to podstawowe informacje, które programiści języka C# powinni zrozumieć. Rodzaj typu wpływa na jego działanie, przy czym może to nie być oczywiste na podstawie samej lektury kodu.

Przed omówieniem tablic opisano dwa mechanizmy wprowadzone w nowszych wersjach języka. Najpierw omówiono modyfikator umożliwiający używanie wartości null (?), dodany w wersji C# 2.0 i pozwalający zapisywać wartość null w typach bezpośrednich (od wersji C# 2.0) i tworzyć typy referencyjne dopuszczające wartość null (od wersji C# 8.0). Ten modyfikator umożliwia deklarowanie dopuszczania wartości null. Z perspektywy technicznej modyfikator pozwala zapisywać null w typach bezpośrednich i jawnie określać w typach referencyjnych to, czy mają dopuszczać wartości null. Przedstawiono też krótki i nową składnię dodaną w C# 7.0, która umożliwia korzystanie z krotek bez konieczności bezpośredniego stosowania typu danych używanego dla krotek.

W końcowej części rozdziału omówiono składnię do obsługi tablic w języku C#, a także różne metody manipulowania tablicami. Dla wielu programistów składnia może się początkowo wydawać kłopotliwa, dlatego w podrozdziale poświęconym tablicom przedstawiono listę typowych błędów popełnianych w trakcie korzystania z tej struktury.

W kolejnym rozdziale omówiono wyrażenia i instrukcje związane z przepływem sterowania. Znajdziesz tam też omówienie instrukcji if, która pojawiła się kilkakrotnie w końcowej części niniejszego rozdziału.

4

Operatory i przepływ sterowania

W TYM ROZDZIALE przeczytasz o operatorach, instrukcjach związanych z przepływem sterowania i preprocesorze języka C#. **Operatory** zapewniają składnię potrzebną do wykonywania różnych obliczeń lub operacji odpowiednich dla używanych operandów. **Instrukcje związane z przepływem sterowania** umożliwiają tworzenie logiki warunkowej w programie lub wielokrotne powtarzanie bloku kodu w pętli. Po omówieniu instrukcji `if` opisano wyrażenia logiczne, stosowane w wielu instrukcjach związanych z przepływem sterowania. Wspomniano też, że liczb całkowitych nie można przekształcać (nawet jawnie) na typ `bool`, i wyjaśniono korzyści płynące z tego ograniczenia. Rozdział kończy się omówieniem dyrektyw preprocesora języka C#.



Operatory

Po zapoznaniu się z predefiniowanymi typami danych (zobacz rozdział 2.) możesz zacząć się uczyć, jak posługiwać się tymi typami w połączeniu z operatorami w celu wykonywania obliczeń. Możesz na przykład przeprowadzać obliczenia na zadeklarowanych zmiennych.

ZAGADNIENIE DLA POCZĄTKUJĄCYCH

Operatory

Operatory służą do wykonywania operacji matematycznych lub logicznych na wartościach (bądź zmiennych) nazywanych **operandami**. W efekcie powstaje nowa wartość, **wynik**. Na przykład na listingu 4.1 operator odejmowania (-) jest używany do wykonania odejmowania na dwóch operandach — liczbach 4 i 2. Wynik tej operacji jest zapisywany w zmiennej difference.

Listing 4.1. Prosty przykład ilustrujący używanie operatora

```
int difference = 4 - 2;
```

Operatory dzielą się na trzy ogólne kategorie: operatory jednoargumentowe, dwuargumentowe i trójargumentowe. Używają one jednego, dwóch lub trzech operandów. Niektóre operatory są reprezentowane za pomocą symboli (na przykład +, -, ?, i ??), a inne mają postać słów kluczowych (takich jak default i is). W tym podrozdziale opisano wybrane z podstawowych operatorów jedno- i dwuargumentowych. Operatory trójargumentowe omówiono dalej w rozdziale.

Operatory jednoargumentowe plus (+) i minus (-)

Czasem potrzebna jest zmiana znaku wartości liczbowych. Przydatny jest wówczas jednoargumentowy operator minus (-). Na przykład kod z listingu 4.2 zmienia wartość dłużu Stanów Zjednoczonych na liczbę ujemną, aby podkreślić, że jest to kwota, jaką państwo jest winne.

Listing 4.2. Tworzenie wartości ujemnych¹

```
// Dług publiczny z dokładnością co do centa
decimal debt = -26457079712930.80M;
```

Użycie operatora minus to odpowiednik odjęcia operandu od zera.

Jednoargumentowy operator plus (+) rzadko² powoduje zmianę wartości. Jest to zbędny dodatek do języka C#, wprowadzony w celu zachowania symetrii.

¹ Na dzień 1 lipca 2020 roku (wg strony <http://www.treasurydirect.gov>).

² Operator jednoargumentowy + może przyjmować operandy typów int, uint, long, ulong, float, double i decimal (włącznie z wersjami tych typów akceptującymi wartość null). Użycie go dla innych typów liczbowych, takich jak short, powoduje przekształcenie operandu na odpowiedni z wymienionych typów.

Arytmetyczne operatory dwuargumentowe (+, -, *, /, %)

Operatory dwuargumentowe wymagają dwóch operandów. W języku C# dla takich operatorów używana jest notacja wrostkowa — operator znajduje się między operandami lewym i prawym. Wynik działania operatora dwuargumentowego innego niż operator przypisania należy w odpowiedni sposób wykorzystać (na przykład jako operand w innym wyrażeniu, takim jak przypisanie).

Porównanie języków — samodzielne instrukcje z operatorem w języku C++

W języku C++ opisana wcześniej reguła nie obowiązuje, dlatego samodzielne wyrażenie z operatorem dwuargumentowym (na przykład `4+5`) może stanowić całą instrukcję i zostanie skompilowane. W języku C# tylko wyrażenia przypisania, wywołania, inkrementacji, dekrementacji i utworzenia obiektu mogą stanowić całe instrukcje.

Przykładowe odejmowanie na listingu 4.3 pokazuje, jak używać operatorów dwuargumentowych (arytmetycznych). Po obu stronach takich operatorów występują operandy, a obliczone wartości są przypisywane. Oprócz operatora odejmowania dostępne są arytmetyczne operatory dwuargumentowe dla dodawania (+), dzielenia (/), mnożenia (*) i reszty (%; inna nazwa to operator modulo).

Listing 4.3. Korzystanie z operatorów dwuargumentowych

```
class Division
{
    static void Main()
    {
        int numerator;
        int denominator;
        int quotient;
        int remainder;

        System.Console.Write("Wprowadź licznik: ");
        numerator = int.Parse(System.Console.ReadLine());

        System.Console.Write("Wrowadź mianownik: ");
        denominator = int.Parse(System.Console.ReadLine());

        quotient = numerator / denominator;
        remainder = numerator % denominator;

        System.Console.WriteLine(
            $"{numerator} / {denominator} = {quotient}, reszta {remainder}");
    }
}
```

Wynik działania kodu z listingu 4.3 pokazano w danych wyjściowych 4.1.

DANE WYJŚCIOWE 4.1.

Wprowadź licznik: 23

Wprowadź mianownik: 3

23 / 3 = 7, reszta 2

W wyróżnionych instrukcjach przypisania operacje dzielenia i obliczania reszty są wykonywane przed przypisaniem. Kolejność działania operatorów zależy od ich **priorytetów i łączności**. Priorytety używanych do tego miejsca operatorów są następujące:

1. Operatory *, / i % mają najwyższy priorytet.
2. Operatory + i - mają niższy priorytet.
3. Operator = ma najniższy priorytet spośród wymienionych operatorów.

Dlatego można przyjąć, że instrukcja działa zgodnie z oczekiwaniemi, a operatory dzielenia i obliczania reszty są przetwarzane przed przypisaniem.

Jeśli zapomnisz przypisać wynik wykonania jednego z operatorów dwuargumentowych, wystąpi błąd komplikacji przedstawiony w danych wyjściowych 4.2.

DANE WYJŚCIOWE 4.2.

```
... error CS0201: Only assignment, call, increment, decrement,
and new object expressions can be used as a statement
```

ZAGADNIENIE DLA POCZĄTKUJĄCYCH**Nawiasy, łączność, priorytety i określanie wartości**

Gdy wyrażenie obejmuje wiele operatorów, czasem nie jest oczywiste, jakie są ich operandy. Na przykład w wyrażeniu $x+y*z$ oczywiste jest, że wyrażenie x to operand dodawania, a z to operand mnożenia. Jednak czy y jest operandem dodawania, czy mnożenia?

Nawiasy umożliwiają jednoznaczne powiązanie operandu z operatorem. Jeśli chcesz zsumować y , możesz zapisać wyrażenie w postaci $(x+y)*z$. Jeżeli zaś y ma być operandem mnożenia, możesz użyć zapisu $x+(y*z)$.

Jednak język C# nie wymaga używania nawiasów w każdym wyrażeniu obejmującym więcej niż jeden operator. Kompilator stosuje zasady łączności i priorytety, by z kontekstu wywnioskować, które nawiasy zostały pominięte. **Łączność** określa, jak w nawiasach są łączone podobne operatory. **Priorytety** pozwalają zaś ustalić, jak w nawiasach łączone są różne operatory.

Operatory dwuargumentowe mogą być *łączne lewostronnie* lub *prawostronnie* (w zależności od tego, czy środkowe wyrażenie powinno się łączyć z operatorem występującym po lewej, czy po prawej stronie). Na przykład wyrażenie $a-b-c$ powinno być interpretowane jako $(a-b)-c$, a nie jako $a-(b-c)$. Oznacza to, że odejmowanie jest łączne lewostronnie. Większość operatorów w języku C# jest łączna lewostronnie. Operator przypisania jest łączny prawostronnie.

Gdy operatory są różne, ich **priorytet** służy do ustalania, z którą stroną należy złączyć środkowy operand. Na przykład mnożenie ma wyższy priorytet niż dodawanie, dlatego wyrażenie $x+y*z$ jest interpretowane jako $x+(y*z)$, a nie jako $(x+y)*z$.

Jednak nawet wtedy, gdy nawias nie zmienia znaczenia wyrażenia, warto go zastosować, aby zwiększyć czytelność kodu. Na przykład w kodzie przekształcającym stopnie Celsjusza na stopnie Fahrenheita wyrażenie $(c*9.0/5.0)+32.0$ jest bardziej czytelne niż $c*9.0/5.0+32.0$, choć nawias jest tu zbędny.

Wskazówka

STOSUJ nawiasy, by zwiększyć czytelność kodu (zwłaszcza gdy priorytety operatorów nie są oczywiste dla przypadkowego czytelnika).

Operatory o wyższym priorytecie muszą zostać przetworzone przed uwzględnieniem operatorów o niższym priorytecie. W wyrażeniu $x+y*z$ najpierw trzeba wykonać mnożenie, a dopiero potem dodawanie, ponieważ wynik mnożenia jest prawym operandem dodawania. Warto jednak zauważyć, że priorytety i łączność wpływają tylko na kolejność wykonywania operatorów. Nie wyznaczają one kolejności określania wartości operandów.

W języku C# wartości operandów zawsze są wyznaczane od lewej do prawej. W wyrażeniu z trzema wywołaniami metod, na przykład $A() + B() * C()$, najpierw określana jest wartość $A()$, potem $B()$, a następnie $C()$. Później operator mnożenia oblicza iloczyn, a na końcu operator dodawania wyznacza sumę. Choć $C()$ jest używane w mnożeniu, a $A()$ w mającym niższy priorytet dodawaniu, metoda $A()$ jest wywoływana przed metodą $C()$.

Porównanie języków — kolejność określania wartości operandów w języku C++

Specyfikacja języka C++ jest niezgodna z opisaną regułą i pozwala określić w implementacji języka kolejność określania wartości operandów. Dla wyrażenia $A() + B() * C()$ kompilator języka C++ może zdecydować się wywołać funkcje w dowolnej kolejności, przy czym iloczyn musi być jedną z sumowanych wartości. Kompilator może na przykład najpierw określić wartość $B()$, potem $A()$, następnie $C()$, później iloczynu, a w ostatnim kroku sumy.

Używanie operatora dodawania dla łańcuchów znaków

Operatory mogą też działać dla operandów nieliczbowych. Można na przykład zastosować operator dodawania w celu złączenia dwóch lub większej liczby łańcuchów znaków, co pokazano na listingu 4.4.

Listing 4.4. Używanie operatorów dwuargumentowych dla typów nieliczbowych

```
class FortyTwo
{
    static void Main()
    {
        short windSpeed = 42;
        System.Console.WriteLine(
            "Pierwszy most Tacoma w Waszyngtonie został\n"
            + "zniszczony przez wiatr wiejący z prędkością "
            + windSpeed + " mil na godzinę.");
    }
}
```

Wynik działania kodu z listingu 4.4 pokazano w danych wyjściowych 4.3.

DANE WYJŚCIOWE 4.3.

Pierwszy most Tacoma w Waszyngtonie został zniszczony przez wiatr wiejący z szybkością 42 mil na godzinę.

Ponieważ struktura zdań w językach z różnych kultur jest inna, programiści powinni unikać stosowania operatora dodawania do łańcuchów znaków, jeśli potrzebna może być lokalizacja aplikacji. Ponadto choć w łańcuchach znaków w języku C# 6.0 można zagnieździć wyrażenia (dzięki interpolacji), tłumaczenie aplikacji na inne języki wymaga przeniesienia łańcuchów znaków do pliku zasobów, co eliminuje działanie interpolacji. Dlatego powinieneś unikać stosowania operatora dodawania. Gdy wiesz, że może być potrzebne lokalizowanie, staraj się korzystać z formatowania złożonego.

Wskazówka

PRZEDŁADAJ łączenie łańcuchów znaków za pomocą formatowania złożonego (nad stosowanie operatorów dodawania), gdy potrzebne może być lokalizowanie aplikacji.

Używanie znaków w operacjach arytmetycznych

Gdy w rozdziale 2. przedstawiano typ char, wspomniano, że choć przechowuje on znaki, a nie liczby, jest typem **całkowitoliczbowym** (opartym na liczbach całkowitych). Dlatego można go używać w operacjach arytmetycznych razem z innymi typami całkowitoliczbowymi. Jednak w obliczeniach z użyciem typu char nie jest uwzględniany znak, ale reprezentująca go wartość. Na przykład cyfra 3 jest reprezentowana w standardzie Unicode za pomocą wartości 0x33 (w kodzie szesnastkowym), równej 51 dla podstawy 10. Cyfra 4 jest reprezentowana w standardzie Unicode za pomocą wartości 0x34, równej 52 dla podstawy 10. Dodanie cyfr 3 i 4 na listingu 4.5 prowadzi do uzyskania wartości szesnastkowej 0x67 (103 dla podstawy 10), odpowiadającej w standardzie Unicode literze g.

Listing 4.5. Używanie operatora plus dla danych typu char

```
int n = '3' + '4';
char c = (char)n;
System.Console.WriteLine(c); // Wyświetla g.
```

Wynik działania kodu z listingu 4.5 pokazano w danych wyjściowych 4.4.

DANE WYJŚCIOWE 4.4.

g

Możesz wykorzystać tę cechę typów znakowych do ustalania odległości między dwoma znakami. Na przykład litera f jest oddalona od litery c o trzy znaki. Możesz to ustalić w wyniku odjęcia litery c od litery f, co pokazano na listingu 4.6.

Listing 4.6. Określanie różnicy między dwoma znakami

```
int distance = 'f' - 'c';
System.Console.WriteLine(distance);
```

Wynik działania kodu z listingu 4.6 przedstawiono w danych wyjściowych 4.5.

DANE WYJŚCIOWE 4.5.

3

Wyjątkowe cechy typów zmiennoprzecinkowych

Binarne typy zmiennoprzecinkowe `float` i `double` mają pewne specjalne cechy (na przykład sposób obsługi precyzji). W tym podrozdziale przedstawiono ilustrujące to przykłady, a także wyjaśniono wybrane wyjątkowe cechy takich typów.

Typ `float` o siedmiu cyfrach precyzji (dla liczb w systemie dziesiętnym) może przechowywać wartości 1 234 567 i 0,1234567. Jeśli jednak dodasz te dwie wartości do siebie, suma zostanie zaokrąglona do liczby 1 234 567, ponieważ dokładny wynik wymaga większej precyzji, niż zapewnia to typ `float`. Błąd spowodowany zaokrąglaniem do siedmiu cyfr może się stać znaczący względem obliczanej wartości (zwłaszcza gdy wykonywanych jest wiele obliczeń). Zobacz też ZAGADNIENIE DLA ZAAWANSOWANYCH „Nieoczekiwane nierówności wartości typów zmiennoprzecinkowych” w dalszej części podrozdziału.

Wewnętrznie binarne typy zmiennoprzecinkowe przechowują ułamek binarny, a nie dziesiętny. Dlatego błędy reprezentacji mogą się pojawiać nawet w wyniku prostych przypisań, takich jak `double number = 140.6F`. Wartość 140,6 to ułamek $703/5$, jednak mianownik nie jest tu potęgą liczby 2, dlatego wartości nie można precyzyjnie przedstawić za pomocą binarnej liczby zmiennoprzecinkowej. Reprezentacja wartości to najbliższy ułamek z potągą liczby 2 w mianowniku mieszczący się w 16 bitach (jest to pojemność typu `float`).

Ponieważ typ `double` potrafi przechowywać dokładniejsze wartości niż typ `float`, kompilator języka C# interpretuje wyrażenie jako `double number = 140.600006103516`, ponieważ 140.600006103516 to liczba typu `float` najbliższa (w notacji binarnej) wartości 140,6. Ponieważ liczba z przedstawionego wyrażenia jest reprezentowana za pomocą typu `double`, ma wartość nieco większą niż 140,6.

Wskazówka

UNIKAJ stosowania binarnych typów zmiennoprzecinkowych, gdy potrzebne są precyzyjne obliczenia arytmetyczne na liczbach dziesiętnych. Stosuj wtedy typ zmiennoprzecinkowy `decimal`.

ZAGADNIENIE DLA ZAAWANSOWANYCH

Nieoczekiwane nierówności wartości typów zmiennoprzecinkowych

Ponieważ liczby zmiennoprzecinkowe mogą być nieoczekiwane zaokrąglane do niedziesięciennych ułamków, porównywanie wartości zmiennoprzecinkowych przynosi czasem zaskakujące wyniki. Przyjrzyj się kodowi z listingu 4.7.

Listing 4.7. Nieoczekiwana nierówność z powodu braku precyzji typów zmiennoprzecinkowych

```
decimal decimalNumber = 4.2M;
double doubleNumber1 = 0.1F * 42F;
double doubleNumber2 = 0.1D * 42D;
float floatNumber = 0.1F * 42F;

Trace.Assert(decimalNumber != (decimal)doubleNumber1);
// 1. Wyświetla: 4.2 != 4.20000006258488
System.Console.WriteLine(
    $"{decimalNumber} != {(decimal)doubleNumber1}");

Trace.Assert((double)decimalNumber != doubleNumber1);
// 2. Wyświetla: 4.2 != 4.20000006258488
System.Console.WriteLine(
    $"{{(double)}decimalNumber} != {doubleNumber1}");

Trace.Assert((float)decimalNumber != floatNumber);
// 3. Wyświetla: (float)4.2M != 4.2F
System.Console.WriteLine(
    $"{{(float)}}{{(float)}}decimalNumber{M} != {floatNumber}{F}");

Trace.Assert(doubleNumber1 != (double)floatNumber);
// 4. Wyświetla: 4.20000006258488 != 4.2000028610229
System.Console.WriteLine(
    $"{doubleNumber1} != {(double)floatNumber}");

Trace.Assert(doubleNumber1 != doubleNumber2);
// 5. Wyświetla: 4.20000006258488 != 4.2
System.Console.WriteLine(
    $"{doubleNumber1} != {doubleNumber2}");

Trace.Assert(floatNumber != doubleNumber2);
// 6. Wyświetla: 4.2F != 4.2D
System.Console.WriteLine(
    $"{{floatNumber}}F != {{doubleNumber2}}D");

Trace.Assert((double)4.2F != 4.2D);
// 7. Wyświetla: 4.1999980926514 != 4.2
System.Console.WriteLine(
    $"{{(double)}}{{(double)}}4.2F} != {4.2D}");
```

Wynik działania kodu z listingu 4.7 przedstawiono w danych wyjściowych 4.6.

DANE WYJŚCIOWE 4.6.

```
4.2 != 4.2000006258488
4.2 != 4.2000006258488
(float)4.2M != 4.2F
4.2000006258488 != 4.2000028610229
4.2000006258488 != 4.2
4.2F != 4.2D
4.1999980926514 != 4.2
4.2F != 4.2D
```

Metoda `Assert()` alarmuje programistę, gdy jej argumenty mają wartość `false`. Jednak żadne z wartości porównywanych na tym listingu nie są sobie równe. Mimo pozornej równości wartości liczby nie są takie same. Przyczyną jest brak precyzji wartości typu `float`.

Wskazówka

UNIKAJ w instrukcjach warunkowych porównań wartości binarnych typów zmienoprzecinkowych. Powinieneś albo odjąć od siebie dwie wartości i sprawdzić, czy różnica między nimi jest mniejsza niż wartość progowa, albo zastosować typ `decimal`.

Powinieneś pamiętać także o innych wyjątkowych cechach typów zmienoprzecinkowych. Standardowo można oczekwać, że dzielenie liczby całkowitej przez zero spowoduje błąd. I rzeczywiście tak się dzieje, gdy używane są typy takie jak `int` lub `decimal`. Jednak typy `float` i `double` przyjmują specjalne wartości. Przyjrzyj się listingowi 4.8 i powiązanym z nim danym wyjściowym 4.7.

Listing 4.8. Dzielenie wartości typu `float` przez zero powoduje wyświetlenie wartości `NaN`

```
float n=0f;
// Wyświetla: NaN
System.Console.WriteLine(n / 0);
```

DANE WYJŚCIOWE 4.7.

NaN

W matematyce niektóre operacje matematyczne są niezdefiniowane. Jedną z nich jest dzielenie przez zero. W języku C# dzielenie zera typu `float` przez zero daje specjalną wartość „nie-liczba”, wyświetlana jako `NaN` (od ang. *not a number*). Także próba wyciągnięcia pierwiastka kwadratowego z liczby ujemnej za pomocą instrukcji `System.Math.Sqrt(-1)` prowadzi do wyświetlenia wartości `NaN`.

Może nastąpić przepełnienie liczby zmienoprzecinkowej. Na przykład górna granica typu `float` to około $3,4 \times 10^{38}$. Jeśli wartość przekroczy to ograniczenie, wynik zostanie zapisany jako **dodatnia nieskończoność**, wyświetlana jako `Infinity`. Podobnie dolna granica typu `float` to $-3,4 \times 10^{38}$. Wyliczona wartość poniżej tej granicy to **ujemna nieskończoność**, reprezentowana

za pomocą łańcucha znaków –Infinity. Kod z listingu 4.9 generuje wartości odpowiadające ujemnej i dodatniej nieskończoności. Efekt wykonania tego kodu pokazano w danych wyjściowych 4.8.

Listing 4.9. Wyjście poza granice typu float

```
// Wyświetla: -Infinity  
System.Console.WriteLine(-1f / 0);  
// Wyświetla: Infinity  
System.Console.WriteLine(3.402823E+38f * 2f);
```

DANE WYJŚCIOWE 4.8.

```
-Infinity  
Infinity
```

Liczby zmiennoprzecinkowe mogą też przechowywać wartość bardzo bliską zera, ale niezerową. Jeśli wartość przekracza dolny próg dla typu float lub double, może zostać przedstawiona jako **ujemne lub dodatnie zero** (w zależności od tego, czy liczba jest dodatnia, czy ujemna). Takie wartości są reprezentowane jako -0 i 0.

Złożone operatory przypisania (+=, -=, *=, /=, %=)

W rozdziale 1. opisano prosty operator przypisania, zapisujący wartość podaną po prawej stronie operatora w zmiennej określonej po stronie lewej. **Złożone operatory przypisania** łączą standardowe obliczenia z użyciem operatora dwuargumentowego z operatorem przypisania. Przyjrzyj się przykładowemu listingowi 4.10.

Listing 4.10. Typowa inkrementacja

```
int x = 123;  
x = x + 2;
```

W tym przypisaniu najpierw obliczana jest wartość $x + 2$, po czym jest ona przypisywana do x . Ponieważ operacje tego rodzaju są wykonywane często, istnieje operator przypisania pozwalający wykonać obliczenia i przypisać wartość w jednym kroku. Operator $+=$ powiększa zmienną podaną po lewej stronie o wartość podaną po stronie prawej. Pokazano to na listingu 4.11.

Listing 4.11. Używanie operatora $+=$

```
int x = 123;  
x += 2;
```

Ten kod działa tak samo jak kod z listingu 4.10.

Istnieje też wiele innych złożonych operatorów przypisania działających w podobny sposób. Możesz użyć operatora przypisania razem z operatorami odejmowania, mnożenia, dzielenia i reszty, co pokazano na listingu 4.12.

Listing 4.12. Inne przykłady użycia operatora przypisania

```
x -= 2;
x /= 2;
x *= 2;
x %= 2;
```

Operatory inkrementacji i dekrementacji (++, --)

Język C# udostępnia specjalne operatory jednoargumentowe służące do inkrementacji i dekrementacji liczników. **Operator inkrementacji**, `++`, przy każdym wywołaniu zwiększa wartość zmiennej o jeden. Oznacza to, że wszystkie wiersze kodu na listingu 4.13 działają tak samo.

Listing 4.13. Operator inkrementacji

```
spaceCount = spaceCount + 1;
spaceCount += 1;
spaceCount++;
```

Możesz też zmniejszyć wartość zmiennej o jeden, używając **operatora dekrementacji**, `--`. Dlatego wszystkie wiersze kodu na listingu 4.14 działają identycznie.

Listing 4.14. Operator dekrementacji

```
lines = lines - 1;
lines -= 1;
lines--;
```

ZAGADNIENIE DLA POCZĄTKUJĄCYCH**Przykładowa dekrementacja w pętli**

Operatory inkrementacji i dekrementacji najczęściej stosowane są w pętlach, na przykład w opisanej dalej w rozdziale pętli `while`. Na listingu 4.15 operator dekrementacji jest używany do pobrania wszystkich liter alfabetu od ostatniej do pierwszej.

Listing 4.15. Wyświetlanie wartości Unicode wszystkich liter w kolejności od ostatniej do pierwszej

```
char current;
int unicodeValue;

// Ustawianie początkowej wartości zmiennej current.
current = 'z';

do
{
    // Pobieranie wartości Unicode dla zmiennej current.
    unicodeValue = current;
    System.Console.WriteLine($"{current}={unicodeValue}\t");

    // Przejście do poprzedniej litery alfabetu.
    current--;
}
while(current >= 'a');
```

Wynik działania kodu z listingu 4.15 pokazano w danych wyjściowych 4.9.

DANE WYJŚCIOWE 4.9.

```
z=122  y=121  x=120  w=119  v=118  u=117  t=116  s=115  r=114
q=113  p=112  o=111  n=110  m=109  l=108  k=107  j=106  i=105
h=104  g=103  f=102  e=101  d=100  c=99  b=98  a=97
```

Operatory inkrementacji i dekrementacji są używane na listingu 4.15 do kontrolowania liczby powtórzeń określonej operacji. Zwróć uwagę, że w tym kodzie operator inkrementacji jest wykorzystywany dla typu znakowego (char). Operatory inkrementacji i dekrementacji można stosować do różnych typów danych, pod warunkiem jednak, że w danym typie można określić następną i poprzednią wartość.

Wiesz już, że operator przypisania najpierw oblicza wartość, a potem ją przypisuje. Wynikiem działania operatora przypisania jest przypisana wartość. Operatory inkrementacji i dekrementacji działają podobnie — obliczają wartość, przypisują ją, a wynikiem jest właśnie ona. Dlatego możliwe jest zastosowanie operatora przypisania razem z operatorem inkrementacji lub dekrementacji, choć brak ostrożności może tu prowadzić do powstania trudnego do zrozumienia kodu. Przyjrzyj się listowi 4.16 i danym wyjściowym 4.10.

Listing 4.16. Używanie operatora postinkrementacji

```
int count = 123;
int result;
result = count++;
System.Console.WriteLine(
    $"result = {result} i count = {count}");
```

DANE WYJŚCIOWE 4.10.

```
result = 123 i count = 124
```

Możesz być zaskoczony tym, że do zmiennej `result` kod przypisał wartość zmiennej `count` *sprzed* inkrementacji. Umiejscowienie operatora inkrementacji lub dekrementacji określa, czy kod ma przypisać wartość operandu sprzed obliczeń, czy wartość końcową. Jeśli chcesz, by do zmiennej `result` przypisywana była wartość przypisana do `count`, umieść operator przed inkrementowaną zmienną — tak jak na listingu 4.17.

Listing 4.17. Używanie operatora preinkrementacji

```
int count = 123;
int result;
result = ++count;
System.Console.WriteLine(
    $"result = {result} i count = {count}");
```

Wynik działania kodu z listingu 4.17 przedstawiono w danych wyjściowych 4.11.

DANE WYJŚCIOWE 4.11.

```
result = 124 i count = 124
```

W tym przykładzie operator inkrementacji występuje przed operandem, dlatego wynik wyrażenia jest jednocześnie wartością przypisywaną do drugiej zmiennej po inkrementacji. Jeśli zmienna count jest równa 123, instrukcja `++count` spowoduje przypisanie do count liczby 124 i zwrócenie wyniku 124. Natomiast operator postinkrementacji, `count++`, przypisuje wartość 124 do count, ale zwraca wartość przechowywaną przez zmienną count przed inkrementacją (123). Niezależnie od tego, czy używany jest operator postinkrementacji, czy preinkrementacji, zmienna count jest zwiększana przed zwróceniem wyniku. Jedyna różnica polega na tym, jaki wynik jest zwracany. Ilustruje to listing 4.18. Wynik jego działania przedstawiono w danych wyjściowych 4.12.

Listing 4.18. Porównanie działania operatorów preinkrementacji i postinkrementacji

```
class IncrementExample
{
    static void Main()
    {
        int x = 123;
        // Wyświetla 123, 124, 125.
        System.Console.WriteLine($"{x++}, {x++}, {x}");
        // Teraz x zawiera wartość 125.
        // Wyświetla 126, 127, 127.
        System.Console.WriteLine($"{++x}, {++x}, {x}");
        // Teraz x zawiera wartość 127.
    }
}
```

DANE WYJŚCIOWE 4.12.

```
123, 124, 125
126, 127, 127
```

Na listingu 4.18 pokazano, że lokalizacja operatorów inkrementacji i dekrementacji względem operandu wpływa na wynik zwracany przez wyrażenie. Operatory przyrostkowe zwracają wartość zmiennej przed inkrementacją lub dekrementacją. Operatory przedrostkowe zwracają wartość zmiennej po inkrementacji lub dekrementacji. Zachowaj ostrożność, gdy używasz takich operatorów wewnątrz instrukcji. Gdy masz wątpliwości odnośnie do przebiegu operacji, zastosuj omawiane operatory osobno, w odrębnych instrukcjach. Dzięki temu kod będzie bardziej czytelny, a jego przeznaczenie stanie się zrozumiałe.

Porównanie języków — zależne od implementacji działanie języka C++

Wcześniej wspomniano, że w języku C++ określanie wartości operandów w wyrażeniu może się odbywać w dowolnej kolejności, natomiast w języku C# wyznaczanie wartości operandów zawsze jest wykonywane od lewej do prawej. Ponadto w języku C++ efekty uboczne inkrementacji i dekrementacji mogą zachodzić w dowolnym porządku. Na przykład w C++ wywołanie `M(x++, x++)`, gdzie x początkowo jest równe 1, może (w zależności od kompilatora) zostać wykonane jako `M(1, 2)` lub `M(2, 1)`. Natomiast język C# zawsze wywoła polecenie `M(1, 2)`, ponieważ gwarantuje dwie rzeczy: (1) argumenty wywołania zawsze są przetwarzane od lewej do prawej, (2) przypisanie inkrementowanej wartości do zmiennej zawsze odbywa się przed użyciem wartości wyrażenia. W języku C++ żadna z tych gwarancji nie obowiązuje.

Wskazówki

UNIKAJ stosowania operatorów inkrementacji i dekrementacji w miejscach, gdzie jest to mylące.

ZACHOWAJ ostrożność, gdy przenosisz między językami C, C++ i C# kod używający operatorów inkrementacji i dekrementacji. W językach C i C++ obowiązują inne reguły niż w C#.

ZAGADNIENIE DLA ZAAWANSOWANYCH

Inkrementacja i dekrementacja bezpieczna ze względu na wątki

Choć operatory inkrementacji i dekrementacji są zwiędłe, nie działają atomowo. W trakcie pracy takiego operatora może nastąpić przełączenie kontekstu wątku i wystąpienie sytuacji wyścigu. Powinieneś stosować instrukcję `lock`, by zapobiec takiej sytuacji. Jednak w prostych inkrementacjach i dekrementacjach można zastosować mniej kosztowną alternatywę — bezpieczne ze względu na wątki metody `Increment()` i `Decrement()` klasy `System.Threading.Interlocked`. Te metody wykorzystują funkcje procesora do wykonywania szybkich, bezpiecznych ze względu na wątki operacji inkrementacji i dekrementacji. Więcej informacji na ten temat zawiera rozdział 19.

Wyrażenia o stałej wartości i stałe lokalne

W rozdziale 3. opisano literaly — wartości umieszczane bezpośrednio w kodzie. Za pomocą operatorów można połączyć kilka takich literalów w **wyrażenie o stałej wartości**. Kompilator języka C# może określić wartość takiego wyrażenia w czasie komplikacji programu (a nie w trakcie jego wykonywania), ponieważ wszystkie operandy są stałe. Wyrażenia o stałej wartości mogą służyć do inicjowania stałych lokalnych, co pozwala nadać nazwę stałej wartości (podobnie jak zmienne lokalne umożliwiają nazwanie miejsca w pamięci). Na przykład obliczenia liczby sekund w dniu można zapisać w stałym wyrażeniu, a następnie za pomocą nazwy wykorzystać wynik tego wyrażenia w innych wyrażeniach.

Słowo kluczowe `const` na listingu 4.19 deklaruje stałą lokalną. Ponieważ stałe lokalne z definicji są przeciwieństwem **zmiennych** (*stałe* są przecież niezmienne), próba późniejszego zmodyfikowania wartości w kodzie spowoduje błąd czasu kompilacji.

Listing 4.19. Deklarowanie stałej

```
// ...
public long Main()
{
    // ...
    Wyrażenie o stałej wartości
    const int secondsPerDay = 60 * 60 * 24;
    const int secondsPerWeek = secondsPerDay * 7;
    Stała
    // ...
}
```

Wskazówka

NIE zapisuj w stałych żadnych wartości, które mogą się później zmienić. Wartość liczby pi lub liczba protonów w atomie złota to stałe. Cena złota, nazwa firmy lub numer wersji programu mogą się zmienić.

Zauważ, że wyrażenie przypisane do nazwy `secondsPerWeek` na listingu 4.19 to wyrażenie o stałej wartości, ponieważ wszystkie operandy tego wyrażenia są stałymi.

Zarządzanie przepływem sterowania

Dalej w tym rozdziale znajduje się listing 4.45, w którym pokazano prosty sposób wyświetlania liczby w postaci binarnej. Jednak nawet tak prostego programu nie da się napisać bez instrukcji związanych z przepływem sterowania. Takie instrukcje kontrolują ścieżkę wykonywania programu. W tym podrozdziale opisano, jak zmienić kolejność wykonywania poleceń na podstawie instrukcji warunkowych. Dalej dowiesz się, jak za pomocą pętli wielokrotnie wykonywać grupy instrukcji.

Podsumowanie instrukcji związanych z przepływem sterowania przedstawiono w tabeli 4.1. Zauważ, że w kolumnie „Ogólna struktura instrukcji” opisano typowy sposób korzystania z instrukcji, a nie jej kompletną strukturę leksykalną. Człon *zagnieżdzona-instrukcja* w tabeli 4.1 może reprezentować dowolną instrukcję inną niż polecenie z etykietą lub deklaracją; zwykle taką „instrukcją” jest blok kodu.

Wszystkie wymienione w tabeli 4.1 instrukcje języka C# związane z przepływem sterowania występują w programie do gry w kółko i krzyżyk, który przedstawiono w kodzie źródłowym do rozdziału 4. w pliku *TicTacToe.cs* (zobacz stronę <http://itl.tc/EssentialCSharpSCC>). Ten program wyświetla planszę do gry w kółko i krzyżyk z prośbą o podanie ruchu przez poszczególnych graczy i po każdym posunięciu aktualizuje planszę.

Tabela 4.1. Instrukcje związane z przepływem sterowania

Instrukcja	OGÓLNA STRUKTURA INSTRUKCJI	Przykład
Instrukcja if	if (<i>wyrażenie-logiczne</i>) zagnieżdżona-instrukcja	if (input == "quit") { System.Console.WriteLine("Koniec gry"); return ; }
	if (<i>wyrażenie-logiczne</i>) zagnieżdżona-instrukcja else zagnieżdżona-instrukcja	if (input == "quit") { System.Console.WriteLine("Koniec gry"); return ; } else GetNextMove();
Instrukcja while	while (<i>wyrażenie-logiczne</i>) zagnieżdżona-instrukcja	while (count < total) { System.Console.WriteLine("count = {count}"); count++; }
Instrukcja do while	do zagnieżdżona-instrukcja while (<i>wyrażenie-logiczne</i>);	do { System.Console.WriteLine("Wprowadź nazwę:"); input = System.Console.ReadLine(); } while (input != "exit");
Instrukcja for	for (<i>inicjowanie-dla-for</i> ; <i>wyrażenie-logiczne</i> ; <i>iterator-dla-for</i>) zagnieżdżona-instrukcja	for (int count = 1; count <= 10; count++) { System.Console.WriteLine(\$"count = {count}"); }
Instrukcja foreach	foreach (<i>typ identyfikator</i> in <i>wyrażenie</i>) zagnieżdżona-instrukcja	foreach (char letter in email) { if (!insideDomain) { if (letter == '@') { insideDomain = true ; } continue ; } System.Console.Write(letter); }
Instrukcja continue	continue ;	

Tabela 4.1. Instrukcje związane z przepływem sterowania — ciąg dalszy

Instrukcja	Ogólna struktura instrukcji	Przykład
Instrukcja switch	switch (<i>wyrażenie-nadrzędne</i>) { ... case <i>stałe-wyrażenie</i> : <i>lista-instrukcji</i> <i>instrukcja-skoku</i> default : <i>lista-instrukcji</i> <i>instrukcja-skoku</i> }	switch (<i>input</i>) { case "exit": case "quit": System.Console.WriteLine("Kończenie pracy..."); break ; case "restart": Reset(); goto case "start"; case "start": GetMove(); break ; default : System.Console.WriteLine(<input/>); break ; }
Instrukcja break	break;	
Instrukcja goto	goto <i>identyfikator</i> ; goto case <i>stałe-wyrażenie</i> ; goto default ;	

W dalszej części rozdziału szczegółowo opisano każdą instrukcję. Po instrukcji **if** omówiono bloki kodu, zasięg, wyrażenia logiczne i operatory bitowe. Dopiero dalej przedstawiono pozostałe instrukcje związane z przepływem sterowania. Osoby, dla których tabela 4.1 wygląda znajomo (co może wynikać z podobieństw języka C# do innych języków), mogą przejść do podrozdziału „Dyrektyny preprocesora języka C#” lub od razu do podrozdziału „Podsumowanie” na końcu rozdziału.

Instrukcja if

Jest to jedna z najczęściej stosowanych instrukcji w języku C#. Sprawdza ona **wyrażenie logiczne** (którego wartość to `true` lub `false`), nazywane **warunkiem**. Jeśli warunek ma wartość `true`, wykonywana jest **instrukcja dla true**. W instrukcji **if** można opcjonalnie umieścić klauzulę **else**, obejmującą **instrukcję dla false** (wykonywaną, jeśli warunek ma wartość `false`). Ogólna postać instrukcji **if** wygląda więc tak:

```
if (warunek)
    instrukcja-dla-true
else
    instrukcja-dla-false
```

Kod z listingu 4.20 działa tak, że jeśli użytkownik wpisał 1, program wyświetla tekst Wybrano grę przeciwko komputerowi. W przeciwnym razie wyświetlana jest informacja Wybrano grę przeciwko drugiej osobie.

Listing 4.20. Przykładowa instrukcja if/else

```
class TicTacToe // Deklaracja klasy TicTacToe.
{
    static void Main() // Deklaracja punktu wejścia do programu.
    {
        string input;

        // Pytanie do użytkownika pozwalające rozpocząć grę dla 1 lub 2 osób.
        System.Console.WriteLine(
            "1 – Gra przeciwko komputerowi\n"+
            "2 – Gra przeciwko innemu graczowi\n"+
            "Wybierz:");
        input = System.Console.ReadLine();

        if (input=="1")
            // Użytkownik wybrał grę przeciwko komputerowi.
            System.Console.WriteLine(
                "Wybrano grę przeciwko komputerowi.");
        else
            // Domyslnie wybierana jest gra dla 2 graczy (nawet jeśli użytkownik nie wpisał 2).
            System.Console.WriteLine(
                "Wybrano grę przeciwko drugiej osobie.");
    }
}
```

Zagnieżdżone instrukcje if

Czasem kod wymaga kilku instrukcji if. Kod z listingu 4.21 najpierw sprawdza, czy użytkownik chce zakończyć pracę programu. By zamknąć aplikację, należy wpisać 0 lub mniejszą liczbę. Jeśli wprowadzono inną wartość, program prosi użytkownika o podanie maksymalnej liczby ruchów w grze w kółko i krzyżyk.

Listing 4.21. Zagnieżdżone instrukcje if

```
1. class TicTacToeTrivia
2. {
3.     static void Main()
4.     {
5.         int input; // Deklaracja zmiennej, w której zapisywane są dane wejściowe.
6.
7.         System.Console.Write(
8.             "Jaka jest maksymalna liczba ruchów "+
9.             "w grze w kółko i krzyżyk?"+
10.            "(Wpisz 0, aby zakończyć): ");
11.
12.        // Instrukcja int.Parse() przekształca wartość
13.        // zwróconą przez ReadLine() na typ int.
```

```
14.     input = int.Parse(System.Console.ReadLine());
15.
16.     if (input <= 0) // Wiersz 16.
17.         // Zmienna input ma wartość 0 lub mniejszą.
18.         System.Console.WriteLine("Zamykanie programu..."); 
19.     else
20.         if (input < 9) // Wiersz 20.
21.             // Zmienna input ma wartość mniejszą niż 9.
22.             System.Console.WriteLine(
23.                 $"Maksymalna liczba ruchów w grze w kółko i krzyżyk " +
24.                 "jest większa niż {input}.");
25.     else
26.         if (input > 9) // Wiersz 26.
27.             // Zmienna input ma wartość większą niż 9.
28.             System.Console.WriteLine(
29.                 $"Maksymalna liczba ruchów w grze w kółko i krzyżyk " +
30.                 "jest mniejsza niż {input}.");
31.     else
32.         // Zmienna input ma wartość 9.
33.         System.Console.WriteLine( // Wiersz 33.
34.             "Dobrze, maksymalna liczba ruchów " +
35.             "w grze w kółko i krzyżyk wynosi 9.");
36.     }
37. }
```

Wynik działania kodu z listingu 4.21 pokazano w danych wyjściowych 4.13.

DANE WYJŚCIOWE 4.13.

Jaka jest maksymalna liczba ruchów w grze w kółko i krzyżyk? (Wpisz 0, aby zakończyć): 9
Dobrze, maksymalna liczba ruchów w grze w kółko i krzyżyk wynosi 9.

Założymy, że użytkownik wpisał 9, a kod z wiersza 14. pobrał tę wartość. Ścieżka wykonania wygląda wtedy tak:

1. Wiersz 16. Kod sprawdza, czy zmienna `input` ma wartość mniejszą niż 0.
Ponieważ tak nie jest, następuje skok do wiersza 20.
2. Wiersz 20. Kod sprawdza, czy zmienna `input` ma wartość mniejszą niż 9.
Ponieważ tak nie jest, następuje skok do wiersza 26.
3. Wiersz 26. Kod sprawdza, czy zmienna `input` ma wartość większą niż 9.
Ponieważ tak nie jest, następuje skok do wiersza 33.
4. Wiersz 33. Kod wyświetla informację o tym, że odpowiedź jest prawidłowa.

Na listingu 4.21 występują zagnieżdżone instrukcje `if`. Aby podkreślić zagnieżdżenie, wiersze kodu wyróżniono wcięciem. Jednak, jak wyjaśniono w rozdziale 1., odstępły nie wpływają na ścieżkę wykonania. Nawet bez wcięć i znaków nowego wiersza kod działałby tak samo. Kod z zagnieżdżonymi instrukcjami `if` z listingu 4.22 działa tak samo jak wersja z listingu 4.21.

Listing 4.22. Instrukcje if/else sformatowane sekwencyjnie

```
if (input < 0)
    System.Console.WriteLine("Zamykanie programu...");
else if (input < 9)
    System.Console.WriteLine(
        $"Maksymalna liczba ruchów w grze w kółko i krzyżyk "+
        "jest większa niż {input}.");
else if (input > 9)
    System.Console.WriteLine(
        $"Maksymalna liczba ruchów w grze w kółko i krzyżyk "+
        "jest mniejsza niż {input}.");
else
    System.Console.WriteLine(
        "Dobrze, maksymalna liczba ruchów " +
        "w grze w kółko i krzyżyk wynosi 9.");
```

Choć druga z tych wersji jest częściej spotykana, w każdej sytuacji należy stosować format, który pozwala uzyskać bardziej przejrzysty kod.

W obu przedstawionych instrukcjach **if** pominięto nawiasy klamrowe. Jednak, co wyjaśniono dalej, nie jest to zgodne ze wskazówkami, ponieważ zalecane jest stosowanie wyróżnionych bloków kodu (oprócz najprostszego, jednowierszowych bloków).

Bloki kodu ({})

W poprzednich przykładach ilustrujących instrukcję **if** po **if** i **else** występowało tylko jedno polecenie — `System.Console.WriteLine()`. Taką sytuację przedstawia także listing 4.23.

Listing 4.23. Instrukcja if bez bloku kodu

```
if (input < 9)
    System.Console.WriteLine("Zamykanie programu");
```

Dzięki nawiasom klamrowym można połączyć grupę instrukcji w jeden **blok kodu (instrukcję blokową)**. Pozwala to zgrupować kilka instrukcji w jeden blok, który jest wykonywany, gdy dany warunek jest spełniony. Przyjrzyj się wyróżnionemu blokowi kodu w obliczeniach powierzchni koła na listingu 4.24.

Listing 4.24. Instrukcja if, po której następuje blok kodu

```
class CircleAreaCalculator
{
    static void Main()
    {
        double radius; // Deklaracja zmiennej przechowującej promień.
        double area;   // Deklaracja zmiennej przechowującej powierzchnię.

        System.Console.Write("Wprowadź promień koła: ");

        // Instrukcja double.Parse przekształca wartość zwróconą
        // przez polecenie ReadLine() na typ double.
        radius = double.Parse(System.Console.ReadLine());
        if (radius >= 0)
```

```
{  
    // Obliczanie powierzchni koła.  
    area = Math.PI * radius * radius;  
    System.Console.WriteLine(  
        $"Powierzchnia koła wynosi: { area : 0.00}");  
}  
else  
{  
    System.Console.WriteLine(  
        $"{ radius } nie jest poprawną wartością promienia.");  
}  
}  
}
```

Wynik działania kodu z listingu 4.24 przedstawiają dane wyjściowe 4.14.

DANE WYJŚCIOWE 4.14.

```
Wprowadź promień koła: 3  
Powierzchnia koła wynosi: 28.27
```

W tym przykładzie instrukcja `if` sprawdza, czy wartość zmiennej `radius` jest dodatnia. Jeśli tak jest, kod oblicza i wyświetla powierzchnię koła. W przeciwnym razie wyświetlany jest komunikat o nieprawidłowej wartości promienia.

Zauważ, że w tym przykładzie po pierwszej instrukcji `if` występują dwa polecenia. Znajdują się one jednak w nawiasie klamrowym. Nawias klamrowy łączy instrukcje w blok kodu, który jest traktowany jak jedna instrukcja.

Jeśli na listingu 4.24 pominiesz nawias klamrowy tworzący blok kodu, warunkowo wykonywana będzie tylko instrukcja znajdująca się bezpośrednio po wyrażeniu logicznym. Dalsze instrukcje będą wykonywane niezależnie od wyrażenia logicznego z instrukcji `if`. Błędną wersję kodu pokazano na listingu 4.25.

Listing 4.25. Poleganie tylko na wcięciach prowadzi do powstania błędnego kodu

```
if (radius >= 0)  
    area = Math.PI * radius * radius;  
    System.Console.WriteLine(  
        $"Powierzchnia koła wynosi: { area:0.00}");
```

W języku C# wcięcia służą jedynie do zwiększenia czytelności kodu. Kompilator ignoruje je, dlatego przedstawiony wcześniej kod jest semantycznie odpowiednikiem kodu z listingu 4.26.

Listing 4.26. Instrukcja if z nawiasami klamrowymi

```
if (radius >= 0)  
{  
    area = Math.PI * radius * radius;  
}  
System.Console.WriteLine(  
    $"Powierzchnia koła wynosi: { area:0.00}");
```

Programiści powinni zachować dużą ostrożność, by uniknąć trudnych do wykrycia błędów, takich jak usterka we wcześniej przedstawionym kodzie. Warto dodawać blok kodu po każdej instrukcji związanej z przepływem sterowania, nawet jeśli ma on obejmować tylko jedną instrukcję. Powszechnie akceptowaną wskazówką jest stosowanie nawiasów klamrowych zawsze z wyjątkiem najprostszych jednowierszowych instrukcji `if`.

Choć zdarza się to rzadko, blok kodu może nie być leksykalnie bezpośrednim elementem instrukcji związanej z przepływem sterowania. Oznacza to, że dodawanie nawiasów klamrowych niezależnie od instrukcji warunkowych lub pętli jest składowo dozwolone.

Na listingach 4.25 i 4.26 wartość liczby `pi` jest przedstawiona za pomocą zmiennej `PI` z klasy `System.Math`. Zamiast zapisywać na sztywno wartości takie jak 3,14 dla liczby `pi` lub stałą Eulera (`e`), w kodzie należy stosować stałe `System.Math.PI` i `System.Math.E`.

Wskazówka

UNIKAJ pomijania nawiasów klamrowych. Wyjątek możesz zrobić dla najprostszych jednowierszowych instrukcji `if`.

Bloki kodu, zasięgi i przestrzenie deklaracji

Bloki kodu czasem nazywa się *zasięgami*, jednak te dwa określenia nie oznaczają dokładnie tego samego. **Zasięg** dla nazwanego elementu to obszar w kodzie źródłowym, w którym można poprawnie używać tego elementu za pomocą niekwalifikowanej nazwy. Na przykład zasięg zmiennej lokalnej to tekst bloku kodu, w którym ta zmienna się znajduje. To dlatego bloki kodu nieraz są nazywane zasięgami.

Zasięgi są też czasem mylone z przestrzeniami deklaracji. **Przestrzeń deklaracji** jest logiczny kontener z nazwanymi elementami, w którym dwa elementy nie mogą mieć tej samej nazwy. Blok kodu definiuje nie tylko zasięg zmiennej lokalnej, ale też jej przestrzeń deklaracji. W jednej przestrzeni deklaracji nie mogą występować deklaracje dwóch zmiennych lokalnych o identycznych nazwach. Podobnie w jednej klasie nie można zadeklarować dwóch metod o sygnaturze `Main()`. Warto zauważyć, że dla metod ta reguła dopuszcza pewne wyjątki. Dwie metody w jednej przestrzeni deklaracji mogą mieć tę samą nazwę, jeśli różnią się sygnaturami. Sygnatura metody obejmuje jej nazwę oraz liczbę i typy parametrów. W bloku zmienną lokalną można wskazywać za pomocą nazwy. Zmienna musi być jedynym elementem zadeklarowanym przy użyciu tej nazwy w bloku. Poza blokiem z deklaracją nie można wskazywać tej zmiennej za pomocą jej nazwy. Zmienna lokalna poza blokiem znajduje się „poza zasięgiem”.

Oto podsumowanie — zasięg służy do określania, którego elementu dotyczy dana nazwa. Przestrzeń deklaracji określa, czy dwa elementy o tej samej nazwie powodują konflikt. Na listingu 4.27 deklaracja zmiennej lokalnej `message` w instrukcji blokowej zagnieżdżonej w instrukcji `if` sprawia, że zasięg zmiennej jest ograniczony tylko do tej instrukcji blokowej. Zmienna lokalna jest poza zasięgiem, gdy jej nazwa jest używana w dalszej części metody. Aby uniknąć błędu, trzeba zadeklarować zmienną poza blokiem.

Listing 4.27. Zmienne są niedostępne poza ich zasięgiem

```
class Program
{
    static void Main(string[] args)
```

```
{  
    int playerCount;  
    System.Console.Write(  
        "Wprowadź liczbę graczy (1 lub 2):");  
    playerCount = int.Parse(System.Console.ReadLine());  
    if (playerCount != 1 && playerCount != 2)  
    {  
        string message =  
            "Podajeś nieprawidłową liczbę graczy.";  
    }  
    else  
    {  
        // ...  
    }  
    // Błąd: zmienna message nie znajduje się w zasięgu.  
    System.WriteLine(message);  
}
```

Efekt działania listingu 4.27 został przedstawiony w danych wyjściowych 4.15.

DANE WYJŚCIOWE 4.15.

```
...  
...\Program.cs(18,26): error CS0103: The name 'message' does not exist  
in the current context
```

Przestrzeń deklaracji, w której nazwa zmiennej lokalnej musi być niepowtarzalna, obejmuje wszystkie podrzędne bloki kodu tekstowo zagnieźdzone w bloku, w którym zadeklarowano daną zmienną. Kompilator języka C# nie pozwala na to, by zmienną lokalną zadeklarowaną bezpośrednio w bloku kodu metody (lub jako parametr) ponownie wykorzystać w podrzędnym bloku kodu. Na listingu 4.27 nazwy args i playerCount są zadeklarowane w bloku kodu metody, dlatego nie można ich ponownie zadeklarować w innym miejscu metody.

Nazwa message oznacza zadeklarowaną zmienną lokalną w zasięgu tej zmiennej (czyli w bloku bezpośrednio obejmującym deklarację). Podobnie nazwa playerCount dotyczy tej samej zmiennej w całym bloku zawierającym jej deklarację — w tym w obu blokach podrzędnych wykonywanych dla wartości true i false z instrukcji if.

Porównanie języków — zasięg zmiennych lokalnych w języku C++

W języku C++ zmienna lokalna zadeklarowana w bloku znajduje się w zasięgu od miejsca deklaracji do końca bloku. Dlatego próba użycia zmiennej lokalnej przed deklaracją zakończy się niepowodzeniem, ponieważ dana zmienna nie znajduje się wtedy w zasięgu. Jeśli jednak w zasięgu dostępny jest inny element o tej samej nazwie, język C++ powiąże nazwę z tym elementem, co może być niezgodne z zamierzeniami programisty. W języku C# reguły wyglądają inaczej. Zmienna lokalna znajduje się w zasięgu w całym bloku, w którym jest zadeklarowana, jednak nie można jej używać przed deklaracją. Oznacza to, że przed deklaracją zmienna lokalna zostanie znaleziona, ale próba jej użycia zostanie uznana za błąd. Jest to jedna z wielu obowiązujących w języku C# reguł, które mają chronić przed błędami często występującymi w programach w języku C++.

Wyrażenia logiczne

Zapisany w nawiasie warunek w instrukcji if to **wyrażenie logiczne**. Na listingu 4.28 warunek jest wyróżniony.

Listing 4.28. Wyrażenie logiczne

```
if (input < 9)
{
    // Wartość zmiennej input jest mniejsza niż 9.
    System.Console.WriteLine(
        $"Maksymalna liczba ruchów w grze w kółko i krzyżyk "+
        "jest większa niż { input }.");
}
// ...
```

Wyrażenia logiczne występują w wielu instrukcjach związanych z przepływem sterowania. Podstawową cechą takich wyrażeń jest to, że ich wartość to zawsze true lub false. Aby kod `input < 9` mógł zostać użyty jako wyrażenie logiczne, musi zwracać wartość typu bool. Kompilator nie zezwala na stosowanie jako wyrażeń logicznych instrukcji takich jak `x = 42`, ponieważ to polecenie przypisuje wartość do zmiennej `x` i zwraca tę wartość. Nie jest to instrukcja sprawdzająca, czy wartość zmiennej wynosi 42.

Porównanie języków — pomyłkowe użycie = zamiast == w języku C++

Język C# eliminuje błąd programistyczny często występujący w językach C i C++. W C++ kod z listingu 4.29 jest dozwolony.

Listing 4.29. Język C++ (w odróżnieniu od C#) dopuszcza używanie przypisania w warunku

```
if (input = 9) // Dozwolone w C++, ale już nie w C#.
    System.Console.WriteLine(
        "Dobrze, maksymalna liczba ruchów w grze w kółko i krzyżyk to 9.");
```

Choć na pozór ten kod sprawdza, czy zmienna `input` ma wartość 9, w rozdziale 1. wyjaśniono, że operator `=` służy do przypisywania wartości, a nie sprawdzania równości. Wartość zwracana przez operator przypisania to przypisywana wartość (tu jest to 9). Liczba 9 jest typu int, dlatego w C# nie może być wyrażeniem logicznym i kompilator tego języka nie dopuszcza, by stosować ją w taki sposób. W językach C i C++ niezerowe liczby całkowite są traktowane jako odpowiednik wartości `true`, a liczba zero to odpowiednik wartości `false`. Język C# wymaga, by warunek miał typ logiczny, i nie dopuszcza stosowania w takim kontekście liczb całkowitych.

Operatory relacyjne i równości

Operatory **relacyjne** i **równości** określają, czy wartość jest większa, mniejsza lub równa względem innej. W tabeli 4.2 wymieniono wszystkie operatory relacyjne i równości. Wszystkie one są operatormi dwuargumentowymi.

Tabela 4.2. Operatory relacyjne i równości

Operator	Opis	Przykład
<	Mniejsze niż	input < 9;
>	Większe niż	input > 9;
<=	Mniejsze lub równe	input <= 9;
>=	Większe lub równe	input >= 9;
==	Operator równości	input == 9;
!=	Operator nierówności	input != 9;

W języku C# do sprawdzania równości służy operator == (podobnie jak w wielu innych językach programowania). Na przykład aby ustalić, czy zmienna input jest równa 9, należy użyć instrukcji input == 9. Operator równości obejmuje dwa znaki równości, co pozwala odróżnić go od operatora przypisania, =. Wykrzyknik w języku C# oznacza NIE, tak więc by sprawdzić, czy wartości są od siebie różne, należy zastosować operator nierówności, !=.

Operatory relacyjne i równości zawsze zwracają wartość typu bool (zobacz listing 4.30).

Listing 4.30. Przypisywanie wyniku zwróconego przez operator relacyjny do zmiennej typu bool

```
bool result = 70 > 7;
```

W kompletnym programie do gry w kółko i krzyżyk operator równości jest używany do sprawdzania, czy użytkownik chce zakończyć pracę programu. Wyrażenie logiczne z listingu 4.31 zawiera operator logiczny OR (||), opisany szczegółowo w następnym podrozdziale.

Listing 4.31. Używanie operatora równości w wyrażeniach logicznych

```
if (input == "" || input == "quit")
{
    System.Console.WriteLine($"Gracz {currentPlayer} zrezygnował!");
    break;
}
```

Operatory logiczne

Operatory logiczne przyjmują operandy logiczne i zwracają wartość logiczną. Takie operatory pozwalają łączyć wiele wyrażeń logicznych w bardziej złożone wyrażenia tego rodzaju. Operatory logiczne to: |, ||, &, && i ^ . Odpowiadają one operacjom OR, AND i XOR. Wersje | i & operatorów OR i AND rzadko stosuje się w wyrażeniach logicznych. Przyczyny opisano w dalszej części tego podrozdziału.

Operator OR (||)

Jeśli użytkownik kodu z listingu 4.31 wpisze quit lub wciśnie klawisz *Enter* bez podawania wartości, kod przyjmie, że należy zamknąć program. Aby umożliwić użytkownikowi dwa sposoby na rezygnację z gry, można wykorzystać logiczny operator OR, ||. Operator || sprawdza wartości wyrażeń logicznych i zwraca wartość true, jeśli *którykolwiek* z operandów ma tę wartość (zobacz listing 4.32).

Listing 4.32. Używanie operatora OR

```
if ((hourOfDay > 23) || (hourOfDay < 0))
    System.Console.WriteLine("Wprowadzona godzina jest nieprawidłowa.");
```

Nie trzeba określać wartości obu stron wyrażenia OR, ponieważ jeśli którykolwiek z operandów ma wartość true, wiadomo, że wynik całego wyrażenia też będzie równy true (drugi operand nie ma wtedy znaczenia). Podobnie jak we wszystkich operatorach języka C# lewy operand jest przetwarzany przed prawym. Dlatego jeśli lewa strona wyrażenia ma wartość true, prawa jest ignorowana. Jeśli w przykładzie z listingu 4.32 zmienna hourOfDay ma wartość 33, wtedy wyrażenie (hourOfDay > 23) też ma tę wartość i operator OR ignoruje drugą połowę wyrażenia. Na tym polega **przetwarzanie skrócone**. Zachodzi ono także dla operatora logicznego AND. Zauważ, że w przedstawionym kodzie nawiasy nie są niezbędne — operatory logiczne mają wyższy priorytet od operatorów relacyjnych. Jednak początkującym łatwiej jest zrozumieć kod, gdy podwyrażenia zostaną dla przejrzystości umieszczone w nawiasach.

Operator AND (&&)

Logiczny operator AND, &&, zwraca wartość true, jeśli oba operandy mają tę wartość. Jeśli którykolwiek z operandów ma wartość false, zwracany jest wynik false. Listing 4.33 wyświetla komunikat, jeśli podana zmienna jest większa od 10 i mniejsza od 24³. Operator AND (podobnie jak operator OR) nie zawsze przetwarza prawą stronę wyrażenia. Jeśli lewy operand ma wartość false, wynik będzie równy false niezależnie od wartości prawego operandu. Dlatego w takiej sytuacji środowisko uruchomieniowe pomija określanie wartości prawego operandu.

Listing 4.33. Używanie operatora AND

```
if ((10 < hourOfDay) && (hourOfDay < 24))
    System.Console.WriteLine(
        "Hej-ho, hej-ho, do pracy by się szło.");
```

Operator XOR (^)

Symbol karetki, ^, to operator XOR. Jeśli jest stosowany do dwóch operandów logicznych, zwraca wartość true tylko wtedy, gdy dokładnie jeden z operandów ma tę wartość. Działanie tego operatora przedstawiono w tabeli 4.3.

Tabela 4.3. Wartości zwracane przez operator XOR

Lewy operand	Prawy operand	Wynik
True	True	False
True	False	True
False	True	True
False	False	False

³ To typowe godziny pracy programistów.

Operator XOR (w odróżnieniu od operatorów logicznych AND i OR) nie stosuje przetwarzania skróconego. Zawsze sprawdza oba operandy, ponieważ nie może wyznaczyć wyniku, jeśli nie są znane wartości obu operandów. Zauważ, że operator XOR działa tak samo jak logiczny operator nierówności.

Logiczny operator negacji (!)

Logiczny operator negacji (operator NOT), !, odwraca wartość typu bool. Jest to operator jednoargumentowy, co oznacza, że wymaga tylko jednego operandu. Działanie tego operatora pokazano na listingu 4.34, a wyniki przedstawiono w danych wyjściowych 4.16.

Listing 4.34. Korzystanie z logicznego operatora negacji

```
bool valid = false;
bool result = !valid;
// Wyświetla "result = True".
System.Console.WriteLine($"result = { result }");
```

DANE WYJŚCIOWE 4.16.

```
result = True
```

Na początku listingu 4.34 zmienna `valid` jest ustawiana na wartość `false`. Następnie używany jest operator negacji do zmiennej `valid`, a wynik zostaje przypisany do zmiennej `result`.

Operator warunkowy (?:)

Zamiast instrukcji `if-else` służącej do wyboru jednej z dwóch wartości można zastosować **operator warunkowy**. Taki operator składa się ze znaku zapytania i dwukropka. Oto jego ogólna postać:

```
warunek ? wyrażenie_dla_true : wyrażenie_dla_false
```

Operator warunkowy jest operatorem trójargumentowym, ponieważ przyjmuje trzy operandy: `warunek`, `wyrażenie_dla_true` i `wyrażenie_dla_false`. Ponieważ jest to jedyny taki operator w języku C#, czasem nazywa się go *operatorem trójargumentowym*. Jednak bardziej zrozumiałe jest stosowanie nazwy operatora zamiast liczby przyjmowanych przez niego operandów. Operator warunkowy, podobnie jak niektóre inne operatory logiczne, stosuje przetwarzanie skrócone. Jeśli `warunek` ma wartość `true`, operator warunkowy przetwarza tylko człon `wyrażenie_dla_true`. Jeżeli `warunek` ma wartość `false`, przetwarzany jest tylko człon `wyrażenie_dla_false`. Wynikiem działania operatora jest wartość przetworzonego wyrażenia.

Na listingu 4.35 pokazano, jak stosować operator warunkowy. Kompletny listing programu znajdziesz w pliku `Chapter04\TicTacToe.cs` w kodzie źródłowym.

Listing 4.35. Operator warunkowy

```
class TicTacToe
{
    static string Main()
    {
```

```
// Początkowo zmienią currentPlayer ma wartość reprezentującą gracza nr 1.
int currentPlayer = 1;

// ...

for (int turn = 1; turn <= 10; turn++)
{
    // ...

    // Zmiana gracza.
    currentPlayer = (currentPlayer == 2) ? 1 : 2;
}

}
```

Ten kod zmienia gracza wykonującego posunięcie. W tym celu kod sprawdza, czy aktualna wartość to 2. Jest to *warunkowa* część wyrażenia. Jeśli warunek ma wartość true, operator ustawia „wartość dla true”, czyli 1. W przeciwnym razie wynik to „wartość dla false”, czyli 2. Wynik operatora warunkowego (inaczej niż w instrukcji if) musi zostać przypisany lub przekazany jako parametr. Operator warunkowy nie może być niezależną instrukcją.

Wskazówka

ROZWAŻ stosowanie instrukcji if-else zamiast nadmiernie skomplikowanych wyrażeń warunkowych.

Język C# wymaga, by wyrażenia dla true i dla false w operatorze warunkowym miały spójny typ. Powinien być on możliwy do określenia bez sprawdzania kontekstu wyrażenia. Kod f ? "abc" : 123 nie jest poprawnym wyrażeniem warunkowym, ponieważ wyrażenia dla true i dla false to łańcuch znaków i liczba, a konwersja między nimi nie jest możliwa. Nawet jeśli wpiszesz kod object result = f ? "abc" : 123;, kompilator języka C# oznaczy wyrażenie jako nieprawidłowe, ponieważ typ spójny z oboma wyrażeniami (czyli object) znajduje się poza samym wyrażeniem warunkowym.

Programowanie z użyciem wartości null

W rozdziale 3. opisano, że null jest bardzo przydatną wartością, jednak powoduje też problemy — mianowicie przed wywołaniem składowej obiektu trzeba sprawdzać, czy jest on różny od null. Można też zmieniać null na wartość bardziej odpowiednią w danym kontekście.

Choć można sprawdzać wartość null za pomocą operatorów równości (a nawet relacyjnych operatorów równości), istnieją też inne techniki, w tym rozbudowany operator is z C# 7.0. Ponadto istnieją operatory zaprojektowane specjalne do pracy z wartościami, które mogą być równe null. Są to operatory ?? (i ??= w wersji C# 8.0) oraz ?... Istnieje nawet operator do informowania kompilatora, że programista uważa, iż dana wartość jest różna od null (ale nie jest to oczywiste dla kompilatora). Jest to operator deklaracji wartości różnej od null (ang. *null-forgiving*; jest to operator ! podawany po zmiennej lub wyrażeniu). Najpierw przyjrzyj się sprawdzaniu, czy wartość jest równa null.

Sprawdzanie wartości null i wartości różnych od null

Istnieje wiele sposobów sprawdzania wartości null (zobacz tabelę 4.4).

 Początek
7.0

 Początek
8.0

Tabela 4.4. Sprawdzanie wartości null

Opis	Przykład
<p>Operatory równości i nierówności</p> <p>Operatory równości i nierówności działają we wszystkich wersjach języka C#.</p> <p>Dodatkowo sprawdzanie wartości null w ten sposób jest bardzo czytelne.</p> <p>Operatory równości i nierówności można przesłaniać, co może skutkować niewielkim spadkiem wydajności.</p>	<pre>string? uri = null; // ... if (uri != null) { System.Console.WriteLine(\$"uri jest równe: { uri }"); } else // (uri == null) { System.Console.WriteLine("uri jest równe null"); }</pre>
<p>ReferenceEquals()</p> <p>Choć <code>object.ReferenceEquals()</code> to dość długi zapis jak na prostą operację, ta składnia (podobnie jak operatory równości i nierówności) działa we wszystkich wersjach języka C# i ma tę zaletę, że nie umożliwia przesłaniania. Dlatego zawsze działa tak, jak wskazuje na to nazwa.</p>	<pre>string? uri = null; // ... if (object.ReferenceEquals(uri, null)) { System.Console.WriteLine("uri jest równe null"); }</pre>
<p>Operator dopasowywania wzorca — <code>is null</code></p> <p>Operator dopasowywania wzorca <code>is</code> umożliwia wykrywanie wartości null poprzez sprawdzanie, czy operand (<code>uri</code>) jest obiektem.</p> <p>W C# 7.0 mechanizm ten został rozszerzony o składnię <code>is null</code> do sprawdzania odwrotnej sytuacji, czyli tego, czy wartość jest równa null.</p>	<pre>if (uri is object) { System.Console.WriteLine(\$"uri jest równe: { uri }"); } else // (uri równe null) { System.Console.WriteLine("uri jest równe null"); }</pre>
<p>Dopasowywanie właściwości — <code>is {}</code></p> <p>Ostatnia możliwość, dostępna od wersji C# 8.0, to zastosowanie składni dopasowania właściwości w celu sprawdzenia, czy operand jest różny od null.</p>	<pre>if (uri is {}) { System.Console.WriteLine(\$"uri jest równe: { uri }"); } else { System.Console.WriteLine("uri jest równe null"); }</pre>

Ponieważ dostępnych jest kilka sposobów sprawdzania wartości null, warto się zastanowić, z której techniki należy korzystać. Jeśli programujesz z użyciem C# 6.0 lub starszej wersji tego języka, operatory równości i nierówności są jedynym rozwiązaniem (oprócz używania składni `is object` do wykrywania wartości różnych od null). W C# 7.0 dostępny jest rozszerzony operator `is`, dlatego można używać składni `is null` do wykrywania wartości null. To podejście jest zalecane, ponieważ nie można zmienić działania operatora `is`, nie trzeba więc martwić się o spadek wydajności. W C# 8.0 dopasowywanie właściwości za pomocą składni `is { }` pozwala sprawdzać wartość null, ale — w odróżnieniu od składni `is object` — nie zgłasza ostrzeżenia, gdy sprawdzanie dotyczy typów niedopuszczających wartości null.

Oto podsumowanie: od wersji C# 7.0 używaj składni `is object` do wykrywania wartości różnych od null i `is null` do wykrywania wartości null. W starszych wersjach używaj zapisu `Object.ReferenceEquals(<obiekt>, null)`, aby zagwarantować oczekiwane działanie kodu, lub składni `== null`, jeśli operator nie jest przeciążany i chcesz poprawić czytelność kodu.

W wierszach 2. i 3. tabeli 4.4 zastosowane jest dopasowywanie do wzorca. Zagadnienie to jest opisane szczegółowo w rozdziale 7.

Koniec
7.0

Operatory ?? i ??=

Operator ?? (ang. *null-coalescing operator*) pozwala w zwięzły sposób zapisać „jeśli dana wartość to null, zastosuj inną wartość”. Postać tego operatora to:

wyrażenie1 ?? wyrażenie2

Dla operatora ?? stosowane jest przetwarzanie skrócone. Jeśli *wyrażenie1* ma wartość różną od null, ta wartość jest zwracana jako wynik operacji. Nie trzeba wtedy przetwarzać drugiego wyrażenia. Jeżeli jednak *wyrażenie1* ma wartość null, operator zwraca wartość *wyrażenie2*. Operator ??, w odróżnieniu od operatora warunkowego, jest dwuargumentowy.

Sposób korzystania z operatora ?? przedstawiono na listingu 4.36.

Listing 4.36. Operator ??

```
string? fullName = GetDefaultDirectory();
// ...

// Operator ??
string fileName = GetFileName() ?? "config.json";
string directory = GetConfigurationDirectory() ??
    GetApplicationDirectory() ??
    System.Environment.CurrentDirectory;

// Operator ??=
fullName ??= $"{ directory }/{ fileName }";
// ...
```

Na tym listingu operator ?? służy do ustawiania zmiennej `fileName` na wartość "config.json", jeśli wywołanie `GetFileName()` zwraca null. Jeżeli `GetFileName()` zwraca wartość różną od null, zwrocona wartość jest przypisywana do zmiennej `fileName`.

Operator ?? dobrze sprawdza się w łańcuchach operacji. Na przykład wyrażenie w formie `x ?? y ?? z` zwraca x, jeśli x jest różne od null. W przeciwnym razie zwraca y, jeśli y nie jest równe null. Jeżeli i x, i y mają wartość null, zwracane jest z. Kod analizuje więc wyrażenia od lewej do prawej i wybiera pierwsze wyrażenie różne od null lub zwraca ostatnie wyrażenie, jeśli wszystkie wcześniejsze są równe null. Przykładem jest przypisywanie wartości do zmiennej `directory` na listingu 4.36.

W C# 8.0 wprowadzono połączenie operatora ?? z operatorem przypisania — **operator ??=**. Pozwala on sprawdzić, czy wartość po lewej stronie jest równa null, i jeśli tak jest, przypisać do zmiennej wartość podaną po prawej stronie. Na listingu 4.36 ten operator jest używany do przypisywania wartości do zmiennej `fullName`.

Operatory ?. i ?[]

Początek
6.0

Ponieważ wzorzec polegający na sprawdzaniu wartości null przed wywołaniem składowej jest stosowany tak często, w wersji C# 6.0 wprowadzono **operator ?.** (ang. *null-conditional operator*) przedstawiony na listingu 4.37.

Listing 4.37. Operator ?.

```
string[]? segments = null;
// ...

int? length = segments?.Length;
if (length is object && length != 0){
    uri = string.Join('/', segments!);
}

// Operator ? i akcesor tablicy, gdy wiadomo, że
// tablica zawiera przynajmniej jeden element:
// uri = segments?[0];

if (uri is null || length is 0){
    System.Console.WriteLine(
        "Brak elementów do połączenia.");
}
else
{
    System.Console.WriteLine(
        $"Uri: { uri }");
}
```

Operator ?. sprawdza, czy operand (segments na listingu 4.37) ma wartość null. Dopiero potem wywołuje metodę lub właściwość (tu jest to Length). Poniżej pokazano logiczny odpowiednik kodu z listingu (przy czym w składni z wersji C# 6.0 wartość zmiennej segments jest określana tylko raz).

```
int? length =
(segments != null) ? (int?)segments.Length : null
```

8.0

Ważną cechą operatora `?.` jest to, że zawsze zwraca wartość dopuszczającą null. W tym przykładzie składowa `string.Length` zwraca wartość typu `int`, który nie dopuszcza wartości `null`, jednak wywołanie `Length` razem z operatorem `?.` daje wartość typu `int` dopuszczającągo `null` (`int?`).

Operator `?.` możesz stosować razem z akcesorem tablic. Na przykład wywołanie `segments?[0]` zwraca pierwszy element tablicy `segments`, jeśli nie jest ona równa `null`. Jednak ta technika jest stosowana dość rzadko, ponieważ przydaje się tylko wtedy, gdy nie wiesz, czy operand jest równy `null`, ale znasz liczbę elementów (a przynajmniej wiesz, że dany element istnieje).

Operator `?.` jest wygodny zwłaszcza dzięki temu, że można go stosować w łańcuchach operacji (razem z operatorami `??` lub bez nich). Na przykład w poniższym kodzie metody `ToLower()` i `StartsWith()` są wywoływanie tylko wtedy, gdy zarówno `segments`, jak i `segments[0]` są różne od `null`:

```
segments?[0]??.ToLower().StartsWith("file:");
```

W tym przykładzie założono, że elementy tablicy `segments` mogą być równe `null`. Dlatego bardziej precyzyjna deklaracja (w wersji C# 8.0) powinna wyglądać tak:

```
string?[]? segments;
```

To oznacza, że tablica `segments` może być równa `null`, a ponadto każdy z jej elementów to łańcuch znaków, który też może mieć wartość `null`.

Gdyłączysz wyrażenia zawierające operator `?.` w łańcuchu, a pierwszy operand ma wartość `null`, stosowane jest przetwarzanie skrócone i dalsze wywołania w łańcuchu nie są uruchamiane. Możesz też umieścić operator `??` na końcu wyrażenia, co pozwala podać wartość domyślną używaną, gdy operand jest równy `null`:

```
string uri = segments?[0]??.ToLower().Trim() ?? "intellitect.com";
```

Zauważ, że typ danych zwracany przez operator `??` nie dopuszcza wartości `null` (przy założeniu, że wartość po prawej stronie operatora — tu `"intellitect.com"` — jest różna od `null`; stosowanie w tym miejscu `null` miałoby niewiele sensu).

Uważaj jednak, by nie zapomnieć przypadkowo o dodatkowych wartościach `null`. Pomyśl na przykład, co by się stało w hipotetycznej sytuacji, gdyby instrukcja `ToLower()` także mogła zwrócić wartość `null`. W takim scenariuszu po wywołaniu `Trim()` zgłoszony zostanie wyjątek `NullReferenceException`. Nie oznacza to, że trzeba używać łańcucha operatorów `?.`, natomiast należy zwracać uwagę na logikę działania kodu. W przedstawionym przykładzie wywołanie `ToLower()` nigdy nie może zwrócić wartości `null`, dlatego dodatkowy operator `?.` nie jest potrzebny.

Choć jest to dość osobliwe działanie (w porównaniu do funkcjonowania innych operatorów), wartość typu dopuszczającego `null` jest zwracana tylko na końcu łańcucha wywołań. Dlatego użycie operatora kropki `(.)` po składowej `Length` spowoduje, że następne wywołanie będzie dotyczyło wartości typu `int` (nie `int?`). Jednak umieszczenie członu `segments?.Length` w nawiasie (co wymusza wygenerowanie wyniku typu `int?` z powodu wysokiego priorytetu nawiasów) spowoduje zwrócenie wartości typu `int?`, dzięki czemu dostępne będą składowe specyficzne dla typów `Nullable<T>` (`HasValue` i `Value`).

6.0

Operator deklaracji wartości różnej od null (!)

Zauważ, że na listingu 4.37 w wywołaniu `Join()` po słowie `segments` znajduje się wykrywki:

```
uri = string.Join('/', segments!);
```

Wcześniej w kodzie wartość wywołania `segments.Length` jest przypisywana do zmiennej `length`, dzięki czemu wiadomo, że zmienna jest różna od `null`. Ponadto instrukcja `if` sprawdza, czy zmienna `segments` jest różna od `null`, badając, czy jej długość (`length`) jest różna od zera.

```
if (length is object && length != 0) { }
```

Jednak kompilator mógłby sam to sprawdzać. Ponieważ wywołanie `Join()` wymaga tablicy łańcuchów znaków niedopuszczającej wartości `null`, zgłosi ostrzeżenie, gdy przekażesz zwykłą zmienną `segments`, której deklaracja dopuszcza wartości `null`. Aby uniknąć tego ostrzeżenia, od wersji C# 8.0 możesz zastosować **operator deklaracji wartości różnej od null** (!). Jest to informacja dla kompilatora, że programista lepiej zna kod i wie, że zmienna `segments` jest różna od `null`. Wtedy w czasie komplikacji kompilator przyjmie, że programista ma większą wiedzę, i pominie ostrzeżenie (choć środowisko uruchomieniowe nadal sprawdzi, czy asercja jest różna od `null`).

Niestety, jest to naiwne, a nawet niebezpieczne rozwiązanie, ponieważ operator `?` daje fałszywe poczucie bezpieczeństwa. Pozornie wydaje się, że jeśli tablica `segments` ma wartość różną od `null`, dany element musi istnieć. Jest to oczywiście błędne założenie. Element może nie istnieć nawet wtedy, jeśli tablica `segments` nie jest równa `null`.

6.0

Koniec
8.0

ZAGADNIENIE DLA ZAAWANSOWANYCH

Stosowanie operatora `?. razem z delegatami`

Operator `?. sam` w sobie jest bardzo wartościowym mechanizmem. Jednak w połączeniu z wywołaniami delegatów rozwiązuje problem, który istniał w języku C# od wersji 1.0. Zauważ, że w przedstawionym na listingu 4.38 kodzie metoda obsługi zdarzenia `PropertyChanged` jest wiązana z jej lokalną kopią (`PropertyChanged`) przed sprawdzeniem, czy przypisywana wartość to `null`, i zgłoszeniem zdarzenia. Jest to najłatwiejszy bezpieczny ze względu na wątki sposób wywoływanego zdarzeń bez narażania się na ryzyko tego, że między momentem sprawdzenia wartości `null` a zgłoszeniem zdarzenia nastąpi zerwanie subskrypcji zdarzenia. Niestety, to podejście jest nieintuicyjne i programiści często stosują inne rozwiązania, co grozi zgłoszeniem wyjątku `NullReferenceException`. Na szczęście wraz z wprowadzeniem operatora `?. w wersji C# 6.0` problem ten został rozwiązany.

Listing 4.38. Operator `?. i obsługa zdarzeń`

```
PropertyChangedEventHandler PropertyChanged =
{
    PropertyChanged;
    if (PropertyChanged != null)
    {
        PropertyChanged(this,
            new PropertyChangedEventArgs(nameof(Name)));
    }
}
```

W wersji C# 6.0 zmiany wartości delegata można wykrywać za pomocą kodu prostszego niż technika pokazana na listingu 4.38:

```
PropertyChanged?.Invoke(propertyChanged,
    this, new PropertyChangedEventArgs(nameof(Name)));
```

Koniec
6.0

Ponieważ zdarzenie jest obsługiwane za pomocą delegata, zawsze można zastosować wzorzec polegający na wywoływaniu delegata z wykorzystaniem operatora `?.` i instrukcji `Invoke()`.

Operatory bitowe (<<, >>, |, &, ^, ~)

Operatory bitowe to dodatkowy zestaw operatorów występujących w prawie wszystkich językach programowania. Te operatory służą do manipulowania wartościami w formacie binarnym.

ZAGADNIENIE DLA POCZĄTKUJĄCYCH

Bity i bajty

Wszystkie wartości w komputerze są reprezentowane w formacie binarnym za pomocą zer i jedynek, nazywanych **cyframi binarnymi** (są to **bity**). Bity są łączone w ośmioelementowe grupy nazywane **bajtami**. W bajcie każdy kolejny bit reprezentuje wartość liczby 2 podniesionej do odpowiedniej potęgi (od 2^0 po prawej stronie do 2^7 po stronie lewej). Przedstawia to rysunek 4.1.

0	0	0	0	0	0	0	0
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

Rysunek 4.1. Wartości odpowiadające kolejnym pozycjom

W wielu sytuacjach, zwłaszcza przy korzystaniu z usług niskopoziomowych lub systemowych, informacje są pobierane jako dane binarne. Aby manipulować urządzeniami i usługami używającymi formatu binarnego, trzeba przetwarzać dane w tym formacie.

Na rysunku 4.2 każda komórka reprezentuje wartość liczby 2 podniesionej do podanej potęgi. Wartość bajta (liczby 8-bitowej) to suma potęg liczby 2 dla wszystkich bitów ustalonych na wartość 1.

0	0	0	0	0	1	1	1
$7 = 4 + 2 + 1$							

Rysunek 4.2. Obliczanie wartości bajta bez znaku

Przetwarzanie wartości binarnych dla liczb ze znakiem przebiega inaczej. Liczby ze znakiem (typu `long`, `short` i `int`) są reprezentowane za pomocą kodu **uzupełnień do dwóch**.

Dzięki tej metodzie dodawanie działa także wtedy, gdy liczba ujemna jest dodawana do dodatnich (operacja odbywa się tak, jakby oba operandy były dodatnie). W tej notacji liczby ujemne są traktowane inaczej niż dodatnie. Liczby ujemne to te, które na pozycji pierwszej od lewej mają wartość 1. Dla takich liczb dodawane są wartości odpowiadające pozycjom, na których występuje 0, a nie wartości odpowiadające komórkom zawierającym 1. Każda lokalizacja odpowiada wtedy ujemnej wartości liczby 2 podniesionej do odpowiedniej potęgi. Ponadto od wyniku trzeba odjąć 1. Cały proces pokazano na rysunku 4.3.

1	1	1	1	1	0	0	1
					-7	= -4	-2 + 0 - 1

Rysunek 4.3. Obliczanie wartości dla bajtów reprezentujących liczby ze znakiem

Tak więc bity 1111 1111 1111 1111 reprezentują -1, a 1111 1111 1111 1001 to -7. Binarna liczba 1000 0000 0000 0000 to najmniejsza ujemna wartość, jaką można zapisać w 16-bitowym typie całkowitoliczbowym.

Operatory przesunięcia (`<<`, `>>`, `<<=`, `>>=`)

Czasem programista chce przesunąć wartość binarną w prawo lub w lewo. Przy przesuwaniu w lewo wszystkie bity w binarnej reprezentacji liczby są przenoszone w lewo o liczbę pozycji podaną w operandzie występującym po prawej stronie operatora przesunięcia. Komórki występujące po prawej stronie liczby binarnej są wtedy uzupełniane zerami. Operator przesunięcia w prawo działa prawie tak samo, ale w odwrotną stronę. Jeśli jednak przesunięcie dotyczy liczby ujemnej, lewa strona liczby binarnej jest uzupełniana jedynkami, a nie zerami. Operatory przesunięcia to `>>` (przesunięcie w prawo) i `<<` (przesunięcie w lewo). Ponadto dostępne są operatory łączące przesunięcie z przypisaniem (`<<=` i `>>=`).

Przyjrzyj się teraz przykładowi. Założmy, że używana jest wartość typu int równa -7. Jej binarna reprezentacja to 1111 1111 1111 1111 1111 1111 1111 1111 1001. Kod z listingu 4.39 przesuwa w prawo binarną reprezentację liczby -7 o dwie pozycje.

Listing 4.39. Używanie operatora przesunięcia w prawo

```
int x;  
x = (-7 >> 2); // 11111111111111111111111111111101 zmienia się w  
// 11111111111111111111111111111110  
// Wyświetla "x = -2."  
System.Console.WriteLine($"x = { x }.");
```

Wynik działania kodu z listingu 4.39 pokazano w danych wyjściowych 4.17.

DANE WYJŚCIOWE 4.17.

$$x = -2.$$

Z powodu przesunięcia w prawo wartości bitów z pozycji występujących po prawej stronie są „wypychane za krawędź”, a bit oznaczający wartość ujemną (występujący po lewej) jest przesuwany o dwie pozycje zastępowane jedynkami. Wynik to -2.

Choć według legend operacja $x \ll 2$ działa szybciej niż $x * 4$, nie należy stosować operatorów przesunięcia bitów do wykonywania mnożenia lub dzielenia. Różnice w wydajności mogły występować w niektórych kompilatorach języka C w latach 70. ubiegłego wieku, jednak nowoczesne kompilatory i mikroprocesory świetnie sobie radzą z optymalizacją operacji arytmetycznych. Stosowanie przesunięcia bitów do mnożenia lub dzielenia liczb jest mylące i często prowadzi do błędów, jeśli osoba odpowiedzialna za konserwację kodu zapomni, że operatory przesunięcia mają niższy priorytet niż operatory arytmetyczne.

Operatory bitowe (&, |, ^)

W pewnych sytuacjach potrzebne są operacje logiczne (na przykład AND, OR lub XOR) wykonywane bit po bicie na dwóch operandach. Służą do tego operatory &, | i ^.

ZAGADNIENIE DLA POCZĄTKUJĄCYCH

Objaśnienie operatorów logicznych

Jeśli występują dwie liczby, tak jak na rysunku 4.4, w operacjach bitowych porównywane są wartości komórek, począwszy od lewej najbardziej znaczącej pozycji aż do końca. Wartość 1 w komórce jest traktowana jak true, a wartość 0 jak false.

12:	<table border="1" style="border-collapse: collapse; width: 100%; text-align: center;"> <tr><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td></tr> </table>	0	0	0	0	1	1	0	0
0	0	0	0	1	1	0	0		
7:	<table border="1" style="border-collapse: collapse; width: 100%; text-align: center;"> <tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td></tr> </table>	0	0	0	0	0	1	1	1
0	0	0	0	0	1	1	1		

Rysunek 4.4. Liczby 12 i 7 przedstawione w postaci binarnej

Bitowa operacja AND na dwóch wartościach z rysunku 4.4 powoduje porównanie po kolejnych bitów pierwszego operandu (12) z bitami drugiego operandu (7). W efekcie uzyskana zostanie wartość binarna 00000100, czyli 4. Bitowa operacja OR na dwóch przykładowych wartościach da wynik 00001111, czyli 15. Wynikiem operacji XOR będzie 00001011, czyli 11 w systemie dziesiętnym.

Na listingu 4.40 pokazano, jak używać operatorów bitowych. Wyniki działania kodu z listingu 4.40 pokazano w danych wyjściowych 4.18.

Listing 4.40. Używanie operatorów bitowych

```
byte and, or, xor;
and = 12 & 7; // and = 4
or = 12 | 7; // or = 15
xor = 12 ^ 7; // xor = 11
System.Console.WriteLine(
    $"and = { and } \nor = { or } \nxor = { xor }");
```

DANE WYJŚCIOWE 4.18.

and = 4
or = 15
xor = 11

Na listingu 4.40 wartość 7 jest **maską** — służy do ustawienia lub wyeliminowania określonych bitów z pierwszego operandu za pomocą specjalnego wyrażenia. Zauważ, że operator & (w odróżnieniu od operatora AND &&), zawsze przetwarza *obie* strony wyrażenia, nawet jeśli po lewej stronie występuje wartość false. Podobnie w wersji | operatora OR nie jest stosowane przetwarzanie skrócone. Operator zawsze sprawdza wartość obu operandów, nawet jeśli po lewej stronie występuje wartość true. W wersjach bitowych operatorów AND i OR nie stosuje się więc przetwarzania skróconego.

Aby przekształcić liczbę na jej reprezentację binarną, trzeba sprawdzić każdy bit liczby. Listing 4.41 to przykładowy program, który przekształcała łańcuch znaków na łańcuch znaków z reprezentacją binarną danej wartości. Wynik działania kodu z listingu 4.41 pokazano w danych wyjściowych 4.19.

Listing 4.41. Pobieranie łańcucha znaków z binarną reprezentacją liczby

```
class BinaryConverter
{
    static void Main()
    {
        const int size = 64;
        ulong value;
        char bit;

        System.Console.Write ("Wprowadź liczbę całkowitą: ");
        // Wywołanie long.Parse() pozwala dodać obsługę liczb ujemnych.
        // Kod bez sprawdzania przypisuje wartość do zmiennej typu ulong.
        value = (ulong)long.Parse (System.Console.ReadLine ());

        // Ustawianie początkowej maski na 100.
        ulong mask = 1UL << size - 1;
        for (int count = 0; count < size; count++)
        {
            bit = ((mask & value) != 0) ? '1': '0';
            System.Console.Write(bit);
            // Przesunięcie maski o jedną pozycję w prawo.
            mask >>= 1;
        }
        System.Console.WriteLine();
    }
}
```

DANE WYJŚCIOWE 4.19.

W każdym powtórzeniu pętli for z listingu 4.41 (pętle są opisane w dalszej części rozdziału) używany jest operator przesunięcia w prawo, co pozwala utworzyć maskę dla każdego bitu z wartości value. Dzięki zastosowaniu operatora bitowego & do zamaskowania wybranych bitów można ustalić, czy dany bit jest ustawiony. Jeśli sprawdzenie maski daje wynik niezerowy, w konsoli wyświetlna jest wartość 1. W przeciwnym razie kod wyświetla 0. W ten sposób tworzone są dane wyjściowe reprezentujące wartość typu ulong w zapisie binarnym.

Zauważ, że nawias w instrukcji `(mask & value) != 0` jest niezbędny. Jest tak, ponieważ operator nierówności ma wyższy priorytet niż operator AND. Bez nawiasu wyrażenie zostałoby zinterpretowane jako `mask & (value != 0)`, co nie ma żadnego sensu, ponieważ lewa strona operatora & to wtedy wartość typu ulong, a prawa to wartość typu bool.

Ten przykład został przedstawiony wyłącznie w celach edukacyjnych. Dostępna jest wbudowana metoda środowiska CLR, `System.Convert.ToString(value, 2)`, która przeprowadza opisaną tu konwersję. Drugi argument tej metody określa podstawę (2 dla liczb binarnych, 10 dla liczb dziesiętnych lub 16 dla liczb szesnastkowych), co pozwala przeprowadzać konwersję na formaty inne niż binarny.

Bitowe złożone operatory przypisania (`&=`, `|=`, `^=`)

Nie jest zaskoczeniem, że można połączyć operatory bitowe z operatorem przypisania. W efekcie powstają operatory `&=`, `|=` i `^=`. Pozwalają one na przykład wykonać operację OR na zmiennej i liczbie, a następnie przypisać wynik do pierwotnej zmiennej, co pokazano na listingu 4.42.

Listing 4.42. Używanie operatorów logicznych z operatorem przypisania

```
byte and = 12, or = 12, xor = 12;
and &= 7; // and = 4
or |= 7; // or = 15
xor ^= 7; // xor = 11
System.Console.WriteLine(
    $"and = { and } \nor = { or }\nxor = { xor }");
```

Wynik działania kodu z listingu 4.42 pokazano w danych wyjściowych 4.20.

DANE WYJŚCIOWE 4.20.

```
and = 4
or = 15
xor = 11
```

Połączenie mapy bitowej z maską w instrukcji takiej jak `fields &= mask` powoduje wyzerowanie w zmiennej fields bitów, które nie są ustawione w masce mask. Odwrotna operacja, `fields &= ~mask`, prowadzi do wyzerowania w zmiennej fields bitów ustawionych w masce.

Bitowy operator dopełnienia (~)

Bitowy operator dopełnienia oblicza dopełnienie każdego bitu z operandu, przy czym operandem może być wartość typu `int`, `uint`, `long` lub `ulong`. Wyrażenie `~1` zwraca wartość binarną `1111 1111 1111 1111 1111 1111 1111 1110`, a wyrażenie `~(1<<31)` zwraca liczbę binarną `0111 1111 1111 1111 1111 1111 1111 1111`.

Instrukcje związane z przepływem sterowania — ciąg dalszy

Po szczegółowym opisaniu wyrażeń logicznych można bardziej precyzyjnie wyjaśnić instrukcje związane z przepływem sterowania dostępne w języku C#. Wiele spośród tych instrukcji będzie znanych doświadczeniom programistom, dlatego jeśli interesują Cię tylko szczegółowe informacje specyficzne dla języka C#, możesz pominąć ten podrozdział. Zwróć jednak uwagę na pętlę `foreach`, ponieważ dla wielu programistów może się ona okazać czymś nowym.

Pętle while i do/while

Do tego miejsca nauczyłeś się pisać programy, które wykonują określone operacje tylko raz. Jednak programy mogą łatwo wykonywać podobne operacje wiele razy. W tym celu należy utworzyć pętlę z instrukcjami. Pierwsza z omawianych tu pętli to `while`. Jest to najprostsza pętla warunkowa. Ogólna postać pętli `while` wygląda tak:

```
while (warunek)
    instrukcja
```

Komputer wielokrotnie wykonuje instrukcję, która stanowi ciało pętli, dopóki warunek (musi nim być wyrażenie logiczne) ma wartość `true`. Gdy warunek ma wartość `false`, kod przechodzi do instrukcji za pętlą. Zauważ, że *instrukcja* jest wykonywana nawet wtedy, gdy spowoduje, że *warunek* przyjmie wartość `false`. Zakończenie pracy pętli ma miejsce dopiero po ponownym sprawdzeniu wartości warunku na początku pętli. Działanie pętli `while` ilustruje kalkulator liczb Fibonacciego przedstawiony na listingu 4.43.

Listing 4.43. Przykładowa pętla while

```
class FibonacciCalculator
{
    static void Main()
    {
        decimal current;
        decimal previous;
        decimal temp;
        decimal input;

        System.Console.Write("Wprowadź dodatnią liczbę całkowitą:");

        // Instrukcja decimal.Parse przekształca wartość pobraną przez metodę ReadLine na liczbę dziesiętną.
    }
}
```

```

input = decimal.Parse(System.Console.ReadLine());

// Inicjowanie zmiennych current i previous wartością 1. Są to dwie
// pierwsze liczby z ciągu Fibonacciego.
current = previous = 1;

// Dopóki obecna liczba Fibonacciego w ciągu ma
// wartość mniejszą niż liczba podana przez użytkownika.
while (current <= input)
{
    temp = current;
    current = previous + current;
    previous = temp; // Wykonywane nawet w sytuacji, gdy poprzednia
    // instrukcja spowodowała, że zmienna current ma wartość większą niż zmienna input.

    System.Console.WriteLine(
        $"Następna liczba Fibonacciego to { current }");
}

```

Liczba Fibonacciego to element **ciągu Fibonacciego**, obejmującego wszystkie liczby będące sumą dwóch poprzednich wartości z ciągu. Pierwsze dwa elementy ciągu to 1 i 1. Kod z listingu 4.43 wyświetla prośbę o wpisanie liczby całkowitej przez użytkownika. Następnie pętla `while` jest używana do znalezienia pierwszej liczby Fibonacciego większej niż liczba podana przez użytkownika.

■ ZAGADNIENIE DLA POCZĄTKUJĄCYCH

Kiedy należy stosować pętlę `while`?

Dalej w rozdziale opisano inne instrukcje, które pozwalają wielokrotnie wykonywać blok kodu. Pojęcie **ciało pętli** oznacza instrukcję (często blok kodu) wykonywaną w instrukcji `while`. Kod jest wykonywany w pętli do czasu wystąpienia warunku wyjścia z niej. Ważne jest, by umieć ocenić, który rodzaj pętli wybrać. Pętla `while` służy do powtarzania kodu dopóty, dopóki warunek ma wartość `true`. Pętla `for` jest odpowiednia, gdy liczba powtórzeń jest znana (na przykład przy odliczaniu od 0 do n). Pętla `do/while` działa podobnie jak `while`, ale zawsze wykonuje kod z ciała pętli przynajmniej raz.

Pętla `do/while` działa bardzo podobnie jak `while`, przy czym jest zalecana w sytuacjach, gdy liczba powtórzeń powinna wynosić od 1 do n dla nieznanego początkowo n . Taka sytuacja często występuje, gdy kod pobiera dane wejściowe od użytkownika. Kod z listingu 4.44 pochodzi z programu do gry w kółko i krzyżyk.

Listing 4.44. Przykładowa pętla `do/while`

```

// Wielokrotne wyświetlanie prośby o wpisanie ruchu do
// momentu podania prawidłowej pozycji na planszy.
bool valid;
do
{

```

```

valid = false;

// Prośba o wpisanie ruchu przez gracza wykonującego posunięcie.
System.Console.WriteLine(
    $"\\nGracz {currentPlayer} – wprowadź posunięcie:");
input = System.Console.ReadLine();

// Sprawdzanie danych wprowadzonych przez gracza wykonującego posunięcie.
// ...

} while (!valid);

```

Kod z listingu 4.44 inicjuje zmienną `valid` wartością `false` na początku każdej **iteracji** (czyli powtórzenia pętli). Dalej kod wyświetla prośbę o wprowadzenie liczby przez użytkownika i pobiera dane. Choć ten fragment pominięto, program sprawdza, czy dane wejściowe są prawidłowe, a jeśli tak jest, przypisuje do zmiennej `valid` wartość `true`. Ponieważ w kodzie używana jest instrukcja `do/while`, a nie `while`, program wyświetla prośbę o podanie danych przynajmniej raz.

Ogólna postać pętli `do/while` wygląda następująco:

```

do
    instrukcja
    while (warunek);

```

Podobnie jak we wszystkich instrukcjach związanych z przepływem sterowania, tak i tu instrukcja jest w rzeczywistości blokiem kodu. Pozwala to wykonywać w ciele pętli zestaw instrukcji. Dozwolone jest jednak zastosowanie prawie dowolnej pojedynczej instrukcji (z wyjątkiem instrukcji z etykietą lub deklaracji zmiennej lokalnej).

Pętla `for`

Pętla `for` wykonuje blok kodu do czasu, w którym spełniony zostanie określony warunek. Bardzo przypomina pod tym względem pętlę `while`. Różnica polega na tym, że pętla `for` udogodnia wbudowaną składnię do inicjowania, inkrementacji i sprawdzania wartości licznika (jest on nazywany **zmienną pętli**). Ponieważ operacja inkrementacji ma określone miejsce w składni omawianej instrukcji, w pętlach `for` często stosuje się operatory inkrementacji i dekrementacji.

Na listingu 4.45 pokazano, jak wykorzystać pętlę `for` do wyświetlenia liczby całkowitej w postaci binarnej. Wynik działania kodu pokazano w danych wyjściowych 4.21.

Listing 4.45. Korzystanie z pętli `for`

```

class BinaryConverter
{
    static void Main()
    {
        const int size = 64;
        ulong value;
        char bit;

        System.Console.Write("Wprowadź liczbę całkowitą: ");

```

```
// Używanie instrukcji long.Parse(), aby zapewnić obsługę liczb ujemnych
// Kod bez sprawdzania przypisuje wartość do zmiennej typu ulong.
value = (ulong)long.Parse(System.Console.ReadLine());

// Ustawianie początkowej maski na 100.
ulong mask = 1UL << size - 1;
for (int count = 0; count < size; count++)
{
    bit = ((mask & value) > 0) ? '1': '0';
    System.Console.Write(bit);
    // Przesunięcie maski o jedną pozycję w prawo.
    mask >>= 1;
}
```

DANE WYJŚCIOWE 4.21.

Kod z listingu 4.45 stosuje maskę bitową 64 razy — jeden raz dla każdego bitu z liczby. Trzy części nagłówka pętli for odpowiadają za: zadeklarowanie i zainicjowanie zmiennej count, opisanie warunku, który musi być spełniony, by program wykonał kod z ciała pętli, i aktualizację licznika pętli. Ogólna postać petli for wygląda tak:

```
for (inicjowanie; warunek; inkrementacja)  
    instrukcja
```

Oto omówienie elementów pętli for:

- W sekcji *inicjowanie* wykonywane są operacje poprzedzające pierwszą iterację. Na listingu 4.45 kod deklaruje i inicjuje w tym miejscu zmienną count. Wyrażenie *inicjowanie* nie musi zawierać deklaracji nowej zmiennej, choć często taka deklaracja jest używana. Można na przykład wcześniej zadeklarować zmienną, a następnie zainicjować ją w pętli for. Można też zupełnie pominąć sekcję inicjowania i pozostawić ją pustą. Zadeklarowane w tym miejscu zmienne są dostępne w nagłówku i ciele pętli for.
 - W sekcji *warunek* pętli for określony jest warunek końcowy. Pętla kończy pracę, gdy warunek przyjmuje wartość false (podobnie działa pętla while). Pętla for wykonuje kod z ciała tylko wtedy, jeśli warunek ma wartość true. Na listingu 4.45 pętla kończy pracę, gdy zmienna count przyjmuje wartość 64 lub większą.
 - Wyrażenie *inkrementacja* jest wykonywane po każdej iteracji. Na listingu 4.45 instrukcja count++ zostaje wywołana po przesunięciu maski w prawo (`mask >>= 1`), ale przed sprawdzeniem warunku. W 64. iteracji zmienna count jest zwiększana do wartości 64, co powoduje, że warunek ma wartość false, a pętla kończy działanie.
 - Człon *instrukcja* w pętli for to kod ciała pętli, wykonywany dopóty, dopóki wyrażenie warunkowe ma wartość true.

Jeśli zapiszesz każdy krok wykonywania pętli `for` w pseudokodzie, nie używając samej pętli, efekt będzie wyglądał tak:

1. Deklarowanie zmiennej `count` i inicjowanie jej wartością 0.
2. Jeśli zmienna `count` ma wartość mniejszą niż 64, należy przejść do kroku 3.
W przeciwnym razie należy przejść do kroku 7.
3. Obliczanie wartości zmiennej `bit` i wyświetlanie jej.
4. Przesuwanie maski.
5. Inkrementacja zmiennej `count` o 1.
6. Przejście do kroku 2.
7. Kontynuowanie wykonywania programu od instrukcji po pętli.

Instrukcja `for` nie wymaga podawania w nagłówku żadnych instrukcji. Wyrażenie `for(;;){...}` jest prawidłowe, przy czym potrzebny jest sposób na wyjście z pętli, by nie była ona wykonywana w nieskończoność. Jeśli warunek nie jest podany, pętla domyślnie przyjmuje, że zawsze ma on wartość `true`.

Wyrażenia *inicjowanie* i *inkrementacja* mają nietypową składnię, co zapewnia obsługę pętli wymagających kilku liczników. Pętlę z kilkoma licznikami pokazano na listingu 4.46.

Listing 4.46. Pętla for z kilkoma wyrażeniami

```
for (int x = 0, y = 5; ((x <= 5) && (y >= 0)); y--, x++)
{
    System.Console.WriteLine(
        $"{{ x }}{ ((x > y) ? '>' : '<' }}}{{ y }}\t";
}
```

Wynik działania kodu z listingu 4.46 przedstawiono w danych wyjściowych 4.22.

DANE WYJŚCIOWE 4.22.

0<5	1<4	2<3	3>2	4>1	5>0
-----	-----	-----	-----	-----	-----

Tu w członie inicjującym znajduje się złożona deklaracja, w której deklarowane i inicjowane są dwa liczniki pętli. Instrukcja ta jest podobna do deklaracji kilku zmiennych lokalnych. Człon z inkrementacją jest zupełnie nietypowy, ponieważ zamiast pojedynczego wyrażenia może obejmować listę wyrażeń rozdzielonych przecinkami.

Wskazówka

ROZWAŻ refaktoryzację metody, gdy zauważysz, że piszesz pętlę `for` ze złożonymi warunkami i wieloma licznikami. Zadbaj o to, by przepływ sterowania był łatwiejszy do zrozumienia.

Pętla for jest tylko wygodniejszym sposobem na zapisanie pętli while. Zawsze możesz przekształcić pętlę for na następującą postać:

```
{
    inicjowanie;
    while (warunek)
    {
        instrukcja;
        inkrementacja;
    }
}
```

Wskazówki

STOSUJ pętlę for, gdy liczba iteracji pętli jest z góry znana, a licznik określający liczbę wykonywanych iteracji potrzebny w pętli.

STOSUJ pętlę while, gdy liczba iteracji nie jest z góry znana i licznik nie jest potrzebny w pętli.

Pętla foreach

Ostatni rodzaj pętli w języku C# to foreach. Pętla foreach przechodzi po kolekcji elementów i ustawia licznik pętli w taki sposób, by po kolej reprezentował każdy z tych elementów. W ciele pętli można wykonywać różne operacje na elementach. Wygodną cechą pętli foreach jest to, że każdy element jest pobierany dokładnie raz. Nie jest możliwe przypadkowe błędne policzenie elementów lub wyjście poza kolekcję, co może się zdarzyć w innych pętlach.

Ogólna postać instrukcji foreach wygląda następująco:

```
foreach (typ zmienna in kolekcja)
    instrukcja
```

Oto omówienie komponentów instrukcji foreach:

- Człon *typ* służy do deklarowania typu danych zmiennej, do której przypisywane są wszystkie kolejne elementy kolekcji. Można zastosować typ var, by kompilator ustalał typ na podstawie typu elementów z kolekcji.
- Człon *zmienna* określa przeznaczoną tylko do odczytu zmienną, do której pętla foreach automatycznie przypisuje kolejne elementy kolekcji. Zasięg tej zmiennej jest ograniczony do ciała pętli.
- Człon *kolekcja* to wyrażenie (na przykład nazwa tablicy) reprezentujące dowolną liczbę elementów.
- Człon *instrukcja* to ciało pętli wykonywane w każdej jej iteracji.

Przyjrzyj się teraz pętli foreach w kontekście prostego przykładu pokazanego na listingu 4.47.

Listing 4.47. Określanie możliwych posunięć za pomocą pętli foreach

```
class TicTacToe // Deklaracja klasy TicTacToe.  
{  
    static void Main() // Deklaracja punktu wejścia do programu.  
    {  
        // Zapisywanie na sztywno początkowego stanu planszy:  
        // ---+---+---  
        // 1 | 2 | 3  
        // ---+---+---  
        // 4 | 5 | 6  
        // ---+---+---  
        // 7 | 8 | 9  
        // ---+---+---  
        char[] cells = {  
            '1', '2', '3', '4', '5', '6', '7', '8', '9'  
        };  
  
        System.Console.Write(  
            "Możliwe posunięcia to: ");  
  
        // Wyświetlanie posunięć dopuszczalnych na początku.  
        foreach (char cell in cells)  
        {  
            if (cell != '0' && cell != 'X')  
            {  
                System.Console.Write($"{ cell } ");  
            }  
        }  
    }  
}
```

Wynik działania kodu z listingu 4.47 pokazano w danych wyjściowych 4.23.

DANE WYJŚCIOWE 4.23.

```
Możliwe posunięcia to: 1 2 3 4 5 6 7 8 9
```

Gdy silnik wykonawczy natrafi na przedstawioną instrukcję foreach, przypisze do zmiennej `cell` pierwszy element z tablicy `cells`. Tu jest nim wartość '1'. Następnie wykonany zostanie kod z bloku z ciałem pętli foreach. Instrukcja `if` określa, czy wartość zmiennej `cell` to '0' lub 'X'. Jeśli nie jest to żadna z tych liter, wartość zmiennej `cell` jest wyświetlana w konsoli. W następnej iteracji do zmiennej `cell` przypisywana jest kolejna wartość z tablicy itd.

Zauważ, że kompilator uniemożliwia modyfikowanie zmiennej (`cell`) w trakcie wykonywania pętli `foreach`. Ponadto zmienna pętli od wersji C# 5.0 języka działa nieco inaczej niż wcześniej. Ta różnica jest odczuwalna tylko wtedy, gdy w ciele pętli zmienna jest używana w wyrażeniu lambda lub w metodzie anonimowej. Szczegółowe informacje znajdziesz w rozdziale 13.

ZAGADNIENIE DLA POCZĄTKUJĄCYCH

W jakich sytuacjach sensowne jest stosowanie instrukcji switch?

Czasem program powinien wielokrotnie sprawdzać tę samą wartość w kilku instrukcjach if. Dotyczy to na przykład wartości zmiennej input na listingu 4.48.

Listing 4.48. Sprawdzanie w instrukcji if danych wejściowych od gracza

```
// ...
// Sprawdzanie danych wejściowych od gracza wykonującego posunięcie.
if ( (input == "1") ||
    (input == "2") ||
    (input == "3") ||
    (input == "4") ||
    (input == "5") ||
    (input == "6") ||
    (input == "7") ||
    (input == "8") ||
    (input == "9") )
{
    // Zapisanie i wykonanie posunięcia zgodnie z poleceniem gracza.
    // ...
    valid = true;
}
else if ( (input == "") || (input == "quit") )
{
    valid = true;
}
else
{
    System.Console.WriteLine(
        "\nBŁĄD: Wprowadź wartość z przedziału od 1 do 9. "
        + "Wciśnij ENTER, by zamknąć program.");
}
// ...
```

Ten kod sprawdza poprawność wprowadzonego tekstu, by ustalić, czy podano prawidłowy ruch w grze w kółko i krzyżyk. Jeśli wartość zmiennej input to 9, program musi wykonać dziewięć różnych testów. Lepszym rozwiązaniem byłoby przejście do odpowiedniego kodu już po jednym teście. Aby to umożliwić, należy zastosować instrukcję switch.

Podstawowa postać instrukcji switch

Podstawowa postać instrukcji switch jest łatwiejsza do zrozumienia niż skomplikowane instrukcje if, gdy trzeba porównać daną wartość z wieloma stałymi. Instrukcja switch wygląda następująco:

```
switch (wyrażenie)
{
    case stała:
        instrukcje
```

```
default:  
    instrukcje  
}
```

Oto omówienie komponentów instrukcji switch:

- Człon *wyrażenie* określa wartość porównywana z różnymi stałymi. Typ tego wyrażenia wyznacza typ nadzędny instrukcji switch. Dozwolone typy nadzędne to: bool, sbyte, byte, short, ushort, int, uint, long, ulong, char i enum (ten ostatni omówiono w rozdziale 9.), odpowiadające im typy przyjmujące wartość null, a także typ string.
- Człon *stała* to dowolne wyrażenie stałe o typie zgodnym z typem nadzędnym.
- Grupa jednej lub kilku etykiet case (albo etykiety default) razem z grupą instrukcji to sekcja instrukcji switch. W przedstawionym wzorcu występują dwie takie sekcje, a na listingu 4.49 znajduje się instrukcja switch z trzema sekcjami.
- Człon *instrukcje* to jedna lub kilka instrukcji wykonywanych, gdy wyrażenie ma wartość równą jednej ze stałych wymienionych w etykiecie danej sekcji. Kod nie może mieć możliwości wyjścia poza punkt końcowy grupy instrukcji. Dlatego zwykle ostatnim poleceniem w grupie jest instrukcja skoku — na przykład break, return lub goto.

Wskazówka

NIE stosuj continue jako instrukcji skoku do wychodzenia z sekcji instrukcji switch. Jest to wprawdzie dopuszczalne, gdy instrukcja switch znajduje się w pętli, jednak trudno zrozumieć wówczas znaczenie polecenia break w dalszych sekcjach.

Instrukcja switch powinna mieć przynajmniej jedną sekcję. Polecenie switch(x){} jest poprawne, ale prowadzi do wygenerowania ostrzeżenia. Warto wspomnieć także o innej kwestii — zgodnie z wcześniej przedstawioną wskazówką nie należy pomijać nawiasów klamrowych. Wyjątkiem od tej reguły jest pomijanie nawiasów dla instrukcji case i break, ponieważ te słowa kluczowe oznaczają początek oraz koniec bloku i z tego względu nie wymagają nawiasów klamrowych.

Listing 4.49 z instrukcją switch działa tak jak seria instrukcji if z listingu 4.48.

Listing 4.49. Zastępowanie instrukcji if instrukcją switch

```
static bool ValidateAndMove(  
    int[] playerPositions, int currentPlayer, string input)  
{  
    bool valid = false;  
  
    // Sprawdzanie danych wejściowych podanych przez użytkownika wykonującego posunięcie.  
    switch (input)  
    {  
        case "1" :  
        case "2" :  
    }
```

```

case "3" :
case "4" :
case "5" :
case "6" :
case "7" :
case "8" :
case "9" :
    // Zapisanie i wykonanie posunięcia zgodnie z polecienniem użytkownika.
    ...
    valid = true;
    break;

case "" :
case "quit":
    valid = true;
    break;
default :
    // Jeśli żadna z pozostałych instrukcji case nie została wykonana,
    // tekst jest nieprawidłowy.
    System.Console.WriteLine(
        "\nBŁĄD: Wprowadź wartość z przedziału od 1 do 9. "
        + "Wciśnij ENTER, by zamknąć program");
    break;
}

return valid;
}

```

Na listingu 4.49 sprawdzanym wyrażeniem jest `input`. Ponieważ zmienna `input` jest typu `string`, typem nadziednym też jest typ `string`. Jeśli wartość zmiennej `input` to jeden z łańcuchów 1, 2, 3, 4, 5, 6, 7, 8 lub 9, posunięcie jest prawidłowe i należy zapisać w odpowiedniej komórce symbol (X lub O) używany przez gracza wykonującego ruch. Po dojściu programu do instrukcji `break` sterowanie jest przekazywane poza instrukcję `switch`.

W następnej sekcji pokazano, jak obsługiwać pusty łańcuch znaków i łańcuch `quit`. W tej sekcji kod ustawia wartość zmiennej `valid` na `true`, jeśli zmienna `input` ma jedną z dwóch wymienionych wartości. Sekcja `default` jest uruchamiana, jeśli żadna z wcześniejszych sekcji nie miała etykiety `case` z wartością pasującą do sprawdzanego wyrażenia.

Porównanie języków — przechodzenie do dalszych sekcji w instrukcji `switch` w języku C++

W języku C++, jeśli sekcja nie kończy się instrukcją skoku, sterowanie jest przekazywane do następnej sekcji, której kod zostaje wykonany. Ponieważ przypadkowe przechodzenie do dalszych sekcji jest w języku C++ często popełnianym błędem, język C# nie dopuszcza takiej możliwości. Projektanci C# uznali, że lepiej jest zapobiec temu częstemu źródłu błędów i zachętać do pisania bardziej czytelnego kodu, niż zachować zgodność z potencjalnie mylącym działaniem omawianej instrukcji w języku C++. Jeśli chcesz, by jedna z sekcji wykonywała też instrukcje z innej sekcji, możesz jawnie zapewnić takie rozwiążanie za pomocą instrukcji `goto`. Tę technikę przedstawiono w dalszej części rozdziału.

Warto zwrócić uwagę na kilka aspektów związanych z instrukcją switch:

- Instrukcja switch pozbawiona sekcji spowoduje zgłoszenie ostrzeżenia przez kompilator, ale zostanie skompilowana.
- Sekcje mogą występować w dowolnej kolejności. Sekcja default nie musi być umieszczona na końcu (a nawet w ogóle nie trzeba jej tworzyć, ponieważ jest opcjonalna).
- Język C# wymaga, by nie można było wyjść poza punkt końcowy poszczególnych sekcji (włącznie z ostatnią sekcją). Dlatego sekcje zwykle kończą się instrukcjami break, return, throw i goto.

W wersji C# 7.0 wprowadzono usprawnienie instrukcji switch, umożliwiające dopasowywanie do wzorca. Dzięki temu w wyrażeniu instrukcji switch można stosować wartości dowolnego typu danych, a nie tylko kilku wymienionych wcześniej typów. Dopasowywanie do wzorca pozwala tworzyć instrukcje switch oparte na typie wyrażenia, a także używać etykiet case, w których deklarowane są zmienne. Ponadto instrukcje switch z dopasowywaniem do wzorca umożliwiają stosowanie wyrażeń warunkowych, dlatego nie tylko typ, ale też wyrażenie logiczne na końcu etykiety case może wpływać na to, który z bloków należy wykonać. Więcej informacji na temat instrukcji switch z dopasowywaniem do wzorca znajdziesz w rozdziale 7.

Początek
7.0

Koniec
7.0

Instrukcje skoku

Możliwa jest zmiana ścieżki wykonywania pętli. Za pomocą instrukcji skoku można wyjść poza pętlę lub pominąć pozostałą część bieżącej iteracji i rozpoczęć następne powtórzenie pętli — nawet jeśli warunek wciąż ma wartość true. W tym podrozdziale omówiono wybrane sposoby przeskakiwania między różnymi punktami ścieżki wykonania programu.

Instrukcja break

Aby w języku C# wyjść z pętli lub instrukcji switch, należy wywołać instrukcję break. Po jej napotkaniu przez program sterowanie jest natychmiast przekazywane poza pętlę lub instrukcję switch. Na listingu 4.50 pokazano to na przykładzie pętli foreach z programu do gry w kółko i krzyżyk.

Listing 4.50. Używanie instrukcji break do wyjścia z pętli po wykryciu wygranej

```
class TicTacToe // Deklaracja klasy TicTacToe.
{
    static void Main() // Deklaracja punktu wejścia do programu.
    {
        int winner = 0;
        // Przechowuje pola wybrane przez poszczególnych graczy.
        int[] playerPositions = { 0, 0 };

        // Zapisanie na sztywno pozycji na planszy.
        // X | 2 | O
```

```

// ---+---+---
// O | O | 6
// ---+---+---
// X | X | X
playerPositions[0] = 449;
playerPositions[1] = 28;

// Wykrywanie wygranej jednego z graczy.
int[] winningMasks = {
    7, 56, 448, 73, 146, 292, 84, 273 };

// Sprawdzenie wszystkich masek oznaczających wygraną, by
// ustalić, czy któryś z graczy zwyciężył.
foreach (int mask in winningMasks)
{
    if ((mask & playerPositions[0]) == mask)
    {
        winner = 1;
        break;
    }
    else if ((mask & playerPositions[1]) == mask)
    {
        winner = 2;
        break;
    }
}

System.Console.WriteLine(
    $"Zwyciężył gracz nr { winner }");
}

```

Wynik działania kodu z listingu 4.50 pokazano w danych wyjściowych 4.24.

DANE WYJŚCIOWE 4.24.

```
Zwyciężył gracz nr 1
```

Na listingu 4.50 instrukcja `break` jest używana, gdy po wykonaniu ruchu przez gracza pozycja na planszy oznacza wygraną. Instrukcja `break` wymusza zakończenie wykonywania zawierającej ją pętli (lub instrukcji `switch`), po czym program przechodzi do następnego wiersza za pętlą. W przedstawionym listingu jeśli wynik porównania bitów to `true` (co oznacza, że pozycja na planszy daje wygraną jednej ze stron), instrukcja `break` powoduje skok w przepływie sterowania i wyświetlenie zwycięzcy.

■ ZAGADNIENIE DLA POCZĄTKUJĄCYCH

Sprawdzanie pozycji za pomocą operatorów bitowych

W przykładowym programie do gry w kółko i krzyżyk operatory bitowe są wykorzystywane do ustalenia, który gracz zwyciężył w rozgrywce. Kod najpierw zapisuje wybrane przez obu graczy pola w mapie bitowej `playerPositions`. Używana jest tu tablica, co pozwala zachować ruchy obu zawodników.

Na początku oba elementy tablicy playerPositions zawierają same wartości 0. Gdy każdy gracz wykonuje posunięcie, we właściwej tablicy ustawiany jest bit odpowiadający ruchowi. Jeśli na przykład gracz zaznaczył pole 3, zmienna shifter jest ustawiana na wartość 3 – 1. Kod odejmuje 1, ponieważ tablice w C# są indeksowane od 0. Dlatego trzeba wprowadzić poprawkę na to, że pierwszej pozycji odpowiada indeks 0, a nie 1. Następnie kod przypisuje do zmiennej position wartość reprezentującą bit odpowiadający polu nr 3. Służy do tego operator przesunięcia w instrukcji `0000000000000001 << shifter`, gdzie shifter ma teraz wartość 2. W ostatnim kroku program ustawia element bieżącego gracza w tablicy playerPositions na wartość `000000000000100` (indeks gracza znów jest pomniejszany o 1, by uwzględnić to, że tablice są indeksowane od 0). Na listingu 4.51 zastosowano operator |=, dlatego po przedniej posunięcia są łączone z najnowszym ruchem.

Listing 4.51. Ustawianie bitów odpowiadających posunięciom obu graczy

```
int shifter; // Liczba pozycji, o jaką należy przesunąć maskę,  
               // by ustawić bit.  
int position; // Ustawiany bit.  
  
// Wywołanie int.Parse() przekształca wartość zmiennej input na liczbę całkowitą.  
// Wywołanie "int.Parse(input) - 1" jest potrzebne, ponieważ  
// tablice są indeksowane od zera.  
shifter = int.Parse(input) - 1;  
  
// Przesunięcie maski 00000000000000000000000000000001  
// o wartość zmiennej shifter.  
position = 1 << shifter;  
  
// Wykonanie operacji OR na polach wcześniej zaznaczonych przez gracza  
// i na nowym polu.  
// Ponieważ zmienią currentPlayer ma wartość 1 lub 2,  
// należy odjąć 1, by wykorzystać wartość tej zmiennej jako  
// indeks w indeksowanej od 0 tablicy.  
playerPositions[currentPlayer-1] |= position;
```

W dalszej części programu można sprawdzić wszystkie maski reprezentujące wygraną, aby ustalić, czy dany gracz doprowadził do pozycji dającej mu zwycięstwo. Ilustruje to listing 4.50.

Instrukcja continue

Założymy, że w pętli wykonywany jest blok z serią instrukcji. Jeśli w określonych warunkach należy wykonać w niektórych iteracjach tylko część tych instrukcji, można zastosować instrukcję `continue`. Powoduje ona przejście na koniec bieżącej iteracji i rozpoczęcie następnego powtórzenia pętli. Instrukcja `continue` wychodzi z bieżącej iteracji (niezależnie od tego, czy pozostały do wykonania dodatkowe instrukcje) i wraca do sprawdzania warunku pętli. Jeśli ten warunek wciąż ma wartość `true`, pętla kontynuuje pracę.

Na listingu 4.52 zastosowano instrukcję `continue`, by wyświetlić tylko litery tworzące domenę w adresie e-mail. Wynik działania kodu z listingu 4.52 przedstawiono w danych wyjściowych 4.25.

Listing 4.52. Określanie domeny adresu e-mail

```

class EmailDomain
{
    static void Main()
    {
        string email;
        bool insideDomain = false;
        System.Console.WriteLine("Wprowadź adres e-mail:");

        email = System.Console.ReadLine();

        System.Console.Write("Domena podanego adresu to: ");

        // Sprawdzenie wszystkich liter w adresie e-mail.
        foreach (char letter in email)
        {
            if (!insideDomain)
            {
                if (letter == '@')
                {
                    insideDomain = true;
                }
                continue;
            }

            System.Console.Write(letter);
        }
    }
}

```

DANE WYJŚCIOWE 4.25.

Wprowadź adres e-mail:
marek@programowanie.pl
Domena podanego adresu to: programowanie.pl

Gdy kod z listingu 4.52 nie doszedł jeszcze do domeny adresu e-mail, używa instrukcji `continue` do przeniesienia sterowania na koniec pętli, po czym przetwarzany jest następny znak adresu.

Prawie zawsze instrukcję `continue` można zastąpić instrukcją `if`. Zwykle skutkuje to poprawą czytelności kodu. Problem z instrukcją `continue` polega na tym, że powoduje ona powstanie wielu ścieżek przepływu sterowania w jednej iteracji, co pogarsza czytelność kodu. Listing 4.53 to zmodyfikowana wersja przykładowego kodu z listingu 4.52. W nowej wersji instrukcję `continue` zastąpiono blokiem `if/else`, by przedstawić bardziej czytelną wersję rozwiązania, niewymagającą instrukcji `continue`.

Listing 4.53. Zastępowanie instrukcji `continue` instrukcją `if`

```

foreach (char letter in email)
{
    if (insideDomain)
    {
        System.Console.Write(letter);
    }
}

```

```

    }
    else
    {
        if (letter == '@')
        {
            insideDomain = true;
        }
    }
}

```

Instrukcja goto

W pierwszych językach programowania nie były dostępne stosunkowo zaawansowane, ustrukturyzowane mechanizmy zarządzania przepływem sterowania, których obecność w nowych językach programowania, takich jak C#, jest uznawana za oczywistość. Dlatego kiedyś programiści musieli w obszarze zarządzania przepływem sterowania polegać na prostych rozgałęzieniach warunkowych (instrukcja if) i bezwarunkowych (instrukcja goto). Rozwijane za pomocą takich instrukcji programy były często mało zrozumiałe. Wielu doświadczonych programistów języka C# uważa występowanie w języku C# instrukcji takich jak goto za anachronizm. Jednak instrukcja ta wciąż jest dostępna i stanowi jedyną metodę umożliwiającą przeskakiwanie do dalszych sekcji w instrukcjach switch. Przyjrzyj się listingowi 4.54. Gdy ustawiona jest opcja /out, kod dzięki instrukcji goto przechodzi do warunku default. Podobnie dzieje się dla opcji /f.

Listing 4.54. Działanie instrukcji switch z instrukcjami goto

```

// ...
static void Main(string[] args)
{
    bool isOutputSet = false;
    bool isFiltered = false;

    foreach (string option in args)
    {
        switch (option)
        {
            case "/out":
                isOutputSet = true;
                isFiltered = false;
                goto default;
            case "/f":
                isFiltered = true;
                isRecursive = false;
                goto default;
            default:
                if (isRecursive)
                {
                    // Rekurencyjne przechodzenie w dół hierarchii.
                    // ...
                }
            else if (isFiltered)
            {

```

```
// Dodawanie opcji do listy filtrów.  
// ...  
}  
break;  
}  
// ...  
}
```

W danych wyjściowych 4.26 pokazano, jak wykonywać kod z listingu 4.54.

DANE WYJŚCIOWE 4.26.

```
C:\SAMPLES>Generate /out fizbottle.bin /f "*.xml" "*.wsdl"
```

Aby przejść do sekcji o etykiecie innej niż default, można zastosować składnię goto case *stała*; gdzie *stała* to stała powiązana z etykietą, do której kod ma przeskoczyć. W celu przejścia do kodu, który nie jest powiązany z sekcją instrukcji switch, poprzedź docelowe polecenie identyfikatorem i dwukropkiem. Ten identyfikator można wykorzystać w instrukcji goto. Możesz na przykład wpisać instrukcję z etykietą *etykieta* : Console.WriteLine();. Polecenie goto *etykieta*; spowoduje wtedy przejście do instrukcji z etykietą. Na szczęście język C# uniemożliwia używanie instrukcji goto do wchodzenia do bloku kodu. Instrukcja goto pozwala jedynie przechodzić do innych miejsc w danym bloku kodu lub w bloku zewnętrznym. Dzięki temu ograniczeniu język C# chroni przed większością poważnych nadużyć instrukcji goto, które zdarzają się w innych językach.

Mimo tych usprawnień stosowanie instrukcji goto zwykle jest uznawane za nieeleganckie. Ta instrukcja prowadzi do powstawania trudnych do zrozumienia programów i jest objawem kodu o złej strukturze. Jeśli chcesz wielokrotnie wykonywać sekcję kodu lub uruchamiać ją w różnych warunkach, zastosuj pętlę lub przenieś dany kod do odrębnej metody.

Wskazówka

UNIKAJ stosowania instrukcji goto.

Dyrektywy preprocessora języka C#

Instrukcje związane z przepływem sterowania przetwarzają wyrażenia w czasie wykonywania programu, natomiast preprocessor języka C# działa w czasie komplikacji. Polecenia preprocessora to dyrektywy skierowane do kompilatora języka C#. Te dyrektywy pozwalają wskazać sekcje kodu, które należy skompilować. Służą też do informowania, jak należy obsługiwać określone błędy i ostrzeżenia powodowane przez kod. Polecenia preprocessora języka C# mogą też być instrukcjami dla edytorów języka C# dotyczącymi organizacji kodu.

Porównanie języków — wstępne przetwarzanie w C++

Języki takie jak C i C++ wykorzystują **preprocesor** do wykonywania operacji na kodzie na podstawie specjalnych znaczników. Dyrektywy preprocesora zwykle informują kompilator o tym, jak należy kompliować kod w pliku, natomiast same nie biorą udziału w procesie komplikacji. Natomiast kompilator języka C# przetwarza dyrektywy preprocesora w ramach zwykłej analizy leksykalnej kodu źródłowego. Dlatego język C# nie obsługuje makr preprocesora (możliwe jest tylko definiowanie stałych). Oznacza to, że nazwa *preprocesor* w języku C# jest myląca.

Każda dyrektywa preprocesora rozpoczyna się od znaku kratki (#), a wszystkie takie dyrektywy muszą się znajdować w jednym wierszu. Końcem dyrektywy jest znak nowego wiersza, a nie średnik.

Listę wszystkich dyrektyw preprocesora znajdziesz w tabeli 4.5.

Tabela 4.5. Dyrektywy preprocesora

Instrukcja lub wyrażenie	Ogólna składnia	Przykład
Dyrektyna #if	#if wyrażenie-preprocesora kod #endif	#if CSHARP2PLUS Console.Clear(); #endif
Dyrektyna #elif	#if wyrażenie-preprocesora1 kod #elif wyrażenie-preprocesora2 kod #endif	#if LINUX ... #elif WINDOWS ... #endif
Dyrektyna #else	#if kod #else kod #endif	#if CSHARP1 ... #else ... #endif
Dyrektyna #define	#define symbol-warunkowy	#define CSHARP2PLUS
Dyrektyna #undef	#undef symbol-warunkowy	#undef CSHARP2PLUS
Dyrektyna #error	#error komunikat-preprocesora	#error Błędna implementacja
Dyrektyna #warning	#warning komunikat-preprocesora	#warning Wymaga sprawdzenia kodu
Dyrektyna #pragma	#pragma warning	#pragma warning disable 1030
Dyrektyna #line	#line nowy-wiersz nowy-plik #line default	#line 467 "TicTacToe.cs" #line default
Dyrektyna #region	#region nazwa-obszaru kod #endregion	#region Metody ... #endregion
Dyrektyna #nullable	#nullable enable disable restore	#nullable enable ..string? text = null; #nullable restore

Początek
8.0

Koniec
8.0

Kod z tej książki często generuje ostrzeżenia, ponieważ wiele listingów jest niekompletnych (pokazywane są wstępne, niedokończone fragmenty kodu). Aby zablokować ostrzeżenia, które w przykładowym kodzie są nieistotne, można dodać do pliku dyrektywy #pragma. W tabeli 4.6 pokazane są niektóre ostrzeżenia blokowane w różnych fragmentach kodu z rozdziału 4.

Tabela 4.6. Przykładowe ostrzeżenia

Kategoria	Ostrzeżenie
CS0168	Zmienna jest zadeklarowana, ale nie jest używana.
CS0219	Zmienna ma przypisaną wartość, która jednak nie jest używana.
IDE0059	Niepotrzebne przypisanie wartości.

Tego rodzaju wyłączające ostrzeżenia dyrektywy #pragma często występują w kodzie źródłowym do tej książki, aby wyeliminować ostrzeżenia, które pojawiają się, ponieważ przykłady mają ilustrować różne zagadnienia i nie są w pełni dopracowane.

Wykluczanie i dołączanie kodu (#if, #elif, #else, #endif)

Dyrektyny preprocessora prawdopodobnie najczęściej stosuje się do kontrolowania tego, kiedy i jak dołączany jest kod. Na przykład aby napisać kod, który można skompilować zarówno za pomocą kompilatorów dla wersji C# 2.0 i nowszych, jak i dla wersji 1.0, można zastosować dyrektywę kompilatora wykluczającą kod specyficzny dla wersji C# 2.0, gdy używany jest kompilator dla wersji 1.0. Ilustruje to przedstawiony na listingu 4.55 fragment z programu do gry w kółko i krzyżyk.

Listing 4.55. Wykluczanie kodu dla wersji C# 2.0, gdy używany jest kompilator dla wersji C# 1.x

```
#if CSHARP2PLUS
System.Console.Clear();
#endif
```

Tu wywoływana jest metoda System.Console.Clear(). Dzięki zastosowaniu dyrektyw preprocessora #if i #endif ten wiersz kodu jest komplikowany tylko wtedy, gdy zdefiniowany jest symbol preprocessora CSHARP2PLUS.

Innym zastosowaniem omawianych dyrektyw preprocessora jest obsługa różnic między systemami operacyjnymi. Na przykład interfejsy API specyficzne dla systemów Windows i Linux można umieścić w dyrektywach #if z nazwami WINDOWS i LINUX. Programiści często stosują tego rodzaju dyrektywy zamiast komentarzy wielowierszowych /* ... */), ponieważ łatwiej jest modyfikować działanie takich dyrektyw za pomocą odpowiednich symboli lub funkcji wyszukiwania i zastępowania.

Ostatnim częstym zastosowaniem dyrektyw jest używanie ich do debugowania. Jeśli umieścisz kod w dyrektywie #if DEBUG, w większości środowisk IDE dany kod zostanie pominięty w trakcie budowania wersji produkcyjnej. W wielu środowiskach IDE symbol DEBUG jest domyślnie zdefiniowany na potrzeby oznaczania komplikacji diagnostycznej, a symbol RELEASE służy do budowania wersji produkcyjnych.

Aby utworzyć warunek `else-if`, możesz zastosować dyrektywę `#elif` w dyrektywie `#if`, zamiast tworzyć dwa zupełnie odrębne bloki `#if`. Ilustruje to listing 4.56.

Listing 4.56. Używanie dyrektyw `#if`, `#elif` i `#endif`

```
#if LINUX
...
#elif WINDOWS
...
#endif
```

Definiowanie symboli preprocesora (#define, #undef)

Symbole preprocesora można definiować na dwa sposoby. Pierwszy polega na użyciu dyrektywy `#define`, co pokazano na listingu 4.57.

Listing 4.57. Przykład zastosowania dyrektywy `#define`

```
#define CSHARP2PLUS
```

Drugie podejście polega na użyciu opcji `define` w wierszu polecień. W danych wyjściowych 4.27 pokazano, jak stosować tę technikę za pomocą narzędzia dotnet.

DANE WYJŚCIOWE 4.27.

```
>dotnet.exe -define:CSHARP2PLUS TicTacToe.cs
```

Aby dodać więcej definicji, rozdziel je średnikami. Zaletą używania opcji kompilatora `define` jest to, że nie trzeba wprowadzać zmian w kodzie źródłowym. Dlatego można wykorzystać te same pliki źródłowe do wygenerowania dwóch różnych plików binarnych.

Jeśli chcesz usunąć definicję symbolu, zastosuj dyrektywę `#undef` w taki sam sposób, jak użyłeś dyrektywy `#define`.

Generowanie błędów i ostrzeżeń (#error, #warning)

Czasem trzeba oznaczyć możliwy problem w kodzie. W tym celu można dodać dyrektywy `#error` lub `#warning`, by wygenerować błąd lub ostrzeżenie. Kod z listingu 4.58 to fragment programu do gry w kółko i krzyżyk. Pokazano tu, jak ostrzec, że kod nie zapobiega jeszcze wielokrotnemu wprowadzeniu tego samego ruchu przez gracza. Wynik działania kodu z listingu 4.58 pokazano w danych wyjściowych 4.28.

Listing 4.58. Definiowanie ostrzeżenia za pomocą dyrektywy `#warning`

```
#warning "Dozwolone jest wielokrotne wprowadzenie tego samego ruchu."
```

DANE WYJŚCIOWE 4.28.

```
Performing main compilation...
...\\tictactoe.cs(471,16): warning CS1030: #warning: '"Dozwolone jest
↪wielokrotne wprowadzenie tego samego ruchu."'
Build complete -- 0 errors, 1 warnings
```

Dołączenie dyrektywy `#warning` gwarantuje, że kompilator poinformuje o ostrzeżeniu (tak jak w danych wyjściowych 4.28). Przedstawione tu ostrzeżenie pozwala poinformować, że kod zostanie usprawniony lub że występuje w nim błąd. Można to traktować jako proste przypomnienie dla programisty o zadaniu oczekującym na wykonanie.

Początek
2.0

Wyłączanie komunikatów z ostrzeżeniami (#pragma)

Ostrzeżenia są pomocne, ponieważ wskazują kod, który może sprawiać problemy. Jednak czasem warto wyłączyć określone ostrzeżenia, ponieważ można je bezpiecznie zignorować. Od wersji C# 2.0 kompilatory obsługują służącą do tego dyrektywę preprocesora `#pragma` (zobacz listing 4.59)⁴.

Listing 4.59. Używanie dyrektywy preprocesora #pragma do wyłączania dyrektywy #warning

```
#pragma warning disable CS1030
```

Zwróci uwagę na to, że w danych wyjściowych od kompilatora numery ostrzeżeń są poprzedzone przedrostkiem CS. Jednak w dyrektywie `#pragma` ten przedrostek nie jest podawany. Jeśli dyrektywa preprocesora nie jest używana, należy podać numer ostrzeżenia wygenerowany przez kompilator.

Aby ponownie aktywować ostrzeżenie, należy w dyrektywie `#pragma` podać opcję `restore` po słowie `warning`. Ilustruje to listing 4.60.

Listing 4.60. Stosowanie dyrektywy preprocesora #pragma do ponownego aktywowania ostrzeżenia

```
#pragma warning restore CS1030
```

Dwie przedstawione dyrektywy mogą obejmować blok kodu, w którym ostrzeżenie jest jawnie określone jako nieistotne.

Jednym z najczęściej wyłączanych ostrzeżeń jest to o numerze CS1591. To ostrzeżenie pojawia się, gdy za pomocą opcji kompilatora `/doc` włączysz generowanie dokumentacji w formacie XML, ale nie udokumentujesz wszystkich publicznych elementów programu.

Opcja nowarn:<lista ostrzeżeń>

Oprócz dyrektywy `#pragma` kompilatory języka C# zwykle obsługują też opcję `nowarn=<lista ostrzeżeń>`. Pozwala ona uzyskać ten sam efekt co dyrektywa `#pragma`, przy czym polecenie można przekazać jako opcję kompilatora, zamiast zapisywać je w kodzie źródłowym. Opcja `nowarn` wpływa na cały proces komplikacji, natomiast dyrektywa `#pragma` dotyczy tylko pliku, w którym występuje. W danych wyjściowych 4.29 pokazano, jak z poziomu wiersza poleceń wyłączyć ostrzeżenie CS0219.

⁴ Została ona wprowadzona w wersji C# 2.0.

DANE WYJŚCIOWE 4.29.

```
> dotnet build /p:NoWarn="0219"
```

Okręsianie numerów wierszy (#line)

Dyrektyna `#line` służy do określania, który numer wiersza kompilator języka C# ma zgłaszać w komunikacie o błędzie lub ostrzeżeniu. Ta dyrektywa jest używana głównie w narzędziach i środowiskach projektowych generujących kod w języku C#. Na listingu 4.61 rzeczywisty numer wiersza z pliku jest pokazany po lewej stronie.

Listing 4.61. Dyrektywa preprocesora `#line`

```
124      #line 113 "TicTacToe.cs"  
125      #warning "Dozwolone jest wielokrotne wprowadzenie tego samego ruchu."  
126      #line default
```

Dodanie tej dyrektywy `#line` powoduje, że kompilator zgłosi ostrzeżenie z wiersza 125, w taki sposób, jakby dotyczyło ono wiersza 113. Ilustruje to wyświetlany przez kompilator komunikat o błędzie widoczny w danych wyjściowych 4.30.

DANE WYJŚCIOWE 4.30.

```
Performing main compilation...  
...\\tictactoe.cs(113,18): warning CS1030: #warning: '" Dozwolone jest  
→wielokrotne wprowadzenie tego samego ruchu."'  
Build complete -- 0 errors, 1 warnings
```

Dodanie dyrektywy `#line` ze słowem `default` anuluje efekt wykonania wszystkich wcześniejszych dyrektyw `#line`. Jest to informacja dla kompilatora, że ma zgłaszać rzeczywiste numery wierszy zamiast tych ustawionych we wcześniejszych wywołaniach dyrektywy `#line`.

Wskazówki dla edytorów kodu z interfejsem graficznym (#region, #endregion)

Dwie dyrektywy preprocesora w języku C#, `#region` i `#endregion`, są przydatne tylko w kontekście edytorów kodu z interfejsem graficznym. Takie edytory, na przykład Microsoft Visual Studio, mogą przeszukiwać kod źródłowy w celu znalezienia wymienionych dyrektyw i na ich podstawie udostępniać programistom określone mechanizmy. Język C# umożliwia zadeklarowanie obszaru kodu za pomocą dyrektywy `#region`. Trzeba ją połączyć z pasującą dyrektywą `#endregion`. W obu tych dyrektywach opcjonalnie można dodać tekstowy opis. Ponadto możliwe jest zagnieżdżanie jednych obszarów w innych.

Na listingu 4.62 pokazano te dyrektywy w programie do gry w kółko i krzyżyk.

Listing 4.62. Dyrektywy preprocesora #region i #endregion

```

...
#region Wyświetlanie planszy do gry w kółko i krzyżyk

#if CSHARP2PLUS
    System.Console.Clear();
#endif

// Wyświetlanie obecnego stanu planszy.
border = 0; // Pozwala dodać pierwszą linię pionową (border[0] = '|')

// Wyświetlanie górnej linii kresek.
// ("|---+---+---|")
System.Console.Write(borders[2]);
foreach (char cell in cells)
{
    // Wyświetlanie wartości komórki i pionowej kreski.
    System.Console.Write($" { cell } { borders[border] }");

    // Inkrementacja w celu przejścia do następnej kreski.
    border++;
}

// Ustawianie zmiennej border na 0, jeśli border jest równe 3.
if (border == 3)
{
    border = 0;
}
#endregion Wyświetlanie planszy do gry w kółko i krzyżyk
...

```

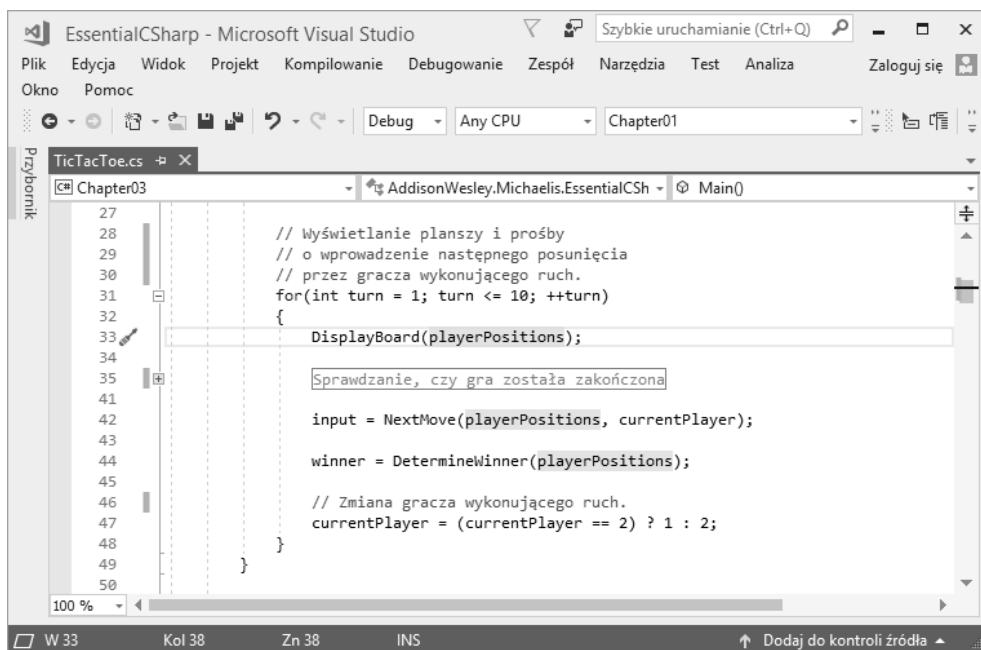
Te dyrektywy preprocesora są używane na przykład w środowisku Microsoft Visual Studio. To środowisko sprawdza kod i udostępnia (po lewej stronie okna edytora kodu) drzewo pozwalające rozwijać i zwijać bloki kodu. Te bloki odpowiadają regionom wyznaczonym przez dyrektywy #region (zobacz rysunek 4.5).

Początek
8.0

Włączanie obsługi typów referencyjnych dopuszczających wartości null (#nullable)

W rozdziale 3. opisano, że dyrektywa preprocesora #nullable włącza (lub wyłącza) obsługę typów referencyjnych dopuszczających wartości null. Dyrektywa #nullable enable włącza obsługę takich typów, a dyrektywa #nullable disable ją dezaktywuje. Ponadto dyrektywa #nullable restore przywraca mechanizm obsługi typów referencyjnych dopuszczających wartości null do jego domyślnego stanu, skonfigurowanego w elemencie Nullable w pliku projektu.

Koniec
8.0



Rysunek 4.5. Zwinięty obszar w środowisku Microsoft Visual Studio .NET

Podsumowanie

Ten rozdział rozpoczął się od przedstawienia operatorów języka C# związanych z przypisywaniem i operacjami arytmetycznymi. Dalej pokazano, jak korzystać z operatorów i jak tworzyć stałe za pomocą słowa kluczowego `const`. Nie opisano jednak po kolej wszytskich operatorów języka C#. Przed omówieniem operatorów relacyjnych i operatorów porównań logicznych przedstawiono instrukcję `if` oraz ważne pojęcia „blok kodu” i „zasięg”. Aby zakończyć opisywanie operatorów, omówiono operatory bitowe z uwzględnieniem masek. Objaśniono też instrukcje związane z przepływem sterowania — pętle, instrukcję `switch` i instrukcję `goto`. W końcowej części rozdziału znalazło się omówienie dyrektyw preprocesora języka C#.

Priorytety operatorów zostały opisane wcześniej w rozdziale. Tabela 4.7 zawiera przegląd priorytetów wszystkich operatorów, w tym kilku, których do tego miejsca książki nie opisano.

Prawdopodobnie najlepszym sposobem na powtórzenie całego materiału omówionego w rozdziałach od 1. do 4. jest przejrzenie programu do gry w kółko i krzyżyk z pliku `Chapter04\TicTacToe.cs`. Dzięki zapoznaniu się z tym programem zobaczysz jeden ze sposobów na połączenie wszystkich zdobytych informacji w celu zbudowania kompletnego programu.

Tabela 4.7. Priorytety operatorów*

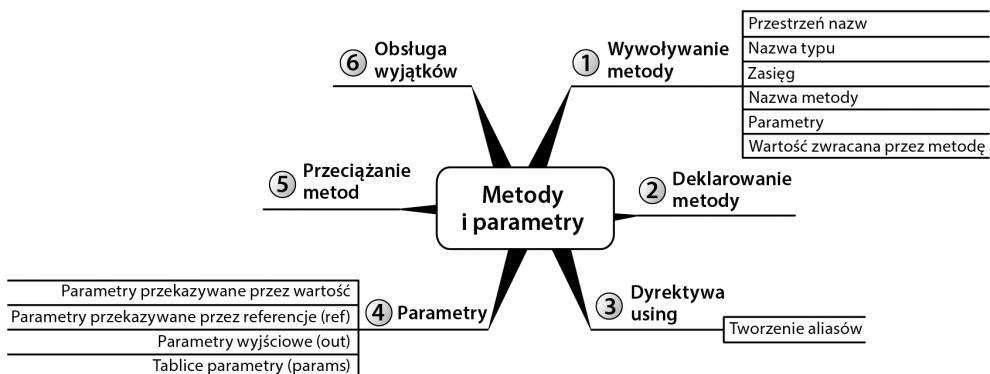
Kategoria	Operator
Podstawowe	x.y f(x) a[x] x++ x-- new typeof(T) checked(x) unchecked(x) default(T) nameof(x) delegate{}()
Jednoargumentowe	+ - ! ~ ++x --x (T)x await x
Mnożenia	* / %
Dodawania	+ -
Przesunięcia	<< >>
Relacyjne i sprawdzania typów	< > <= >= is as
Równości	== !=
Logiczne AND	&
Logiczne XOR	^
Logiczne OR	
Warunkowe AND	&&
Warunkowe OR	
Operator ??	??
Operator trójargumentowy (?:)	?:
Przypisania i wyrażeń lambda	= *= /= %= += -= <<= >>= &= ^= = =>

*Wiersze określają priorytety od najwyższego do najniższego.

5

Metody i parametry

NA PODSTAWIE TEGO, CZEGO JUŻ NAUCZYŁEŚ SIĘ o programowaniu w języku C#, powinieneś móc pisać programy składające się z listy instrukcji, podobne do aplikacji z lat 70. ubiegłego wieku. Jednak od tamtego czasu w dziedzinie programowania poczyniono duże postępy. Gdy programy zaczęły stawać się bardziej skomplikowane, pojawiły się nowe paradigmaty pomagające radzić sobie ze złożonością. Programowanie *proceduralne* lub *ustrukturyzowane* zapewnia konstrukcje pozwalające grupować instrukcje w jednostki. Ponadto dzięki programowaniu ustrukturyzowanemu można przekazywać dane do grupy instrukcji, a następnie, po wykonaniu poleceń, pobierać zwracane wartości.



Oprócz podstaw związanych z wywoywaniem i definiowaniem metod w tym rozdziale omówiono też bardziej zaawansowane zagadnienia: rekurencję, przeciążanie metod, parametry opcjonalne i argumenty nazwane. Wszystkie wywołania metod przedstawione do tej pory i do końca tego rozdziału są statyczne (temat wywołań statycznych szczegółowo opisano w rozdziale 6.).

Tego, jak definiować metody, dowiedziałeś się już na przykładzie programu `HelloWorld` z rozdziału 1. W tym programie zdefiniowałeś metodę `Main()`. W niniejszym rozdziale poznasz więcej szczegółów na temat tworzenia metod. Między innymi zobaczysz specjalną składnię języka C# (słowa `ref` i `out`) służącą do tworzenia parametrów, które pozwalają przekazywać do metod zmienne zamiast wartości. W końcowej części rozdziału zapoznasz się z podstawami obsługi błędów.

Wywoływanie metody

ZAGADNIENIE DLA POCZĄTKUJĄCYCH

Czym jest metoda?

Do tego miejsca wszystkie instrukcje pisanych programów pojawiały się w jednej grupie w metodzie o nazwie `Main()`. W aplikacjach bardziej skomplikowanych niż programy przedstawiane do tej pory kod oparty na jednej metodzie szybko staje się trudny w konserwacji. Taki kod niełatwo jest przeczytać i zrozumieć.

Metoda to komponent, który pozwala połączyć sekwencję instrukcji w celu wykonania określonej operacji lub obliczenia wyniku. W ten sposób można nadać kodowi strukturę i uporządkować instrukcje programu. Pomyśl na przykład o metodzie `Main()`, która zlicza wiersze kodu źródłowego z plików z katalogu. Zamiast tworzyć jedną dużą metodę `Main()`, możesz napisać krótszą wersję, co pozwala w razie potrzeby skoncentrować się na szczegółach implementacji poszczególnych mniejszych metod. Przykładowe rozwiązanie przedstawia listing 5.1.

Listing 5.1. Grupowanie instrukcji w metodach

```
class LineCount
{
    static void Main()
    {
        int lineCount;
        string files;
        DisplayHelpText();
        files = GetFiles();
        lineCount = CountLines(files);
        DisplayLineCount(lineCount);
    }
    // ...
}
```

Zamiast umieszczać wszystkie instrukcje w metodzie `Main()`, na listingu rozdzielono kod między grupy instrukcji nazywane metodami. Instrukcje `System.Console.WriteLine()`, które wyświetlają tekst z pomocą, przeniesiono do metody `DisplayHelpText()`. Wszystkie instrukcje używane do określania sprawdzanych plików znalazły się w metodzie `GetFiles()`. Aby zliczyć wiersze w plikach, kod wywołuje metodę `CountLines()`, po czym wyświetla wyniki za pomocą metody `DisplayLineCount()`. Teraz można szybko przejrzeć kod i zrozumieć jego ogólną strukturę, ponieważ nazwy metod opisują ich przeznaczenie.

Wskazówka

NADAWAJ metodom nazwy w postaci czasowników.

Metoda zawsze jest powiązana z typem (zwykle z **klassą**), który umożliwia połączenie ze sobą powiązanych metod.

Metody mogą przyjmować dane, przekazywane w postaci **argumentów** odpowiadających **parametrom** metod. Parametry to zmienne służące do przekazywania danych z **jednostki wywołującej** (kodu zawierającego wywołanie metody) do uruchamianej metody (`Write()`, `WriteLine()`, `GetFiles()`, `CountLines()` itd.). Na listingu 5.1 `files` i `lineCount` to argumenty przekazywane do metod `CountLines()` i `DisplayLineCount()` za pomocą parametrów. Metody mogą też zwracać dane do jednostki wywołującej. Służy do tego **zwracana wartość** (na listingu 5.1 metoda `GetFiles()` zwraca wartość przypisywaną do zmiennej `files`).

Zaczniemy od ponownego przyjrzenia się metodom `System.Console.WriteLine()`, `System.Console.ReadLine()` i `System.Console.ReadLine()` z rozdziału 1. Tym razem zostaną one omówione jako przykłady ogólnych wywołań metod. Nie są tu istotne szczegóły dotyczące wyświetlania i pobierania danych w konsoli. Na listingu 5.2 wywoływana jest każda z tych trzech metod.

Listing 5.2. Proste wywołania metod

```
class HeyYou
{
    static void Main()
    {
        string firstName;
        string lastName;

        System.Console.WriteLine("Hej, ty!");

        System.Console.Write("Wprowadź imię: ");

        firstName = System.Console.ReadLine();
        System.Console.Write("Wprowadź nazwisko: ");
        lastName = System.Console.ReadLine();
        System.Console.WriteLine(
            $"Twoje imię i nazwisko to { firstName } { lastName }.");
    }
}
```

Wywołania metod obejmują nazwę metody, listę argumentów i zwracaną wartość. Pełna nazwa metody obejmuje przestrzeń nazw, nazwę typu i nazwę metody. Poszczególne człony takiej nazwy rozdzielają kropki. Dalej zobaczysz, że metody często są wywoływane za pomocą tylko fragmentu pełnej nazwy.

Przestrzenie nazw

Przestrzenie nazw to mechanizm pozwalający pogrupować wszystkie typy związane z mechanizmami z określonego obszaru. Przestrzenie nazw są hierarchiczne. Hierarchia może obejmować dowolną liczbę poziomów, choć rzadko jest ich więcej niż sześć. Zwykle hierarchia rozpoczyna się od nazwy firmy, po czym następuje nazwa produktu i obszaru funkcjonalnego. Na przykład w nazwie `Microsoft.Win32.Networking` zewnętrzną przestrzenią nazw jest `Microsoft`. Obejmuje ona wewnętrzną przestrzeń nazw `Win32`, która zawiera jeszcze bardziej zagnieżdżoną przestrzeń `Networking`.

Przestrzenie nazw służą głównie do porządkowania typów na podstawie obszarów funkcyjonalnych. Dzięki temu typy można łatwiej odszukać i zrozumieć. Ponadto przestrzenie nazw pozwalały uniknąć konfliktów nazw typów. Kompilator potrafi na przykład odróżnić dwa typy o nazwie `Button`, jeśli typy te znajdują się w różnych przestrzeniach nazw. W ten sposób można odróżnić od siebie typy `System.Web.UI.WebControls.Button` i `System.Windows.Controls.Button`.

Na listingu 5.2 typ `Console` znajduje się w przestrzeni nazw `System`. Ta przestrzeń nazw obejmuje typy umożliwiające programistom wykonywanie wielu podstawowych operacji programistycznych. W prawie wszystkich programach w języku C# używane są typy z przestrzeni nazw `System`. W tabeli 5.1 znajdziesz listę różnych często stosowanych przestrzeni nazw.

Początek
4.0**Tabela 5.1.** Często używane przestrzenie nazw

Przestrzeń nazw	Opis
<code>System</code>	Obejmuje typy podstawowe, typy używane do konwersji między typami, typy matematyczne, typy do wywoływania programów i typy do zarządzania środowiskiem.
<code>System.Collections.Generics</code>	Obejmuje kolekcje ze ścisłą kontrolą typów wykorzystujące typy generyczne.
<code>System.Data</code>	Obejmuje typy używane do pracy z bazami danych.
<code>System.Drawing</code>	Obejmuje typy do wyświetlania elementów na wyświetlaczu i pracy z grafiką.
<code>System.IO</code>	Obejmuje typy służące do pracy z katalogami oraz manipulowania, wczytywania i zapisywania plików.
<code>System.Linq</code>	Obejmuje klasy i interfejsy przeznaczone do tworzenia zapytań o dane z kolekcji za pomocą technologii Language Integrated Query.
<code>System.Text</code>	Obejmuje typy używane do pracy z łańcuchami znaków i tekstem o różnym kodowaniu. Typy z tej grupy służą też do zmieniania kodowania.
<code>System.Text.RegularExpressions</code>	Obejmuje typy służące do pracy z wyrażeniami regularnymi.
<code>System.Threading</code>	Obejmuje typy do obsługi programowania wielowątkowego.
<code>System.Threading.Tasks</code>	Obejmuje typy do obsługi asynchroniczności z wykorzystaniem zadań.
<code>System.Web</code>	Obejmuje typy umożliwiające komunikację przeglądarki z serwerem (zwykle za pomocą HTTP). Mechanizmy z tej przestrzeni nazw służą do obsługi technologii ASP.NET.
<code>System.Windows</code>	Obejmuje typy przeznaczone do tworzenia rozbudowanych interfejsów użytkownika za pomocą platformy .NET 3.0 z wykorzystaniem technologii WPF (ang. Windows Presentation Framework; służy ona do budowania interfejsów użytkownika). W tej technologii wykorzystuje się język XAML (ang. Extensible Application Markup Language) do deklaratywnego projektowania interfejsów.
<code>System.Xml</code>	Obejmuje opartą na standardach obsługę przetwarzania danych w formacie XML.

Koniec
4.0

W wywołaniu metody nie zawsze trzeba podawać przestrzeń nazw. Jeśli wywołanie wyrażenia znajduje się w typie z tej samej przestrzeni nazw, z której pochodzi wywoływana metoda, kompilator potrafi wywnioskować, że należy zastosować tę właśnie przestrzeń nazw. Dalej w rozdziale zobaczysz, że także dyrektywa `using` pozwala wyeliminować konieczność podawania przestrzeni nazw.

Wskazówki

STOSUJ Notację Pascalową dla nazw przestrzeni nazw.

ROZWAŻ dostosowanie hierarchii katalogów z plikami z kodem źródłowym do hierarchii przestrzeni nazw.

Nazwa typu

W wywołaniach metod statycznych potrzebny jest kwalifikator określający nazwę typu. Nie jest on niezbędny, gdy docelowa metoda znajduje się w tym samym typie, w którym jest ona wywoływana¹. W dalszej części rozdziału wyjaśniono, że dyrektywa `using static` pozwala pominąć nazwę typu. Na przykład wywołanie `Console.WriteLine()` w metodzie `HelloWorld.Main()` wymaga podania typu (`Console`). Jednak, podobnie jak w przypadku przestrzeni nazw, C# umożliwia pominięcie nazwy typu w wywołaniu, jeśli metoda należy do typu obejmującego to wywołanie. Przykładowe wywoływanie tego rodzaju znajdziesz na listingu 5.4. Nazwa typu jest wtedy zbędna, ponieważ kompilator potrafi ją wywnioskować na podstawie lokalizacji wywołania. Gdy kompilator nie może tego zrobić, w wywołaniu trzeba podać nazwę typu.

Typy służą do grupowania metod i powiązanych z nimi danych. Na przykład `Console` to typ obejmujący (między innymi) metody `Write()`, `WriteLine()` i `ReadLine()`. Wszystkie te metody należą do jednej *grupy*, ponieważ są częścią typu `Console`.

Zasięg

W poprzednim rozdziale dowiedziałeś się, że *zasięg* elementu programu to obszar w tekście, w którym można używać danego elementu bez stosowania kwalifikatorów w nazwie. Wywołanie w deklaracji typu metody zadeklarowanej w tym samym typie nie wymaga kwalifikatora, ponieważ metoda znajduje się w zasięgu w całym zawierającym ją typie. Podobnie typ pozostaje w zasięgu przestrzeni nazw, w której jest zadeklarowany. Dlatego wywołanie metody w typie z określonej przestrzeni nazw nie wymaga podawania nazwy tej przestrzeni.

Nazwa metody

W każdym wywołaniu metody znajduje się nazwa metody. Można do niej dodać kwalifikatory w postaci nazw przestrzeni nazw i typu, przy czym nie zawsze są one potrzebne. Po nazwie metody znajduje się lista argumentów. Jest to podana w nawiasie lista rozdzielonych przecinkami wartości odpowiadających parametrom metody.

¹ Lub w klasie bazowej.

Parametry i argumenty

Metoda może przyjmować dowolną liczbę parametrów. Każdy parametr ma określony typ danych. Wartości parametrów podawane w jednostce wywołującej to **argumenty**. Każdy argument musi odpowiadać określonemu parametrowi. W poniższym wywołaniu metody znajdują się trzy argumenty.

```
System.IO.File.Copy(
    oldFileName, newFileName, false)
```

Jest to metoda z klasy `File` z przestrzeni nazw `System.IO`. Ta metoda przyjmuje trzy parametry. Pierwszy i drugi są typu `string`, a trzeci jest typu `bool`. W tym przykładzie do określenia dawnej i nowej nazwy pliku używane są zmienne (`oldFileName` i `newFileName`) typu `string`, a wartość `false` informuje, że kopiowanie powinno się zakończyć niepowodzeniem, jeśli plik o nazwie ze zmiennej `newFileName` już istnieje.

Wartości zwracane przez metody

W odróżnieniu od wywołania `System.Console.WriteLine()`, wywołanie `System.Console.ReadLine()` (zobacz listing 5.2) nie przyjmuje żadnych argumentów, ponieważ metoda `ReadLine()` jest zadeklarowana jako bezparametrowa. Ta metoda jednak ma **zwracaną wartość**. Za pomocą tej wartości można przekazać wyniki z wywołanej metody do jednostki wywołującej. Ponieważ metoda `System.Console.ReadLine()` ma zwracaną wartość, można przypisać tę wartość do zmiennej `firstName`. Ponadto można przekazać zwracaną wartość jako argument w wywołaniu innej metody, co pokazano na listingu 5.3.

Listing 5.3. Przekazywanie zwracanej wartości jako argumentu w wywołaniu innej metody

```
class Program
{
    static void Main()
    {
        System.Console.Write("Wprowadź imię: ");
        System.Console.WriteLine("Witaj, {0}!",
            System.Console.ReadLine());
    }
}
```

Zamiast przypisywać zwracaną wartość do zmiennej, a następnie podawać tę zmienną jako argument w wywołaniu `System.Console.WriteLine()`, na listingu 5.3 wywołano metodę `System.Console.ReadLine()` w wywołaniu metody `System.Console.WriteLine()`. W czasie wykonywania programu metoda `System.Console.ReadLine()` jest wywoływana wcześniej, a zwrócona przez nią wartość zostaje przekazana bezpośrednio do metody `System.Console.WriteLine()` zamiast do zmiennej.

Nie wszystkie metody zwracają dane. Nie robią tego na przykład metody `System.Console.WriteLine()` i `System.Console.ReadKey()`. Dalej zobaczysz, że w takich metodach typ zwracanej wartości to `void` (podobnie jak w deklaracji metody `Main` w klasie `HelloWorld`).

Instrukcje a wywołania metod

Na listingu 5.3 widoczna jest różnica między instrukcją a wywołaniem metody. Choć `System.Console.WriteLine("Witaj, {0}!", System.Console.ReadLine());` to jedna instrukcja, obejmuje ona dwa wywołania metod. Instrukcja często obejmuje jedno lub więcej wyrażeń. W tym przykładzie dwa z użytych wyrażeń to wywołania metod. Tak więc wywołanie metody może być częścią instrukcji.

Choć umieszczanie wielu wywołań metod w pojedynczych instrukcjach często zmniejsza ilość kodu, nie zawsze poprawia to jego czytelność i rzadko daje istotne korzyści w zakresie wydajności. Programiści powinni przedkładać czytelność nad zwięzłość.

Uwaga

Programiści powinni zwykle przedkładać czytelność nad zwięzłością. Czytelność jest niezwykle ważna w kontekście pisania kodu, który sam jest swoją dokumentacją, a tym samym jest łatwiejszy w późniejszej konserwacji.

Deklarowanie metody

Początek
6.0

W tym podrozdziale dokładnie omówiono deklarowanie metod z uwzględnieniem parametrów i typu zwracanej wartości. Na listingu 5.4 pokazano, jak stosować te elementy, a wynik działania kodu przedstawiono w danych wyjściowych 5.1.

Listing 5.4. Deklarowanie metody

```
class IntroducingMethods
{
    public static void Main()
    {
        string firstName;
        string lastName;
        string fullName;
        string initials;

        System.Console.WriteLine("Hej, ty!");

        firstName = GetUserInput("Wprowadź imię: ");
        lastName = GetUserInput("Wprowadź nazwisko: ");

        fullName = GetFullName(firstName, lastName);
        initials = GetInitials(firstName, lastName);
        DisplayGreeting(fullName, initials);
    }

    static string GetUserInput(string prompt)
    {
        System.Console.Write(prompt);
        return System.Console.ReadLine();
    }
}
```

```

static string GetFullName( // Metoda z ciałem w postaci wyrażenia wprowadzona w C# 6.0.
    string firstName, string lastName) =>
    $"'{ firstName } { lastName }";

static void DisplayGreeting(string fullName, string initials)
{
    System.Console.WriteLine(
        $"Witaj, { fullName }! Twój inicjały to { initials }");
    return;
}

static string GetInitials(string firstName, string lastName)
{
    return $"'{ firstName[0] }. { lastName[0] }.'";
}

```

DANE WYJŚCIOWE 5.1.

Hej, ty!
Wprowadź imię: **Inigo**
Wprowadź nazwisko: **Montoya**
Witaj, Inigo Montoya! Twój inicjały to I. M.

Koniec
6.0

Na listingu 5.4 zadeklarowanych jest pięć metod. W metodzie Main() kod wywołuje metodę GetUserInput(), a następnie metody GetFullName() i GetInitials(). Wszystkie trzy ostatnie metody zwracają wartość i przyjmują argumenty. Ponadto kod wywołuje metodę DisplayGreeting(), która nie zwraca danych. Żadna metoda w języku C# nie może istnieć poza typem. Tu typem obejmującym metody jest klasa IntroducingMethods. Nawet opisana w rozdziale 1. metoda Main musi należeć do jakiegoś typu.

Porównanie języków — metody globalne w językach C++ i Visual Basic

Język C# nie obsługuje metod globalnych. Wszystkie metody muszą się znajdować w deklaracjach typów. To dlatego metodę Main() oznaczono jako statyczną (static). Jest to odpowiednik metod globalnych z języka C++ i metod shared z języka Visual Basic.

ZAGADNIENIE DLA POCZĄTKUJĄCYCH**Refaktoryzacja kodu przez umieszczenie go w metodach**

Przenoszenie zbiorów instrukcji z dużych metod do odrębnych metod to jedna z form **refaktoryzacji**. Refaktoryzacja służy do zmniejszania liczby powtórzeń w kodzie, ponieważ metodę można wywoływać w wielu miejscach bez duplikowania kodu. Refaktoryzacja pozwala też zwiększyć czytelność kodu. W trakcie programowania dobrą praktyką jest stałe przeglądanie kodu i szukanie możliwości przeprowadzenia refaktoryzacji. Ten proces obejmuje wyszukiwanie bloków kodu, które trudno jest szybko zrozumieć, i przenoszenie ich do odrębnych metod o nazwach jasno definiujących działanie danych fragmentów. Często przedkłada się tę praktykę nad dodawanie komentarzy dla bloków kodu, ponieważ nazwa metody może opisywać działanie danego kodu.

Na przykład metoda `Main()` z listingu 5.4 działa podobnie jak metoda `Main()` z listingu 1.16 z rozdziału 1. Prawdopodobnie ważniejsze jest to, że choć oba listingi są bardzo łatwe do zrozumienia, z kodem z listingu 5.4 można się szybciej zapoznać dzięki samemu przejrzaniu zawartości metody `Main()`. Nie trzeba wtedy martwić się o szczegóły implementacji poszczególnych metod.

W środowisku Visual Studio można wybrać grupę instrukcji, kliknąć ją prawym przyciskiem myszy, a następnie wybrać opcję *Szybkie akcje i refaktoryzacja...* (*Ctrl+.*) z menu kontekstowego. Pozwala to automatycznie przenieść grupę instrukcji do nowej metody.

Deklarowanie parametrów formalnych

Przyjrzyj się deklaracjom metod `DisplayGreeting()`, `GetFullName()` i `GetInitials()`. Tekst znajdujący się w nawiasie w deklaracji metody to **lista parametrów formalnych**. Przy okazji omawiania typów generycznych dowiesz się też, że metody mogą mieć również **listę parametrów określających typ**. Gdy z kontekstu jasno wynika, jakiego rodzaju parametry są omawiane, są one nazywane po prostu *parametrami* na liście parametrów. Dla każdego parametru na liście należy podać typ i nazwę. Do rozdzielania parametrów na liście służą przecinki.

Większość parametrów działa niemal identycznie jak zmienne lokalne. Dla parametrów obowiązują te same co dla zmiennych konwencje tworzenia nazw. Dlatego w nazwach parametrów stosuj notację Wielbłąda. Ponadto nie można zadeklarować zmiennej lokalnej (zmiennej w metodzie) o tej samej nazwie, jaką ma parametr tej metody, ponieważ spowodowałoby to utworzenie dwóch *zmiennych lokalnych* o identycznych nazwach.

Wskazówka

STOSUJ notację Wielbłąda dla nazw parametrów.

Deklaracja typu wartości zwracanej przez metodę

W metodach `GetUserInput()`, `GetFullName()` i `GetInitials()` oprócz parametrów trzeba też określić **typ zwracanej wartości**. Możesz stwierdzić, że metoda zwraca wartość, ponieważ typ danych tej wartości pojawia się bezpośrednio przed nazwą metody w deklaracji. Każda z wymienionych przykładowych metod zwraca wartość typu `string`. Można podać tylko jeden typ zwracanej wartości (liczba podawanych parametrów może być dowolna).

Metody zwracające wartość (takie jak `GetUserInput()` i `GetInitials()`) prawie zawsze obejmują jedną lub więcej instrukcji `return`, które zwracają sterowanie do jednostki wywołującej. Instrukcja `return` obejmuje słowo kluczowe `return`, po którym następuje wyrażenie obliczające wartość zwracaną przez metodę. Na przykład w metodzie `GetInitials()` instrukcja `return` wygląda tak: `return $"{ firstName[0] } . { lastName[0] } .";` Wyrażenie (tu jest nim interpolowany łańcuch znaków) podawane po słowie kluczowym `return` musi mieć typ zgodny z zadeklarowanym typem wartości zwracanej przez metodę.

Jeśli metoda ma określony typ zwracanej wartości, w bloku instrukcji w ciele metody musi się znajdować *punkt końcowy, poza który kod nie może wyjść*. Oznacza to, że nie może być żadnego sposobu na wyjście sterowania poza koniec metody bez zwrócenia przez nią wartości. Często najłatwiejszym sposobem na upewnienie się, że ten warunek jest spełniony, jest umieszczenie instrukcji return na końcu metody. Jednak instrukcje return mogą występować też w innych miejscach. Na przykład instrukcja if lub switch w implementacji metody może obejmować zagnieźdzoną instrukcję return. Taki przykład przedstawiono na listingu 5.5.

Listing 5.5. Instrukcja return przed końcem metody

```
class Program
{
    static bool MyMethod()
    {
        string command = ObtainCommand();
        switch(command)
        {
            case "quit":
                return false;
                // Inne bloki case pominięto.
            default:
                return true;
        }
    }
}
```

Zauważ, że instrukcja return przekazuje sterowanie poza instrukcję switch. Dlatego w sekcjach kończących się instrukcją return instrukcja break nie jest potrzebna, by zapobiec niedozwolonemu przepływowi sterowania do dalszych sekcji.

Na listingu 5.5 ostatnia instrukcja metody nie jest instrukcją return. Zamiast niej występuje instrukcja switch. Jednak kompilator potrafi wykryć, że wszystkie możliwe ścieżki kodu w metodzie prowadzą do instrukcji return. Dlatego nie jest możliwe wyjście poza koniec metody. Metoda jest więc poprawna, choć nie kończy się instrukcją return.

Jeśli jakieś ścieżki kodu obejmują polecenia, do których nie da się dotrzeć (ponieważ wcześniej wywoływana jest instrukcja return), kompilator zgłosi ostrzeżenie z informacją, że dodatkowe polecenia nigdy nie będą uruchamiane.

Choć język C# dopuszcza tworzenie metod z wieloma instrukcjami return, kod jest zwykle bardziej czytelny i łatwiejszy w konserwacji, jeśli istnieje jeden punkt wyjścia z metody zamiast wielu instrukcji return rozrzuconych po różnych ścieżkach wykonania.

Ustawienie void jako typu zwracanej wartości oznacza, że metoda nie zwraca danych. Dlatego wywołania takiej metody nie można użyć w przypisaniu wartości do zmiennej lub w miejscu parametru w wywołaniu innej metody. Metody void mogą być używane tylko jako instrukcje. Ponadto w ciele takiej metody instrukcja return jest opcjonalna. Jeśli zastosujesz tę instrukcję, nie możesz podać po niej wartości. Na przykład typ zwracanej wartości metody Main() na listingu 5.4 to void, a metoda ta nie zawiera instrukcji return. Natomiast metoda DisplayGreeting() zawiera (opcjonalną) instrukcję return, po której jednak nie jest podana żadna zwracana wartość.

Choć teoretycznie metoda może zwracać tylko jedną wartość, może nią być krotka. Dlatego od wersji C# 7.0 można za pomocą składni dla krotek zwracać wiele wartości umieszczonych w krotce. Możesz na przykład zadeklarować pokazaną na listingu 5.6 metodę GetName().

Listing 5.6. Zwracanie wielu wartości za pomocą krotki

```
class Program
{
    static string GetUserInput(string prompt)
    {
        System.Console.Write(prompt);
        return System.Console.ReadLine();
    }

    static (string First, string Last) GetName()
    {
        string firstName, lastName;
        firstName = GetUserInput("Podaj imię: ");
        lastName = GetUserInput("Podaj nazwisko: ");
        return (firstName, lastName);
    }

    static public void Main()
    {
        (string First, string Last) name = GetName();
        System.Console.WriteLine($"Witaj, {name.First} {name.Last}!");
    }
}
```

W kontekście technicznym oczywiście nadal zwracana jest jedna wartość typu `Value` `Tuple<string, string>`. Jednak w praktyce możesz zwracać dowolną liczbę (najlepiej rozsądnej) wartości.

Metody z ciałem w postaci wyrażenia

Aby umożliwić tworzenie bardzo prostych deklaracji metod bez formalnego ciała, w wersji C# 6.0 wprowadzono **metody z ciałem w postaci wyrażenia**. Takie metody deklaruje się za pomocą wyrażenia, a nie z pełnym ciałem metody. Na listingu 5.4 `GetFullName()` to przykładowa metoda tego rodzaju.

```
static string GetFullName( string firstName, string lastName ) =>
    $"{ firstName } { lastName }";
```

Zamiast nawiasów klamrowych (stosowanych standardowo do tworzenia ciała metody) w metodach z ciałem w postaci wyrażenia używany jest operator `=>`, opisany szczegółowo w rozdziale 13. Typ danych zwracany przez podane po tym operatorze wyrażenie musi pasować do typu wartości zwracanej przez metodę. Oznacza to, że choć w metodzie z ciałem w postaci wyrażenia nie występuje jawnie podana instrukcja `return`, typ wartości zwracanej przez wyrażenie musi pasować do typu zwracanej wartości z deklaracji metody.

Metody z ciałem w postaci wyrażenia to składniowy skrót odpowiadający metodom z kompletnym ciałem. Dlatego tę technikę należy stosować tylko do najprostszych metod, których kod można zapisać w jednym wierszu.

Porównanie języków — pliki nagłówkowe w języku C++

W języku C# (inaczej niż w C++) implementacja klas nigdy nie jest oddzielana od ich deklaracji. W C# nie występują pliki nagłówkowe (.h) i pliki z implementacją (.cpp). Deklaracja i implementacja występują razem w jednym pliku. Język C# obsługuje jednak zaawansowany mechanizm tworzenia *metod częściowych*, który pozwala oddzielić deklarację definiującą metodę od implementacji. W tym rozdziale uwzględniane są jednak tylko zwykłe metody. Brak podziału na deklarację i implementację w języku C# sprawia, że nie trzeba utrzymywać nadmiarowych informacji o deklaracji w dwóch miejscach, co zdarza się w językach takich jak C++, gdzie występują oddzielne pliki nagłówkowe i pliki z implementacją.

ZAGADNIENIE DLA POCZĄTKUJĄCYCH

Przestrzenie nazw

Wcześniej wspomniano już, że **przestrzenie nazw** to mechanizm służący do kategoryzowania i grupowania powiązanych ze sobą typów. Jeśli programista zna jakiś typ, może się zapoznać z typami powiązanymi, sprawdzając zawartość przestrzeni nazw obejmującej znany typ. Ponadto dzięki przestrzeniom nazw różne typy mogą mieć te same nazwy, o ile znajdują się w różnych przestrzeniach.

Dyrektiva using

Kompletne nazwy bywają długie i skomplikowane. Można jednak zaimportować do pliku wszystkie typy z jednej lub kilku przestrzeni nazw, aby stosować nazwy elementów bez kwalifikatorów. Programiści języka C# mogą importować przestrzeń nazw za pomocą dyrektywy `using`, umieszczanej zwykle na początku pliku. Na przykład na listingu 5.7 nazwa `Console` nie jest poprzedzona przedrostkiem `System`. Przestrzeń nazw można pominąć, ponieważ na początku listingu znajduje się dyrektywa `using System`.

Listing 5.7. Przykład zastosowania dyrektywy `using System`

```
// Dyrektywa using importuje wszystkie typy
// z określonej przestrzeni nazw do całego pliku.
using System;

class HelloWorld
{
    static void Main()
    {
        // Przed nazwą Console nie trzeba używać kwalifikatora System,
        // ponieważ wcześniej dodano dyrektywę using.
        Console.WriteLine("Witaj, nazywam się Inigo Montoya.");
    }
}
```

Wyniki działania kodu z listingu 5.7 pokazano w danych wyjściowych 5.2.

DANE WYJŚCIOWE 5.2.

Witaj, nazywam się Inigo Montoya.

Dyrektyna using, na przykład using System, nie pozwala na pomijanie członu System dla typów zadeklarowanych w przestrzeniach nazw podległych względem System. Na przykład jeśli w kodzie używany jest typ StringBuilder z przestrzeni nazw System.Text, trzeba albo dodać dodatkową dyrektywę using System.Text;, albo stosować pełną nazwę typu System.Text.StringBuilder. Nie wystarczy podanie nazwy Text.StringBuilder. Dyrektywa using nie importuje więc typów z **zagnieżdżonych przestrzeni nazw**. Zagnieżdżone przestrzenie nazw, podawane po kropce po nazwie nadrzędnej przestrzeni, zawsze trzeba importować jawnie.

Porównanie języków — symbole wieloznaczne w dyrektywie import w Javie

Java umożliwia importowanie przestrzeni nazw z wykorzystaniem symboli wieloznacznych, na przykład:

```
import javax.swing.*;
```

Język C# nie obsługuje symboli wieloznacznych w dyrektywie using. Zamiast tego wymaga, by każdą przestrzeń nazw zaimportować jawnie.

Porównanie języków — dyrektywa Imports o zasięgu projektu w Visual Basic .NET

Visual Basic .NET, w odróżnieniu od języka C#, umożliwia dodanie instrukcji Imports (jest to odpowiednik dyrektywy using) dotyczącej całego projektu, a nie tylko konkretnego pliku. Oznacza to, że Visual Basic .NET umożliwia podanie z poziomu wiersza poleceń odpowiednika dyrektywy using obowiązującej w trakcie całej komplikacji.

Gdy często korzystasz z typów z danej przestrzeni nazw, dodanie dyrektywy using z tą przestrzenią do dobry pomysłu, ponieważ nie trzeba stosować pełnych nazw dla wszystkich typów z tej przestrzeni. Dlatego prawie wszystkie pliki z kodem w języku C# obejmują na początku dyrektywę using System. W dalszej części książki na listingach dyrektywa using System często jest pomijana. Jednak inne dyrektywy są bezpośrednio podawane.

Ciekawym skutkiem dodania dyrektywy using System jest to, że typ danych string można podawać z różną wielkością pierwszej litery: String lub string. Pierwsza z tych wersji jest dostępna dzięki dyrektywie using System, natomiast w drugiej używane jest słowo kluczowe string. Oba zapisy to poprawne nazwy typu System.String języka C#. Wynikowy kod CIL jest taki sam niezależnie od użytej wersji².

² Ja preferuję słowo kluczowe string, jednak niezależnie od tego, którą wersję programista wybierze, warto stosować ją konsekwentnie w całym projekcie.

ZAGADNIENIE DLA ZAAWANSOWANYCH

Zagnieżdżone dyrektywy using

Dyrektywy `using` można podawać nie tylko na początku pliku, ale też na początku deklaracji przestrzeni nazw. Na przykład jeśli zadeklarujesz nową przestrzeń nazw `EssentialCSharp`, możesz dodać dyrektywę `using` na początku deklaracji tej przestrzeni, co pokazano na listingu 5.8.

Listing 5.8. Dodawanie dyrektywy `using` w deklaracji przestrzeni nazw

```
namespace EssentialCSharp
{
    using System;

    class HelloWorld
    {
        static void Main()
        {
            // Nie trzeba dodawać kwalifikatora System przed typem Console,
            // ponieważ wcześniej znajduje się dyrektywa using.
            Console.WriteLine("Witaj, nazywam się Inigo Montoya.");
        }
    }
}
```

Wyniki działania kodu z listingu 5.8 przedstawiono w danych wyjściowych 5.3.

DANE WYJŚCIOWE 5.3.

```
Witaj, nazywam się Inigo Montoya.
```

Różnica między umieszczeniem dyrektywy `using` na początku pliku a dodaniem jej na początku deklaracji przestrzeni nazw polega na tym, że w drugim przypadku dyrektywa jest aktywna tylko w deklaracji danej przestrzeni nazw. Jeśli w kodzie nad lub pod deklaracją przestrzeni `EssentialCSharp` pojawi się nowa deklaracja przestrzeni nazw, dyrektywa `using System` nie będzie aktywna w tej nowej przestrzeni. Programiści rzadko piszą kod w ten sposób (zwłaszcza dlatego, że standardowo w jednym pliku deklarowany jest tylko jeden typ).

Początek
6.0

Dyrektyna `using static`

Dyrektyna `using` pozwala skrócić nazwę typu dzięki pominięciu przestrzeni nazw. Wystarczy wtedy podać samą nazwę typu, jeśli należy on do przestrzeni nazw określonej w dyrektywie. Dyrektywa `using static` umożliwia pominięcie zarówno przestrzeni nazw, jak i nazwy typu, gdy używane są składowe tego typu. Na przykład dyrektywa `using static System.Console` pozwala na podanie nazwy `WriteLine()` zamiast pełnej nazwy `System.Console.WriteLine()`. W ramach kontynuowania omawianego przykładu można zaktualizować listing 5.2 i wykorzystać dyrektywę `using static System.Console`. Efekt pokazano na listingu 5.9.

Listing 5.9. Dyrektywa using static

```
using static System.Console;

class HeyYou
{
    static void Main()
    {
        string firstName;
        string lastName;

        WriteLine("Hej, ty!");

        Write("Wprowadź imię: ");

        firstName = ReadLine();
        Write("Wprowadź nazwisko: ");
        lastName = ReadLine();
        WriteLine(
            $"Twoje imię i nazwisko to { firstName } { lastName }.");
    }
}
```

W tym przykładzie kod nie traci na czytelności. Wiadomo, że metody `WriteLine()`, `Write()` i `ReadLine()` są powiązane z dyrektywą `Console`. Można nawet stwierdzić, że uzyskany kod jest prostszy, a tym samym bardziej przejrzysty w porównaniu z poprzednią wersją.

Czasem jednak jest inaczej. Na przykład jeśli w kodzie używane są klasy z metodami o takich samych nazwach (na przykład z metodami `Exists()` dotyczącymi plików i katalogów), zastosowanie dyrektywy `using static` zmniejszy przejrzystość kodu w miejscach, gdzie wywoływana jest metoda `Exists()`. Podobnie jeśli klasa pisana przez programistę ma składowe (na przykład metody `Display()` lub `Write()`) o nazwach pokrywających się z nazwami z typu zaimportowanego za pomocą instrukcji `using static`, użycie tej dyrektywy może zmniejszać przejrzystość kodu dla jego czytelników.

Kompilator nie pozwala na taką wieloznaczność. Jeśli dostępne są dwie składowe o identycznej sygnaturze (w wyniku dodania dyrektyw `using static` lub niezależnych deklaracji składowych), wieloznaczne wywołania spowodują błąd kompilacji.

Koniec
6.0

Tworzenie aliasów

Za pomocą dyrektywy `using` można też utworzyć **alias** dla przestrzeni nazw lub typu. Alias to zastępca nazwa, którą można stosować w tekście, gdzie obowiązuje dana dyrektywa `using`. Dwa najważniejsze powody stosowania aliasów to umożliwienie rozróżniania dwóch typów o takich samych nazwach i skracanie długich nazw. Na listingu 5.10 zadeklarowano alias `CountDownTimer` reprezentujący typ `System.Timers.Timer`. Samo dodanie dyrektywy `using System.Timers` nie pozwala uniknąć stosowania pełnej nazwy dla typu `Timer`. Wynika to z tego, że przestrzeń nazw `System.Threading` także obejmuje typ `Timer`, a więc użycie w kodzie samej nazwy `Timer` byłoby wieloznaczne.

Listing 5.10. Deklarowanie aliasu typu

```
using System;
using System.Threading;
using CountDownTimer = System.Timers.Timer;

class HelloWorld
{
    static void Main()
    {
        CountDownTimer timer;
        // ...
    }
}
```

Na listingu 5.10 jako alias zastosowano zupełnie nową nazwę, CountDownTimer. Można jednak wykorzystać jako alias nazwę Timer, tak jak zrobiono to na listingu 5.11.

Listing 5.11. Deklarowanie aliasu typu za pomocą nazwy tego typu

```
using System;
using System.Threading;

// Deklarowanie aliasu Timer reprezentującego typ System.Timers.Timer, by
// uniknąć konfliktu z nazwą System.Threading.Timer.
using Timer = System.Timers.Timer;

class HelloWorld
{
    static void Main()
    {
        Timer;
        // ...
    }
}
```

Dzięki dyrektywie z aliasem nazwa Timer nie jest teraz dwuznaczna. Ponadto aby zastosować typ System.Threading.Timer, trzeba albo podać jego pełną nazwę, albo zdefiniować inny alias.

Zwracane wartości i parametry metody Main()

Do tej pory deklaracje metody Main() z pliku wykonywalnego miały najprostszą możliwą postać. Deklaracje tej metody nie obejmowały żadnych parametrów ani typów zwracanej wartości. Jednak C# umożliwia pobieranie argumentów z wiersza poleceń w wykonywanym programie. Metoda Main() może też zwracać kod statusu.

Środowisko uruchomieniowe przekazuje argumenty z wiersza poleceń do metody Main() w jednym parametrze — w tablicy wartości typu string. Aby w kodzie pobrać te parametry, wystarczy uzyskać dostęp do tablicy, co ilustruje listing 5.12. Program z tego listingu pobiera

plik, którego lokalizacja jest podawana za pomocą adresu URL. Pierwszy argument w wierszu polecen określa ten adres, a drugi — nazwę, pod jaką plik zostanie zapisany. Kod rozpoczyna się od instrukcji switch, która sprawdza liczbę parametrów (args.Length) i działa w następujący sposób:

1. Jeśli nie wprowadzono dwóch parametrów, kod wyświetla komunikat o błędzie z informacją, że należy podać adres URL i nazwę pliku.
2. Obecność dwóch argumentów oznacza, że użytkownik podał zarówno adres URL zasobu, jak i nazwę, pod jaką należy zapisać pobrany plik.

Listing 5.12. Przekazywanie argumentów z wiersza polecen do metody Main

```
using System;
using System.IO;
using System.Net.Http;

class Program
{
    static int Main(string[] args)
    {
        int result;
        switch (args.Length)
        {
            default:
                // Należy podać dokładnie dwa argumenty. W przeciwnym razie kod zgłasza błąd.
                Console.WriteLine(
                    "BŁĄD: należy podać adres URL "
                    + "i nazwę pliku.");
                Console.WriteLine(
                    "Użycowanie: Downloader.exe <URL> <nazwa_docelowego_pliku>");
                result = 1;
                break;
            case 2:
                WebClient webClient = new WebClient();
                webClient.DownloadFile(args[0], args[1]);
                result = 0;
                break;
        }
        return result;
    }
}
```

Wyniki działania kodu z listingu 5.12 pokazano w danych wyjściowych 5.4.

DANE WYJŚCIOWE 5.4.

```
>Downloader.exe
BŁĄD: należy podać adres URL i nazwę pliku.
Downloader.exe <URL> <nazwa_docelowego_pliku>
```

Jeśli kod z powodzeniem określił nazwę docelowego pliku, można wykorzystać ją do zapisania pobranego pliku. W przeciwnym razie należy wyświetlić tekst z pomocą. Oprócz pobierania parametrów metoda Main() zwraca wartość typu int (nie użyto tu typu void).

Jest to opcjonalne w deklaracji metody `Main()`, ale jeśli taka wartość jest używana, program może zwracać do jednostki wywołującej (na przykład skryptu lub pliku wsadowego) kod statusu. Standardowo wartość różna od zera oznacza błąd.

Choć wszystkie argumenty z wiersza poleceń można przekazać do metody `Main()` za pomocą tablicy łańcuchów znaków, czasem wygodne jest umożliwienie dostępu do argumentów także w innych metodach. Metoda `System.Environment.GetCommandLineArgs()` zwraca tablicę argumentów z wiersza poleceń mającą taką samą postać, w jakiej deklaracja `Main(string[] args)` przekazuje argumenty do metody `Main()`.

■ ZAGADNIENIE DLA ZAAWANSOWANYCH

Odróżnianie od siebie różnych metod `Main()`

Jeśli program obejmuje dwie klasy z metodami `Main()`, można w wierszu poleceń wskazać klasę, z której ma pochodzić uruchamiana metoda. W środowisku Visual Studio kliknij prawym przyciskiem myszy projekt w oknie *Eksplorator rozwiązań* i wybierz opcję *Właściwości*. Nad plikiem projektu pojawi się wtedy interfejs użytkownika. Po wybraniu zakładki *Aplikacja* możesz zmodyfikować obiekt uruchomieniowy i wskazać typ, z którego metoda `Main()` ma rozpoczynać pracę programu. Tę samą wartość można też podać w wierszu poleceń, ustawiając właściwość `StartupObject` w trakcie budowania programu. Oto przykład:

```
dotnet build /p:StartupObject=AddisonWesley.Program2
```

Tu `AddisonWesley.Program2` to przestrzeń nazw i klasa zawierająca wybraną metodę `Main()`.

■ ZAGADNIENIE DLA POCZĄTKUJĄCYCH

Stos wywołań i miejsce wywołania

Gdy kod jest wykonywany, metody wywołują inne metody, te uruchamiają jeszcze następne metody itd. W prostym kodzie z listingu 5.4 metoda `Main()` wywołuje metodę `GetUserInput()`, ta uruchamia metodę `System.Console.ReadLine()`, która z kolei wywołuje wewnętrznie jeszcze inne metody. Przy każdym wywołaniu nowej metody środowisko uruchomieniowe tworzy *zapis aktywacji* z informacjami o argumentach przekazanych do nowego wywołania, zmiennych lokalnych tego wywołania oraz o tym, gdzie należy przekazać sterowanie po zwróceniu go przez wywoływaną metodę. Zestaw sekwencji wywołań z serią zapisów aktywacji to **stos wywołań**³. Gdy złożoność programu rośnie, razem z wywołaniami kolejnych metod zwiększa się także stos wywołań. Jednak po zakończeniu wykonywania metody stos maleje do momentu wywołania następnej metody. Proces usuwania zapisów aktywacji ze stosu wywołań jest nazywany **odwijaniem stosu**. Ten proces zawsze odbywa się w kolejności odwrotnej względem wywołań metod. Gdy metoda kończy pracę, sterowanie wraca do **miejsca wywołania**, czyli do lokalizacji, w której dana metoda została uruchomiona.

³ Wyjątkiem są metody asynchroniczne lub metody iteratora. Dla takich metod zapisy aktywacji są umieszczane na stercie.

Zaawansowane parametry metod

Kod prezentowany do tego miejsca zwracał dane w tradycyjny sposób, za pomocą instrukcji return. W tym podrozdziale pokazano, że metody mogą też zwracać dane przy użyciu parametrów. Poznasz tu też metody przyjmujące zmienną liczbę parametrów.

Parametry przekazywane przez wartość

Argumenty w wywołaniach metod są zwykle **przekazywane przez wartość**. To oznacza, że wartość wyrażenia podanego jako argument jest kopiwana do docelowego parametru. Na przykład na listingu 5.13 wartość każdej zmiennej używanej przez metodę Main() w wywołaniu metody Combine() jest kopiwana do parametrów tej ostatniej. Wynik działania kodu z tego listingu pokazano w danych wyjściowych 5.5.

Listing 5.13. Przekazywanie zmiennych przez wartość

```
class Program
{
    static void Main()
    {
        // ...
        string fullName;
        string driveLetter = "C:";
        string folderPath = "Data";
        string fileName = "index.html";

        fullName = Combine(driveLetter, folderPath, fileName);

        Console.WriteLine(fullName);
        // ...
    }

    static string Combine(
        string driveLetter, string folderPath, string fileName)
    {
        string path;
        path = string.Format("{0}{1}{2}{3}",
            System.IO.Path.DirectorySeparatorChar,
            driveLetter, folderPath, fileName);
        return path;
    }
}
```

DANE WYJŚCIOWE 5.5.

C:\Data\index.html

Nawet jeśli metoda Combine() przed zwróceniem sterowania przypisze wartość null do zmiennych driveLetter, folderPath i fileName, odpowiadające im zmienne w metodzie Main() zachowają pierwotne wartości, ponieważ w momencie wywołania metody zmienne są kopowane. Gdy stos wywołań jest odwijany na końcu wywołania, skopiowane dane są usuwane.

ZAGADNIENIE DLA POCZĄTKUJĄCYCH

Dopasowywanie zmiennych z jednostki wywołującej do nazw parametrów

Na listingu 5.13 nazwy zmiennych w jednostce wywołującej są takie same jak nazwy parametrów w wywoływanej metodzie. To dopasowanie ma jedynie zwiększać czytelność kodu. Nazwy przekazywanych zmiennych nie mają żadnego wpływu na wywołanie metody. Parametry wywoływanej metody i zmienne lokalne metody wywołującej znajdują się w różnych przestrzeniach deklaracji i nie mają ze sobą nic wspólnego.

ZAGADNIENIE DLA ZAAWANSOWANYCH

Typy referencyjne a typy bezpośrednie

W tym punkcie nie ma znaczenia, czy przekazywane parametry są typu bezpośredniego, czy referencyjnego. Istotne jest to, czy wywoływana metoda może zapisywać wartość w pierwotnej zmiennej z jednostki wywołującej. Ponieważ tworzona jest kopia wartości zmiennej z jednostki wywołującej, w wywołanej metodzie nie można zmienić wartości pierwotnej zmiennej. Mimo to warto zrozumieć różnicę między zmiennymi typu bezpośredniego a zmiennymi typu referencyjnego.

Wartością zmiennej typu referencyjnego jest, jak wskazuje na to nazwa, referencja prowadząca do miejsca, w którym przechowywane są dane powiązane z obiektem. To, w jaki sposób środowisko uruchomieniowe reprezentuje wartość zmiennej typu referencyjnego, zależy od implementacji danego środowiska. Zwykle reprezentacją wartości jest adres określający lokalizację danych obiektu w pamięci, jednak nie musi tak być.

Jeśli zmienna typu referencyjnego jest przekazywana przez wartość, do parametru metody kopiwana jest z jednostki wywołującej sama referencja. Dlatego wywołana metoda nie może zaktualizować wartości zmiennej z jednostki wywołującej, natomiast może zmodyfikować dane, do których prowadzi referencja.

Jeśli parametr metody to zmienna typu bezpośredniego, do parametru kopiwana jest sama wartość. Zmiana wartości parametru w wywołanej metodzie nie wpływa wtedy na pierwotną zmienną z jednostki wywołującej.

Parametry przekazywane przez referencję (z modyfikatorem ref)

Przyjrzyj się listingu 5.14, zawierającemu wywołanie funkcji, która zamienia dwie wartości miejscami. Wynik działania kodu pokazano w danych wyjściowych 5.6.

Listing 5.14. Przekazywanie zmiennych przez referencję

```
class Program
{
    static void Main()
    {
        // ...
        string first = "Witaj";
        string second = "Żegnaj";
        Swap(ref first, ref second);
```

```

Console.WriteLine(
    $"{first} = \"{ first }\", second = \"{ second }\" );
// ...
}

static void Swap(ref string x, ref string y)
{
    string temp = x;
    x = y;
    y = temp;
}

```

DANE WYJŚCIOWE 5.6.

```
first = "Żegnaj", second = "Witaj"
```

Wartości przypisane do zmiennych `first` i `second` są z powodzeniem przestawiane. Aby było to możliwe, zmienne trzeba **przekazać przez referencję**. Oczywistą różnicą między wywołaniem metody `Swap()` a wywołaniem metody `Combine()` z listingu 5.13 jest słowo kluczowe `ref` przed typem danych parametru. To słowo kluczowe zmienia wywołanie w taki sposób, że zmienne używane jako argumenty są przekazywane przez referencję. Dlatego wywołana metoda może zmienić wartości pierwotnych zmiennych z jednostki wywołującej.

Gdy w wywoływanej metodzie parametr ma modyfikator `ref`, jednostka wywołująca musi jako argument przekazać zmienną, a nie wartość. Ponadto przed przekazywaną zmienną trzeba dodać słowo `ref`. W ten sposób w jednostce wywołującej można jawnie zaznaczyć, że docelowa metoda może zmienić wartości zmiennych powiązanych z przyjmowanymi parametrami z modyfikatorem `ref`. Ponadto konieczne jest zainicjowanie zmiennych lokalnych przekazywanych z modyfikatorem `ref`, ponieważ jeśli o tym zapomnisz, docelowa metoda może wczytać dane z takich parametrów przed przypisaniem do nich wartości. Na przykład na listingu 5.14 do zmiennej `temp` przypisywana jest wartość zmiennej `first`. Wymaga to przyjęcia założenia, że zmienna przekazana za pomocą parametru `first` została zainicjowana przez jednostkę wywołującą. Parametr z modyfikatorem `ref` działa więc jak alias przekazanej zmiennej. Kod nadaje wtedy nazwę parametru istniejącej zmiennej, zamiast tworzyć nową zmienną i kopiować do niej wartość argumentu.

Parametry wyjściowe (z modyfikatorem `out`)

Początek
7.0

Wcześniej wspomniano, że do zmiennej używanej jako parametr z modyfikatorem `ref` trzeba przypisać wartość przed przekazaniem do wywoływanej metody. Jest tak, ponieważ metoda może odczytywać wartość tej zmiennej. W przykładowej metodzie `swap` kod musi odczytywać i zapisywać wartość obu przekazanych zmiennych. Często jest jednak tak, że metoda przyjmująca referencję do zmiennej ma zapisywać wartość, ale jej nie wczytuje. Wtedy przekazywanie przez referencję niezainicjowanej zmiennej lokalnej powinno być bezpieczne.

W takich sytuacjach w kodzie należy dodać do typu parametru słowo kluczowe `out`. Tę technikę zastosowano w metodzie `TryGetPhoneNumber()` na listingu 5.15. Metoda ta zwraca numer przycisku telefonu na podstawie otrzymanego znaku.

Listing 5.15. Przekazywanie samych zmiennych z modyfikatorem out

```

class ConvertToPhoneNumber
{
    static int Main(string[] args)
    {
        if (args.Length == 0)
        {
            Console.WriteLine(
                "ConvertToPhoneNumber.exe <znaki>");
            Console.WriteLine(
                "'-' oznacza, że nie podano standardowego przycisku telefonu");
            return 1;
        }
        foreach (string word in args)
        {
            foreach (char character in word)
            {
                if (TryGetPhoneButton(character, out char button))
                {
                    Console.Write(button);
                }
                else
                {
                    Console.Write('_');
                }
            }
        }
        Console.WriteLine();
        return 0;
    }

    static bool TryGetPhoneButton(char character, out char button)
    {
        bool success = true;
        switch( char.ToLower(character) )
        {
            case '1':
                button = '1';
                break;
            case '2': case 'a': case 'b': case 'c':
                button = '2';
                break;
            // ...
            case '-':
                button = '-';
                break;
            default:
                // Przypisywanie do zmiennej button wartości oznaczającej nieprawidłowy znak
                button = '_';
                success = false;
                break;
        }
        return success;
    }
}

```

Wynik działania kodu z listingu 5.15 pokazano w danych wyjściowych 5.7.

DANE WYJŚCIOWE 5.7.

```
>ConvertToStringNumber.exe CSharpIsGood  
274277474663
```

W tym przykładzie metoda TryGetPhoneButton() zwraca wartość true, jeśli potrafi z powodzeniem ustalić przycisk telefonu odpowiadający parametrowi character. Metoda zwraca też ustalony przycisk za pomocą parametru button (jest to parametr z modyfikatorem out).

Parametry out działają bardzo podobnie do parametrów ref. Jedyna różnica dotyczy wymagań języka związanych z odczytem i zapisem wartości zmiennej, dla której utworzono alias. Gdy parametr ma modyfikator out, kompilator sprawdza, czy wartość parametru jest ustawiana we wszystkich ścieżkach wykonania w metodzie, które w normalny sposób zwracają sterowanie (bez zgłaszania wyjątku). Jeśli kod w którejś ze ścieżek wykonania nie przypisze wartości do zmiennej button, kompilator zgłosi błąd informujący, że nie zainicjowano tej zmiennej. Na listingu 5.15 do zmiennej button przypisywany jest znak podkreślenia, ponieważ nawet gdy nie można ustalić prawidłowego przycisku telefonu, konieczne jest ustawienie wartości tej zmiennej.

Błędem często popełnianym w trakcie używania parametrów out jest brak deklaracji zmiennej out przed jej zastosowaniem. Od wersji C# 7.0 zmienne out można deklarować wewnętrznie, w miejscu wywołania funkcji. Na listingu 5.15 wykorzystano tę możliwość w instrukcji TryGetPhoneButton(character, out char button) bez wcześniejszego deklarowania zmiennej button. W wersjach starszych od C# 7.0 trzeba było najpierw zadeklarować zmienną button, a dopiero potem można było wywołać funkcję za pomocą instrukcji TryGetPhoneButton(character, out button).

Inną techniką z wersji C# 7.0 jest możliwość całkowitego pominięcia parametru out. Jeśli chcesz na przykład sprawdzić, czy dany znak jest poprawnym klawiszem telefonu, ale nie planujesz zwracać wartości liczbowej, możesz pominąć parametr button i posłużyć się znakiem podkreślenia: TryGetPhoneButton(character, out _).

Przed wprowadzeniem (w C# 7.0) składni dla krotek programista mógł zadeklarować jeden lub kilka parametrów z modyfikatorem out, by obejść ograniczenie, zgodnie z którym w metodzie można określić tylko jedną zwracaną wartość. Jeśli takie wartości mają być dwie, jedną z nich można zwrócić normalnie (za pomocą instrukcji return), a drugą w wyniku zapisania wartości w zmiennej z aliasem, przekazanej jako parametr z modyfikatorem out. Choć ta technika jest często spotykana i poprawna, zwykle istnieją lepsze sposoby na osiągnięcie celu. Jeżeli na przykład chcesz zwracać z metody dwie lub większą liczbę wartości i korzystasz z wersji C# 7.0, przeważnie lepiej jest zastosować składnię dla krotek. Wcześniej mogłeś rozważyć utworzenie dwóch metod, po jednej dla każdej wartości, lub użycie typu System.ValueType (co wymaga dodania pakietu NuGet i typu System.ValueType), ale bez składni z wersji C# 7.0.

7.0

Uwaga

Każda zwykła ścieżka wykonania musi obejmować przypisanie wartości do wszystkich parametrów z modyfikatorem out.

Przekazywanie przez referencję w trybie tylko do odczytu (in)

W wersji C# 7.2 dodano obsługę przekazywania wartości typów bezpośrednich przez referencję w trybie tylko do odczytu. Obok przekazywania takich wartości do funkcji z możliwością modyfikowania danych dodano przekazywanie przez referencję w trybie tylko do odczytu. W tej technice wartość nie jest kopiowana, a wywołana metoda nie może zmieniać otrzymanej wartości. Celem tego mechanizmu jest zmniejszenie ilości pamięci kopowanej w momencie przekazywania wartości. Ta pamięć jest jednocześnie oznaczana jako przeznaczona tylko do odczytu, co poprawia wydajność. Składnia polega na dodaniu do parametru modyfikatora `in`. Oto przykład:

```
int Method(in int number) { ... }
```

Dzięki modyfikatorowi `in` próby ponownego przypisania wartości do zmiennej `number` (np. `number++`) spowodują błąd komplikacji i wyświetlenie informacji, że zmienna służy tylko do odczytu.

Zwracanie przez referencję

Innym dodatkiem w C# 7.0 jest umożliwienie zwracania referencji do zmiennych. Przyjrzyj się przykładowej funkcji, która zwraca pierwszy piksel obrazu występujący w czerwonych oczach na zdjęciu (listing 5.16).

Listing 5.16. Zwracanie i lokalne deklarowanie referencji

```
// Zwracanie referencji.
public static ref byte FindFirstRedEyePixel(byte[] image)
{
    // Zaawansowane wykrywanie cech obrazu, możliwe, że z użyciem uczenia maszynowego.
    for (int counter = 0; counter < image.Length; counter++)
    {
        if (image[counter] == (byte)ConsoleColor.Red)
        {
            return ref image[counter];
        }
    }
    throw new InvalidOperationException("Nie występują czerwone piksele.");
}
public static void Main()
{
    byte[] image = new byte[254];
    // Wczytywanie obrazu.
    int index = new Random().Next(0, image.Length - 1);
    image[index] =
        (byte)ConsoleColor.Red;
    System.Console.WriteLine(
        $"image[{index}]={(ConsoleColor)image[index]}");
    // ...
}

// Pobieranie referencji do pierwszego czerwonego piksela.
ref byte redPixel = ref FindFirstRedEyePixel(image);
// Zmienianie koloru na czarny.
redPixel = (byte)ConsoleColor.Black;
```

```
System.Console.WriteLine(
    $"image[{index}]={(ConsoleColor)image[redPixel]}");
}
```

Dzięki zwracaniu referencji do zmiennej jednostka wywołująca może zmodyfikować kolor piksela, co ilustruje wyróżniony wiersz z listingu 5.16. Jeśli sprawdzisz wartość w tablicy, zobaczyysz, że piksel po zmianie jest czarny.

Zwracanie przez referencję związane jest z dwoma ważnymi ograniczeniami. Oba wynikają z czasu życia obiektu. Po pierwsze, mechanizm przywracania pamięci nie powinien odzyskiwać pamięci zajmowanej przez obiekt, dopóki istnieją referencje do tego obiektu. Po drugie, obiekt nie powinien zajmować pamięci, jeśli nie prowadzą do niego żadne referencje. Aby wymusić przestrzeganie tych ograniczeń, w funkcji można zwracać tylko następujące rodzaje referencji:

- referencje do pól lub elementów tablic;
- właściwości lub funkcje zwracające referencje;
- referencje przekazane jako parametry do funkcji zwracającej wartość przez referencję.

Na przykład funkcja `FindFirstRedEyePixel()` zwraca referencję do elementu z tablicy `image` przekazanej jako parametr tej funkcji. Podobnie jeśli tablica `image` zostanie zapisana jako pole klasy, można zwrócić to pole przez referencję:

```
byte[] _Image;
public ref byte[] Image { get { return ref _Image; } }
```

Ponadto referencje lokalne są inicjowane w taki sposób, aby wskazywały określona zmenną; nie można ich zmodyfikować, aby prowadziły do innej zmiennej.

Mechanizm zwracania przez referencję ma kilka cech, o których trzeba pamiętać:

- Jeśli zwracasz referencję, koniecznie musisz wykonać tę operację. Na listingu 5.16 oznacza to, że nawet jeśli na zdjęciu nie ma czerwonych pikseli, trzeba zwrócić bajt z referencją. Jedyna inna możliwość to zgłoszenie wyjątku. Z kolei gdy przekazujesz parametr przez referencję, możesz pozostawić ten parametr niezmieniony i zwracać wartość typu `bool` oznaczającą powodzenie. W wielu sytuacjach jest to preferowane rozwiązanie.
- Gdy deklarujesz lokalną zmienną referencyjną, trzeba ją zainicjować. Możesz w tym celu przypisać do niej referencję zwróconą przez funkcję lub referencję do zmiennej:

7.0

```
ref string text; // Błąd.
```

- Choć w C# 7.0 można zadeklarować lokalną zmienną referencyjną, deklarowanie pola typu `ref` jest niedozwolone:

```
class Thing { ref string _Text; /* Błąd. */ }
```

- Nie można zadeklarować typu przekazywanego przez referencję dla automatycznie implementowanej właściwości:

```
class Thing { ref string Text { get;set; } /* Błąd. */ }
```

- Dozwolone są właściwości zwracające referencje:

```
class Thing { string _Text = "Inigo Montoya";
    ref string Text { get { return ref _Text; } } }
```

- Lokalnej zmiennej referencyjnej nie można zainicjować za pomocą wartości (np. null lub stałej). Trzeba zastosować składową zwracającą wartość przez referencję, zmienną lokalną, pole lub element tablicy:

```
ref int number = 42; // BŁĄD.
```

Koniec
7.0

Tablice parametrów (params)

W przykładach prezentowanych do tej pory liczba przekazywanych argumentów była stała i zależała od liczby parametrów określonych w deklaracji wywoływanej metody. Jednak czasem wygodnie jest zachować możliwość zmiany liczby argumentów. Wróć do metody `Combine()` z listingu 5.13. Należy do niej przekazać literę napędu, ścieżkę do katalogu i nazwę pliku. Co zrobić, jeśli w ścieżce występuje więcej niż jeden katalog, a programista chce, by metoda scalała dodatkowe katalogi w pełną ścieżkę? Prawdopodobnie najlepszym rozwiązaniem jest przekazanie tablicy łańcuchów znaków reprezentujących te katalogi. Jednak to sprawia, że wywołanie staje się bardziej skomplikowane. Konieczne jest wtedy utworzenie tablicy w celu przekazania argumentu.

Aby ułatwić wywoływanie takich metod, w języku C# udostępniono słowo kluczowe, które umożliwia zmianę liczby argumentów w wywołaniu. Nie trzeba podawać tej liczby w docełowej metodzie. Przed omówieniem sposobu deklarowania takich metod przyjrzyj się wywołaniu w metodzie `Main()` na listingu 5.17.

Listing 5.17. Przekazywanie listy o zmiennej liczbie parametrów

```
using System;
using System.IO;
class PathEx
{
    static void Main()
    {
        string fullName;
        // ...
        // Wywołanie metody Combine() z czterema argumentami.
        fullName = Combine(
            Directory.GetCurrentDirectory(),
            "bin", "config", "index.html");
        Console.WriteLine(fullName);
        // ...
        // Wywołanie metody Combine() z tylko trzema argumentami.
        fullName = Combine(
            Environment.SystemDirectory,
            "Temp", "index.html");
        Console.WriteLine(fullName);
    }
}
```

```
// ...
// Wywołanie metody Combine() z argumentem w postaci tablicy.
fullName = Combine(
    new string[] {
        "C:\\\\", "Data",
        "HomeDir", "index.html" } );
Console.WriteLine(fullName);
// ...
}

static string Combine(params string[] paths)
{
    string result = string.Empty;
    foreach (string path in paths)
    {
        result = System.IO.Path.Combine(result, path);
    }
    return result;
}
```

Wynik działania kodu z listingu 5.17 pokazano w danych wyjściowych 5.8.

DANE WYJŚCIOWE 5.8.

```
C:\Data\mark\bin\config\index.html
C:\WINDOWS\system32\Temp\index.html
C:\Data\HomeDir\index.html
```

W pierwszym wywołaniu metody `Combine()` podane są cztery argumenty. Drugie wywołanie obejmuje tylko trzy. W ostatnim wywołaniu przekazywany jest pojedynczy argument w postaci tablicy. Oznacza to, że metoda `Combine()` przyjmuje zmienną liczbę argumentów. Można przekazać albo dowolną liczbę rozdzielonych przecinkami łańcuchów znaków, albo jedną tablicę łańcuchów znaków. Pierwsza składnia to *rozwinięta* postać wywoływania metody, a druga jest nazywana postacią *normalną*.

Aby możliwe było stosowanie opisanej składni, w metodzie `Combine()` trzeba zastosować następujące rozwiązania:

1. Umieścić słowo `params` bezpośrednio przed ostatnim parametrem w deklaracji metody.
2. Zadeklarować ostatni parametr jako tablicę.

Dzięki deklaracji **tablicy parametrów** dostęp do każdego argumentu można uzyskać w taki sposób, jakby argumenty były elementami tablicy z modyfikatorem `params`. W metodzie `Combine()` kod dla każdego elementu tablicy `paths` wywołuje metodę `System.IO.Path.Combine()`. Ta ostatnia metoda automatycznie łączy członły ścieżki i dodaje znak separatora katalogów odpowiedni dla używanego systemu. Zauważ, że metoda `PathEx.Combine()` działa prawie identycznie jak metoda `Path.Combine()`. Różnica polega na tym, że metoda `PathEx.Combine()` obsługuje zmienną liczbę parametrów (zamiast tylko dwóch).

Tablice parametrów mają kilka istotnych cech:

- Tablica parametrów nie musi być jedynym parametrem metody.
- Tablicę parametrów trzeba podać jako ostatni parametr w deklaracji metody. Ponieważ tylko ostatni parametr może być tablicą parametrów, metoda nie może mieć więcej niż jednej takiej tablicy.
- W jednostce wywołujączej można przekazać zero argumentów odpowiadających tablicy parametrów. W takiej sytuacji jako tablica parametrów przekazywana jest tablica bez elementów.
- Tablice parametrów są bezpieczne ze względu na typ. Przekazywane argumenty muszą mieć typ zgodny z typem elementów tablicy parametrów.
- W jednostce wywołujączej można jawnie utworzyć tablicę, zamiast przekazywać listę parametrów rozdzielonych przecinkami. Wynikowy kod CIL jest w obu sytuacjach identyczny.
- Jeśli w kodzie wywoływanej metody potrzebna jest minimalna liczba parametrów, powinny się one pojawiać jawnie w deklaracji metody. Wymusza to zgłoszenie błędu kompilatora, dzięki czemu nie trzeba polegać na obsłudze błędów czasu wykonania, gdy wymagane parametry nie są dostępne. Na przykład jeśli metoda wymaga jednego lub więcej argumentów całkowitoliczbowych, zadeklaruj ją tak: `int Max(int first, params int[] operands)`, a nie tak: `Max(params int[] operands)`. Dzięki temu do metody `Max()` przekazana zostanie przynajmniej jedna wartość.

Z pomocą tablicy parametrów można przekazywać do metody zmienną liczbę argumentów tego samego typu. W podrozdziale „Przeciążanie metod” w dalszej części rozdziału opisano, jak zapewnić obsługę zmiennej liczby parametrów, które mogą być różnych typów.

Wskazówka

STOSUJ tablice parametrów, gdy metoda może przyjmować dowolną liczbę (także zero) dodatkowych argumentów.

Przy okazji warto zauważyć, że funkcja `Combine()` jest tu nieco naciąganym przykładem, ponieważ istnieje przeciążona wersja funkcji `System.IO.Path.Combine()` przyjmująca tablicę parametrów.

Rekurencja

Rekurencyjne wywoływanie metody lub implementowanie metody z wykorzystaniem **rekurencji** polega na wywoływaniu metody przez nią samą. Rekurencja jest czasem najprostszym sposobem na napisanie określonego algorytmu. Kod z listingu 5.18 zlicza wiersze wszystkich plików z kodem źródłowym w języku C# (plików *.cs) z katalogu i jego podkatalogów.

Listing 5.18. Zliczanie wierszy z plików *.cs na podstawie katalogu

```
#nullable enable
using System.IO;

public static class LineCounter
{
    // Pierwszy argument (jeśli jest podany) określa przeszukiwany katalog.
    // Domyślnie używany jest katalog bieżący.
    public static void Main(string[] args)
    {
        int totalLineCount = 0;
        string directory;
        if (args.Length > 0)
        {
            directory = args[0];
        }
        else
        {
            directory = Directory.GetCurrentDirectory();
        }
        totalLineCount = DirectoryCountLines(directory);
        System.Console.WriteLine(totalLineCount);
    }

    static int DirectoryCountLines(string directory)
    {
        int lineCount = 0;
        foreach (string file in
            Directory.GetFiles(directory, "*.cs"))
        {
            lineCount += CountLines(file);

        foreach (string subdirectory in
            Directory.GetDirectories(directory))
        {
            lineCount += DirectoryCountLines(subdirectory);
        }

        return lineCount;
    }

    private static int CountLines(string file)
    {
        string? line;
        int lineCount = 0;
        FileStream stream =
            new FileStream(file, FileMode.Open);4
        StreamReader reader = new StreamReader(stream);
        line = reader.ReadLine();
    }
}
```

⁴ Ten kod można poprawić za pomocą instrukcji `using`, którą jednak pominięto, ponieważ nie została ona jeszcze opisana.

```

while (line is object)
{
    if (line.Trim() != "")
    {
        lineCount++;
    }
    line = reader.ReadLine();
}

reader.Close(); // Automatycznie zamyka strumień.
return lineCount;
}
}

```

Wynik działania kodu z listingu 5.18 pokazano w danych wyjściowych 5.9.

DANE WYJŚCIOWE 5.9.

104

Program najpierw przekazuje pierwszy argument z wiersza poleceń do metody `DirectoryCountLines()`. Jeśli nie podano żadnego argumentu, przekazywany jest bieżący katalog. Wspomniana metoda przetwarza każdy plik z podanego katalogu i sumuje liczbę wierszy kodu z wszystkich plików. Po przetworzeniu wszystkich plików z katalogu kod przechodzi do wszystkich podkatalogów. W tym celu podkatalog jest przekazywany do metody `DirectoryCountLines()`, co powoduje ponowne jej uruchomienie dla podanego podkatalogu. Ten sam proces jest powtarzany rekurencyjnie do momentu, w którym do przetworzenia nie pozostaje już żaden katalog.

Dla osób nieznających rekurencji może się to wydawać skomplikowane. Często jednak zastosowanie rekurencji to najprostszy sposób na napisanie kodu (zwłaszcza gdy przetwarzane są dane typu hierarchicznego, na przykład system plików). Choć rekurencja często pozwala uzyskać najbardziej czytelny kod, zwykle nie jest on jednak najbardziej wydajny. Jeśli wydajność jest istotna, programiści powinni pomyśleć o rozwiązaniu innym niż kod oparty na rekurencji. W trakcie dokonywania wyboru trzeba zwykle zadbać o równowagę między czytelnością i wydajnością kodu.

ZAGADNIENIE DLA POCZĄTKUJĄCYCH

Błąd nieskończonej rekurencji

W metodach rekurencyjnych często popełniany jest błąd programistyczny prowadzący do przepełnienia stosu w trakcie wykonywania programu. Zwykle powodem przepełnienia stosu jest **nieskończona rekurencja**, czyli ciągłe wywoływanie metody przez nią samą bez dojścia do punktu oznaczającego zakończenie rekurencji. Dobrą praktyką programistyczną jest sprawdzenie każdej metody, w której wykorzystywana jest rekurencja, i zbadanie, czy rekurencyjne wywołania się kończą.

Poniżej za pomocą pseudokodu pokazano standardowy wzorzec stosowania rekurencji:

```
M(x)
{
    if x jest proste
        zwróć wynik
    else
        a. Wykonaj pewne operacje, by uprościć problem
        b. Rekurencyjnie wywołaj M, by rozwiązać uproszczony problem
        c. Oblicz wynik na podstawie a. i b.
    zwróć wynik
}
```

Problemy pojawiają się, gdy programista nie przestrzega tego wzorca. Możliwe, że zapomina o uproszczeniu problemu lub nie obsługuje wszystkich możliwych najprostszych przypadków. Wtedy rekurencyjne wywołania nigdy się nie kończą.

Przeciążanie metod

Kod z listingu 5.18 wywołuje metodę `DirectoryCountLines()`, zliczającą wiersze w plikach `*.cs`. Jeśli jednak chcesz zliczyć wiersze kodu w plikach `*.h`, `*.cpp` lub `*.vb`, metoda `DirectoryCountLines()` nie zadziała. Potrzebna jest metoda, która przyjmuje rozszerzenie pliku, ale zachowuje definicję obecnej wersji metody, tak by domyślnie obsługiwane były pliki `*.cs`.

Wszystkie metody w klasie muszą mieć unikatowe sygnatury. W języku C# unikatowość można zapewnić przez zmiany w nazwach metod, typach danych parametrów lub liczbie parametrów. Nie wystarczy zmiana typu zwracanej wartości. Jeśli zdefiniujesz dwie metody różniące się tylko typem zwracanej wartości, wystąpi błąd komplikacji. Jest to prawda nawet wtedy, gdy typem zwracanej wartości są dwie różne krotki. **Przeciążanie metod** polega na tworzeniu w klasie dwóch lub więcej metod o tej samej nazwie, ale o różnej liczbie parametrów lub o różnych typach danych parametrów.

Uwaga

Metoda jest uznawana za unikatową, jeśli różni się od innych nazwą, typami danych parametrów lub liczbą parametrów.

Przeciążanie metod to jedna z odmian **polimorfizmu operacyjnego**. Polimorfizm występuje, gdy te same operacje logiczne przyjmują wiele (*poly*) postaci (*morf*) z powodu zmian w danych. Gdy w wywołaniu metody `WriteLine()` przekażesz łańcuch znaków formatowania razem z parametrami, kod zadziała inaczej niż w sytuacji, gdy tę samą metodę wywołasz z przekazaną liczbą całkowitą. Jednak logicznie, z perspektywy jednostki wywołującej, wspomniana metoda odpowiada za wyświetlanie danych. Nie ma znaczenia, jak wykonywane są wewnętrzne operacje tej metody. Przykładowy kod znajdziesz na listingu 5.19, a wynik jego działania pokazano w danych wyjściowych 5.10.

Listing 5.19. Zliczanie wierszy z plików *.cs z wykorzystaniem przeciążania

```
#nullable enable
using System.IO;

public static class LineCounter
{
    public static void Main(string[] args)
    {
        int totalLineCount;

        if (args.Length > 1)
        {
            totalLineCount =
                DirectoryCountLines(args[0], args[1]);
        }
        if (args.Length > 0)
        {
            totalLineCount = DirectoryCountLines(args[0]);
        }
        else
        {
            totalLineCount = DirectoryCountLines();
        }

        System.Console.WriteLine(totalLineCount);
    }

    static int DirectoryCountLines()
    {
        return DirectoryCountLines(
            Directory.GetCurrentDirectory());
    }

    static int DirectoryCountLines(string directory)
    {
        return DirectoryCountLines(directory, "*.cs");
    }

    static int DirectoryCountLines(
        string directory, string extension)
    {
        int lineCount = 0;
        foreach (string file in
            Directory.GetFiles(directory, extension))
        {
            lineCount += CountLines(file);
        }

        foreach (string subdirectory in
            Directory.GetDirectories(directory))
        {
            lineCount += DirectoryCountLines(subdirectory);
        }
        return lineCount;
    }

    private static int CountLines(string file)
```

```

{
    int lineCount = 0;
    string? line;
    FileStream stream =
        new FileStream(file, FileMode.Open);5
    StreamReader reader = new StreamReader(stream);
    line = reader.ReadLine();
    while (line is object)
    {
        if (line.Trim() != "")
        {
            lineCount++;
        }
        line = reader.ReadLine();
    }

    reader.Close(); // Automatycznie zamyka strumień.
    return lineCount;
}
}

```

DANE WYJŚCIOWE 5.10.

```
>LineCounter.exe .\ *.cs
28
```

Przeciążenie metody zapewnia dodatkowe sposoby jej wywoływania. W metodzie Main() pokazano, że metodę DirectoryCountLines() można wywoływać z parametrami określającymi przeszukiwany katalog i rozszerzenie pliku, jak i bez takich parametrów.

Zauważ, że bezparametrowa wersja metody DirectoryCountLines() wywołuje wersję z jednym parametrem (int DirectoryCountLines (string directory)). Jest to wzorzec, który jest często stosowany do pisania metod przeciążonych. Pomysł polega na tym, by programista zaimplementował w jednej metodzie tylko podstawową logikę i wywoływał tę metodę w jej pozostałych przeciążonych wersjach. Jeśli podstawowa logika się zmieni, wystarczy ją zmodyfikować w jednym miejscu — nie trzeba tego robić we wszystkich wersjach. Ten wzorzec jest powszechnie stosowany zwłaszcza wtedy, gdy przeciążanie ma umożliwiać podanie dodatkowych parametrów, których wartości w czasie komplikacji nie są znane (dla tego nie można ich podać za pomocą parametrów opcjonalnych).

Uwaga

Umieszczenie podstawowej logiki w jednej metodzie wywoływanej przez wszystkie pozostałe przeciążone wersje sprawia, że zmiany w implementacji wystarczy wprowadzić w podstawowej metodzie. Modyfikacje zostaną wtedy uwzględnione we wszystkich pozostałych metodach.

⁵ Ten kod można poprawić za pomocą instrukcji using, którą jednak pominięto, ponieważ nie została ona jeszcze opisana.

Parametry opcjonalne

Projektanci języka C# dodali też obsługę **parametrów opcjonalnych**⁶. Jeśli w deklaracji metody powiązesz parametr ze stałą, będziesz mógł wywołać metodę bez przekazywania argumentów odpowiadających tak zadeklarowanym parametrom (zobacz listing 5.20).

Listing 5.20. Metody z parametrami opcjonalnymi

```
#nullable enable
using System.IO;

public static class LineCounter
{
    public static void Main(string[] args)
    {
        int totalLineCount;

        if (args.Length > 1)
        {
            totalLineCount =
                DirectoryCountLines(args[0], args[1]);
        }
        if (args.Length > 0)
        {
            totalLineCount = DirectoryCountLines(args[0]);
        }
        else
        {
            totalLineCount = DirectoryCountLines();
        }

        System.Console.WriteLine(totalLineCount);
    }

    static int DirectoryCountLines()
    {
        // ...
    }

/*
static int DirectoryCountLines(string directory)
{ ... }
*/
```

```
static int DirectoryCountLines(
    string directory, string extension = ".cs")
{
    int lineCount = 0;
    foreach (string file in
        Directory.GetFiles(directory, extension))
    {
        lineCount += CountLines(file);
    }
}
```

⁶ Ten mechanizm pojawił się w wersji C# 4.0.

```
foreach (string subdirectory in
    Directory.GetDirectories(directory))
{
    lineCount += DirectoryCountLines(subdirectory);
}

return lineCount;
}

private static int CountLines(string file)
{
    // ...
}
```

Na listingu 5.20 usunięto deklarację wersji metody `DirectoryCountLines()` przyjmującej jeden parametr (wersja ta znalazła się w komentarzu), ale w metodzie `Main()` nadal znajduje się wywołanie z jednym parametrem. Gdy w wywołaniu nie ma podanej wartości parametru `extension`, używana jest wartość przypisywana do `extension` w deklaracji (tu jest to `*.cs`). Dzięki temu w wywołaniu nie trzeba podawać wartości, a jednocześnie nie jest potrzebna następna wersja metody, która byłaby konieczna bez zastosowania tej techniki. Zauważ, że opcjonalne parametry muszą się pojawiać po parametrach wymaganych (czyli tych, które nie mają wartości domyślnych). Ponadto wartość domyślna musi być stałą możliwą do ustalenia na etapie komilacji, co jest poważnym ograniczeniem. Nie możesz na przykład zadeklarować metody w następujący sposób:

```
DirectoryCountLines(
    string directory = Environment.CurrentDirectory,
    string extension = "*.cs")
```

Powodem jest to, że `Environment.CurrentDirectory` nie jest stałą. Jednak ponieważ `"*.cs"` to stała, C# umożliwia wykorzystanie jej jako wartości domyślnej parametru opcjonalnego.

Wskazówki

PODAWAJ dobre wartości domyślne dla wszystkich parametrów, jeśli jest to możliwe.

UDOSTĘPNIAJ prostą wersję przeciążanej metody, o małej liczbie wymaganych parametrów.

ROZWAŻ uporządkowanie wersji przeciążanej metody od najprostszej do najbardziej złożonej.

Drugą funkcją wywołań metod są **argumenty nazwane**⁷. Za ich pomocą jednostka wywołująca może jawnie wskazać nazwę parametru, którego wartość jest określana. Dzięki temu nie trzeba polegać wyłącznie na kolejności parametrów i argumentów, by je ze sobą powiązać (zobacz listing 5.21).

⁷ Są one dostępne od wersji C# 4.0.

Listing 5.21. Podawanie parametrów za pomocą nazw

```
#nullable enable
using System.IO;

class Program
{
    static void Main()
    {
        DisplayGreeting(
            firstName: "Inigo", lastName: "Montoya");
    }

    public static void DisplayGreeting(
        string firstName,
        string? middleName = null,
        string? lastName = null)
    {
        // ...
    }
}
```

Na listingu 5.21 w wywołaniu metody `DisplayGreeting()` w metodzie `Main()` wartość jest przypisywana do parametru określonego za pomocą nazwy. Spośród dwóch opcjonalnych parametrów (`middleName` i `lastName`) tylko `lastName` jest podany jako argument. Gdy metoda przyjmuje wiele parametrów i liczne z nich są opcjonalne (zdarza się to często w trakcie korzystania z bibliotek COM Microsoftu), używanie składni z nazwanymi parametrami jest dużym udogodnieniem. Jednak wygoda okupiona jest kosztem niższej elastyczności interfejsu metody. W przeszłości zmiana nazw parametrów metody nie skutkowała niemożnością komplikacji kodu w języku C# z jej wywołaniami. Wraz z pojawiением się nazwanych parametrów ich nazwy stały się częścią interfejsu, ponieważ modyfikacja nazwy sprawia, że nie można skompilować kodu używającego danego parametru.

Wskazówka

TRAKTUJ nazwy parametrów jak część interfejsu API. Unikaj zmianiania nazw, jeśli chcesz zachować zgodność między wersjami interfejsu API.

Dla wielu doświadczonych programistów języka C# jest to zaskakujące ograniczenie. Obowiązuje ono jednak w specyfikacji CLS już od wersji .NET 1.0. Ponadto w języku Visual Basic od zawsze możliwe było wywoływanie metod za pomocą nazwanych argumentów. Dlatego autorzy bibliotek już od dawna powinni unikać zmianiania nazw parametrów, aby umożliwić współdziałanie różnych wersji kodu z różnymi językami platformy .NET. Tak więc od wersji C# 4.0 w tym języku obowiązuje to samo ograniczenie w zakresie modyfikowania nazw parametrów, które już wcześniej występowało w wielu innych językach platformy .NET.

Z powodu przeciążania metod, parametrów opcjonalnych i parametrów nazwanych określanie, którą metodę należy wywołać, stało się bardziej skomplikowane. Wywołanie jest **zgodne** (możliwe do zastosowania) z metodą, jeśli dla wszystkich parametrów występuje

dokładnie jeden argument (określony za pomocą nazwy lub pozycji) o odpowiednim typie; nie jest to wymagane dla parametrów opcjonalnych i dla tablic parametrów. Choć ta reguła ogranicza liczbę metod, które mogą zostać wywołane, nie określa ściśle unikatowej metody. Aby dodatkowo ustalić, która metoda zostanie wywołana, kompilator posługuje się tylko parametrami jawnie podanymi w jednostce wywołujączej. Ignoruje wtedy wszystkie opcjonalne parametry, które nie zostały podane. Dlatego jeśli zgodne są dwie metody (ponieważ jedna z nich przyjmuje parametr opcjonalny), kompilator wybiera metodę bez parametru opcjonalnego.

ZAGADNIENIE DLA ZAAWANSOWANYCH

Określanie wywoływanej metody

Gdy kompilator musi ustalić, która z kilku zgodnych metod najlepiej odpowiada danemu wywołaniu, wybiera wersję z najbardziej specyficznymi typami parametrów. Jeśli są dwie zgodne metody, a każda z nich wymaga niejawnej konwersji z typu argumentu na typ parametru, wybierana jest metoda z parametrem, którego typ znajduje się niżej w hierarchii dziedziczenia (jest „bardziej pochodny”).

Na przykład jeśli jednostka wywołująca przekaże argument typu `int`, wybrana zostanie metoda przyjmująca parametr typu `double`, a nie wersja przyjmująca parametr typu `object`. Dzieje się tak, ponieważ typ `double` jest bardziej specyficzny niż typ `object`. Istnieją obiekty, które nie są liczbami zmiennoprzecinkowymi o podwójnej precyzyji, natomiast nie ma takich liczb, które nie są obiektami. Oznacza to, że typ `double` jest bardziej specyficzny.

Jeśli istnieje więcej niż jedna zgodna metoda, ale nie da się jednoznacznie ustalić jej najlepszej wersji, kompilator zgłasza błąd z informacją, że wywołanie nie jest jednoznaczne.

Przyjrzyj się podanym niżej metodom:

```
static void Method(object thing){}
static void Method(double thing){}
static void Method(long thing){}
static void Method(int thing){}
```

Dla wywołania `Method(42)` wybierana jest wersja `Method(int thing)`, ponieważ typ argumentu dokładnie pasuje do typu parametru. Gdyby ta wersja nie istniała, kod wybrałby metodę z parametrem typu `long`, który jest bardziej specyficzny niż typy `double` i `object`.

W specyfikacji języka C# opisane są też dodatkowe reguły dotyczące niejawnej konwersji między typami `byte`, `ushort`, `uint`, `ulong` i innymi typami liczbowymi. Jednak ogólnie lepiej posługiwać się rzutowaniem, by ułatwić wykrycie docelowej metody.

Podstawowa obsługa błędów z wykorzystaniem wyjątków

W tym podrozdziale opisano, jak obsługiwać zgłoszane błędy za pomocą mechanizmu **obsługi wyjątków**. Dzięki obsłudze wyjątków metoda może przekazywać informacje o błędzie do metody wywołującej bez używania zwracanej wartości lub jawnie podanych parametrów. Listing 5.22 zawiera zmodyfikowaną wersję programu `HeyYou` z listingu 1.16 z rozdziału 1. Ta wersja zamiast prosić o nazwisko użytkownika, pobiera jego wiek.

Listing 5.22. Przekształcanie typu wartości ze string na int

```
using System;

class ExceptionHandling
{
    static void Main()
    {
        string firstName;
        string ageText;
        int age;

        Console.WriteLine("Hej, ty!");

        Console.Write("Wprowadź imię: ");
        firstName = System.Console.ReadLine();

        Console.Write("Wprowadź wiek: ");
        ageText = Console.ReadLine();
        age = int.Parse(ageText);

        Console.WriteLine(
            $"Hej, {firstName}! Twój wiek w miesiącach to { age*12 }.");
    }
}
```

Wynik działania kodu z listingu 5.22 pokazano w danych wyjściowych 5.11.

DANE WYJŚCIOWE 5.11.

```
Hej, ty!
Wprowadź imię: Inigo
Wprowadź wiek: 42
Hej, Inigo! Twój wiek w miesiącach to 504.
```

Wartość zwracana przez metodę `System.Console.ReadLine()` jest zapisywana w zmiennej `ageText`, a następnie przekazywana do metody `Parse()` z typu `int`. Ta metoda przyjmuje łańcuch znaków reprezentujący liczbę i przekształca go na typ `int`.

ZAGADNIENIE DLA POCZĄTKUJĄCYCH**Wartość 42 jako łańcuch znaków i jako liczba całkowita**

Język C# wymaga, by każda wartość różna od `null` miała jednoznacznie określony typ. Dlatego ważna jest nie tylko sama wartość, ale też jej typ. Łańcuch znaków 42 to zupełnie inny obiekt niż liczba całkowita 42. Ten łańcuch składa się z dwóch znaków, 4 i 2, natomiast wartość typu `int` to liczba 42.

Końcowa instrukcja `System.Console.WriteLine()` otrzymuje łańcuch znaków przekształcony w liczbę i wyświetla wiek w miesiącach w wyniku pomnożenia liczby lat przez 12.

Co się jednak stanie, jeśli użytkownik nie poda łańcucha znaków z poprawną liczbą całkowitą i wpisze na przykład „czterdzieści dwa”? Metoda `Parse()` nie poradzi sobie z taką konwersją.

Ta metoda wymaga, by użytkownik wprowadziłłańcuch znaków zawierający tylko cyfry. Jeśli metoda Parse() otrzyma nieprawidłową wartość, potrzebuje sposobu na przekazanie informacji o tym do jednostki wywołującej.

Przechwytywanie błędów

Aby poinformować jednostkę wywołującą o błędny parametrze, metoda int.Parse() zgłasza wyjątek. Zgłoszenie wyjątku wstrzymuje dalsze wykonywanie kodu w bieżącej ścieżce przepływu sterowania. Następuje wtedy skok do pierwszego bloku kodu ze stosu wywołań, gdzie można obsłużyć wyjątek.

Ponieważ nie dodales jeszcze obsługi wyjątków, program poinformuje użytkownika o nieobsłużonym wyjątku. Jeśli w systemie nie zarejestrowano debugera, błąd pojawi się w konsoli w postaci komunikatu takiego jak pokazany w danych wyjściowych 5.12.

DANE WYJŚCIOWE 5.12.

```
Hej, ty!  
Wprowadź imię: Inigo  
Wprowadź wiek: czterdzieści dwa  
  
Unhandled Exception: System.FormatException: Input string was  
not in a correct format.  
at System.Number.ParseInt32(String s, NumberStyles style,  
    NumberFormatInfo info)  
at ExceptionHandling.Main()
```

Te informacje o błędzie oczywiście nie są zbyt pomocne. Aby rozwiązać problem, trzeba udostępnić mechanizm obsługi błędu. Ten mechanizm może na przykład przekazywać użytkownikowi bardziej przydatny komunikat o błędzie.

Służy do tego proces **przechwytywania wyjątku**. Składnię tego procesu pokazano na lisingu 5.23, a wyniki działania kodu znajdziesz w danych wyjściowych 5.13.

Listing 5.23. Przechwytywanie wyjątku

```
using System;  
  
class ExceptionHandling  
{  
    static int Main()  
    {  
        string firstName;  
        string ageText;  
        int age;  
        int result = 0;  
  
        Console.Write("Wprowadź imię: ");  
        firstName = Console.ReadLine();  
  
        Console.Write("Wprowadź wiek: ");  
        ageText = Console.ReadLine();  
  
        try  
        {
```

```

        age = int.Parse(ageText);
        Console.WriteLine(
            $"Hej, { firstName }! Twój wiek w miesiącach to { age*12 }.");
    }
    catch (FormatException )
    {
        Console.WriteLine(
            $"Wprowadzony wiek, { ageText }, nie jest prawidłowy.");
        result = 1;
    }
    catch(Exception exception)
    {
        Console.WriteLine(
            $"Nieoczekiwany błąd: { exception.Message }");
        result = 1;
    }
    finally
    {
        Console.WriteLine($"Żegnaj, { firstName }!");
    }
}

return result;
}

```

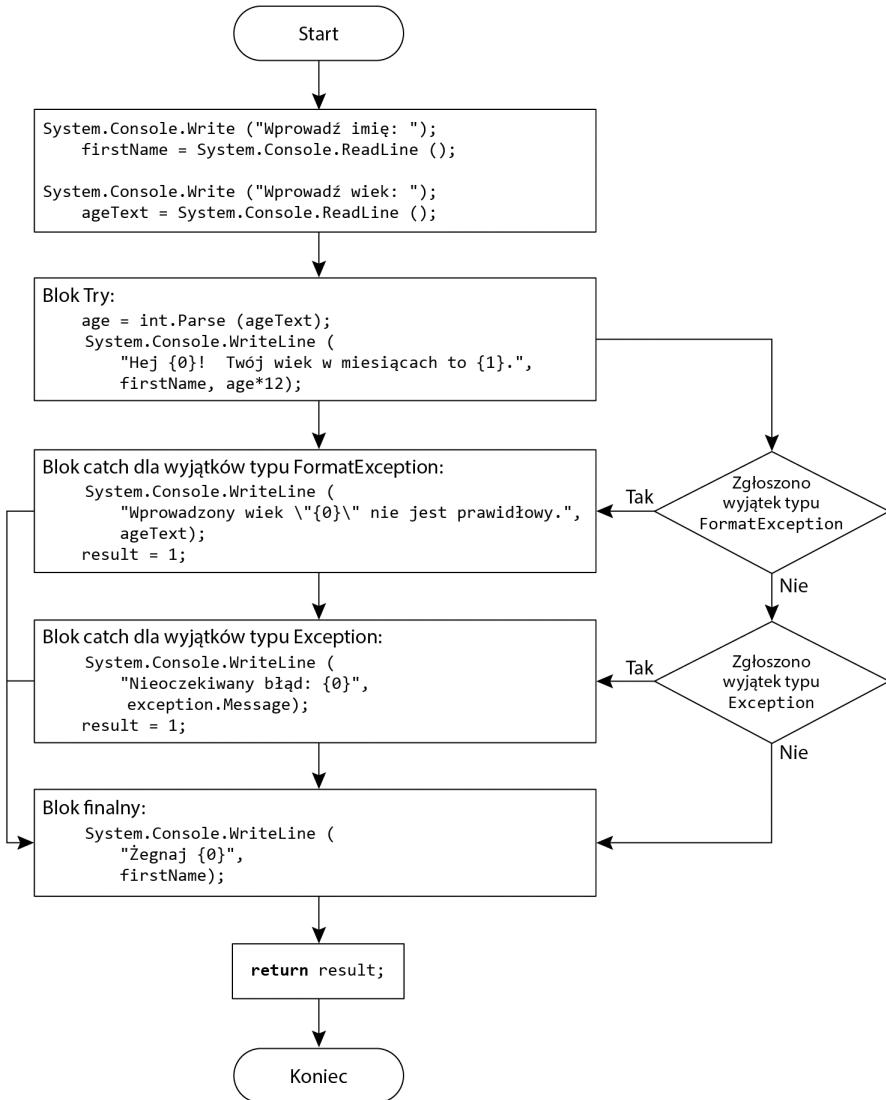
DANE WYJŚCIOWE 5.13.

Wprowadź imię: **Inigo**
 Wprowadź wiek: **czterdzieści dwa**
 Wprowadzony wiek, czterdzieści dwa, nie jest prawidłowy.
 Żegnaj, Inigo!

Najpierw umieść kod, który może zgłaszać wyjątki (`age = int.Parse()`), w **bloku try**. Ten blok rozpoczyna się od słowa kluczowego `try`. Jest to informacja dla kompilatora, że programista wie, iż kod w tym bloku może zgłaszać wyjątki. Jeśli wyjątek zostanie zgłoszony, jeden z **bloków catch** spróbuje go złapnąć.

Bezpośrednio po bloku `try` musi się znajdować przynajmniej jeden blok `catch` (lub `finally`). W nagłówku bloku `catch` (zobacz ZAGADNIENIE DLA ZAAWANSOWANYCH „Ogólny blok catch” w dalszej części rozdziału) opcjonalnie można podać typ danych wyjątków. Jeśli podany typ pasuje do typu zgłoszonego wyjątku, wykonany zostanie odpowiedni blok `catch`. Jeżeli jednak nie istnieje odpowiedni blok `catch`, wyjątek zostanie przekazany dalej i pozostało nieobsłużony — tak jakby w ogóle nie stosowano obsługi wyjątków. Przepływ sterowania związany z wyjątkami pokazano na rysunku 5.1.

Załóżmy, że użytkownik przykładowego programu podał jako wiek tekst „czterdzieści dwa”. Metoda `int.Parse()` zgłasza wtedy wyjątek typu `System.FormatException` (informuje on, że łańcuch znaków nie ma właściwego formatu umożliwiającego poprawne przetworzenie danych), a sterowanie zostaje przekazane do zestawu bloków `catch`. Ponieważ pierwszy blok `catch` pasuje do typu wyjątku zgłoszonego przez metodę `int.Parse()`, wykonywany jest kod z tego bloku. Jeśli instrukcja w bloku `try` zgłosi inny wyjątek, wykonany zostanie kod z drugiego bloku `catch`, ponieważ wszystkie wyjątki są typu `System.Exception`.



Rysunek 5.1. Przepływ sterowania w programie z obsługą wyjątków

Gdyby nie było bloku catch dla wyjątków typu `System.FormatException`, wykonany zostałby blok catch dla wyjątków typu `System.Exception`, choć metoda `int.Parse` zgłasza wyjątek typu `System.FormatException`. Dzieje się tak, ponieważ wyjątek typu `System.FormatException` jest jednocześnie wyjątkiem typu `System.Exception` (`System.FormatException` to bardziej wyspecjalizowana wersja ogólnego typu wyjątków, `System.Exception`).

Kolejność bloków obsługi wyjątków ma znaczenie. Bloki catch trzeba umieszczać od najbardziej do najmniej specyficznego. Typ `System.Exception` jest najmniej specyficzny, dla tego odpowiadający mu blok znajduje się na końcu. Blok dla typu `System.FormatException` pojawia się na początku, ponieważ jest to najbardziej specyficzny typ wyjątków obsługiwany na listingu 5.23.

Niezależnie od tego, czy sterowanie wychodzi z bloku try w zwykły sposób, czy z powodu zgłoszenia wyjątku, po opuszczeniu przez program obszaru chronionego blokiem try zawsze wykonywany jest **blok finally**. W bloku finally należy umieścić kod wykonywany niezależnie od sposobu zakończenia pracy przez bloki try i catch. Nie jest tu istotne, czy wyjątek został zgłoszony. Można też utworzyć blok try z blokiem finally, ale bez bloku catch. Blok finally jest uruchamiany niezależnie od tego, czy blok try zgłosił wyjątek i czy istnieje blok catch przeznaczony do obsługi tego wyjątku. Na listingu 5.24 przedstawiono blok try/finally, a wynik działania kodu pokazano w danych wyjściowych 5.14.

Listing 5.24. Blok finally bez bloku catch

```
using System;

class ExceptionHandling
{
    static int Main()
    {
        string firstName;
        string ageText;
        int age;
        int result = 0;

        Console.WriteLine("Wprowadź imię: ");
        firstName = Console.ReadLine();

        Console.WriteLine("Wprowadź wiek: ");
        ageText = Console.ReadLine();

        try
        {
            age = int.Parse(ageText);
            Console.WriteLine(
                $"Hej, { firstName }! Twój wiek w miesiącach to { age*12 }.");
        }
        finally
        {
            Console.WriteLine($"Żegnaj, { firstName }!");
        }

        return result;
    }
}
```

DANE WYJŚCIOWE 5.14.

```
Wprowadź imię: Inigo
Wprowadź wiek: czterdzieści dwa
```

```
Unhandled Exception: System.FormatException: Input string was not in a correct format.
  at System.Number.StringToNumber(String str, NumberStyles options, NumberBuffer& number,
    ↳NumberFormatInfo info, Boolean parseDecimal)
  at System.Number.ParseInt32(String s, NumberStyles style, NumberFormatInfo info)
  at ExceptionHandling.Main()
Żegnaj, Inigo!
```

Uważni czytelnicy zapewne zwróciли uwagę na ciekawe zjawisko — środowisko uruchomieniowe najpierw zgłosiło nieobsłużony wyjątek, a potem uruchomiło blok `finally`. Jak wyjaśnić to niezwykłe działanie programu?

Jest to poprawne zachowanie środowiska, ponieważ reakcja środowiska uruchomieniowego na nieobsłużony wyjątek jest zdefiniowana przez implementację tego środowiska. Dlatego każde działanie środowiska jest prawidłowe! Środowisko uruchomieniowe wybiera przedstawione rozwiązanie, ponieważ przed wykonaniem bloku `finally` wykrywa, że wyjątek nie zostanie obsłużony. Środowisko na tym etapie zakończyło już sprawdzanie wszystkich zapisów aktywacji ze stosu wywołań i stwierdziło, że żaden z nich nie jest związany z blokiem `catch` pasującym do zgłoszonego wyjątku.

Gdy tylko środowisko uruchomieniowe wykryje, że wyjątek nie zostanie obsłużony, sprawdza, czy na komputerze zainstalowany jest debugger (ponieważ możliwe jest, że kod został uruchomiony przez programistę analizującego dany błąd). Jeśli debugger jest dostępny, środowisko umożliwia użytkownikowi dołączenie debugera do procesu *przed* uruchomieniem bloku `finally`. Jeśli debugger nie jest zainstalowany lub użytkownik nie chce debugować problemu, domyślnie w konsoli wyświetlane są informacje o nieobsłużonym wyjątku, po czym środowisko sprawdza, czy istnieje blok `finally`. Ponieważ działanie tego procesu jest zależne od implementacji, środowisko uruchomieniowe nie musi w opisanej sytuacji uruchamiać bloku `finally`. Decydują o tym autorzy poszczególnych implementacji.

Wskazówki

UNIKAJ jawnego zgłaszania wyjątków w blokach `finally`. Niejawnie zgłoszane wyjątki wynikające z wywołań metod są dopuszczalne.

PRZEDKŁADAJ stosowanie bloków `try/finally` nad bloki `try/catch`, gdy piszesz kod odpowiedzialny za porządkowanie zasobów.

ZGŁASZAJ wyjątki opisujące, jakie nietypowe warunki wystąpiły i (jeśli to możliwe) jak im zapobiec.

ZAGADNIENIE DLA ZAAWANSOWANYCH

Dziedziczenie klas wyjątków

Wszystkie obiekty zgłasiane jako wyjątki są typów pochodnych od typu `System.Exception`⁸. Obiekty zgłaszane z poziomu kodu w innym języku, które nie są typu pochodnego od `System.Exception`, są automatycznie opakowywane w obiekt właściwego typu. Dzięki temu wszystkie wyjątki można obsługiwać za pomocą bloku `catch(System.Exception exception)`. Zaleca się jednak dodawanie bloków `catch` pasujących do najbardziej specyficznego typu (na przykład do typu `System.FormatException`), ponieważ pozwala to uzyskać najwięcej informacji na temat wyjątku i obsłużyć go w ścisłe określony sposób. Dzięki temu instrukcja

⁸ Jest tak od wersji C# 2.0.

catch dopasowana do najbardziej specyficznego typu może w konkretny sposób obsługiwać wyjątek danego typu, korzystając z danych powiązanych ze zgłoszonym wyjątkiem. Nie trzeba wtedy używać warunków do ustalenia, jakiego typu wyjątek wystąpił.

To dlatego w języku C# obowiązuje reguła, zgodnie z którą bloki catch powinny występować od najbardziej do najmniej specyficznego. Na przykład instrukcja catch przechwytyująca wyjątki typu `System.Exception` nie może się znajdować przed instrukcją przechwytyującą wyjątki typu `System.FormatException`, ponieważ `System.FormatException` to typ pochodny od `System.Exception`.

Metody mogą zgłaszać wyjątki różnych typów. W tabeli 5.2 wymieniono typy wyjątków często stosowane w platformie .NET.

Tabela 5.2. Często używane typy wyjątków

Typ wyjątku	Opis
<code>System.Exception</code>	Wyjątek typu bazowego. Wszystkie inne typy wyjątków są pochodne od tego typu.
<code>System.ArgumentException</code>	Informuje, że jeden z argumentów przekazanych do metody jest nieprawidłowy.
<code>System.ArgumentNullException</code>	Informuje, że argument ma wartość <code>null</code> , a nie jest ona poprawna dla danego parametru.
<code>System.ApplicationException</code>	Tego typu należy unikać. Pierwotnie zakładano, że programiści mogą chcieć używać jednego mechanizmu obsługi dla wyjątków systemowych i innego dla wyjątków aplikacji. Choć wydaje się to dobrym pomysłem, w praktyce to rozwiązywanie się nie sprawdza.
<code>System.FormatException</code>	Informuje, że łańcuch znaków ma format niepozwalający na konwersję danych.
<code>System.IndexOutOfRangeException</code>	Informuje, że kod próbował uzyskać dostęp do nieistniejącego elementu tablicy lub innej kolekcji.
<code>System.InvalidCastException</code>	Informuje, że podjęta próba konwersji z jednego typu danych na inny jest niedozwolona.
<code>System.InvalidOperationException</code>	Informuje, że wystąpił nieoczekiwany scenariusz i aplikacja znajduje się w nieprawidłowym stanie.
<code>System.NotImplementedException</code>	Informuje, że choć sygnatura metody istnieje, sama metoda nie została w pełni zaimplementowana.
<code>System.NullReferenceException</code>	Zgłaszały, gdy kod próbuje znaleźć obiekt wskazywany przez referencję o wartości <code>null</code> .
<code>System.ArithmeticException</code>	Informuje o nieprawidłowej operacji matematycznej (innej niż dzielenie przez zero).
<code>System.ArrayTypeMismatchException</code>	Występuje przy próbie zapisania w tablicy elementu o niewłaściwym typie.
<code>System.StackOverflowException</code>	Informuje o nieoczekiwanie dużej liczbie poziomów rekurencji.

ZAGADNIENIE DLA ZAAWANSOWANYCH

Ogólny blok catch

Możliwe jest utworzenie bloku catch bez parametrów, tak jak na listingu 5.25.

Listing 5.25. Ogólne bloki catch

```
// Wcześniejsza klauzula catch przechwytuje wszystkie wyjątki
#pragma warning disable CS1058

...
try
{
    age = int.Parse(ageText);
    System.Console.WriteLine(
        $"Hej, { firstName }! Twój wiek w miesiącach to { age*12 }.");
}
catch (System.FormatException exception)
{
    System.Console.WriteLine(
        $"Wprowadzony wiek, { ageText }, jest nieprawidłowy.");
    result = 1;
}
catch(System.Exception exception)
{
    System.Console.WriteLine(
        $"Nieoczekiwany błąd: { exception.Message }");
    result = 1;
}
catch
{
    System.Console.WriteLine("Nieoczekiwany błąd!");
    result = 1;
}
finally
{
    System.Console.WriteLine($"Żegnaj, { firstName }!");
}
...
```

Blok catch bez określonego typu danych (czyli **ogólny blok catch**) to odpowiednik bloku catch dla typu danych object — catch(object exception){...}. Dlatego zgłaszanego jest ostrzeżenie z informacją, że blok catch już istnieje. Z tego powodu w kodzie używana jest dyrektywa #pragma warning disable.

Ponieważ wszystkie klasy są (pośrednio lub bezpośrednio) pochodne od klasy object, ogólny blok catch musi znajdować się na końcu.

Ogólne bloki catch stosuje się rzadko, ponieważ nie da się w nich uzyskać żadnych informacji na temat wyjątku. Ponadto C# nie umożliwia zgłaszania wyjątków typu object. Tylko biblioteki napisane w językach takich jak C++ umożliwiają zgłaszanie wyjątków dowolnego typu.

Wskazówki

UNIKAJ stosowania ogólnych bloków catch. Zamiast nich twórz bloki dla typu System.Exception.

UNIKAJ przechwytywania wyjątków, jeśli nie da się określić właściwych działań, które pozwolą go uniknąć. Lepiej pozostawić wyjątek nieobsłużony, niż obsługiwać go w nieprawidłowy sposób.

UNIKAJ przechwytywania i rejestrowania wyjątku przed ponownym zgłoszeniem go. Pozwól na przekazanie wyjątku do miejsca, w którym można go odpowiednio obsłużyć.

Zgłaszanie błędów za pomocą instrukcji throw

Język C# umożliwia programistom zgłaszanie wyjątków w pisany kodzie. Ilustrują to listing 5.26 i dane wyjściowe 5.15.

Listing 5.26. Zgłaszanie wyjątku

```
// Wcześniej klawzula catch przechwytuje wszystkie wyjątki.
using System;
public class ThrowingExceptions
{
    public static void Main()
    {
        try
        {
            Console.WriteLine("Rozpoczęcie wykonywania kodu");

            Console.WriteLine("Zgłoszenie wyjątku");
            throw new Exception("Dowolny wyjątek");
            Console.WriteLine("Koniec wykonywania kodu");
        }
        catch(FormatException exception)
        {
            Console.WriteLine(
                "Zgłoszono wyjątek FormatException");
        }
        catch(Exception exception)
        {
            Console.WriteLine(
                $"Nieoczekiwany błąd: { exception.Message }");
        }
        catch
        {
            Console.WriteLine("Nieoczekiwany błąd!");
        }
        Console.WriteLine(
            "Zamykanie programu");
    }
}
```

DANE WYJŚCIOWE 5.15.

Początek wykonywania kodu
Zgłoszenie wyjątku
Nieoczekiwany błąd: Dowolny wyjątek
Zamykanie programu

Strzałki na listingu 5.26 pokazują, że zgłoszenie wyjątku powoduje przeniesienie sterowania z miejsca zgłoszenia wyjątku do pierwszego bloku catch zgodnego z typem tego wyjątku⁹. W przykładowym kodzie drugi blok catch obsługuje zgłoszony wyjątek i wyświetla komunikat o błędzie. Na listingu 5.26 nie ma bloku finally, dlatego sterowanie przechodzi do instrukcji System.Console.WriteLine() po bloku try/catch.

Aby zgłosić wyjątek, trzeba utworzyć obiekt typu wyjątku. Na listingu 5.26 taki obiekt dworzony jest w wyniku podania słowa kluczowego new i typu wyjątku. Większość typów wyjątków umożliwia podawanie komunikatu w momencie zgłoszania wyjątku. Dlatego po zgłoszeniu wyjątku można pobrać podany komunikat.

Czasem blok catch przechwytuje dany wyjątek, ale nie potrafi go poprawnie lub w pełni obsłużyć. W takich sytuacjach w bloku catch można ponownie zgłosić wyjątek za pomocą instrukcji throw. Nie wymaga to określania wyjątku (zobacz listing 5.27).

Listing 5.27. Ponowne zgłaszanie wyjątku

```
...
catch(Exception exception)
{
    Console.WriteLine(
        $"Ponowne zgłaszenie nieoczekiwanego błędu: {exception.Message}");
    throw;
}
...
...
```

Na listingu 5.27 instrukcja throw jest pusta. Nie trzeba w niej określać, że zgłoszony ma zostać wyjątek zapisany w zmiennej exception. To pokazuje drobną różnicę między różnymi wywołaniami — wywołanie throw; powoduje zachowanie w wyjątku informacji o *stosie wywołań*, natomiast wywołanie throw exception; zastępuje te informacje bieżącymi danymi o stosie wywołań. W trakcie debugowania zwykle bardziej przydatne są informacje o pierwotnym stosie wywołań.

Wskazówki

STOSUJ pustą instrukcję throw, gdy przechwytyujesz i ponownie zgłaszasz wyjątek. Pozwala to zachować stos wywołań.

INFORMUJ o niepowodzeniu w wykonywaniu kodu za pomocą zgłaszanych wyjątków, a nie przy użyciu zwracanych kodów błędu.

NIE stosuj składowych publicznych, które przekazują wyjątki za pomocą zwracanej wartości lub w parametrach typu out. Aby informować o błędach, zgłaszaj wyjątki, zamiast przekazywać je w zwracanej wartości.

⁹ Wyjątki można też przechwytywać za pomocą zgodnego filtra w bloku catch.

Unikaj stosowania obsługi wyjątków do radzenia sobie z oczekiwany sytuacjami

Programiści powinni unikać zgłaszania wyjątków dotyczących oczekiwanych warunków i normalnego przepływu sterowania. Nie należy na przykład oczekwać, że użytkownicy zawsze wprowadzą poprawną wartość, gdy podają swój wiek¹⁰. Dlatego zamiast używać wyjątków do sprawdzania poprawności wprowadzonych danych, należy udostępnić mechanizm badający dane przed podjęciem próby ich konwersji. Jeszcze lepszym rozwiązaniem jest uniemożliwianie użytkownikom wprowadzania błędnych danych. Wyjątki służą do wykrywania wyjątkowych, nieoczekiwanych i potencjalnie awaryjnych sytuacji. Stosowanie wyjątków do innych celów, na przykład do obsługi oczekiwanych wydarzeń, sprawia, że kod jest mało czytelny oraz trudny do zrozumienia i konserwacji.

Początek
2.0

Rozważ na przykład metodę Parse(), używaną w rozdziale 2. do przekształcania łańcuchów znaków na liczby całkowite. Kod przekształcał dane wyjściowe od użytkownika, które nie zawsze były liczbą. Jednym z problemów z metodą Parse() jest to, że jedynym sposobem na ustalenie, czy konwersja zakończy się powodzeniem, jest próba zrzutowania wartości i przechwytcenie wyjątku, gdy operacja się nie powiedzie. Ponieważ zgłaszanie wyjątku to stosunkowo kosztowna operacja, lepiej spróbować przeprowadzić konwersję bez obsługi wyjątków. W tym celu można zastosować jedną z metod z rodziny TryParse(), na przykład int.TryParse(). Wymaga ona użycia słowa kluczowego out, ponieważ funkcja TryParse() zwraca wartość typu bool zamiast przekształconej wartości. Kod z listingu 5.28 pokazuje, jak przeprowadzić konwersję za pomocą metody int.TryParse().

Listing 5.28. Konwersja z wykorzystaniem metody int.TryParse()

```
if (int.TryParse(ageText, out int age))
{
    Console.WriteLine(
        $"Hej, { firstName }! "
        + $"Twój wiek w miesiącach to { age*12 }.");
}
else
{
    Console.WriteLine(
        $"Wprowadzony wiek, { ageText }, nie jest prawidłowy.");
}
```

Koniec
2.0

Dzięki metodzie TryParse() nie trzeba już dodawać bloku try/catch na potrzeby obsługi konwersji danych z łańcuchów znaków na liczby.

Ponadto w C# (podobnie jak w większości innych języków) zgłaszanie wyjątku związane jest ze spadkiem wydajności kodu. Zgłaszcenie wyjątków zajmuje mikrosekundy w porównaniu do nanosekund potrzebnych na wykonanie większości operacji. To opóźnienie zwykle jest nieodczuwalne dla użytkownika, chyba że wyjątek nie zostaje obsłużony. Na przykład jeśli użytkownik kodu z listingu 5.22 wprowadzi błędny wiek, wyjątek nie zostanie obsłużony,

¹⁰ Programiści zawsze powinni się spodziewać, że użytkownicy będą wykonywali nieoczekiwane operacje. Dlatego należy pisać defensywny kod, radzący sobie z „głupimi sztuczkami użytkowników”.

dla tego nastąpi odczuwalne opóźnienie wynikające z wyszukiwania debuggera przez środowisko uruchomieniowe. Na szczęście niska wydajność w momencie zamykania programu nie jest problemem, którym należy się martwić.

Wskazówka

NIE stosuj wyjątków do obsługi zwykłych, oczekiwanych sytuacji. Wyjątki należy stosować w nieoczekiwanych przypadkach.

Podsumowanie

W tym rozdziale omówiono szczegółowo deklarowanie i wywoływanie metod, w tym stosowanie słów kluczowych `out` i `ref` do przekazywania i zwracania zmiennych zamiast ich wartości. Oprócz deklarowania metod w rozdziale opisano też obsługę wyjątków.

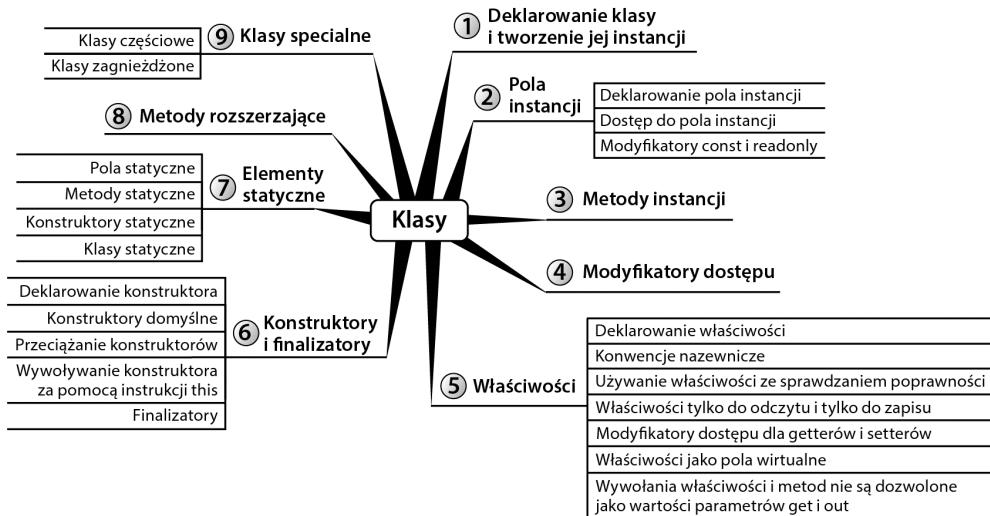
Metody są podstawowym elementem niezbędnym do pisania czytelnego kodu. Zamiast pisać duże metody z wieloma instrukcjami, powinieneś się starać tworzyć „akapity” w kodzie, budując metody o długości do mniej więcej dziesięciu instrukcji. Proces podziału dużych metod na mniejsze części to jeden ze sposobów refaktoryzacji kodu w celu poprawy jego czytelności i ułatwienia konserwacji.

W następnym rozdziale opisano klasy i wyjaśniono, że łączą one metody (operacje) z polami (danymi) w odrębnej jednostce.

6

Klasy

ROZDZIALE 1. KRÓTKO POKAZANO, jak zadeklarować nową klasę `HelloWorld`. W rozdziałach 2. i 3. zapoznałeś się z wbudowanymi typami prostymi języka C#. Ponieważ przeczytałeś już także o przepływie sterowania i o deklarowaniu metod, pora przejść do omówienia definiowania własnych typów. Jest to podstawowy element każdego programu w języku C#. To obsługa klas i tworzonych na ich podstawie obiektów sprawia, że C# jest językiem obiektowym.



Ten rozdział zawiera wprowadzenie do programowania obiektowego w języku C#. Skoncentrowano się tu na definiowaniu **klas**, które są szablonami służącymi do tworzenia obiektów.

Wszystkie elementy programowania strukturalnego opisane we wcześniejszych rozdziałach dotyczą także programowania obiektowego. Jednak dzięki umieszczeniu tych elementów w klasach można tworzyć większe, bardziej uporządkowane programy, których konserwacja jest łatwiejsza.

Jedną z podstawowych zalet programowania obiektowego jest to, że zamiast tworzyć nowe programy od podstaw, można połączyć ze sobą kolekcję istniejących, opracowanych wcześniej obiektów. Można też rozszerzać klasy o kolejne funkcje i budować nowe klasy, dodając w ten sposób nowe mechanizmy.

Czytelnicy, którzy nie mają doświadczenia w stosowaniu programowania obiektowego, powinni przeczytać ZAGADNIENIE DLA POCZĄTKUJĄCYCH, gdzie znajdą wprowadzenie do tego tematu. Tekst poza zagadnieniami dla początkujących dotyczy programowania obiektowego w języku C#. Autorzy zakładają, że czytelnicy tych fragmentów znają pojęcia z obszaru programowania obiektowego.

W tym rozdziale wyjaśniono obsługę hermetyzacji w języku C#. Służą do tego klasy, właściwości i modyfikatory dostępu (a także omówione w rozdziale 5. metody). Rozdział 7. jest rozwinięciem niniejszego i znajdziesz tam wprowadzenie do dziedziczenia i polimorfizmu (są to techniki dostępne dzięki programowaniu obiektowemu).

ZAGADNIENIE DLA POCZĄTKUJĄCYCH

Programowanie obiektowe

Kluczem do sukcesów w programowaniu jest obecnie umiejętność zapewniania uporządkowania i struktury implementacji złożonych wymagań związanych z rozbudowanymi aplikacjami. Programowanie obiektowe to jedna z najważniejszych metod pozwalających osiągnąć ten cel. Programistom stosującym podejście obiektowe trudno wyobrazić sobie powrót do programowania strukturalnego (chyba że w trakcie pisania najprostszych aplikacji).

Podstawowym elementem programowania obiektowego są klasy. Grupa klas często stanowi abstrakcję programistyczną, model lub szablon obiektów z rzeczywistego świata. Na przykład klasa `OpticalStorageMedia` może udostępniać metodę `Eject()`, która powoduje wysunięcie płyty z odtwarzacza. Klasa `OpticalStorageMedia` to abstrakcja programistyczna obiektu z rzeczywistego świata — odtwarzacza płyt CD lub DVD.

Z klasami związane są trzy podstawowe aspekty programowania obiektowego — hermetyzacja, dziedziczenie i polimorfizm.

Hermetyzacja

Hermetyzacja umożliwia ukrywanie szczegółów. W razie konieczności można uzyskać do nich dostęp, jednak dzięki inteligentnemu ukryciu szczegółów zrozumienie dużych programów jest łatwiejsze, dane są chronione przed przypadkową modyfikacją, a kod staje się prostszy w konserwacji, ponieważ skutki modyfikacji kodu są ograniczone do hermetycznej jednostki. Przykładem zastosowania hermetyzacji są metody. Choć kod metody może się znajdować bezpośrednio w kodzie jednostki wywołującej, umieszczenie kodu w metodzie zapewnia korzyści płynące z hermetyzacji.

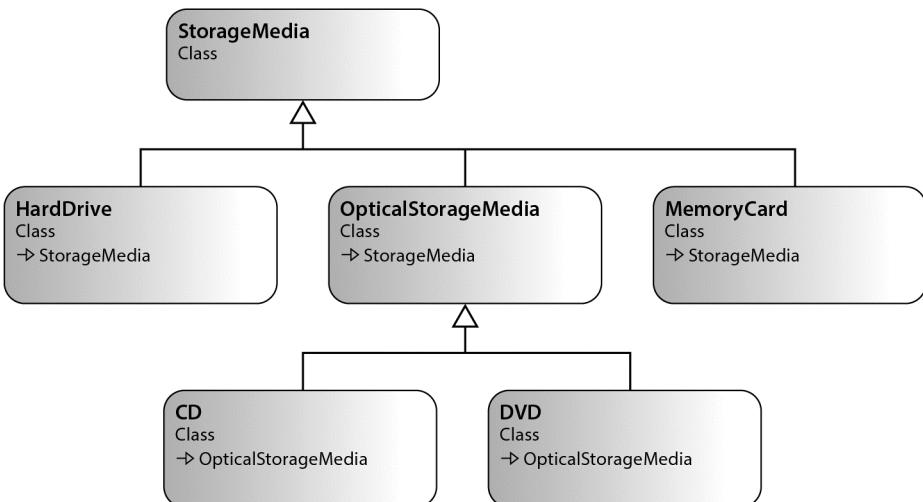
Dziedziczenie

Rozważ następujący przykład — odtwarzacz DVD to rodzaj napędu optycznego. Odczytuje płyty o określonej pojemności, na których można zapisać film w formacie cyfrowym. Odtwarzacz CD to także napęd optyczny, ale o innych cechach. Zabezpieczenie płyt CD przed kopiowaniem jest inne niż w przypadku płyt DVD. Oba rodzaje płyt mają też inną pojemność. Napędy CD i DVD różnią się od dysków twardych, dysków USB i stacji dyskietek

(pamiętasz je jeszcze?). Wszystkie wymienione napędy to urządzenia pamięci masowej, mające jednak specyficzne cechy. Poszczególne urządzenia różnią się nawet podstawowymi funkcjami, na przykład obsługiwany systemami plików i tym, czy nośniki są przeznaczone tylko do odczytu, czy do odczytu i zapisu.

Dziedziczenie w programowaniu obiektowym umożliwia budowanie relacji „jest odmianą” między podobnymi, ale odrębnymi obiektami. Można powiedzieć, że napęd DVD „jest odmianą” urządzenia pamięci masowej. To samo dotyczy napędu CD. Dlatego oba te urządzenia służą do przechowywania danych. Można też stwierdzić, że oba wymienione urządzenia „są odmianą” napędu optycznego, który sam „jest odmianą” urządzenia pamięci masowej.

Jeśli zdefiniujesz klasy odpowiadające wymienionym rodzajom urządzeń pamięci masowej, uzyskasz **hierarchię klas**. Taka hierarchia to zbiór relacji „jest odmianą”. Klasą bazową, po której dziedziczą wszystkie urządzenia pamięci masowej, może być klasa `StorageMedia`. Klasy reprezentujące napędy CD i DVD, dyski twardy, dyski USB i stacje dyskietek dziedziczą po klasie `StorageMedia`. Jednak klasy reprezentujące napędy CD i DVD nie muszą dziedziczyć bezpośrednio po tej klasie. Zamiast tego mogą dziedziczyć po pośredniej klasie `OpticalStorageMedia`. Wizualnie tę hierarchię klas można przedstawić za pomocą języka UML (ang. *Unified Modeling Language*) w formie diagramu klas, takiego jak pokazany na rysunku 6.1.



Rysunek 6.1. Hierarchia klas

Relacja dziedziczenia obejmuje przynajmniej dwie klasy, z których jedna jest bardziej specyficzną odmianą drugiej. Na rysunku 6.1 `HardDrive` to bardziej specyficzna wersja klasy `StorageMedia`. Choć bardziej wyspecjalizowany typ `HardDrive` jest odmianą typu `StorageMedia`, relacja ta nie działa w drugą stronę. Obiekt typu `StorageMedia` nie zawsze jest obiektem typu `HardDrive`. Na rysunku 6.1 widać też, że dziedziczenie może dotyczyć więcej niż dwóch klas.

Bardziej wyspecjalizowany typ jest nazywany **typem pochodnym** lub **podtypem**. Bardziej ogólny typ to **typ bazowy** lub **nadtyp**. Typ bazowy czasem nazywany jest też z angielskiego *rodzicem*, a jego typy pochodne — *dziećmi*, co jest jednak mylące. W końcu dziecko nie jest odmianą rodzica! Dlatego w tej książce stosowane są określenia *typ pochodny* i *typ bazowy*.

Utworzenie typu **pochodnego** lub **dziedziczącego** polega na zbudowaniu **wyspecjalizowanej** wersji tego typu. Oznacza to dostosowanie typu bazowego w taki sposób, by lepiej nadawał się do określonego celu. Typ bazowy może obejmować szczegóły implementacji wspólne wszystkim typom pochodnym od niego.

Najważniejszą cechą dziedziczenia jest to, że wszystkie typy pochodne dziedziczą składowe typu bazowego. Często implementację składowych bazowych można zmodyfikować, jednak niezależnie od tego typ pochodny obejmuje składowe typu bazowego plus inne składowe, które utworzono bezpośrednio w typie pochodnym.

Dzięki typom pochodnym można uporządkować klasy w spójną hierarchię, w której typy pochodne są bardziej specyficzne od ich typów bazowych.

Polimorfizm

Nazwa **polimorfizm** pochodzi od słów *poly* (czyli wiele) i *morf* (czyli postać). W kontekście programowania obiektowego polimorfizm oznacza tyle, że jedna metoda lub jeden typ mogą mieć wiele postaci implementacji.

Założymy, że odtwarzacz obsługuje zarówno płyty CD, jak i płyty DVD z plikami MP3. Jednak implementacja metody `Play()` zależy od typu nośnika. Wywołanie metody `Play()` obiektów reprezentujących płyty CD z muzyką lub płyty DVD z muzyką spowoduje w obu przypadkach odtworzenie nagrania, ponieważ typy obu obiektów potrafią to zrobić. Odtwarzacz zna tylko wspólny typ bazowy `OpticalStorageMedia` i wie, że zdefiniowana jest w nim metoda `Play()`. Polimorfizm to zasada, zgodnie z którą typ odpowiada za szczegóły implementacji metody występującej w różnych typach pochodnych od tego samego typu bazowego (lub interfejsu), obejmującego sygnaturę danej metody.

Deklarowanie klasy i tworzenie jej instancji

Aby zdefiniować klasę, należy najpierw podać słowo kluczowe `class`, a następnie identyfikator. Przedstawia to listing 6.1.

Listing 6.1. Definiowanie klasy

```
public class Employee  
{  
}
```

Cały kod klasy należy umieścić między nawiasami klamrowymi podanymi po deklaracji klasy. Choć nie jest to wymagane, zwykle każdą klasę umieszcza się w odrębnym pliku. Dzięki temu można łatwiej znaleźć kod definiujący określoną klasę, ponieważ zgodnie z konwencją nazwa pliku to nazwa klasy.

Wskazówki

NIE umieszczaj więcej niż jednej klasy w jednym pliku z kodem źródłowym.

NAZYWAJ pliki z kodem źródłowym na podstawie nazwy publicznego typu zawartego w danym pliku.

Po zdefiniowaniu nowej klasy możesz korzystać z niej tak, jakby była wbudowaną klasą platformy. Oznacza to, że możesz zadeklarować zmienną danego typu lub zdefiniować metodę przyjmującą parametr typu nowej klasy. Takie deklaracje przedstawiono na listingu 6.2.

Listing 6.2. Deklarowanie zmiennych typu klasy

```
class Program
{
    static void Main()
    {
        Employee employee1, employee2;
        // ...
    }

    static void IncreaseSalary(Employee employee)
    {
        // ...
    }
}
```

ZAGADNIENIE DLA POCZĄTKUJĄCYCH

Czym są obiekty, a czym klasy?

W codziennych rozmowach pojęcia *klasa* i *obiekt* czasem są stosowane wymiennie. Jednak określenia obiekt i klasa oznaczają coś innego. **Klasa** to szablon określający, jak obiekt będzie wyglądał po utworzeniu jej instancji. Tak więc **obiekt** jest instancją klasy. Klassy są jak forma określająca wygląd produktu. Obiekty są jak produkty wytwarzane za pomocą tej formy. Proces tworzenia obiektu na podstawie klasy nazywany jest **tworzeniem instancji**, ponieważ obiekt to instancja klasy.

Po zdefiniowaniu nowego typu (klasy) można utworzyć na jego podstawie obiekt. W C# (podobnie jak w poprzedzających go językach) do tworzenia obiektów służy słowo kluczowe new (zobacz listing 6.3).

Listing 6.3. Tworzenie instancji klasy

```
class Program
{
    static void Main()
    {
        Employee employee1 = new Employee();
        Employee employee2;
        employee2 = new Employee();

        IncreaseSalary(employee1);
        IncreaseSalary(employee2);
    }
}
```

Nie jest zaskoczeniem, że wartość można przypisać w instrukcji z deklaracją lub w odrębnym poleceniu.

W przeciwnieństwie do używanych do tego miejsca typów prostych, w tym przypadku nie ma sposobu na utworzenie obiektu typu `Employee` za pomocą literala. Operator `new` jest instrukcją dla środowiska uruchomieniowego, oznaczającą, że należy przydzielić pamięć na obiekt typu `Employee`, utworzyć ten obiekt i zwrócić prowadzącą do niego referencję.

Choć istnieje operator przeznaczony do jawnego przydziału pamięci, nie ma analogicznego operatora służącego do jej zwalniania. Środowisko uruchomieniowe automatycznie odzyskuje pamięć po tym, jak obiekt przestanie być dostępny. Za to automatyczne przywracanie pamięci odpowiada **mechanizm odzyskiwania pamięci**. Określa on, które obiekty nie są już używane przez inne aktywne obiekty, a następnie odzyskuje pamięć zajmowaną przez niepotrzebne obiekty. To sprawia, że w trakcie komplikacji nie da się określić w programie miejsca, w którym pamięć zostanie odzyskana i zwrócona do systemu.

W przedstawionym prostym przykładzie z typem `Employee` nie są powiązane żadne dane ani metody. Dlatego obiekt tego typu jest bezużyteczny. W następnym podrozdziale zobaczysz, jak dodawać dane do klas.

■ ZAGADNIENIE DLA POCZĄTKUJĄCYCH

Hermetyzacja, część 1. Obiekty łączą dane z metodami

Jeśli otrzymasz stos karteczek z imionami, stos karteczek z nazwiskami oraz stos karteczek z wysokością pensji pracowników, te dane będą mało wartościowe, chyba że wiesz, iż informacje o danej osobie znajdują się w tym samym miejscu każdego stosu. Jednak nawet wtedy trudno korzystać z tych danych, ponieważ ustalenie imienia i nazwiska pracownika wymaga przeszukania dwóch stosów. Co gorsza, jeśli usuniesz jeden ze stosów (na przykład z imieniem), nie da się ponownie powiązać imienia z nazwiskiem i wynagrodzeniem. Dlatego bardziej pomocny jest jeden stos karteczek, na których zapisane są wszystkie dane o poszczególnych osobach. W tym podejściu imię, nazwisko i pensja są połączone w hermetyczną jednostkę.

Poza kontekstem programowania obiektowego **hermetyzacja** zbioru elementów polega na umieszczeniu ich w hermetycznym opakowaniu. Podobnie w programowaniu obiektowym hermetyzacja metod i danych oznacza umieszczenie ich w obiekcie. W ten sposób grupowane są wszystkie **składowe klasy** (czyli jej dane i metody), dzięki czemu nie trzeba ich przetwarzać niezależnie. Zamiast przekazywać imię, nazwisko i wynagrodzenie jako trzy odrębne parametry metody, dzięki obiektom możesz przekazać referencję do obiektu reprezentującego pracownika. Gdy wywołana metoda otrzyma referencję do obiektu, może przekazać do niego komunikat (na przykład wywołać metodę taką jak `AdjustSalary()`), aby zażądać wykonania określonej operacji.

Porównanie języków — operator `delete` w języku C++

Programiści języka C# powinni traktować wywołanie operatora `new` jak narzędzie do tworzenia obiektu, a nie do przydziału pamięci. Operator `new` służy do tworzenia obiektów umieszczanych na stercie i obiektów umieszczanych na stosie, co jest podkrešleniem faktu, że `new` nie określa sposobu przydziału pamięci i tego, czy potrzebne jest jej zwolnienie.

Dlatego w języku C# operator `delete` dostępny w języku C++ nie jest potrzebny. Przydzielanie i zwalnianie pamięci to operacje, którymi zarządza środowisko uruchomieniowe. Dzięki temu programista może się skoncentrować na logice działania aplikacji. Jednak choć środowisko uruchomieniowe zarządza pamięcią, nie robi tego w przypadku innych zasobów (takich jak połączenia z bazą danych, portami sieciowymi itd.). Język C#, inaczej niż jest to w C++, nie zapewnia **niejawnego deterministycznego przywracania zasobów** (na etapie komplikacji nie jest definiowane miejsce w kodzie, gdzie nastąpi niejawne usunięcie obiektu). Na szczęście C# obsługuje **jawne deterministyczne przywracanie zasobów** (z wykorzystaniem instrukcji `using`) i **niejawne niedeterministyczne przywracanie zasobów** (przy użyciu finalizatorów).

Pola instancji

Jednym z najważniejszych aspektów projektowania obiektowego jest grupowanie danych w celu zapewnienia im struktury. W tym podrozdziale wyjaśniono, jak dodać dane do klasy `Employee`. Ogólnym obiektowym określeniem zmiennej przechowującej dane w klasie jest **zmienna składowa**. To pojęcie jest zrozumiałe w kontekście języka C#, natomiast bardziej standardowa nazwa, używana także w specyfikacji, to **pole**. Jest to nazwana jednostka przechowywania danych powiązana z zawierającym ją typem. **Pola instancji** to zmienne deklarowane na poziomie klasy, służące do przechowywania danych powiązanych z obiektem. **Powiązanie** jest tu relacją między typem danych a polem z danymi.

Deklarowanie pola instancji

Na listingu 6.4 klasę `Employee` zmodyfikowano przez dodanie do niej trzech pól — `FirstName`, `LastName` i `Salary`. Warto zauważyć, że ten listing i powiązane listingi z tego rozdziału nie są oznaczone jako specyficzne dla wersji C# 8.0, choć pole `Salary` jest zadeklarowane jako wprowadzony w tej wersji typ referencyjny dopuszczający wartość `null`.

Listing 6.4. Deklarowanie pól

```
public class Employee
{
    public string FirstName;
    public string LastName;
    public string? Salary;
}
```

Po dodaniu tych pól można zapisać podstawowe dane w każdym obiekcie typu `Employee`. Tu pola są poprzedzone modyfikatorem dostępu `public`. Użycie tego modyfikatora oznacza, że dane w polu są dostępne dla klas innych niż `Employee` (zobacz punkt „Modyfikatory dostępu” w dalszej części rozdziału).

Deklaracja pola (podobnie jak deklaracje zmiennych lokalnych) obejmuje typ danych pola. Ponadto w deklaracji pola można przypisać do niego wartość początkową, tak jak do pola `Salary` na listingu 6.5.

Listing 6.5. Ustawianie początkowych wartości pól w deklaracji

```
// Wyłączanie w niekompletnym kodzie ostrzeżenia o niezainicjowanym polu niedopuszczającym wartości null
#pragma warning disable CS8618
public class Employee
{
    public string FirstName;
    public string LastName;
    public string? Salary = "Za niskie";
}
```

Wskazówki dotyczące nazw i dodawania pól znajdziesz w dalszej części rozdziału, po omówieniu właściwości w języku C#. W tym miejscu warto zauważyć, że kod z dotychczasowych listingów *nie* jest zgodny z ogólnymi konwencjami. Może on generować następujące ostrzeżenia:

- CS0649: *Field is never assigned to, and will always have its default value null*
(do pola nie jest przypisywana wartość i zawsze ma ono wartość domyślną null);
- CS8618: *Non-nullable field is uninitialized. Consider declaring as nullable*
(pole niedopuszczające wartości null jest niezainicjowane; rozważ zadeklarowanie go jako pola dopuszczającego wartość null).

W tym kodzie pola FirstName i LastName są niezainicjowane, dlatego powodują ostrzeżenia CS8618.

Dla wyjaśnienia należy wspomnieć, że te ostrzeżenia są ignorowane. W kodzie źródłowym do książek są one wyłączane za pomocą dyrektywy #pragma (do czasu omówienia wszystkich potrzebnych zagadnień w dalszej części rozdziału).

Dostęp do pól instancji

Możliwe jest ustawianie i pobieranie danych w polach. Brak modyfikatora static oznacza, że dane pole jest polem instancji. Dostęp do pól instancji jest możliwy tylko w instancji (w obiekcie) danej klasy. Nie można korzystać bezpośrednio z pól za pomocą klasy (bez utworzenia jej instancji).

Listing 6.6 zawiera nową wersję klasy Program korzystającej z klasy Employee. Efekt działania kodu pokazano w danych wyjściowych 6.1.

Listing 6.6. Dostęp do pól

```
class Program
{
    static void Main()
    {
        Employee employee1 = new Employee();
        Employee employee2;
        employee2 = new Employee();

        employee1.FirstName = "Inigo";
        employee1.LastName = "Montoya";
        employee1.Salary = "Za niskie";
        IncreaseSalary(employee1);
```

```

Console.WriteLine(
    "{0} {1}: {2}",
    employee1.FirstName,
    employee1.LastName,
    employee1.Salary);
    // ...
}

static void IncreaseSalary(Employee employee)
{
    employee.Salary = "Wystarczające, by przeżyć";
}

```

DANE WYJŚCIOWE 6.1.

Inigo Montoya: Wystarczające, by przeżyć

Na listingu 6.6 (podobnie jak wcześniej) tworzone są dwa obiekty typu Employee. Dalej kod ustawia każde pole, wywołuje metodę IncreaseSalary() w celu zmiany poziomu wynagrodzenia, a następnie wyświetla każde pole powiązane z obiektem employee1.

Zauważ, że najpierw trzeba wskazać instancję klasy Employee, której dotyczą operacje. Dlatego gdy kod przypisuje wartość do pola i pobiera ją, przed nazwą pola pojawia się nazwa zmiennej, employee1.

Metody instancji

Zamiast formatować nazwy w wywołaniu metody `WriteLine()` w metodzie `Main()`, można udostępnić w klasie `Employee` odpowiedzialną za to metodę. Zmiana działania kodu tak, by znajdował się w klasie `Employee`, a nie w klasie `Program`, jest zgodna z zasadą hermetyzacji klasy. Dlaczego nie umieścić metod dotyczących imienia i nazwiska pracownika w klasie, która zawiera te dane? Na listingu 6.7 pokazano, jak utworzyć potrzebną metodę.

Listing 6.7. Dostęp do pól w zawierającej je klasie

```

public class Employee
{
    public string FirstName;
    public string LastName;
    public string? Salary;

    public string GetName()
    {
        return $"{ FirstName } { LastName }";
    }
}

```

W tej metodzie nie ma nic niezwykłego w porównaniu do metod, które poznaleś w rozdziale 5. Różnica polega na tym, że teraz metoda `GetName()` używa pól obiektu zamiast zmiennych lokalnych. Ponadto w deklaracji metody nie ma modyfikatora `static`. Dalej

w tym rozdziale zobaczysz, że metody statyczne nie mają bezpośredniego dostępu do występujących w klasie pól instancji. Aby użyć składowej instancji (metody lub pola), trzeba najpierw utworzyć instancję.

Po dodaniu metody `GetName()` możesz zmodyfikować metodę `Program.Main()` i wykorzystać w niej `GetName()`. Nowy kod przedstawiono na listingu 6.8 i w danych wyjściowych 6.2.

Listing 6.8. Dostęp do pól spoza zawierającej je klasy

```
class Program
{
    static void Main()
    {
        Employee employee1 = new Employee();
        Employee employee2;
        employee2 = new Employee();

        employee1.FirstName = "Inigo";
        employee1.LastName = "Montoya";
        employee1.Salary = "Za niskie";
        IncreaseSalary(employee1);
        Console.WriteLine(
            $"{ employee1.GetName() }: { employee1.Salary }");
        // ...
    }
    // ...
}
```

DANE WYJŚCIOWE 6.2.

```
Inigo Montoya: Wystarczające, by przeżyć
```

Stosowanie słowa kluczowego this

Referencję do obiektu możesz otrzymać na podstawie składowych instancji należących do danej klasy. W języku C# by jawnie podać, że potrzebne pola lub metody są składowymi instancji danej klasy, należy zastosować słowo kluczowe `this`. Gdy wywoływana jest dowolna składowa instancji, to słowo jest dodawane niejawnie. Reprezentuje ono używaną w danym momencie instancję klasy.

Przyjrzyj się metodzie `SetName()` przedstawionej na listingu 6.9.

Listing 6.9. Używanie słowa kluczowego `this` do jawnego wskazywania elementu, do którego należy pole

```
public class Employee
{
    public string FirstName;
    public string LastName;
    public string? Salary;

    public string GetName()
    {
        return $"{ FirstName } { LastName }";
```

```

    }

public void SetName(
    string newFirstName, string newLastName)
{
    this.FirstName = newFirstName;
    this.LastName = newLastName;
}
}

```

W tym przykładzie słowo kluczowe `this` służy do określenia, że pola `FirstName` i `LastName` to składowe instancji danej klasy.

Choć słowo kluczowe `this` można umieszczać przed dowolnymi (i wszystkimi) referencjami do lokalnych składowych klasy, ogólnie zaleca się unikać zaśmiecania kodu, jeśli instrukcje nie dają dodatkowej wartości. Dlatego nie powinieneś stosować słowa kluczowego `this`, chyba że jest to niezbędne. Na listingu 6.12 (przedstawionym w dalszej części rozdziału) pokazano jedną z nielicznych sytuacji, w których to słowo jest potrzebne. Jednak na listingach 6.9 i 6.10 jest ono zbędne. Na listingu 6.9 słowo kluczowe `this` można całkowicie pominąć bez zmieniającego znaczenia kodu. Na listingu 6.10 (przedstawionym dalej) zmiana konwencji nazewniczej dla pól pozwala uniknąć konfliktów między nazwami zmiennych lokalnych i pól.

ZAGADNIENIE DLA POCZĄTKUJĄCYCH

Wykorzystywanie stylu programowania do unikania wieloznaczności

W metodzie `SetName()` nie trzeba używać słowa kluczowego `this`, ponieważ nazwa `FirstName` różni się od `newFirstName`. Założmy jednak, że zamiast nazywać parametr `newFirstName`, wybrałeś nazwę `FirstName` (w NotacjiPascalowej), tak jak na listingu 6.10.

Listing 6.10. Używanie słowa kluczowego `this` w celu uniknięcia wieloznaczności

```

public class Employee
{
    public string FirstName;
    public string LastName;
    public string? Salary;

    public string GetName()
    {
        return $"'{ FirstName } { LastName }'";
    }

    // Uwaga — nazwy parametrów zapisane w NotacjiPascalowej.
    public void SetName(string FirstName, string LastName)
    {
        this.FirstName = FirstName;
        this.LastName = LastName;
    }
}

```

W tym przykładzie nie da się wskazać pola FirstName bez jawnego określania, że należy ono do obiektu typu Employee. Słowo kluczowe this działa w tym kontekście tak jak przedrostek w postaci zmiennej employee1 używany w metodzie Program.Main() na listingu 6.8. Tu słowo this określa, że używany ma być obiekt, dla którego wywołano metodę SetName().

Na listingu 6.10 nie zastosowano konwencji nazewniczej z języka C#, zgodnie z którą parametry deklaruje się w taki sposób jak zmienne lokalne, przy użyciu notacji Wielbłąda. Może to prowadzić do trudnych do wykrycia usterek, ponieważ kod z przypisaniem pola FirstName do parametru FirstName skompiluje się, a nawet będzie działał. Aby uniknąć tego problemu, warto stosować jedne konwencje nazewnicze dla parametrów i zmiennych lokalnych, a inne dla pól. Przykładową konwencję opisano w dalszej części rozdziału.

Porównanie języków — dostęp do instancji klasy za pomocą słowa kluczowego Me w języku Visual Basic

Słowo kluczowe this z języka C# działa tak samo jak słowo kluczowe Me w Visual Basicu.

Na listingach 6.9 i 6.10 słowo kluczowe this nie jest używane w metodzie GetName(), ponieważ jest opcjonalne. Jednak jeśli istnieją zmienne lokalne lub parametry o nazwie identycznej z nazwą pola (tak jak w metodzie SetName() na listingu 6.10), pominięcie słowa kluczowego this spowoduje dostęp do zmiennej lokalnej lub parametru, choć kod powinien użyć pola. W takiej sytuacji słowo kluczowe this jest niezbędne.

Słowo kluczowe this możesz też wykorzystać do tego, by jawnie użyć metod klasy. Na przykład w metodzie SetName() można wywołać metodę this.GetName(), by wyświetlić nowo przypisane imię i nazwisko (zobacz listing 6.11 i dane wyjściowe 6.3).

Listing 6.11. Używanie słowa kluczowego this do metody

```
public class Employee
{
    // ...

    public string GetName()
    {
        return $"{ FirstName } { LastName }";
    }

    public void SetName(string newFirstName, string newLastName)
    {
        this.FirstName = newFirstName;
        this.LastName = newLastName;
        Console.WriteLine(
            $"Imię i nazwisko zmieniono na '{ this.GetName() }'.");
    }
}

class Program
{
    static void Main()
    {
```

```
Employee employee = new Employee();
employee.SetName("Inigo", "Montoya");
// ...
}
// ...
```

DANE WYJŚCIOWE 6.3.

Imię i nazwisko zmieniono na 'Inigo Montoya'.

Czasem konieczne jest użycie słowa kluczowego `this`, by przekazać referencję do obecnie używanego obiektu. Przyjrzyj się metodzie `Save()` z listingu 6.12.

Listing 6.12. Przekazywanie obiektu `this` w wywołaniu metody

```
public class Employee
{
    public string FirstName;
    public string LastName;
    public string? Salary;

    public void Save()
    {
        DataStorage.Store(this);
    }
}

class DataStorage
{
    // Zapisywanie obiektu employee w pliku
    // o nazwie odpowiadającej imieniu i nazwisku pracownika.
    public static void Store(Employee employee)
    {
        // ...
    }
}
```

Metoda `Save()` z listingu 6.12 wywołuje metodę `Store()` klasy `DataStorage`. Do tej metody trzeba przekazać przeznaczony do utrwalenia obiekt typu `Employee`. Obiekt ten jest przekazywany za pomocą słowa kluczowego `this`, pozwalającego tu przekazać instancję typu `Employee`, dla której wywołano metodę `Save()`.

Zapisywanie i wczytywanie plików

W kodzie metody `Store()` w klasie `DataStorage` używane są klasy z przestrzeni nazw `System.IO` (zobacz listingu 6.13). W metodzie `Store()` kod najpierw tworzy obiekt typu `FileStream`, związany z plikiem o nazwie odpowiadającej imieniu i nazwisku pracownika. Parametr `FileMode.Create` oznacza, że jeśli taki plik nie istnieje, należy utworzyć nowy o nazwie `<firstname><lastname>.dat`. Jeżeli plik istnieje, jego zawartość zostanie zastąpiona nową.

Dalej kod tworzy obiekt klasy `StreamWriter`. Ta klasa odpowiada za zapis tekstu w obiekcie typu `FileStream`. Zapis danych odbywa się za pomocą metody `WriteLine()` (przebiega to podobnie jak wyświetlanie tekstu w konsoli).

Listing 6.13. Utrwalanie danych w pliku

```
using System;
// Dodanie przestrzeni nazw IO.
using System.IO;

class DataStorage
{
    // Zapis obiektu employee w pliku o nazwie
    // odpowiadającej imieniu i nazwisku pracownika.
    // Kod do obsługi błędów został pominięty.
    public static void Store(Employee employee)
    {
        // Tworzenie obiektu typu FileStream z nazwą pliku
        // FirstNameLastName.dat. Parametr FileMode.Create powoduje
        // utworzenie nowego pliku lub zastąpienie zawartości istniejącego.
        FileStream stream = new FileStream(
            employee.FirstName + employee.LastName + ".dat",
            FileMode.Create);1

        // Tworzenie obiektu typu StreamWriter na potrzeby zapisu
        // tekstu w obiekcie typu FileStream.
        StreamWriter writer = new StreamWriter(stream);

        // Zapis wszystkich danych dotyczących pracownika.
        writer.WriteLine(employee.FirstName);
        writer.WriteLine(employee.LastName);
        writer.WriteLine(employee.Salary);

        // Zamknięcie obiektu StreamWriter i powiązany z nim strumień.
        writer.Dispose(); // Automatycznie zamknie strumień.
    }
    // ...
}
```

Po zakończeniu operacji zapisu obiekty typów `FileStream` i `StreamWriter` trzeba zamknąć, by nie pozostały otwarte w nieskończoność w trakcie oczekiwania na uruchomienie mechanizmu odzyskiwania pamięci. Na listingu 6.13 nie ma kodu do obsługi błędów, dlatego jeśli zgłoszony zostanie wyjątek, żadna metoda `Close()` nie zostanie wywołana.

Proces wczytywania danych odbywa się podobnie (zobacz listing 6.14).

Listing 6.14. Pobieranie danych z pliku

```
public class Employee
{
    // ...
}
```

¹ Kod można poprawić za pomocą instrukcji `using`, której jednak nie zastosowano, ponieważ nie została jeszcze omówiona.

```
// Dodanie przestrzeni nazw IO.  
using System;  
using System.IO;  
  
class DataStorage  
{  
    // ...  
  
    public static Employee Load(string firstName, string lastName)  
    {  
        Employee employee = new Employee();  
  
        // Tworzenie obiektu typu FileStream dla pliku o nazwie  
        // FirstNameLastName.dat. Parametr FileMode.Open powoduje  
        // otwarcie istniejącego pliku lub zgłoszenie błędu.  
        FileStream stream = new FileStream(  
            firstName + lastName + ".dat", FileMode.Open);2  
  
        // Tworzenie obiektu typu StreamReader przeznaczonego do odczytu tekstu z pliku.  
        StreamReader reader = new StreamReader(stream);  
  
        // Wczytywanie każdego wiersza z pliku i zapisywanie  
        // danych w odpowiedniej właściwości.  
        employee.FirstName = reader.ReadLine()??  
        throw new InvalidOperationException("FirstName nie może być równe null");  
        employee.LastName = reader.ReadLine()??  
        throw new InvalidOperationException("LastName nie może być równe null");  
        employee.Salary = reader.ReadLine();  
  
        // Zamknięcie obiektu typu StreamReader i powiązanego z nim strumienia.  
        reader.Dispose(); // Automatycznie zamknie strumień.  
  
        return employee;  
    }  
}  
  
class Program  
{  
    static void Main()  
    {  
        Employee employee1;  
  
        Employee employee2 = new Employee();  
        employee2.SetName("Inigo", "Montoya");  
        employee2.Save();  
  
        // Modyfikowanie obiektu employee2 po jego zapisaniu.  
        IncreaseSalary(employee2);  
  
        // Wczytywanie danych obiektu employee1 z zapisanej wersji obiektu employee2.  
        employee1 = DataStorage.Load("Inigo", "Montoya");
```

² Kod można poprawić za pomocą instrukcji using, której jednak nie zastosowano, ponieważ nie została jeszcze omówiona.

```
Console.WriteLine(  
    $"{{ employee1.GetName() }}: {{ employee1.Salary }}");  
  
    // ...  
}  
// ...  
}
```

Wyniki działania kodu pokazano w danych wyjściowych 6.4.

DANE WYJŚCIOWE 6.4.

```
Imię i nazwisko zmieniono na 'Inigo Montoya'.  
Inigo Montoya:
```

Na listingu 6.14 przedstawiono proces odwrotny do zapisu danych. Tu użyto klasy `StreamReader` zamiast `StreamWriter`. Także w tym przypadku po wczytaniu danych trzeba wywołać metodę `Close()` obiektów typów `FileStream` i `StreamReader`.

W danych wyjściowych 6.4 po członie Inigo Montoya: nie pojawia się wynagrodzenie. Dzieje się tak, ponieważ zostało ono ustalone (w wywołaniu metody `IncreaseSalary()`) na wartość `Wystarczające`, by przeżyć dopiero po wywołaniu metody `Save()`.

Zauważ, że w metodzie `Main()` metoda `Save()` jest wywoływana dla instancji klasy `Employee`, natomiast w celu wczytania danych nowego pracownika kod wywołuje metodę `DataStorage.Load()`. Aby wczytać dane pracownika, nie jest potrzebna instancja klasy `Employee`. Dlatego metoda odpowiedzialna za wczytywanie nie powinna być metodą instancji tej klasy. Zamiast używać metody `Load` klasy `DataStorage`, można dodać do klasy `Employee` statyczną metodę `Load()` (zobacz podrozdział „Składowe statyczne” w dalszej części rozdziału). Dzięki temu możliwe będzie wywołanie metody `Employee.Load()` dla klasy `Employee`, a nie dla jej instancji.

Zwróć uwagę na dyrektywę `System.IO` w początkowej części listingu. Dzięki niej można korzystać ze wszystkich klas z przestrzeni nazw `IO` bez konieczności poprzedzania ich nazwą tej przestrzeni.

Modyfikatory dostępu

Gdy wcześniej w rozdziale deklarowałeś pole, poprzedziłeś jego deklarację słowem kluczowym `public`. To słowo jest **modyfikatorem dostępu**, określającym poziom hermetyzacji składowej, przy której występuje. Dostępnych jest sześć modyfikatorów dostępu: `public`, `private`, `protected`, `internal`, `protected internal` i `private protected`. W tym podrozdziale opisano pierwsze dwa z nich.

ZAGADNIENIE DLA POCZĄTKUJĄCYCH

Hermetyzacja, część 2. Ukrywanie informacji

Jedną z funkcji hermetyzacji jest umożliwienie umieszczania danych i metod w odrębnej jednostce. Oprócz tego hermetyzacja służy do ukrywania wewnętrznych szczegółów dotyczących danych i zachowania obiektów. W pewnym stopniu umożliwiają to także metody.

Poza metodą dla jednostki wywołującej widoczna jest tylko deklaracja. Wewnętrzna implementacja metody nie jest dostępna. Programowanie obiektowe umożliwia zrobienie następnego kroku, ponieważ pozwala kontrolować zakres, w jakim składowe są widoczne spoza klasy. Składowe, które nie są widoczne poza klasą, to **składowe prywatne**.

W programowaniu obiektowym *hermetyzacja* to pojęcie oznaczające nie tylko grupowanie danych i operacji, ale też ukrywanie dotyczących ich szczegółów w klasie (hermetycznej jednostce), dzięki czemu wewnętrzne mechanizmy klasy nie są ujawniane. Zmniejsza to ryzyko niewłaściwej modyfikacji danych przez jednostkę wywołującą lub pisania kodu na podstawie wewnętrznej implementacji, która może się w przyszłości zmienić.

Modyfikatory dostępu mają umożliwiać hermetyzację. Modyfikator `public` pozwala jawnie określić, że dopuszczalny jest dostęp do danego pola spoza klasy `Employee` (na przykład w klasie `Program`).

Przyjmij jednak teraz, że klasa `Employee` obejmuje pole `Password` z hasłem. Możliwe powinno być wywołanie metody `Logon()` obiektu typu `Employee` w celu sprawdzenia poprawności hasła. Jednak dostęp do pola `Password` takiego obiektu spoza klasy nie powinien być dozwolony.

Aby zdefiniować pole `Password` jako ukryte i niedostępne spoza zawierającej je klasy, należy podać modyfikator dostępu w postaci słowa kluczowego `private`, a nie `public` (zobacz listing 6.15). W efekcie pole `Password` będzie dostępne tylko w klasie `Employee`.

Listing 6.15. Używanie modyfikatora dostępu `private`

```
public class Employee
{
    public string FirstName;
    public string LastName;
    public string? Salary;
    // Niezaszyfrowane hasła są tu używane tylko w celach edukacyjnych;
    // nie zaleca się stosowania tego podejścia.
    private string Password;
    private bool IsAuthenticated;

    public bool Logon(string password)
    {
        if (Password == password)
        {
            IsAuthenticated = true;
        }
        return IsAuthenticated;
    }

    public bool GetIsAuthenticated()
    {
        return IsAuthenticated;
    }
    // ...
}
```

```
class Program
{
```

```

static void Main()
{
    Employee employee = new Employee();

    employee.FirstName = "Inigo";
    employee.LastName = "Montoya";

    // ...

    // Pole Password jest prywatne, dlatego
    // nie można uzyskać do niego dostępu spoza klasy.
    // Console.WriteLine(
    //     ("Password = { employee.Password }");
}
// ...
}

```

Choć na listingu 6.15 nie jest to pokazane, modyfikator dostępu `private` można dodać także do metody.

Jeśli przed składową nie podano żadnego modyfikatora, w deklaracji domyślnie używany jest modyfikator `private`. Oznacza to, że składowe domyślnie są prywatne. Programista musi jawnie określić, że dana składowa ma być publiczna.

Właściwości

W poprzednim podrozdziale („Modyfikatory dostępu”) pokazano, jak używać słowa kluczowego `private` do ukrycia hasła i zablokowania dostępu do niego spoza klasy. Tego rodzaju hermetyzacja często jest jednak zbyt ograniczająca. Czasem trzeba na przykład zdefiniować pola dostępne do odczytu dla klas zewnętrznych, ale modyfikowane tylko wewnętrznie. W innej sytuacji programista może umożliwić zapis określonych danych w klasie, ale chce sprawdzać poprawność wprowadzonych zmian. W jeszcze innym scenariuszu potrzebne może być generowanie danych „w locie”. Tradycyjnie w językach obsługuje się takie sytuacje przez oznaczenie pola jako prywatnego i udostępnienie gettera oraz settera pozwalających pobierać i modyfikować dane. W kodzie na listingu 6.16 pola `FirstName` i `LastName` są prywatne. Publiczne gettery i settery dla tych pól umożliwiają pobieranie i modyfikowanie danych.

Listing 6.16. Deklarowanie getterów i setterów

```

public class Employee
{
    private string FirstName;
    // Getter dla pola FirstName.
    public string GetFirstName()
    {
        return FirstName;
    }
    // Setter dla pola FirstName.
    public void SetFirstName(string newFirstName)
    {
        if (newFirstName != null && newFirstName != "")
        {
            FirstName = newFirstName;
        }
    }
}

```

```

    }

private string LastName;
// Getter dla pola LastName.
public string GetLastName()
{
    return LastName;
}
// Setter dla pola LastName.
public void SetLastName(string newLastName)
{
    if (newLastName != null && newLastName != "")
    {
        LastName = newLastName;
    }
}
// ...
}

```

Niestety, te zmiany utrudniają pisanie kodu w klasie Employee. Nie można teraz zastosować operatora przypisania w celu ustawienia danych w klasie. Nie można też uzyskać dostępu do danych bez wywoływania metody.

Deklarowanie właściwości

Ponieważ opisany wzorzec jest potrzebny bardzo często, projektanci języka C# udostępnili specjalną składnię do korzystania z niego. Jest to składnia do tworzenia **właściwości** (zobacz listing 6.17 i dane wyjściowe 6.5).

Listing 6.17. Definiowanie właściwości

```

class Program
{
    static void Main()
    {
        Employee employee = new Employee();

        // Wywołanie settora właściwości FirstName.
        employee.FirstName = "Inigo";

        // Wywołanie gettera właściwości FirstName.
        System.Console.WriteLine(employee.FirstName);
    }
}

public class Employee
{
    // Właściwość FirstName.
    public string FirstName
    {
        get
        {
            return _FirstName;
        }
    }
}

```

```

set
{
    _FirstName = value;
}
private string _FirstName;

// ...
}

```

DANE WYJŚCIOWE 6.5.

Inigo

Pierwszą rzeczą, na jaką warto zwrócić uwagę na listingu 6.17, wcale nie jest kod właściwości, ale kod w klasie Program. Choć nie istnieją już pola o nazwach FirstName i LastName, w kodzie klasy Program tego nie widać. Składnia używana do uzyskania dostępu do imienia i nazwiska pracownika w ogóle się nie zmieniła. Nadal można przypisać imię lub nazwisko za pomocą prostego operatora przypisania (na przykład employee.FirstName = "Inigo").

Ważne jest to, że właściwości zapewniają składnię, która w trakcie programowania działa podobnie jak zwykłe pole. W rzeczywistości pole w ogóle nie istnieje. Deklaracja właściwości wygląda tak jak deklaracja pola, ale dalej podane są nawiasy klamrowe z implementacją właściwości. Ta implementacja obejmuje dwie opcjonalne części. Blok get służy do definiowania gettera właściwości. Odpowiada on bezpośrednio funkcjom GetFirstName() i GetLastName() zdefiniowanym na listingu 6.16. Aby uzyskać dostęp do właściwości FirstName, należy wywołać metodę employee.FirstName. Podobnie settory (blok set w implementacji) umożliwiają stosowanie składni typowej do przypisania wartości do pól.

```
employee.FirstName = "Inigo";
```

W składni służącej do definiowania właściwości używane są trzy kontekstowe słowa kluczowe. Słowa get i set określają części właściwości służące do pobierania i do przypisywania wartości. Ponadto w setterze używane jest słowo kluczowe value, określające prawą stronę operacji przypisania. Gdy w metodzie Program.Main() wywoływana jest instrukcja employee.FirstName = "Inigo", zmienna value jest ustawiana w setterze na wartość "Inigo", którą można przypisać do właściwości _FirstName. Najczęściej stosuje się implementację właściwości w postaci pokazanej na listingu 6.17. Gdy wywoływany jest getter (na przykład w wyrażeniu Console.WriteLine(employee.FirstName)), program pobiera wartość z pola _FirstName i wyświetla ją w konsoli.

Początek
7.0

Od wersji C# 7.0 można też deklarować gettery i settory właściwości przy użyciu składowych z ciałem w postaci wyrażenia. Ilustruje to listing 6.18.

Listing 6.18. Definiowanie właściwości za pomocą składowych z ciałem w postaci wyrażenia

```

public class Employee
{
    // Właściwość FirstName.
    public string FirstName
    {
        get

```

```

    {
        return _FirstName;
    }
    set
    {
        _FirstName = value;
    }
}
private string _FirstName;
// Właściwość LastName.
public string LastName
{
    get => _FirstName;
    set => _FirstName = value;
}
private string _LastName;
// ...
}

```

Na listingu 6.18 używane są dwie różne składnie do zaimplementowania identycznych właściwości. W kodzie pisany w praktyce staraj się konsekwentnie stosować tę samą składnię.

Koniec
7.0Początek
3.0

Automatycznie implementowane wartości

W C# 3.0 dodano skróconą wersję składni używanej do tworzenia właściwości. Ponieważ tworzenie właściwości z jednym polem, którego wartość jest ustawiana i pobierana za pomocą gettera i settera, jest tak proste i częste (zobacz implementacje właściwości FirstName i LastName), od wersji C# 3.0 kompilator umożliwia deklarację właściwości bez implementacji akcesora i deklaracji pola. Na listingu 6.19 pokazano, jak za pomocą tej składni utworzyć właściwości Title i Manager. Wynik działania kodu pokazano w danych wyjściowych 6.6.

Listing 6.19. Automatycznie implementowane właściwości

```

class Program
{
    static void Main()
    {
        Employee employee1 =
            new Employee();
        Employee employee2 =
            new Employee();

        // Wywołanie settera właściwości FirstName.
        employee1.FirstName = "Inigo";

        // Wywołanie gettera właściwości FirstName.
        System.Console.WriteLine(employee1.FirstName);

        // Przypisanie wartości do automatycznie zaimplementowanej właściwości.
        employee2.Title = "Maniak komputerowy";
        employee1.Manager = employee2;

        // Wyświetlanie stanowiska menedżera osoby reprezentowanej przez obiekt employee1.
        System.Console.WriteLine(employee1.Manager.Title);
    }
}

```

```

public class Employee
{
    // Właściwość FirstName.
    public string FirstName
    {
        get
        {
            return _FirstName;
        }
        set
        {
            _FirstName = value;
        }
    }
    private string _FirstName;

    // Właściwość LastName.
    public string LastName
    {
        get => _FirstName;
        set => _FirstName = value;
    }
    private string _LastName;

    public string? Title { get; set; }

    public Employee? Manager { get; set; }

    public string? Salary { get; set; } = "Za niskie";
    // ...
}

```

DANE WYJŚCIOWE 6.6.

Inigo
Maniak komputerowy

Automatycznie implementowane właściwości upraszczają pisanie właściwości (i ułatwiają czytanie kodu). Ponadto gdy zechcesz dodać sprawdzanie poprawności danych w setterze, nie będziesz musiał modyfikować kodu wywołującego daną właściwość, choć zmieni się jej deklaracja (potrzebne będzie dodanie implementacji).

Dalej w książce często natrafisz na opisaną tu składnię z wersji C# 3.0. W dalszych wystąpieniach tej składni nie znajdziesz już informacji o tym, że funkcja ta została wprowadzona w C# 3.0.

Ostatnią rzeczą, o jakiej warto wspomnieć na temat automatycznie deklarowanych właściwości, jest to, że w wersji C# 6.0 można zainicjować je tak jak właściwość Salary na lisingu 6.19:

```
public string? Salary { get; set; } = "Za niskie";
```

W starszych wersjach języka właściwości można było inicjować tylko za pomocą metod (włącznie z konstruktorem, co opisano dalej w rozdziale). Od wersji C# 6.0 automatycznie implementowane właściwości można inicjować w deklaracji, używając składni bardzo podobnej do tej służącej do inicjowania pól.

Wskazówki dotyczące właściwości i pól

Ponieważ można pisać jawnie settery i gettery zamiast właściwości, czasem możesz się zastanawiać, czy lepiej jest zastosować właściwość, czy metodę. Ogólna zasada jest taka, że metody powinny reprezentować operacje, a właściwości — dane. Właściwości mają zapewniać prosty dostęp do prostych danych wymagających tylko prostych obliczeń. Wywołanie właściwości nie powinno być dużo bardziej kosztowne niż dostęp do pola.

Jeśli chodzi o nazwy, zwróć uwagę na to, że na listingu 6.19 nazwa właściwości to FirstName, a nazwa pola zmieniła się w porównaniu do wcześniejszych listingów na _FirstName (zapis z NotacjiPascalowej z przedrostkiem w postaci podkreślenia). Inne często stosowane konwencje nazewnictwa służące do tworzenia pól prywatnych powiązanych z właściwościami to _firstName. Czasem używa się też notacjiWielbłada, podobnie jak dla zmiennych lokalnych³. Należy jednak jej unikać, ponieważ stosuje się ją także dla zmiennych lokalnych i parametrów, co grozi konfliktami nazw. Ponadto aby zachować zgodność z regułami hermetyzacji, pola nie powinny być deklarowanie jako publiczne lub chronione.

Wskazówki

STOSUJ właściwości, aby zapewnić prosty dostęp do prostych danych z prostymi obliczeniami.

UNIKAJ zgłaszania wyjątków w getterach właściwości.

ZACHOWUJ pierwotną wartość właściwości, jeśli właściwość zgłasza wyjątek.

PRZEDKŁADAJ automatycznie implementowane właściwości nad właściwości powiązane z prostym polem, gdy nie jest potrzebny dodatkowy kod.

Niezależnie od tego, który wzorzec nazewnictwa będziesz stosował dla pól prywatnych, dla właściwości standardowo używana jest NotacjaPascalowa. Dlatego właściwości powinny mieć nazwy zgodne ze wzorcem LastName i FirstName oraz reprezentować rzeczownik, frazę nominalną lub przymiotnik. Zdarza się, że nazwa właściwości jest taka sama jak nazwa typu. Może to dotyczyć na przykład właściwości Address typu Address w obiekcie Person.

Wskazówki

ROZWAŻ użycie tej samej wielkości liter dla pola powiązanego z właściwością i dla samej właściwości, przy czym przed nazwą pola dodaj przedrostek _.

NAZYWAJ właściwości za pomocą rzeczowników, fraz nominalnych lub przymiotników.

ROZWAŻ nadanie właściwości nazwy takiej samej jak nazwa typu.

³ Autorzy preferują zapis _FirstName, ponieważ litera *m* przed nazwą jest zbędna, gdy stosowane jest już podkreślenie (_). Ponadto dzięki zastosowaniu tej samej wielkości liter co w nazwie właściwości wystarczy utworzyć jeden łańcuch znaków w szablonach do rozwijania kodu w środowisku Visual Studio; nie trzeba tworzyć po jednym łańcuchu znaków dla nazwy właściwości i nazwy pola.

UNIKAJ tworzenia nazw pól z wykorzystaniem notacji Wielbłąda.

PREFERUJ popredzanie nazw właściwości logicznych członami Is, Can lub Has, jeśli ma to dodatkową wartość.

NIE deklaruj publicznych ani chronionych pól instancji. Udostępnij takie pola za pomocą właściwości.

NAZYWAJ właściwości z wykorzystaniem Notacji Pascalowej.

PRZEDKŁADAJ automatycznie implementowane właściwości nad pola.

PRZEDKŁADAJ automatycznie implementowane właściwości nad standarde właściwości, jeśli nie potrzebujesz dodatkowej logiki.

Używanie właściwości ze sprawdzaniem poprawności

Zauważ, że na listingu 6.20 w metodzie Initialize() klasy Employee właściwość jest używana zamiast pola także w przypisaniu. Choć nie jest to konieczne, można dzięki temu przeprowadzić sprawdzanie poprawności w setterze właściwości w instrukcjach wywoływanych w klasie i poza nią. Pomyśl, co się stanie, jeśli zmodyfikujesz właściwość LastName w taki sposób, by sprawdzała wartość value (czy jest ona różna od null lub pustego łańcucha znaków) przed jej przypisaniem do pola _LastName. Sprawdzanie wartości jest konieczne, ponieważ choć używany jest typ string niedopuszczający wartości null, w jednostce wywołującej takie typy mogą nie być obsługiwane. Metoda może być też wywoływana w wersji C# 7.0 lub starszej, czyli sprzed wprowadzenia obsługi typów referencyjnych dopuszczających wartość null.

Listing 6.20. Sprawdzanie poprawności właściwości

```
public class Employee
{
    // ...
    public void Initialize(
        string newFirstName, string newLastName)
    {
        // Używanie właściwości także wewnętrz
        // klasie Employee.
        FirstName = newFirstName;
        LastName = newLastName;
    }

    // Właściwość LastName.
    public string LastName
    {
        get => _LastName;
        set
        {
            // Sprawdzanie poprawności w trakcie przypisywania wartości właściwości LastName.
            if (value == null)
            {
                // Zgłaszanie błędu.
                // W wersji C# 6.0 należy zastąpić value wywołaniem nameof(value).
                throw new ArgumentNullException("value");
            }
        }
    }
}
```

```
        }
    else
    {
        // Usuwanie odstępów wokół nazwiska.
        value = value.Trim();
        if (value == "")
        {
            // Zgłaszanie błędu.
            throw new ArgumentException(
                // W wersjach starszych niż C# 6.0 użyj value zamiast
                // nameof(value).
                "Właściwość LastName nie może być pusta.", nameof(value));
        }
        else
            _LastName = value;
    }
}
private string _LastName;
// ...
}
```

W nowej wersji kod zgłasza wyjątek, jeśli do właściwości `LastName` przypisywana jest niepoprawna wartość. Przypisanie może się odbywać albo w innej składowej tej samej klasy, albo w wyniku bezpośredniego ustawienia właściwości `LastName` w metodzie `Program.Main()`. Możliwość przechwycenia operacji przypisania i sprawdzenia poprawności parametrów przy zachowaniu interfejsu API podobnego jak dla pól to jedna z zalet właściwości.

Dobrą praktyką jest używanie powiązanego z właściwością pola wyłącznie w kodzie właściwości. Oznacza to, że zawsze należy posługiwać się właściwością, zamiast bezpośrednio wywoływać pole. W wielu sytuacjach zasada ta dotyczy nawet kodu w klasie, w której dana właściwość się znajduje. Jeśli będziesz stosował się do tej praktyki, to gdy dodasz na przykład kod do sprawdzania poprawności, natychmiast zacznie z niego korzystać cała klasa⁴.

Choć zdarza się to rzadko, można ustawić wartość zmiennej `value` w setterze, tak jak na listingu 6.20. Na listingu wywołanie `value.Trim()` powoduje usunięcie odstępów wokół nowego nazwiska.

Przed wersją C# 6.0 należało podać "value" jako wartość argumentu `paramName` wyjątku. Jednak w C# 6.0 można zamiast tego użyć zapisu `nameof(value)` (więcej informacji znajdziesz w zagadnienniu dla zaawansowanych: „Operator `nameof`”). W pozostałej części rozdziału używana jest składnia `nameof(value)`, dlatego jeśli kompilujesz kod na potrzeby wersji C# 5.0 lub starszej, musisz zastąpić tę składnię zapisem "value".

Początek
6.0

⁴ Dalej w rozdziale opisano, że jest inaczej, gdy pole jest oznaczone jako tylko do odczytu. Wtedy wartość można ustawić jedynie w konstruktorze. W wersji C# 6.0 możliwe jest bezpośrednie ustawianie wartości właściwości przeznaczonych tylko do odczytu, co kompletnie eliminuje konieczność stosowania pól tylko do odczytu.

ZAGADNIENIE DLA ZAAWANSOWANYCH

Operator nameof

Jeśli w trakcie sprawdzania poprawności stwierdzisz, że przypisywana nowa wartość jest nieprawidłowa, konieczne jest zgłoszenie wyjątku. Zwykle jest on typu `ArgumentException()` lub `ArgumentNullException()`. Wyjątki obu tych typów przyjmują argument `paramName` typu `string` określający nazwę nieprawidłowego parametru. Na listingu 6.20 argumentem odpowiadającym temu parametrowi jest `value`, jednak w wersji C# 6.0 wprowadzono usprawnienie w postaci operatora `nameof`. Ten operator przyjmuje identyfikator (na przykład zmienną `value`) i zwraca łańcuch znaków reprezentujący tę nazwę (tu jest to "value"). Na listingu 6.20 zastosowano tę technikę do informowania o drugim błędzie.

Zaletą stosowania operatora `nameof` jest to, że gdy zmieni się nazwa identyfikatora, narzędzie używane do refaktoryzacji automatycznie zmieni też argument w wywołaniu `nameof`. Jeśli nie zastosujesz takiego narzędzia, kod się nie skompiluje, co zmusi programistę do ręcznej zmiany argumentu. Oznacza to, że operator `nameof` pozwala nawet wykrywać literówki. Wynika z tego następująca wskazówka: STOSUJ operator `nameof` dla argumentu `paramName` przekazywanego do wyjątków takich jak `ArgumentNullException`, które przyjmują ten parametr. Więcej informacji znajdziesz w rozdziale 18.

Wskazówki

UNIKAJ dostępu do pól powiązanych z właściwością poza nią (nawet w klasie, w której dana właściwość się znajduje).

STOSUJ konstrukcję `nameof(value)` (reprezentującą wartość "value") w argumencie `paramName` w wywołaniach konstruktorów `ArgumentException()` lub `ArgumentNullException()` ("value" to nazwa nadawana niejawnie parametrowi w setterach właściwości).

Właściwości przeznaczone tylko do odczytu i tylko do zapisu

Usunięcie gettera lub settera z właściwości pozwala zmienić sposób dostępu do niej. Właściwości z samym setterem są przeznaczone tylko do zapisu i występują rzadko. Właściwości z samym getterem służą tylko do odczytu. Próby przypisania wartości do takich właściwości kończą się błędem komplikacji. Aby właściwość Id była przeznaczona tylko do odczytu, należy zastosować kod przedstawiony na listingu 6.21.

Listing 6.21. Definiowanie właściwości przeznaczonej tylko do odczytu w wersjach starszych niż C# 6.0

```
class Program
{
    static void Main()
    {
        Employee employee1 = new Employee();
        employee1.Initialize(42);
```

```
// BŁĄD: nie można przypisać wartości do właściwości
// lub indeksera 'Employee.Id'. Ten element służy tylko do odczytu.
// employee1.Id = "490";
}

public class Employee
{
    public void Initialize(int id)
    {
        // Używanie pola, ponieważ właściwość Id nie ma settera
        // (jest przeznaczona tylko do odczytu).
        _Id = id.ToString();
    }

    // ...
    // Deklaracja właściwości Id.
    public string Id
    {
        get => _Id;
        // Brak settera.
    }

    private string _Id;
}
}
```

Na listingu 6.21 wartość pola (`_Id = id`) jest ustawiana w metodzie `Employee Initialize()`, a nie we właściwości. Przypisanie wartości we właściwości spowoduje błąd komplikacji (podobnie stanie się w metodzie `Program.Main()`).

Od wersji C# 6.0 obsługiwane są też **automatycznie implementowane właściwości** tylko do odczytu:

```
public bool[,] Cells { get; } = new bool[2, 3, 3];
```

Jest to znaczne usprawnienie w porównaniu z wersjami starszymi niż C# 6.0, zwłaszcza jeśli wziąć pod uwagę to, jak często stosuje się właściwości tylko do odczytu dla wartości takich jak tablice elementów lub `Id` z listingu 6.21.

Warto pamiętać, że kompilator wymaga, by automatycznie implementowane właściwości tylko do odczytu (podobnie jak pola tylko do odczytu) były inicjowane w konstruktorze lub inicjatorze. W przedstawionym fragmencie kodu używany jest inicjator, choć wartość do właściwości `Cells` można też przypisać w konstruktorze.

Ponieważ zgodnie ze wskazówką nie należy korzystać z pola poza powiązaną z nim właściwością, programiści używający C# 6.0 zauważą, że nie ma potrzeby stosować składni ze starszych wersji języka. Zawsze można wykorzystać automatycznie implementowaną właściwość tylko do odczytu. Jedyny wyjątek może wystąpić, gdy typ danych modyfikowanego pola tylko do odczytu nie pasuje do typu danych właściwości (na przykład pole jest typu `int`, a właściwość tylko do odczytu ma być typu `double`).

Wskazówki

TWÓRZ właściwości tylko do odczytu, jeśli wartości właściwości nie należy zmieniać.

TWÓRZ automatycznie implementowane właściwości tylko do odczytu zamiast właściwości tylko do odczytu powiązanych z polem, jeśli nie należy zmieniać wartości właściwości (dotyczy wersji języka od C# 6.0).

Właściwości obliczane

W niektórych sytuacjach w ogóle nie trzeba tworzyć konkretnego pola powiązanego z właściwością. Zamiast tego getter właściwości zwraca obliczoną wartość, natomiast setter (jeśli istnieje) przetwarza podaną wartość i utrzymuje ją w innych polach składowych. Przyjrzyj się na przykład implementacji właściwości Name z listingu 6.22. Wynik działania kodu pokazano w danych wyjściowych 6.7.

Listing 6.22. Definiowanie właściwości obliczanych

```
class Program
{
    static void Main()
    {
        Employee employee1 = new Employee();

        employee1.Name = "Inigo Montoya";
        System.Console.WriteLine(employee1.Name);

        // ...
    }
}

public class Employee
{
    // ...

    // Właściwość FirstName.
    public string FirstName
    {
        get
        {
            return _FirstName;
        }
        set
        {
            _FirstName = value;
        }
    }
    private string _FirstName;

    // Właściwość LastName.
    public string LastName
    {
```

```
get => _LastName;
set => _LastName = value;
}
private string _LastName;
// ...

// Właściwość Name.
public string Name
{
    get
    {
        return $"{ FirstName } { LastName }";
    }
    set
    {
        // Podział przypisanej wartości na
        // imię i nazwisko.
        string[] names;
        names = value.Split(new char[] { ' ' });
        if (names.Length == 2)
        {
            FirstName = names[0];
            LastName = names[1];
        }
        else
        {
            // Zgłaszanie wyjątku, jeśli nie
            // przypisano imienia i nazwiska.
            throw new System.ArgumentException(
                $"Przypisana wartość '{ value }' jest nieprawidłowa", nameof(value));
        }
    }
    public string Initials => $"{ FirstName[0] } { LastName[0] }";
    // ...
}
```

DANE WYJŚCIOWE 6.7.

Inigo Montoya

Getter właściwości Name złącza wartości zwrocone przez właściwości FirstName i LastName. To połączenie imienia i nazwiska nie jest nigdzie zapisywane. Gdy kod przypisuje dane do właściwości Name, wartość podana po prawej stronie przypisania jest rozbijana na imię i nazwisko.

Modyfikatory dostępu w getterach i setterach

Wcześniej wspomniano, że do dobrych praktyk należy nieużywanie pól poza właściwościami. Łamanie tej reguły powoduje pominięcie sprawdzania poprawności i dodatkowych operacji, które mogą się znajdować we właściwości.

⁵ Zobacz ZAGADNIENIE DLA ZAAWANSOWANYCH: „Operator nameof” we wcześniejszej części rozdziału lub kompletne omówienie z rozdziału 18.

Modyfikator dostępu można umieszczać przy getterze lub setterze (ale już nie przy obu tych metodach)⁶, aby zmienić modyfikator ustawiony w deklaracji właściwości. Tę technikę pokazano na listingu 6.23.

Listing 6.23. Dodawanie modyfikatora dostępu do settera

```

class Program
{
    static void Main()
    {
        Employee employee1 = new Employee();
        employee1.Initialize(42);
        // BŁĄD: nie można użyć właściwości ani indeksera 'Employee.Id'
        // w tym kontekście, ponieważ setter jest niedostępny.
        employee1.Id = "490";
    }
}

public class Employee
{
    public void Initialize(int id)
    {
        // Ustawianie właściwości Id.
        Id = id.ToString();
    }

    // ...
    // Deklaracja właściwości Id.
    public string Id
    {
        get => _Id;
        // Od wersji C# 2.0 dozwolone jest dodawanie modyfikatora dostępu.
        private set => _Id = value;
    }
    private string _Id;
}

```

Dzięki użyciu słowa kluczowego **private** do settera właściwość poza klasą **Employee** działa tak, jakby była przeznaczona tylko do odczytu. W klasie **Employee** właściwość umożliwia odczyt i zapis, co pozwala przypisać do niej wartość w konstruktorze. Gdy dodajesz modyfikator dostępu do getterów lub setterów, zadbaj o to, by był on bardziej restrykcyjny niż modyfikator całej właściwości. Próba zadeklarowania właściwości **private** i ustawienie settera na **public** zakończy się błędem komplikacji.

⁶ Tę możliwość wprowadzono w wersji C# 2.0. W C# 1.0 nie można było ustawić innego poziomu dostępu dla gettera i settera właściwości. Nie dało się więc utworzyć publicznego gettera i prywatnego settera, tak by zewnętrzne klasy miały dostęp do właściwości w trybie tylko do odczytu, a kod klasy zachował możliwość zmiany wartości właściwości.

Wskazówki

STOSUJ odpowiednie modyfikatory dostępu w implementacjach getterów i setterów we wszystkich właściwościach.

NIE twórz właściwości tylko do zapisu lub właściwości, w których setter ma wyższy poziom dostępności niż getter.

Koniec
2.0

Właściwości i wywołania metod nie są dozwolone jako wartości parametrów ref i out

W języku C# właściwości mogą być używane w prawie identyczny sposób jak pola. Wyjątkiem jest proces przekazywania wartości parametrów ref i out. Wewnętrznie proces ten polega na przekazaniu adresu pamięci do docelowej metody. Jednak ponieważ właściwości mogą być polami wirtualnymi (niepowiązanymi z rzeczywistym polem), a także mogą być przeznaczone tylko do odczytu lub tylko do zapisu, nie da się przekazać adresu do przechowywanych danych. Dlatego przekazywanie właściwości jako wartości parametrów ref i out jest niedozwolone. Gdy kod musi przekazać właściwość lub wywołanie metody jako wartość takich parametrów, trzeba najpierw skopiować wartość do zmiennej, a potem ją przekazać. Po zakończeniu wykonywania metody kod musi przypisać wartość zmiennej z powrotem do właściwości.

ZAGADNIENIE DLA ZAAWANSOWANYCH

Wewnętrzne mechanizmy właściwości

Na listingu 6.24 pokazano, że gettery i settery w kodzie CIL mają postać `get_FirstName()` i `set_FirstName()`.

Listing 6.24. Kod CIL generowany na podstawie właściwości

```
// ...  
  
.field private string _FirstName  
.method public hidebysig specialname instance string  
    get_FirstName() cil managed  
{  
    // Code size    12 (0xc)  
    .maxstack 1  
    .locals init (string V_0)  
    IL_0000: nop  
    IL_0001: ldarg.0  
    IL_0002: ldfld      string Employee::_FirstName  
    IL_0007: stloc.0  
    IL_0008: br.s       IL_000a  
  
    IL_000a: ldloc.0  
    IL_000b: ret  
} // End of method Employee::get_FirstName  
  
.method public hidebysig specialname instance void  
    set_FirstName(string 'value') cil managed  
{  
    // Code size    9 (0x9)
```

```

.maxstack 8
IL_0000: nop
IL_0001: ldarg.0
IL_0002: ldarg.1
IL_0003: stfld      string Employee::_FirstName
IL_0008: ret
} // End of method Employee::set_FirstName

.property instance string FirstName()
{
    .get instance string Employee::get_FirstName()
    .set instance void Employee::set_FirstName(string)
} // End of property Employee::FirstName

// ...

```

Właściwości wyglądają więc w kodzie CIL podobnie jak zwykłe metody. Równie ważne jest to, że właściwości w kodzie CIL są odrębnym elementem. Na listingu 6.25 pokazano, że gettery i settory to elementy właściwości z kodu CIL, które są tu odrębną konstrukcją. Dlatego języki i kompilatory nie muszą zawsze interpretować właściwości na podstawie konwencji nazewniczych. Zamiast tego właściwości z kodu CIL umożliwiają obsługę specjalnej składni w kompilatorach i edytorech kodu.

Listing 6.25. Właściwości w kodzie CIL są odrębnymi elementami

```

.property instance string FirstName()
{
    .get instance string Program::get_FirstName()
    .set instance void Program::set_FirstName(string)
} // End of property Program::FirstName

```

Zauważ, że na listingu 6.24 gettery i settory będące częścią właściwości obejmują metadane `specialname`. Ten modyfikator jest używany przez środowiska (na przykład Visual Studio) jako opcja, która powoduje ukrycie opatrzonych nią składowych w mechanizmie IntelliSense.

Automatycznie implementowane właściwości działają prawie identycznie jak właściwości zdefiniowane za pomocą powiązanego pola. Kompilator języka C# nie używa wtedy ręcznie zdefiniowanego pola, ale generuje w kodzie CIL pole o nazwie `<NazwaWłaściwości>k_BackingField`. Wygenerowane pole obejmuje atrybut ([zobacz rozdział 18.](#)) `System.Runtime.CompilerServices.CompilerGeneratedAttribute`. Gettery i settory mają ten sam atrybut, ponieważ także one są generowane. Ich implementacja wygląda tak samo jak właściwości z listinguów 5.23 i 5.24.

Początek
3.0

Koniec
3.0

Początek
8.0

Konstruktory

Gdy już dodałeś do klasy pola, co pozwala zapisywać w niej dane, należy pomyśleć o sprawdzaniu poprawności tych informacji. Na listingu 6.3 zobaczyłeś, że można utworzyć obiekt za pomocą operatora `new`. Może to jednak skutkować utworzeniem obiektu o nieprawidłowych danych. Bezpośrednio po przypisaniu nowego obiektu do zmiennej powstaje obiekt typu `Employee`, w którym imię, nazwisko i wynagrodzenie nie są zainicjowane. Na wspomnianym listingu kod przypisuje wartość do niezainicjowanych pól bezpośrednio po utworzeniu obiektu.

Jednak pominięcie etapu inicjowania nie powoduje zgłoszenia ostrzeżenia przez kompilator. Dlatego może powstać obiekt typu Employee z błędnym imieniem i nazwiskiem. W wersji C# 8.0 typy referencyjne niedopuszczające wartości null generują ostrzeżenia sugerujące, że należy zmienić typ danych na dopuszczający null, aby uniknąć problemów z wartością domyślną null. Mimo to inicjowanie jest konieczne, aby uniknąć tworzenia obiektów z polemami zawierającymi nieprawidłowe dane.

Deklarowanie konstruktora

Aby rozwiązać opisany wcześniej problem, należy umożliwić podanie potrzebnych danych w momencie tworzenia obiektu. Służy do tego konstruktor, co pokazano na listingu 6.26.

Listing 6.26. Definiowanie konstruktora

```
public class Employee
{
    // Konstruktor w klasie Employee.
    public Employee(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
    }

    public string FirstName{ get; set; }
    public string LastName{ get; set; }
    public string? Salary{ get; set; } = "Za niskie";

    // ...
}
```

W kodzie pokazano, że aby zdefiniować konstruktor, należy utworzyć metodę bez typu zwracanej wartości i o nazwie identycznej z nazwą klasy. Konstruktor to metoda, którą środowisko uruchomieniowe wywołuje, by zainicjować obiekt. Tu konstruktor przyjmuje parametry określające imię i nazwisko, co pozwala programiście podać te dane w trakcie tworzenia obiektu typu Employee. Na listingu 6.27 pokazano, jak wywołać konstruktor.

Listing 6.27. Wywoływanie konstruktora

```
class Program
{
    static void Main()
    {
        Employee employee;
        employee = new Employee("Inigo", "Montoya");
        employee.Salary = "Za niskie";

        System.Console.WriteLine(
            "{0} {1}: {2}",
            employee.FirstName,
            employee.LastName,
            employee.Salary);
    }
    // ...
}
```

Zauważ, że operator new zwraca tu obiekt podanego typu (choć w deklaracji i implementacji konstruktora nie podano typu zwracanej wartości ani instrukcji return). Ponadto usunięto kod inicjujący imię i nazwisko, ponieważ operacje te są wykonywane w konstruktorze. Przykładowy kod nie inicjuje pola Salary w konstruktorze, dlatego nadal używana jest instrukcja ustawiająca poziom wynagrodzenia.

Programiści powinni zachować ostrożność, gdy jednocześnie stosują przypisania w deklaracjach i przypisania w konstruktorze. Przypisania w konstruktorze odbywają się po przypisaniach wykonywanych w deklaracjach pól (na przykład string Salary = "Za niskie" na listingu 6.5). Dlatego przypisanie w konstruktorze zastąpi wartość ustawioną w czasie deklaracji. Może to prowadzić do błędnej interpretacji kodu przez niedoświadczonego czytelnika, który zakłada, że po utworzeniu obiektu ustawiona będzie wartość z deklaracji pola. Dlatego warto pomyśleć o zastosowaniu stylu programowania, w którym w jednej klasie nie są łączone przypisania w deklaracjach i przypisania w konstruktorze.

■ ZAGADNIENIE DLA ZAAWANSOWANYCH

Szczegóły implementacji operatora new

W tym miejscu opisano wewnętrzną interakcję między operatorem new a konstruktorem. Operator new otrzymuje pustą pamięć od menedżera pamięci, a następnie wywołuje podany konstruktor, przekazując mu referencję do pustej pamięci. Ta referencja jest przekazywana jako niejawny parametr this. Następnie uruchamiany jest łańcuch konstruktorów, między którymi też przekazywana jest wspomniana referencja. Żaden konstruktor nie ma typu zwracanej wartości. Wszystkie konstruktory zwracają void. Gdy łańcuch konstruktorów zakończy pracę, operator new zwróci referencję do pamięci, prowadzącą teraz do pamięci zainicjowanymi danymi.

Konstruktory domyślne

Gdy jawnie dodasz konstruktor, nie będziesz już mógł utworzyć w metodzie Main() obiektu typu Employee bez podawania imienia i nazwiska. Dlatego kod z listingu 6.28 się nie skompiluje.

Listing 6.28. Konstruktor domyślny nie jest już dostępny

```
class Program
{
    static void Main()
    {
        Employee employee;
        // BŁĄD: brak odpowiedniej wersji przeciążonej metody, ponieważ
        // wywołana tu metoda Employee przyjmuje 0 argumentów.
        employee = new Employee();
        // ...
    }
}
```

Jeśli w klasie nie istnieje jawnie zdefiniowany konstruktor, kompilator języka C# tworzy go w trakcie komplikacji. Taki konstruktor nie przyjmuje parametrów i jest **konstruktorem domyślnym**. Gdy jawnie dodasz konstruktor do klasy, kompilator języka C# nie doda konstruktora domyślnego. Dlatego po zdefiniowaniu konstruktora `Employee(string firstName, string lastName)` konstruktor domyślny `Employee()` nie jest dodawany przez kompilator. Można samodzielnie dodać taki konstruktor, jednak wtedy znów możliwe będzie utworzenie obiektu typu `Employee` bez podawania imienia i nazwiska pracownika.

Nie musisz polegać na konstruktorze domyślnym zdefiniowanym przez kompilator. Możesz też jawnie zdefiniować taki konstruktor i na przykład zainicjować w nim wybrane pola określonymi wartościami. Aby zdefiniować konstruktor domyślny, wystarczy zadeklarować konstruktor, który nie przyjmuje parametrów.

Iinicjatory obiektów

W wersji C# 3.0 zespół odpowiedzialny za rozwój języka dodał funkcję inicjowania dostępnych pól i właściwości za pomocą **inicjatora obiektu**, który obejmuje zestaw inicjatorów składowych. Te inicjatory są podane w nawiasie klamrowym po wywołaniu konstruktora tworzącym obiekt. Inicjator każdej składowej to przypisanie wartości do dostępnego pola lub dostępnej właściwości (zobacz listing 6.29).

Listing 6.29. Wywoływanie inicjatora obiektu z jawnym przypisaniem wartości do składowych

```
class Program
{
    static void Main()
    {
        Employee employee1 = new Employee("Inigo", "Montoya")
        { Title = "Maniak komputerowy", Salary = "Za niskie"};
        // ...
    }
}
```

Zauważ, że nawet gdy używany jest inicjator obiektu, obowiązują te same reguły dotyczące konstruktora. Wynikowy kod CIL jest dokładnie taki sam jak w sytuacji, gdy wartości pól i właściwości są przypisywane w odrębnych instrukcjach bezpośrednio po wywołaniu konstruktora. W C# kolejność inicjatorów wyznacza używaną w kodzie CIL sekwencję przypisów wartości do właściwości i pól w instrukcjach po wywołaniu konstruktora.

Zwykle do momentu zakończenia pracy konstruktora wszystkie właściwości powinny zostać zainicjowane sensownymi wartościami domyślnymi. Ponadto dzięki zastosowaniu kodu sprawdzającego poprawność w setterze można ograniczyć ryzyko przypisania nieprawidłowych danych do właściwości. Jednak czasem wartości jednych właściwości mogą spowodować, że w innych właściwościach tego samego obiektu pojawią się nieprawidłowe dane. W takiej sytuacji ze zgłoszeniem wyjątków dotyczących nieprawidłowego stanu należy oczekać do momentu, w którym w powiązanych właściwościach zapisane zostaną docelowe wartości.

Wskazówki

PODAWAJ sensowne wartości domyślne wszystkich właściwości. Zadbaj o to, by wartości domyślne nie powodowały luk w zabezpieczeniach lub niskiej wydajności kodu.

UMOŻLIWIJAJ ustawianie wartości właściwości w dowolnym porządku, nawet jeśli skutkuje to tymczasowo nieprawidłowym stanem obiektu.

ZAGADNIENIE DLA ZAAWANSOWANYCH

Inicjatory kolekcji

Inicjatory kolekcji zostały dodane w wersji C# 3.0, a ich składnia jest podobna do składni inicjatorów obiektów. Inicjatory kolekcji działają podobnie jak inicjatory obiektów, ale dotyczą kolekcji. Umożliwiają przypisywanie wartości elementów kolekcji w trakcie tworzenia jej instancji. Składnia inicjatorów kolekcji jest zapożyczona ze składni tablic i pozwala zainicjować każdy element kolekcji w czasie jej tworzenia. Na przykład w trakcie inicjowania listy obiektów typu Employee możesz podać każdy element w nawiasie klamrowym po wywołaniu konstruktora. Pokazano to na listingu 6.30.

Listing 6.30. Wywoływanie inicjatorów obiektów

```
class Program
{
    static void Main()
    {
        List<Employee> employees = new List<Employee>()
        {
            new Employee("Inigo", "Montoya"),
            new Employee("Kevin", "Bost")
        };
        // ...
    }
}
```

Koniec
3.0

Po przypisaniu wartości do nowej instancji kolekcji wygenerowany przez kompilator kod tworzy po kolei każdy podany obiekt i dodaje go do kolekcji za pomocą metody Add().

ZAGADNIENIE DLA ZAAWANSOWANYCH

Finalizatory

Konstruktorzy określają, co dzieje się w trakcie tworzenia instancji klasy. Do definiowania procesu usuwania obiektów służy w języku C# finalizator. Finalizatory w odróżnieniu od destruktorek z języka C++ nie są uruchamiane natychmiast po wyjściu obiektu z zasięgu. Finalizator jest wykonywany w nieokreślonym czasie po momencie, w którym obiekt staje się nieosiągalny. Mechanizm odzyskiwania pamięci w cyklu przywracania pamięci wykrywa obiekty mające finalizator i zamiast natychmiast zwolnić pamięć zajmowaną przez te obiekty, dodaje je do kolejki finalizacji. Odrębny wątek przetwarzania wszystkie obiekty z tej kolejki

i wywołuje ich finalizatory, po czym usuwa obiekty z kolejki i ponownie udostępnia je mechanizmowi odzyskiwania pamięci. Szczegółowe omówienie tego procesu oraz zwalniania zasobów znajdziesz w rozdziale 10.

Przeciążanie konstruktorów

Konstruktory można przeciążać. Możliwe jest utworzenie kilku konstruktorów, przy czym muszą się one różnić liczbą parametrów lub ich typami. Na przykład na listingu 6.31 pokazano, że można udostępnić konstruktory przyjmujące identyfikator, imię i nazwisko pracownika oraz sam identyfikator.

Listing 6.31. Przeciążanie konstruktora

```
class Employee
{
    public Employee(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
    }

    public Employee(
        int id, string firstName, string lastName )
    {
        Id = id;
        FirstName = firstName;
        LastName = lastName;
    }

    // FirstName i LastName są ustawiane w setterze właściwości Id.
    #pragma warning disable CS8618
    public Employee(int id) => Id = id;
    #pragma warning restore CS8618

    public int Id
    {
        get => Id;
        private set
        {
            // Wyszukiwanie imienia i nazwiska pracownika...
            // ...
        }
    }

    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string? Salary { get; set; } = "Za niskie";
    // ...
}
```

To podejście umożliwia utworzenie w metodzie `Program.Main()` instancji klasy `Employee` z imieniem i nazwiskiem na dwa sposoby — w wyniku przekazania samego identyfikatora pracownika lub przez przekazanie imienia, nazwiska i identyfikatora. Konstruktor z imieniem, nazwiskiem i identyfikatorem należy stosować w trakcie dodawania nowych pracowników

do systemu. Konstruktor z samym identyfikatorem pozwala wczytać dane pracownika z pliku lub bazy danych.

Podobnie jak przy przeciążaniu metod, kilka konstruktorów pozwala zapewnić obsługę prostych scenariuszy wymagających niewielkiej liczby parametrów i złożonych sytuacji z dodatkowymi parametrami. Pomyśl o zastosowaniu opcjonalnych parametrów zamiast przeciążania, tak by domyślne wartości opcjonalnych właściwości były widoczne w interfejsie API. Na przykład sygnatura konstruktora `Person(string firstName, string lastName, int? age = null)` zapewnia dokumentację informującą, że jeśli programista nie poda właściwości Age w obiekcie typu Person, domyślnie ustawiona zostanie wartość null.

Od wersji C# 7.0 możliwe jest implementowanie konstruktorów jako składowych z ciałem w postaci wyrażenia. Oto przykład:

```
// FirstName i LastName są ustawiane w setterze właściwości Id.
#pragma warning disable CS8618
public Employee(int id) => Id = id;
```

W tym kodzie właściwość Id jest wywoływana w celu przypisania wartości do właściwości FirstName i LastName. Niestety, kompilator nie wykrywa tego przypisania, dlatego od wersji C# 8.0 zgłasza ostrzeżenie z sugestią, aby oznaczyć te właściwości jako dopuszczające wartości null. Ponieważ kod ustawia wartości tych właściwości, ostrzeżenie zostaje wyłączone.

Wskazówki

STOSUJ te same nazwy dla parametrów konstruktora (w notacji Wielbłąda) i właściwości (w Notacji Pascalowej), gdy parametry konstruktora służą tylko do ustawiania właściwości.

UDOSTĘPNIAJ opcjonalne parametry w konstruktorze i/lub wygodne wersje przeciążonego konstruktora inicjujące właściwości dobrymi wartościami domyślnymi.

UMOŻLIWIAJ ustawianie właściwości w dowolnym porządku, nawet jeśli skutkuje to tymczasowo nieprawidłowym stanem obiektu.

Łączenie konstruktorów w łańcuch — wywoływanie innego konstruktora za pomocą słowa kluczowego this

Zauważ, że na listingu 6.31 kod inicjujący obiekt typu Employee powtarza się w wielu miejscach. Dlatego trzeba go konserwować w różnych fragmentach klasy. Jest to niewielka ilość kodu, jednak można wyeliminować powtórzenia, wywołując jeden konstruktor w innym (czyli tworząc **łańcuch wywołań konstruktorów**) za pomocą **inicjatorów konstruktora**. Inicjatory konstruktora określają, który konstruktor należy wywołać przed uruchomieniem kodu aktualnie wykonywanego konstruktora (zobacz listing 6.32).

Listing 6.32. Wywoływanie jednego konstruktora w innym

```
class Employee
{
    public Employee(string firstName, string lastName)
    {
```

```

FirstName = firstName;
LastName = lastName;
}

public Employee(
    int id, string firstName, string lastName )
: this(firstName, lastName)
{
    Id = id;
}

// FirstName i LastName są ustawiane w setterze właściwości Id.
#pragma warning disable CS8618
public Employee(int id)
{
    Id = id;

    // Wyszukiwanie nazwiska pracownika.
    // ...

    // UWAGA: konstruktory nie można
    // jawnie wywoływać wewnętrznie.
    // this(id, firstName, lastName);
}
#pragma warning restore CS8618

public int Id { get; private set; }
public string FirstName { get; set; }
public string LastName { get; set; }
public string Salary { get; set; } = "Za niskie";

// ...
}

```

Aby wywołać jeden konstruktor w innym w tej samej klasie (dla tego samego obiektu), w C# należy dodać dwukropki i słowo kluczowe `this`, a następnie listę parametrów z deklaracji docelowego konstruktora. Tu konstruktor przyjmujący dwa parametry jest wywoływany przez konstruktor przyjmujący trzy parametry. Często ten wzorzec jest odwarty — w konstruktorze przyjmującym najmniej parametrów wywoływany jest konstruktor o największej liczbie parametrów, przy czym dla nieznanych parametrów przekazywane są wartości domyślne.

ZAGADNIENIE DLA POCZĄTKUJĄCYCH

Centralizowanie inicjowania

Zauważ, że w implementacji konstruktora `Employee(int id)` z listingu 6.32 nie można wywołać instrukcji `this(firstName, lastName)`, ponieważ w wykonywanym konstruktorze takie parametry nie są dostępne. Aby umożliwić wykonywanie całego kodu inicjującego w jednej metodzie, musisz utworzyć odrębną metodę. Tę technikę pokazano na listingu 6.33.

Listing 6.33. Udostępnianie inicjatora

```
public class Employee
{
    // FirstName i LastName są ustawiane w metodzie Initialize().
    #pragma warning disable CS8618
    public Employee(string firstName, string lastName)
    {
        int id;
        // Generowanie identyfikatora pracownika.
        // ...
        Initialize(id, firstName, lastName);
    }

    public Employee(int id, string firstName, string lastName )
    {
        Initialize(id, firstName, lastName);
    }

    public Employee(int id)
    {
        string firstName;
        string lastName;
        Id = id;

        // Wyszukiwanie danych pracownika.
        // ...

        Initialize(id, firstName, lastName);
    }
    #pragma warning restore CS8618

    private void Initialize(
        int id, string firstName, string lastName)
    {
        Id = id;
        FirstName = firstName;
        LastName = lastName;
    }
    // ...
}
```

Tu nowa metoda to `Initialize()`. Przyjmuje ona imię, nazwisko i identyfikator pracownika. Zauważ, że nadal można wywoływać jeden konstruktor w innym, tak jak na listingu 6.32.

Podobnie jak kompilator nie wykrywał ustawiania właściwości `LastName` i `FirstName` we właściwości `Id`, nie wykrywa także ustawiania tych właściwości w metodzie `Initialize`. Dlatego w kodzie wyłączane są związane z tym ostrzeżenia.

Konstruktory a właściwości typów referencyjnych niedopuszczających wartości null

W tym rozdziale konsekwentnie wyłączane jest ostrzeżenie języka C# dotyczące dopuszczania wartości null:

CS8618: Nienullowały element jest niezainicjowany. Rozważ zadeklarowanie elementu jako nullowego.

Gdy używasz typu referencyjnego dla (1) pola niedopuszczającego wartości null lub (2) automatycznie implementowanej właściwości niedopuszczającej wartości null, oczywiste jest, że pole lub właściwość trzeba zainicjować przed zakończeniem tworzenia nadzawanego obiektu. Jeśli tego nie zrobisz, pole lub właściwość będzie mieć wartość domyślną null, dla tego nie należy deklarować ich jako niedopuszczających wartości null.

Problem polega na tym, że pola i właściwości niedopuszczające wartości null często są inicjowane pośrednio, poza bezpośrednim zasięgiem konstruktora, a tym samym poza zasięgiem kodu analizowanego przez kompilator (nie ma tu znaczenia, że takie pola lub właściwości są w rzeczywistości inicjowane, na przykład za pomocą metody lub właściwości wywoływanej przez konstruktor⁷). Oto kilka przykładów:

- Prosta właściwość ze sprawdzaniem poprawności, badająca, czy wartość przypisywana do pola jest różna od null, a następnie ustawiająca wartość w polu, które kompilator zgłasza jako niezainicjowane (listing 6.20).
- Obliczana właściwość Name (na przykład z listingu 6.22), która ustawia w klasie inne właściwości lub pola niedopuszczające wartości null.
- Scentralizowane inicjowanie przeprowadzane w sposób pokazany na listingach 6.32 i 6.33.
- Publiczne właściwości inicjowane przez zewnętrzne agenty, które uruchamiają tworzenie obiektów i inicjują ich właściwości⁸.

W większości sytuacji pola lub automatycznie implementowane właściwości, których typem jest typ referencyjny niedopuszczający wartości null (w tym podrozdziale używane są określenia „pola/właściwości niedopuszczające wartości null”; „typ referencyjny” jest tu domniemany), są ustawiane pośrednio, za pomocą właściwości lub metod wywoływanych przez konstruktor. Niestety, kompilator języka C# nie wykrywa pośredniego przypisywania wartości do pól i właściwości niedopuszczających wartości null.

Ponadto dla wszystkich pól i właściwości niedopuszczających wartości null trzeba z gwarantować, że nie zostanie przypisana im wartość null. Pola trzeba w tym celu opakować we właściwości i w setterach chronić przed ustawieniem wartości null. Pamiętaj, że sprawdzanie poprawności pól wymaga zastosowania się do wskazówki, zgodnie z którą nie należy

⁷ Albo z wykorzystaniem zewnętrznego agenta, na przykład za pomocą mechanizmu refleksji (zobacz rozdział 18.).

⁸ Tak działają na przykład właściwość TestContext w narzędziu MSTest lub obiekty inicjowane z wykorzystaniem wstrzykiwania zależności.

8.0

używać pól poza opakowującymi je właściwościami. Dla samodzielnie implementowanych niedopuszczających wartości null właściwości typu referencyjnego przeznaczonych do odczytu i zapisu należy zastosować sprawdzanie poprawności i zapobiegać przypisywaniu wartości null.

Automatycznie implementowane właściwości niedopuszczające wartości null powinny być dostępne tylko do odczytu; wartości należą im przypisywać w momencie tworzenia obiektu i sprawdzać wtedy, czy są różne od null. Unikaj automatycznie implementowanych właściwości typu referencyjnego niedopuszczających wartości null (zwłaszcza mających publiczne settery), ponieważ trudno jest wtedy zapobiegać przypisaniu im null. Choć można uniknąć ostrzeżeń kompilatora dotyczących niezainicjowanych właściwości niedopuszczających wartości null, jest to niewystarczające rozwiązanie — właściwość jest przeznaczona do odczytu i zapisu, dlatego null może zostać do niej przypisane po utworzeniu. Jest to sprzeczne z zamiarem programisty, który chciał utworzyć właściwość niedopuszczającą wartości null.

Niedopuszczające wartości null właściwości typu referencyjnego przeznaczone do odczytu i zapisu

Na listingu 6.34 pokazane jest, jak przekazać kompilatorowi potrzebne informacje i uniknąć błędnych ostrzeżeń o niezainicjowanych polach lub właściwościach niedopuszczających wartości null. Ostatecznym celem jest umożliwienie programistie poinformowania kompilatora o tym, że właściwości lub pola nie dopuszczają wartości null, aby kompilator mógł poinformować jednostki wywołujące o niedopuszczaniu wartości null przez te elementy.

Listing 6.34. Sprawdzanie poprawności we właściwości niedopuszczającej wartości null

```
public class Employee
{
    public Employee(string name)
    {
        Name = name;
    }

    public string Name
    {

        get => _Name!;
        set => _Name =
            value ?? throw new ArgumentNullException(nameof(value));
    }
    private string? _Name;
    // ...
}
```

Fragment kodu do obsługi właściwości lub pól, które nie dopuszczają wartości null i nie są bezpośrednio inicjowane przez konstruktor, ma kilka ważnych cech (kolejność ich opisu nie jest tu istotna):

1. Setter właściwości sprawdza wartość `null` przed przypisaniem wartości do pola niedopuszczającego `null`. Na listingu 6.34 kod używa do tego operatora `??` i zgłasza wyjątek `ArgumentNullException`, jeśli nową wartością jest `null`.
2. Konstruktor wywołuje metodę lub właściwość, która pośrednio ustawia pole niedopuszczające wartości `null`. Kompilator nie potrafi jednak wykryć, że pole zostało zainicjowane wartością inną niż `null`.
3. Podstawowe pole jest zadeklarowane jako niedopuszczające wartości `null`, aby uniknąć ostrzeżeń kompilatora dotyczących niezainicjowanego pola.
4. Getter zwraca pole, używając operatora braku wartości `null`. Jest to informacja, że pole jest różne od `null`, o czym wiadomo dzięki sprawdzaniu poprawności w setterze.

We właściwości niedopuszczającej wartości `null` deklarowanie podstawowego pola jako dopuszczającego wartość `null` wydaje się bezsensowne. Jest to jednak konieczne, ponieważ kompilator nie wykrywa przeprowadzanych poza konstruktorem przypisań wartości do pól i właściwości niedopuszczających wartości `null`. Na szczęście jest to sytuacja, w której uzasadnione jest używanie przez programistę operatora braku `null` przy zwracaniu wartości pola, ponieważ sprawdzanie wartości `null` w setterze gwarantuje, że pole nigdy nie jest równe `null`.

Automatycznie implementowane właściwości typu referencyjnego tylko do odczytu

Wcześniej w podrozdziale wspomniałem, że automatycznie implementowane właściwości typu referencyjnego powinny być przeznaczone tylko do odczytu, aby uniknąć nieprawidłowych przypisań wartości `null`. Nadal jednak trzeba sprawdzać poprawność parametrów przypisywanych w trakcie tworzenia obiektu. Ilustruje to listing 6.35.

Listing 6.35. Sprawdzanie poprawności automatycznie implementowanych właściwości typu referencyjnego niedopuszczających wartości `null`

```
public class Employee
{
    public Employee(string name)
    {
        Name = name ?? throw new ArgumentNullException(nameof(name));
    }

    public string Name { get; }
}
```

Można dyskutować, czy w automatycznie implementowanych właściwościach typu referencyjnego niedopuszczających wartości `null` dozwolony powinien być prywatny setter. Choć można go używać, bardziej odpowiednie pytanie dotyczy tego, czy klasa może błędnie przypisać `null` do takiej właściwości. Jeśli nie zapewnisz hermetyzacji pola i sprawdzania poprawności w setterze, to jak możesz mieć pewność, że ktoś nie przypisze omyłkowo wartości `null` do danej właściwości? Choć kompilator weryfikuje wymagania programisty w trakcie tworzenia obiektu, wątpliwe jest, że programista zawsze będzie pamiętał o sprawdzaniu wartości `null` w przekazywanych do klasy danych, które nie powinny dopuszczać tej wartości (odpowiednie sprawdzanie ma miejsce w konstruktorze na listingu 6.35).

Wskazówki

IMPLEMENTUJ przeznaczone do odczytu i zapisu samodzielnie tworzone właściwości typu referencyjnego niedopuszczające wartości null, używając podstawowego pola dopuszczającego wartość null; stosuj wtedy operator braku null, gdy zwracasz pole w getterze, i sprawdzanie wartości null w setterze.

PRZYPISUJ wartość do właściwości typu referencyjnego niedopuszczających wartości null przed zakończeniem tworzenia obiektu.

PISZ automatycznie implementowane właściwości typu referencyjnego niedopuszczające wartości null jako przeznaczone tylko do odczytu.

SPRAWDZAJ wartość null we wszystkich właściwościach i polach typu referencyjnego, które nie są inicjowane przed zakończeniem tworzenia obiektu.

Atrybuty dopuszczające wartość null

Zamiast wyłączać możliwość używania wartości null lub ostrzeżenia dotyczące wartości null, czasem warto udostępnić kompilatorowi wskazówki informujące o zamiarach programisty. Jest to możliwe dzięki stosowaniu metadanych. Takie wskazówki można umieszczać bezpośrednio w kodzie w postaci atrybutów (zobacz rozdział 18.). Istnieje siedem atrybutów wartości null. Są one zdefiniowane w przestrzeni nazw System.Diagnostics.CodeAnalysis i identyfikowane jako warunki wstępne i warunki końcowe (tabela 6.1).

Tabela 6.1. Atrybuty dotyczące dopuszczania wartości null

Atrybut	Kategoria	Opis
AllowNull	Warunek wstępny	Wejściowy argument niedopuszczający null może być równy null.
DisallowNull	Warunek wstępny	Wejściowy argument dopuszczający null nie może być równy null.
BeNull	Warunek końcowy	Zwracana wartość niedopuszczająca null może być równa null.
NotNull	Warunek końcowy	Zwracana wartość dopuszczająca null nie może być równa null.
NotNullWhen	Warunek końcowy	Argument wejściowy niedopuszczający null może być równy null, jeśli metoda zwraca określoną wartość logiczną.
NotNullWhenNot	Warunek końcowy	Argument wejściowy dopuszczający null nie może być równy null, jeśli metoda zwraca określoną wartość logiczną.
NotNullIfNotNull	Warunek końcowy	Zwracana wartość nie jest równa null, jeśli argument powiązany z danym parametrem nie jest równy null.

Te atrybuty są pomocne, ponieważ zdarza się, że obsługa dopuszczania wartości null dla typu danych nie wystarcza. Możesz wyeliminować ten problem za pomocą atrybutu, który opisuje w metodzie dane wejściowe (atrybut wartości null używany jako warunek wstępny) lub wyjściowe (atrybut wartości null używany jako warunek końcowy). Warunek wstępny informuje jednostkę wywołującą o tym, czy dana wartość może być równa null. Warunek końcowy określa dopuszczalność wartości null w danych wyjściowych. Rozważ na przykład metody ze wzorca try-get uzyte na listingu 6.36.

Listing 6.36. Używanie atrybutów NotNullWhen i NotNullIfNotNull

```
using System.Diagnostics.CodeAnalysis;
// ...
static bool TryGetDigitAsText(
    char number, [NotNullWhen(true)]out string? text) =>
{
    text = number switch
    {
        '1' => "jeden",
        '2' => "dwa",
        // ...
        '9' => "dziewięć",
        => null
    })- is string;

[return: NotNullIfNotNull("text")]
static public string? TryGetDigitsAsText(string? text)
{
    if (text is null) return null;

    string result = "";
    foreach (char character in text)
    {
        if (TryGetDigitAsText(character, out string? digitText))
        {
            if (result != "") result += '-';
            result += digitText.ToLower();
        }
    }
    return result;
}
```

Zauważ, że w wywołaniu `digitText.ToLower()` w metodzie `TryGetDigitsAsText()` operator ?? nie jest używany, a mimo to kompilator nie zgłasza ostrzeżenia, choć parametr `text` jest zadeklarowany jako dopuszczający wartość null. Dzieje się tak, ponieważ w metodzie `TryGetDigitsAsText()` parametr `text` jest opatrzony atrybutem `NotNullWhen(true)`, który informuje kompilator o tym, że jeśli metoda zwraca true (ta wartość jest podana w atrybucie `NotNullWhen`), programista oczekuje, iż parametr `digitText` będzie różny od null. Atrybut `NotNullWhen` jest deklaracją warunku końcowego i informuje jednostkę wywołującą, że gdy metoda zwraca true, dane wyjściowe (`text`) są różne od null.

Ponadto jeśli wartość podana w metodzie `TryGetDigitsAsText()` jako parametr `text` jest różna od null, zwracana wartość też będzie różna od null. Dzieje się tak, ponieważ warunek wstępny w postaci atrybutu `NotNullIfNotNull` na podstawie wartości wejściowej parametru `text` określa, czy zwracana wartość może być równa null.

ZAGADNIENIE DLA ZAAWANSOWANYCH

Dodawanie modyfikatora dopuszczania wartości null do parametrów określających typ w elementach generycznych

Gdy programista deklaruje składową generyczną lub typ generyczny, czasem chce opatrzyć parametr określający typ modyfikatorem dopuszczania wartości null. Problem polega na tym, że typ bezpośredni dopuszczający null (`Nullable<T>`) różni się od typu referencyjnego dopuszczającego null. Dlatego parametry określające typ opatrzone modyfikatorem dopuszczania wartości null wymagają ograniczenia, które informuje, czy używany jest typ bezpośredni, czy typ referencyjny. Bez takiego ograniczenia pojawi się następujący błąd:

CS8627 Nullowalny parametr typu musi być typem wartości lub nienullowalnym typem referencyjnym. Rozważ dodanie elementu "class", "struct" lub ograniczenia typu.

Jednak gdy logika jest identyczna dla typów bezpośrednich i referencyjnych, implementowanie dwóch różnych metod może być frustrujące — tym bardziej że różne ograniczenia nie oznaczają różnych sygnatur, nie można więc utworzyć przeciążonych wersji metod. Przyjrzyj się kodowi z listingu 6.37.

Listing 6.37. Możliwa zwracana wartość null i atrybut `MaybeNull`

```
// ...
[return: MaybeNull]
static public T GetObject<T>(
    System.Collections.Generic.IEnumerable<T> sequence, Func<T, bool> match)
=>
// ...
```

Wyobraź sobie, że metoda ma zwracać element z kolekcji, jeśli pasuje on do predykatu. Jednak gdy taki element nie istnieje, metoda ma zwracać wartość `default(T)`, która dla typów referencyjnych jest równa null. Niestety, jeśli nie użyjesz ograniczenia, kompilator nie zezwoli na zapis `T?`. Aby uniknąć ostrzeżenia i jednocześnie zadeklarować na potrzeby jednostek wywołujących, że zwracana wartość może być równa null, użyj warunku końcowego w postaci atrybutu `MaybeNull` i zwracanego typu `T` (bez modyfikatora określającego dopuszczalność null).

Koniec
8.0

Początek
7.0

Dekonstruktory

Konstruktory pozwalają pobrać zestaw parametrów i umieścić je w jednym obiekcie. Do wersji C# 7.0 nie istniał mechanizm do wykonywania odwrotnej operacji — przetwarzania hermetycznego elementu na jego części składowe. Oczywiście można było samodzielnie przypisać każdą właściwość do zmiennej. Jednak przy dużej liczbie zmiennych wymagało to wielu odrębnych instrukcji. Dzięki wprowadzonej w C# 7.0 składni dla krotek zadanie stało się dużo łatwiejsze. Można na przykład zadeklarować metodę `Deconstruct()`, taką jak pokazana na listingu 6.38.

Listing 6.38. Definiowanie i używanie dekonstruktora

```

public class Employee
{
    public void Deconstruct(
        out int id, out string firstName,
        out string lastName, out string salary)
    {
        (id, firstName, lastName, salary) =
            (Id, FirstName, LastName, Salary);
    }
    // ...
}

class Program
{
    static void Main()
    {
        Employee employee;
        employee = new Employee("Inigo", "Montoya");
        employee.Salary = "Za niskie";

        employee.Deconstruct(out _, out string firstName,
            out string lastName, out string salary)

        System.Console.WriteLine(
            "{0} {1}: {2}",
            firstName, lastName, salary);
    }
}

```

Taką metodę można wywoływać bezpośrednio, zgodnie z informacjami z rozdziału 5., deklarując parametry `out` wewnętrznie do krotki (tu zakładamy, że zmienne, do których przypisywane są wartości, zostały już wcześniej zadeklarowane):

`(_, firstName, lastName, salary) = employee;`

Ta składnia powoduje wygenerowanie identycznego kodu CIL jak wyróżnione instrukcje z listingu 6.38. Jest to po prostu krótsza składnia (i mniej oczywista, jeśli chodzi o wywołanie metody `Deconstruct()`). Warto zauważyć, że nowa składnia umożliwia użycie składni dla krotek i zmiennych pasujących do przypisywanych parametrów `out`. Nie można zastosować typu krotki (`tuple`):

`(int, string, string, string) tuple = employee;`

Niedozwolone są też elementy z nazwami:

`(int id, string firstName, string lastName, string salary) tuple = employee`

Aby zadeklarować dekonstruktork, trzeba utworzyć metodę o nazwie `Deconstruct`. W tej metodzie typem zwracanej wartości musi być `void` i przyjmować można wyłącznie parametry typu `out` (przynajmniej dwa). Za pomocą metody o takiej sygnaturze można przypisać instancję obiektu do krotki bez bezpośredniego wywoływania metody.

Składowe statyczne

W omówieniu przykładowego programu `HelloWorld` w rozdziale 1. pokrótkę opisano słowo kluczowe `static`. W tym podrozdziale znajdziesz dokładniejsze objaśnienie tego słowa.

Rozważ pewien przykład. Przyjmij, że wartość pola `Id` musi być dla każdego pracownika unikatowa. Jednym ze sposobów osiągnięcia tego celu jest zapisanie licznika, który śledzi identyfikatory wszystkich pracowników. Jeśli jednak wartość licznika jest przechowywana w polu instancji, każde utworzenie obiektu prowadzi do utworzenia nowego pola `NextId`, przez co każdy obiekt typu `Employee` potrzebuje pamięci na to pole. Największy problem polega na tym, że każde utworzenie obiektu typu `Employee` wymaga zmodyfikowania wartości `NextId` we wszystkich wcześniej utworzonych elementach tego typu i zapisania nowej wartości następnego identyfikatora. Potrzebne jest więc pojedyncze pole, współużytkowane przez wszystkie obiekty typu `Employee`.

Porównanie języków — zmienne i funkcje globalne w językach C++ i Visual Basic

C#, w odróżnieniu od wielu innych starszych języków, nie udostępnia zmiennych ani funkcji globalnych. W C# wszystkie pola i metody są częścią klas. Odpowiednikiem globalnych pól i funkcji w języku C# są pola i funkcje statyczne. Działanie zmiennych i funkcji globalnych oraz metod i pól statycznych języka C# jest prawie identyczne. Różnica polega na tym, że do statycznych pól i metod można dodać modyfikatory dostępu, na przykład `private`, co pozwala ograniczyć dostęp do takich składowych i zwiększyć hermetyzację.

Pola statyczne

By zdefiniować dane dostępne w wielu obiektach, zastosuj słowo kluczowe `static`, co pokazano na listingu 6.39.

Listing 6.39. Deklarowanie pól statycznych

```
public class Employee
{
    public Employee(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
        Id = NextId;
        NextId++;
    }
    // ...

    public static int NextId;
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string? Salary { get; set; } = "Za niskie";
    // ...
}
```

W tym przykładzie w deklaracji pola NextId znajduje się modyfikator static. Dlatego powstaje **pole statyczne**. Wszystkie obiekty typu Employee korzystają z jednej lokalizacji pola NextId w pamięci (inaczej niż w przypadku pola Id). W konstruktorze klasy Employee pole Id nowego obiektu tej klasy jest ustawiane na wartość NextId, po czym wartość pola NextId jest zwiększana. W trakcie tworzenia nowego obiektu klasy Employee używana będzie więc większa wartość pola NextId, dlatego w polu Id nowego obiektu znajdzie się inna wartość niż w polu poprzedniego obiektu.

Pola statyczne, podobnie jak **pola instancji** (pola niestatyczne), można inicjować w deklaracji, co pokazano na listingu 6.40.

Listing 6.40. Przypisywanie wartości do pola statycznego w deklaracji

```
public class Employee
{
    // ...
    public static int NextId = 42;
    // ...
}
```

Jeśli pole statyczne (pól instancji to nie dotyczy) nie zostanie zainicjowane, otrzyma wartość domyślną (0, null, false itd.) — tak jak w wywołaniu default(T), gdzie T to nazwa typu. Dlatego dostęp do pola statycznego jest możliwy nawet wtedy, gdy nie przypisano do niego jawnie wartości w kodzie w języku C#.

Pola niestatyczne (pola instancji) wymagają utworzenia nowej lokalizacji w pamięci w każdym obiekcie. Pola statyczne nie należą do instancji, ale do samej klasy. Dlatego dostęp do pola statycznego poza klasą odbywa się z użyciem nazwy klasy. Przyjrzyj się nowej wersji klasy Program z listingu 6.41 (używana jest tu klasa Employee z listingu 6.39).

Listing 6.41. Dostęp do pola statycznego

```
using System;

class Program
{
    static void Main()
    {
        Employee.NextId = 1000000;

        Employee employee1 = new Employee(
            "Inigo", "Montoya");
        Employee employee2 = new Employee(
            "Princess", "Buttercup");

        Console.WriteLine(
            "{0} {1} ({2})",
            employee1.FirstName,
            employee1.LastName,
            employee1.Id);
        Console.WriteLine(
            "{0} {1} ({2})",
            employee2.FirstName,
            employee2.LastName,
```

```
employee2.Id);  
  
Console.WriteLine(  
    $"NextId = { Employee.NextId }");  
}  
  
// ...  
}
```

Wynik działania kodu z listingu 6.41 pokazano w danych wyjściowych 6.8.

DANE WYJŚCIOWE 6.8.

```
Inigo Montoya (1000000)  
Princess Buttercup (1000001)  
NextId = 1000002
```

Do ustawiania i pobierania początkowej wartości pola statycznego `NextId` używana jest nazwa klasy, `Employee`, a nie referencja do instancji tego typu. Jedyne miejsce, w którym można pominąć nazwę klasy, to kod tej klasy (lub klasy pochodnej od niej). Oznacza to, że w konstruktorze `Employee(...)` nie trzeba używać zapisu `Employee.NextId`, ponieważ kod znajduje się w klasie `Employee`, więc kontekst wywołania jest oczywisty. Zasięg zmiennej to tekst kodu programu, w którym danej zmiennej można używać bez podawania pełnej nazwy. Zasięg pola statycznego to tekst jego klasy (i klas pochodnych).

Choć pola statyczne są używane w nieco inny sposób niż pola instancji, nie można zdefiniować w jednej klasie pola statycznego i pola instancji za pomocą tej samej nazwy. Możliwość błędnego wskazania niewłaściwego pola byłaby wtedy zbyt duża, dlatego projektanci języka C# zdecydowali się uniemożliwić pisanie takiego kodu. Stosowanie identycznych nazw powoduje więc konflikt w przestrzeni deklaracji.

ZAGADNIENIE DLA POCZĄTKUJĄCYCH

Dane można powiązać zarówno z klasą, jak i z obiektami

Dane można wiązać zarówno z klasami, jak i z obiektami (podobnie jest z formą i wykonowanymi z niej odlewami).

Na przykład dane dotyczące formy mogą określać liczbę wykonanych przy jej użyciu odlewów, numer następnego odlewu, kolor plastiku wlanego do formy i liczbę odlewów wytwarzanych w ciągu godziny. Odlew też może mieć numer i kolor, a także datę i godzinę wyprodukowania. Choć kolor odlewu odpowiada kolorowi plastiku z formy z czasu produkcji, nie musi odpowiadać kolorowi plastiku obecnie znajdującej się w formie. Odlew nie zawiera też danych określających numer kolejnego wytwarzanego odlewu.

W trakcie projektowania obiektów programiści powinni dbać o odpowiednie zadeklarowanie pól i metod (statycznych oraz instancji). Zwykle metody, które nie korzystają z danych instancji, deklaruje się jako statyczne, a metody wymagające takich danych — jako metody instancji (jeśli instancja nie jest przekazywana jako parametr). Pola statyczne przechowują dane powiązane z klasą, na przykład wartości domyślne nowych instancji lub liczby utworzonych instancji. W polach instancji znajdują się dane powiązane z obiektem.

Metody statyczne

Dostęp do metod statycznych (podobnie jak do pól statycznych) można uzyskać bezpośrednio za pomocą nazwy klasy — na przykład `Console.ReadLine()`. By użyć takiej metody, instancja nie jest potrzebna.

Na listingu 6.42 pokazano przykładowy kod z deklaracją i wywołaniem metody statycznej.

Listing 6.42. Definiowanie statycznej metody w klasie DirectoryInfoExtension

```

public static class DirectoryInfoExtension
{
    public static void CopyTo(
        DirectoryInfo sourceDirectory, string target,
        SearchOption option, string searchPattern)
    {
        if (target[target.Length - 1] !=
            Path.DirectorySeparatorChar)
        {
            target += Path.DirectorySeparatorChar;
        }
        if (!Directory.Exists(target))
        {
            Directory.CreateDirectory(target);
        }

        for (int i = 0; i < searchPattern.Length; i++)
        {
            foreach (string file in
                Directory.GetFiles(
                    sourceDirectory.FullName, searchPattern))
            {
                File.Copy(file,
                    target + Path.GetFileName(file), true);
            }
        }

        // Rekurencyjne kopiowanie podkatalogów.
        if (option == SearchOption.AllDirectories)
        {
            foreach (string element in
                Directory.GetDirectories(
                    sourceDirectory.FullName))
            {
                Copy(element,
                    target + Path.GetFileName(element),
                    searchPattern);
            }
        }
    }

    // ...
    DirectoryInfo directory = new DirectoryInfo(".\\Source");
    directory.MoveTo(".\\Root");
    DirectoryInfoExtension.CopyTo(
        directory, ".\\Target",
        SearchOption.AllDirectories, "*");
    // ...
}

```

Na listingu 6.42 metoda `DirectoryInfoExtension.CopyTo()` przyjmuje obiekt typu `DirectoryInfo` i kopiuje zapisaną w tym obiekcie strukturę katalogów w nowe miejsce.

Ponieważ metody statyczne nie są używane w instancjach, słowo kluczowe `this` jest w tych metodach niedozwolone. Ponadto w metodach statycznych nie można wskazywać bezpośrednio pól ani metod instancji; konieczne jest podanie konkretnej instancji, do której należy dane pole lub dana metoda. Metodą statyczną jest także `Main()`.

Można by się spodziewać, że przedstawiona metoda znajduje się w klasie `System.IO.Directory` lub jest metodą instancji w klasie `System.IO.DirectoryInfo`. Ponieważ tak nie jest, na listingu 6.42 zdefiniowano metodę w zupełnie nowej klasie. W podrozdziale „Metody rozszerzające” w dalszej części rozdziału zobaczysz, jak udostępnić tę metodę jako metodę instancji w klasie `DirectoryInfo`.

Konstruktory statyczne

Oprócz pól i metod statycznych w C# można tworzyć **konstruktory statyczne**. Pozwalają one inicjować samą klasę zamiast jej instancji. Takie konstruktory nie są wywoływane jawnie. Środowisko uruchomieniowe uruchamia je automatycznie w momencie pierwszego dostępu do klasy (w wyniku wywołania zwykłego konstruktora lub dostępu do statycznej metody bądź statycznego pola klasy). Ponieważ konstruktor statyczny nie może być jawnie wywoływany, nie przyjmuje żadnych parametrów.

Konstruktory statyczne służą do inicjowania statycznych danych w klasie określonymi wartościami. Stosowane są zwłaszcza w sytuacji, gdy początkowa wartość jest wyznaczana w bardziej skomplikowany sposób niż przez proste przypisanie w deklaracji. Przyjrzyj się listowi 6.43.

Listing 6.43. Deklarowanie konstruktora statycznego

```
public class Employee
{
    static Employee()
    {
        Random randomGenerator = new Random();
        NextId = randomGenerator.Next(101, 999);
    }

    // ...
    public static int NextId = 42;
    // ...
}
```

Kod na listingu 6.43 przypisuje jako początkową wartość pola `NextId` losową liczbę całkowitą z przedziału od 100 do 1000. Ponieważ określenie początkowej wartości wymaga wywołania metody, kod inicjujący pole `NextId` umieszczono w konstruktorze statycznym, a nie w deklaracji pola.

Jeśli wartość jest przypisywana do pola `NextId` zarówno w konstruktorze statycznym, jak i w deklaracji, nie jest oczywiste, jaka będzie ta wartość po jej zainicjowaniu. Kompilator języka C# generuje kod CIL, w którym przypisanie z deklaracji jest przenoszone i staje się pierwszą

instrukcją w konstruktorze statycznym. Dlatego w polu `NextId` znajdzie się wartość zwrócona przez wywołanie `randomGenerator.Next(101, 999)`, a nie wartość przypisana w deklaracji pola. Tak więc przypisania z konstruktora statycznego są ważniejsze niż przypisania z deklaracji pola (podobnie jest w przypadku pól instancji). Zauważ, że nie jest możliwe zdefiniowanie finalizatora statycznego.

Uważaj, by nie zgłaszać wyjątków w konstruktorze statycznym. Taki wyjątek sprawi, że typ będzie niezdatny do użytku przez cały dalszy czas pracy aplikacji⁹.

ZAGADNIENIE DLA ZAAWANSOWANYCH

Preferuj statycznąinicjację w deklaracji

Konstruktory statyczne są uruchamiane przed pierwszym dostępem do którejkolwiek składowej klasy (pola statycznego, innej składowej statycznej lub konstruktora instancji). Dlatego kompilator dla wszystkich składowych i konstruktorów statycznych typu stosuje test, aby zapewnić, że najpierw wykonany zostanie konstruktor statyczny.

Jeśli konstruktor statyczny nie istnieje, kompilator inicjuje wszystkie składowe statyczne wartościami domyślnymi i nie wykonuje wspomnianego testu. Wtedy statyczne przypisania inicjujące są wywoływanie przed dostępem do pól statycznych, ale niekoniecznie przed wywołaniem pierwszej metody statycznej lub konstruktora instancji. Może to prowadzić do wzrostu wydajności, jeśli inicjowanie składowych statycznych jest kosztowne, ale nie trzeba go wykonywać przed dostępem do pól statycznych. Dlatego powinieneś rozważyć inicjowanie pól statycznych wewnętrznie zamiast przy użyciu konstruktora statycznego lub w deklaracjach.

Wskazówka

ROZWAŻ inicjowanie pól statycznych wewnętrznie zamiast jawnie przy użyciu konstruktorów statycznych lub wartości przypisywanych w deklaracji.

Właściwości statyczne

Początek
2.0

Możesz też deklarować właściwości jako statyczne. Na przykład na listingu 6.44 dane następnego identyfikatora są zapisane we właściwości.

Listing 6.44. Deklarowanie właściwości statycznej

```
public class Employee
{
    // ...
    public static int NextId
    {
        get
        {
            return _NextId;
        }
    }
}
```

⁹ Technicznie ważny jest tu czas pracy domeny aplikacji — używanego w środowisku CLR wirtualnego odpowiednika procesu z systemu operacyjnego.

```

    }
    private set
    {
        _NextId = value;
    }
}
public static int _NextId = 42;
// ...
}

```

Prawie zawsze lepiej jest zastosować właściwość statyczną niż publiczne pole statyczne. Jest tak, ponieważ publiczne pola statyczne mogą być wywoływane w dowolnym miejscu, natomiast właściwości statyczne zapewniają przynajmniej pewien poziom hermetyzacji.

W C# 6.0 całą implementację właściwości NextId (włącznie z niedostępnym powiązanym polem) można uprościć, stosując automatycznie implementowaną właściwość z inicjatorem:

```
public static int NextId { get; private set; } = 42;
```

Klasy statyczne

Niektóre klasy nie zawierają żadnych pól instancji. Przyjrzyj się na przykład klasie SimpleMath, zawierającej funkcje Max() i Min() odpowiadające operacjom matematycznym (zobacz listing 6.45).

Listing 6.45. Deklarowanie klasy statycznej

```

// Klasy statyczne wprowadzono w C# 2.0.
public static class SimpleMath
{
    // Słowo kluczowe params umożliwia podawanie różnej liczby parametrów.
    public static int Max(params int[] numbers)
    {
        // Sprawdzanie, czy w tablicy numbers znajduje się przynajmniej jeden element.
        if (numbers.Length == 0)
        {
            throw new ArgumentException(
                "Tablica numbers nie może być pusta", "numbers");
        }

        int result;
        result = numbers[0];
        foreach (int number in numbers)
        {
            if (number > result)
            {
                result = number;
            }
        }
        return result;
    }

    // Słowo kluczowe params umożliwia podawanie różnej liczby parametrów.
    public static int Min(params int[] numbers)
    {
        // Sprawdzanie, czy w tablicy numbers znajduje się przynajmniej jeden element.
        if (numbers.Length == 0)

```

Początek
6.0
Koniec
6.0

2.0

```
{  
    throw new ArgumentException(  
        "Tablica numbers nie może być pusta", "numbers");  
}  
  
int result;  
result = numbers[0];  
foreach (int number in numbers)  
{  
    if (number < result)  
    {  
        result = number;  
    }  
}  
return result;  
}  
}  
public class Program  
{  
    public static void Main(string[] args)  
    {  
        int[] numbers = new int[args.Length];  
        for (int count = 0; count < args.Length; count++)  
        {  
            numbers[count] = args[count].Length;  
        }  
        Console.WriteLine(  
            $"Długość najdłuższego argumentu = {  
                SimpleMath.Max(numbers) }");  
        Console.WriteLine(  
            $"Długość najkrótszego argumentu = {  
                SimpleMath.Min(numbers) }");  
    }  
}
```

W tej klasie nie występują żadne pola (ani metody) instancji, więc tworzenie instancji tej klasy nie ma sensu. Dlatego w deklaracji klasy znalazło się słowo kluczowe `static`. To słowo przy klasie pełni dwie funkcje. Po pierwsze, uniemożliwia programistom pisanie kodu tworzącego instancje klasy `SimpleMath`. Po drugie, zapobiega deklarowaniu pól i metod instancji w tej klasie. Ponieważ nie można tworzyć instancji tej klasy, tworzenie składowych instancji jest nieuzasadnione. Klasa `Program` z wcześniejszych listingów także dobrze nadaje się do utworzenia jako klasa statyczna, ponieważ również obejmuje same składowe statyczne.

Inną charakterystyczną cechą klas statycznych jest to, że kompilator języka C# automatycznie opatruje je w kodzie CIL modyfikatorami `abstract` i `sealed`. W ten sposób klasa oznaczana jest jako **nierozszerzalna**. Oznacza to, że nie można budować klas pochodnych od danej ani tworzyć instancji tej klasy.

W rozdziale 5. pokazano, że dla klas statycznych (takich jak `SimpleMath`) można stosować dyrektywę `using static`. Na przykład dodanie dyrektywy `using static SimpleMath;` na początku listingu 6.45 pozwala wywołać metodę `Max` bez przedrostka `SimpleMath`:

```
Console.WriteLine(  
    $"Długość najdłuższego argumentu = { Max(numbers) }");
```

Koniec
2.0

Początek
3.0

Początek
6.0

Koniec
6.0

Metody rozszerzające

Pomyśl o klasie System.IO.DirectoryInfo. Służy ona do manipulowania katalogami w systemie plików. Ta klasa umożliwia między innymi wyświetlenie wszystkich plików i podkatalogów (DirectoryInfo.GetFiles()), a także przenoszenie katalogów (DirectoryInfo.MoveTo()). Jednym z mechanizmów, których ta klasa bezpośrednio nie obsługuje, jest kopiowanie. Jeśli potrzebujesz metody do kopирования, musisz sam ją zaimplementować, tak jak zrobiono to wcześniej na listingu 6.42.

Metoda DirectoryInfoExtension.CopyTo() ma deklarację standardową dla metod statycznych. Zauważ jednak, że wywołanie tej metody CopyTo() wygląda inaczej niż wywołanie metody DirectoryInfo.MoveTo(). Nie jest to korzystne. Najlepiej byłoby dodać metodę do klasy DirectoryInfo, tak by po utworzeniu instancji można było wywołać CopyTo() jako metodę instancji (directory.CopyTo()).

Od wersji C# 3.0 można tworzyć metody instancji w innych klasach za pomocą **metod rozszerzających**. W tym celu wystarczy zmienić sygnaturę omawianej metody statycznej w taki sposób, by pierwszy parametr (określający rozszerzany typ danych) był poprzedzony słowem kluczowym `this` (zobacz listing 6.46).

Listing 6.46. Statyczna metoda CopyTo dla klasy DirectoryInfo

```
public static class DirectoryInfoExtension
{
    public static void CopyTo(
        this DirectoryInfo sourceDirectory, string target,
        SearchOption option, string searchPattern)
    {
        // ...
    }
    // ...
    DirectoryInfo directory = new DirectoryInfo(".\\Source");
    directory.CopyTo(".\\Target",
        SearchOption.AllDirectories, "*");
    // ...
}
```

Dzięki metodom rozszerzającym można teraz dołączać „metody instancji” do dowolnej klasy, w tym do klas, które znajdują się w innych podzespołach. Wynikowy kod CIL generowany przez kompilator jest jednak taki sam jak w sytuacji, gdy metoda rozszerzająca jest wywoływana jak zwykła metoda statyczna.

Oto wymagania związane z metodami rozszerzającymi:

- Pierwszy parametr musi określać typ rozszerzany za pomocą metody lub taki, którym dana metoda manipuluje.
- Aby utworzyć metodę rozszerzającą, poprzedź nazwę rozszerzanego typu modyfikatorem `this`.
- Aby użyć metody jako rozszerzającej, zimportuj przestrzeń nazw rozszerzającego typu za pomocą dyrektywy `using` (możesz też umieścić rozszerzającą klasę w tej samej przestrzeni nazw, w której działa kod wywołujący daną metodę).

Jeśli sygnatura metody rozszerzającej jest taka sama jak sygnatura już znajdująca się w rozszerzanym typie (czyli gdy na przykład metoda `CopyTo()` już występuje w klasie `DirectoryInfo`), metoda rozszerzająca może być wywoływana tylko jak normalna metoda statyczna.

Zauważ, że zaleca się tworzenie wyspecjalizowanych wersji typu za pomocą dziedziczenia (zagadnienie to opisano szczegółowo w rozdziale 7.) zamiast stosowania metod rozszerzających. Metody rozszerzające nie pozwalają precyzyjnie zarządzać kodem, ponieważ jeśli dodasz do rozszerzanego typu metodę o takiej samej sygnaturze, jaką ma metoda rozszerzająca, wywoływana będzie nowa metoda, a programista nie otrzyma żadnego ostrzeżenia o zmianie. To zjawisko jest odczuwalne zwłaszcza w sytuacji, gdy nie kontrolujesz kodu źródłowego rozszerzanych klas. Innym (mniejszym) problemem jest to, że choć mechanizm IntelliSense w środowiskach IDE obsługuje metody rozszerzające, w trakcie czytania kodu z wywołaniami takich metod nie jest oczywiste, że używane są metody rozszerzające.

Dlatego powinieneś rzadko posługiwać się metodami rozszerzającymi. Nie definiuj ich na przykład dla typu `object`. W rozdziale 8. wyjaśniono, jak stosować metody rozszerzające razem z interfejsami. W innych kontekstach metody rozszerzające są używane rzadko.

Wskazówka

UNIKAJ pochopnego definiowania metod rozszerzających. Dotyczy to zwłaszcza typów, nad którymi nie masz kontroli.

Koniec
3.0

Hermetyzacja danych

Oprócz stosowania właściwości i modyfikatorów dostępu, omówionych we wcześniejszej części rozdziału, istnieje też kilka innych specjalnych technik hermetyzacji danych w klasie. Istnieją między innymi dwa inne modyfikatory pól. Pierwszy z nich to modyfikator `const`, z którym zetknąłeś się już w trakcie deklarowania zmiennych lokalnych. Drugi modyfikator umożliwia definiowanie pól jako przeznaczonych tylko do odczytu.

Modyfikator `const`

Pola `const` (podobnie jak wartości `const`) zawierają wyznaczoną w trakcie komplikacji wartość, której w czasie wykonywania programu nie można zmieniać. Do deklaracji w stałym polu dobrze nadaje się na przykład wartość `pi`. Na listingu 6.47 pokazano przykładową deklarację pola z modyfikatorem `const`.

Listing 6.47. Deklarowanie stałego pola

```
class ConvertUnits
{
    public const float CentimetersPerInch = 2.54F;
    public const int CupsPerGallon = 16;
    // ...
}
```

Stałe pola są automatycznie tworzone jako statyczne, dlatego nie jest potrzebne nowe pole instancji dla każdego obiektu. Zadeklarowanie pola stałego z jawnie dodanym modyfikatorem static spowoduje błąd komplikacji. Ponadto pola stałe zwykle deklarowane są tylko dla typów przyjmujących literały (na przykład dla typów string, int i double). Stałe pola nie mogą być na przykład typu Program lub System.Guid.

Ważne jest, by wartości podawane w stałych publicznych były niezmienne w czasie. Dobrymi przykładami są liczba pi, liczba Avogadra lub obwód Ziemi. Nie należy jednak zapisywać jako stałych wartości, które mogą się zmieniać w czasie. Złym pomysłem jest zapisywanie jako stałej numeru wersji, liczby ludności lub kursu wymiany walut.

Wskazówki

ZAPISUJ w stałych polach wartości, które nigdy się nie zmieniają.

NIE zapisuj w stałych polach wartości, które zmieniają się w czasie.

ZAGADNIENIE DLA ZAAWANSOWANYCH

Stałe publiczne powinny być niezmiennymi wartościami

Publicznie dostępne stałe powinny być niezmienne, ponieważ modyfikacja wartości stałej niekoniecznie zostanie uwzględniona w podzespołach, który z niej korzysta. Jeśli dany podzespoł korzysta ze stałej z innego podzespołu, wartość stałej jest w trakcie komplikacji zapisywana bezpośrednio w podzespoły używającym stałej. Dlatego jeśli wartość w podzespołach zawierającym stałą się zmieni, ale podzespoły używające tej stałej nie zostaną ponownie skompilowane, zastosowana zostanie pierwotna wartość, a nie jej nowa wersja. Wartości, które mogą zostać w przyszłości zmodyfikowane, należy tworzyć z modyfikatorem readonly.

Modyfikator readonly

Modyfikator readonly (w odróżnieniu od modyfikatora const) jest dostępny tylko dla pól, a nie dla zmiennych lokalnych. Ten modyfikator określa, że wartość pola można modyfikować tylko w konstruktorze lub za pomocą inicjatora w trakcie jej deklarowania. Na listingu 6.48 pokazano, jak zadeklarować pole przeznaczone tylko do odczytu.

Listing 6.48. Deklarowanie pola z modyfikatorem readonly

```
public class Employee
{
    public Employee(int id)
    {
        _Id = id;
    }

    // ...

    public readonly int _Id;
    public int Id
    {
```

```

    get { return _Id; }
}

// Błąd: do pola z modyfikatorem readonly nie można przypisać wartości
// (jest to dozwolone tylko w konstruktorze i inicjatorze zmiennej)
// public void SetId(int id) =>
//     _Id = id;

// ...
}

```

Pola z modyfikatorem `readonly` (inaczej niż pola stałe) mogą mieć różną wartość w poszczególnych instancjach. Wartość pola przeznaczonego tylko do odczytu można zmienić w konstruktorze. Ponadto takie pola można tworzyć jako składowe instancji lub składowe statyczne. Inną ważną cechą takich pól jest to, że ich wartość można ustawić nie tylko w czasie komplikacji, ale też w trakcie wykonywania programu. Ponieważ wartość pola przeznaczonego tylko do odczytu musi być podawana w konstruktorze lub inicjatorze, takie pola są jednym z powodów, dla których kompilator wymaga dostępu do pól w kodzie poza powiązanymi z nimi właściwościami. W innych sytuacjach należy unikać dostępu do pól powiązanych z właściwościami poza daną właściwością.

Inną cechą odróżniającą pola z modyfikatorem `readonly` od pól z modyfikatorem `const` jest to, że te pierwsze nie muszą być typu przyjmującego literały. Można na przykład zadeklarować pole instancji typu `System.Guid`, które jest polem tylko do odczytu.

```
public static readonly Guid ComIUnknownGuid =
    new Guid("00000000-0000-0000-C000-00000000046");
```

Tego typu nie można zastosować jednak dla stałej, gdyż w języku C# nie istnieją literały reprezentujące identyfikatory GUID.

Ponieważ zaleca się, by nie używać pól poza powiązanymi z nimi właściwościami, osoby programujące w języku C# 6.0 zauważają, że modyfikator `readonly` prawie nigdy nie jest potrzebny. Zamiast stosować ten modyfikator, lepiej utworzyć automatycznie implementowaną właściwość tylko do odczytu. We wcześniejszej części rozdziału pokazano, jak to zrobić.

Na listingu 6.49 pokazano następną przykładową właściwość tylko do odczytu.

Listing 6.49. Deklarowanie automatycznie implementowanej właściwości tylko do odczytu

```

class TicTacToeBoard
{
    // Ustawianie planszy, na której żaden gracz nie wykonał posunięcia.
    //   |   |
    //   ---+---+---
    //   |   |
    //   ---+---+---
    //   |   |

    public bool[,] Cells { get; } = new bool[2, 3, 3];
    // Błąd: nie można przypisać wartości do właściwości Cells,
    // ponieważ jest ona przeznaczona tylko do odczytu.
    public void SetCells(bool[,] value) =>
        Cells = new bool[2, 3, 3];

    // ...
}

```

Początek
6.0

Niezależnie od tego, czy używasz automatycznie implementowanych właściwości tylko do odczytu z wersji C# 6.0, czy modyfikatora `readonly` dla pól, zapewnienie niezmienności referencji do tablicy to przydatna technika z obszaru programowania defensywnego. Dzięki temu instancja tablicy pozostaje taka sama, a zapisane w niej elementy mogą się zmieniać. Jeśli referencja nie jest przeznaczona tylko do odczytu, występuje ryzyko przypisania do niej nowej tablicy, co skutkuje usunięciem istniejącej tablicy (a nie aktualizacją poszczególnych jej elementów). Utworzenie tablicy tylko do odczytu nie blokuje więc możliwości modyfikowania jej zawartości. Ograniczenie zapisu dotyczy tylko instancji tablicy (a tym samym i liczb przechowywanych w niej elementów), ponieważ nie można ponownie przypisać do niej nowej instancji. Możliwość zapisu elementów tablicy zostaje jednak zachowana.

Wskazówki

PRZEDKŁADAJ tworzenie automatycznie implementowanych właściwości tylko do odczytu (dostępnych od wersji C# 6.0) nad definiowanie pól tylko do odczytu.

STOSUJ pola z modyfikatorami `public static readonly` do tworzenia predefiniowanych pól instancji, jeśli używasz wersji języka starszych niż C# 6.0.

UNIKAJ przekształcania publicznych pól z modyfikatorem `readonly` z wersji języka starszych niż C# 6.0 na automatycznie implementowane właściwości tylko do odczytu w wersjach od C# 6.0, jeśli chcesz zachować zgodność interfejsów API.

Koniec
6.0

Klasy zagnieżdżone

W klasach oprócz metod i pól można też definiować inne klasy. Są one nazywane **klasami zagnieżdżonymi**. Klasy zagnieżdżone są używane, gdy stosowanie danej klasy poza bieżącą nie ma sensu.

Wyobraź sobie klasę, która obsługuje opcje programu podane w wierszu poleceń. Taka klasa jest specyficzna dla każdego programu, dlatego nie ma powodu, by klasa `CommandLine` była dostępna poza klasą zawierającą metodę `Main()`. Tego rodzaju klasę zagnieżdżoną pokazano na listingu 6.50.

Listing 6.50. Definiowanie klasy zagnieżdżonej

```
// Klasa CommandLine jest zagnieżdżona w klasie Program.
class Program
{
    // Definiowanie klasy zagnieżdżonej służącej do przetwarzania wiersza poleceń.
    private class CommandLine
    {
        public CommandLine(string[] arguments)
        {
            for (int argumentCounter=0;
                argumentCounter<arguments.Length;
                argumentCounter++)
        }
    }
}
```

```

{
    switch (argumentCounter)
    {
        case 0:
            Action = arguments[0].ToLower();
            break;
        case 1:
            Id = arguments[1];
            break;
        case 2:
            FirstName = arguments[2];
            break;
        case 3:
            LastName = arguments[3];
            break;
    }
}

public string? Action { get; };
public string? Id { get; };
public string? FirstName { get; };
public string? LastName { get; };
}

static void Main(string[] args)
{
    CommandLine commandLine = new CommandLine(args);

    switch (commandLine.Action)
    {
        case "new":
            // Tworzenie nowego obiektu reprezentującego pracownika.
            // ...
            break;
        case "update":
            // Aktualizowanie danych w istniejącym obiekcie reprezentującym pracownika.
            // ...
            break;
        case "delete":
            // Usuwanie pliku z danymi pracownika.
            // ...
            break;
        default:
            Console.WriteLine(
                "Employee.exe " +
                "new|update|delete <id> [imię] [nazwisko]");
            break;
    }
}
}

```

Tu klasa zagnieżdżona to Program.CommandLine. Składowe klasy nie wymagają podawania jej nazwy, gdy są w niej używane. To samo dotyczy klas zagnieżdżonych, dlatego wystarczy nazwa CommandLine.

Wyjątkową cechą klas zagnieżdżonych jest możliwość użycia modyfikatora `private` dla samej klasy. Ponieważ przedstawiona klasa ma przetwarzać instrukcje z wiersza poleceń i umieszczać każdy argument w odrębnym polu, klasa `Program.CommandLine` jest przydatna tylko w klasie `Program` z danej aplikacji. Użycie modyfikatora dostępu `private` określa pożądaną dostępność klasy i zapobiega dostępowi do niej poza nadzorowaną klasą. Ten modyfikator można dodawać tylko do klas zagnieżdżonych.

Składowa `this` w klasie zagnieżdżonej oznacza instancję klasy zagnieżdżonej, a nie klasy nadzorowanej. Jednym ze sposobów na uzyskanie w klasie zagnieżdżonej dostępu do instancji klasy nadzorowanej jest jawne przekazanie takiej instancji (za pomocą parametru konstruktora lub innej metody).

Inną ciekawą cechą klas zagnieżdżonych jest to, że mają one dostęp do wszystkich składowych klasy nadzorowanej — w tym do składowych prywatnych. Nie działa to jednak w drugą stronę. Klasa nadzorowana nie ma dostępu do prywatnych składowych klasy zagnieżdżonej.

Klasy zagnieżdżone spotyka się rzadko. Nie należy ich tworzyć, jeśli istnieje możliwość użycia ich poza typem nadzorowanym. Ponadto z podejrliwością traktuj klasy zagnieżdżone z modyfikatorem `public`. Takie klasy mogą wskazywać na kod o niskiej jakości i trudny do zrozumienia.

Wskazówka

UNIKAJ tworzenia publicznie dostępnych typów zagnieżdżonych. Na odstępstwo od tej reguły możesz zdecydować się tylko wtedy, gdy użycie takiego typu w innym miejscu jest mało prawdopodobne lub gdy potrzebujesz możliwości zaawansowanego modyfikowania kodu.

Porównanie języków — klasy wewnętrzne w Javie

W Javie występują nie tylko klasy zagnieżdżone, ale też klasy wewnętrzne. Te ostatnie służą do tworzenia obiektów powiązanych z instancją klasy nadzorowanej i nie określają tylko relacji składowej. W języku C# podobny efekt można uzyskać, tworząc w klasie nadzorowanej pole instancji typu zagnieżdżonego. Metoda fabryczna lub konstruktor pozwalają zapewnić, że referencja do odpowiedniej instancji klasy nadzorowanej zostanie zapisana także w instancji klasy wewnętrznej.

Klasy częściowe

Klasy częściowe¹⁰ są to części klasy, które kompilator może połączyć w kompletną klasę. Choć można zdefiniować dwie klasy częściowe (lub większą ich liczbę) w jednym pliku, zwykła technika ta służy do podziału definicji klasy między kilka plików. Jest to przydatne zwłaszcza w narzędziach generujących lub modyfikujących kod. Dzięki klasom częściowym narzędzia mogą przetwarzać inny plik niż ten, który programista pisze ręcznie.

¹⁰ Wprowadzone w wersji C# 2.0.

Definiowanie klasy częściowej

W C# klasę częściową można zadeklarować, umieszczając kontekstowe słowo kluczowe `partial` bezpośrednio przed słowem `class`, co pokazano na listingu 6.51.

Listing 6.51. Definiowanie klasy częściowej

```
// Plik Program1.cs.
partial class Program
{
}
// Plik Program2.cs.
partial class Program
{
}
```

Tu każdy fragment klasy `Program` jest zapisany w innym pliku podanym w komentarzu.

Innym (oprócz wykorzystania w generatorach kodu) częstym zastosowaniem klas częściowych jest użycie ich do umieszczania klas zagnieżdżonych w odrębnych plikach. Jest to zgodne z konwencją programistyczną, wedle której definicja każdej klasy powinna być zapisana w odrębnym pliku. Na przykład kod z listingu 6.52 umieszcza klasę `Program.CommandLine` w innym pliku niż ten, w którym znajdują się podstawowe składowe klasy `Program`.

Listing 6.52. Definiowanie klasy zagnieżdżonej w odrębnym pliku z klasą częściową

```
// Plik Program.cs.
partial class Program
{
    static void Main(string[] args)
    {
        CommandLine commandLine = new CommandLine(args);

        switch (commandLine.Action)
        {
            // ...
        }
    }
// Plik Program+CommandLine.cs.
partial class Program
{
    // Definicja klasy zagnieżdżonej służącej do przetwarzania instrukcji z wiersza poleceń.
    private class CommandLine
    {
        // ...
    }
}
```

Klasy częściowe nie służą do rozszerzania skompilowanych klas ani klas z innych podzespołów. Są one jedynie narzędziem umożliwiającym podział implementacji klasy między kilka plików z jednego podzespołu.

Metody częściowe

Metody¹¹ częściowe są techniki klas częściowych. Metody częściowe można tworzyć tylko w klasach częściowych. Głównym przeznaczeniem metod częściowych (podobnie jak klas częściowych) jest ułatwienie generowania kodu.

Wyobraź sobie narzędzie do generowania kodu, które generuje plik *Person.Designer.cs* z klasą Person na podstawie tabeli Person z bazy danych. To narzędzie analizuje tabelę, a następnie tworzy właściwości odpowiadające każdej kolumnie z tej tabeli. Problem polega na tym, że narzędzie często nie potrafi wygenerować kodu do sprawdzania poprawności, ponieważ logika takiego kodu wynika z reguł biznesowych nieujętych w definicji tabeli bazy danych. Aby rozwiązać ten problem, autor klasy Person musi dodać kod sprawdzający poprawność. Nie powinien jednak modyfikować bezpośrednio pliku *Person.Designer.cs*, ponieważ jeśli ten plik zostanie ponownie wygenerowany (na przykład w celu uwzględnienia nowej kolumny z bazy danych), poprawki zostaną utracone. Strukturę kodu klasy Person należy więc podzielić na części, tak by wygenerowany kod znajdował się w jednym pliku, a ręcznie napisany kod (z regułami biznesowymi) był zapisany w innym pliku, na który nie wpływa ponowne generowanie kodu. Wcześniej zobaczyłeś, że klasy częściowe dobrze nadają się do podziału klas na wiele plików, jednak nie zawsze są wystarczającym rozwiązaniem. W wielu sytuacjach potrzebne są też **metody częściowe**.

Metody częściowe umożliwiają zadeklarowanie metody bez jej implementowania. Jednak gdy właściwa implementacja zostanie dodana, będzie można ją umieścić w definicji klasy częściowej (zwykle w odrębnym pliku). Na listingu 6.53 pokazano deklarację metody częściowej i implementację klasy Person.

Listing 6.53. Definiowanie metody częściowej ma potrzeby dostępu do jej implementacji

```
// Plik Person.Designer.cs.
public partial class Person
{
    #region Definicje metod umożliwiające rozbudowanie klasy
    partial void OnLastNameChanging(string value);
    partial void OnFirstNameChanging(string value);
    #endregion
    // ...
    public string LastName
    {
        get
        {
            return _LastName;
        }
        set
        {
            if (_LastName != value)
            {
                OnLastNameChanging(value);
                _LastName = value;
            }
        }
    }
}
```

¹¹ Wprowadzone w wersji C# 3.0.

```
}

private string _LastName;

// ...
public string FirstName
{
    get
    {
        return _FirstName;
    }
    set
    {
        if (_FirstName != value)
        {
            OnFirstNameChanging(value);
            _FirstName = value;
        }
    }
}
private string _FirstName;
}

// Plik Person.cs.
partial class Person
{
    partial void OnLastNameChanging(string value)
    {
        if (value is null)
        {
            throw new ArgumentNullException(nameof(value));
        }
        if (value.Trim().Length == 0)
        {
            throw new ArgumentException(
                "Argument LastName nie może być pusty.",
                nameof(value));
        }
    }
}
```

Na listingu z kodem z pliku *Person.Designer.cs* znajdują się deklaracje metod *OnLastNameChanging()* i *OnFirstNameChanging()*. Ponadto właściwości reprezentujące imię i nazwisko wywołują odpowiednie metody zmieniające te dane. Choć deklaracje tych metod są puste, kod się skompiluje. Dzieje się tak, ponieważ deklaracje metod są poprzedzone kontekstowym słowem kluczowym *partial*, po którym następuje nazwa klasy zawierającej te metody.

Na listingu 6.53 znajduje się tylko implementacja metody *OnLastNameChanging()*. Tu implementacja sprawdza podaną nową wartość właściwości *LastName*. Jeśli wartość jest nieprawidłowa, metoda zgłasza wyjątek. Zauważ, że sygnatury metody *OnLastNameChanging()* w obu miejscach pasują do siebie.

Metody częściowe muszą zwracać *void*. Jeśli metoda nie zwraca *void*, a programista nie udostępnia jej implementacji, co zostanie zwrócone w wyniku wywołania niezaimplementowanej metody? Aby uniknąć błędnych założeń dotyczących zwracanej wartości, projektanci języka C# zdecydowali się uniemożliwić tworzenie metod zwracających dane inne niż *void*.

Metody częściowe nie mogą też przyjmować parametrów `out`. Jeśli taka metoda ma zwracać wartość, można przekazać ją za pomocą parametrów `ref`.

Metody częściowe umożliwiają generowanie kodu wywołującego metody, które nie zostały jeszcze zaimplementowane. Ponadto jeśli metoda częściowa nie jest zaimplementowana, nie pojawia się w kodzie CIL. Dzięki temu ilość kodu jest niewielka, a programista zachowuje dużą swobodę.

Podsumowanie

W tym rozdziale objaśniono elementy klas z języka C# oraz obiektowość w tym języku. Opisano tu pola, a także dostęp do nich w instancji klasy.

W rozdziale omówiono też istotną decyzję dotyczącą tego, czy przechowywać dane w konkretnych instancjach, czy w jednym miejscu dla wszystkich instancji danego typu. Dane statyczne są powiązane z klasą, a dane instancji są przechowywane w poszczególnych obiektach.

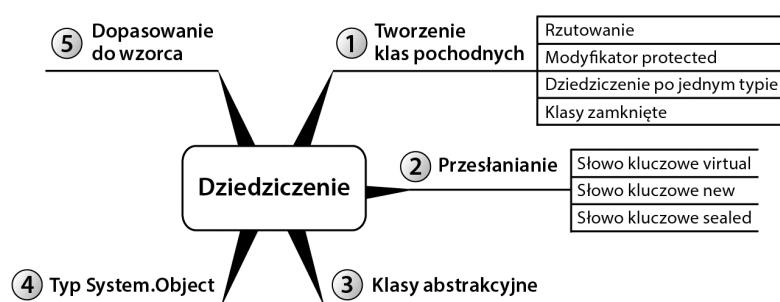
Ponadto opisano tu hermetyzację oraz powiązane z nią modyfikatory dostępu do metod i danych. Przedstawiono też używany w C# mechanizm właściwości i pokazano, jak wykorzystać go do hermetyzacji pól prywatnych.

Następny rozdział dotyczy tworzenia relacji między klasami za pomocą dziedziczenia. Znajdziesz tam wyjaśnienie korzyści, jakie zapewnia ta obiektowa technika.

7

Dziedziczenie

ROZDZIALE 6. WYJAŚNIONO, ŻE JEDNA KLASA może używać innych klas za pomocą pól i właściwości. W tym rozdziale opisano, jak za pomocą dziedziczenia między klasami budować hierarchie klas tworzące relację „jest odmianą”.



ZAGADNIENIE DLA POCZĄTKUJĄCYCH

Definicje z obszaru dziedziczenia

W rozdziale 6. przedstawiono ogólne omówienie dziedziczenia. Oto przegląd zdefiniowanych pojęć:

- *Dziedziczyć* lub *tworzyć typ pochodny* — utworzyć wyspecjalizowaną wersję klasy bazowej, aby dołączyć dodatkowe składowe lub zmodyfikować składowe z klasy bazowej.
- *Typ pochodny* lub *podtyp* — wyspecjalizowany typ, który dziedziczy składowe bardziej ogólnego typu.
- *Typ bazowy* lub *nadtyp* — ogólny typ, którego składowe dziedziczy typ pochodny.

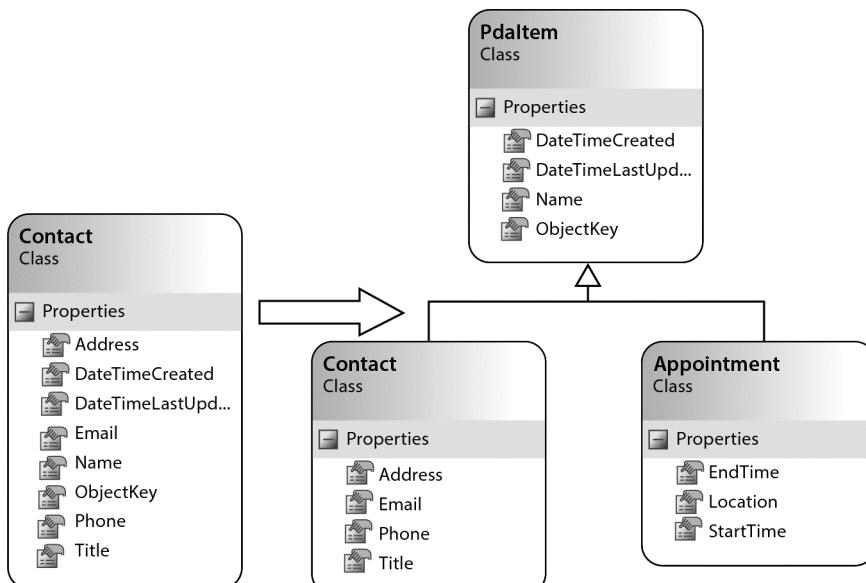
Dziedziczenie pozwala tworzyć relację „jest odmianą”. Typ pochodny zawsze jest pośrednio takŜe typem bazowym. Dysk twardy jest odmianą urządzenia pamięci masowej. TakŜe inne typy pochodne od typu takich urządzeń są odmianą urządzenia pamięci masowej. Zauważ, Ŝe relacja ta nie obowiązuje w drugą stronę. Urządzenie pamięci masowej nie zawsze jest dyskiem twardym.

■ Uwaga

Dziedziczenie w kodzie służy do definiowania relacji „jest odmianą” między dwoma klasami, gdzie klasa pochodna jest wyspecjalizowaną wersją klasy bazowej.

Tworzenie klas pochodnych

Programista często chce rozszerzyć dany typ i dodać do niego nowe elementy — na przykład operacje i dane. Dziedziczenie służy właśnie temu. Jeśli istnieje klasa Person, możesz utworzyć klasę Employee, która dodatkowo obejmuje właściwości EmployeeId i Department. Można też zastosować odwrotne podejście. Założymy, że w systemie komputera kieszonkowego istnieje klasa Contact, a programista chce dodać do systemu obsługę kalendarza. W tym celu tworzy klasę Appointment. Jednak zamiast ponownie definiować metody i właściwości wspólne obu wymienionym klasom, może przeprowadzić **refaktoryzację** klasy Contact. Może na przykład przenieść wspólne metody i właściwości z klasy Contact do klasy bazowej PdaItem, po której dziedziczyć będą klasy Contact i Appointment. Takie rozwiązanie pokazano na rysunku 7.1.



Rysunek 7.1. Refaktoryzacja w celu utworzenia klasy bazowej

Wspólnymi elementami są tu Created, LastUpdated, Name, ObjectKey itd. Dzięki dziedziczeniu metody zdefiniowane w klasie bazowej PdaItem są dostępne we wszystkich klasach pochodnych od niej.

Gdy deklarujesz klasę pochodną, po identyfikatorze klasy dodaj dwukropek i nazwę klasy bazowej, tak jak pokazano na listingu 7.1.

Listing 7.1. Tworzenie klasy pochodnej od innej

```
public class PdaItem
{
    [DisallowNull]
    public string? Name { get; set; }

    public DateTime LastUpdated { get; set; }
}
```

```
// Definiowanie klasy Contact pochodnej od PdaItem.
public class Contact : PdaItem
{
    public string Address { get; set; }
    public string Phone { get; set; }
}
```

Listing 7.2 pokazuje, jak uzyskać dostęp do właściwości zdefiniowanych w klasie Contact.

Listing 7.2. Korzystanie z metod dziedziczonych

```
public class Program
{
    public static void Main()
    {
        Contact contact = new Contact();
        contact.Name = "Inigo Montoya";

        // ...
    }
}
```

Choć klasa Contact nie zawiera bezpośrednio zadeklarowanej właściwości Name, wszystkie instancje tej klasy mają dostęp do właściwości Name z klasy PdaItem. Tę właściwość można stosować tak, jakby była częścią klasy Contact. Ponadto inne klasy pochodne od Contact też dziedziczą składowe klasy PdaItem i wszelkich jej klas bazowych. Długość łańcucha dziedziczenia nie ma odczuwalnych w praktyce ograniczeń. Każda klasa pochodna ma wszystkie składowe klas bazowych z całego łańcucha dziedziczenia (zobacz listing 7.3). Oznacza to, że choć klasa Customer nie dziedziczy bezpośrednio po klasie PdaItem, i tak dziedziczy składowe tej ostatniej.

Listing 7.3. Klasa dziedzicząca jedną po drugiej tworzą łańcuch dziedziczenia

```
public class PdaItem : object
{
    // ...
}

public class Appointment : PdaItem
{
    // ...
}
```

```
public class Contact : PdaItem
{
    // ...
}
```

```
public class Customer : Contact
{
    // ...
}
```

Uwaga

W wyniku dziedziczenia każda składowa klasy bazowej występuje też w łańcuchu klas pochodnych.

Na listingu 7.3 klasa `PdaItem` dziedziczy bezpośrednio po klasie `object`. Choć język C# umożliwia pisanie takiego kodu, jest on zbędny, ponieważ wszystkie klasy, które nie są pochodne od innej klasy, dziedziczą właśnie po klasie `object`. Nie trzeba tego jawnie określić.

Uwaga

Jeśli nie została podana inna klasa bazowa, wszystkie klasy domyślnie dziedziczą po klasie `object`.

Rzutowanie między typami bazowymi i pochodnymi

Na listingu 7.4 pokazano, że ponieważ dziedziczenie tworzy relację „jest odmianą”, wartość typu pochodnego zawsze można bezpośrednio przypisać do zmiennej typu bazowego.

Listing 7.4. Niewjawne rzutowanie na typ bazowy

```
public class Program
{
    public static void Main()
    {
        // Wartości typów pochodnych można niewiadomo przekształcać na typy bazowe.
        Contact contact = new Contact();
        PdaItem item = contact;
        // ...

        // Typy bazowe trzeba jawnie rzutować na typy pochodne.
        contact = (Contact)item;
        // ...
    }
}
```

Typ pochodny `Contact` jest też typem `PdaItem`. Dlatego wartość typu `Contact` można bezpośrednio przypisać do zmiennej typu `PdaItem`. Jest to **konwersja niewiadoma**, ponieważ nie wymaga operatora rzutowania i zawsze kończy się powodzeniem (nie powoduje zgłoszenia wyjątku).

Ta relacja jest jednak jednostronna. Obiekt typu PdaItem nie zawsze jest obiektem typu Contact — może być obiektem typu Appointment lub jeszcze innego typu pochodnego. Dlatego rzutowanie z typu bazowego na typ pochodny wymaga **jawnego rzutowania**, które w trakcie wykonywania programu może się zakończyć niepowodzeniem. Aby przeprowadzić jawnie rzutowanie, należy podać w nawiasie typ docelowy, a następnie referencję do pierwotnej wartości, co pokazano na listingu 7.4.

Stosując jawnie rzutowanie, programista informuje kompilator, że wie, co robi. Kompilator języka C# pozwala wtedy na przeprowadzenie konwersji, jeśli typ docelowy jest typem pochodnym od typu pierwotnej wartości. Choć kompilator języka C# dopuszcza na etapie komplikacji jawną konwersję między potencjalnie zgodnymi typami, środowisko CLR sprawdza poprawność tej operacji w czasie wykonywania programu i zgłasza wyjątek, jeśli pierwotny obiekt nie pasuje do docelowego typu.

Kompilator języka C# umożliwia stosowanie operatora rzutowania nawet wtedy, gdy hierarchia typów pozwala na przeprowadzenie niejawnego rzutowania. Na przykład w przypisaniu obiektu contact do zmiennej item można zastosować operator rzutowania w następujący sposób:

```
item = (PdaItem)contact;
```

Możliwe jest też zastosowanie operatora wtedy, gdy jawnie rzutowanie nie jest konieczne:

```
contact = (Contact)contact;
```

Uwaga

Obiekt typu pochodnego można niejawnie przekształcić na obiekt typu bazowego. Konwersja z typu bazowego na typ pochodny wymaga operatora jawnego rzutowania, ponieważ ta operacja może się zakończyć niepowodzeniem. Choć kompilator zezwala na jawnie rzutowanie, jeśli może się ono zakończyć sukcesem, środowisko uruchomieniowe zapobiega błędному rzutowaniu w trakcie wykonywania programu, zgłaszając wyjątek.

ZAGADNIENIE DLA POCZĄTKUJĄCYCH

Rzutowanie w łańcuchu dziedziczenia

Niejawnna konwersja na typ bazowy nie prowadzi do utworzenia nowej instancji. Zamiast tego pierwotna instancja jest następnie używana jak obiekt typu bazowego, a jej możliwości (dostępne składowe) są ograniczone do tego typu. Wyobraź sobie, że napęd CD-ROM zostanie użyty jako urządzenie pamięci masowej. Ponieważ nie wszystkie takie urządzenia obsługują operację wysuwania, napęd CD-ROM traktowany jak urządzenie pamięci masowej też nie wykonuje tego zadania. Dlatego kod z wywołaniem storageDevice.Eject() nie skompiluje się, nawet jeśli pierwotnie utworzony obiekt był typu CDROM i udostępniał metodę Eject().

Podobnie rzutowanie z typu bazowego na typ pochodny powoduje, że użyty zostaje bardziej wyspecjalizowany typ, co zwiększa zakres dostępnych możliwości. Obowiązuje przy tym ograniczenie, zgodnie z którym pierwotny typ obiektu musi być typem docelowym (lub pochodnym od niego).

ZAGADNIENIE DLA ZAAWANSOWANYCH

Definiowanie niestandardowych konwersji

Konwersje między typami są możliwe nie tylko dla typów z tego samego łańcucha dziedziczenia. Możliwe są także konwersje między niepowiązanymi ze sobą typami, na przykład z Address na string i w drugą stronę. Wymaga to udostępnienia operatora konwersji między odpowiednimi typami. C# umożliwia dodawanie do typów operatorów jawnej i niejawnej konwersji. Jeśli operacja może się zakończyć niepowodzeniem (dotyczy to na przykład rzutowania z typu long na typ int), programista powinien zdefiniować operator jawnej konwersji. Jest to ostrzeżenie dla programistów przeprowadzających konwersję, że powinni wykonywać tę operację tylko wtedy, gdy są pewni jej sukcesu. W przeciwnym razie powinni przygotować kod do przechwytywania wyjątków. Także gdy konwersja prowadzi do utraty precyzji, należy stosować konwersjęawną. Na przykład w trakcie przekształcania wartości z typu float na typ int następuje obcięcie części dziesiętnej, której nie da się odzyskać w wyniku rzutowania odwrotnego (z typu int z powrotem na float).

Na listingu 7.5 pokazano sygnaturę przykładowego operatora konwersji.

Listing 7.5. Definiowanie operatorów rzutowania

```
class GPSCoordinates
{
    // ...

    public static implicit operator UTMCoordinates(
        GPSCoordinates coordinates)
    {
        // ...
    }
}
```

Jest to operator niejawnej konwersji z typu GPSCoordinates na typ UTMCoordinates. Podobny operator konwersji można napisać dla odwrotnej operacji. Zauważ, że aby dodać obsługę konwersji jawnej, słowo `implicit` należy zastąpić słowem `explicit`.

Modyfikator dostępu private

Wszystkie składowe klasy bazowej (z wyjątkiem konstruktorów i destruktatorów) są dziedziczone przez klasy pochodne. Jednak to, że składowa zostaje dziedziczona, nie oznacza jeszcze, iż będzie dostępna. Na przykład na listingu 7.6 pole `_Name` ma modyfikator `private` i nie jest dostępne w klasie `Contact`, ponieważ składowe prywatne są dostępne tylko w typie, w którym je zadeklarowano.

Listing 7.6. Składowe prywatne są dziedziczone, ale nie są dostępne w klasach pochodnych

```
public class PdaItem
{
    private string _Name;
    public string Name
```

```
{  
    get { return _Name; }  
    set { _Name = value; }  
}  
// ...  
}
```

```
public class Contact : PdaItem  
{  
    // ...  
}
```

```
public class Program  
{  
    public static void Main()  
    {  
        Contact contact = new Contact();  
  
        // BŁĄD: pole 'PdaItem. _Name' jest niedostępne  
        // z powodu poziomu ochrony.  
        // contact._Name = "Inigo Montoya";  
    }  
}
```

Aby zasada hermetyzacji była respektowana, klasy pochodne nie mają dostępu do składowych zadeklarowanych za pomocą modyfikatora `private`¹. To zmusza programistę klasy bazowej do jawnego określenia, czy klasa pochodna ma mieć dostęp do danej składowej. W przedstawionym kodzie w klasie bazowej zdefiniowany jest interfejs API, w którym wartość pola `_Name` można zmienić tylko za pomocą właściwości `Name`. Dzięki temu jeśli dodany zostanie kod sprawdzający poprawność, będzie on automatycznie wykorzystywany w klasie pochodnej, ponieważ ta od początku nie miała bezpośredniego dostępu do pola `_Name`.

Uwaga

W klasach pochodnych nie ma dostępu do składowych zadeklarowanych w klasie bazowej jako prywatne.

Modyfikator dostępu `protected`

Dzięki modyfikatorom dostępu innym niż `public` i `private` można bardziej precyzyjnie zarządzać hermetyzacją. Możesz zdefiniować w klasach bazowych składowe, które będą dostępne tylko w klasach pochodnych. Przyjrzyj się na przykład właściwości `ObjectKey` z listingu 7.7.

¹ Wyjątkiem jest nietypowy przypadek, gdy klasa pochodna jest jednocześnie klasą zagnieżdżoną klasy bazowej.

Listing 7.7. Składowe z modyfikatorem protected są dostępne tylko w klasach pochodnych

```

using System.IO;

public class PdaItem
{
    public PdaItem(Guid objectKey) => ObjectKey = objectKey;
    protected Guid ObjectKey { get; }
    // ...
}

public class Contact : PdaItem
{
    public Contact(Guid objectKey)
        : base(objectKey) { }

    public void Save()
    {
        // Tworzenie obiektu typu FileStream z nazwą pliku
        // w formacie <ObjectKey>.dat.
        using FileStream stream = File.OpenWrite(
            ObjectKey + ".dat");
        // ...
        stream.Dispose();
    }

    static public Contact Copy(Contact contact) =>
        new Contact(contact.ObjectKey);

    static public Contact Copy(PdaItem pdaItem) =>
        // Blad: brak dostępu do chronionej składowej PdaItem.ObjectKey.
        new Contact(pdaItem.ObjectKey);
    }
}

public class Program
{
    public static void Main()
    {
        Contact contact = new Contact(Guid.NewGuid());

        // BŁĄD: właściwość 'PdaItem.ObjectKey' jest niedostępna
        Console.WriteLine(contact.ObjectKey);
    }
}

```

Właściwość ObjectKey jest zdefiniowana z modyfikatorem dostępu protected. W efekcie właściwość ta poza klasą PdaItem jest dostępna tylko w klasach pochodnych od PdaItem. Klasa Contact dziedziczy po PdaItem, dlatego wszystkie składowe klasy Contact (na przykład Save()) mają dostęp do właściwości ObjectKey. Klasa Program nie dziedziczy po PdaItem, dlatego próba użycia właściwości ObjectKey w klasie Program skutkuje błędem komplikacji.

Uwaga

Składowe chronione z klasy bazowej są dostępne tylko w tej klasie oraz pochodnych od niej klasach z łańcucha dziedziczenia.

Warto zwrócić uwagę na pewien szczegół związany z metodą statyczną `Contact.Copy` (`→(PdaItem pdaItem)`). Programiści często są zaskoczeni tym, że w kodzie klasy `Contact` nie można uzyskać dostępu do chronionej właściwości `ObjectKey` należącej bezpośrednio do obiektu typu `PdaItem`. W końcu klasa `Contact` jest pochodna od `PdaItem`. Jednak obiekt typu `PdaItem` może być jednocześnie obiektem typu `Address`, a klasa `Contact` nie powinna mieć dostępu do składowych chronionych klasy `Address`. Dlatego hermetyzacja chroni przed modyfikacją właściwości `ObjectKey` klasy `Address` w kodzie klasy `Contact`. Udane rzutowanie obiektu na typ `Contact` (`((Contact)pdaItem).ObjectKey`) pozwala ominąć to ograniczenie. Można też użyć wywołania `contact.ObjectKey`. Nadrzędna reguła jest taka, że dostęp do składowej chronionej w klasie pochodnej wymaga ustalenia w czasie komplikacji, że dana składowa chroniona należy do instancji danej klasy pochodnej.

Metody rozszerzające

Metody rozszerzające technicznie nie są składowymi typu, dlatego nie są dziedziczone. Jednak ponieważ obiekt każdej klasy pochodnej może zostać użyty jako obiekt dowolnej z jej klas bazowych, metody rozszerzające jednego typu rozszerzają też wszystkie typy pochodne. Jeśli rozszerzysz klasę bazową `PdaItem`, dodane do niej metody rozszerzające będą dostępne także w klasach pochodnych. Jednak, podobnie jak w przypadku wszystkich metod rozszerzających, jako ważniejsze traktowane są metody z danej instancji. Jeżeli więc w łańcuchu dziedziczenia występuje metoda o identycznej sygnaturze, zostanie ona użyta zamiast metody rozszerzającej.

Metody rozszerzające rzadko są potrzebne w typach bazowych. Jeśli dostępny jest kod typu bazowego, lepiej zmodyfikować właśnie ten typ (dotyczy to zresztą wszystkich metod rozszerzających). Nawet w sytuacji, gdy kod typu bazowego nie jest dostępny, programiści powinni rozważyć, czy nie lepiej będzie dodać metody rozszerzające do interfejsu implementowanego w typie bazowym lub w poszczególnych typach pochodnych. Interfejsy i używanie ich razem z metodami rozszerzającymi omówiono w rozdziale 8.

Dziedziczenie po jednym typie

Teoretycznie w drzewie dziedziczenia można umieścić nieograniczoną liczbę klas. Na przykład klasa `Customer` dziedziczy po klasie `Contact`, ta po klasie `PdaItem`, a ta po klasie `object`. C# jest językiem, w którym obowiązuje **dziedziczenie po jednym typie** (dotyczy to także języka CIL, do którego kompilowany jest kod w języku C#). To oznacza, że klasa nie może dziedziczyć bezpośrednio po dwóch klasach. Nie jest więc możliwe, by klasa `Contact` dziedziczyła bezpośrednio po klasach `PdaItem` i `Person`.

Początek
3.0

Koniec
3.0

Porównanie języków — wielodziedziczenie w języku C++

Dziedziczenie po jednym typie obowiązujące w C# jest jedną z ważnych obiektowych różnic między tym językiem a językiem C++.

W rzadkich sytuacjach, w których potrzebna jest struktura klas z wielodziedziczeniem, możliwym rozwiązaniem jest zastosowania **agregacji**. Wtedy zamiast tworzyć jedną klasę jako pochodną od innej, można umieścić w danej klasie instancję typu drugiej klasy. W C# 8.0 dostępne są dodatkowe mechanizmy agregacji, dlatego omówienie szczegółów implementowania agregacji odłożono do rozdziału 8.

Klasy zamknięte

Poprawne zaprojektowanie klasy możliwej do rozszerzania przez innych programistów za pomocą dziedziczenia to trudne zadanie. Wymaga to testów i przykładowego kodu, by sprawdzić, czy dziedziczenie będzie prawidłowo działać. Na listingu 7.8 pokazano, jak dzięki oznaczeniu klasy jako **zamkniętej** (modyfikator sealed) uniknąć nieoczekiwanych scenariuszy i problemów związanych z dziedziczeniem.

Listing 7.8. Tworzenie klasy zamkniętej w celu uniemożliwienia dziedziczenia

```
public sealed class CommandLineParser
{
    // ...
}

// BŁĄD: nie można tworzyć klas pochodnych od klas zamkniętych.
public sealed class DerivedCommandLineParser :
    CommandLineParser
{
    // ...
}
```

Klasy zamknięte są tworzone za pomocą modyfikatora sealed. Po takich klasach nie można dziedziczyć. Przykładowym typem, w którym zastosowano modyfikator sealed, by uniemożliwić dziedziczenie, jest typ string.

Przesłanianie składowych z klas bazowych

Klasa pochodna dziedziczy wszystkie składowe klasy bazowej z wyjątkiem konstruktorów i destruktörów. Jednak czasem klasa bazowa nie zawiera optymalnej implementacji danej składowej. Pomyśl na przykład o właściwości Name z klasy PdaItem. Jej implementacja jest akceptowalna w klasie pochodnej Appointment. Jednak w klasie Contact właściwość Name powinna zwracać wartość uzyskaną przez połączenie właściwości FirstName i LastName. Natomiast w trakcie przypisywania wartości do właściwości Name należy rozbić dane na właściwości FirstName i LastName. Tak więc nawet jeśli deklaracja właściwości w klasie bazowej jest

odpowiednia dla klasy pochodnej, to implementacja może być niewłaściwa. Dlatego potrzebny jest mechanizm **przesłaniania** implementacji klasy bazowej niestandardową implementacją w klasie pochodnej.

Modyfikator virtual

C# obsługuje przesłanianie metod i właściwości instancji, nie jest jednak możliwe przesłanianie pól ani innych składowych statycznych. Aby przesłonić składową, niezbędne jest dodanie odpowiedniego kodu zarówno w klasie bazowej, jak i w klasie pochodnej. W klasie bazowej wszystkie składowe umożliwiające przesłonięcie trzeba opatrzyć modyfikatorem `virtual`. Jeśli składowa publiczna lub chroniona nie ma modyfikatora `virtual`, w klasach pochodnych nie będzie można jej przesłonić.

Porównanie języków — w Javie metody są domyślnie wirtualne

W Javie metody są domyślnie wirtualne i trzeba je jawnie zamknąć, jeśli mają być niewirtualne.
W języku C# domyślnie składowe są niewirtualne.

Na listingu 7.9 pokazano przykład ilustrujący przesłanianie właściwości.

Listing 7.9. Przesłanianie właściwości

```
public class PdaItem
{
    public virtual string Name { get; set; }
    // ...
}

public class Contact : PdaItem
{
    public override string Name
    {
        get
        {
            return $"{ FirstName } { LastName }";
        }

        set
        {
            string[] names = value.Split(' ');
            // Pominięto kod do obsługi błędów.
            FirstName = names[0];
            LastName = names[1];
        }
    }

    public string FirstName { get; set; }
    public string LastName { get; set; }
}
// ...
```

W klasie PdaItem przy właściwości Name znajduje się modyfikator virtual, a w klasie Contact właściwość Name jest opatrzona słowem kluczowym override. Jeśli pominiesz słowo virtual, wystąpi błąd, a pominięcie słowa override spowoduje wygenerowanie ostrzeżenia (o czym przekonasz się dalej). W języku C# trzeba bezpośrednio dodać słowo kluczowe override do przeciążających metod. Modyfikator virtual informuje, że dana metoda lub właściwość może zostać zastąpiona (przesłonięta) w typie pochodnym.

Porównanie języków — niejawne przesłanianie w językach Java i C++

W C#, inaczej niż w Javie i C++, przesłanianie wymaga dodania słowa kluczowego override w klasie pochodnej. C# nie umożliwia niejawnego przesłaniania. Aby można było przesłonić metodę, składowe w klasie bazowej i pochodnej muszą pasować do siebie i być opatrzone słowami kluczowymi virtual i override. Dodanie słowa kluczowego override powoduje, że implementacja z klasy pochodnej zastępuje implementację z klasy bazowej.

Przesłonięcie składowej powoduje, że środowisko uruchomieniowe wywołuje najbardziej pochodną implementację (zobacz listing 7.10).

Listing 7.10. Środowisko uruchomieniowe wywołuje najbardziej pochodną implementację metody wirtualnej

```
public class Program
{
    public static void Main()
    {
        Contact contact;
        PdaItem item;

        contact = new Contact();
        item = contact;

        // Ustawianie imienia i nazwiska za pomocą zmiennej typu PdaItem.
        item.Name = "Inigo Montoya";

        // Wyświetlanie informacji o ustaleniu
        // właściwości FirstName i LastName.
        Console.WriteLine(
            $"{ contact.FirstName } { contact.LastName }");
    }
}
```

Wynik działania kodu z listingu 7.10 pokazano w danych wyjściowych 7.1.

DANE WYJŚCIOWE 7.1.

Inigo Montoya

Na listingu 7.10 wywoływana jest właściwość item.Name. Zmienna item jest typu PdaItem. Jednak wywołanie skutkuje ustawieniem właściwości FirstName i LastName w zmiennej contact. Reguła jest taka, że gdy środowisko uruchomieniowe natrafi na metodę wirtualną, wywołuje najbardziej pochodną przeciążającą implementację składowej wirtualnej. Tu kod tworzy

instancję klasy Contact i wywołuje właściwość Contact.Name, ponieważ klasa Contact zawiera najbardziej pochodną implementację właściwości Name.

Metody wirtualne udostępniają tylko domyślną implementację — taką, którą w klasach pochodnych można w całości przesłonić. Jednak z powodu złożoności projektu dziedziczenia ważne jest, by rozważyć (i uwzględnić w kodzie) konkretny scenariusz, który wymaga definiowania metod wirtualnych. Jest to lepsze niż domyślne deklarowanie składowych z modyfikatorem `virtual`.

Jest to ważne także dlatego, że przekształcenie metody wirtualnej w niewirtualną może uniemożliwić działanie klas pochodnych przesłaniających tę metodę. Udostępniona składowa wirtualna musi taka pozostać; w przeciwnym razie kod może przestać działać. Dlatego zachowaj ostrożność, gdy tworzysz składową wirtualną. Możesz na przykład użyć modyfikatorów `private protected`.

Porównanie języków

— wiązanie wywołań metod w trakcie konstrukcji w języku C++

W języku C++ wywołania metod w trakcie konstrukcji nie powodują użycia metod wirtualnych. W trakcie konstrukcji typ jest wiązany z typem bazowym, a nie z typem pochodnym, dlatego używana jest implementacja metod wirtualnych z klasy bazowej. W języku C# wybierane są metody wirtualne z najbardziej pochodnego typu. Jest to spójne z zasadą wywoływanego najbardziej pochodnej składowej wirtualnej, nawet gdy pochodny konstruktor nie został w pełni wykonany. Niezależnie od tego w C# należy unikać takich wywołań.

Tylko składowe instancji mogą być wirtualne. Środowisko CLR używa konkretnego typu, określonego w momencie tworzenia instancji, do ustalenia, do jakiego kodu należy kierować wywołania metod wirtualnych. Dlatego metody z modyfikatorem `static virtual` są nieprawidłowe i kompilator nie pozwala na ich tworzenie.

Modyfikator new

Jeśli przeciążająca metoda nie jest opatrzona modyfikatorem `override`, kompilator zgłasza ostrzeżenie podobne do tych z danych wyjściowych 7.2 i 7.3.

DANE WYJŚCIOWE 7.2.

```
warning CS0114: '<derived method name>' hides inherited member  
'<base method name>'. To make the current member override that  
implementation, add the override keyword. Otherwise add the new  
keyword.
```

DANE WYJŚCIOWE 7.3.

```
warning CS0108: The keyword new is required on '<derived property  
name>' because it hides inherited member '<base property name>'
```

Oczywiste rozwiązanie polega na dodaniu modyfikatora `override` (jeśli składowa bazowa jest wirtualna). Jednak, jak wyjaśniono w ostrzeżeniach, można też zastosować modyfikator `new`. Rozważ scenariusz przedstawiony w tabeli 7.1. Jest to przykład ilustrujący ogólniejszy problem **klasy bazowej podatnej na błędy**.

Tabela 7.1. Dlaczego stosować modyfikator new?

Czynności	Kod
Programista A definiuje klasę Person zawierającą właściwości FirstName i LastName.	<pre>public class Person { public string FirstName { get; set; } public string LastName { get; set; } }</pre>
Programista B tworzy klasę Contact pochodną od Person. Nowa klasa zawiera dodatkową właściwość Name. Ponadto programista definiuje klasę Program, której metoda Main() tworzy instancję klasy Contact, przypisuje wartość do właściwości Name, a następnie wyświetla imię i nazwisko.	<pre>public class Contact : Person { public string Name { get { return FirstName + " " + LastName; } set { string[] names = value.Split(' '); // Obsługę błędów pominięto. FirstName = names[0]; LastName = names[1]; } } }</pre>
Później programista A dodaje właściwość Name, ale zamiast zaimplementować getter z instrukcją FirstName + " " + LastName, używa kodu LastName + " " + FirstName. Ponadto nie definiuje tej właściwości jako wirtualnej, natomiast wywołuje ją w metodzie DisplayName().	<pre>// ... public class Person { public string Name { get { return LastName + ", " + FirstName; } set { string[] names = value.Split(", "); // Obsługę błędów pominięto. LastName = names[0]; FirstName = names[1]; } } } public static void Display(Person person) { // Display <LastName>, <FirstName> Console.WriteLine(person.Name); }</pre>

Ponieważ właściwość Person.Name nie jest wirtualna, programista A oczekuje, że metoda Display() wywoła implementację z klasy Person — nawet jeśli użyty zostanie obiekt typu Contact pochodnego od Person. Jednak programista B oczekuje, że gdy typem danych jest Contact, to zawsze wywoływana będzie właściwość Contact.Name. W kodzie programisty B nie ma wywołań Person.Name, ponieważ początkowo właściwość Person.Name nie istniała. Aby umożliwić dodanie właściwości Person.Name bez naruszania działania kodu któregokolwiek z programistów, nie można przyjmować, że właściwość ta od początku miała być wirtualna. Ponadto ponieważ C# wymaga, by do składowych przesyłających jawnie dodawać modyfikator override, trzeba udostępnić inne rozwiązanie. Dodanie składowej do klasy bazowej nie powinno uniemożliwiać komplikacji klasy pochodnej.

Tym rozwiązaniem jest modyfikator new, który ukrywa ponownie zadeklarowaną składową z klasy pochodnej przed klasą bazową. Zamiast wywoływać globalnie najbardziej pochodną składową, w klasie bazowej wywoływana jest wtedy najbardziej pochodna składowa z łańcucha dziedziczenia do miejsca wystąpienia składowej z modyfikatorem new. Jeśli łańcuch dziedziczenia obejmuje tylko dwie klasy, składowa w klasie bazowej będzie działać tak, jakby w klasie pochodnej nie zadeklarowano jej odpowiednika (nawet jeśli w klasie pochodnej znajduje się przesyłająca wersja składowej z klasą bazową). Choć jeżeli programista nie doda ani modyfikatora override, ani modyfikatora new, kompilator zgłosi ostrzeżenie pokazane w danych wyjściowych 7.2 i 7.3, to przyjmie, że użyty został modyfikator new, co pozwala zachować pożądane bezpieczeństwo wersji.

Przyjrzyj się przykładowemu kodowi z listingu 7.11. Wynik jego działania pokazano w danych wyjściowych 7.4.

Listing 7.11. Modyfikatory override i new

```
public class Program
{
    public class BaseClass
    {
        public void DisplayName()
        {
            Console.WriteLine("BaseClass");
        }
    }

    public class DerivedClass : BaseClass
    {
        // OSTRZEŻENIE OD KOMPILATORA: metoda DisplayName() przesyłania
        // odziedziczoną składową. Dodaj słowo kluczowe new, jeśli jest to zamierzane.
        public virtual void DisplayName()
        {
            Console.WriteLine("DerivedClass");
        }
    }

    public class SubDerivedClass : DerivedClass
    {
        public override void DisplayName()
        {
            Console.WriteLine("SubDerivedClass");
        }
    }
}
```

```

}

public class SuperSubDerivedClass : SubDerivedClass
{
    public new void DisplayName()
    {
        Console.WriteLine("SuperSubDerivedClass");
    }
}

public static void Main()
{
    SuperSubDerivedClass superSubDerivedClass
        = new SuperSubDerivedClass();

    SubDerivedClass subDerivedClass = superSubDerivedClass;
    DerivedClass derivedClass = superSubDerivedClass;
    BaseClass baseClass = superSubDerivedClass;

    superSubDerivedClass.DisplayName();
    subDerivedClass.DisplayName();
    derivedClass.DisplayName();
    baseClass.DisplayName();
}
}

```

DANE WYJŚCIOWE 7.4.

```

SuperSubDerivedClass
SubDerivedClass
SubDerivedClass
BaseClass

```

Poniżej wyjaśniono, z czego wynikają takie a nie inne dane wyjściowe:

- SuperSubDerivedClass — metoda SuperSubDerivedClass.DisplayName() wyświetla SuperSubDerivedClass, ponieważ nie istnieje klasa pochodna, a tym samym nie ma przesłaniającej wersji metody.
- SubDerivedClass — SubDerivedClass.DisplayName() to najbardziej pochodna składowa przesłaniająca składową wirtualną z klasy bazowej. Metoda SuperSub→DerivedClass.DisplayName() jest ukryta z powodu użycia modyfikatora new.
- SubDerivedClass — DerivedClass.DisplayName() to metoda wirtualna, a SubDerivedClass.DisplayName() to najbardziej pochodna składowa, która ją przesłania. Podobnie jak wcześniej metoda SuperSubDerivedClass.DisplayName(), jest ukryta z powodu użycia modyfikatora new.
- BaseClass — metoda BaseClass.DisplayName() nie jest wirtualna i nie przesłania żadnej składowej klasy bazowej, dlatego jest wywoływana bezpośrednio.

Modyfikator new nie wpływa na to, jakie instrukcje kompilator generuje w kodzie CIL. Jednak do metody z tym modyfikatorem dodawany jest atrybut metadanych newslot. W kontekście samego języka C# skutkuje to tylko usunięciem ostrzeżenia kompilatora w miejscu, w którym normalnie by się ono pojawiło.

Modyfikator sealed

Stosując modyfikator sealed do klasy, można uniemożliwić tworzenie klas pochodnych od niej. Podobnie można „zamknąć” składowe wirtualne (zobacz listing 7.12). To podejście uniemożliwia klasom pochodnym przesłanianie składowej, którą pierwotnie zadeklarowano jako wirtualną na wyższym poziomie łańcucha dziedziczenia. Taka sytuacja zachodzi, gdy w podklasie B przesłaniana jest składowa z klasy bazowej A, ale programista chce zapobiec dalszemu przesłanianiu tej składowej w klasach pochodnych od B.

Listing 7.12. Stosowanie modyfikatora sealed do zmiennych składowych

```
class A
{
    public virtual void Method()
    {
    }
}

class B : A
{
    public override sealed void Method()
    {
    }
}

class C : B
{
    // BŁĄD: nie można przesłaniać składowych z modyfikatorem sealed.
    // public override void Method()
    // {
    // }
}
```

W tym przykładzie zastosowanie modyfikatora sealed w deklaracji metody Method() w klasie B uniemożliwia przesłonięcie tej metody w klasie C.

Klasy rzadko są oznaczane modyfikatorem sealed. Tę technikę należy stosować tylko w sytuacji, gdy istnieją dobre powody do wprowadzania takiego ograniczenia. Modyfikatora sealed dla klas warto unikać także z powodu coraz większej popularności testów jednostkowych. Te testy często tworzą atrapy (obiekty zastępcze), używane zamiast rzeczywistej implementacji klasy. Klasy warto zamykać wtedy, gdy koszt zamykania poszczególnych zmiennych wirtualnych przewyższa korzyści, jakie daje pozostawienie klasy otwartej. Jednak zwykle korzystniejsze jest precyzyjne zamykanie poszczególnych składowych (na przykład wtedy, gdy do poprawnego działania kodu niezbędne jest używanie implementacji z klasy bazowej).

Składowa base

Nawet gdy programista zdecyduje się przesłonić składową, często chce wywołać wersję tej składowej z klasy bazowej (zobacz listing 7.13).

Listing 7.13. Dostęp do składowej z klasy bazowej

```
using static System.Environment;

public class Address
{
    public string StreetAddress;
    public string City;
    public string State;
    public string Zip;

    public override string ToString()
    {
        return $"{StreetAddress + NewLine }"
            + $"{ City },{ State } { Zip }";
    }
}

public class InternationalAddress : Address
{
    public string Country;

    public override string ToString()
    {
        return base.ToString() +
            NewLine + Country;
    }
}
```

Na listingu 7.13 klasa InternationalAddress dziedziczy po klasie Address i zawiera implementację metody `ToString()`. Aby wywołać wersję tej metody z klasy bazowej, należy zastosować słowo kluczowe `base`. Składnia wygląda identycznie jak dla słowa kluczowego `this`. Słowo `base` można też stosować w konstruktorze, co opisano w dalszej części rozdziału.

Warto zauważyc, że metoda `Address.ToString()` wymaga dodania modyfikatora `override`, ponieważ `ToString()` to składowa klasy `object`. Wszystkie składowe opatrzone modyfikatorem `override` automatycznie są traktowane jako wirtualne, dlatego w dalszych klasach pochodnych można utworzyć jeszcze bardziej wyspecjalizowane wersje takich składowych.

 **Uwaga**

Wszystkie metody opatrzone modyfikatorem `override` automatycznie stają się wirtualne. Metodę z klasy bazowej można przesłonić tylko wtedy, jeśli jest wirtualna. Dlatego przesłaniająca metoda także jest wirtualna.

Wywoływanie konstruktorów klas bazowych

Gdy tworzysz instancję klasy pochodnej, środowisko uruchomieniowe najpierw wywołuje konstruktor klasy bazowej, aby jej inicjowanie nie zostało pominięte. Jeśli jednak w klasie bazowej nie występuje dostępny (nieprywatny) konstruktor domyślny, nie jest oczywiste, jak należy zainicjować tę klasę. Dlatego kompilator języka C# zgłasza wtedy błąd.

Aby uniknąć błędu spowodowanego brakiem dostępnego konstruktora domyślnego, programiści powinni jawnie określać (w nagłówku konstruktora klasy pochodnej), który konstruktor klasy bazowej ma zostać użyty (listing 7.14).

Listing 7.14. Określanie, który konstruktor klasy bazowej ma zostać wywołany

```
public class PdaItem
{
    public PdaItem(string name)
    {
        Name = name;
    }
    public virtual string Name { get; set; }
    // ...
}

public class Contact : PdaItem
{
    // Wyłączanie ostrzeżeń, ponieważ FirstName i LastName są ustawiane we właściwości Name
    #pragma warning disable CS8618 // Pole niedopuszczające wartości null nie jest zainicjowane.
    public Contact(string name) :
        base(name)
    {
        Name = name;
    }
    #pragma warning restore CS8618

    public override string Name
    {
        get
        {
            return $"{ FirstName } { LastName }";
        }
        set
        {
            string[] names = value.Split(' ');
            // Kod obsługi błędów nie jest pokazany.
            FirstName = names[0];
            LastName = names[1];
        }
    }

    [NotNull][DisallowNull]
    public string FirstName { get; set; }
    [NotNull][DisallowNull]
    public string LastName { get; set; }
    // ...
}

public class Appointment : PdaItem
{
    public Appointment(string name,
        string location, DateTime startTime, DateTime endTime) :
        base(name)
    {
```

```

        Location = location;
        StartDateTime = startDateTime;
        EndDateTime = endDateTime;
    }

    public DateTime StartDateTime { get; set; }
    public DateTime EndDateTime { get; set; }
    public string Location { get; set; }
}

```

Wskazanie konstruktora klasy bazowej w kodzie pozwala środowisku uruchomieniowemu określić, który z konstruktorów tej klasy ma wywołać przed uruchomieniem konstruktora klasy pochodnej.

Klasy abstrakcyjne

W wielu pokazanych wcześniej przykładach ilustrujących dziedziczenie zdefiniowana była klasa `PdaItem`, obejmująca metody i właściwości wspólne dla pochodnych od niej klas `Contact`, `Appointment` itd. Jednak programy nie mają tworzyć instancji samej klasy `PdaItem`. Instancja tej klasy jest nieprzydatna. Sama klasa `PdaItem` jest potrzebna tylko jako klasa bazowa, ponieważ pozwala udostępniać domyślne implementacje metod w grupie typów pochodnych. Te cechy oznaczają, że `PdaItem` powinna być klasą **abstrakcyjną**, a nie klasą **konkretną**. Klasy abstrakcyjne służą tylko do tworzenia klas pochodnych. Nie da się utworzyć instancji klasy abstrakcyjnej (chyba że w kontekście tworzenia instancji klasy pochodnej). Klasy, które nie są abstrakcyjne i które umożliwiają bezpośrednie tworzenie instancji, to klasy konkretne.

Początek
8.0

Klasy abstrakcyjne są jedną z podstawowych technik obiektowych, dlatego są tu szczegółowo opisane. Od wersji C# 8.0 i .NET Core 3.0 interfejsy udostępniają nadzbiór mechanizmów dostępnych wcześniej w klasach abstrakcyjnych (wyjątkiem są niedostępne w interfejsach pola instancji). Szczegółowe omówienie nowych możliwości interfejsów znajdziesz w rozdziale 8., jednak najpierw należy zrozumieć zagadnienia związane ze składowymi abstrakcyjnymi, dlatego w tym miejscu opisane są klasy abstrakcyjne.

Koniec
8.0

■ ZAGADNIENIE DLA POCZĄTKUJĄCYCH

Klasy abstrakcyjne

Klasy abstrakcyjne są reprezentacją abstrakcyjnych rzeczy. **Składowe abstrakcyjne** definiują, co klasa pochodna od abstrakcyjnej powinna obejmować, ale nie zawierają implementacji. Często wiele funkcji klasy abstrakcyjnej nie jest implementowanych. Aby utworzyć klasę pochodną od abstrakcyjnej, trzeba w tej pierwszej umieścić implementację metod abstrakcyjnych z abstrakcyjnej klasy bazowej.

Aby zdefiniować klasę abstrakcyjną w języku C#, należy dodać modyfikator `abstract` do definicji klasy, co pokazano na listingu 7.15.

Listing 7.15. Definiowanie klasy abstrakcyjnej

```
// Definiowanie klasy abstrakcyjnej.
public abstract class PdaItem
{
    public PdaItem(string name)
    {
        Name = name;
    }

    public virtual string Name { get; set; }
}

public class Program
{
    public static void Main()
    {
        PdaItem item;
        // BŁĄD: nie można utworzyć instancji klasy abstrakcyjnej.
        // item = new PdaItem("Inigo Montoya");
    }
}
```

Choć nie można tworzyć instancji klas abstrakcyjnych, to ograniczenie nie jest najistotniejszą cechą takich klas. Najważniejsza funkcja klas abstrakcyjnych jest związana ze **składowymi abstrakcyjnymi**. Składowa abstrakcyjna to metoda lub właściwość bez implementacji. W ten sposób można wymusić dodanie odpowiedniej implementacji we wszystkich klasach pochodnych.

Przyjrzyj się przykładowemu kodowi z listingu 7.16.

Listing 7.16. Definiowanie składowych abstrakcyjnych

```
// Definiowanie klasy abstrakcyjnej.
public abstract class PdaItem
{
    public PdaItem(string name)
    {
        Name = name;
    }

    public virtual string Name { get; set; }
    public abstract string GetSummary();
}

using static System.Environment;

public class Contact : PdaItem
{
    public override string Name
    {
        get
        {
            return $"{ FirstName } { LastName }";
        }
    }
```

```

        }

    set
    {
        string[] names = value.Split(' ');
        // Obsługę błędów pominięto.
        FirstName = names[0];
        LastName = names[1];
    }
}

public string FirstName
{
    get
    {
        return _FirstName;
    }
    set
    {
        _FirstName = value ??
            throw new ArgumentNullException(nameof(value));
    }
}
private string? _FirstName;

public string LastName
{
    get
    {
        return _LastName!;
    }
    set
    {
        _LastName = value ??
            throw new ArgumentNullException(nameof(value));
    }
}
private string? _LastName;
public string? Address { get; set; }

public override string GetSummary()
{
    return @"FirstName: { FirstName + NewLine }"
        + $"LastName: { LastName + NewLine }"
        + $"Address: { Address + NewLine }";
}

// ...

public class Appointment : PdaItem
{
    public Appointment(string name) :
        base(name)
    {
        Location = location;
    }
}

```

```

    StartDateTime = startDateTime;
    EndDateTime = endDateTime;
}

public DateTime StartDateTime { get; set; }
public DateTime EndDateTime { get; set; }
public string Location { get; set; }

// ...

public override string GetSummary()
{
    return $"Subject: { Name + NewLine }"
        + $"Start: { StartDateTime + NewLine }"
        + $"End: { EndDateTime + NewLine }"
        + $"Location: { Location }";
}
}

```

Na listingu 7.16 składowa `GetSummary()` jest zdefiniowana jako abstrakcyjna, dlatego nie zawiera implementacji. Dalej w kodzie ta składowa jest przesłonięta w klasie `Contact`, gdzie dodano odpowiednią implementację. Ponieważ składowe abstrakcyjne są przeznaczone do przesyłania, automatycznie są uznawane za wirtualne i nie można ich jawnie zadeklarować w ten sposób. Ponadto składowe abstrakcyjne nie mogą być prywatne, ponieważ byłyby niewidoczne w klasach pochodnych.

Opracowanie dobrze zaprojektowanej hierarchii klas jest zaskakująco trudne. Dlatego gdy tworzysz typy abstrakcyjne, koniecznie zaimplementuj przynajmniej jeden typ konkretny (a najlepiej większą ich liczbę) pochodny od danego typu abstrakcyjnego. W ten sposób sprawdzisz poprawność projektu.

Jeśli w klasie `Contact` nie udostępnisz implementacji metody `GetSummary()`, kompilator zgłosi błąd.

Uwaga

Składowe abstrakcyjne wymagają przesłonięcia, dlatego są automatycznie wirtualne i nie można ich jawnie zadeklarować jako wirtualnych.

Porównanie języków — funkcje czysto wirtualne w języku C++

Język C++ umożliwia definiowanie funkcji abstrakcyjnych za pomocą nieintuicyjnego zapisu `=0`. W języku C++ takie funkcje nazywa się funkcjami czysto wirtualnymi. W C++, inaczej niż w C#, sama klasa nie musi być zadeklarowana w specjalny sposób. W C# do klas używany jest modyfikator `abstract`, natomiast w C++ klasy zawierające funkcje czysto wirtualne są deklarowane w standardowy sposób.

Uwaga

Deklarując składową abstrakcyjną, jej autor informuje, że aby utworzyć relację „jest odmianą” między klasą konkretną i abstrakcyjną klasą bazową (tu jest nią PdaItem), trzeba zaimplementować składowe abstrakcyjne. Są to składowe, dla których w klasie abstrakcyjnej nie da się utworzyć odpowiedniej implementacji domyślnej.

ZAGADNIENIE DLA POCZĄTKUJĄCYCH

Polimorfizm

Gdy implementacja składowych o tej samej sygnaturze jest różna w dwóch klasach (lub większej ich liczbie), występuje ważne zjawisko z obszaru programowania obiektowego — **polimorfizm**. *Poly* oznacza „wiele”, a *morf* to „postać”. W tym kontekście polimorfizm dotyczy tego, że istnieje wiele implementacji o tej samej sygnaturze. Ponieważ w jednej klasie ta sama sygnatura nie może występować więcej niż raz, poszczególne implementacje składowej o tej samej sygnaturze muszą się znajdować w różnych klasach.

Polimorfizm związany jest z założeniem, że to dany obiekt najlepiej potrafi wykonać określoną operację. Ponadto poprzez wymuszanie jednego sposobu wykonywania danej operacji polimorfizm zachęca do wielokrotnego używania kodu dzięki wykorzystaniu wspólnych cech danej operacji występujących w różnych obiektach. Jeśli istnieje wiele rodzajów dokumentów, klasy reprezentujące poszczególne z tych rodzajów najlepiej potrafią wykonać metodę Print() dla określonych dokumentów. Dlatego zamiast definiować jedną taką metodę, obejmującą instrukcję switch z kodem drukującym dokumenty poszczególnych typów, dzięki polimorfizmowi można wywołać metodę Print() odpowiadającą specyficzemu typowi dokumentu. Na przykład wywołanie metody Print() z klasy przetwarzającej dokumenty tekstowe da inny efekt niż wywołanie tej samej metody z klasy obsługującej grafikę. Jednak niezależnie od typu dokumentu wystarczy wywołać metodę Print(), by go wydrukować.

Przeniesienie niestandardowej implementacji operacji drukowania poza instrukcję switch ma wiele zalet w obszarze konserwacji. Po pierwsze, implementacja znajduje się wtedy w kontekście klas reprezentujących poszczególne typy dokumentów, a nie w innym, odległym miejscu. Pozwala to zachować hermetyzację. Po drugie, po dodaniu nowego typu dokumentów nie trzeba wtedy modyfikować instrukcji switch. Aby dodać obsługę drukowania w klasie reprezentującej nowy typ dokumentu, wystarczy zaimplementować metodę o sygnaturze Print().

Składowe abstrakcyjne mają umożliwiać polimorfizm. W klasie bazowej określona jest sygnatura metody, a ich implementacja jest dodawana w klasach pochodnych (zobacz listing 7.17).

Listing 7.17. Stosowanie polimorfizmu do wyświetlania obiektów z rodziny PdaItem

```
public class Program
{
    public static void Main()
    {
        PdaItem[] pda = new PdaItem[3];

        Contact contact = new Contact("Sherlock Holmes")
```

```

{
    Address = "221B Baker Street, Londyn, Anglia"
};
pda[0] = contact;

new Appointment(
    "Zawody piłkarskie", "Estádio da Machava",
    new DateTime(2008, 7, 19), new DateTime(2008, 7, 18));
pda[1] = appointment;

contact = new Contact("Hercules Poirot");
contact.Address =
    "Apt 56B, Whitehaven Mansions, Sandhurst Sq, Londyn";
pda[2] = contact;

List(pda);
}

public static void List(PdaItem[] items)
{
    // Zaimplementowana z wykorzystaniem polimorfizmu. Typ pochodny
    // zna szczegóły potrzebne do zaimplementowania metody GetSummary().
    foreach (PdaItem item in items)
    {
        Console.WriteLine("_____");
        Console.WriteLine(item.GetSummary());
    }
}

```

Efekt działania kodu z listingu 7.17 pokazano w danych wyjściowych 7.5.

DANE WYJŚCIOWE 7.5

```

FirstName: Sherlock
LastName: Holmes
Address: 221B Baker Street, Londyn, Anglia

Subject: Zawody piłkarskie
Start: 7/18/2008 12:00:00 AM
End: 7/19/2008 12:00:00 AM
Location: Estádio da Machava

FirstName: Hercules
LastName: Poirot
Address: Apt 56B, Whitehaven Mansions, Sandhurst Sq, Londyn

```

W ten sposób można wywołać metodę klasy bazowej, ale z implementacją specyficzną dla klasy pochodnej. W danych wyjściowych 7.5 widać, że metoda `List()` z listingu 7.17 potrafi z powodzeniem wyświetlać obiekty klas `Contact` i `Address` w sposób dostosowany do każdego z tych typów. Wywołanie abstrakcyjnej metody `GetSummary()` skutkuje wywołaniem przeciążającej metody odpowiedniej dla używanej instancji.

Wszystkie klasy są pochodne od System.Object

W każdej klasie (niezależnie od tego, czy została ona napisana przez programistę, czy jest wbudowana w system) zdefiniowane są metody wymienione w tabeli 7.2.

Tabela 7.2. Składowe klasy System.Object

Nazwa metody	Opis
public virtual bool Equals(object o)	Zwraca true, jeśli obiekt podany jako parametr ma identyczną wartość (niekoniecznie referencję) jak dana instancja.
public virtual int GetHashCode()	Zwraca liczbę całkowitą określającą wartość skrótu o równomiernym rozkładzie. Jest to przydatne na przykład w kolekcjach typu HashTable.
public Type GetType()	Zwraca obiekt typu System.Type reprezentujący typ danej instancji.
public static bool ReferenceEquals(↳(object a, object b)	Zwraca true, jeśli oba podane parametry są referencjami do tego samego obiektu.
public virtual string ToString()	Zwraca łańcuch znaków reprezentujący daną instancję.
public virtual void Finalize()	Jest to alias destruktora. Nakazuje obiektovi przygotowanie się do usunięcia. W C# nie można bezpośrednio wywoływać tej metody.
protected object MemberwiseClone()	Klonuje dany obiekt, tworząc jego płytka kopię. Dla typów referencyjnych kopowane są referencje, ale już nie dane.

Wszystkie metody wymienione w tabeli 7.2 są w wyniku dziedziczenia dostępne w każdym obiekcie. Po klasie object dziedziczą (pośrednio lub przez łańcuch dziedziczenia) wszystkie klasy. Wymienione metody są dostępne nawet dla literałów, co umożliwia pisanie dziwnego kodu:

```
Console.WriteLine( 42.ToString() );
```

Po klasie object dziedziczą nawet klasy, w których deklaracji w ogóle nie jest to zaznaczone. Dlatego obie deklaracje klasy PdaItem przedstawione na listingu 7.18 prowadzą do wygenerowania identycznego kodu CIL.

Listing 7.18. Niewidoczne dziedziczenie po klasie System.Object bez kodu oznaczającego jawnego dziedziczenia

```
public class PdaItem
{
    // ...
}
public class PdaItem : object
{
    // ...
}
```

Gdy domyślna implementacja z klasy object nie jest wystarczająca, programiści mogą przesłonić jedną lub kilka spośród trzech metod wirtualnych. W rozdziale 10. poznasz szczegóły wykonywania tej operacji.

Dopasowanie do wzorca za pomocą operatora is

Operator `is` jest dostępny od wersji C# 1.0, przy czym w C# 7.0 i C# 8.0 wprowadzono istotne usprawnienia, jeśli chodzi o dopasowywanie wzorca. Warto zauważyć, że wiele nowych technik ma stosunkowo niewielkie znaczenie, chyba że używa się ich w instrukcjach lub wyrażeniach `switch`, co opisane jest w dalszej części rozdziału.

Sprawdzanie typu za pomocą operatora is

Ponieważ C# dopuszcza rzutowanie na typy znajdujące się niżej w łańcuchu dziedziczenia, czasem warto sprawdzić rzeczywisty typ obiektu przed próbą przeprowadzenia konwersji. Ponadto sprawdzanie typu może być konieczne do przeprowadzania specyficznych dla niego operacji, jeśli nie są dostępne polimorficzne metody. W C# do ustalania rzeczywistego typu służy operator `is` (zobacz listing 7.19).

Listing 7.19. Używanie operatora `is` do określania typu

```
public static void Save(object data)
{
    if (data is string)
    {
        string text = (string) data;
        if (text.Length > 0)
        {
            data = Encrypt(text);
            // ...
        }
    }
    else if (data is null)
    {
        // ...
    }
    // ...
}
```

Kod z listingu 7.19 szyfruje dane, gdy obiekt jest typu `string`. To rozwiązanie działa inaczej niż szyfrowanie wartości dowolnego typu danych, którą można zrzutować na typ `string` (niektóre typy obsługują rzutowanie na typ `string`, choć na nim nie bazują).

Choć bardziej przejrzyste byłoby sprawdzanie wartości `null` na początku metody, w tym przykładzie operacja ta jest wykonywana w dalszym kodzie. Pozwala to pokazać, że nawet gdy zmienna ma wartość `null`, operator `is` zwraca wartość `false`, a kod sprawdza wartość `null`.

Warto zauważyć, że przy stosowaniu jawnego rzutowania programista musi na tyle dobrze rozumieć kod, aby uniknąć wyjątków z powodu błędного rzutowania. Jeśli występuje ryzyko nieprawidłowego rzutowania, lepiej jest używać operatora `is`, aby uniknąć wyjątków. Zaletą stosowania operatora `is` jest to, że pozwala utworzyć ścieżkę wykonania kodu, gdzie jawnie rzutowanie może się nie powieść, a mimo to nie trzeba ponosić kosztów obsługi wyjątku. Ponadto w wersjach C# 7.0 i nowszych operator `is` obok sprawdzania typu danych może też przypisywać wartości.

Dopasowanie do wzorca w postaci typu, stałej i var z użyciem operatora is

Od wersji C# 7.0 operator `is` jest wzbogacony o obsługę dopasowania do wzorca. Problem z operatorem `is` polegał na tym, że po sprawdzeniu, czy dane są typu `string`, nadal trzeba było zrzucać je na ten typ (przy założeniu, że programista chce ich używać jako wartości typu `string`). Lepszym podejściem byłoby jednoczesne sprawdzanie oraz — gdy wynik testu to `true` — przypisywanie wartości do nowej zmiennej. Dzięki wprowadzeniu w C# 7.0 **dopasowania do wzorca** takie rozwiązywanie stało się możliwe dla typów, stałych i `var`. W C# 8.0 ten mechanizm został rozbudowany o dopasowywanie krotek i właściwości oraz dopasowywanie pozycyjne i rekurencyjne. W większości sytuacji nowe techniki pozwalały zastąpić prostszy operator `is`.

W tabeli 7.3 pokazane są przykładowe zastosowania technik dopasowania do wzorca wprowadzonych w C# 7.0.

Tabela 7.3. Dopasowanie do wzorca w postaci typu, stałej i `var` za pomocą operatora `is`

Opis	Przykładowy kod
Dopasowanie do wzorca w postaci typu <p>Wynik wywołania <code>GetObjectById(id)</code> jest w jednym wyrażeniu porównywany z typem <code>Employee</code> i przypisywany do zmiennej <code>employee</code>. Jeśli wynikiem wywołania <code>GetObjectById(id)</code> jest <code>null</code> lub wartość typu innego niż <code>Employee</code>, generowana jest wartość <code>false</code> i kod wykonuje klauzulę <code>else</code>.</p> <p>Zmienna <code>employee</code> jest dostępna w instrukcji <code>if</code> i po niej. Wymaga jednak przypisania wartości przed dostępem do niej w klauzuli <code>else</code> lub po niej.</p>	<pre>// ... string id = "92e80a67-d453-4998-8d85-f430fa02d6c7"; if (GetObjectById(id) is Employee employee) { Display(employee); } else { ReportError(\$"Błędny identyfikator {id}."); }</pre>
Dopasowanie wzorca w postaci stałej <p>W rozdziale 4. dopasowanie do wzorca w postaci stałej polegało na użyciu operatora <code>is</code> do wykrywania wartości <code>null</code> (<code>data is null</code>). Tak samo można sprawdzać dowolne stałe. Możesz na przykład porównać zmienną <code>data</code> z pustym łańcuchem znaków (<code>data is ""</code>). W takim porównaniu trzeba użyć stałej. W wyrażeniu <code>data is string</code> <code>Empty</code> używana jest właściwość zamiast stałej, dlatego jest ono nieprawidłowe.</p>	<pre>public static void Save(object data) { // ... else if (data is "") { return; // ... } }</pre>
Dopasowanie do <code>var</code> <p>Inaczej niż przy dopasowaniu do wzorca w postaci typu tu można użyć <code>var</code> jako typu, aby przechwycić dowolną wartość, w tym <code>null</code>. Oczywiście zalety tej techniki w porównaniu ze zwykłym przypisaniem (<code>var result = GetObjectById(id)</code>) są wątpliwe, ponieważ instrukcja zawsze kończy się powodzeniem.</p> <p>Podejście to jest jednak bardziej przydatne w instrukcjach <code>switch</code>, gdzie pozwala utworzyć blok przechwytyujący wszystkie wartości.</p>	<pre>// ... else (GetObjectById(id) is var result) { // ... }</pre>

W C# 8.0 dopasowanie do wzorca jest bardziej skomplikowane, ponieważ możliwe jest tam dopasowywanie do krotek i właściwości oraz dopasowywanie pozycyjne i rekurencyjne.

Dopasowanie krotek

Dopasowanie krotek pozwala sprawdzać stałe w krotkach lub przypisywać elementy krotek do zmiennych (zobacz listing 7.20).

Listing 7.20. Dopasowanie krotek za pomocą operatora is

```
public class Program
{
    const int Action = 0;
    const int FileName = 1;
    public const string DataFile = "data.dat";

    static public void Main(params string[] args)
    {
        // ...

        if ((args.Length, args[Action]) is (1, "show"))
        {
            Console.WriteLine(File.ReadAllText(DataFile));
        }
        else if ((args.Length, args[Action].ToLower(), args[FileName]) is
            (2, "encrypt", string fileName))
        {
            string data = File.ReadAllText(DataFile);
            File.WriteAllText(fileName, Encrypt(data).ToString());
        }
        // ...
    }
}
```

W tym przykładzie wzorzec jest dopasowywany do krotki, która jest zapełniana długością i elementami tablicy args. W pierwszym warunku `if` oczekiwany jest jeden argument i operacja "show". W drugim warunku `if` kod sprawdza, czy pierwszy element tablicy ma wartość "encrypt". Jeśli tak jest, trzeci element krotki jest przypisywany do zmiennej `fileName`. Każdy dopasowywany element może być stałą lub zmienną. Ponieważ krotka jest tworzona przed wykonaniem operatora `is`, nie można najpierw użyć scenariusza z wartością "encrypt", ponieważ `args[FileName]` nie będzie poprawnym indeksem, gdy użytkownik zaząda operacji "show".

Dopasowanie pozycyjne

Na bazie wprowadzonego w wersji C# 7.0 dekonstruktora (zobacz rozdział 6.) w C# 8.0 udostępniono dopasowanie pozycyjne. Składnia tej techniki jest analogiczna jak przy dopasowywaniu krotek (listing 7.21).

Listing 7.21. Dopasowanie pozycyjne z wykorzystaniem operatora is

```

public class Person
{
    // ...

    public void Deconstruct(out string firstName, out string lastName) =>
        (firstName, lastName) = (FirstName, LastName);
}

public class Program
{
    static public void Main(string[] args)
    {
        Person person = new Person("Inigo", "Montoya");

        // Dopasowanie pozycyjne
        if (person is (string firstName, string lastName))
        {
            Console.WriteLine($"{firstName} {lastName}");
        }
    }
}

```

W tym przykładzie stałe nie są używane. Zamiast tego wszystkie elementy z dekonstruktora są przypisywane do zmiennych w nowo tworzonej krotce. Dozwolone jest też sprawdzanie, czy w krotce występują określone stałe.

Dopasowanie właściwości

Wzorce właściwości pozwalają dopasowywać wartości na podstawie nazwy właściwości i wartości typu danych z wyrażenia switch (listing 7.22).

Listing 7.22. Dopasowanie właściwości z wykorzystaniem operatora is

```

// ...
Person person = new Person("", "");

// Dopasowanie właściwości
if (person is {FirstName: string firstName, LastName: string lastName })
{
    Console.WriteLine($"{firstName} {lastName}");
}
// ...

```

Na pozór listing 7.22 wygląda prawie identycznie jak listing 7.21 i przypomina dopasowywanie pozycyjne. Występują tu jednak dwie ważne różnice. Po pierwsze przy dopasowywaniu właściwości do określania, z czym należy porównywać dane, używane są nawiasy klamrowe zamiast zwykłych. Po drugie pozycje argumentów (ważne przy dopasowywaniu pozycyjnym i krotek) są nieistotne przy dopasowywaniu właściwości, ponieważ zamiast nich uwzględniane są nazwy właściwości. Warto też zauważyć, że dopasowywanie właściwości jest używane przy sprawdzaniu wartości null za pomocą składni is { }.

Dopasowywanie rekurencyjne

Wcześniej wspomniano, że wartość dopasowywania właściwości staje się w pełni widoczna dopiero w instrukcjach i wyrażeniach switch. Wyjątkiem jest sytuacja, gdy dopasowanie do wzorca jest przeprowadzane rekurencyjnie. Trzeba przyznać, że kod z listingu 7.23 jest bezsensowny. Ilustruje jednak złożoność, jaka może się pojawić przy rekurencyjnym stosowaniu wzorców.

Listing 7.23. Dopasowywanie rekurencyjne z wykorzystaniem operatora is

```
// ...
Person inigo = new Person("Inigo", "Montoya");
var buttercup =
    (FirstName: "Princess", LastName: "Buttercup");

(Person inigo, (string FirstName, string LastName) buttercup) couple =
    (inigo, buttercup);

if (couple is
    ( // Krotka
        ( // Pozycyjne
            { // Właściwości
                Length: int inigoLength1 },
            _ // Ignorowanie
        ),
    // Właściwości
    FirstName: string buttercupFirstName ))) {
    Console.WriteLine($"{inigoFirstNameLength}, {buttercupFirstName}");
}
else
{
    // ...
}
// ...
```

W tym przykładzie couple jest następującego typu:

```
(Person, (string FirstName, string LastName))
```

Dlatego pierwsze dopasowanie dotyczy krotki zewnętrznej (inigo, buttercup). Następnie kod wykonuje dopasowanie pozycyjne względem inigo, wykorzystując dekonstruktor klasy Person. To powoduje wybranie krotki (FirstName, LastName), z której za pomocą dopasowywania właściwości pobierana jest długość (Length) wartości inigo.FirstName. Człon LastName, dla którego używane jest dopasowywanie pozycyjne, jest ignorowany, do czego służy podkreślenie. W ostatnim kroku dopasowanie właściwości powoduje pobranie wartości buttercup.LastName.

Wprowadzony w wersji C# 8.0 mechanizm dopasowywania właściwości jest bardzo przydatną techniką pobierania danych, ale ma pewne ograniczenia. W odróżnieniu od opisanej w rozdziale 4. instrukcji switch i klauzuli when dopasowywanie nie pozwala stosować predykatów (na przykład sprawdzać, czy długości właściwości FirstName i LastName

są różne od zera). Warto też zwrócić uwagę na czytelność. Listing 7.23 nawet po opatrzeniu komentarzami jest trudny do zrozumienia. Bez komentarzy kod byłby jeszcze mniej czytelny:

```
if (couple is ( { Length: int inigoFirstNameLength }, _ ),
{ FirstName: string buttercupFirstName })) { ...}
```

Miejscem, gdzie dopasowanie do wzorca jest najbardziej przydatne, są instrukcje i wyrażenia switch.

Początek
7.0

Dopasowanie do wzorca w wyrażeniu switch

Listing 7.23 zawiera prostą instrukcję if-else. Można jednak wyobrazić sobie podobny przykład, w którym sprawdzanie dotyczy także innych typów obok typu string. Choć instrukcja if też się wtedy sprawdzi, instrukcja switch z wyrażeniem pasującym do dowolnego typu będzie dużo bardziej czytelna. W przykładzie z listingu 7.24 kod formatuje datę na różne sposoby.

Listing 7.24. Dopasowanie do wzorca w wyrażeniu switch

```
public static string? CompositeFormatDate(
    object input, string compositFormatString) =>
input switch
{
    DateTime { Year: int year, Month: int month, Day: int day }
        => (year, month, day),
    DateTimeOffset
    { Year: int year, Month: int month, Day: int day }
        => (year, month, day),
    string dateText => DateTime.TryParse(
        dateText, out DateTime dateTime) ?
        (dateTime.Year, dateTime.Month, dateTime.Day) :
        default((int Year, int Month, int Day)?),
        => null
} is { } date ? string.Format(
    compositFormatString, date.Year, date.Month, date.Day) : null;
```

W pierwszym bloku tego wyrażenia switch używane jest dopasowanie do wzorca w postaci typu (C# 7.0), aby sprawdzić, czy dane wejściowe są typu DateTime. Jeśli wynik to true, stosowane jest dopasowanie właściwości, aby zadeklarować i przypisać wartości zmiennych year, month i day. Następnie te zmienne są używane w wyrażeniu, które zwraca krotkę (year, month, day). Blok dla typu DateTimeOffset działa analogicznie.

W bloku dla typu string ani w bloku domyślnym (_) kod nie używa dopasowywania rekurencyjnego. Zauważ, że dla typu string po nieudanym wywołaniu TryParse() zwracana jest wartość default((int Year, int Month, int Day)?)² równa null. Nie można po prostu zwrócić null, ponieważ nie istnieje automatyczna konwersja między typem (int Year, int Month, int Day) (zwracanym w innych blokach) i null. Trzeba podać krotkę dopuszczającą

² Więcej informacji znajdziesz w rozdziale 12.

wartość null, aby precyzyjnie określić typ tego wyrażenia switch. Zamiast używania operatora default można też zastosować rzutowanie: ((int Year, int Month, int Day)?) null. Dopuszczalność wartości null jest tu istotna, aby wyrażenie input switch {} is {} date nie zwróciło wartości true po nieudanym parsowaniu danych.

Zauważ, że na listingu 7.24 nie ma żadnych klauzul when z predykatami, które dodatkowo ograniczałyby dopasowywane dane. Dostępna jest jednak pełna obsługa klauzul w postaci:

```
DateTime  
{ Year: int year, Month: int month, Day: int day } tempDate  
when tempDate < DateTime.Now => (year, month, day)
```

Taka klauzula dopuszcza tylko przyszłe daty.

Unikaj dopasowania do wzorca, gdy możliwy jest polimorfizm

Choć dopasowanie do wzorca to ważny mechanizm, przed zastosowaniem operatora is powinieneś rozważyć problemy związane z polimorfizmem. Polimorfizm umożliwia udostępnianie operacji w innych typach danych bez konieczności wprowadzania jakichkolwiek zmian w implementacji tych operacji. Na przykład lepiej jest umieścić zmienną Name w klasie bazowej PdaItem, a następnie używać wartości typów pochodnych od klasy PdaItem, niż stosować dopasowywanie do wzorca z instrukcjami case dla każdego typu. Pierwsze z tych rozwiązań pozwala dodać nowe typy pochodne od PdaItem (które mogą nawet znajdować się w innym podzespole) bez konieczności ponownej komplikacji kodu. Z kolei druga technika wymaga dodatkowego modyfikowania kodu z dopasowaniem do wzorca, aby uwzględnić nowy typ. Niezależnie od tego polimorfizm nie zawsze można zastosować.

Polimorfizm nie sprawdzi się między innymi w sytuacji, gdy nie istnieje odpowiednia hierarchia obiektów — na przykład gdy używasz klas należących do niepowiązanych systemów. Ponadto kod wymagający polimorfizmu często jest poza kontrolą programisty i nie można go modyfikować. Dotyczy to na przykład dat z listingu 7.24. Polimorfizm nie jest też odpowiedni, jeśli potrzebne funkcje nie należą do podstawowych cech danych klas. Na przykład opłaty płacone na autostradzie są inne dla różnych rodzajów pojazdów, jednak sama opłata nie jest jedną z podstawowych cech pojazdu.

Koniec
7.0
Koniec
8.0

ZAGADNIENIE DLA ZAAWANSOWANYCH

Konwersja z wykorzystaniem operatora as

Obok operatora is C# udostępnia też operator as. Pierwotnie operator as wykonywał dodatkową operację w porównaniu z operatorem is: nie tylko sprawdzał, czy operant jest określonego typu, ale też próbował przeprowadzić konwersję na określony typ i ustawić wartość null, jeśli źródłowej wartości nie można było (z uwzględnieniem łańcucha dziedziczenia) przekształcić na docelowy typ. To rozwiązanie jest przydatne, ponieważ pozwala uniknąć wyjątku, który może być skutkiem rzutowania. Na listingu 7.25 pokazano, jak używać operatora as.

Listing 7.25. Konwersja danych z wykorzystaniem operatora as

```

public class PdaItem
{
    protected Guid ObjectKey { get; }
    // ...
}

public class Contact: PdaItem
{
    // ...
    static public Contact Load(PdaItem pdaItem)
    {
        #pragma warning disable IDE0019 // Stosowanie dopasowania do wzorca
        Contact? contact = pdaItem as Contact;
        if (contact != null)
        {
            System.Diagnostics.Trace.WriteLine(
                $"ObjectKey: {contact.ObjectKey}");
            return (Contact)pdaItem;
        }
        else
        {
            throw new ArgumentException(
                $"{nameof(pdaItem)} nie jest typu {nameof(Contact)}");
        }
    }
}

```

Zastosowanie operatora as pozwala uniknąć dodatkowego bloku try/catch, związanego z obsługą nieprawidłowej konwersji. Dzieje się tak, ponieważ operator as umożliwia przeprowadzenie próby rzutowania bez zgłaszania wyjątku w sytuacji, gdy operacja ta kończy się niepowodzeniem.

Zaletą operatora is w porównaniu do operatora as jest to, że ten drugi nie pozwala precyzyjnie określić typu obiektu. Operator as może niejawnie przeprowadzić rzutowanie na typ znajdujący się wyżej lub niżej w łańcuchu dziedziczenia, a także na typy udostępniające operator rzutowania. Operator is, w odróżnieniu od operatora as, pozwala określić używanego typu. Ponadto operator as działa tylko dla typów referencyjnych, natomiast operator is można stosować do wszystkich typów.

Co ważniejsze, operator as zwykle wymaga wykonania dodatkowego kroku — sprawdzenia, czy zmienna ma wartość null. Ponieważ operator is z dopasowaniem do wzorca wykonuje taki test automatycznie, w praktyce eliminuje konieczność stosowania operatora as — jeśli używana jest wersja C# 7.0 lub nowsza.

Podsumowanie

W tym rozdziale opisano, jak przygotować wyspecjalizowaną wersję klasy, tworząc klasę pochodną od niej i dołączając dodatkowe metody i właściwości. Omówiono też modyfikatory dostępu, `private` i `protected`, służące do kontrolowania poziomu hermetyzacji.

W rozdziale wyjaśniono również przesłanianie implementacji z klasy bazowej, a także ukrywanie przesłaniających składowych za pomocą modyfikatora `new`. Do kontrolowania przesłaniania w języku C# służy modyfikator `virtual`. Informuje on programistę klasy pochodnej o tym, które składowe są przeznaczone do przesłaniania. Aby zapobiec tworzeniu klas pochodnych, można zastosować do klasy modyfikator `sealed`. Dodanie modyfikatora `sealed` do składowej uniemożliwia przesłanianie jej w klasach pochodnych.

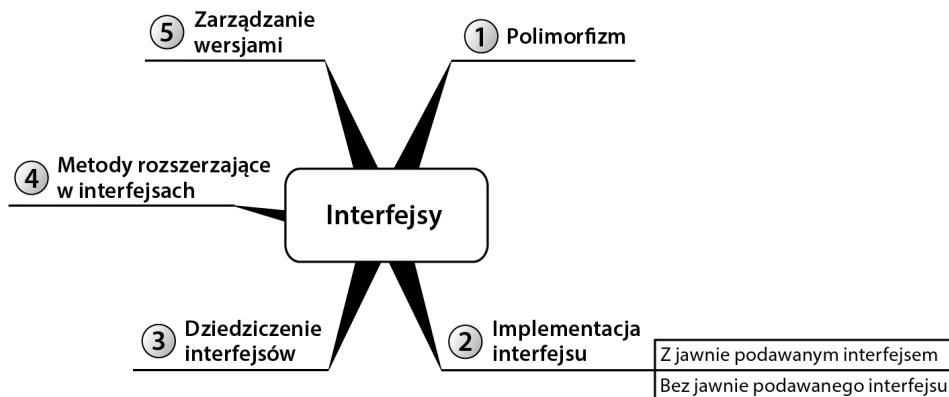
Rozdział obejmuje też krótkie omówienie tego, że wszystkie typy są pochodne od typu `object`. W rozdziale 10. opisano tę kwestię bardziej szczegółowo. Wyjaśniono tam, że klasa `object` obejmuje trzy metody wirtualne, których przesłaniem rzadzą określone reguły i wytyczne. Zanim jednak dotrzesz do wspomnianego rozdziału, powinieneś zapoznać się z innym mechanizmem programistycznym z obszaru programowania obiektowego; są nim interfejsy, będące tematem rozdziału 8.

Na zakończenie niniejszego rozdziału omówiono dopasowanie do wzorca z użyciem operatora `is` oraz wyrażeń i instrukcji `switch`. W C# 7.0 i 8.0 znacznie rozbudowano te techniki, choć nie wszystkie ich możliwości są powszechnie wykorzystywane. Powodem tego jest przede wszystkim fakt, że tam, gdzie to możliwe, zaleca się stosowanie polimorfizmu zamiast dopasowania do wzorca.

8

Interfejsy

POLIMORFIZM W C# MOŻNA OSIĄGNĄĆ NIE TYLKO dzięki dziedziczeniu (opisanemu w rozdziale 7.), ale też za pomocą interfejsów. Interfejsy, w odróżnieniu od klas abstrakcyjnych, aż do wersji C# 8.0 nie mogły obejmować implementacji (przy czym nawet w C# 8.0 wątpliwe jest, czy należy wykorzystywać tę możliwość do czegoś innego niż zarządzanie wersjami i interfejsami). Jednak, podobnie jak klasy abstrakcyjne, definiują zestaw składowych, co do których jednostka wyołującą może przyjąć, że są zaimplementowane.



Implementacja interfejsu w danym typie określa możliwości tego typu. **Powstaje w ten sposób relacja „potrafi”.** Typ z implementacją interfejsu „potrafi” robić to, czego dany interfejs wymaga. Interfejs definiuje kontrakt między typami z implementacją interfejsu a kodem, który tego interfejsu używa. W typach z implementacją interfejsu trzeba zadeklarować metody o tych samych sygnaturach, jakie mają metody zadeklarowane w implementowanym interfejsie. W tym rozdziale opisano, jak implementować interfejsy i jak z nich korzystać. W końcowej części rozdziału dowiesz się, jak tworzyć w interfejsach składowe z implementacją domyślną, a także poznasz różne modele (i komplikacje) związane z tą nową funkcją.

Wprowadzenie do interfejsów

ZAGADNIENIE DLA POCZĄTKUJĄCYCH

Po co stosować interfejsy?

Interfejsy są przydatne, ponieważ (w odróżnieniu od klas abstrakcyjnych) umożliwiają całkowite oddzielenie szczegółów implementacji od udostępnianych usług. W rzeczywistym świecie można powiedzieć, że „interfejsem” jest gniazdko elektryczne w ścianie. To, jak prąd dociera do gniazdka, jest szczegółem implementacji. Prąd może być generowany za pomocą energii chemicznej, jądrowej lub słonecznej; generator może się znajdować w pokoju obok lub bardzo daleko itd. Gniazdko określa „kontrakt”. Ma udostępniać określone napięcie z określona częstotliwością, a urządzenie, które korzysta z tego interfejsu, ma w zamian mieć właściwą wtyczkę. Dla urządzenia nie jest ważne, w jaki sposób szczegóły implementacji prowadzą do dostarczenia prądu do gniazdka. Urządzenie ma tylko zapewniać odpowiednią wtyczkę.

Pomyśl o następującym przykładzie — dostępnych jest wiele formatów kompresji plików (.zip, .7-zip, .cab, .lha, .tar, .tar.gz, .tar.bz2, .bh, .rar, .arj, .arc, .ace, .zoo, .gz, .bzip2, .xxe, .mime, .uue i .yenc to tylko wybrane z nich). Jeśli utworzysz klasę do obsługi każdego z tych formatów, możesz otrzymać metody o różnych sygnaturach dla poszczególnych implementacji. Nie da się wtedy wywoływać metod w jeden standardowy sposób. Potrzebną metodę można zadeklarować jako metodę abstrakcyjną w klasie bazowej. Jednak konieczność tworzenia klas pochodnych od wspólnej klasy bazowej oznacza, że dostępna jest tylko jedna możliwość dziedziczenia. Ponadto jest mało prawdopodobne, że w klasie bazowej uda się umieścić wspólny kod odpowiedni dla różnych formatów kompresji. W ten sposób utracone zostają potencjalne zalety korzystania z implementacji klasy bazowej. Należy zapamiętać, że klasy bazowe umożliwiają współużytkowanie implementacji i sygnatur składowych, natomiast interfejsy służą tylko do współużytkowania sygnatur (bez implementacji).

Zamiast używać wspólnej klasy bazowej, należy w klasach obsługujących różne formaty kompresji zaimplementować wspólny interfejs. Interfejsy definiują kontrakt, który klasa realizuje, by umożliwić interakcje z nią innym klasom oczekującym danego interfejsu. Jeśli we wszystkich klasach do obsługi kompresji zaimplementujesz interfejs `IFileCompression` i dostępne w nim metody `Compress()` i `Uncompress()`, w kodzie algorytmu korzystającego z dowolnej z tych klas można przeprowadzić konwersję na interfejs `IFileCompression` i wywoływać jego składowe. W efekcie powstaje polimorficzne rozwiązywanie, ponieważ każda klasa do obsługi kompresji ma tę samą sygnaturę metody, ale już implementacje tych metod są różne.

Na listingu 8.1 przedstawiono kod przykładowego interfejsu `IFileCompression`. Zgodnie z konwencją (tak powszechną, że stosowaną wszędzie) nazwa interfejsu jest zapisana w NotacjiPascalowej z wielką literą *I* na początku.

Listing 8.1. Definiowanie interfejsu

```
interface IFileCompression
{
    void Compress(string targetFileName, string[] fileList);
    void Uncompress(
        string compressedFileName, string expandDirectoryName);
}
```

W interfejsie `IFileCompression` zdefiniowano metody, które trzeba zaimplementować w danym typie, by móc z niego korzystać w ten sam sposób jak z innych klas związanych z kompresją. Zaletą interfejsów jest to, że umożliwiają one zmianę implementacji w jednostce wywołującej bez konieczności modyfikowania kodu korzystającego z tego interfejsu.

W wersjach starszych niż C# 8.0 jedną z istotnych cech interfejsów było to, że nie obejmowały one implementacji ani danych. Deklaracje metod w interfejsach zawsze miały po nagłówku jeden średnik zamiast nawiasów klamrowych. Właściwości, choć wyglądały tak, jakby były automatycznie implementowane, nie były powiązane z polami. W deklaracji interfejsu nie mogły występować pola (dane).

W C# 8.0 wiele tych reguł zostało rozluźnionych, aby umożliwić wprowadzanie zmian w udostępnionych interfejsach zgodnie z regułami. Jednak do podrozdziału „Zarządzanie wersjami interfejsów w wersjach C# 8.0 i nowszych” nowe możliwości są pomijane, a interfejsy są omawiane na potrzeby zapewniania polimorfizmu. Wartość interfejsów związana jest właśnie z polimorfizmem i łatwiej jest je opisać w tym kontekście, a dopiero potem przejść do nowych funkcji i nietypowych scenariuszy. Dlatego dla uproszczenia na razie obowiązuje założenie, że interfejsy nie mogą mieć implementacji (bez wspomniania o wersji C# 8.0). To ograniczenie zostanie wyeliminowane dopiero w miejscu omawiania mechanizmów z C# 8.0.

Składowe zadeklarowane w interfejsie określają, jakie składowe muszą być dostępne w typie z implementacją. Składowe niepubliczne nie są dostępne poza klasą. Dlatego w języku C# składowe interfejsu nie mogą mieć modyfikatorów dostępu. Wszystkie składowe interfejsów są automatycznie definiowane jako publiczne.

Wskazówka

STOSUJ Notację Pascalową z przedrostkiem I do tworzenia nazw interfejsów.

Początek
8.0

Koniec
8.0

Polimorfizm oparty na interfejsach

Przyjrzyj się teraz następnemu przykładowi, przedstawionemu na listingu 8.2. W interfejsie `IListable` zdefiniowano składowe potrzebne w klasie, aby mogła być ona wyświetlna za pomocą klasy `ConsoleListControl`. Każda klasa z implementacją interfejsu `IListable` może więc używać klasy `ConsoleListControl` do wyświetlania swoich danych. Interfejs `IListable` wymaga właściwości tylko do odczytu o nazwie `CellValues`.

Listing 8.2. Implementowanie i używanie interfejsów

```
interface IListable
{
    // Zwraca wartość każdej komórki wiersza.
    string?[] CellValues { get; }
}

public abstract class PdaItem
{
    public PdaItem(string name)
```

```
{  
    Name = name;  
}  
  
public virtual string Name{get; set;}  
}  
class Contact : PdaItem, IListable  
{  
    public Contact(string firstName, string lastName,  
        string address, string phone) :  
        base(GetName(firstName, lastName))  
    {  
        FirstName = firstName;  
        LastName = lastName;  
        Address = address;  
        Phone = phone;  
    }  
  
    public string LastName { get; }  
    // ...  
    public string? FirstName { get; }  
    public string? Address { get; }  
    public string? Phone { get; }  
    public static string GetName(string firstName, string lastName)  
        => $"{firstName} {lastName}";  
  
    public string?[] CellValues  
    {  
        get  
        {  
            return new string?[]  
            {  
                FirstName,  
                LastName,  
                Phone,  
                Address  
            };  
        }  
    }  
  
    public static string[] Headers  
    {  
        get  
        {  
            return new string[] {  
                "Imię", "Nazwisko",  
                "Telefon",  
                "Adres"  
            };  
        }  
    }  
    // ...  
}  
class Publication : IListable  
{  
    public Publication(string title, string author, int year)  
    {
```

```
Title = title;
Author = author;
Year = year;
}

public string Title { get; }
public string Author { get; }
public int Year { get; }

public string?[] CellValues
{
    get
    {
        return new string?[]
        {
            Title,
            Author,
            Year.ToString()
        };
    }
}

public static string[] Headers
{
    get
    {
        return new string[] {
            "Tytuł",
            "Autor",
            "Rok"};
    }
}

// ...
}

class Program
{
    public static void Main()
    {
        Contact[] contacts = new Contact[]
        {
            new Contact(
                "Jan", "Kowal",
                "ul. Budowlanych 12/13, Opole 45-287",
                "123-123-1234"),
            new Contact(
                "Andrzej", "Litwin",
                "ul. Krótka 1/23, 30-037 Kraków",
                "555-123-4567"),
            new Contact(
                "Maria", "Harbiel",
                "ul. Liliowa 7, 25-129 Kępno",
                "444-123-4567"),
            new Contact(
                "Janusz", "Stocki",
                "ul. Opolska 5/7, 43-290 Wrocław",
                "222-987-6543"),
            new Contact(
```

```

    "Patrycja", "Wielgosz",
    "ul. Kościuszki 321, 28-092 Sochaczew",
    "123-456-7890"),
  new Contact(
    "Janina", "Frątczak",
    "ul. Majowa 9/18, 01-154 Warszawa",
    "333-345-6789")
};

// Klasę są niejawnie rzutowane na
// zaimplementowane w nich interfejsy.
ConsoleListControl.List(Contact.Headers, contacts);

Console.WriteLine();

Publication[] publications = new Publication[3] {
  new Publication("Koniec z nędzą. Zadanie dla naszego pokolenia",
    "Jeffrey Sachs", 2006),
  new Publication("Ortodoksyjna",
    "G.K. Chesterton", 1908),
  new Publication(
    "Autostopem przez galaktykę",
    "Douglas Adams", 1979)
};
ConsoleListControl.List(
  Publication.Headers, publications);
}

class ConsoleListControl
{
  public static void List(string[] headers, IListable[] items)
  {
    int[] columnWidths = DisplayHeaders(headers);

    for (int count = 0; count < items.Length; count++)
    {
      string?[] values = items[count].CellValues;
      DisplayItemRow(columnWidths, values);
    }
  }

  /// <summary>Wyświetla nagłówki kolumn</summary>
  /// <returns>Zwraca tablicę szerokości kolumn</returns>
  private static int[] DisplayHeaders(string[] headers)
  {
    // ...
  }

  private static void DisplayItemRow(
    int[] columnWidths, string?[] values)
  {
    // ...
  }
}

```

Wynik działania kodu z listingu 8.2 pokazano w danych wyjściowych 8.1.

DANE WYJŚCIOWE 8.1.

Imię	Nazwisko	Telefon	Adres
Jan	Kowal	123-123-1234	ul. Budowlanych 12/13, Opole 45-287
Andrzej	Litwin	555-123-4567	ul. Krótka 1/23, 30-037 Kraków
Maria	Harbiel	444-123-4567	ul. Liliowa 7, 25-129 Kępno
Janusz	Stocki	222-987-6543	ul. Opolska 5/7, 43-290 Wrocław
Patrycja	Wielgosz	123-456-7890	ul. Kościuszki 321, 28-092 Sochaczew
Janina	Frątczak	333-345-6789	ul. Majowa 9/18, 01-154 Warszawa
Tytuł	Autor	Rok	
Koniec z nedzą. Zadanie dla naszego pokolenia	Jeffrey Sachs	2006	
Ortodoxja	G.K. Chesterton	1908	
Autostopem przez galaktykę	Douglas Adams	1979	

Klasa `ConsoleListControl` z listingu 8.2 potrafi wyświetlać zawartość pozornie niepowiązanych ze sobą klas (`Contact` i `Publication`). Możliwe jest wyświetlenie zawartości dowolnej klasy, pod warunkiem jednak, że zawiera ona implementację potrzebnego interfejsu. Metoda `ConsoleListControl.List()` wykorzystuje polimorfizm do poprawnego wyświetlania przekazanego do niej zbioru obiektów. W każdej klasie znajduje się inna implementacja właściwości `CellValues`, a konwersja obiektu danej klasy na typ `IListable` pozwala wywołać implementację z określonej klasy.

Implementacja interfejsu

Deklarowanie, że klasa zawiera implementację interfejsu, odbywa się podobnie jak tworzenie klasy pochodnej od wskazanej klasy bazowej — należy podać implementowany interfejs obok klasy bazowej na rozdzielonej przecinkami liście. Najpierw musi się znajdować nazwa klasy bazowej (jeśli jest używana), natomiast kolejność podawanych interfejsów nie ma znaczenia. W klasach można implementować wiele interfejsów, natomiast dziedziczyć bezpośrednio można tylko po jednej klasie. Przykładowy kod pokazano na listingu 8.3.

Listing 8.3. Implementowanie interfejsu

```
public class Contact : PdaItem, IListable, IComparable
{
    // ...

    #region IComparable Members
    /// <summary>
    /// 
    /// </summary>
    /// <param name="obj"></param>
    /// <returns>
    /// Mniej niż zero — dana instancja ma wartość mniejszą niż obj.
    /// Zero — dana instancja ma wartość równą obj.
    /// Więcej niż zero — dana instancja ma wartość większą niż obj.
    /// </returns>
    public int CompareTo(object? obj) => obj switch
    {
        null => 1,
        Contact contact when ReferenceEquals(this, contact) => 0,
```

```

Contact { LastName: string lastName }
    when LastName.CompareTo(lastName) != 0 =>
        LastName.CompareTo(lastName),
Contact { FirstName: string firstName }
    when FirstName.CompareTo(firstName) != 0 =>
        FirstName.CompareTo(firstName),
Contact _ => 0,
    => throw new ArgumentException(
        $"Parametr nie jest wartością typu { nameof(Contact) }",
        nameof(obj))
};

#endregion

#region Składowe interfejsu IListable
string?[] IListable.CellValues
{
    get
    {
        return new string?[]
        {
            FirstName,
            LastName,
            Phone,
            Address
        };
    }
}
#endregion
}

```

Gdy w klasie zadeklarowano, że zawiera ona implementację interfejsu, trzeba zaimplementować w niej wszystkie (abstrakcyjne¹) składowe z podanego interfejsu. W klasie abstrakcyjnej można utworzyć abstrakcyjną implementację składowych interfejsu. W implementacji nieabstrakcyjnej można w ciele metody zgłosić wyjątek typu `NotImplementedException`, trzeba jednak w jakiś sposób udostępnić implementację składowej.

Jedną z ważnych cech interfejsów jest to, że nie można tworzyć ich instancji. Nie możesz użyć operatora `new` do utworzenia instancji interfejsu. Interfejsy nie mają ani konstruktorów, ani finalizatorów. Instancje interfejsów są dostępne tylko w postaci instancji typów z implementacją danych interfejsów. Ponadto interfejsy nie mogą zawierać składowych statycznych². Interfejsy często stosuje się w celu uzyskania polimorfizmu, a polimorfizm bez instancji typów z implementacją danego interfejsu nie jest przydatny.

Każda (niezaimplementowana³) składowa interfejsu jest abstrakcyjna (w klasach pochodnych konieczne jest zaimplementowanie tych składowych). Dlatego nie można jawnie używać modyfikatora `abstract` do składowych interfejsu⁴.

¹ Możliwość umieszczania nieabstrakcyjnych składowych w interfejsach została wprowadzona w C# 8.0, jednak do końcowej części rozdziału takie składowe są pomijane.

² W wersjach starszych niż C# 8.0.

³ Tylko w wersjach C# 8.0 i nowszych.

⁴ W wersjach starszych niż C# 8.0.

Składowe interfejsu można zaimplementować w typie na dwa sposoby — **z jawnie podawanym interfejsem** lub **bez jawnego podawania interfejsu**. Do tej pory pokazano tylko implementację bez jawnego podawania interfejsu, polegającą na tym, że składowa interfejsu jest publiczną składową typu z jego implementacją.

Składowe z jawnie podawanym interfejsem

Metody z jawnie podawanym interfejsem są dostępne tylko w wyniku wywołania ich za pomocą samego interfejsu. Zwykle odbywa się to po zrzutowaniu obiektu na interfejs. Aby na przykład wywołać właściwość `IListable.CellValues` na listingu 8.4, musisz najpierw zrzutować obiekt typu `Contact` na interfejs `IListable`. Jest tak, ponieważ właściwość `CellValues` wymaga jawnego podawania interfejsu.

Listing 8.4. Wywoływanie składowych z jawnie podawanym interfejsem

```
string?[] values;
Contact contact = new Contact("Inigo Montoya");

// ...

// BŁĄD: nie można wywołać właściwości CellValues() bezpośrednio
//       dla obiektu contact.
// values = contact.CellValues;

// Najpierw trzeba zrzutować obiekt na interfejs IListable.
values = ((IListable)contact).CellValues;
// ...
```

Tu rzutowanie typu i wywołanie właściwości `CellValues` znajdują się w tej samej instrukcji. Inna możliwość to przypisanie zmiennej `contact` do zmiennej typu `IListable` przed wywołaniem wspomnianej właściwości.

Aby dodać implementację składowej z jawnie podawanym interfejsem, poprzedź nazwę składowej nazwą interfejsu (zobacz listing 8.5).

Listing 8.5. Implementacja składowej z jawnie podawanym interfejsem

```
public class Contact : PdaItem, IListable, IComparable
{
    // ...

    #region Składowe interfejsu IListable
    string?[] IListable.CellValues
    {
        get
        {
            return new string?[]
            {
                FirstName,
                LastName,
                Phone,
                Address
            };
        }
    }
}
```

```
        };
    }
}

#endif
```

Na listingu 8.5 znajduje się implementacja właściwości `CellValues` z jawnie podawanym interfejsem. Nazwa właściwości jest tu poprzedzona nazwą interfejsu (`IListable`). Ponadto ponieważ ta właściwość jest bezpośrednio powiązana z interfejsem, nie potrzebuje modyfikatorów `virtual`, `override` ani `public`. Co więcej, te modyfikatory są w omawianym kontekście niedozwolone. Metoda nie jest wtedy traktowana jak publiczna składowa klasy, dlatego oznaczenie jej modyfikatorem `public` byłoby mylące.

Należy zauważać, że choć w interfejsach nie można stosować słowa kluczowego `override`, w tekście używane jest określenie „przesłaniać” dotyczące składowych z implementacją sygnatury zdefiniowanej w interfejsie.

Składowe bez jawnie podawanego interfejsu

Zauważ, że przy metodzie `CompareTo()` na listingu 8.5 nie występuje przedrostek `IComparable`. Jest to więc metoda bez jawnie podawanego interfejsu. W tym podejściu konieczne jest tylko to, by składowa była publiczna i by jej sygnatura pasowała do sygnatury składowej interfejsu. W implementacji składowej interfejsu nie trzeba używać słowa kluczowego `override` ani określić w inny sposób, że składowa jest powiązana z interfejsem. Ponadto ponieważ składowa jest zadeklarowana w taki sam sposób jak inne składowe klasy, w kodzie wywołującym takie składowe można to zrobić bezpośrednio. Wywołania wyglądają wtedy tak jak dla dowolnych innych składowych klasy:

```
result = contact1.CompareTo(contact2);
```

Tak więc składowe bez jawnie podawanego interfejsu nie wymagają rzutowania, ponieważ możliwość bezpośredniego wywoływania takich składowych w klasach z ich implementacją nie jest zablokowana.

Różne modyfikatory, które nie są dozwolone w implementacji składowych z jawnie podawanym interfejsem, są wymagane lub opcjonalne w implementacji niejawnej. Na przykład składowa bez jawnie podawanego interfejsu musi mieć modyfikator `public`. Opcjonalnie można też dodać modyfikator `virtual` (w zależności od tego, czy w klasach pochodnych można przesłaniać implementację danej składowej). Jeśli pominiesz modyfikator `virtual`, składowa będzie traktowana tak, jakby była opatrzona modyfikatorem `sealed`.

Porównanie implementacji z jawnie podawanym interfejsem i bez podawania interfejsu

Najważniejsza różnica między składowymi z jawnie podawanym interfejsem i bez podawania interfejsu dotyczy nie składni używanej w deklaracji metody, ale możliwości dostępu do metody za pomocą instancji typu lub samego interfejsu.

W trakcie tworzenia hierarchii klas warto odzwierciedlić relacje „jest odmianą” ze świata rzeczywistego (na przykład żyrafa jest odmianą ssaka). Są to relacje *semantyczne*. Interfejsy często służą do reprezentowania relacji *mechanizmu*. O obiekcie typu `PdaItem` nie powiemy, że „jest odmianą” obiektu porównywalnego, ale taki obiekt może zawierać implementację interfejsu `IComparable`. Ten interfejs nie ma nic wspólnego z modelem semantycznym — opisuje jedynie szczegół mechanizmu implementacji danego obiektu. Składowe z jawnie podawanym interfejsem to technika umożliwiająca oddzielenie kwestii związanych z mechanizmami od aspektów dotyczących modelu. Wymuszanie na jednostce wywołującej przekształcania obiektu na interfejs (na przykład `IComparable`), zanim będzie można potraktować obiekt jako „porównywalny”, pozwala wyraźnie podkreślić w kodzie, kiedy należy zwrócić uwagę na model semantyczny, a kiedy ważne są mechanizmy implementacji.

Zwykle warto dbać o to, by publiczne elementy klasy były w całości aspektami modelu. Należy w jak największym stopniu ograniczyć występowanie zewnętrznych mechanizmów. Niestety, w platformie .NET czasem nie da się ich uniknąć. Nie można na przykład przypisać żyrafie skrótu potrzebnego przy haszowaniu lub przekształcić jej na łańcuch znaków. Można jednak pobrać wartość skrótu z klasy `Giraffe` (za pomocą wywołania `GetHashCode()`) i przekształcić go na typ `string` (przy użyciu metody `ToString()`). Zastosowanie wspólnej klasy bazowej `object` w platformie .NET powoduje połączenie kodu modelu z kodem mechanizmu, choć tylko w ograniczonym zakresie.

Poniżej znajduje się zestaw wytycznych, które pomogą Ci dokonać wyboru między składowymi z jawnie podawanym interfejsem i bez podawania interfejsu.

■ Czy dana składowa reprezentuje jedną z podstawowych funkcji klasy?

Pomyśl o implementacji właściwości `CellValues` w klasie `Contact`. Ta składowa nie jest integralną częścią typu `Contact`; jest tylko poboczną składową, z której prawdopodobnie korzystać będzie tylko klasa `ConsoleListControl`. Dlatego nie ma sensu, by składowa ta była bezpośrednio widoczna w obiekcie `Contact` i dodatkowo zmniejszała czytelność długiej listy składowych.

Następnie przyjrzyj się składowej `IFileCompression.Compress()`. Dodanie składowej `Compress()` bez jawnie podawanego interfejsu do klasy `ZipCompression` jest uzasadnionym rozwiązaniem. Metoda `Compress()` reprezentuje jedną z podstawowych operacji klasy `ZipCompression`, dlatego powinna być bezpośrednio dostępna w tej klasie.

■ Czy nazwa składowej interfejsu dobrze nadaje się na nazwę składowej w określonej klasie?

Pomyśl o interfejsie `ITrace` ze składową `Dump()`, która zapisuje dane klasy w dzienniku śledzenia. Metoda `Dump()` bez jawnie podawanego interfejsu w klasach `Person` lub `Truck` spowoduje, że nie będzie zrozumiałe, jakie operacje ta metoda wykonuje. Zamiast tego lepiej zaimplementować składową z jawnie podawanym interfejsem, aby metodę `Dump()` można było wywoływać tylko za pomocą typu `ITrace`, gdzie jej znaczenie jest bardziej jasne. Rozważ użycie składowej z jawnie podawanym interfejsem, jeśli w klasie z implementacją znaczenie określonej składowej nie jest jasne.

- Czy w klasie znajduje się już składowa o identycznej sygnaturze?

Implementacja składowej z jawnie podawanym interfejsem nie powoduje dodania elementu do przestrzeni deklaracji typu danych zawierającego kod składowej.

Dlatego jeśli typ obejmuje już składową, która może spowodować konflikt, można dodać drugą składową o tej samej nazwie lub sygnaturze, pod warunkiem że będzie to składowa z jawnie podawanym interfejsem.

Często decyzja o zastosowaniu składowych z jawnie podawanym interfejsem lub bez podawania interfejsu sprawdza się do posłużenia się intuicją. Jednak przedstawione wcześniej pytania zawierają sugestie określające, jakie kwestie warto rozważyć w trakcie dokonywania tego wyboru. Ponieważ zmiana składowej z wersji bez podawania interfejsu w wersję z podaniem interfejsu skutkuje naruszeniem poprawności kodu, w razie wątpliwości lepiej skłaniać się ku implementacji z jawnym podaniem interfejsu. Można wtedy później zmienić ją na wersję bez podawania interfejsu. Ponadto ponieważ nie wszystkie składowe muszą być implementowane w ten sam sposób, dla niektórych metod można wybrać implementację z podaniem interfejsów, a dla innych wersję bez podawania interfejsu.

Przekształcanie między klasą z implementacją i interfejsami

Konwersja z typu z implementacją na implementowany interfejs (podobnie jak z typu pochodzącego na klasę bazową) odbywa się niejawnie. Operator rzutowania nie jest potrzebny, ponieważ instancja typu z implementacją zawsze udostępnia wszystkie składowe interfejsu. Dlatego obiekt zawsze można z powodzeniem przekształcić na typ interfejsu.

Jednak choć konwersja z typu z implementacją na implementowany interfejs zawsze kończy się sukcesem, implementację danego interfejsu może obejmować wiele różnych typów. Dlatego nigdy nie wiadomo, czy rzutowanie „w dół” z interfejsu na jeden z typów z implementacją zakończy się powodzeniem. Z tego względu przekształcanie z interfejsu na typy z implementacją wymaga jawnego rzutowania.

Dziedziczenie interfejsów

Możliwe jest tworzenie interfejsów pochodnych od innych interfejsów. W efekcie powstaje interfejs dziedziczący wszystkie składowe interfejsów bazowych⁵. Na listingu 8.6 pokazano interfejs bezpośrednio dziedziczący po IReadableSettingsProvider, używany jako bezpośredni interfejs bazowy.

⁵ Wyjątkiem są wprowadzone w C# 8.0 składowe nieprywatne.

Listing 8.6. Tworzenie interfejsu pochodnego od innego interfejsu

```

interface IReadableSettingsProvider
{
    string GetSetting(string name, string defaultValue);
}
interface ISettingsProvider : IReadableSettingsProvider
{
    void SetSetting(string name, string value);
}
class FileSettingsProvider : ISettingsProvider
{
    #region Składowe z interfejsu ISettingsProvider
    public void SetSetting(string name, string value)
    {
        // ...
    }
    #endregion

    #region Składowe z interfejsu IReadableSettingsProvider
    public string GetSetting(string name, string defaultValue)
    {
        // ...
    }
    #endregion
}

```

W tym kodzie ISettingsProvider jest pochodny od interfejsu IReadableSettingsProvider, dlatego dziedziczy składowe tego ostatniego. Gdyby IReadableSettingsProvider także miał bezpośredni interfejs bazowy, ISettingsProvider odziedziczyłby również składowe tego interfejsu bazowego. Pełny zestaw interfejsów w hierarchii dziedziczenia to suma wszystkich interfejsów bazowych.

Zauważ, że gdyby metoda GetSetting() wymagała podawania interfejsu, trzeba byłoby użyć interfejsu IReadableSettingsProvider. Przedstawiony na listingu 8.7 kod z deklaracją dla interfejsu ISettingsProvider się nie skompiluje.

Listing 8.7. Deklaracja składowej z jawnie podawanym interfejsem, ale bez wskazania zawierającego ją interfejsu, prowadzi do błędu

```

// BŁĄD: metoda GetSetting() nie pochodzi z interfejsu ISettingsProvider.
string ISettingsProvider.GetSetting(
    string name, string defaultValue)
{
    // ...
}

```

Wynik działania kodu z listingu 8.7 pokazano w danych wyjściowych 8.2.

DANE WYJŚCIOWE 8.2.

W jawniej deklaracji interfejsu nie znaleziono elementu 'ISettingsProvider.GetSetting' wśród składowych interfejsu, które można implementować.

Te dane wyjściowe pojawiają się razem z błędem informującym o tym, że metoda `IReadableSettingsProvider.GetSetting()` nie jest zaimplementowana. Pełna nazwa składowej używana dla składowej z jawnie podawanym interfejsem musi obejmować nazwę interfejsu, w którym tę składową pierwotnie zadeklarowano.

Choć w klasie zaimplementowano interfejs `ISettingsProvider` pochodny od interfejsu bazowego `IReadableSettingsProvider`, można w niej jawnie zadeklarować implementację obu interfejsów, co pokazano na listingu 8.8.

Listing 8.8. Używanie interfejsu bazowego w deklaracji klasy

```
class FileSettingsProvider : ISettingsProvider,  
    IReadableSettingsProvider  
{  
    #region Składowe z interfejsu ISettingsProvider  
    public void SetSetting(string name, string value)  
    {  
        // ...  
    }  
    #endregion  
  
    #region Składowe z interfejsu IReadableSettingsProvider  
    public string GetSetting(string name, string defaultValue)  
    {  
        // ...  
    }  
    #endregion  
}
```

Na tym listingu implementacja interfejsu w klasie wygląda tak samo. Choć deklaracja dodatkowego interfejsu w nagłówku klasy nie jest potrzebna, poprawia czytelność kodu.

Decyzja, by podać kilka interfejsów zamiast jednego połączonego, zależy głównie od tego, czego projektant interfejsu wymaga od klasy z implementacją. Podając interfejs `IReadableSettingsProvider`, projektant komunikuje, że trzeba zaimplementować tylko dostawcę ustawień, który je pobiera. Nie jest konieczna możliwość zapisu tych ustawień. Upraszczają to implementację, ponieważ nie trzeba zajmować się kodem odpowiedzialnym za zapis ustawień.

Implementacja interfejsu `ISettingsProvider` jest związana z założeniem, że klasa, która potrafi zapisywać ustawienia, ale ich nie wczytuje, nie jest potrzebna. Relacja dziedziczenia między interfejsami `ISettingsProvider` i `IReadableSettingsProvider` wymusza więc implementację obu interfejsów w klasie z implementacją interfejsu `ISettingsProvider`.

Na koniec ważna uwaga — choć *dziedziczenie* jest w tym kontekście poprawną nazwą, bardziej precyzyjne byłoby określenie, że interfejs reprezentuje kontrakt, a w jednym kontraku można określić, że należy przestrzegać także innego. Tak więc kod `ISettingsProvider : IReadableSettingsProvider` oznacza, że kontrakt `ISettingsProvider` wymaga przestrzegania także kontraktu `IReadableSettingsProvider`. Nie oznacza to natomiast, że interfejs `ISettingsProvider` „jest odmianą” interfejsu `IReadableSettingsProvider`. W dalszej części rozdziału używane są jednak pojęcia związane z dziedziczeniem, co jest zgodne ze standartową terminologią z obszaru języka C#.

Dziedziczenie po wielu interfejsach

Podobnie jak w klasach można implementować wiele interfejsów, tak interfejsy mogą dziedziczyć po wielu innych interfejsach. Składnia wygląda tu tak samo jak w przypadku tworzenia klas pochodnych i implementowania interfejsów w klasach. Pokazano ją na listingu 8.9.

Listing 8.9. Dziedziczenie po wielu interfejsach

```
interface IReadableSettingsProvider
{
    string GetSetting(string name, string defaultValue);
}
interface IWriteableSettingsProvider
{
    void SetSetting(string name, string value);
}
interface ISettingsProvider : IReadableSettingsProvider,
    IWriteableSettingsProvider
{
}
```

Interfejsy bez składowych są tworzone rzadko, jeśli jednak jakieś dwa interfejsy bardzo często są stosowane wspólnie, można zastosować przedstawione tu rozwiązanie. Różnica między listingami 8.9 i 8.6 polega na tym, że teraz można zaimplementować interfejs IWriteableSettingsProvider bez dodawania mechanizmu odczytu ustawień. Klasa FileSettingsProvider z listingu 8.6 nie wymaga zmian, jeśli jednak zechcesz zastosować składowe z jawnie podawanym interfejsem, będziesz musiał zmodyfikować sposób określania interfejsu, do którego ta składowa należy.

Metody rozszerzające i interfejsy

Początek
3.0

Jedną z najważniejszych cech metod rozszerzających jest to, że można je stosować nie tylko do klas, ale też do interfejsów. Używana tu składnia jest taka sama jak dla metod rozszerzających w klasach. Rozszerzany typ (pierwszy parametr z przedrostkiem `this`) jest tu interfejsem. Na listingu 8.10 znajdziesz metodę rozszerzającą interfejs `IListable`. Jest ona zadeklarowana w klasie `Listable`.

Listing 8.10. Metoda rozszerzająca interfejs

```
class Program
{
    public static void Main()
    {
        Contact[] contacts = new Contact[] {
            new Contact(
                "Dariusz", "Tracz",
                "ul. Wysoka 123, 45-287 Poznań",
                "123-123-1234")
        // ...
    };
}
```

// Klasy są niejawnie przekształcane

```
// na obsługiwane interfejsy.
contacts.List(Contact.Headers);

Console.WriteLine();

Publication[] publications = new Publication[3] {
    new Publication("Koniec z nędzą. Zadanie dla naszego pokolenia",
        "Jeffrey Sachs", 2006),
    new Publication("Ortodoksyjna",
        "G.K. Chesterton", 1908),
    new Publication(
        "Autostopem przez galaktykę",
        "Douglas Adams", 1979)
};
publications.List(Publication.Headers);
}

static class Listable
{
    public static void List(
        this IListable[] items, string?[] headers)
    {
        int[] columnWidths = DisplayHeaders(headers);

        for (int itemCount = 0; itemCount < items.Length; itemCount++)
        {
            string?[] values = items[itemCount].ColumnValues;

            DisplayItemRow(columnWidths, values);
        }
    }
    // ...
}
```

W tym przykładzie metoda rozszerzająca przyjmuje parametr `IListable[]`, a nie `IListable` (choć ten też można by było zastosować). To pokazuje, że C# umożliwia tworzenie metod rozszerzających nie tylko dla określonego typu, ale też dla kolekcji obiektów danego typu. Obsługa metod rozszerzających jest podstawą, na której zbudowano technologię LINQ. `IEnumerable` to podstawowy interfejs implementowany we wszystkich kolekcjach. Dzięki zdefiniowaniu metod rozszerzających dla interfejsu `IEnumerable` dodano obsługę technologii LINQ do wszystkich kolekcji. To znacznie zmieniło sposób programowania z wykorzystaniem kolekcji. Szczegółowe omówienie tego zagadnienia znajdziesz w rozdziale 15.

Koniec
3.0

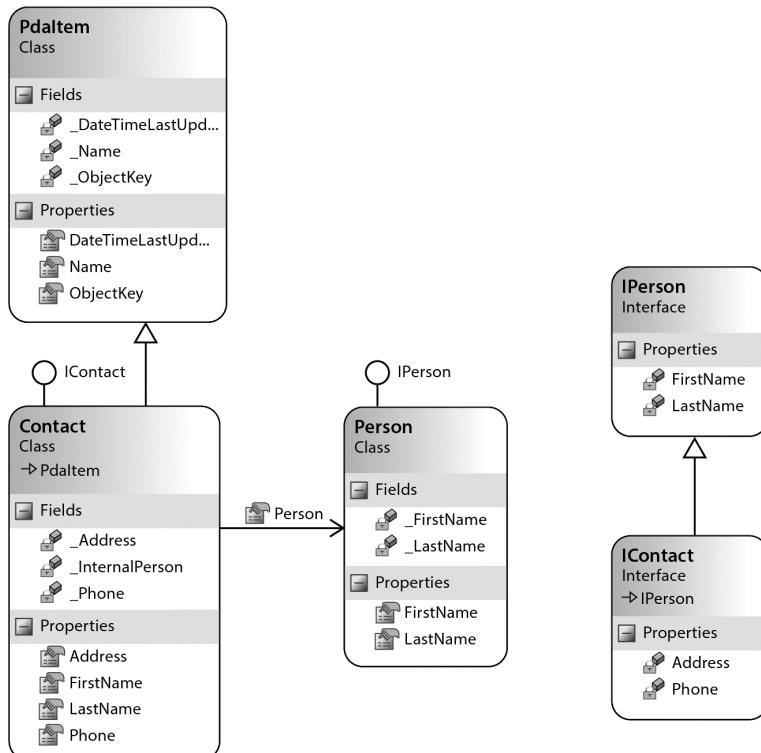
■ ZAGADNIENIE DLA ZAAWANSOWANYCH I POCZĄTKUJĄCYCH

Diagramy interfejsów

Interfejsy na diagramach UML-owych⁶ przyjmują dwie postacie. Pierwsza z nich związana jest z przedstawianiem interfejsu za pomocą relacji dziedziczenia (podobnie prezentowane

⁶ UML (ang. *Unified Modeling Language*) to standardowa specyfikacja służąca do przedstawiania projektów obiektowych za pomocą notacji graficznej.

jest dziedziczenie klas). Na rysunku 8.1 w ten sposób pokazana jest relacja między interfejsami IPerson i IContact. Druga możliwość to przedstawianie interfejsu za pomocą małego kółka w kształcie *lizaka*, widocznego na rysunku 8.1 przy interfejsach IPerson i IContact.



Rysunek 8.1. Zastosowanie agregacji i interfejsu do rozwiązania problemu dziedziczenia po tylko jednej klasie

Na rysunku 8.1 klasa Contact jest pochodna od klasy PdaItem i zawiera implementację interfejsu IContact. Ponadto w ramach relacji agregacji zawiera obiekt klasy Person, w której zaimplementowany jest interfejs IPerson. Choć w narzędziu Class Designer w środowisku Visual Studio to rozwiązanie nie jest używane, relacja między interfejsami a klasą jest często przedstawiana za pomocą strzałki reprezentującej dziedziczenie. W tym podejściu należało by zmienić „lizak” z nazwą IPerson na strzałkę z klasy Person do tego interfejsu.

Zarządzanie wersjami

Przed pojawiением się wersji C# 8.0 nie należało modyfikować interfejsów w trakcie pisania nowych wersji komponentów i aplikacji przeznaczonych dla innych programistów. Ponieważ interfejsy definiują kontrakt między klasą, która je implementuje, a klasą używającą interfejsu, modyfikacja interfejsu oznacza zmianę kontraktu. Może to spowodować, że kod używający interfejsu przestanie działać.

Modyfikacja lub usunięcie sygnatury składowej w interfejsie jest oczywiście zmianą, która może naruszać działanie kodu, ponieważ wywołanie tej składowej nie skompiluje się (konieczne jest więc wprowadzenie poprawek). To samo dotyczy modyfikacji sygnatur publicznych i chronionych składowych w klasie. Jednak (inaczej niż w klasach) także dodanie składowych do interfejsu może uniemożliwić komplikację kodu, jeśli programista nie wprowadzi dodatkowych poprawek. Problem polega na tym, że w klasie implementującej interfejs trzeba uwzględnić go w całości i zaimplementować wszystkie składowe. Gdy pojawiają się nowe składowe interfejsu, kompilator wymaga, aby programista dodał je w klasie implementującej ten interfejs.

W C# 8.0 reguła „nie modyfikuj interfejsów” działa trochę inaczej. W tej wersji języka umożliwiono udostępnianie domyślnej implementacji składowych interfejsów. Dzięki temu dodanie składowej nie musi powodować błędów komplikacji (choć nadal nie można usuwać ani modyfikować istniejących składowych w sposób naruszający zgodność wersji). W wersjach starszych niż C# 8.0 efekt podobny do modyfikacji interfejsu można było uzyskać, wprowadzając dodatkowy interfejs. W tym podrozdziale opisane są obie te możliwości.

Wskazówka

NIE dodawaj składowych do interfejsu, który został już wcześniej udostępniony.

Zarządzanie wersjami interfejsów przed C# 8.0

Interfejs `IDistributedSettingsProvider` z listingu 8.11 to dobry przykład ilustrujący rozszerzanie interfejsu w sposób zgodny z już istniejącym kodem. Wyobraź sobie, że w pierwszej wersji kodu zdefiniowany był tylko interfejs `ISettingsProvider` (tak jak na listingu 8.6). Jednak w trakcie prac nad następną wersją okazało się, że ustawienia mogą trafić do różnych zasobów (reprezentowanych przez identyfikatory URI⁷) z różnych maszyn. Aby umożliwić ich obsługę, utworzono interfejs `IDistributedSettingsProvider` pochodny od `ISettingsProvider`.

Listing 8.11. Tworzenie interfejsu pochodnego od innego interfejsu

```
interface IDistributedSettingsProvider : ISettingsProvider
{
    /// <summary>
    /// Pobieranie ustawień dla konkretnego identyfikatora URI.
    /// </summary>
    /// <param name="uri">
    /// Identyfikator URI, którego dotyczą ustawienia.</param>
    /// <param name="name">Nazwa ustawienia.</param>
    /// <param name="defaultValue">
    /// Wartość zwracana, gdy nie można znaleźć danego ustawienia.</param>
    /// <returns>Określone ustawienie.</returns>
    string GetSetting(
        string machineName, string name, string defaultValue);
```

⁷ Od ang. *universal resource identifier*.

```

    /// <summary>
    /// Określanie ustawienia dla konkretnego identyfikatora URI.
    /// </summary>
    /// <param name="uri">
    /// Identyfikator URI, którego dotyczy ustawienie.</param>
    /// <param name="name">Nazwa ustawienia.</param>
    /// <param name="value">Zapisywana wartość.</param>
    /// <returns>Określone ustawienie.</returns>
void SetSetting(
    string machineName, string name, string value);
}

```

Ważne jest to, że programiści klas z implementacją interfejsu `ISettingsProvider` mogą zdecydować się zmodyfikować implementację i dodać obsługę interfejsu `IDistributedSettingsProvider` lub zignorować nowy interfejs.

Gdyby zamiast utworzenia nowego interfejsu dodano metody związane z identyfikatorami URI do interfejsu `ISettingsProvider`, klasy z implementacją tego interfejsu mogłyby zgłaszać wyjątki w czasie wykonywania programu i niemożliwe byłoby skompilowanie ich razem z nową definicją interfejsu. Taka zmiana powoduje, że poprzednia wersja kodu przejstaje działać poprawnie — zarówno na poziomie kodu binarnego, jak i na poziomie kodu źródłowego.

Modyfikowanie interfejsów na etapie ich pisania jest oczywiście akceptowalne, choć może wymagać dużo pracy, jeśli dany interfejs jest implementowany w wielu klasach. Jednak po udostępnieniu interfejsu nie należy go zmieniać. Można wtedy utworzyć drugi interfejs (może on być pochodny od pierwotnego). Na listingu 8.11 znajdują się komentarze w XML-u opisujące składowe interfejsu. Omówienie takich komentarzy znajdziesz w rozdziale 10.

Zarządzanie wersjami interfejsów w C# 8.0 i nowszych edycjach

Do tej pory tylko wspomniano o nowych cechach interfejsów z wersji C# 8.0; oprócz tego były one ignorowane. W tym punkcie się to zmieni — znajdziesz tu omówienie wprowadzonych w C# 8.0 **domyślnych składowych interfejsów**. Wcześniej opisano, że w wersjach starszych niż C# 8.0 modyfikacje w udostępnionym interfejsie powodowały, że kod implementujący ten interfejs przestawał działać. Dlatego udostępnionych interfejsów nie należało zmieniać. Jednak od wersji C# 8.0 i platformy .NET Core 3.0 Microsoft wprowadził nową cechę języka, która pozwala tworzyć w interfejsach składowe z implementacją (czyli składowe konkretne zamiast samych deklaracji). Rozważ na przykład właściwość `CellColors` z listingu 8.12.

Listing 8.12. Zarządzanie wersjami interfejsów ze składowymi domyślnymi

8.0

```

public interface IListable
{
    // Zwraca wartość każdej komórki z wiersza
    string?[] CellValues { get; }

    ConsoleColor[] CellColors
    {

```

```

get
{
    var result = new ConsoleColor[CellValues.Length];
    // Zapelnianie tablicy za pomocą generycznej metody z klasy Array
    // (zobacz rozdział 12.)
    Array.Fill(result, DefaultColumnColor);
    return result;
}
}

public static ConsoleColor DefaultColumnColor { get; set; }

public class Contact : PdaItem, IListable
{
    //...

    #region IListable
    public string[] CellValues
    {
        get
        {
            return new string[]
            {
                FirstName,
                LastName,
                Phone,
                Address
            };
        }
    }
    // *** Brak implementacji metody CellColors ***
    #endregion IListable
}

public class Publication : IListable
{
    //...

    #region IListable
    string?[] IListable.CellValues
    {
        get
        {
            return new string?[]
            {
                Title,
                Author,
                Year.ToString()
            };
        }
    }
}

ConsoleColor[] IListable.CellColors
{
    get
}

```

```

    {
        string?[] columns = ((IListable)this).CellValues;
        ConsoleColor[] result = ((IListable)this).CellColors;
        if (columns[YearIndex] ?.Length != 4)
        {
            result[YearIndex] = ConsoleColor.Red;
        }
        return result;
    }

}
#endregion IListable
// ...
}

```

Zwróć uwagę na pojawienie się w tym kodzie gettera właściwości `CellColors`. Zawiera on implementację, choć jest składową interfejsu. Ten mechanizm jest nazywany *domyślnymi składowymi interfejsu*, ponieważ zapewnia implementację domyślnej metody, dzięki czemu kod skompiluje się bez konieczności wprowadzania w nim zmian, nawet jeśli do interfejsu dodane zostały składowe. Na przykład w klasie `Contact` nie istnieje implementacja gettera właściwości `CellColors`, dlatego klasa ta używa implementacji domyślnej udostępnionej w interfejsie `IListable`.

Nie jest zaskoczeniem, że w klasie implementującej interfejs można przesłonić implementację domyślną, aby udostępnić inne działania, które w danej klasie mają więcej sensu. Jest to spójne z zapewnianiem polimorfizmu, co opisane zostało na początku rozdziału.

Z domyślnymi składowymi interfejsu powiązane są też dodatkowe mechanizmy. Mają one przede wszystkim umożliwić refaktoryzację składowych domyślnych interfejsu (choć nie wszyscy zgadzają się z tym stwierdzeniem). Stosowanie ich w innych celach zwykle wskazuje na błędy w strukturze kodu, ponieważ oznacza to, że interfejs jest używany do czegoś innego niż polimorfizm. W tabeli 8.1 opisane są dodatkowe konstrukcje języka i ich ważne ograniczenia.

Warto zwrócić uwagę na kilka punktów z tabeli 8.1. Po pierwsze należy zauważyć, że w interfejsach niedostępne są automatycznie implementowane właściwości, ponieważ pola instancji (wiązane z automatycznie implementowanymi właściwościami) nie są obsługiwane. Jest to ważna różnica w porównaniu z klasami abstrakcyjnymi, które obsługują pola instancji i automatycznie implementowane właściwości.

Po drugie domyślny poziom dostępu jest inny dla składowych instancji i składowych statycznych. Składowe statyczne są domyślnie prywatne, natomiast składowe instancji — publiczne. Ta różnica wynika z tego, że składowe statyczne zawsze mają implementację i są bardzo podobne do — też domyślnie prywatnych — składowych statycznych klas. Natomiast składowe instancji interfejsu mają umożliwiać polimorfizm, dlatego domyślnie są publiczne. Jest to zgodne z tradycyjnym działaniem interfejsów w wersjach starszych niż C# 8.0.

Tabela 8.1. Mechanizmy do refaktoryzacji składowych domyślnych interfejsów

Związany z interfejsami mechanizm wprowadzony w C# 8.0	Przykładowy kod
Składowe statyczne Można definiować w interfejsie składowe statyczne, w tym pola, konstruktory i metody. Dotyczy to też definiowania statycznej metody Main — punktu wejścia do programu.	<pre>public interface ISampleInterface { private static string? _Field; public static string? Field { get => _Field; private set => _Field = value; } static ISampleInterface() => Field = "Nelson Mandela"; public static string? GetField() => Field; }</pre>
Implementacje właściwości i metod instancji Możesz implementować w interfejsach właściwości i inne składowe. Ponieważ pola instancji nie są obsługiwane, we właściwościach nie można używać podstawowych pól. Z powodu braku pól instancji nie można też tworzyć automatycznie implementowanych właściwości. Aby uzyskać dostęp do domyślnej zaimplementowanej właściwości, trzeba zrzutować obiekt na typ interfejsu zawierającego tę składową. W klasie Person domyślna składowa interfejsu nie będzie dostępna (chyba że zostanie zaimplementowana w tej klasie).	<pre>public interface IPerson { // Definicje standardowych właściwości abstrakcyjnych string FirstName { get; set; } string LastName { get; set; } string MiddleName { get; set; } // Implementacja właściwości i metod instancji public string Name => GetName(); public string GetName() => \$"{FirstName} {LastName}"; public class Person { // ... } public class Program { public static void Main() { Person inigo = new Person("Inigo", "Montoya"); Console.WriteLine(((IPerson)inigo).Name); } } }</pre>
Modyfikator dostępu public Domyślnie wszystkie składowe instancji w interfejsach są publiczne. Słowo kluczowe public możesz zastosować, aby jednoznacznie określić poziom dostępu do kodu. Kod CIL generowany przez kompilator jest identyczny dla wersji z modyfikatorem public i bez niego.	<pre>public interface IPerson { // Wszystkie składowe domyślnie są publiczne string FirstName { get; set; } public string LastName { get; set; } string Initials => \$"{FirstName[0]}{LastName[0]}"; public string Name => GetName(); public string GetName() => \$"{FirstName} {LastName}"; }</pre>
Modyfikator dostępu protected Zobacz punkt „Modyfikator dostępu protected”.	

Tabela 8.1. Mechanizmy do refaktoryzacji składowych domyślnych interfejsów — ciąg dalszy

Związany z interfejsami mechanizm wprowadzony w C# 8.0	Przykładowy kod
Modyfikator dostępu private Jest to ustawienie domyślne dla składowych statycznych. Modyfikator private powoduje, że składową można wywoływać tylko w interfejsie z jej deklaracją. Ta technika ma ułatwiać refaktoryzację domyślnych składowych interfejsu. Wszystkie składowe prywatne muszą mieć implementację.	<pre>public interface IPerson { string FirstName { get; set; } string LastName { get; set; } string Name => GetName(); private string GetName() => \$"{FirstName} {LastName}"; }</pre>
Modyfikator dostępu internal Składowe z modyfikatorem internal są widoczne tylko w podzespole, w którym zostały zadeklarowane.	<pre>public interface IPerson { string FirstName { get; set; } string LastName { get; set; } string Name => GetName(); internal string GetName() => \$"{FirstName} {LastName}"; }</pre>
Modyfikator dostępu protected internal Składowe z modyfikatorem protected internal są dostępne w bieżącym podzespole i w typach pochodnych od klasy z deklaracją danej składowej.	<pre>public interface IPerson { string FirstName { get; set; } string LastName { get; set; } string Name => GetName(); protected internal string GetName() => \$"{FirstName} {LastName}"; }</pre>
Modyfikator dostępu private protected Dostęp do składowych z modyfikatorem private protected jest możliwy tylko w zawierającym je interfejsie i interfejsach pochodnych od niego. Nawet klasy implementujące dany interfejs nie mają dostępu do składowych z modyfikatorem private protected. Obrzuca to właściwość PersonTitle w klasie Person.	<pre>class Program { static void Main() { IPerson? person = null; // W klasach niepochodnych nie można wywoływać // składowej z modyfikatorem private protected. // _ = person?.GetName(); Console.WriteLine(person); } } public interface IPerson { string FirstName { get; } string LastName { get; } string Name => GetName(); private protected string GetName() => \$"{FirstName} {LastName}"; } public interface IEmployee: IPerson { int EmployeeId => GetName().GetHashCode(); } public class Person : IPerson { public Person(string firstName, string lastName)</pre>

Tabela 8.1. Mechanizmy do refaktoryzacji składowych domyślnych interfejsów — ciąg dalszy

Związany z interfejsami mechanizm wprowadzony w C# 8.0	Przykładowy kod
Modyfikator dostępu private protected (ciąg dalszy)	<pre>{ FirstName = firstName ?? throw new ArgumentNullException(nameof(firstName)); LastName = lastName ?? throw new ArgumentNullException(nameof(lastName)); } public string FirstName { get; } public string LastName { get; } // Składowe interfejsu mające modyfikator private protected // są niedostępne w klasach pochodnych. // public int PersonTitle => // GetName().ToUpper(); }</pre>
Modyfikator virtual Zaimplementowane składowe interfejsu domyślnie są wirtualne (modyfikator <code>virtual</code>). Wtedy wywołanie składowej interfejsu skutkuje wywołaniem implementacji metody o tej sygnaturze z klas pochodnych. Możesz (podobnie jak w przypadku modyfikatora <code>public</code>) jawnie opatrzyć składową modyfikatorem <code>virtual</code> , aby zwiększyć czytelność kodu. Dla składowych interfejsu bez implementacji modyfikator <code>virtual</code> jest niedozwolony. Ponadto modyfikatora <code>virtual</code> nie można stosować razem z modyfikatorami <code>private</code> , <code>static</code> i <code>sealed</code> .	<pre>public interface IPerson { // Modyfikator virtual jest niedozwolony // dla składowych bez implementacji. /* virtual */ string FirstName { get; set; } string LastName { get; set; } virtual string Name => GetName(); private string GetName() => \$"{FirstName} {LastName}"; }</pre>
Modyfikator sealed Aby uniemożliwić przesłanianie metody w klasach pochodnych, należy opatrzyć metodę modyfikatorem <code>sealed</code> . To gwarantuje, że w klasach pochodnych nie będzie można zmodyfikować implementacji takiej metody. Więcej dowiesz się z listingu 8.13.	<pre>public interface IWorkflowActivity { // Prywatna, a więc niewirtualna private void Start() => Console.WriteLine("IWorkflowActivity.Start()..."); // Modyfikator sealed, aby uniemożliwić przesłanianie sealed void Run() { try { Start(); InternalRun(); } finally { Stop(); } } protected void InternalRun(); }</pre>

Tabela 8.1. Mechanizmy do refaktoryzacji składowych domyślnych interfejsów — ciąg dalszy

Związany z interfejsami mechanizm wprowadzony w C# 8.0	Przykładowy kod
Modyfikator sealed (ciąg dalszy)	// Prywatna, a więc niewirtualna private void Stop() => Console.WriteLine("IWorkflowActivity.Stop()..."); }
Modyfikator abstract Modyfikator abstract jest dozwolony tylko dla składowych bez implementacji. To słowo kluczowe nic jednak nie zmienia, ponieważ takie składowe domyślnie są abstrakcyjne. Wszystkie składowe abstrakcyjne są automatycznie wirtualne. Jawne opatrzenie składowych abstrakcyjnych modyfikatorem virtual skutkuje błędem komplikacji.	public interface IPerson { // Modyfikator virtual jest niedozwolony // dla składowych bez implementacji /* virtual */ abstract string FirstName { get ; set ; } string LastName { get ; set ; } // abstract is not allowed on members // with implementation /* abstract */ string Name => GetName(); private string GetName() => \$"{FirstName} {LastName}"; }
Interfejs i metody częściowe Możesz utworzyć implementację częściową metody bez danych wyjściowych (instrukcji return albo parametrów ref lub out) i opcjonalnie kompletną zaimplementowaną metodę w drugiej deklaracji tego samego interfejsu. Metody częściowe zawsze są prywatne i nie można do nich stosować modyfikatorów dostępu.	public partial interface IThing { string Value { get ; protected set ; } void SetValue(string value) { AssertValueIsValid(value); Value = value; } partial void AssertValueIsValid(string value); } public partial interface IThing { partial void AssertValueIsValid(string value) { // Błędna wartość powoduje zgłoszenie wyjątku. switch (value) { case null : throw new ArgumentNullException(nameof(value)); case ":" : throw new ArgumentException("Pusty łańcuch jest niedozwolony", nameof(value)); case string when string.IsNullOrEmpty (value): throw new ArgumentException("Spacje są niedozwolone", nameof(value)); }; } }

Ułatwienie hermetyzacji i polimorfizmu dzięki składowym interfejsu z modyfikatorem `protected`

W trakcie tworzenia klasy programista powinien uważać na dopuszczanie przesłaniania metod, ponieważ nie może kontrolować implementacji klas pochodnych. Metody wirtualne nie powinny zawierać krytycznego kodu, ponieważ może on nie być wywoływany, jeśli metoda zostanie przesłonięta w klasie pochodnej.

Na listingu 8.13 używana jest wirtualna metoda `Run()`. Jeśli programista klasy `WorkflowActivity` wywołuje metodę `Run()`, spodziewając się, że uruchomione zostaną krytyczne metody `Start()` i `Stop()`, mogą wystąpić problemy.

Listing 8.13. Nieostrożne poleganie na implementacji metody wirtualnej

```
public class WorkflowActivity
{
    private void Start()
    {
        // Kod krytyczny
    }
    public virtual void Run()
    {
        Start();
        // Jakiś operacje...
        Stop();
    }
    private void Stop()
    {
        // Kod krytyczny
    }
}
```

Programista nowej wersji metody `Run()` może nie wywołać krytycznych metod `Start()` i `Stop()`.

Teraz rozważ kompletną implementację tego scenariusza z następującymi wymogami dotyczącymi hermetyzacji:

- Przesłanianie metody `Run()` powinno być niedozwolone.
- Wywoływanie metod `Start()` i `Stop()` powinno być niemożliwe, ponieważ kolejność ich wykonywania jest pod całkowitą kontrolą zawierającego je typu (nazwanego tu `IWorkflowActivity`).
- Możliwe powinno być zastąpienie kodu z bloku *Jakiś operacje....*
- Jeśli sensowne jest przesłanianie typów `Start()` i `Stop()`, klasa z ich implementacją nie musi mieć możliwości ich wywoływania — są one częścią implementacji typu bazowego.
- W typach pochodnych należy umożliwić utworzenie metody `Run()`, jednak nie powinna być ona wywoływana, gdy wykonywana jest metoda `Run()` z typu `IWorkflowActivity`.

Aby umożliwić spełnienie wszystkich tych wymogów, w C# 8.0 udostępniono chronione składowe interfejsu (modyfikator `protected`), które nieco różnią się od chronionych składowych klas. Te różnice są przedstawione na listingu 8.14, a w danych wyjściowych 8.3 pokazane są efekty wykonania tego kodu.

Listing 8.14. Wymuszanie pożąданie hermetyzacji metody Run()

```

public interface IWorkflowActivity
{
    // Prywatna, a więc niewirtualna
    private void Start() =>
        Console.WriteLine(
            "IWorkflowActivity.Start()...");

    // Modyfikator sealed uniemożliwia przesyłanie
    sealed void Run()
    {
        try
        {
            Start();
            InternalRun();
        }
        finally
        {
            Stop();
        }
    }

    protected void InternalRun();

    // Prywatna, a więc niewirtualna
    private void Stop() =>
        Console.WriteLine(
            "IWorkflowActivity.Stop()...");
    }

public interface IExecuteProcessActivity : IWorkflowActivity
{
    protected void RedirectStandardInOut() =>
        Console.WriteLine(
            "IEexecuteProcessActivity.RedirectStandardInOut()...");

    // Jeśli metoda jest przesyłana, nie można używać modyfikatora sealed
    /* sealed */ void IWorkflowActivity.InternalRun()
    {
        RedirectStandardInOut();
        ExecuteProcess();
        RestoreStandardInOut();
    }

    protected void ExecuteProcess();
    protected void RestoreStandardInOut() =>
        Console.WriteLine(
            "IEexecuteProcessActivity.RestoreStandardInOut()...");
    }

class ExecuteProcessActivity : IExecuteProcessActivity
{
}

```

```

public ExecuteProcessActivity(string executablePath) =>
    ExecutableName = executablePath
    ?? throw new ArgumentNullException(nameof(executablePath));

public string ExecutableName { get; }

void IExecuteProcessActivity.RedirectStandardInOut()=>
    Console.WriteLine(
        "ExecuteProcessActivity.RedirectStandardInOut()...");

void IExecuteProcessActivity.ExecuteProcess() =>
    Console.WriteLine(
        $"ExecuteProcessActivity.IExecuteProcessActivity.ExecuteProcess()...");

public void Run()
{
    ExecuteProcessActivity activity
        = new ExecuteProcessActivity("dotnet");
    // Składowych chronionych nie można wywoływać w klasie
    // implementującej interfejs, nawet jeśli
    // są zaimplementowane w danej klasie.
    // ((IWorkflowActivity)this).InternalRun();
    // activity.RedirectStandardInOut();
    // activity.ExecuteProcess();
    Console.WriteLine(
        @$$"Wykonywanie niepolimorficznej metody Run() w procesie '{activity.ExecutableName}'.");
}

public class Program
{
    public static void Main()
    {
        ExecuteProcessActivity activity
            = new ExecuteProcessActivity("dotnet");

        Console.WriteLine(
            "Wywołanie ((IEexecuteProcessActivity)activity).Run()...");
        // Dane wyjściowe:
        // Wywolanie ((IEexecuteProcessActivity)activity).Run()...
        // IWorkflowActivity.Start()...
        // ExecuteProcessActivity.RedirectStandardInOut()...
        // IEexecuteProcessActivity.IExecuteProcessActivity.ExecuteProcess()...
        // IEexecuteProcessActivity.RestoreStandardInOut()...
        // IWorkflowActivity.Stop()...
        ((IEexecuteProcessActivity)activity).Run();

        // Dane wyjściowe:
        // Wywolanie activity.Run()...
        // Wykonywanie niepolimorficznej metody Run() w procesie 'dotnet'.
        Console.WriteLine();
        Console.WriteLine(
            "Wywołanie activity.Run()...");
        activity.Run();
    }
}

```

DANE WYJŚCIOWE 8.3.

```
Wywołanie ((IExecuteProcessActivity)activity).Run()...
IWorkflowActivity.Start()...
ExecuteProcessActivity.RedirectStandardInOut()...
ExecuteProcessActivity.IExecuteProcessActivity.ExecuteProcess()...
IExecuteProcessActivity.RestoreStandardInOut()...
IWorkflowActivity.Stop()...
Wywołanie activity.Run()...
Wykonywanie niepolimorficznej metody Run() w procesie 'dotnet'.
```

Zastanów się teraz, w jaki sposób listing 8.14 spełnia podane wcześniej wymogi.

- Zauważ, że metoda IWorkflowActivity.Run() jest opatrzona modyfikatorem sealed, dlatego jest niewirtualna. Chroni to ją przed zmianami implementacji w typach pochodnych. Każde wywołanie metody Run() z użyciem typu IWorkflowActivity spowoduje wykonanie implementacji z tego typu.
- Metody Start() i Stop() z typu IWorkflowActivity są prywatne, dlatego pozostają niewidoczne dla wszystkich innych typów. Choć typ IExecuteProcessActivity udostępnia operacje do uruchamiania i zatrzymywania działań, typ IWorkflowActivity nie zezwala na zastąpienie ich implementacji.
- W typie IWorkflowActivity zdefiniowana jest chroniona metoda InternalRun(), którą można przeciążyć w typie IExecuteProcessActivity (i w razie potrzeby także w typie ExecuteProcessActivity). Zauważ jednak, że składowe z typu ExecuteProcessActivity nie mogą wywoływać metody InternalRun(). Możliwe, że tej metody nigdy nie należy uruchamiać poza sekwencją obejmującą metody Start() i Stop(), dlatego tylko interfejsy (IWorkflowActivity lub IExecuteProcessActivity) w hierarchii typów mogą wywoływać tę chronioną składową.
- Wszystkie chronione składowe interfejsu mogą przesłaniać dowolne domyślne składowe interfejsu, przy czym trzeba to zrobić jawnie. Na przykład implementacje metod RedirectStandardInOut() i RestoreStandardInOut() w klasie ExecuteProcessActivity są poprzedzone przedrostkiem IExecuteProcessActivity. Ponadto, podobnie jak w przypadku chronionej metody InternalRun(), typ implementujący interfejs nie może wywoływać składowych chronionych. Na przykład w klasie ExecuteProcessActivity nie można wywoływać metod RedirectStandardInOut() i RestoreStandardInOut(), choć są one zaimplementowane w tej klasie.
- Obie metody (RedirectStandardInOut() i RestoreStandardInOut()) są wirtualne, choć tylko jedna z nich jest jawnie zadeklarowana w ten sposób. Składowe interfejsów są domyślnie wirtualne, chyba że dodany zostanie modyfikator sealed. Używana jest więc implementacja z ostatniego typu pochodnego. Dlatego gdy metoda IExecuteProcessActivity.InternalRun() wywołuje metodę RedirectStandardInOut(), używana jest implementacja z klasy ExecuteProcessActivity, a nie z interfejsu IExecuteProcessActivity.

- Implementacja typu pochodnego może zawierać metodę pasującą do sygnatury z modyfikatorem sealed z klasy bazowej. Na przykład, jeśli klasa ExecuteProcessActivity zawiera metodę Run() o sygnaturze zgodnej z metodą Run() z interfejsu IWorkflowActivity, użycia zostanie implementacja z danego typu, a nie z ostatniej klasy pochodnej. Dlatego wywołanie `((IExecuteProcessActivity)activity).Run()` w metodzie Program.Main() spowoduje wykonanie metody IExecuteProcessActivity.Run(), natomiast wywołanie `activity.Run()` uruchomi metodę ExecuteProcessActivity.Run() (activity jest tu typu ExecuteProcessActivity).

Podsumowując: chronione składowe interfejsu w połączeniu z innymi modyfikatorami zapewniają kompletny mechanizm hermetyzacji, choć trzeba przyznać, że jest on skomplikowany.

Metody rozszerzające a domyślne składowe interfejsu

Jeśli chcesz rozszerzyć opublikowany interfejs o nowe możliwości, to czy lepiej jest używać domyślnych składowych interfejsu, czy utworzyć metodę rozszerzającą lub drugi interfejs pochodny od pierwszego i zawierający dodatkowe składowe? W trakcie podejmowania takiej decyzji należy uwzględnić następujące kwestie:

- Oba podejścia umożliwiają przesyłanie składowych za pomocą implementacji metody o tej samej sygnaturze w instancji danego interfejsu.
- Metody rozszerzające można dodać spoza podzespołu zawierającego definicję danego interfejsu.
- Choć dozwolone jest tworzenie domyślnych właściwości interfejsu, niedostępne jest miejsce na dane instancji z wartością właściwości (pole nie są dozwolone). Dlatego można stosować wyłącznie właściwości obliczane.
- Choć nie można tworzyć właściwości rozszerzających, obliczenia można umieścić w metodach rozszerzających pełniących funkcję gettera (na przykład `GetData()`), dostępnych już w wersjach starszych niż .NET Core 3.0 i kolejne wydania platformy.
- Utworzenie drugiego, pochodnego interfejsu pozwala definiować właściwości i metody bez wprowadzania niezgodności wersji lub zmagania się z ograniczeniami platformy.
- Podejście z interfejsem pochodnym wymaga, aby w typach z implementacją dodać nowy interfejs i zastosować nowe mechanizmy.
- Domyślne składowe interfejsu można wywoływać tylko w interfejsie. Nawet obiekty z implementacją danego interfejsu nie mają dostępu do domyślnych składowych (chyba że zrzutujesz obiekt na typ interfejsu). Oznacza to, że domyślne składowe interfejsu działają jak jawnie zaimplementowane składowe interfejsu, chyba że w klasie bazowej znajduje się ich implementacja.
- W interfejsie można zdefiniować chronione składowe wirtualne, jednak są one dostępne tylko w interfejsach pochodnych; nie są dostępne w klasach implementujących dany interfejs.

- Domyślną składową interfejsu można przesłonić w klasie z implementacją interfejsu, dzięki czemu w każdej klasie można w razie potrzeby zdefiniować odpowiednie działania. Metody rozszerzające wymagają, aby taka metoda była dostępna w czasie komplikacji. Implementacja jest więc ustalana na etapie komplikacji, a nie w czasie wykonywania programu. Dlatego programista klasy implementującej interfejs nie może udostępnić innej implementacji metody, jeśli jest ona wywoływana z poziomu biblioteki. Na przykład metoda `System.Linq.Enumerable.Count()` udostępnia specjalną implementację dla kolekcji opartych na indeksach, rzuającą typ na implementację listy i pobierającą liczbę elementów. Jedynym sposobem na skorzystanie z bardziej wydajnego kodu jest tu zaimplementowanie interfejsu opartego na liście. Natomiast gdy używana jest domyślna implementacja interfejsu, w dowolnej klasie z jego implementacją można przesłonić daną metodę, aby udostępnić jej lepszą wersję.

Oto podsumowanie: polimorficzne działanie właściwości można zapewnić tylko za pomocą drugiego interfejsu lub domyślnych składowych interfejsu. Jeśli potrzebna jest obsługa platform starszych niż .NET Core 3.0 i chcesz dodać właściwości, lepiej jest utworzyć nowy interfejs. Gdy modyfikacje dotyczą tylko metod (a nie właściwości), preferowane są metody rozszerzające.

Wskazówka

ROZWAŻ utworzenie metod rozszerzających lub dodatkowego interfejsu zamiast domyślnych składowych interfejsu, gdy dodajesz metody do wcześniej udostępnionego interfejsu.

STOSUJ metody rozszerzające, jeśli nie masz kontroli nad działającym w sposób polimorficzny interfejsem.

UTWÓRZ dodatkowy interfejs, jeśli w polimorficznej hierarchii potrzebne są właściwości i zamierzasz zapewnić obsługę wersji starszych niż .NET Core 3.0.

Interfejsy a klasy abstrakcyjne

Interfejsy to inny rodzaj typów danych. Są one jedną z odmian typów, które nie dziedziczą po klasie `System.Object`⁸. Interfejsy, w odróżnieniu od klas, nie umożliwiają tworzenia instancji. Instancja interfejsu jest dostępna tylko za pomocą referencji do obiektu z implementacją danego interfejsu. Nie można użyć operatora `new` do interfejsu. Związane jest z tym to, że interfejsy nie mogą zawierać konstruktorów ani finalizatorów. Ponadto przed wersją C# 8.0 w interfejsach nie można było tworzyć składowych statycznych.

8.0

⁸ Inne takie kategorie to typy wskaźnikowe i typy parametrów określających typ. Jednak każdy interfejs można przekształcić na typ `System.Object`. Ponadto obiekty z implementacją dowolnego interfejsu mogą wywoływać metody klasy `System.Object`, dlatego można uznać, że ta cecha interfejsów ma znaczenie czysto teoretyczne.

Interfejsy przypominają więc klasy abstrakcyjne. Obie te struktury nie pozwalają tworzyć instancji. Porównanie interfejsów i klas abstrakcyjnych znajdziesz w tabeli 8.2. Ponieważ klasy abstrakcyjne i interfejsy mają określone wady i zalety, wyboru między jedną z tych struktur należy dokonać w wyniku analizy kosztów i korzyści na podstawie informacji z tabeli 8.2 oraz przedstawionych dalej wskazówek.

Tabela 8.2. Porównanie klas abstrakcyjnych i interfejsów

Klasy abstrakcyjne	Interfejsy
Nie można bezpośrednio tworzyć ich instancji. Trzeba utworzyć instancję klasy pochodnej.	Nie można bezpośrednio tworzyć ich instancji. Trzeba utworzyć instancję typu z implementacją interfejsu.
Klasa pochodna od abstrakcyjnej albo sama musi być abstrakcyjna, albo musi zawierać implementację wszystkich składowych abstrakcyjnych.	W typach z implementacją interfejsu trzeba zaimplementować wszystkie jego składowe.
Można dodawać nowe składowe nieabstrakcyjne, dziedziczone w klasach pochodnych. Nie powoduje to naruszenia zgodności z wcześniejszą wersją kodu.	Od wersji C# 8.0 i .NET Core 3.0 można dodawać domyślne składowe interfejsu. Są one dziedziczone we wszystkich klasach pochodnych i nie narusza to zgodności z wcześniejszą wersją kodu.
Można deklarować metody, właściwości i pola (a także wszystkie inne rodzaje składowych, w tym konstruktory i finalizatory).	Jako składowe instancji można deklarować tylko metody i właściwości, natomiast pola, konstruktory i finalizatory nie są dozwolone. Dozwolone są dowolne składowe statyczne, w tym konstruktory, zdarzenia i pola.
Można tworzyć składowe instancji, statyczne i opcjonalnie abstrakcyjne. Można udostępnić implementację składowych nieabstrakcyjnych; zostanie ona wykorzystana w klasach pochodnych.	Od wersji C# 8.0 i .NET Core 3.0 można tworzyć składowe instancji, abstrakcyjne i statyczne. Można też udostępnić implementację składowych nieabstrakcyjnych i korzystać z niej w klasach implementujących dany interfejs.
Składowe mogą być opcjonalnie zadeklarowane jako wirtualne. Składowe, których nie należy przesyłać (zobacz listing 8.13), nie powinny być wirtualne.	Wszystkie składowe bez modyfikatora sealed są wirtualne (nie trzeba tego samodzielnie deklarować). Dlatego nie istnieje sposób na to, by uniemożliwić przesyłanie operacji interfejsu.
Klasa pochodna może dziedziczyć tylko po jednej klasie bazowej.	W typie z implementacją można zaimplementować wiele interfejsów.

Wskazówki

PRZEDKŁADAJ interfejsy nad klasy abstrakcyjne, gdy chcesz uzyskać polymorfizm od wersji C# 8.0 i .NET Core 3.0; w starszych wersjach preferuj klasy abstrakcyjne.

ROZWAŻ zdefiniowanie interfejsu, jeśli chcesz dodać obsługę funkcji z interfejsu w typach, które dziedziczą już po innym typie.

Jeśli możesz używać platformy .NET Core 3.0, to wiedz, interfejsy zdefiniowane w C# 8.0 i nowszych wersjach języka oferują wszystkie możliwości klas abstrakcyjnych oprócz pól instancji. Jednak ponieważ w typie implementującym interfejs można przesłonić właściwość z tego interfejsu i umożliwić w ten sposób przechowywanie danych, interfejsy udostępniają nadzbiór możliwości klas abstrakcyjnych. Ponadto interfejsy zapewniają wielodziedziczenie i większą hermetyzację przy dostępie chronionym. Dlatego jeśli chcesz uzyskać polimorfizm, warto preferować interfejsy, ponieważ pozwala to w wersjach od C# 8.0 i .NET Core 3.0 oddzielić kontrakty (opis działania typu) od szczegółów implementacji (sposobu wykonywania operacji).

Interfejsy a atrybuty

Interfejsy, które nie zawierają żadnych składowych (ani odziedziczonych, ani zwykłych), służą czasem do reprezentowania informacji o typie. Możesz na przykład utworzyć znacznikowy interfejs `IObsolete` przeznaczony do oznaczania, że określony typ został zastąpiony przez inny typ. Zwykle uważane jest to za nadużycie możliwości, jakie dają interfejsy. Interfejsy należy stosować do określania funkcji, jakie typ może wykonywać, a nie do opisywania faktów dotyczących wybranych typów. Zamiast posługiwać się interfejsami znacznikowymi, zastosuj atrybuty. Ich szczegółowe omówienie znajdziesz w rozdziale 18.

Wskazówka

UNIKAJ stosowania interfejsów znacznikowych (niemających składowych).
Zamiast nich korzystaj z atrybutów.

Podsumowanie

Interfejsy są ważnym aspektem programowania obiektowego w języku C#. Zapewniają podobny mechanizm jak klasy abstrakcyjne, ale nie mają ograniczenia w postaci możliwości dziedziczenia tylko po jednym typie. W jednej klasie można zaimplementować wiele interfejsów. Od wersji C# 8.0 i .NET Core 3.0 interfejsy mogą zawierać implementację (w składowych domyślnych), przez co oferują prawie nadzbiór możliwości klas abstrakcyjnych — jeśli nie trzeba zachować zgodności ze starszymi wersjami języka.

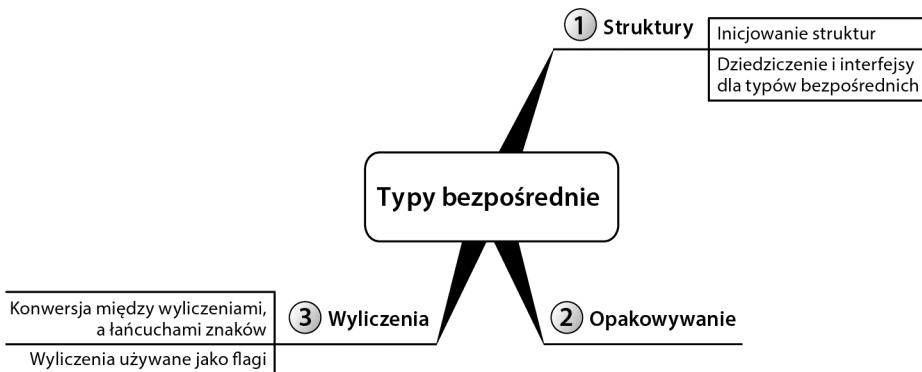
W C# interfejsy można implementować jawnie lub niejawnie. Trzeba zdecydować, czy klasa z implementacją ma udostępniać składową interfejsu bezpośrednio (implementacja niejawnna), czy tylko w wyniku konwersji typu na interfejs (implementacja jawnia). Rodzaj implementacji jest podawany na poziomie składowych. W tym samym interfejsie jedne składowe mogą być implementowane niejawnie, a inne jawnie.

W rozdziale 9. opisano typy bezpośrednie i znaczenie definiowania niestandardowych typów tego rodzaju. Wskazano też pewne subtelne problemy, jakie mogą się pojawić w związku z tymi typami.

■ 9 ■

Typy bezpośrednie

W TEJ KSIĄŻCE UŻYWAŁEŚ JUŻ TYPÓW BEZPOŚREDNICH. Takim typem jest na przykład `int`. W tym rozdziale opisano nie tylko korzystanie z typów bezpośrednich, ale też definiowanie niestandardowych typów tego rodzaju. Są dwie kategorie niestandardowych typów bezpośrednich — struktury i wyliczenia. W tym rozdziale wyjaśniono, że struktury umożliwiają programistom definiowanie nowych typów bezpośrednich działających bardzo podobnie jak większość wbudowanych typów opisanych w rozdziale 2. Ważne jest to, że nowo definiowane typy bezpośrednie mają własne niestandardowe dane i metody. W tym rozdziale omówiono, jak używać wyliczeń do definiowania zbiorów stałych wartości.



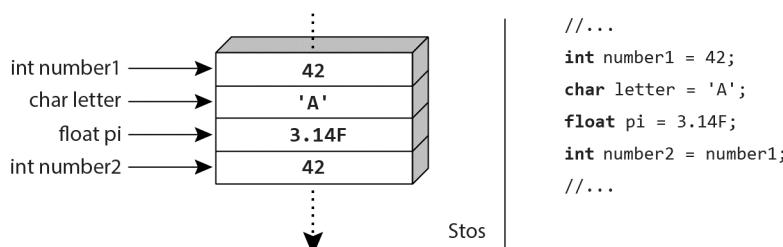
■ ZAGADNIENIE DLA POCZĄTKUJĄCYCH

Kategorie typów

Wszystkie opisane do tego miejsca typy należą do jednej z dwóch kategorii — typów referencyjnych i typów bezpośrednich. Różnice między tymi grupami dotyczą innych strategii kopiowania danych, co przekłada się na to, że typy są w inny sposób przechowywane w pamięci. W tym zagadnienniu dla początkujących przedstawiono omówienie typów bezpośrednich i referencyjnych. Będzie ono przydatne dla osób, dla których są to nowe informacje.

Typy bezpośrednie

Zmienne typów bezpośrednich przechowują wartość bezpośrednio, co pokazano na rysunku 9.1. Nazwa zmiennej jest powiązana bezpośrednio z lokalizacją wartości w pamięci. Dlatego gdy do drugiej zmiennej przypisana jest wartość pierwotnej zmiennej, wykonywana jest kopia wartości pierwotnej zmiennej. Kopia ta jest zapisywana w pamięci w lokalizacji powiązanej z drugą zmienną. Te dwie zmienne nigdy nie prowadzą do tej samej lokalizacji w pamięci (chyba że jedna lub obie zmienne to parametry out bądź ref, które z definicji są aliasami dla innych zmiennych). Zmiana wartości pierwotnej zmiennej nie wpływa na wartość drugiej zmiennej, ponieważ każda zmienna jest powiązana z inną lokalizacją w pamięci. Tak więc modyfikacja wartości jednej zmiennej typu bezpośredniego nie prowadzi do zmiany wartości innych takich zmiennych.



Rysunek 9.1. Typy bezpośrednie przechowują dane bezpośrednio

Zmienna typu bezpośredniego jest jak kartka papieru z zapisaną na niej liczbą. Jeśli chcesz zmienić tę liczbę, możesz ją wymazać i zastąpić inną. Jeżeli masz drugą kartkę papieru, możesz skopiować liczbę z pierwszej kartki, jednak obie kartki papieru są niezależne od siebie. Wymazanie i zastąpienie liczby na jednej z nich nie prowadzi do zmiany na drugiej.

Podobnie przekazanie instancji typu bezpośredniego do metody (takiej jak `Console.WriteLine()`) skutkuje utworzeniem kopii danych. Wartość z lokalizacji powiązanej z argumentem jest kopiwana do lokalizacji powiązanej z parametrem. Wszelkie zmiany parametru w metodzie nie prowadzą do modyfikacji pierwotnej wartości w jednostce wywołującej. Ponieważ typy bezpośrednie wymagają tworzenia kopii w pamięci, zwykle powinny zajmować niewielką jej ilość (przeważnie jest to 16 bajtów lub mniej).

Wskazówka

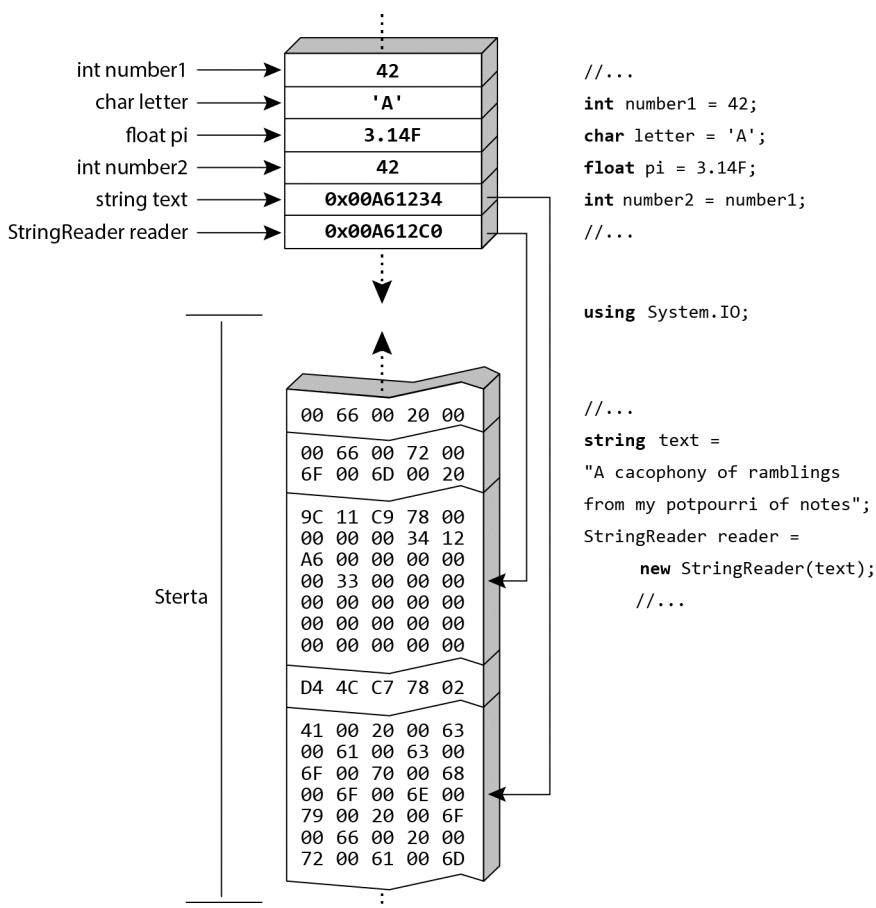
NIE twórz typów bezpośrednich zajmujących więcej niż 16 bajtów pamięci.

Wartości typów bezpośrednich często są używane przez krótki czas. W wielu sytuacjach wartość jest potrzebna tylko we fragmencie wyrażenia lub w celu aktywowania metody. W takich scenariuszach zmienne i wartości tymczasowe typów bezpośrednich często można przechowywać w **puli pamięci tymczasowej** nazywanej *stosem*. Jest to myląca nazwa, ponieważ nie ma wymogu, by pula tymczasowa korzystała z pamięci stosu. Jest to szczegół zależny od implementacji. Często pula tymczasowa przydziela pamięć w dostępnych rejestrach.

Odzyskiwanie pamięci z puli tymczasowej jest mniej kosztowne niż ze sterty. Jednak typy bezpośrednie są zwykle kopiowane częściej niż typy referencyjne, a samo kopiowanie może negatywnie wpływać na wydajność. Dlatego nie zakładaj błędnie, że „typy bezpośrednie są szybsze, ponieważ pamięć dla nich można alokować na stosie”.

Typy referencyjne

Wartość zmiennej typu referencyjnego to referencja do instancji (zobacz rysunek 9.2). Zmienne typu referencyjnego przechowują referencję (zwykle w postaci adresu do lokalizacji w pamięci), określającą, gdzie znajdują się dane instancji. W takich typach (inaczej niż w typach bezpośrednich) dane nie są więc zapisywane bezpośrednio. Dlatego aby uzyskać dostęp do danych, środowisko uruchomieniowe musi wczytać referencję ze zmiennej, a następnie przeprowadzić dereferencję i w ten sposób dotrzeć do lokalizacji w pamięci, gdzie znajdują się dane określonej instancji.



Rysunek 9.2. Typy referencyjne prowadzą do sterty

Tak więc zmienna typu referencyjnego jest powiązana z dwoma lokalizacjami — z miejscem bezpośrednio powiązanym ze zmienną i z lokalizacją wskazywaną przez referencję (jest to wartość zapisana w zmiennej).

Zmienna typu referencyjnego też jest jak kartka papieru, na której jest coś zapisane. Wyobraź sobie kartkę papieru z adresem, na przykład „Kraków, ul. Sezamkowa 123”. Ta kartka to zmienna, a adres to referencja do budynku. Ani kartka, ani zapisany na niej adres nie są budynkiem. Ponadto lokalizacja kartki nie musi mieć nic wspólnego z lokalizacją docelowego budynku. Jeśli zapiszesz kopię referencji na innej kartce papieru, adresy z obu kartek będą prowadziły do tego samego budynku. Jeśli przemalujesz go na zielono, budynek wskazywany na obu kartkach stanie się zielony, ponieważ obie referencje prowadzą do tego samego obiektu.

Lokalizacja bezpośrednio powiązana ze zmienną (lokalizacja wartości pośredniej) jest tu traktowana tak samo jak lokalizacja powiązana ze zmienną typu bezpośredniego. Jeśli wiadomo, że zmienna będzie używana przez krótki czas, alokowana pamięć pochodzi z puli pamięci krótkoterminowej. Wartość zmiennej typu referencyjnego to zawsze albo referencja do lokalizacji w pamięci ze sterty (obsługiwanej przez mechanizm odzyskiwania pamięci), albo null.

W porównaniu do zmiennych typu bezpośredniego, które bezpośrednio przechowują dane w instancji, dostęp do danych powiązanych z referencją wymaga dodatkowego „skoku”. Najpierw trzeba przeprowadzić dereferencję, by ustalić lokalizację docelowych danych, a dopiero potem można wczytać lub zapisać same dane. Kopiowanie wartości typu referencyjnego polega na skopiowaniu samej referencji. Referencja zajmuje niewiele miejsca. Wiadomo, że nie jest ona większa niż liczba bitów słowa procesora. W komputerach 32-bitowych używane są referencje 4-bajtowe, w komputerach 64-bitowych referencje mają po 8 bajtów itd. Kopiowanie wartości typu bezpośredniego polega na skopiowaniu wszystkich danych. Mogą one zajmować dużo miejsca. Dlatego w niektórych sytuacjach kopowanie typów referencyjnych okazuje się bardziej wydajne. To dlatego zgodnie z wytycznymi nie należy tworzyć typów bezpośrednich zajmujących więcej niż 16 bajtów. Jeśli kopowanie typu bezpośredniego jest więcej niż cztery razy bardziej kosztowne od kopowania referencji, prawdopodobnie lepiej będzie zastosować typ referencyjny.

Ponieważ typy referencyjne wymagają kopowania samej referencji do danych, dwie różne zmienne mogą prowadzić do tych samych danych. W takiej sytuacji modyfikacja danych za pomocą jednej zmiennej będzie widoczna także w drugiej zmiennej. Dotyczy to zarówno przypisań, jak i wywołań metod.

Wróćmy do przedstawionej wcześniej analogii. Jeśli przekażesz adres budynku do metody, utworzysz kopię kartki z referencją i przekażesz ją metodzie. Metoda nie może zmienić zawartości pierwotnej kartki w taki sposób, by prowadziła do innego budynku. Jeśli jednak metoda pomaluje docelowy budynek, to po zwróceniu przez nią sterowania w jednostce wywołujączej okaże się, że ten sam budynek ma teraz inny kolor.

Struktury

Prawie wszystkie wbudowane typy języka C#, na przykład `bool` i `decimal`, to typy bezpośrednie. Wyjątkami są typy referencyjne `string` i `object`. W platformie dostępnych jest też wiele dodatkowych typów bezpośrednich. Ponadto programiści mogą definiować własne typy tego rodzaju.

Aby zdefiniować niestandardowy typ bezpośredni, należy zastosować składnię podobną do tej używanej do definiowania klas i interfejsów. Najważniejszą różnicą jest to, że w definicji typu bezpośredniego należy podać słowo kluczowe `struct`, co pokazano na listingu 9.1. Na tym listingu znajduje się typ bezpośredni służący do precyzyjnego reprezentowania kąta w stopniach, minutach i sekundach. *Minuta* to jedna sześćdziesiąta stopnia, a *sekunda* to jedna sześćdziesiąta minuty. Ten system jest używany w nawigacji, ponieważ ma tę wygodną cechę, że na równiku łuk odpowiadający minucie ma długość równą jednej mili morskiej.

Listing 9.1. Deklarowanie struktury

```
// Do deklarowania typów bezpośrednich służy słowo kluczowe struct.
struct Angle
{
    public Angle(int degrees, int minutes, int seconds)
    {
        Degrees = degrees;
        Minutes = minutes;
        Seconds = seconds;
    }

    // Używanie przeznaczonych tylko do odczytu i automatycznie implementowanych
    // właściwości z języka C# 6.0.
    public int Degrees { get; }
    public int Minutes { get; }
    public int Seconds { get; }

    public Angle Move(int degrees, int minutes, int seconds)
    {
        return new Angle(
            Degrees + degrees,
            Minutes + minutes,
            Seconds + seconds);
    }
}

// Deklaracja klasy, czyli typu referencyjnego.
// Użycie w deklaracji słowa kluczowego struct spowodowałoby
// powstanie typu bezpośredniego zajmującego więcej niż 16 bajtów.
class Coordinate
{
    public Angle Longitude { get; set; }

    public Angle Latitude { get; set; }
}
```

Na tym listingu `Angle` to typ bezpośredni przechowujący stopnie, minuty i sekundy kąta (długości i szerokości geograficznej). Wynikowy typ języka C# to **struktura**.

Zauważ, że struktura `Angle` na listingu 9.1 jest niezmienna, ponieważ wszystkie właściwości są zadeklarowane jako przeznaczone tylko do odczytu automatycznie implementowane właściwości wprowadzone w języku C# 6.0. Aby utworzyć właściwość tylko do odczytu w starszych wersjach języka, programiści muszą zadeklarować właściwość z samym getterem, który pobiera dane z pola opatrzonego modyfikatorem `readonly` (zobacz listingu 9.3). Język C# 6.0 pozwala znacznie skrócić kod potrzebny do definiowania niezmiennych typów.

Początek
7.2

Od wersji C# 7.2 można zapewnić, że zdefiniowana zostanie struktura przeznaczona tylko do odczytu. Służy do tego następująca deklaracja:

```
readonly struct Angle { }
```

Koniec
7.2

Teraz kompilator będzie zapewniał niemodyfikowalność całej struktury i zgłaszał błąd, jeśli istnieje pole nieprzeznaczone tylko do odczytu lub właściwość z setterem.

Początek
8.0
6.0

Jeśli potrzebujesz bardziej precyzyjnej kontroli niż deklaracja całej klasy jako przeznaczonej tylko do odczytu, to w C# 8.0 możesz zdefiniować dowolną składową struktury jako tylko do odczytu (w tym metody, a nawet gettery, które potencjalnie mogą zmieniać stan obiektu, choć nie powinny tego robić). Na listingu 9.1 można dodać modyfikator `readonly` do metody `Move()`:

```
readonly public Angle Move(int degrees, int minutes, int seconds) { ... }
```

Jest to dopuszczalne, choć zbędne, gdy cała klasa jest przeznaczona tylko do odczytu.

Koniec
8.0

Jeśli składowa tylko do odczytu modyfikuje dane struktury (właściwości lub pola) albo wywołuje składową umożliwiającą zapis, zgłoszony zostanie błąd kompilacji. Zgodnie z modelem dostępu do składowych w trybie tylko do odczytu programiści deklarują swoje zamiary, określając, czy składowa może modyfikować dany obiekt. Zauważ, że jeśli chcesz zastosować modyfikator `readonly` tylko do gettera lub tylko do settera, musisz użyć właściwości, które nie są automatycznie implementowane. W celu dodania tego modyfikatora zarówno do gettera, jak i do settera umieść go przy samej właściwości, a nie przy getterze lub setterze.

Uwaga

Choć nic w języku tego nie wymusza, zgodnie z wartościową wskazówką typy bezpośrednie powinny być niezmienne. Po utworzeniu instancji typu bezpośredniego nie powinno być możliwe modyfikowanie wartości tej samej instancji. W sytuacjach, w których zmiana wartości jest potrzebna, należy utworzyć nową instancję. Na listingu 9.1 dostępna jest metoda `Move()`, która nie modyfikuje instancji typu `Angle`, ale zwraca zupełnie nową instancję.

Ta wytyczna wynika z dwóch dobrych powodów. Po pierwsze, typy bezpośrednie powinny reprezentować wartości. Gdy dodajesz dwie liczby całkowite, nie traktujesz tej operacji jak zmiany którejś z tych wartości. Obie liczby są niezmienne, a w wyniku dają trzecią liczbę.

Po drugie, ponieważ typy bezpośrednie są kopiowane przez wartość, a nie przez referencję, bardzo łatwo można się pomylić i błędnie przyjąć, że modyfikacja jednej zmiennej typu bezpośredniego spowoduje też zmianę innej takiej zmiennej (jak ma to miejsce w przypadku typów referencyjnych).

Wskazówka

TWÓRZ niezmienne typy bezpośrednie.

Zaskakujące jest to, że krotki (typ `System.ValueTuple`) są niezgodne z tą wskazówką. Aby dowiedzieć się, z czego wynika ten wyjątek, odwiedź stronę <https://IntelliTect.com/WhyTupleBreaksTheImmutableRules>.

Inicjowanie struktur

Oprócz właściwości i pól struktury mogą obejmować metody i konstruktory. Dla struktur nie można jednak zdefiniować konstruktora domyślnego. Gdy konstruktor domyślny nie jest podany, kompilator języka C# tworzy go automatycznie. Wygenerowany tak konstruktor inicjuje wszystkie pola wartościami domyślnymi. Dla pól typów referencyjnych jest to wartość `null`, dla pól typów liczbowych — wartość zero, dla pól typu logicznego — wartość `false` itd.

Aby zagwarantować, że lokalna zmienna typu referencyjnego zostanie w pełni zainicjowana przez konstruktor, w każdym konstruktorze struktury trzeba inicjować wszystkie pola (oraz automatycznie implementowane właściwości tylko do odczytu). W C# 6.0 wystarczy zainicjować automatycznie implementowaną właściwość tylko do odczytu, ponieważ wiązane z nią pole nie jest dostępne i nie da się go zainicjować. Jeśli nie zainicujesz wszystkich pól struktury, wystąpi błąd komplikacji. Pewną komplikacją jest to, że C# nie umożliwia stosowania inicjatorów pól w strukturze. Na przykład kod z listingu 9.2 się nie skompiluje, jeśli instrukcja `_Degrees = 42` znajdzie się poza komentarzem.

Listing 9.2. Inicjowanie pola struktury w deklaracji prowadzi do błędu

```
struct Angle
{
    // ...
    // BŁĄD: pole nie można inicjować w ich deklaracji.
    // int _Degrees = 42;
    // ...
}
```

Jeśli programista nie utworzy jawnie instancji za pomocą wywołania operatora `new` i konstruktora, wszystkie dane struktury zostaną niejawnie zainicjowane wartościami domyślnymi. Jednocześnie aby uniknąć błędu komplikacji, trzeba jawnie zainicjować wszystkie dane typu bezpośredniego. Powstaje przy tym pytanie — kiedy możliwa jest niejawnia inicjacja typu bezpośredniego bez jawnego tworzenia instancji? Taka sytuacja może mieć miejsce w trakcie tworzenia instancji typu referencyjnego, która zawiera pole typu bezpośredniego bez przypisanej wartości, a także w czasie tworzenia instancji tablicy wartości typu bezpośredniego, gdy nie jest używany inicjator tablicy.

W celu zastosowania się do wymogu inicjowania struktury trzeba zainicjować wszystkie jawnie zadeklarowane pola. Inicjowanie musi przebiegać bezpośrednio. Gdyby na listingu 9.3 konstruktor inicjalizował właściwość (tak jak w kodzie opatrzonym komentarzem), a nie pole, wystąpiłby błąd komilacji.

Listing 9.3. Dostęp do właściwości przed zainicjowaniem wszystkich pól

```
struct Angle
{
    // BŁĄD: obiektu 'this' nie można użyć przed
    // przypisaniem wartości do wszystkich jego pól.
    // public Angle(int degrees, int minutes, int seconds)
    // {
    //     Degrees = degrees; // To skrócona postać zapisu this.Degrees = ...;
    //     Minutes = minutes;
    //     Seconds = seconds;
    // }

    public Angle(int degrees, int minutes, int seconds)
    {
        _Degrees = degrees;
        _Minutes = minutes;
        _Seconds = seconds;
    }

    public int Degrees { get { return _Degrees; } }
    readonly private int _Degrees;

    public int Minutes { get { return _Minutes; } }
    readonly private int _Minutes;

    public int Seconds { get { return _Seconds; } }
    readonly private int _Seconds;

    // ...
}
```

Dostęp do obiektu `this` nie jest dozwolony do czasu stwierdzenia przez komplator, że wszystkie pola zostały zainicjowane. Wywołanie `Degrees` jest niejawnie przekształcane na postać `this.Degrees`. Aby rozwiązać problem, należy bezpośrednio zainicjować pola, tak jak w znajdującym się poza komentarzem kodzie konstruktora na listingu 9.3.

Z powodu wymogu inicjowania pól struktury, związki automatycznie implementowanej właściwości tylko do odczytu dodanych w C# 6.0 i wytycznej zalecającej unikanie dostępu do pól poza powiązaną z nimi właściwością od wersji C# 6.0 w strukturach należy przedkładać takie właściwości nad pola.

Wskazówka

ZAPEWNIAJ prawidłową wartość domyślną struktury. Zawsze możliwe jest uzyskanie struktury z domyślnymi wartościami w postaci samych zer.

ZAGADNIENIE DLA ZAAWANSOWANYCH

Używanie operatora new do typów bezpośrednich

Wywołanie operatora new dla typu referencyjnego powoduje, że środowisko uruchomieniowe tworzy nową instancję na stercie z obsługą odzyskiwania pamięci, inicjuje wszystkie pola wartościami domyślnymi i wywołuje konstruktor, do którego przekazuje referencję do instancji jako obiekt this. Efekt to referencja do instancji, którą można następnie skopiować do docelowej lokalizacji. Wywołanie operatora new dla typu bezpośredniego sprawia, że środowisko uruchomieniowe tworzy nową instancję w puli pamięci tymczasowej, inicjuje wszystkie pola wartościami domyślnymi i wywołuje konstruktor (jako obiekt this przekazując lokalizację w pamięci tymczasowej za pomocą zmiennej typu ref). W wyniku tego w pamięci tymczasowej jest zapisywana wartość, którą można potem skopiować do docelowej lokalizacji.

Struktury, w odróżnieniu od klas, nie mogą zawierać finalizatorów. Struktury są kopowane przez wartość. Nie można dla nich określić „identyczności referencji”, co jest możliwe dla typów referencyjnych. Dlatego trudno ustalić, kiedy można bezpiecznie wywołać finalizator i zwolnić niezarządzane zasoby zajmowane przez strukturę. Mechanizm odzyskiwania pamięci wie, kiedy nie występują już aktywne referencje do instancji typu referencyjnego; dlatego może uruchomić finalizator instancji w dowolnym momencie, w którym nie ma już żadnych aktywnych referencji. Jednak żadna część środowiska uruchomieniowego nie śledzi, ile kopii typu bezpośredniego istnieje w danym momencie.

Porównanie języków — w C++ słowo struct definiuje typ ze składowymi publicznymi

W C++ różnica między typami zadeklarowanymi za pomocą słów kluczowych `struct` i `class` dotyczy tego, czy domyślnie składowe są publiczne, czy prywatne. W języku C# różnica w działaniu tych słów kluczowych jest znacznie większa, ponieważ polega na tym, czy instancje danego typu są kopowane przez wartość, czy przez referencję.

Dziedziczenie i interfejsy a typy bezpośrednie

Wszystkie typy bezpośrednie są automatycznie zamknięte. Ponadto wszystkie typy bezpośrednie z wyjątkiem typów wyliczeniowych są pochodne od typu `System.ValueType`. Dlatego łańcuch dziedziczenia dla struktur zawsze prowadzi od typu `object` do typu `System.ValueType` i do struktury.

W typach bezpośrednich można implementować interfejsy. Wiele wbudowanych typów platformy .NET zawiera implementację interfejsów takich jak `IComparable` i `IFormattable`.

Typ `System.ValueType` zapewnia, że pochodne od niego typy będą działały jak typy bezpośrednie, ale nie obejmuje żadnych dodatkowych składowych. W tym typie modyfikacje dotyczą głównie przesłonięcia wszystkich składowych wirtualnych z typu `object`. Reguły przesyłania metod typu bazowego w strukturze są prawie takie same jak w klasach (zobacz rozdział 10.). Jedna z różnic polega na tym, że w typach bezpośrednich domyślna implementacja metody `GetHashCode()` przekazuje wywołanie do pierwszego różnego od `null` pola struktury.

Ponadto w metodzie `Equals()` wykorzystywany jest mechanizm refleksji. Dlatego jeśli dany typ bezpośredni jest często używany w kolekcjach (dotyczy to przede wszystkim słowników, gdzie używane są skróty), należy w nim przesłonić metody `Equals()` i `GetHashCode()`, by zapewnić wysoką wydajność kodu. Więcej informacji na ten temat zawiera rozdział 10.

Wskazówka

PRZECIĄŻAJ operatory równości (`Equals()`, `==` i `!=`) w typach bezpośrednich, jeśli sprawdzanie równości jest istotne. Rozważ też zaimplementowanie interfejsu `IEquatable<T>`.

Opakowywanie

Wiesz już, że zmienne typu bezpośredniego zawierają bezpośrednio dane, natomiast zmienne typu referencyjnego przechowują referencję do innej lokalizacji w pamięci. Co się jednak dzieje, gdy wartość typu bezpośredniego jest przekształcana na jeden z zaimplementowanych interfejsów lub na nadrzedną klasę bazową `object`? Wynikiem takiej konwersji musi być referencja do lokalizacji w pamięci zawierającej coś, co wygląda jak instancja typu referencyjnego, jednak w rzeczywistości jest zmienną z wartością typu bezpośredniego. Taka konwersja, nazywana **opakowywaniem** (ang. *boxing*), przebiega w specjalny sposób. Konwersja zmiennej typu bezpośredniego, która bezpośrednio zawiera dane, na typ referencyjny prowadzący do lokalizacji na stercie z obsługą odzyskiwania pamięci odbywa się w kilku krokach.

1. Pamięć jest alokowana na stercie, na której znajdują się dane typu bezpośredniego oraz inne elementy (indeks `SyncBlockIndex` i wskaźnik do tablicy metod) niezbędne, by obiekt wyglądał tak jak każda inna zarządzana instancja typu referencyjnego.
2. Wartość typu bezpośredniego jest kopiwana z bieżącej lokalizacji w zaalokowane właśnie miejsce na stercie.
3. Wynikiem konwersji jest referencja do nowej lokalizacji na stercie.

Odwrotną operacją jest **wypakowywanie** (ang. *unboxing*). W jej ramach sprawdzane jest, czy typ opakowanej wartości jest zgodny z typem docelowym, po czym następuje skopiowanie wartości zapisanej na stercie.

Opakowywanie i wypakowywanie są istotne, ponieważ proces ten wpływa na wydajność i działanie kodu. Programista może nauczyć się wykrywać takie konwersje w kodzie w języku C#, a także zliczać w kodzie CIL instrukcje opakowywania (`box`) i wypakowywania (`unbox`) z danego fragmentu programu. Dla obu tych operacji występują specyficzne instrukcje, przedstawione w tabeli 9.1.

Jeśli opakowywanie i wypakowywanie są rzadkie, wpływ tych operacji na wydajność jest niewielki. Jednak opakowywanie może zachodzić w nieoczekiwanych sytuacjach, a jeśli dzieje się to często, może to mieć istotny wpływ na wydajność. Przyjrzyj się listingowi 9.4 i danym wyjściowym 9.1. Typ `ArrayList` przechowuje listę referencji do obiektów. Dlatego dodanie liczby całkowitej lub zmiennoprzecinkowej do listy powoduje opakowanie wartości, co pozwala uzyskać referencję.

Tabela 9.1. Kod CIL z instrukcjami opakowywania

Kod w języku C#	Kod CIL
<pre>static void Main() { int number; object thing; number = 42; // Opakowywanie. thing = number; // Wypakowywanie. number = (int)thing; return; }</pre>	<pre>.method private hidebysig static void Main() cil managed { .entrypoint // Code size 21 (0x15) .maxstack 1 .locals init ([0] int32 number, [1] object thing) IL_0000: nop IL_0001: ldc.i4.s 42 IL_0003: stloc.0 IL_0004: ldloc.0 IL_0005: box [mscorlib]System.Int32 IL_000a: stloc.1 IL_000b: ldloc.1 IL_000c: unbox.any [mscorlib]System.Int32 IL_0011: stloc.0 IL_0012: br.s IL_0014 IL_0014: ret } } // end of method Program::Main</pre>

Listing 9.4. Trudne do wykrycia instrukcje opakowywania i wypakowywania

```
class DisplayFibonacci
{
    static void Main()
    {

        int totalCount;
        // Celowo używany jest typ ArrayList, aby zilustrować opakowywanie
        System.Collections.ArrayList list =
            new System.Collections.ArrayList();

        Console.WriteLine("Wprowadź liczbę z przedziału od 2 do 1000:");
        totalCount = int.Parse(Console.ReadLine());

        if (totalCount == 7) // „Magiczna” liczba używana w testach
        {
            // Błąd czasu wykonania:
            // list.Add(0); // Potrzebne jest rzutowanie na typ double lub przyrostek 'D'.
            // Niezależnie od wybranego rozwiązania
            // kod CIL jest identyczny.
        }
        else
        {
            list.Add((double)0);
        }

        list.Add((double)0);
        list.Add((double)1);
        for (int count = 2; count < totalCount; count++)
        {
            list.Add(

```

```

        (double)list[count - 1]! +
        (double)list[count - 2]! );
    }

    // Używanie pętli foreach zamiast instrukcji Console.WriteLine(string.Join(", ", list.ToArray()));
    // aby uwidoczyć operacje opakowywania i wypakowywania.
    foreach (double count in list)
    {
        Console.Write("{0}, ", count);
    }
}

```

DANE WYJŚCIOWE 9.1.

Wprowadź liczbę z przedziału od 2 do 1000:**42**
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597,
↳ 2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393, 196418,
↳ 317811, 514229, 832040, 1346269, 2178309, 3524578, 5702887, 9227465,
↳ 14930352, 24157817, 39088169, 63245986, 102334155, 165580141,

Kompilacja kodu z listingu 9.4 sprawi, że w kodzie CIL znajdzie się pięć instrukcji opakowywania i trzy instrukcje wypakowywania.

- Pierwsze dwie instrukcje opakowywania związane są z początkowymi wywołaniami `list.Add()`. Sygnatura tej metody z klasy `ArrayList`¹ to `int Add(object value)`. Dlatego każda wartość typu bezpośredniego przekazana do tej metody zostaje opakowana.
- Dalej następują dwie instrukcje wypakowywania w wywołaniach `Add()` w pętli `for`. Wartość zwracana za pomocą operatora indeksu kolekcji `ArrayList` zawsze ma typ `object`, ponieważ to tego typu wartości zawiera ta kolekcja. Aby dodać dwie wartości, trzeba zrzutować je z powrotem na typ `double`. Rzutowanie z referencji do obiektu na typ bezpośredni odbywa się za pomocą wypakowania wartości.
- Następnie wynik dodawania jest zapisywany w instancji kolekcji `ArrayList`, co znów wymaga opakowania wartości. Zauważ, że dwie pierwsze instrukcje wypakowywania i ta instrukcja opakowywania zachodzą w pętli.
- W pętli `foreach` kod przypisuje każdy kolejny element z kolekcji `ArrayList` do zmiennej `count`. Wcześniej dowiedziałeś się już, że elementy z tej kolekcji to referencje do obiektów typu `object`. Dlatego przypisanie ich do zmiennej typu `double` wymaga wypakowania każdego z nich.
- Sygnatura metody `Console.WriteLine()` wywoływanej w pętli `foreach` to `void Console.WriteLine(string format, object arg)`. Dlatego w każdym wywołaniu tej metody wartość typu `double` jest opakowywana w typ `object`.

Każda operacja opakowywania wymaga alokacji pamięci i skopiowania danych. Każda operacja wypakowywania obejmuje sprawdzenie typu i kopowanie danych. Wykonanie całego zadania za pomocą wypakowanego typu pozwala wyeliminować alokację pamięci

¹ Istotne jest, że w kolekcji używany jest tu typ `object`; nie jest to — w odróżnieniu od kolekcji generalnych — kolekcja ze ścisłe określonym typem (zobacz rozdział 12.).

i sprawdzanie typu. W przedstawionym kodzie można łatwo poprawić wydajność kodu przez pozbycie się wielu operacji opakowywania. Ten efekt można uzyskać na przykład dzięki zastosowaniu typu `object` zamiast typu `double` w ostatniej pętli `foreach`. Można też zmienić typ danych kolekcji `ArrayList` na kolekcję generyczną (zobacz rozdział 12.). Warto jednak zauważyc, że czasem trudno wykryć operacje opakowywania. Dlatego programiści powinni zwracać na tę kwestię szczególną uwagę i starać się dostrzegać sytuacje, w których opakowywanie może zachodzić wielokrotnie i wpływać na wydajność.

Inny problem związany z opakowywaniem także dotyczy czasu wykonania. Jeśli chcesz zmienić dwa pierwsze wywołania metody `Add()`, aby wyeliminować w nich rzutowanie (bez konieczności podawania literala typu `double`), musisz wstawić liczby całkowite do kolekcji `ArrayList`. Ponieważ wartości typu `int` są niejawnie przekształcane na typ `double`, na pozór ta zmiana będzie nieszkodliwa. Jednak rzutowanie na typ `double` w pętli `for` i przypisania do zmiennej `count` w pętli `foreach` zakończą się wtedy niepowodzeniem. Wynika to z tego, że bezpośrednio po operacji wypakowania następuje próba skopiowania pamięci z wartością opakowanego obiektu `int` do obiektu typu `double`. Nie można tego zrobić bez wcześniejszego rzutowania wartości na typ `int`. Jeśli tego nie zrobisz, kod w czasie wykonywania programu zgłosi wyjątek `InvalidCastException`. Na listingu 9.5 podobny błąd jest umieszczony w komentarzu, pod którym znajduje się prawidłowe rzutowanie.

Listing 9.5. Wypakowywanie musi się odbywać do używanego typu

```
// ...
int number;
object thing;
double bigNumber;

number = 42;
thing = number;
// BŁĄD: InvalidCastException
// bigNumber = (double)thing;
bigNumber = (double)(int)thing;
// ...
```

ZAGADNIENIE DLA ZAAWANSOWANYCH

Typy bezpośrednie w instrukcji lock

Język C# udostępnia instrukcję `lock` służącą do synchronizowania kodu. Ta instrukcja po komplikacji jest przekształcana na wywołania metod `Enter()` i `Exit()` z klasy `System.Threading.Monitor`. Te dwie metody trzeba wywoływać w parach. Metoda `Enter()` zapisuje przekazany argument w postaci unikatowej referencji, tak by po wywołaniu metody `Exit()` dla tej samej referencji można było zwolnić blokadę. Gdy używane są typy bezpośrednie, pojawia się problem wynikający z opakowywania — przy każdym wywołaniu metody `Enter()` lub `Exit()` na stercie tworzona jest nowa wartość. Porównanie referencji do jednej kopii z referencją do innej kopii zawsze zwraca wartość `false`, dlatego nie można powiązać wywołania metody `Enter()` z odpowiednim wywołaniem metody `Exit()`. To powoduje, że nie można stosować typów bezpośrednich w instrukcjach `lock()`.

Na listingu 9.6 przedstawiono kilka innych osobliwości dotyczących opakowywania występujących w czasie wykonywania programu. Wynik działania kodu znajdziesz w danych wyjściowych 9.2.

Listing 9.6. Osobliwości związane z opakowywaniem

```

interface IAngle
{
    void MoveTo(int degrees, int minutes, int seconds);
}

struct Angle : IAngle
{
    // ...

    // UWAGA: ten kod sprawia, że typ Angle staje się modyfikowalny.
    // Jest to niezgodne ze wskazówkami.
    public void MoveTo(int degrees, int minutes, int seconds)
    {
        _Degrees = degrees;
        _Minutes = minutes;
        _Seconds = seconds;
    }
}

class Program
{
    static void Main()
    {
        // ...

        Angle angle = new Angle(25, 58, 23);
        // Przykład 1. Prosta operacja opakowywania.
        object objectAngle = angle; // Opakowywanie.
        Console.WriteLine( ((Angle)objectAngle).Degrees);

        // Przykład 2. Wypakowywanie wartości, modyfikowanie wypakowanej wartości i usuwanie jej.
        ((Angle)objectAngle).MoveTo(26, 58, 23);
        Console.WriteLine(", " + ((Angle)objectAngle).Degrees);

        // Przykład 3. Opakowywanie wartości, modyfikowanie opakowanej wartości i usuwanie referencji do niej.
        ((IAngle)angle).MoveTo(26, 58, 23);
        Console.WriteLine(", " + ((Angle)angle).Degrees);

        // Przykład 4. Bezpośrednie modyfikowanie opakowanej wartości.
        ((IAngle)objectAngle).MoveTo(26, 58, 23);
        Console.WriteLine(", " + ((Angle)objectAngle).Degrees);

        // ...
    }
}

```

DANE WYJŚCIOWE 9.2.

25, 25, 25, 26

Na listingu 9.6 używane są struktura Angle i interfejs IAngle. Zauważ też, że metoda IAngle.MoveTo() powoduje, iż typ Angle staje się modyfikowalny. Ta zmiana ujawnia niektóre osobliwości zmiennych typów bezpośrednich i dowodzi znaczenia wytycznej, zgodnie z którą struktury powinny być niemodyfikowalne.

Przykład 1. z programu z listingu 9.6 inicjuje zmienną angle, a następnie opakowuje ją w zmienną objectAngle. Dalej następuje wywołanie metody MoveTo(), które zmienia wartość pola _Degrees na 26. Jednak, jak widać w danych wyjściowych, nie skutkuje to rzeczywistą zmianą wartości. Problem wynika z tego, że w celu wywołania metody MoveTo() kompilator wypakowuje wartość ze zmiennej objectAngle i (zgodnie z definicją tego procesu) tworzy kopię tej wartości. Typy bezpośrednie są kopiowane przez wartość. Choć skopiowana wartość jest z powodzeniem modyfikowana w trakcie wykonywania programu, kopia zostaje usunięta, a na stercie w lokalizacji wskazywanej przez zmienną objectAngle nie zachodzą żadne zmiany.

Przypomnij sobie analogię, zgodnie z którą zmienne typów bezpośrednich są jak kartki papieru z zapisaną wartością. Gdy opakowujesz wartość, powstaje umieszczana w pudełku kserokopia kartki. Gdy wypakowujesz wartość, tworzysz kserokopię kartki z tego pudełka. Modyfikacja innej kopii nie powoduje zmian kartki z pudełka.

W przykładzie 3. występuje podobny problem, choć w kontekście odwrotnej operacji. Tu kod nie wywołuje metody MoveTo() bezpośrednio, ale najpierw rzutuje wartość na typ IAngle. Konwersja na interfejs powoduje opakowanie wartości, dlatego środowisko uruchomieniowe kopiuje dane ze zmiennej angle na stertę i udostępnia referencję do opakowania. Następnie wywołanie metody modyfikuje wartość w tym opakowaniu. Jednak wartość zapisana w zmiennej angle pozostaje niezmieniona.

W ostatnim fragmencie rzutowanie na interfejs IAngle polega na konwersji referencji, a nie na konwersji związanej z opakowywaniem. Wartość została już wcześniej opakowana w wyniku konwersji na typ object, dlatego teraz nie trzeba kopować danych. Wywołanie MoveTo() zmienia wartość pola _Degrees zapisaną w opakowaniu, a kod działa zgodnie z oczekiwaniemi.

Na tym przykładzie widać, że modyfikowalne typy bezpośrednie są kłopotliwe, ponieważ często nie wiadomo, czy modyfikowana jest kopia wartości, czy lokalizacja w pamięci, na której zmienieniu zależy programiście. Unikanie zmiennych typów bezpośrednich pozwala wyeliminować tego rodzaju niejasności.

Wskazówka

UNIKAJ modyfikowalnych typów bezpośrednich.

ZAGADNIENIE DLA ZAAWANSOWANYCH

Jak uniknąć opakowywania w wywołaniach metod?

Gdy metoda jest wywoływana dla odbiorcy typu bezpośredniego, odbiorca (reprezentowany w ciele tej metody za pomocą słowa this) musi być zmienną, a nie wartością, ponieważ metoda może próbować zmodyfikować odbiorcę. Oczywiście metoda powinna modyfikować lokalizację odbiorcy w pamięci; nie powinna jednak modyfikować kopii wartości odbiorcy,

która później zostaje usunięta. Drugi i czwarty przykład z listingu 9.6 pokazują, jak wpływa to na wydajność działania metod dla opakowanych wartości typu bezpośredniego.

W drugim przykładzie operacja konwersji związana z wypakowywaniem zwraca opakowaną wartość, a nie referencję do lokalizacji na stercie zawierającej opakowaną kopię wartości. Która lokalizacja jest więc przekazywana jako obiekt `this` do wywołania metody zmieniającej wartość? Nie może to być lokalizacja opakowanej wartości ze sterty, ponieważ konwersja w trakcie wypakowywania polega na utworzeniu kopii wartości, a nie referencji do wspomnianej lokalizacji.

W takiej sytuacji (gdy potrzebna jest zmienna typu bezpośredniego, ale dostępna jest tylko wartość) zachodzi jeden z dwóch możliwych procesów. Pierwszy polega na tym, że kompilator języka C# generuje kod, który tworzy nową, tymczasową lokalizację i kopiuje wartość z opakowania w to nowe miejsce. Wtedy tymczasowa lokalizacja staje się potrzebną zmienną. Druga możliwość to zgłoszenie przez kompilator błędu i uniemożliwienie wykonania operacji. W przedstawionym kodzie kompilator stosuje pierwszą strategię. Odbiorcą wywołania jest nowa tymczasowa lokalizacja w pamięci, a po wprowadzeniu zmiany ta tymczasowa lokalizacja jest usuwana.

Ten proces (sprawdzenie typu opakowanej wartości, wypakowanie jej w celu utworzenia lokalizacji z opakowaną wartością, alokacja tymczasowej zmiennej, skopiowanie wartości z opakowania do tymczasowej zmiennej i wywołanie metody dla tymczasowej lokalizacji) zachodzi za każdym razem, gdy stosowany jest wzorzec „wypakuj, a potem wywołaj”. Nie ma tu znaczenia, czy metoda modyfikuje zmienną. Jeśli metoda nie zmienia danych, niektórych operacji można uniknąć. Ponieważ jednak kompilator języka C# nie wie, czy wywołana metoda będzie próbowała modyfikować odbiorcę, musi wybrać ostrożne rozwiązanie.

Kosztowne operacje można wyeliminować, jeśli metoda interfejsu jest wywoływana dla opakowanej wartości typu bezpośredniego. Wtedy oczekiwane jest takie, że odbiorcą wywołania jest lokalizacja opakowanej wartości. Jeśli metoda interfejsu modyfikuje lokalizację, zmieniona powinna być właśnie lokalizacja opakowanej wartości. To pozwala uniknąć sprawdzania typu, alokacji nowej tymczasowej lokalizacji i kopowania danych. Środowisko uruchomieniowe musi tylko zastosować lokalizację opakowanej wartości jako odbiorcę wywołania metody struktury.

Na listingu 9.7 wywoływana jest dwuargumentowa wersja metody `ToString()` występująca w interfejsie `IFormattable`. Ten interfejs jest zaimplementowany w typie bezpośredni `int`. W tym przykładzie odbiorcą wywołania jest opakowana wartość typu bezpośredniego, nie jest ona jednak wypakowywana w celu wywołania wspomnianej metody interfejsu.

Listing 9.7. Unikanie wypakowywania i kopiowania

```
int number;
object thing;
number = 42;
// Opakowywanie.
thing = number;
// Bez konwersji wynikającej z wypakowywania danych.
string text = ((IFormattable)thing).ToString(
    "X", null);
Console.WriteLine(text);
```

Możliwe, że zastanawiasz się teraz nad następującą kwestią — co by się stało, gdyby wywołano wirtualną metodę `ToString()` z typu `object` dla odbiorcy w postaci instancji typu bezpośredniego? Czy ta instancja będzie opakowana, czy wypakowana? W zależności od szczegółów budowy programu mogą wystąpić różne scenariusze.

- Jeśli odbiorca jest wypakowany, a w strukturze przesłonięto metodę `ToString()`, nastąpi bezpośrednie wywołanie przesłaniającej metody. Nie trzeba wywoływać metody wirtualnie, ponieważ nie może ona zostać przesłonięta w dalszych klasach pochodnych. Wynika to z tego, że wszystkie typy bezpośrednie są automatycznie zamknięte.
- Jeśli odbiorca jest wypakowany, ale w strukturze nie przesłonięto metody `ToString()`, trzeba wywołać implementację z klasy bazowej. Oczekuje ona odbiorcy w postaci referencji do obiektu. Dlatego odbiorca zostaje opakowany.
- Jeśli odbiorca jest opakowany, a w strukturze przesłonięta jest metoda `ToString()`, lokalizacja z opakowania jest przekazywana (bez wypakowywania wartości) do przesłaniającej metody.
- Jeśli odbiorca jest opakowany, ale w strukturze nie przesłonięto metody `ToString()`, referencja do opakowania jest przekazywana do (oczekującej referencji) implementacji metody z klasy bazowej.

Wyliczenia

Porównaj dwa fragmenty kodu przedstawione na listingu 9.8.

Listing 9.8. Porównanie instrukcji `switch` dla liczb całkowitych i dla wartości z wyliczenia

```
int connectionState;
// ...
switch (connectionState)
{
    case 0:
        // ...
        break;
    case 1:
        // ...
        break;
    case 2:
        // ...
        break;
    case 3:
        // ...
        break;
}
ConnectionState connectionState;
// ...
switch (connectionState)
{
    case ConnectionState.Connected:
        // ...
```

```

break;
case ConnectionState.Connecting:
    // ...
    break;
case ConnectionState.Disconnected:
    // ...
    break;
case ConnectionState.Disconnecting:
    // ...
    break;
}

```

Oczywiście różnica, jeśli chodzi o czytelność, jest bardzo duża. W drugim fragmencie warunki w instrukcji są formą dokumentacji. Jednocześnie wydajność obu wersji w czasie wykonywania programu jest identyczna. Aby uzyskać ten efekt, w drugim fragmencie wykorzystano w poszczególnych warunkach **wartości wyliczeniowe**.

Wyliczenie to typ bezpośredni, który może zadeklarować programista. Ważną cechą wyliczeń jest to, że pozwalają zadeklarować na etapie komplikacji zestaw stałych wartości, które można wskazywać za pomocą nazw. Umożliwia to poprawę czytelności kodu. Składnię typowego wyliczenia pokazano na listingu 9.9.

Listing 9.9. Definiowanie wyliczenia

```

enum ConnectionState
{
    Disconnected,
    Connecting,
    Connected,
    Disconnecting
}

```

Uwaga

Wyliczenie może być używane także jako bardziej czytelny zastępnik wartości logicznych. Na przykład wywołanie metody `SetState(true)` jest mniej zrozumiałe niż `SetState(DeviceState.On)`.

Aby zastosować wartość z wyliczenia, należy poprzedzić ją nazwą tego wyliczenia. Na przykład w celu podania wartości `Connected` należy zastosować składnię `ConnectionState.Connected`. Nie używaj nazwy typu wyliczeniowego jako członu nazwy wartości. Unikniesz dzięki temu nadmiarowości (występującej na przykład w nazwie `ConnectionState.ConnectionStateConnected`). Zgodnie z konwencją nazwa wyliczenia powinna być podana w liczbie pojedynczej (chyba że wyliczenie zawiera flagi bitowe, co opisano dalej). Oznacza to, że powinno się używać nazwy `ConnectionState` zamiast `ConnectionStates`.

Wartości w wyliczeniu odpowiadają stałym całkowitoliczbowym. Domyślnie pierwsza wartość w wyliczeniu odpowiada liczbie 0, a każda kolejna liczbie o jeden większej. W wyliczeniach możesz też jednak jawnie przypisywać liczby, co pokazano na listingu 9.10.

Listing 9.10. Definiowanie typu wyliczeniowego

```
enum ConnectionState : short
{
    Disconnected,
    Connecting = 10,
    Connected,
    Joined = Connected,
    Disconnecting
}
```

W tym kodzie wartość `Disconnected` odpowiada domyślnej liczbie 0, a do wartości `Connecting` jawnie przypisano liczbę 10. Do wartości `Connected` przypisana zostanie liczba 11. Do wartości `Joined` też przypisano 11 (czyli liczbę ustawioną dla wartości `Connected`); w tym miejscu nie trzeba poprzedzać wartości `Connected` nazwą typu wyliczeniowego, ponieważ wartość znajduje się w zasięgu tego typu. Do wartości `Disconnecting` przypisywana jest liczba 12.

Dla wyliczenia zawsze określany jest typ wiążany z wartościami. Może to być dowolny typ całkowitoliczbowy inny niż `char`. Wydajność typu wyliczeniowego jest identyczna jak podstawowego typu całkowitoliczbowego. Domyślnie stosowany jest typ `int`, możesz jednak wykorzystać inny typ, używając składni charakterystycznej dla dziedziczenia. Na przykład na listingu 9.10 użyty typ to `short`. Dla zachowania spójności składnia wykorzystywana w wyliczeniach jest odpowiednikiem składni do tworzenia typów pochodnych, jednak tu nie służy ona do budowania relacji dziedziczenia. Klasą bazową wszystkich wyliczeń jest `System.Enum`, która sama dziedziczy po typie `System.ValueType`. Ponadto typy wyliczeniowe są zamknięte. Nie możesz utworzyć typu pochodnego od istniejącego typu wyliczeniowego, aby dodać nowe składowe.

Wskazówka

ROZWAŻ używanie domyślnych 32-bitowych liczb całkowitych jako typu podstawowego w wyliczeniu. Mniej pojemne typy stosuj tylko w celu zapewnienia współdziałania między kodem w różnych językach. Z bardziej pojemy typów korzystaj jedynie do tworzenia wyliczeń z flagami² zawierających więcej niż 32 wartości.

Wyliczenie to tylko zestaw nazw reprezentujących wartości typu podstawowego. Nie istnieje mechanizm sprawiający, że do zmiennej typu wyliczeniowego można przypisywać wyłącznie wartości wymienione w deklaracji tego typu. Ponieważ można zrzutować liczbę całkowitą 42 na typ `short`, można też zrzutować tę liczbę na zmienną typu `ConnectionState`, choć w wyliczeniu `ConnectionState` nie ma wartości odpowiadającej tej liczbie. Jeśli wartość można przekształcić na liczbę typu podstawowego z wyliczenia, konwersja na typ wyliczeniowy też zakończy się powodzeniem.

² Zobacz omówienie w podrozdziale „Wyliczenia jako flagi” w dalszej części rozdziału.

Zaletą tego dziwnego rozwiązania jest to, że do wyliczeń w nowszych wersjach interfejsu API można dodawać kolejne wartości bez naruszenia poprawności istniejącego kodu. Ponadto wyliczenie określa nazwy znanych wartości, natomiast w czasie wykonywania programu można stosować także nieznane wartości. Uciążliwe jest to, że programista musi pisać kod defensywnie, z uwzględnieniem nienazwanych wartości. Nierożsądne byłoby na przykład zastąpienie warunku case ConnectionState.Disconnecting warunkiem default w oczekiwaniu, że jedyna możliwa wartość przypadku default to właśnie ConnectionState.Disconnecting. Należy jawnie obsługiwać wartość Disconnecting, a w warunku default można zgłaszać błąd lub wykonywać nieszkodliwe operacje. Jak jednak wspomniano wcześniej, konwersja między wyliczeniem a typem podstawowym i w drugą stronę wymaga jawnego rzutowania (nie zachodzi tu konwersja niejawnna). Nie można na przykład umieścić w kodzie wywołania ReportState(10), jeśli sygnatura metody to void ReportState(ConnectionState state). Jedyny wyjątek to przekazanie liczby 0. Konwersja tej wartości na dowolny typ wyliczeniowy zachodzi niejawnie.

Choć możesz dodać kolejne wartości do wyliczenia w późniejszych wersjach kodu, powinieneś zachować przy tym ostrożność. Wstawienie wartości między innymi elementami wyliczenia powoduje, że wartości wszystkich dalszych elementów się zwiększą. Jeśli na przykład dodasz wartość Flooded lub Locked przed Connected, liczba przypisana do Connected się zmieni. Zostanie to uwzględnione w kodzie ponownie komplilowanym z wykorzystaniem nowej wersji wyliczenia. Jednak kod skompilowany z wykorzystaniem starszej wersji nadal będzie używał dawnych, innych liczb. Ten problem można rozwiązać, zawsze umieszczając nowe wartości na końcu listy lub jawnie przypisując liczby do poszczególnych wartości.

Wskazówki

ROZWAŻ dodanie nowych wartości do istniejącego wyliczenia, ale pamiętaj przy tym o ryzyku naruszenia zgodności z istniejącym kodem.

UNIKAJ tworzenia wyliczeń reprezentujących „niekompletne” zbiory wartości (takie jak numery wersji produktu).

UNIKAJ tworzenia w wyliczeniach wartości „zarezerwowanych do późniejszego użytku”.

UNIKAJ tworzenia wyliczeń zawierających jedną wartość.

UDOSTĘPNIJ wartość 0 (oznaczającą brak wartości) w prostych wyliczeniach. Jest to wartość domyślna, ustawiana, gdy programista nie zainicjuje jawnie danych.

Wyliczenia różnią się od innych typów bezpośrednich, ponieważ dziedziczą po typie System.Enum (dopiero on jest pochodny od typu System.ValueType).

Zgodność typów wyliczeniowych

C# nie obsługuje bezpośredniego rzutowania między tablicami opartymi na dwóch różnych wyliczeniach. Takie rzutowanie jest jednak możliwe w środowisku CLR, jeśli oba wyliczenia korzystają z tego samego typu podstawowego. Aby obejść ograniczenie obowiązujące w języku C#, należy zrzutować dane najpierw na typ System.Array, tak jak w końcowej części listingu 9.11.

Listing 9.11. Rzutowanie między tablicami opartymi na wyliczeniach

```

enum ConnectionState1
{
    Disconnected,
    Connecting,
    Connected,
    Disconnecting
}
enum ConnectionState2
{
    Disconnected,
    Connecting,
    Connected,
    Disconnecting
}
class Program
{
    static void Main()
    {
        ConnectionState1[] states =
            (ConnectionState1[])(Array)new ConnectionState2[42];
    }
}

```

W tym przykładzie wykorzystano to, że w środowisku CLR zgodność typów w przypisaniach nie musi być tak ścisła jak w języku C#. Tę samą sztuczkę można zastosować, by przeprowadzić inne niedozwolone w języku konwersje, na przykład z typu `int[]` na `uint[]`. Posługuj się jednak tą techniką z rozwiązań, ponieważ specyfikacja języka C# nie wymaga, aby to rozwiązanie działało we wszystkich implementacjach środowiska CLR.

Konwersja między wyliczeniami a łańcuchami znaków

Jedną z wygodnych cech wyliczeń jest to, że metoda `ToString()` (wywoływaną na przykład w metodach takich jak `System.Console.WriteLine()`) wyświetla identyfikator wartości z wyliczenia:

```
System.Diagnostics.Trace.WriteLine(
    $"Aktualny stan połączenia to { ConnectionState.Disconnecting }");
```

Ten kod zapisuje tekst z danych wyjściowych 9.3 w buforze śledzenia.

DANE WYJŚCIOWE 9.3.

Aktualny stan połączenia to Disconnecting.

Konwersja z łańcucha znaków na wyliczenie jest trudniejsza, ponieważ wymaga użycia metody statycznej klasy bazowej `System.Enum`. Na listingu 9.12 pokazano, jak wywołać tę metodę bez używania typów generycznych (zobacz rozdział 12.). Efekt działania kodu znajdziesz w danych wyjściowych 9.4.

Listing 9.12. Konwersja łańcucha znaków na wartość typu wyliczeniowego za pomocą metody `Enum.Parse()`

```
ThreadPriorityLevel priority = (ThreadPriorityLevel)Enum.Parse(
    typeof(ThreadPriorityLevel), "Idle");
Console.WriteLine(priority);
```

DANE WYJŚCIOWE 9.4.

Idle

Pierwszym parametrem wywołania `Enum.Parse()` jest tu typ określany za pomocą słowa kluczowego `typeof()`. W tym kodzie pokazano, jak na etapie komplikacji zidentyfikować typ — na przykład literału z wartością tego typu (zobacz rozdział 18.).

Do wersji .NET Framework 4 nie istniała metoda `TryParse()`, dlatego kod mający działać w starszych wersjach platformy wymaga odpowiedniej obsługi wyjątków, jeśli możliwe jest, że łańcuch znaków nie będzie odpowiadał żadnej wartości z danego wyliczenia. Metoda `TryParse<T>()` z wersji .NET Framework 4 wykorzystuje typy generyczne, przy czym parametr określający typ może zostać ustalony w wyniku inferencji. Dlatego możliwa jest konwersja łańcucha znaków na wyliczenie w sposób przedstawiony na listingu 9.13.

Listing 9.13. Przekształcanie łańcucha znaków na wyliczenie za pomocą metody `Enum.TryParse<T>()`

```
System.Diagnostics.ThreadPriorityLevel priority;
if (Enum.TryParse("Idle", out priority))
{
    Console.WriteLine(priority);
}
```

Ta technika eliminuje konieczność stosowania obsługi wyjątków w sytuacjach, w których konwersja łańcucha znaków może się zakończyć niepowodzeniem. Wystarczy sprawdzić w kodzie wartość logiczną zwroconą przez wywołanie `TryParse<T>()`.

Niezależnie od tego, czy w kodzie używane jest rozwiązywanie oparte na metodzie `Parse`, czy `TryParse`, w kontekście konwersji z łańcucha znaków na wyliczenie należy pamiętać o tym, że w rzutowaniu nie są uwzględniane tłumaczenia. Dlatego programiści powinni stosować rzutowanie tego rodzaju tylko do komunikatów, które nie są udostępniane użytkownikom (jeśli program jest tłumaczony).

Wskazówka

UNIKAJ bezpośredni konwersji między wyliczeniami i łańcuchami znaków, jeśli tekst musi być tłumaczony na język użytkowników.

Wyliczenia jako flagi

Programiści często chcą, by wartości w wyliczeniach nie tylko były unikatowe, ale też by mogły reprezentować kombinacje różnych wartości. Pomyśl na przykład o typie `System.IO.FileAttributes`. To wyliczenie (przedstawione na listingu 9.14) określa różne atrybuty

pliku (tylko do odczytu, ukryty, archiwum itd.). Wartości wyliczenia `FileAttributes` (inaczej niż w wyliczeniu `ConnectionState`, gdzie wartości wzajemnie się wykluczają) są przeznaczone do łączenia. Plik może być jednocześnie przeznaczony tylko do odczytu i ukryty. Aby umożliwić łączenie wartości wyliczenia, każda z nich jest reprezentowana za pomocą jednego bitu.

Listing 9.14. Używanie wartości wyliczeń jako flag

```
[Flags] public enum FileAttributes
{
    ReadOnly = 1<<0,      // 00000000000000000001
    Hidden = 1<<1,        // 00000000000000000010
    System = 1<<2,        // 0000000000000000000100
    Directory = 1<<4,     // 000000000000000000010000
    Archive = 1<<5,       // 0000000000000000000100000
    Device = 1<<6,        // 00000000000000000001000000
    Normal = 1<<7,        // 000000000000000000010000000
    Temporary = 1<<8,     // 0000000000000000000100000000
    SparseFile = 1<<9,     // 0000000000000000000100000000
    ReparsePoint = 1<<10,   // 0000000000000000000100000000
    Compressed = 1<<11,   // 00000000000000000001000000000
    Offline = 1<<12,      // 000000000000000000010000000000
    NotContentIndexed = 1<<13, // 000010000000000000000000000000
    Encrypted = 1<<14,     // 0010000000000000000000000000000
    IntegrityStream = 1<<15, // 00100000000000000000000000000000
    NoScrubData = 1<<17,    // 10000000000000000000000000000000
}
```

Uwaga

Zauważ, że nazwy wyliczeń reprezentujących flagi bitowe są zwykle podawane w liczbie mniej. Jest to informacja, że wartość typu reprezentuje zbiór flag.

Do łączenia wartości z wyliczenia służy bitowy operator OR. Do sprawdzania, czy dana flaga jest ustawiona, służy wprowadzona w platformie Microsoft .NET Framework 4.0 metoda `HasFlag()`. Można też wykorzystać do tego bitowy operator AND. Obie te operacje pokazano na listingu 9.15.

Listing 9.15. Używanie bitowych operatorów OR i AND do wyliczeń zawierających flagi³

```
using System;
using System.IO;

public class Program
{
    public static void Main()
    {
        // ...
        string fileName = @"enumtest.txt";
```

³ Warto zauważyć, że w systemie Linux wartość `FileAttributes.Hidden` nie działa.

```

System.IO.FileInfo file =
    new System.IO.FileInfo(fileName);

file.Attributes = FileAttributes.Hidden |
    FileAttributes.ReadOnly;

Console.WriteLine($"{file.Attributes} = {(int)file.Attributes}");

// Dodane w C# 4.0 i platformie Microsoft .NET Framework 4.0.
if (!(
    System.Runtime.InteropServices.RuntimeInformation.IsOSPlatform(
        OSPlatform.Linux) ||
    System.Runtime.InteropServices.RuntimeInformation.IsOSPlatform(
        OSPlatform.OSX)))
{
    if (!file.Attributes.HasFlag(FileAttributes.Hidden))
    {
        throw new Exception("Plik nie jest ukryty.");
    }
}
// W wersjach starszych niż C# 4.0 i .NET 4.0 należy używać operatorów bitowych.
if ((file.Attributes & FileAttributes.ReadOnly) !=
    FileAttributes.ReadOnly)
{
    throw new Exception("Plik nie jest przeznaczony tylko do odczytu.");
}

// ...
}

```

Wynik działania kodu z listingu 9.15 pokazano w danych wyjściowych 9.5.

DANE WYJŚCIOWE 9.5.

```
Hidden | ReadOnly = 3
```

Użycie bitowego operatora OR umożliwia ustawienie flag oznaczających, że plik jest przeznaczony tylko do odczytu i ukryty.

Wartości w wyliczeniu nie muszą reprezentować tylko jednej flagi. Sensownym rozwiąza niem jest zdefiniowanie dodatkowych flag, reprezentujących często używane kombinacje wartości. Pokazano to na listingu 9.16.

Listing 9.16. Definiowanie w wyliczeniu wartości reprezentujących często używane kombinacje flag

```

[Flags] enum DistributedChannel
{
    None = 0,
    Transacted = 1,
    Queued = 2,
    Encrypted = 4,
    Persisted = 16,
    FaultTolerant =
        Transacted | Queued | Persisted
}

```

Dobrą praktyką jest dodawanie do wyliczeń z flagami wartości `None` odpowiadającej liczbie zero. Początkowa wartość domyślna pola typu wyliczeniowego (lub elementu w tablicy z wartościami typu wyliczeniowego) to właśnie 0. Unikaj dodawania ostatniej wartości o nazwie `Maximum` (lub podobnej), ponieważ może ona zostać uznana za prawidłową wartość wyliczenia. Aby sprawdzić, czy jakąś wartość występuje w wyliczeniu, wywołaj metodę `System.Enum.IsDefined()`.

Wskazówki

STOSUJ atrybut `FlagsAttribute` do oznaczania wyliczeń zawierających flagi.

UDOSTĘPNIAJ we wszystkich wyliczeniach z flagami wartość `None` reprezentującą 0.

UNIKAJ tworzenia wyliczeń z flagami, gdzie wartość zero oznacza coś innego niż brak ustawionych flag.

ROZWAŻ udostępnienie specjalnych wartości odpowiadających często używanym kombinacjom flag.

NIE dodawaj wartości końcowej (na przykład o nazwie `Maximum`). Takie wartości mogą być mylące dla użytkowników.

STOSUJ potęgi liczby dwa, by umożliwić przedstawienie w unikatowy sposób wszystkich kombinacji flag.

ZAGADNIENIE DLA ZAAWANSOWANYCH

Atrybut `FlagsAttribute`

Jeśli zdecydujesz się utworzyć wyliczenie z flagami bitowymi, deklarację takiego wyliczenia należy opatrzyć atrybutem typu `FlagsAttribute`. Atrybut należy podać w nawiasie kwadratowym (zobacz rozdział 18.) tuż przed deklaracją wyliczenia, co pokazano na listingu 9.17.

Listing 9.17. Używanie atrybutu typu `FlagsAttribute`

```
// Typ FileAttributes jest zdefiniowany w przestrzeni nazw System.IO.

[Flags] // Dodawanie atrybutu typu FlagsAttribute do wyliczenia.
public enum FileAttributes
{
    ReadOnly =      1<<0, // 0000000000000001
    Hidden =        1<<1, // 0000000000000010
    ...
}
using System;
using System.Diagnostics;
using System.IO;

class Program
{
    public static void Main()
    {
        string fileName = @"enumtest.txt";
```

```
FileInfo file = new FileInfo(fileName);
file.Open(FileMode.Create).Close();

FileAttributes startingAttributes =
    file.Attributes;

file.Attributes = FileAttributes.Hidden |
    FileAttributes.ReadOnly;

Console.WriteLine("{0} wyświetlane jako \"{1}\",
    file.Attributes.ToString().Replace(", ", " |"),
    file.Attributes);

FileAttributes attributes =
    (FileAttributes) Enum.Parse(typeof(FileAttributes),
    file.Attributes.ToString());

Console.WriteLine(attributes);

File.SetAttributes(fileName,
    startingAttributes);
file.Delete();
}
}
```

Efekt działania kodu z listingu 9.17 przedstawiono w danych wyjściowych 9.6.

DANE WYJŚCIOWE 9.6.

```
"ReadOnly | Hidden" wyświetlane jako "ReadOnly, Hidden"
ReadOnly, Hidden
```

Ten atrybut informuje, że wartości z wyliczenia można ze sobą łączyć, a ponadto zmienia działanie metod `ToString()` i `Parse()`. Wywołanie metody `ToString()` dla obiektu wyliczenia opatrzonego atrybutem typu `FlagsAttribute` powoduje wyświetlenie łańcuchów znaków reprezentujących wszystkie ustawione flagi z tego wyliczenia. Na listingu 9.17 wywołanie `file.Attributes.ToString()` zwraca wartość `ReadOnly, Hidden` zamiast liczb 3, która pojawi się, jeśli pominiesz atrybut typu `FlagsAttribute`. Jeśli dwie wartości wyliczenia są takie same, metoda `ToString()` zwróci pierwszą z nich. Jednak, o czym wspomniano już wcześniej, należy stosować tę technikę z rozwagą, ponieważ nie obsługuje ona tłumaczeń.

Możliwe jest też przekształcenie łańcucha znaków na wartość z wyliczenia. Każdą podawaną wartość z wyliczenia należy oddzielić od innych przecinkiem.

Zauważ, że atrybut typu `FlagsAttribute` nie powoduje automatycznie przypisania unikatowych wartości do flag ani sprawdzenia, czy flagi są unikatowe. Nie miałoby to sensu, ponieważ powtórzenia i różne kombinacje wartości są często przydatne. Programista musi jawnie przypisać liczby do poszczególnych wartości wyliczenia.

Podsumowanie

Ten rozdział rozpoczął się od omówienia definiowania niestandardowych typów bezpośrednich. Ponieważ modyfikowalne typy bezpośrednio często prowadzą do powstawania trudnego do zrozumienia lub błędного kodu (a także dlatego, że typy bezpośrednio często mają odzwierciedlać niemodyfikowalne wartości), warto tworzyć tylko niemodyfikowalne typy bezpośredni. W rozdziale omówiono też opakowywanie typów bezpośredni w sytuacjach, gdy ze względu na polimorfizm trzeba je traktować jako typy referencyjne.

Osobliwości związane z opakowywaniem są trudne do zauważenia. Większość z nich prowadzi do problemów w czasie wykonywania programu, a nie na etapie komplikacji. Wprawdzie ważne jest, by poznać te osobliwości, co pozwoli im zapobiegać, jednak zbytnie koncentrowanie się na potencjalnych kłopotach może sprawić, że nie dostrzeżesz przydatności typów bezpośredni i korzyści, jakie zapewniają w obszarze wydajności. Programiści nie powinni obawiać się korzystać z typów bezpośredni. Takie typy są powszechnie używane w prawie każdym rozdziale tej książki, jednak rzadko prowadzą do problemów. Zaprezentowano tu kod dotyczący poszczególnych trudności, aby je zilustrować, jednak w praktyce wzorce tego rodzaju występują rzadko. By uniknąć większości z nich, należy stosować się do wskazówek i nie tworzyć modyfikowalnych typów bezpośredni. To właśnie ta wytyczna wyjaśnia, dlaczego wbudowane typy bezpośrednie są niemodyfikowalne.

Prawdopodobnie jedyny problem, który pojawia się stosunkowo często, to wielokrotne opakowywanie danych w pętlach. Jednak typy generyczne pozwalają znacznie ograniczyć liczbę operacji opakowywania, a nawet bez tych typów wydajność kodu rzadko spada na tyle, by uzasadnić unikanie typów bezpośredni (chyba że stosowany algorytm wymagający opakowywania został uznany za przyczynę wąskiego gardła).

Programiści stosunkowo rzadko tworzą własne struktury. Struktury oczywiście odgrywają ważną rolę w programowaniu w języku C#, jednak liczba niestandardowych struktur rozwijanych przez typowych programistów jest zwykle niewielka w porównaniu do liczby rozwijanych niestandardowych klas. Duża liczba niestandardowych struktur jest najczęściej spotykana w kodzie mającym współdziałać z kodem niezarządzanym.

Wskazówka

NIE definiuj struktury, chyba że logicznie reprezentuje ona jedną wartość, zajmuje do 16 bajtów pamięci, jest niemodyfikowalna i rzadko wymaga opakowywania.

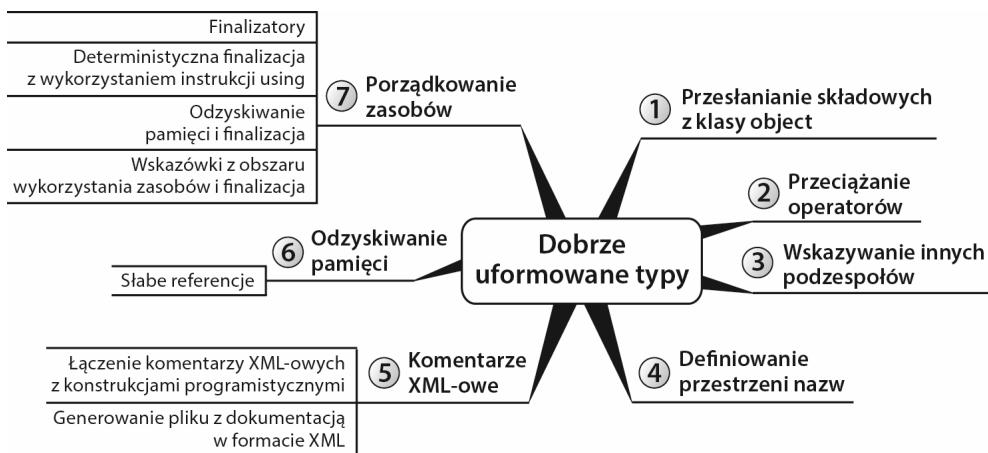
W tym rozdziale przedstawiono też wyliczenia. Typy wyliczeniowe to standardowe konstrukcje występujące w wielu językach programowania. Pomagają zwiększyć użyteczność interfejsów API i czytelność kodu.

W rozdziale 10. znajdziesz więcej wskazówek na temat tworzenia dobrze uformowanych typów — zarówno bezpośredni, jak i referencyjnych. Na początku zapoznasz się z przesłaniem składowych wirtualnych obiektów i definiowaniem metod przeciążających operator. Te dwa zagadnienia dotyczą zarówno struktur, jak i klas, są jednak bardziej istotne po zapoznaniu się z definiowaniem struktur i zapewnianiem ich dobrego uformowania.

10

Dobrze uformowane typy

W WCZEŚNIEJSZYCH ROZDZIAŁACH OPISANO WIĘKSZOŚĆ składni służącej do definiowania klas i struktur. Pozostało jeszcze jednak kilka szczegółów przydatnych do uzupełniania definicji o końcowe szlify. W tym rozdziale wyjaśniono, jak dopracować deklarację typu.



Przesłanianie składowych z klasy object

W rozdziale 7. opisano, że wszystkie klasy i struktury dziedziczą po klasie object. Ponadto omówiono każdą metodę z tej klasy i wyjaśniono, że niektóre z tych metod są wirtualne. W tym podrozdziale poznasz szczegóły dotyczące przesłaniania metod wirtualnych.

Przesłanianie metody `ToString()`

Wywołanie metody `ToString()` dla dowolnego obiektu domyślnie powoduje zwrócenie pełnej nazwy danej klasy. Na przykład jeśli wywołasz tę metodę dla obiektu typu `System.IO.FileStream`, otrzymaszła串uch znaków `System.IO FileStream`. Jednak w niektórych klasach metoda `ToString()` może zwracać bardziej znaczące dane. Na przykład dla typu `string` metoda `ToString()` zwraca zawartość 串uchu znaków. Podobnie zwrócenie imienia i nazwiska w obiekcie typu `Contact` byłoby sensowniejsze niż generowanie nazwy klasy. Na lisingu 10.1 przesłonięto metodę `ToString()`, tak by zwracała tekstową reprezentację obiektu typu `Coordinate`.

Listing 10.1. Przesłanianie metody `ToString()`

```
public struct Coordinate
{
    public Coordinate(Longitude longitude, Latitude latitude)
    {
        Longitude = longitude;
        Latitude = latitude;
    }

    public Longitude Longitude { get; }
    public Latitude Latitude { get; }

    public override string ToString() =>
        $"{Longitude} {Latitude}";

    // ...
}
```

Metody z rodziny „write”, na przykład `Console.WriteLine()` i `System.Diagnostics.Trace.Write()`, wywołują metodę `ToString()` obiektu¹, dlatego przesłonięcie tej metody pozwala wyświetlać bardziej przydatne informacje niż przy użyciu domyślnej implementacji. Warto więc rozważyć przesłonięcie metody `ToString()`, jeśli w danych wyjściowych można wyświetlać wartościowe informacje diagnostyczne. Jest tak zwłaszcza wtedy, gdy docelowymi odbiorcami komunikatów są programiści, ponieważ domyślnie metoda `object.ToString()` wyświetla nazwę typu, co nie jest pomocne. Metoda `ToString()` przydaje się do debugowania w środowisku IDE programisty i do zapisu danych w plikach dzienników. Dlatego generowane 串uchy znaków powinny być krótkie (nie dłuższe niż szerokość ekranu), co pozwoli uniknąć ich przycinania. Jednak brak obsługi tłumaczeń i innych zaawansowanych mechanizmów formatowania sprawia, że to podejście nie nadaje się dobrze do wyświetlania informacji użytkownikom końcowym.

¹ Chyba że bezpośrednio używany jest operator rzutowania, co opisano w ZAGADNIENIU DLA ZAAWANSOWANYCH „Operator rzutowania”.

Wskazówka

PRZESŁANIAJ metodę `ToString()`, jeśli możliwe jest zwracanie przydatnych programistom informacji diagnostycznych.

STARAJ się dbać o to, by metoda `ToString()` zwracała krótkie łańcuchy znaków.

NIE zwracaj pustych łańcuchów znaków ani wartości `null` w metodzie `ToString()`.

UNIKAJ zgłaszania wyjątków i wywoływanie zauważalnych efektów ubocznych (polegających na zmianie stanu obiektu) w metodzie `ToString()`.

UDOSTĘPNIAJ przeciążoną wersję `ToString(string format)` lub implementację interfejsu `IFormattable`, jeśli zwracana wartość wymaga modyfikacji związanych z kulturą lub formatowania (dotyczy to na przykład obiektów typu `DateTime`).

ROZWAŻ zwracanie unikatowych łańcuchów znaków w metodzie `ToString()`, by pozwalały identyfikować instancje obiektów.

Przesłanianie metody `GetHashCode()`

Przesłanianie metody `GetHashCode()` jest bardziej skomplikowane niż przesłanianie metody `ToString()`. Powinieneś jednak przesłaniać ją, jeśli przesłaniasz także metodę `Equals()`. Jeżeli tego nie zrobisz, kompilator wyświetli ostrzeżenie. Metodę `GetHashCode()` warto przesłonić także wtedy, gdy używasz jej do tworzenia kluczy w kolekcji z haszowaniem (typu `System.Collections.Hashtable` lub `System.Collections.Generic.Dictionary`).

Skróty (ang. *hash codes*) mają umożliwiać *wydajne równoważenie tablicy z haszowaniem* za pomocą liczb odpowiadających wartości obiektu. Poniżej wymieniono wybrane cechy dobrej implementacji metody `GetHashCode()`.

- *Wymagane* — równe sobie obiekty muszą mieć takie same skróty (jeśli `a.Equals(b)`, to `a.GetHashCode() == b.GetHashCode()`).
- *Wymagane* — wartości zwracane przez metodę `GetHashCode()` w czasie życia danego obiektu powinny być takie same — także wtedy, gdy dane w obiekcie się zmieniają. W wielu sytuacjach należy zapisać zwróconą wartość, by spełnić ten wymóg. Jeśli jednak to robisz, nie stosuj skrótów do sprawdzania równości obiektów, ponieważ dwa identyczne obiekty — jeden z zapisanym skrótem wygenerowanym dla dawnych właściwości — mogą wtedy zostać uznane za różne.
- *Wymagane* — metoda `GetHashCode()` nie powinna zgłaszać wyjątków. Zawsze powinna z powodzeniem zwrócić wartość.
- *Wydajność* — skróty powinny być unikatowe, jeśli jest to możliwe. Jednak ponieważ skróty to liczby typu `int`, mogą się powtarzać, jeśli obiektów może być więcej niż wartości w typie `int`. Dotyczy to więc prawie wszystkich typów. Oczywistym przykładem jest tu typ `long`, ponieważ istnieje więcej możliwych wartości typu `long` niż typu `int`.

- *Wydajność* — możliwe wartości skrótów powinny być równomiernie rozłożone po przedziale wartości typu int. Na przykład jeśli programista nie uwzględni tego, że rozkład znaków w łańcuchach w językach opartych na alfabetie łacińskim pokrywa głównie 128 początkowych znaków ASCII, otrzyma bardzo nierównomierny rozkład. Utworzony w ten sposób algorytm w metodzie GetHashCode() nie będzie dobry.
- *Wydajność* — należy zoptymalizować wydajność metody GetHashCode(). Jest ona często używana w metodzie Equals(), by uprościć sprawdzanie równości w sytuacjach, gdy skróty różnią się od siebie. Dlatego metoda GetHashCode() jest często wywoływana, gdy dany typ jest używany jako typ kluczowy w słownikach.
- *Wydajność* — niewielkie różnice między dwoma obiektami powinny prowadzić do dużych różnic w wartościach skrótów. Najlepiej jest, gdy jednobitowa różnica między obiektami skutkuje średnio około 16 bitami różnicy w skrócie. To pomaga zagwarantować, że tablica z haszowaniem będzie zrównoważona niezależnie od tego, jak skróty są dzielone na kubelki.
- *Bezpieczeństwo* — napastnicy powinni mieć trudność z utworzeniem obiektu o określonym skrócie. Atak może polegać na przesłaniu do tablicy z haszowaniem bardzo dużej liczby obiektów o tych samych skrótach. Tablica z haszowaniem staje się wtedy niewydajna, co umożliwia przeprowadzenie ataku DoS.

Te wytyczne i reguły są oczywiście sprzeczne w niektórych punktach. Bardzo trudno opracować algorytm generowania skrótów, który jednocześnie jest szybki i spełnia wszystkie wymienione warunki. Podobnie jak w przypadku każdego problemu z obszaru projektowania, trzeba przeprowadzić właściwą ocenę sytuacji i dokonać rzetelnych pomiarów wydajności, by uzyskać dobre rozwiązanie.

Przyjrzyj się przedstawionej na listingu 10.2 implementacji metody GetHashCode() dla typu Coordinate.

Listing 10.2. Implementowanie metody GetHashCode()

```
public struct Coordinate
{
    public Coordinate(Longitude longitude, Latitude latitude)
    {
        Longitude = longitude;
        Latitude = latitude;
    }

    public Longitude Longitude { get; }
    public Latitude Latitude { get; }

    public override int GetHashCode() => =>
        HashCode.Combine(
            Longitude.GetHashCode(), Latitude.GetHashCode());
}

// ...
}
```

Istnieje wiele sprawdzonych algorytmów używanych do implementowania metody `GetHashCode()`. Każdy z nich działa zgodnie z opisanymi wytycznymi (zobacz <http://bit.ly/39yP8lm>). Jednak najłatwiej jest wywołać metodę `System.HashCode.Combine()` z argumentami w postaci wyników wywołań metody `GetHashCode()` dla każdego pola identyfikującego obiekt (są to pola określające unikalność obiektu). Jeśli pola identyfikujące obiekt są liczbami, uważaj, abyomyłkowo nie użyć wartości pól zamiast skrótów. Typ `ValueTuple` wywołuje metodę `HashCode.Combine()`, dlatego możliwe, że łatwiej będzie Ci zapamiętać, że należy utworzyć obiekt typu `ValueTuple` z samymi polami identyfikującymi (a nie ich skrótami), a następnie wywołać składową `GetHashCode()` wynikowej krotki.

W typie `Coordinate` wartość skrótu nie jest zapisywana. Ponieważ wszystkie pola uwzględniane w obliczeniach skrótu są przeznaczone tylko do odczytu, wartość skrótu nie może się zmienić. Jeśli jednak wartości pól mogą się zmieniać lub gdy zapisanie skrótu daje znaczne korzyści ze względu na wydajność, skrót należy zachować. Jeżeli zdecydujesz się zapisywać skróty, nie używaj ich do sprawdzania równości. Może to spowodować, że identyczne obiekty nie zostaną uznane za równe, ponieważ skróty zostały obliczone przed zmianą właściwości.

Przesłanianie metody `Equals()`

Przesłonięcie metody `Equals()` bez przesłonięcia metody `GetHashCode()` skutkuje wyświetleniem ostrzeżenia w postaci przedstawionej w danych wyjściowych 10.1.

DANE WYJŚCIOWE 10.1.

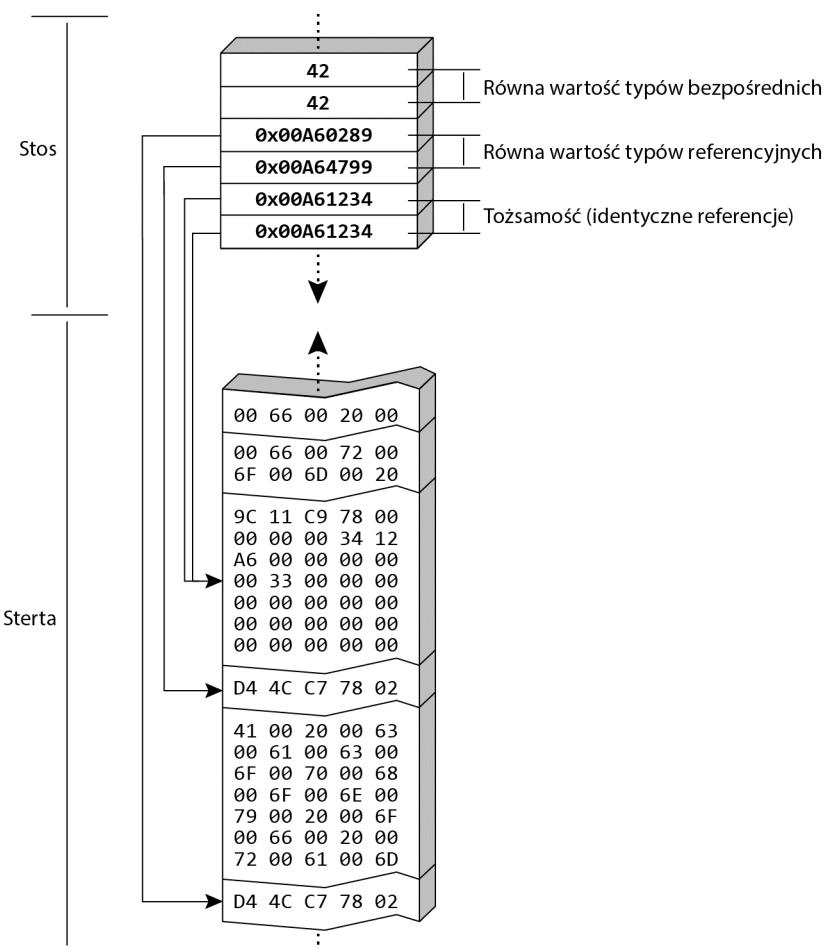
```
warning CS0659: '<Class Name>' overrides Object.Equals(object o) but  
does not override Object.GetHashCode()
```

Programiści zwykle oczekują, że przesłonięcie metody `Equals()` będzie bardzo proste. Aby jednak wykonać to zadanie, trzeba uwzględnić zaskakująco dużą liczbę drobiazgów, które wymagają starannego przemyślenia i testów.

Tożsamość obiektów a równe wartości obiektów

Dwie referencje są identyczne, jeśli obie prowadzą do tej samej instancji. Klasa `object` (i w wyniku dziedziczenia wszystkie pochodne od niej typy) obejmuje metodę statyczną `ReferenceEquals()`, która bezpośrednio sprawdza, czy tego rodzaju tożsamość obiektów ma miejsce (zobacz rysunek 10.1).

Jednak równość referencji nie jest jedynym rodzajem równości. Dwie instancje można uznać za równe, jeśli wartości niektórych lub wszystkich składowych są identyczne. Przjrzyj się porównaniu dwóch obiektów typu `ProductSerialNumber` na listingu 10.3.



Rysunek 10.1. Tożsamość

Listing 10.3. Przesłanianie operatora równości

```

public sealed class ProductSerialNumber
{
    // ...
}
class Program
{
    static void Main()
    {
        ProductSerialNumber serialNumber1 =
            new ProductSerialNumber("PV", 1000, 09187234);
        ProductSerialNumber serialNumber2 = serialNumber1;
        ProductSerialNumber serialNumber3 =
            new ProductSerialNumber("PV", 1000, 09187234);

        // Te obiekty są tożsame.
        if (!ProductSerialNumber.ReferenceEquals(serialNumber1,
            serialNumber2))
    }
}

```

```
{  
    throw new Exception(  
        "serialNumber1 NIE zawiera tej samej "+  
        "referencji co serialNumber2");  
}  
// A tym samym są sobie równe.  
else if (!serialNumber1.Equals(serialNumber2))  
{  
    throw new Exception(  
        "serialNumber1 NIE ma wartości równej serialNumber2");  
}  
else  
{  
    Console.WriteLine(  
        "serialNumber1 ma referencję równą z serialNumber2");  
    Console.WriteLine(  
        "serialNumber1 ma wartość równą z serialNumber2");  
}  
  
// Te numery seryjne NIE są tożsame.  
if (ProductSerialNumber.ReferenceEquals(serialNumber1,  
    serialNumber3))  
{  
    throw new Exception(  
        "serialNumber1 MA referencję równą"+  
        "z serialNumber3");  
}  
// Mają jednak równe wartości (jeśli metoda Equals jest przeciążona).  
else if (!serialNumber1.Equals(serialNumber3) ||  
    serialNumber1 != serialNumber3)  
{  
    throw new Exception(  
        "serialNumber1 NIE ma wartości równej z serialNumber3");  
}  
  
Console.WriteLine("serialNumber1 ma wartość równą z serialNumber3");  
}  
}
```

Wynik działania kodu z listingu 10.3 pokazano w danych wyjściowych 10.2.

DANE WYJŚCIOWE 10.2.

```
serialNumber1 ma referencję równą z serialNumber2  
serialNumber1 ma wartość równą z serialNumber2  
serialNumber1 ma wartość równą z serialNumber3
```

W ostatniej asercji zastosowano wywołanie `ReferenceEquals()`, by pokazać, że obiekty `serialNumber1` i `serialNumber3` mają różne referencje. Jednak w kodzie do obu tych obiektów przypisano tę samą wartość, dlatego oba reprezentują ten sam fizyczny produkt. Jeśli jedna instancja jest tworzona na podstawie danych z bazy, a druga na podstawie ręcznie wprowadzonych informacji, należy uznać, że instancje mają równą wartość, a zatem produkt nie zostanie zduplikowany (ponownie zapisany) w bazie. Dwie identyczne referencje z definicji prowadzą do tej samej wartości. Jednak dwa różne obiekty mogą reprezentować tę samą

wartość, ale mieć różne referencje. Takie obiekty nie są tożsame ze względu na referencje, ale mogą mieć klucz oznaczający, że ich wartość jest taka sama.

Tylko typy referencyjne mogą być równe ze względu na referencje (czyli tożsame). Wywołanie metody `ReferenceEquals()` dla obiektów typu bezpośredniego zawsze daje wartość `false`, ponieważ typy bezpośrednie są opakowywane w momencie przekształcania ich na typ `object` na potrzeby tego wywołania. Nawet jeśli jako oba parametry metody `ReferenceEquals()` przekazana zostanie ta sama zmienna typu bezpośredniego, metoda zwróci `false`, ponieważ wartości zmiennych zostaną opakowane osobno. Ilustruje to listing 10.4. Każdy argument jest na nim umieszczany w innym opakowaniu, dlatego obiekty nigdy nie są równe ze względu na referencje.

Uwaga

Wywołanie metody `ReferenceEquals()` dla wartości typów bezpośrednich zawsze daje wynik `false`.

Listing 10.4. Obiekty typów bezpośrednich nigdy nie są równe ze względu na referencje

```
public struct Coordinate
{
    public Coordinate(Longitude longitude, Latitude latitude)
    {
        Longitude = longitude;
        Latitude = latitude;
    }

    public Longitude Longitude { get; }
    public Latitude Latitude { get; }
    // ...
}

class Program
{
    public void Main()
    {
        //...

        Coordinate coordinate1 =
            new Coordinate( new Longitude(48, 52),
                new Latitude(-2, -20));

        // Obiekty typów bezpośrednich nigdy nie są równe ze względu na referencje.
        if ( Coordinate.ReferenceEquals(coordinate1,
            coordinate1) )
        {
            throw new Exception(
                "coordinate1 jest równa ze względu na referencje z coordinate1");
        }
        Console.WriteLine(
            "coordinate1 NIE jest równa ze względu na referencje z sobą samą");
    }
}
```

W rozdziale 9. Coordinate zdefiniowano jako typ referencyjny. Tu jest to typ bezpośredni (struct), ponieważ połączenie danych typów Longitude i Latitude można logicznie potraktować jako wartość, a jej wielkość jest mniejsza niż 16 bajtów (w rozdziale 9. w typie Coordinate używano typu Angle zamiast typów Longitude i Latitude). Innym czynnikiem wpływającym na zadeklarowanie typu Coordinate jako typu bezpośredniego jest to, że zawiera on (złożoną) wartość liczbową, na której wykonywane są określone operacje. Natomiast obiekty typów referencyjnych (na przykład typu Employee) nie są wartością liczbową, którą można manipulować; reprezentują one obiekty ze świata rzeczywistego.

Implementowanie metody Equals()

Aby ustalić, czy dwa obiekty są sobie równe (czyli czy mają te same identyfikujące je dane), należy wywołać metodę Equals(). W implementacji tej wirtualnej metody w typie object do ustalania równości używana jest metoda ReferenceEquals(). Ponieważ często nie jest to odpowiednie rozwiązanie, czasem trzeba przesłonić metodę Equals() bardziej odpowiednią implementacją.

Uwaga

Implementacja metody object.Equals() (czyli implementacja domyślna we wszystkich obiektach, w których metoda ta nie jest przesłonięta) wykorzystuje wyłącznie metodę ReferenceEquals().

Aby obiekty zostały uznane za *równe*, muszą mieć tę samą wartość identyfikujących je danych. Na przykład w typie ProductSerialNumber równe muszą być wartości pól ProductSeries, Model i Id. W typie Employee wystarczające może być porównanie pól EmployeeId. By poprawić implementację metody object.Equals(), trzeba ją przesłonić. W typach bezpośrednich należy w implementacji metody Equals() porównać wartości odpowiednich pól z danego typu.

Oto kroki, jakie należy wykonać w nowej wersji przesłanianej metody Equals():

1. Sprawdzenie, czy obiekt jest różny od null.
2. Sprawdzenie, czy typy do siebie pasują.
3. Wywołanie metody pomocniczej dla danego typu, która potrafi zinterpretować operand jako porównywanyą wartość rozwijanego typu, a nie jako wartość typu object (zobacz metodę Equals(Coordinate obj) na listingu 10.5).
4. Opcjonalne sprawdzenie równości skrótów, co pozwala przyspieszyć długie porównania kolejnych pól. Równe sobie obiekty nie mogą mieć innych skrótów.
5. Sprawdzenie wartości wywołania base.Equals().
6. Sprawdzenie, czy wszystkie pary identyfikujących pól mają te same wartości.
7. Przesłonięcie metody GetHashCode().
8. Przesłonięcie operatorów == i != (zobacz następny podrozdział).

Przykładową implementację metody Equals() pokazano na listingu 10.5.

Listing 10.5. Przesłanianie metody Equals()

```
public struct Longitude
{
    // ...
}

public struct Latitude
{
    // ...
}

public struct Coordinate
{
    public Coordinate(Longitude longitude, Latitude latitude)
    {
        Longitude = longitude;
        Latitude = latitude;
    }

    public Longitude Longitude { get; }
    public Latitude Latitude { get; }

    public override bool Equals(object? obj)
    {
        // KROK 1. Sprawdzenie, czy obiekt jest różny od null.
        if (obj is null)
        {
            return false;
        }

        // KROK 2. Sprawdzenie, czy typy pasują do siebie.
        // Można pominąć ten krok, jeśli typ jest zamknięty.
        if (GetType() != obj.GetType())
        {
            return false;
        }

        // Krok 3. Wywołanie pomocniczej wersji metody Equals() dla konkretnego typu.
        return Equals((Coordinate)obj);
    }

    public bool Equals(Coordinate obj)
    {
        // KROK 1. Sprawdzenie (w przypadku typów referencyjnych),
        // czy obiekt jest różny od null.
        // if (ReferenceEquals(obj, null))
        // {
        //     return false;
        // }

        // KROK 4. Opcjonalne sprawdzenie, czy skróty są identyczne.
        // Pomijane, gdy właściwości identyfikujące są modyfikowalne i skrót jest zapisywany.
        // if (GetHashCode() != obj.GetHashCode())
        // {
        //     return false;
        // }

        // KROK 5. Sprawdzenie wyniku wywołania base.Equals, jeśli
        // w klasie bazowej przesłonięta jest metoda Equals().
        // if (!base.Equals(obj))
        // {
        //     return false;
        // }
    }
}
```

```
// }

// KROK 6. Sprawdzenie, czy pola identyfikujące mają równą wartość.
// Tu używana jest wersja metody Equals z typów Longitude i Latitude.
return ( (Longitude.Equals(obj.Longitude)) &&
        (Latitude.Equals(obj.Latitude)) );
}

// KROK 7. Przesłonięcie metody GetHashCode.
public override int GetHashCode() { /* ... */ }
```

W tej implementacji dwa pierwsze testy są łatwe do zrozumienia. Warto jednak zwrócić uwagę na to, że można pominąć krok 2., jeśli typ jest zamknięty.

Kroki od 4. do 6. są wykonywane w nowej wersji przesłanianej metody `Equals()`. Ta nowa wersja przyjmuje obiekt typu `Coordinate`. Dzięki temu w trakcie porównywania dwóch obiektów typu `Coordinate` można uniknąć wykonywania metody `Equals(object? obj)` i testów wartości zwracanych przez metodę `GetType()`.

Ponieważ wynik zwracany przez metodę `GetHashCode()` nie jest zapisywany, a sama metoda nie jest bardziej wydajna niż kod z kroku 6., porównanie skrótów umieszczone w komentarzu. Ponieważ metoda `GetHashCode()` nie musi zwracać unikatowych wartości (pozwala jedynie określić, czy operandy są różne), nie umożliwia jednoznacznego wykrycia różnych obiektów. Ponadto nie należy porównywać skrótów, jeśli wartości identyfikujące są modyfikowalne i skróty są zapisywane. W przeciwnym razie porównanie takich samych obiektów może dać wartość `false`.

Jeśli metoda `base.Equals()` nie jest zaimplementowana, można pominąć krok 5. Jeżeli jednak programista doda później implementację tej metody, w kodzie zabraknie ważnego kroku. Dlatego powinieneś domyślnie go stosować.

Metoda `Equals()` (podobnie jak `GetHashCode()`) nigdy nie powinna zgłaszać wyjątków. Można porównać dowolny obiekt z dowolnym innym obiektem, a operacja ta nigdy nie powinna skutkować zgłoszeniem wyjątku.

Wskazówki

IMPLEMENTUJ metody `GetHashCode()`, `Equals()` i operatory `==` oraz `!=` w zestawie. Nie implementuj jednego z tych elementów bez trzech pozostałych.

STOSUJ ten sam algorytm w implementacji metody `Equals()` oraz operatorów `==` i `!=`.

UNIKAJ zgłaszania wyjątków w implementacjach metod `GetHashCode()` i `Equals()` oraz operatorów `==` i `!=`.

UNIKAJ przeciążania operatorów sprawdzania równości w modyfikowalnych typach referencyjnych oraz w sytuacjach, gdy nowa implementacja jest wyraźnie wolniejsza od domyślnej.

IMPLEMENTUJ wszystkie metody związane ze sprawdzaniem równości, gdy dodajesz implementację interfejsu `IEquatable`.

Przesłanianie funkcji GetHashCode() i Equals() z użyciem krotek

W poprzednich dwóch punktach pokazano, że implementacje funkcji `Equals()` i `GetHashCode()` są dość skomplikowane, jednak ich kod jest w dużej części szablonowy. W funkcji `Equals()` trzeba porównać wszystkie struktury danych identyfikujące obiekt, a jednocześnie unikać rekurencji nieskończonej i wyjątków związanych z referencjami o wartości `null`. W funkcji `GetHashCode()` należy uzyskać za pomocą operacji XOR unikatowy skrót wszystkich różnych od `null` struktur danych identyfikujących obiekt. Przy użyciu krotek z C# 7.0 okazuje się to całkiem proste.

Na potrzeby funkcji `Equals(Coordinate coordinate)` można zapisać w krotce wszystkie składowe identyfikujące obiekt i porównać je z argumentem tego samego typu:

```
public bool Equals(Coordinate? coordinate) =>
    return (Longitude, Latitude).Equals(
        (coordinate?.Longitude, coordinate?.Latitude));
```

Możliwe, że kod byłby bardziej czytelny, gdyby bezpośrednio porównywać każdą składową identyfikującą obiekt, pozostawiam to jednak ocenie czytelników. Wewnętrznie dla krotki (`System.ValueTuple<...>`) używany jest typ `EqualityComparer<T>`, wykorzystujący implementację interfejsu `IEquatable<T>` z parametrem określającym typ (ta implementacja obejmuje tylko jedną składową — `Equals<T>(T other)`). Dlatego aby poprawnie przesłonić funkcję `Equals`, należy zastosować się do wskazówki „IMPLEMENTUJ interfejs `IEquatable<T>`, gdy przesłaniasz funkcję `Equals()`”. Dzięki temu we własnych niestandardowych typach danych będziesz mógł używać własnej niestandardowej implementacji funkcji `Equals()` zamiast funkcji `Object.Equals()`.

Zapewne wygodniejsze jest wykorzystanie krotek w funkcji `GetHashCode()`. Zamiast stosować skomplikowane operacje XOR na składowych identyfikujących, które nie mają wartości `null`, wystarczy utworzyć krotkę z wszystkimi składowymi identyfikującymi obiekt i zwrócić wartość funkcji `GetHashCode()` wywołanej dla tej krotki:

```
public override int GetHashCode() =>
    return (Radius, StartAngle, SweepAngle).GetHashCode();
```

Warto zauważyć, że od wersji C# 7.3 krotki udostępniają operatory `==` i `!=` (te operatory powinny być dostępne w krotkach od momentu wprowadzenia tych ostatnich). Dalej znajdziesz omówienie przeciążania takich operatorów.

Przeciążanie operatorów

W poprzednim podrozdziale opisano przesłanianie metody `Equals()`. Zgodnie z jedną ze wskazówek obok metody `Equals()` należy też zaimplementować operatory `==` i `!=`. Implementowanie operatora nazywane jest jego *przeciążaniem*. W tym podrozdziale wyjaśniono, jak przeciązać `==` i `!=`, a także inne obsługiwane operatory.

Na przykład typ `string` udostępnia operator `+`, który scalą dwa łańcuchy znaków. Nie jest to zaskoczeniem, ponieważ `string` to predefiniowany typ, więc jego specjalna obsługa przez kompilator jest zrozumiała. Jednak C# umożliwia dodanie obsługi operatora `+` do różnych klas i struktur. Dołączyć można większość operatorów z wyjątkiem operatorów `x.y`, `f(x)`, `new`,

typeof, default, checked, unchecked, delegate, is, as, = i =>. Wartym uwagi operatorem, którego nie można samodzielnie zaimplementować, jest operator przypisania. Dlatego nie można zmienić działania operatora =.

Przed przejściem do implementowania przeciążonych operatorów warto wspomnieć, że takie operatory nie są wykrywane przez mechanizm IntelliSense. Dlatego jeśli nie chcesz utworzyć typu działającego jak typy proste (na przykład liczbowe), powinieneś unikać przeciążania operatorów.

Operatory porównania (==, !=, <, >, <=, >=)

Gdy przesłoniś metodę Equals(), może się pojawić pewna niespójność. Metoda Equals() wywołana dla dwóch obiektów może zwracać wartość true, natomiast operator == — wartość false (ponieważ domyślnie sprawdza równość ze względu na referencje). Aby rozwiązać ten problem, ważne jest, by przeciążyć także operatory równości (==) i nierówności (!=).

W większości sytuacji w implementacji tych operatorów można wykorzystać metodę Equals() (lub na odwrotnie). Jednak w typach referencyjnych niezbędne jest sprawdzenie na początku, czy obiekt jest różny od null (zobacz listing 10.6).

Listing 10.6. Implementowanie operatorów == i !=

```
public sealed class ProductSerialNumber
{
    // ...

    public static bool operator ==(ProductSerialNumber leftHandSide,
                                  ProductSerialNumber rightHandSide)
    {
        // Sprawdzanie, czy operand leftHandSide jest równy null.
        // Użycie operatora == spowodowałoby rekurencję.
        if (leftHandSide is null)
        {
            // Zwraca true, jeśli operand rightHandSide też jest równy null.
            // W przeciwnym razie zwracana wartość to false.
            return rightHandSide is null;
        }
        return leftHandSide.Equals(rightHandSide);
    }

    public static bool operator !=(ProductSerialNumber leftHandSide,
                                  ProductSerialNumber rightHandSide)
    {
        return !(leftHandSide == rightHandSide);
    }
}
```

Zauważ, że w tym przykładzie użyto typu ProductSerialNumber zamiast Coordinate, by zdemonstrować kod potrzebny dla typu referencyjnego. Kod dla takich typów jest bardziej skomplikowany, ponieważ trzeba uwzględnić wartość null.

Pamiętaj, by nie sprawdzać wartości null przy użyciu operatora równości (na przykład `leftHandSide == null`). Spowodowałoby to rekurencyjne wywołanie tego operatora, zapętlenie kodu i przepełnienie stosu. Aby uniknąć tego problemu, można użyć składni `is null` (od wersji C# 7.0) lub sprawdzić wartość null za pomocą wywołania metody `ReferenceEquals()`.

Uwaga

UNIKAJ stosowania operatora równości (==) w implementacji przeciążonej wersji tego operatora.

Operatory dwuargumentowe (+, -, *, /, %, &, |, ^, <<, >>)

Do obiektu typu `Coordinate` można dodać wartość typu `Arc`. Jednak kod obu typów nie obsługuje na razie operatora dodawania. Dlatego trzeba zdefiniować potrzebny operator, co pokazano na listingu 10.7.

Listing 10.7. Dodawanie operatora

```
struct Arc
{
    public Arc(
        Longitude longitudeDifference,
        Latitude latitudeDifference)
    {
        LongitudeDifference = longitudeDifference;
        LatitudeDifference = latitudeDifference;
    }

    public Longitude LongitudeDifference { get; }
    public Latitude LatitudeDifference { get; }
}

struct Coordinate
{
    // ...
    public static Coordinate operator +(
        Coordinate source, Arc arc)
    {
        Coordinate result = new Coordinate(
            new Longitude(
                source.Longitude + arc.LongitudeDifference),
            new Latitude(
                source.Latitude + arc.LatitudeDifference));
        return result;
    }
}
```

Operatory `+, -, *, /, %, &, |, ^, << i >>` są implementowane jako dwuargumentowe metody statyczne, w których przynajmniej jeden parametr ma typ obejmujący tę metodę. Nazwa metody to dany operator poprzedzony słowem kluczowym `operator`. Na listingu 10.8 dzięki zdefiniowaniu operatorów dwuargumentowych – i + można dodawać (odejmować)

obiekty typu Arc do (od) obiektów typu Coordinate. Zauważ, że należy zaimplementować operator + także dla typów Longitude i Latitude, ponieważ jest on używany w wyrażeniach source.Longitude + arc.LongitudeDifference i source.Latitude + arc.LatitudeDifference.

Listing 10.8. Wywoływanie operatorów dwuargumentowych – i +

```
public class Program
{
    public static void Main()
    {
        Coordinate coordinate1,coordinate2;
        coordinate1 = new Coordinate(
            new Longitude(48, 52), new Latitude(-2, -20));
        Arc arc = new Arc(new Longitude(3), new Latitude(1));

        coordinate2 = coordinate1 + arc;
        Console.WriteLine(coordinate2);

        coordinate2 = coordinate2 - arc;
        Console.WriteLine(coordinate2);

        coordinate2 += arc;
        Console.WriteLine(coordinate2);
    }
}
```

Wynik działania kodu z listingu 10.8 znajdziesz w danych wyjściowych 10.3.

DANE WYJŚCIOWE 10.3.

```
51° 52' 0 E -1° -20' 0 N
48° 52' 0 E -2° -20' 0 N
51° 52' 0 E -1° -20' 0 N
```

W typie Coordinate operatory – i + zwracają współrzędne po dodaniu lub odjęciu obiektu typu Arc. Dzięki temu można połączyć kilka operatorów i operandów w łańcuch, na przykład w wyrażeniu result = ((coordinate1 + arc1) + arc2) + arc3. Ponadto dzięki dodaniu obsługi tych samych operatorów (+ i -) w typie Arc (zobacz listing 10.9) można wyeliminować nawiasy. To podejście działa, ponieważ pierwszym operandem (wynikiem operacji arc1 + arc2) jest nowy obiekt typu Arc, który można dodać do następnego operandu typu Arc lub Coordinate.

Pomyśl teraz, co się stanie, jeśli udostępnisz operator – przyjmujący dwa parametry typu Coordinate i zwracający liczbę double, która reprezentuje odległość między dwoma współrzędnymi reprezentowanymi przez te parametry. Dodawanie wartości typu double do obiektu typu Coordinate jest niezdefiniowane, dlatego nie można połączyć operatorów i operandów w łańcuchu. Z tego względu należy zachować ostrożność w trakcie definiowania operatorów zwracających wartość innego typu; takie rozwiązanie jest nieintuicyjne.

Łączenie przypisania z operatorami dwuargumentowymi (`+=`, `-=`, `*=`, `/=`, `%=`, `&=`, ...)

Wcześniej wspomniano, że nie można przeciągać operatora przypisania. Jednak przeciążenie operatora dwuargumentowego skutkuje przeciążeniem połączenia operatora przypisania z danym operatorem dwuargumentowym (`+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=` lub `>>=`). Na podstawie definicji samego operatora dwuargumentowego (bez przypisania) język C# automatycznie umożliwia stosowanie kombinacji przypisania z wybranym operatorem. Dlatego używając definicji typu `Coordinate` z listingu 10.7, można zapisać następujące wyrażenie:

```
coordinate += arc;
```

Działa ono tak samo jak poniższy kod:

```
coordinate = coordinate + arc;
```

Logiczne operatory warunkowe (`&&`, `||`)

Logicznych operatorów warunkowych (podobnie jak operatora przypisania) nie można przeciągać bezpośrednio. Jednak ponieważ możliwe jest przeciążenie operatorów `&` i `|`, z których składają się operatory logiczne, możliwe jest również przeciążenie tych ostatnich. Wyrażenie `x && y` jest przetwarzane jako wyrażenie `x & y`, przy czym `y` jest sprawdzane tylko wtedy, gdy `x` jest równe `true`. Podobnie wyrażenie `x || y` jest przetwarzane jako `x | y`, gdzie `y` jest sprawdzane tylko wtedy, gdy `x` to `false`. Aby umożliwić sprawdzanie (na przykład w instrukcji `if`), czy obiekt danego typu ma wartość `true` lub `false`, trzeba przeciążyć operatory jednoargumentowe `true` i `false`.

Operatory jednoargumentowe (`+, -, !, ~, ++, --, true, false`)

Przeciążanie operatorów jednoargumentowych przebiega podobnie jak przeciążanie operatorów dwuargumentowych. Różnica polega na tym, że operatory jednoargumentowe przyjmują jeden parametr zawierającego je typu. Na listingu 10.9 przeciążono operatory `+ i -` w typach `Longitude` i `Latitude` oraz wykorzystano te operatory do przeciążenia ich w typie `Arc`.

Listing 10.9. Przeciążanie operatorów jednoargumentowych – `i +`

```
public struct Latitude
{
    // ...
    public static Latitude operator -(Latitude latitude)
    {
        return new Latitude(-latitude.DecimalDegrees);
    }
    public static Latitude operator +(Latitude latitude)
    {
        return latitude;
    }
}
public struct Longitude
{
```

```
// ...
public static Longitude operator -(Longitude longitude)
{
    return new Longitude(-longitude.DecimalDegrees);
}
public static Longitude operator +(Longitude longitude)
{
    return longitude;
}
}
public struct Arc
{
// ...
public static Arc operator -(Arc arc)
{
    // Wykorzystano tu jednoargumentowy operator — zdefiniowany
    // w typach Longitude i Latitude.
    return new Arc(-arc.LongitudeDifference,
        -arc.LatitudeDifference);
}
public static Arc operator +(Arc arc)
{
    return arc;
}
}
```

Operator + na tym listingu (podobnie jak w typach numerycznych) nie wykonuje żadnych operacji; udostępniono go tylko w celu zachowania symetrii.

Gdy przeciążane są operatory true i false, konieczne jest spełnienie dodatkowego wymagania — trzeba przeciążyć je oba. Nie można przeciążyć tylko jednego z nich. Sygnatury wyglądają tu podobnie jak w kontekście przeciążania innych operatorów, ale typem zwartanej wartości musi być bool, co pokazano na listingu 10.10.

Listing 10.10. Przeciążanie operatorów true i false

```
public static bool operator false(IsValid item)
{
    // ...
}
public static bool operator true(IsValid item)
{
    // ...
}
```

Typy z przeciążonymi operatorami true i false można stosować w wyrażeniach sterujących przepływem: if, do, while i for.

Operatory konwersji

Na razie w typach Longitude, Latitude i Coordinate nie dodano obsługi rzutowania na inne typy. Nie ma na przykład możliwości zrzutowania wartości typu double na instancję typów Longitude lub Latitude. Nie można też przypisać wartości do typu Coordinate za pomocą

typu `string`. Na szczęście C# udostępnia definicję metod przeznaczonych do obsługi konwersji danych z jednego typu na inny. W deklaracjach takich metod można określić, czy konwersja ma się odbywać niejawnie, czy jawnie.

ZAGADNIENIE DLA ZAAWANSOWANYCH

Operator rzutowania — ()

Implementowanie operatorów konwersji jawnnej i niejawnnej technicznie nie polega na przeciążaniu operatora rzutowania — (). Jednak w praktyce właśnie to się dzieje, dlatego implementowanie konwersji jawnnej lub niejawnnej często określa się mianem *definiowania operatora rzutowania*.

Definiowanie operatora konwersji przebiega podobnie jak definiowanie każdego innego operatora. Różnica polega na tym, że tu „operatorem” jest wynikowy typ konwersji. Ponadto słowo kluczowe `operator` pojawia się po słowie kluczowym określającym, czy konwersja ma przebiegać niejawnie, czy jawnie (zobacz listing 10.11).

Listing 10.11. Dodawanie obsługi konwersji niejawnnej między typami `Latitude` i `double`

```
public struct Latitude
{
    // ...

    public Latitude(double decimalDegrees)
    {
        DecimalDegrees = Normalize(decimalDegrees);
    }

    public double DecimalDegrees { get; }

    // ...

    public static implicit operator double(Latitude latitude)
    {
        return latitude.DecimalDegrees;
    }
    public static implicit operator Latitude(double degrees)
    {
        return new Latitude(degrees);
    }

    // ...
}
```

Z pomocą przedstawionych operatorów konwersji możesz przekształcać niejawnie wartości typu `double` na obiekty typu `Latitude` i wykonywać odwrotną operację. Jeśli podobna konwersja jest możliwa także w obiektach typu `Longitude`, możesz uprościć tworzenie obiektów typu `Coordinate`, podając w notacji dziesiętnej obie współrzędne (na przykład `coordinate = new Coordinate(43, 172);`).

Uwaga

Gdy implementujesz operator konwersji, typ parametru lub zwracanej wartości musi być typem zawierającym implementację. Pozwala to zachować hermetyzację kodu. Język C# nie umożliwia definiowania konwersji poza zasięgiem przekształcanego typu.

Wytyczne dotyczące operatorów konwersji

Różnica między definiowaniem operatorów niejawnej i jawniej konwersji polega na zapobieganiu przypadkowej niejawnej konwersji, która mogłaby prowadzić do niepożądanych efektów. Powinieneś pamiętać o dwóch możliwych konsekwencjach stosowania operatora jawniej konwersji. Po pierwsze, konwersja zgłaszająca wyjątki zawsze powinna być jawną. Na przykład wysoce prawdopodobne jest, że łańcuch znaków nie będzie miał formatu potrzebnego do przeprowadzenia konwersji z typu `string` na typ `Coordinate`. Ponieważ konwersja może się nie powieść, należy zdefiniować ją jako jawną. Programista musi wtedy celowo przeprowadzać konwersję i dbać o to, aby format łańcucha znaków był prawidłowy (inną możliwość to udostępnienie kodu do obsługi wyjątków). Często stosowany jest wzorzec, w którym konwersja w jedną stronę (z typu `string` na typ `Coordinate`) jest jawną, natomiast w drugą stronę (z typu `Coordinate` na typ `string`) niejawna.

Druga kwestia dotyczy tego, że czasem konwersja prowadzi do utraty danych. Konwersja z wartości 4.2 typu `float` na typ `int` jest w pełni poprawna, przy czym programista powinien mieć świadomość, że wartość ułamkowa z typu `float` zostanie utracona. Konwersja, która może prowadzić do utraty danych i nie pozwala na konwersję powrotną na wyjątkową wartość z typu pierwotnego, powinna być zdefiniowana jako jawną. Jeśli jawnie rzućowanie nieoczekiwanie skutkuje utratą danych lub błędem, możesz zgłaszać wyjątek `System.InvalidCastException`.

Wskazówki

NIE udostępniaj operatora konwersji niejawnej, jeśli konwersja może prowadzić do utraty danych.

NIE zgłaszaj wyjątków w konwersji niejawnej.

Wskazywanie innych podzespołów

Zamiast umieszczać cały kod w jednym monolitycznym pliku binarnym, w języku C# i platformie CLI można rozdzielić kod między wiele podzespołów. Dzięki temu podejściu można wielokrotnie wykorzystywać podzespoły w licznych plikach wykonywalnych.

ZAGADNIENIE DLA POCZĄTKUJĄCYCH

Biblioteki klas

Program `HelloWorld` to jedna z najprostszych aplikacji, jaką można napisać. W praktyce programy są dużo bardziej skomplikowane, a wraz ze wzrostem poziomu ich złożoności warto wprowadzić w nich uporządkowanie i dzielić kod na wiele części. W tym celu programiści przenoszą fragmenty programu do odrębnych kompilowanych jednostek nazywanych **bibliotekami klas** (lub po prostu **bibliotekami**). Programy następnie używają bibliotek i wykorzystują dostępne w nich potrzebne funkcje. Wartość tego rozwiązania jest taka, że dwie platformy mogą korzystać z tej samej biblioteki. W ten sposób funkcje biblioteki są dzielone między oba programy, co zmniejszałączną ilość potrzebnego kodu.

Oznacza to, że wystarczy napisać funkcję raz, umieścić ją w bibliotece klas i umożliwić wielu programom dołączenie tej funkcji za pomocą referencji do tej samej biblioteki. Na późniejszych etapach rozwijania kodu, gdy programiści naprawią błędy lub dodają funkcje do biblioteki, wszystkie programy mają dostęp do usprawnionych funkcji, ponieważ ciągle korzystają z poprawionej wersji biblioteki.

Pisany kod często jest przydatny w więcej niż jednym programie. Klasy `Longitude`, `Latitude` i `Coordinate` mogą być używane na przykład w programie do obsługi map i geokodowania zdjęć cyfrowych. Możesz też napisać klasę parsera instrukcji z wiersza polecen. Klasy i grupy klas tego rodzaju można napisać raz, a następnie stosować w wielu różnych programach. Dlatego takie klasy trzeba połączyć w podzespoł nazywany *biblioteką* lub *biblioteką klas* i pisać z myślą o wielokrotnym użytkowaniu (zamiast o zastosowaniu w jednym tylko programie).

W celu utworzenia biblioteki zamiast programu konsolowego wykonaj instrukcje z rozdziału 1., jednak użyj szablonu *Biblioteka klas* zamiast *Aplikacja konsoli*.

W środowisku Visual Studio 2019 wybierz opcję *Plik/Nowy/Projekt* (*Ctrl+Shift+N*) i użyj pola tekstowego *Wyszukaj szablony*, aby znaleźć wszystkie szablony z rodziny *Biblioteka klas*. Następnie wybierz opcję *Biblioteka klas (.NET Standard)* — naturalnie wersję dla języka Visual C#. Utwórz projekt o nazwie `GeoCoordinates`.

Następnie umieść kod źródłowy z listingu 10.9 w odrębnych plikach dla poszczególnych struktur. Pliki nazwij zgodnie z nazwami struktur i zbuduj projekt. Zbudowanie projektu spowoduje komplikację kodu w języku C# do postaci podzespołu — pliku `GeoCoordinates.dll` — i umieszczenie go w podkatalogu katalogu `.\bin\`.

Wskazywanie biblioteki

Gdy masz już bibliotekę, musisz dodać **referencję** do niej w programie (wskaź ją). W nowym programie konsolowym z klasą `Program` z listingu 10.8 trzeba dodać referencję do podzespołu `GeoCoordinates.dll`. Należy podać lokalizację biblioteki i metadane jednoznacznie identyfikujące bibliotekę w programie. Operacje te można wykonać na kilka sposobów. Pierwsza metoda to wskazanie pliku z projektem biblioteki (`*.csproj`). Pozwala to określić, który projekt zawiera kod źródłowy biblioteki. W ten sposób powstaje zależność między oboma projektami. Nie można wtedy skompilować programu używającego biblioteki bez

wcześniejowej komplikacji samej biblioteki. Ta zależność powoduje, że w momencie kompilowania programu kompilowana jest też biblioteka (jeśli nie jest jeszcze skompilowana).

Druga technika to wskazanie samego pliku z podzespołem. Innymi słowy, należy wskazać skompilowaną bibliotekę (plik *.dll) zamiast projektu. Ma to sens, gdy biblioteka jest kompilowana niezależnie od programu, na przykład przez inny zespół w organizacji.

Trzecia metoda to wskazanie pakietu NuGet, co opisano w następnym punkcie.

Warto zauważyc, że biblioteki i pakiety można wskazywać nie tylko w programach konsolowych. W każdym podzespołe można wskazać dowolny inny podzespoł. Często zdarza się, że jedna biblioteka używa innej biblioteki, przez co powstaje łańcuch zależności.

Wskaazywanie projektu lub biblioteki w Dotnet CLI

W rozdziale 1. opisano tworzenie programu konsolowego. W ten sposób powstał program zapisany w metodzie Main. Ta metoda jest punktem wejścia, od którego rozpoczyna się wykonywanie programu. Aby dodać referencję do nowo utworzonego podzespołu, kontynuujmy rozwój tego programu. Oto dodatkowe polecenie dodające referencję:

```
dotnet add .\HelloWorld\HelloWord.csproj package .\GeoCoordinates\bin\
↳Debug\netcoreapp2.0\GeoCoordinates.dll
```

Po argumencie add znajduje się ścieżka do skompilowanego podzespołu wskazywanego w projekcie.

Zamiast wskazywać podzespoły można podać plik projektu. Jak już wspomniałem, łączy to projekty ze sobą w taki sposób, że budowanie programu spowoduje wcześniejszą komplikację biblioteki klas (jeśli nie jest ona jeszcze skompilowana). Zaletą tej techniki jest to, że gdy program jest kompilowany, automatycznie znajdowany jest skompilowany podzespoł z biblioteką klas — na przykład w katalogu *debug* lub *release*. Oto polecenie wskazujące plik projektu:

```
dotnet add .\HelloWorld\HelloWord.csproj reference .\GeoCoordinates \
↳GeoCoordinates.csproj
```

Jeśli masz kod źródłowy biblioteki klas lub gdy ten kod często się zmienia, rozważ wskazywanie pliku projektu biblioteki zamiast skompilowanego podzespołu.

Po dodaniu referencji do projektu lub skompilowanego podzespołu projekt można skompilować, używając kodu źródłowego klasy Program z listingu 10.8.

Wskaazywanie projektu lub biblioteki w Visual Studio 2019

W rozdziale 1. opisano też tworzenie programu konsolowego za pomocą środowiska Visual Studio. Utworzony został tam program z funkcją Main. Aby dodać w nim referencję do projektu GeoCoordinates, wybierz opcję *Projekt/Dodaj odwołanie do projektu....*. Następnie w zakładce *Projekty\Rozwiążanie* wybierz projekt *GeoCoordinates* i kliknij OK, aby zatwierdzić wybór.

Dodawanie referencji do podzespołu wygląda podobnie. Wybierz opcję *Projekt/Dodaj odwołanie do projektu...*, jednak potem kliknij przycisk *Przeglądaj...* i znajdź podzespoły *GeoCoordinates.dll*.

Podobnie jak w Dotnet CLI możesz skompilować projekt programu za pomocą kodu źródłowego klasy Program z listingu 10.8.

Początek
4.0

Pakiety NuGet

Od wersji Visual Studio 2010 dostępny jest system NuGet służący do obsługi pakietów bibliotek. System ten ma umożliwiać łatwe współdzielenie bibliotek między projektami i firmami. Podzespoły biblioteki często obejmują więcej elementów niż jeden skompilowany plik. Używane mogą być pliki konfiguracyjne, dodatkowe zasoby i powiązane metadane. Do czasu wprowadzenia systemu NuGet nie istniał plik manifestu określający wszystkie zależności. Ponadto nie było standardowego dostawcy ani biblioteki pakietów, gdzie można było znaleźć wskazywane podzespoły.

System NuGet rozwiązuje oba te problemy. Nie tylko obejmuje manifest, który identyfikuje autorów, firmy, zależności itd., ale też zapewnia domyślnego dostawcę pakietów w postaci serwisu NuGet.org, gdzie pakiety mogą być przesypane, aktualizowane, indeksowane, a następnie pobierane przez projekty, które chcą wykorzystywać poszczególne pakiety. Za pomocą systemu NuGet można wskazać **pakiet NuGet (*.nupkg)**, a zostanie on automatycznie zainstalowany z wykorzystaniem jednego ze skonfigurowanych adresów URL dostawców pakietów.

Do pakietu NuGet dołączony jest manifest (plik **.nuspec*), który zawiera wszystkie dodatkowe dane dołączone do pakietu. Ponadto pakiet zawiera wszystkie dodatkowe zasoby — pliki z wersjami językowymi, pliki konfiguracyjne, pliki z materiałami itd. Pakiet NuGet jest archiwum zawierającym wszystkie zasoby połączone w jeden plik ZIP, przy czym ma on rozszerzenie *.nupkg*. Dlatego jeśli zmienisz rozszerzenie takiego archiwum na *.ZIP*, będziesz mógł je otworzyć i przejrzeć pliki za pomocą dowolnego standardowego programu kompresującego.

Początek
7.0

Referencje do pakietów NuGet w Dotnet CLI

Aby dodać pakiet NuGet do projektu w Dotnet CLI, należy wykonać jedno polecenie:

```
>dotnet add .\HelloWorld\HelloWorld.csproj package Microsoft.Extensions.Logging.Console
```

To polecenie sprawdza u wszystkich zarejestrowanych dostawców pakietów NuGet dostępność określonego pakietu i go pobiera. Pobieranie pakietu można też uruchomić bezpośrednio, za pomocą polecenia *dotnet restore*.

W celu utworzenia lokalnego pakietu NuGet zastosuj polecenie *dotnet pack*. Wygeneruje ono plik *GeoCoordinates.1.0.0.nupkg*, który można wskazać za pomocą instrukcji *add ... package*.

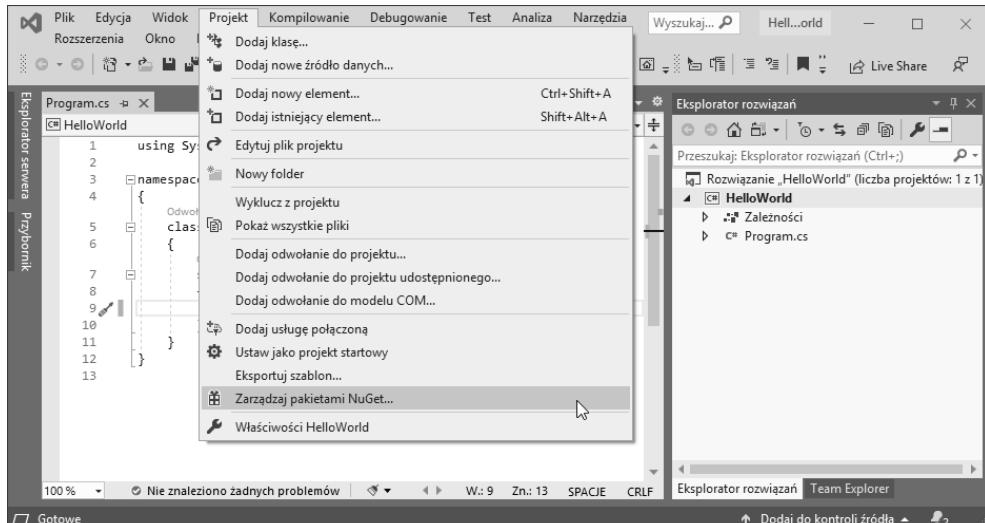
Cyfry widoczne po nazwie podzespołu oznaczają numer wersji pakietu. Aby bezpośrednio podać numer wersji, zmodyfikuj plik projektu (**.csproj*) i dodaj element potomny *<Version>...</Version>* w elemencie *PropertyGroup*.

Koniec
7.0

Dodawanie referencji do pakietów NuGet w Visual Studio 2019

Jeśli wykonałeś instrukcje z rozdziału 1., uzyskałeś już projekt HelloWorld. W Visual Studio 2019 możesz dodać pakiet NuGet do tego projektu w następujący sposób:

1. Wybierz opcję *Projekt/Zarządzaj pakietami NuGet* (zobacz rysunek 10.2).



Rysunek 10.2. Menu Projekt

2. Kliknij filtr *Przeglądaj* (zwykle aktywny jest filtr *Zainstalowane*, dlatego pamiętaj o wybraniu filtra *Przeglądaj*, aby dodać nowe referencje do pakietów), a następnie wpisz *Microsoft.Extensions.CommandLineUtils* w polu tekstowym *Wyszukaj* (*Ctrl+E*). Warto zauważyć, że także fragment nazwy, na przykład *CommandLineUtils*, pozwala przefiltrować listę (zobacz rysunek 10.3).

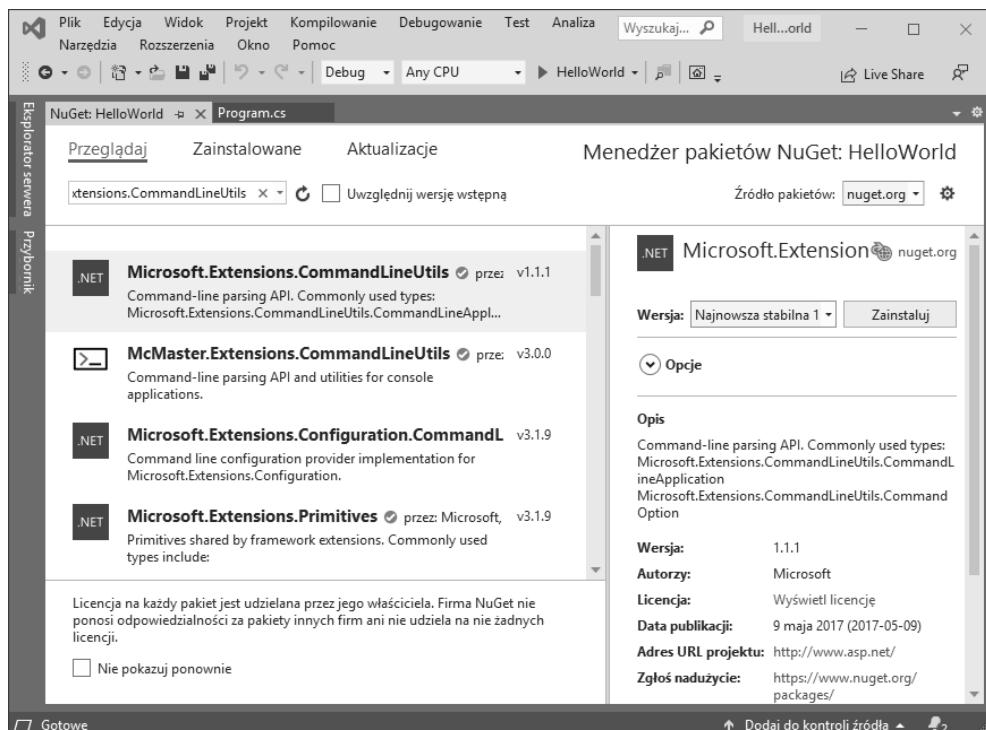
3. Kliknij przycisk *Zainstaluj*, aby zainstalować pakiet w projekcie.

Po wykonaniu tych kroków można zacząć korzystać z biblioteki *Microsoft.Extensions.Logging.Console* i powiązanych komponentów (automatycznie dodawanych do procesu).

W środowisku Visual Studio (podobnie jak w Dotnet CLI) można utworzyć własny pakiet NuGet. Służy do tego opcja *Kompilowanie/Spakuj element <nazwa projektu>*. Numer wersji pakietu można podać w zakładce *Pakiet* w oknie właściwości projektu.

Wywoływanie wskazywanego pakietu lub projektu

Po wskazaniu pakietu lub projektu można zacząć go używać w taki sam sposób, jakby jego kod źródłowy znajdował się w projekcie. Na listingu 10.12 pokazano na przykład, jak używać biblioteki *Microsoft.Extensions.Logging*. W danych wyjściowych 10.4 widoczne są przykładowe dane wyjściowe.



Rysunek 10.3. Filtr Przeglądaj

Listing 10.12. Wywoływanie pakietu NuGet

```
public class Program
{
    public static void Main(string[] args)
    {
        using ILoggerFactory loggerFactory =
            LoggerFactory.Create(builder =>
                builder.AddConsole()/*.AddDebug()*/);

        ILogger logger = loggerFactory.CreateLogger(
            categoryName: "Console");

        logger.LogInformation($"Kody triażu: = '{
            string.Join("", "", args)}'");
        // ...

        logger.LogWarning("To test kodów triażu...");

        // ...
    }
}
```

DANE WYJŚCIOWE 10.4.

```
>dotnet run -- czarny niebieski brązowy CBR pomarańczowy fioletowy czerwony żółty
info: Console[0]
      Kody triażu: = 'czarny', 'niebieski', 'brązowy', 'CBR',
'pomarańczowy', 'fioletowy', 'czerwony', 'żółty'
warn: Console[0]
      To test kodów triażu...
```

Pakiet NuGet z biblioteką Microsoft.Extensions.Logging.Console służy do wyświetlania danych w konsoli. Tu w konsoli pokazywane są zarówno komunikaty informacyjne, jak i komunikaty.

Jeśli dodasz też bibliotekę Microsoft.Extensions.Logging.Debug, będziesz mógł umieścić wywołanie .AddDebug() przed wywołaniem AddConsole() lub po nim. Wtedy komunikaty podobne do tych z danych wyjściowych 10.4 będą pojawiać się także w oknie danych wyjściowych w środowisku Visual Studio (opcja *Debugowanie/Okna/Dane wyjściowe*) lub Visual Studio Code (opcja *View/Debug Console*).

Z pakietem NuGet Microsoft.Extensions.Logging.Console powiązane są trzy zależności, w tym pakiet Microsoft.Extensions.Logging. Każdy z nich jest widoczny w węźle *Zależności/Pakiety* w oknie *Eksplorator rozwiązań* środowiska Visual Studio. Gdy dodasz pakiet NuGet, automatycznie dołączone zostaną też jego zależności.

Koniec
4.0

Hermetyzacja typów

Klasy służą do hermetyzacji operacji i danych, natomiast podzespoły umożliwiają hermetyzację grup typów. Programiści mogą podzielić system na podzespoły, a następnie korzystać z tych podzespołów w wielu aplikacjach lub zintegrować je z podzespołami pochodząymi z niezależnych źródeł.

Modyfikatory dostępu public i internal w deklaracjach typów

Klasa bez modyfikatora dostępu domyślnie jest definiowana jako wewnętrzna (`internal`)². Skutkuje to tym, że jest niedostępna spoza podzespołu. Nawet jeśli inny podzespoł korzysta z podzespołu zawierającego daną klasę, wszystkie klasy wewnętrzne z używanego podzespołu są niedostępne.

Słowa kluczowe `private` i `protected` określają poziom hermetyzacji składowych w klasie. Na podobnej zasadzie w C# można stosować modyfikatory dostępu na poziomie klasy, by kontrolować poziom hermetyzacji klas w podzespołach. Służą do tego modyfikatory dostępu `public` i `internal`. Aby udostępnić klasę poza podzespołem, należy ją opatrzyć modyfikatorem `public`. Dlatego przed skompilowaniem podzespołu `Coordinates.dll` należy dodać do deklaracji typów modyfikator `public` (zobacz listing 10.13).

² Nie dotyczy to typów zagnieżdżonych (są one domyślnie prywatne).

Listing 10.13. Zapewnianie dostępności typów poza podzespołem

```
public struct Coordinate
{
    // ...
}
public struct Latitude
{
    // ...
}
public struct Longitude
{
    // ...
}
public struct Arc
{
    // ...
}
```

Również deklaracje `class` i `enum` można opatrzyć modyfikatorami `public` lub `internal`³. Modyfikator dostępu `internal` można stosować nie tylko do deklaracji typów. Działa on także dla składowych typów. Możesz więc opatrzyć typ modyfikatorem `public`, ale do konkretnych metod z tego typu dodać modyfikator `internal`. Dzięki temu te metody będą dostępne tylko w danym podzespołe. Nie można jednak sprawić, by składowe były bardziej dostępne niż zawierający je typ. Jeśli klasa jest opatrzona modyfikatorem `internal`, składowe publiczne z tego typu będą dostępne tylko w ramach danego podzespołu.

Modyfikator `protected internal` dla składowych w typach

Innym modyfikatorem dostępu dla składowych w typach jest `protected internal`. Składowe opatrzone tym modyfikatorem są dostępne we wszystkich miejscach zawierającego je podzespołu *oraz* w klasach pochodnych od danego typu, nawet jeśli nie znajdują się one w tym samym podzespołe. Domyślnie składowe są prywatne, dlatego jeśli dodasz jakiś modyfikator (inny niż `public`), składowa stanie się dostępna w nieco większym stopniu.

Uwaga

Składowe opatrzone modyfikatorem `protected internal` są dostępne we wszystkich miejscach zawierającego je podzespołu *oraz* w klasach pochodnych od danego typu, nawet jeśli klasy te znajdują się w innym podzespołe.

³ Do klas zagnieżdżonych można dodać dowolny modyfikator dostępu stosowany do innych składowych klasy (na przykład modyfikator `private`). Jednak poza zasięgiem klasy jedyne obsługiwane modyfikatory dostępu to `public` i `internal`.

ZAGADNIENIE DLA POCZĄTKUJĄCYCH

Początek
7.2

Modyfikatory dostępu dla składowych w typach

Pełną listę modyfikatorów dostępu znajdziesz w tabeli 10.1.

Tabela 10.1. Modyfikatory dostępu

Modyfikator	Opis
public	Określa, że składowa jest dostępna wszędzie tam, gdzie dostępny jest dany typ.
internal	Składowa opatrzona tym modyfikatorem jest dostępna tylko w podzespołe.
private	Składowa jest dostępna wyłącznie w zawierającym ją typie.
protected	Składowa jest dostępna w zawierającym ją typie i w typach pochodnych (niezależnie od tego, czy znajdują się one w tym samym podzespołe).
protected internal	Składowa jest dostępna w zawierającym ją podzespołe oraz w typach pochodnych od zawierającego ją typu, nawet jeśli typy te znajdują się w innych podzespołach.
private protected	Składowa jest dostępna we wszystkich typach pochodnych od zawierającego ją typu, przy czym muszą znajdować się one w tym samym podzespołe. Tę możliwość dodano w C# 7.2.

Koniec
7.2

Definiowanie przestrzeni nazw

W rozdziale 2. wspomniano, że wszystkie typy danych są identyfikowane na podstawie połączenia przestrzeni nazw i nazwy typu. Jednak w środowisku CLR nie istnieje coś takiego jak „przestrzeń nazw”. Nazwą typu jest tam jego kompletna nazwa (włącznie z przestrzenią nazw). Klasy zdefiniowane wcześniej w książce nie miały jawnie zadeklarowanej przestrzeni nazw. Takie klasy automatycznie stają się elementami domyślnej globalnej przestrzeni nazw. Prawdopodobne jest, że nazwy takich klas spowodują konflikt (dzieje się tak, gdy programista próbuje zdefiniować dwie klasy o tej samej nazwie). Gdy korzystasz z innych podzespołów z niezależnych źródeł, prawdopodobieństwo wystąpienia konfliktu nazw rośnie jeszcze bardziej.

Ważniejsze jest jednak to, że w platformie CLI występują tysiące typów, a poza nią jest ich jeszcze więcej. Dlatego znalezienie właściwego typu do wykonywanego zadania może się okazać bardzo trudne.

Rozwiązywanie obu tych problemów polega na uporządkowaniu wszystkich typów poprzez pogrupowanie ich w logicznie powiązane kategorie — przestrzenie nazw. Na przykład klasy spoza przestrzeni nazw System zwykle są umieszczone w przestrzeni nazw odpowiadającej nazwie firmy, nazwie produktu lub obu tym cechom. Klasy z wydawnictwa Addison-Wesley trafiają więc do przestrzeni nazw Awl i AddisonWesley, a klasy opracowane przez Microsoft (oprócz tych z przestrzeni System) — do przestrzeni nazw Microsoft. Drugim członem nazwy przestrzeni nazw powinna być stabilna nazwa produktu, która nie zmienia się między wersjami. Stabilność jest zresztą ważna na wszystkich poziomach. Modyfikowanie nazw przestrzeni to zmiana naruszająca zgodność między wersjami kodu, dlatego należy się tego wystrzegać.

Z tego powodu warto unikać stosowania w nazwach określeń (jednostek w hierarchii firmy, zmieniających się marek itd.), które nie są stałe.

Przestrzenie nazw należy zapisywać w Notacji Pascalowej. Jeśli jednak nazwa marki jest zapisana w nietypowy sposób, akceptowalne jest wykorzystanie niestandardowej notacji. Ważna jest jednak konsekwencja, dlatego jeśli masz wątpliwości, wybierz rozwiązanie, które pozwoli uzyskać większą spójność. Aby utworzyć przestrzeń nazw i przypisać do niej klasę, należy się posłużyć słowem kluczowym `namespace` (co pokazano na listingu 10.14).

Listing 10.14. Definiowanie przestrzeni nazw

```
// Definicja przestrzeni nazw AddisonWesley.  
namespace AddisonWesley  
{  
    class Program  
    {  
        // ...  
    }  
}  
// Koniec deklaracji przestrzeni nazw AddisonWesley.
```

Cały kod w nawiasie klamrowym po deklaracji przestrzeni nazw będzie należał do danej przestrzeni. Na przykład na listingu 10.14 klasa `Program` jest umieszczona w przestrzeni nazw `AddisonWesley`, dlatego pełna nazwa tej klasy to `AddisonWesley.Program`.

Uwaga

W środowisku CLR nie ma czegoś takiego jak „przestrzeń nazw”. Nazwą typu jest jego kompletna nazwa z kwalifikatorem.

Przestrzenie nazw (podobnie jak klasy) można zagnieździć. Pozwala to uporządkować klasy w hierarchii. Na przykład w przestrzeni nazw `System` wszystkie klasy związane z interfejsami API do obsługi sieci znajdują się w przestrzeni `System.Net`, a klasy związane z internetem są zapisane w przestrzeni `System.Web`.

Przestrzenie nazw można zagnieździć na dwa sposoby. Pierwszy z nich polega na umieszczeniu jednej przestrzeni nazw w innej (podobnie jak w przypadku klas), co pokazano na listingu 10.15.

Listing 10.15. Zagnieźdzanie jednej przestrzeni nazw w innej

```
// Definicja przestrzeni nazw AddisonWesley.  
namespace AddisonWesley  
{  
    // Definicja przestrzeni nazw AddisonWesley.Michaelis.  
    namespace Michaelis  
    {  
        // Definicja przestrzeni nazw  
        // AddisonWesley.Michaelis.EssentialCSharp.  
        namespace EssentialCSharp
```

```

{
    // Deklaracja klasy
    // AddisonWesley.Michaelis.EssentialCSharp.Program.
    class Program
    {
        // ...
    }
}

// Koniec deklaracji przestrzeni nazw AddisonWesley.

```

Zagnieżdżenia spowodują tu umieszczenie klasy Program w przestrzeni nazw AddisonWesley.

→ Michaelis.EssentialCSharp.

Drugie podejście polega na użyciu pełnej nazwy przestrzeni nazw w deklaracji określonej przestrzeni. Poszczególne identyfikatory należy wtedy rozdzielić kropkami, co pokazano na listingu 10.16.

Listing 10.16. Zagnieżdżanie przestrzeni nazw przy użyciu kropki rozdzielającej poszczególne identyfikatory

```

// Definicja przestrzeni nazw AddisonWesley.Michaelis.EssentialCSharp.
namespace AddisonWesley.Michaelis.EssentialCSharp
{
    class Program
    {
        // ...
    }
}

// Koniec deklaracji przestrzeni nazw AddisonWesley.

```

Niezależnie od tego, czy w deklaracji przestrzeni nazw stosowany jest schemat z listingu 10.15, czy z listingu 10.16 (lub połączenie ich obu), wynikowy kod CIL będzie identyczny. Tę samą przestrzeń nazw można stosować wielokrotnie w różnych plikach, a nawet w różnych podzespołach. Na przykład gdy zachowana jest relacja jeden do jednego między plikami i klasami, można każdą klasę zdefiniować w odrębnym pliku i umieścić poszczególne klasy w deklaracjach tej samej przestrzeni nazw.

Ponieważ przestrzeń nazw to najważniejszy mechanizm służący do porządkowania typów, często warto go wykorzystać do zarządzania wszystkimi plikami klas. Pomoże jest utworzenie katalogów przeznaczonych na poszczególne przestrzenie nazw. Można wtedy na przykład umieścić klasę AddisonWesley.Fezzik.Services.RegistrationService w hierarchii katalogów w folderze odpowiadającym tej nazwie.

Jeśli w projektach w środowisku Visual Studio nazwa projektu to AddisonWesley.Fezzik, można utworzyć podkatalog o nazwie *Services* i umieścić w nim plik *RegistrationService.cs*. Następnie można dodać nowy katalog, na przykład *Data*, i zapisać w nim klasy związane z jednostkami używanymi w programie (takie jak *RealestateProperty*, *Buyer* i *Seller*).

Wskazówka

POPRAZDJAJ przestrzenie nazw nazwą własnej firmy, by odróżnić je od takich samych przestrzeni nazw innych organizacji.

STOSUJ stabilne, niezależne od wersji nazwy produktów jako drugi człon nazw przestrzeni nazw.

NIE definiuj typów bez umieszczenia ich w przestrzeni nazw.

ROZWAŻ utworzenie struktury katalogów odpowiadającej hierarchii przestrzeni nazw.

Komentarze XML-owe

W rozdziale 1. wstępnie opisano komentarze. Komentarze XML-owe można wykorzystać nie tylko do pisania notatek dla innych programistów przeglądających kod źródłowy. Takie komentarze są oparte na technice spopularyzowanej w Javie. Choć kompilator języka C# ignoruje wszystkie komentarze w trakcie generowania wynikowego programu wykonywalnego, programista może wykorzystać opcje w wierszu poleceń do poinformowania kompilatora⁴ o tym, że komentarze XML-owe należy umieścić w odrębnym pliku XML. W ten sposób programista może wygenerować dokumentację interfejsu API na podstawie komentarzy XML-owych. Ponadto edytory języka C# potrafią przetwarzanie komentarze XML-owe z kodu i wyświetlać je jako odrębne obszary (na przykład w kolorze innym niż używany dla reszty kodu), a także przetwarzanie fragmentów danych z takich komentarzy i pokazywać je programistom.

Na rysunku 10.4 pokazano, jak środowisko IDE może wykorzystać komentarze XML-owe do wyświetlenia programiście pomocnej wskazówki na temat pisanego kodu. Takie wskazówki są bardzo przydatne w dużych programach (zwłaszcza gdy nad kodem pracuje wiele osób). Aby wyświetlanie takich informacji było możliwe, programista musi poświęcić czas na dodanie komentarzy XML-owych do kodu i nakazać kompilatorowi utworzenie pliku XML. W następnym podrozdziale wyjaśniono, jak to zrobić.

The screenshot shows a code editor in Visual Studio with the following code:

```

/// <summary>
/// Wyświetla podany tekst w <strong>konsoli</strong>.
/// </summary>
/// <param name="text">Wyświetlany tekst.</param>

odwołanie
private static void Display(string text)
{
    Console.WriteLine(text);
}

Odwzorowanie: 0
static void Main()
{
    Display("Inigo Montoya");
}

```

A tooltip is displayed over the call to `Display("Inigo Montoya");`, containing the XML comment from the `Display` method:

void Program.Display(string text)
Wyświetla podany tekst w konsoli.

Rysunek 10.4. Komentarze XML-owe jako podpowiedzi w środowisku IDE Visual Studio

⁴ W standardzie języka C# nie jest określone, czy to kompilator języka C#, czy osobne narzędzie ma odpowiadać za wyodrębnianie danych w formacie XML. Jednak wszystkie podstawowe kompilatory języka C# udostępniają potrzebny mechanizm za pomocą opcji ustawianej w czasie komplikacji, a nie przy użyciu dodatkowego narzędzia.

Od wersji Visual Studio 2019 można też umieszczać w komentarzach prosty kod w HTML-u uwzględniany w wyświetlanych podpowiedziach. Na przykład, jeśli umieścisz słowo „konsoli” między znacznikami `` i ``, wyraz ten będzie wyświetlany pogrubioną czcionką (zobacz rysunek 10.4).

Wiązanie komentarzy XML-owych z konstrukcjami programistycznymi

Przyjrzyj się kodowi klasy `DataStorage` pokazanemu na listingu 10.17.

Listing 10.17. Opisywanie kodu za pomocą komentarzy XML-owych

```
/// <summary>
/// Typ DataStorage służy do utrwalania i pobierania
/// zapisywanych w pliku danych o pracownikach.
/// </summary>
class DataStorage
{
    /// <summary>
    /// Zapisuje obiekt employee do pliku
    /// o nazwie odpowiadającej nazwisku i imieniu pracownika.
    /// </summary>
    /// <remarks>
    /// Ta metoda używa typu
    /// <seealso cref="System.IO.FileStream"/>
    /// oraz typu
    /// <seealso cref="System.IO.StreamWriter"/>
    /// </remarks>
    /// <param name="employee">
    /// Utrwalany w pliku obiekt typu Employee</param>
    /// <date>1 stycznia 2000</date>
    public static void Store(Employee employee)
    {
        // ...
    }

    /**
     * <summary>
     * Wczytuje obiekt typu Employee
     * </summary>
     * <remarks>
     * Ta metoda używa typu
     * <seealso cref="System.IO.FileStream"/>
     * i typu
     * <seealso cref="System.IO.StreamReader"/>
     * </remarks>
     * <param name="firstName">
     * Imię pracownika</param>
     * <param name="lastName">
     * Nazwisko pracownika</param>
     * <returns>
     * Obiekt typu Employee z imieniem i nazwiskiem pracownika
     * </returns>
     * <date>1 stycznia 2000</date> ***/
    public static Employee Load(
        string firstName, string lastName)
    {
        // ...
    }
}
```

Jednowierszowe komentarze XML-owe

Komentarz XML-owy z ogranicznikami (od wersji C# 2.0)

Komentarz XML-owy z ogranicznikami (od wersji C# 2.0)

```

    }
}

class Program
{
    // ...
}

```

Na listingu 10.17 zastosowano zarówno XML-owe komentarze z ogranicznikami, obejmujące wiele wierszy, jak i jednowierszowe komentarze XML-owe, w których w każdym wierszu potrzebna jest sekwencja trzech ukośników (///).

Ponieważ komentarze XML-owe służą do dokumentowania interfejsu API, powinny być używane w języku C# tylko do deklaracji (na przykład do deklaracji klasy i metody na listingu 10.17). Próba umieszczenia komentarza XML-owego wewnętrznie w kodzie (niezależnie od deklaracji) prowadzi do zgłoszenia ostrzeżenia przez kompilator. Kompilator łączy daną jednostkę z jej opisem tylko wtedy, gdy komentarz XML-owy występuje bezpośrednio przed deklaracją tej jednostki.

Choć język C# umożliwia stosowanie w komentarzach dowolnych znaczników XML-owych, w standardzie języka jawnie zdefiniowany jest zestaw używanych znaczników. Kod <seealso href="System.IO.StreamWriter"/> ilustruje, jak zastosować znacznik `seealso`. Ten znacznik tworzy w tekście odnośnik do klasy `System.IO.StreamWriter`.

Koniec
2.0

Generowanie pliku z dokumentacją w formacie XML

Kompilator sprawdza, czy komentarze XML-owe są dobrze uformowane, a jeśli nie są, zgłasza ostrzeżenie. Aby wygenerować plik XML, należy dodać podelement `DocumentationFile` do elementu `ProjectProperties`:

```
<DocumentationFile>$($ŚcieżkaWyjściowa)\$(DocelowaPlatforma)\$(NazwaPodzespołu).
➥xml</DocumentationFile>
```

Ten podelement powoduje wygenerowanie w trakcie budowania kodu pliku XML w katalogu wyjściowym. Nazwą tego pliku jest `<nazwapodzespołu>.xml`. Za pomocą opisanej wcześniej klasy `CommentSamples` i podanych tu opcji kompilatora uzyskasz plik `CommentSamples.xml` pokazany na listingu 10.18:

Listing 10.18. Plik `Comments.xml`

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>DataStorage</name>
    </assembly>
    <members>
        <member name="T:DataStorage">
            <summary>
                Typ DataStorage służy do utrwalania i pobierania
                zapisywanych w pliku danych o pracownikach.
            </summary>
        </member>
        <member name="M:DataStorage.Store(Employee)">
            <summary>
```

```

Zapisuje obiekt employee do pliku
o nazwie odpowiadającej nazwisku i imieniu pracownika.
</summary>
<remarks>
Ta metoda używa typu
<seealso cref="T:System.IO.FileStream"/>
i typu
<seealso cref="T:System.IO.StreamWriter"/>
</remarks>
<param name="employee">
Utrwalany w pliku obiekt typu Employee</param>
<date>1 stycznia 2000</date>
</member>
<member name="M:DataStorage.Load(
System.String, System.String)">
<summary>
Wczytuje obiekt typu Employee
</summary>
<remarks>
Ta metoda używa typu
<seealso cref="T:System.IO.FileStream"/>
oraz typu
<seealso cref="T:System.IO.StreamReader"/>
</remarks>
<param name="firstName">
Imię pracownika</param>
<param name="lastName">
Nazwisko pracownika</param>
<returns>
Obiekt typu Employee z imieniem i nazwiskiem pracownika
</returns>
<date>1 stycznia 2000</date>*
</member>
</members>
</doc>

```

Wynikowy plik obejmuje tylko tyle metadanych, ile potrzeba do powiązania elementów z odpowiadającymi im deklaracjami z kodu w języku C#. Warto zwrócić na to uwagę, ponieważ zwykle dane wyjściowe w formacie XML trzeba stosować z wygenerowanym podzespołem, by uzyskać sensowną dokumentację. Na szczęście istnieją narzędzia przeznaczone do generowania dokumentacji (na przykład bezpłatny program GhostDoc⁵ i projekt o otwartym dostępie do kodu źródłowego NDoc⁶).

Wskazówka

UDOSTĘPNIJ komentarze XML-owe powiązane z publicznymi interfejsami API, gdy dzięki temu możesz zapewnić bogatszy kontekst niż za pomocą samych sygnatur takich interfejsów. W ten sposób możesz udostępnić opisy składowych i parametrów oraz przykłady korzystania z danego interfejsu API.

⁵ Więcej informacji o narzędziu GhostDoc znajdziesz na stronie <http://submain.com/>.

⁶ Więcej informacji o narzędziu NDoc znajdziesz na stronie <http://ndoc.sourceforge.net>.

Odzyskiwanie pamięci

Odzyskiwanie pamięci to jedna z podstawowych funkcji środowiska uruchomieniowego. Mechanizm ten ma przywracać zajmowaną przez obiekty pamięć, gdy nie ma już żadnych referencji do niej. Najważniejsze w tym opisie są pamięć i referencje. Mechanizm odzyskiwania pamięci odpowiada jedynie za przywracanie pamięci. Nie obsługuje innych zasobów, takich jak połączenia z bazą danych, uchwyty (do plików, okien itd.), porty sieciowe i urządzenia (na przykład porty szeregowe). Ten mechanizm określa pamięć, którą należy odzyskać, na podstawie tego, czy istnieją prowadzące do niej referencje. Oznacza to, że mechanizm odzyskiwania pamięci działa tylko dla obiektów typów referencyjnych i przywraca wyłącznie pamięć ze sterty. Z opisu wynika też, że przechowywanie referencji do obiektu opóźnia moment, w którym mechanizm przywróci pamięć zajmowaną przez obiekt.

ZAGADNIENIE DLA ZAAWANSOWANYCH

Odzyskiwanie pamięci w platformie .NET

Wiele szczegółowych informacji związanych z mechanizmem odzyskiwania pamięci dotyczy konkretnych implementacji interfejsu CLI, dlatego może się zmieniać. W tym podrozdziale opisano implementację z platformy .NET, ponieważ to ona jest najczęściej spotykana.

W platformie .NET mechanizm odzyskiwania pamięci wykorzystuje algorytm „znakowania i kompaktowania” (ang. *mark-and-compact*). Na początku iteracji mechanizm znajduje wszystkie **podstawowe referencje** do obiektów. Podstawowe referencje to wszystkie referencje w zmiennych statycznych, rejestrach procesora, zmiennych lokalnych i instancjach parametrów (czyli w opisanych w dalszej części podrozdziału obiektach zapisywanych w kolejce finalizacji). Na podstawie tej listy mechanizm odzyskiwania pamięci przechodzi po drzewie wyznaczanym przez każdą referencję podstawową i rekurencyjnie sprawdza wszystkie obiekty, do których prowadzi dana referencja. W ten sposób powstaje graf obejmujący wszystkie obiekty, do których można dotrzeć.

Zamiast tworzyć listę wszystkich niedostępnych obiektów, mechanizm odzyskiwania pamięci przywraca ją w wyniku umieszczenia wszystkich dostępnych obiektów jeden obok drugiego (na tym polega kompaktowanie). To powoduje, że pamięć zajmowana przez niedostępne obiekty (uznawane za „śmieci”) zostaje zastąpiona innymi danymi.

Wyszukiwanie i przenoszenie wszystkich dostępnych obiektów wymaga, by system zachował stały stan w trakcie działania mechanizmu odzyskiwania pamięci. Aby możliwe było osiągnięcie tego celu, wszystkie wątki zarządzane w procesie są w czasie odzyskiwania pamięci wstrzymywane. Może to prowadzić do krótkich przerw w pracy aplikacji. Zwykle są one nieodczuwalne, chyba że konieczny jest wyjątkowo długi cykl odzyskiwania pamięci. Aby zmniejszyć prawdopodobieństwo wystąpienia cyklu odzyskiwania pamięci w niewygodnym momencie, obiekt `System.GC` udostępnia metodę `Collect()`, którą można wywołać bezpośrednio przed kodem o krytycznym znaczeniu. Wprawdzie nie zapobiega to późniejszemu uruchomieniu mechanizmu odzyskiwania pamięci, ale zmniejsza prawdopodobieństwo tej operacji (chyba że kod o krytycznym znaczeniu intensywnie korzysta z pamięci).

Zaskakującym aspektem działania mechanizmu odzyskiwania pamięci w platformie .NET jest to, że w trakcie jego pracy nie zawsze przywracana jest cała pamięć. Z analizy czasu życia obiektów wynika, że częściej potrzebne jest odzyskiwanie pamięci niedawno utworzonych obiektów niż obiektów używanych już przez długi czas. Na podstawie tego zjawiska mechanizm odzyskiwania pamięci w platformie .NET działa dla różnych generacji obiektów w inny sposób — próbuje odzyskiwać pamięć niedawno utworzonych obiektów częściej niż w przypadku obiektów, które nie zostały usunięte w poprzednich cyklach. Obiekty są przypisywane do trzech generacji. Za każdym razem, gdy obiekt przetrwa cykl odzyskiwania pamięci, jest przenoszony do następnej generacji, aż trafi do generacji o numerze 2 (numeracja rozpoczyna się od zera). Odzyskiwanie pamięci jest przeprowadzane częściej dla obiektów z generacji 0 niż dla obiektów z generacji 2.

Mimo obaw, jakie wzbudzały pierwsze wersje beta platformy .NET (gdy porównywano je do pracy kodu niezarządzanego), używany w niej mechanizm odzyskiwania pamięci okazał się bardzo wydajny. Co ważniejsze, zyski wynikające z wyższej produktywności programistów znacznie przewyższają koszty pracy w nielicznych sytuacjach, w których trzeba zrezygnować z kodu zarządzanego, by zoptymalizować niektóre algorytmy.

Słabe referencje

Wszystkie opisywane do tej pory referencje były **mocnymi referencjami**. Takie referencje pozwalają zachować dostępność obiektu i zapobiegają przywróceniu pamięci zajmowanej przez ten obiekt. Platforma umożliwia też tworzenie **słabych referencji**. Nie blokują one uruchomienia mechanizmu odzyskiwania pamięci dla danego obiektu, ale pozwalają zachować referencję. Dlatego jeśli mechanizm odzyskiwania pamięci nie usunie obiektu, będzie można go ponownie wykorzystać.

Słabe referencje są przeznaczone dla obiektów, których tworzenie i utrzymywanie w systemie jest kosztowne. Wyobraź sobie na przykład długą listę obiektów wczytywanych z bazy danych i wyświetlanych użytkownikowi. Proces wczytywania listy może być kosztowny, dlatego gdy użytkownik ją zamknie, należy ją udostępnić mechanizmowi odzyskiwania pamięci. Jeśli jednak użytkownik wielokrotnie żąda listy, konieczne będzie ponowne jej kosztowne wczytanie. Gdy używana jest słaba referencja, kod może sprawdzić, czy lista wciąż jest dostępna. Jeśli mechanizm odzyskiwania pamięci jeszcze jej nie usunął, można ponownie wykorzystać tę samą listę. Słabe referencje działają więc jak pamięć podrzczna obiektów. Obiekty z pamięci podrzcznej są szybko dostępne, jeśli jednak mechanizm odzyskiwania pamięci przywrócił już pamięć zajmowaną przez dany obiekt, trzeba będzie ten obiekt utworzyć od nowa.

Gdy programista uzna, że dla obiektu (lub dla kolekcji obiektów) warto utworzyć słabą referencję, należy zastosować typ `System.WeakReference` (zobacz listing 10.19).

Listing 10.19. Używanie słabych referencji

```
public static class ByteArrayDataSource
{
    static private byte[] LoadData()
    {
        // Wyobraź sobie użycie znacznie większej liczby.
```

```

byte[] data = new byte[1000];
// Wczytywanie danych.
// ...
return data;
}

static private WeakReference<byte[]>? Data { get; set; }

static public byte[] GetData()
{
    byte[]? target;
    if (Data is null)
    {
        target = LoadData();
        Data = new WeakReference<byte[]>(target);
        return target;
    }
    else if (Data.TryGetTarget(out target))
    {
        return target;
    }
    else
    {
        // Ponowne wczytywanie danych i przypisywanie ich do zmiennej (co powoduje utworzenie
        // silnej referencji) przed przypisaniem ich do właściwości Target słabej referencji
        // i zwróceniem ich.
        target = LoadData();
        Data.SetTarget(target);
        return target;
    }
}
// ...

```

W tym kodzie używane są typy generyczne opisane w tej książce dopiero w rozdziale 12. Jednak możesz zignorować człon `<byte[]>` w deklaracji właściwości `Data` i w przypisaniu do niej wartości. Choć istnieje także niegeneryczna wersja typu `WeakReference`, trudno znaleźć powód do jej stosowania⁷.

Większość logiki znajduje się w metodzie `GetData()`. Ma ona zawsze zwracać instancję danych — albo z pamięci podręcznej, albo po ponownym ich wczytaniu. Metoda `GetData()` najpierw sprawdza, czy właściwość `Data` ma wartość `null`. Jeśli tak jest, dane są wczytywane i przypisywane do zmiennej lokalnej `target`. To powoduje utworzenie referencji do danych, dzięki czemu mechanizm przywracania pamięci ich nie usunie. Następnie kod tworzy obiekt typu `WeakReference` i przekazuje referencję prowadzącą do wczytanych danych, dzięki czemu obiekt ma dostęp do danych (są to jego docelowe dane). Następnie na żądanie można zwrócić instancję z danymi. Nie przekazuj do obiektu typu `WeakReference` instancji, dla której nie istnieje lokalna referencja, ponieważ dane mogą zostać usunięte, zanim program będzie miał możliwość ich zwrócenia (nie używaj więc wywołań o składni `new WeakReference<byte[]>(LoadData())`).

⁷ Chyba że używasz platformy .NET 4.5 lub starszej.

Jeśli właściwość `Data` jest już powiązana z obiektem typu `WeakReference`, kod może wywołać metodę `TryGetTarget()` i jeżeli dostępny jest obiekt, przypisać wartość do zmiennej `target`. Powstaje wtedy referencja, a mechanizm przywracania pamięci nie usunie potrzebnych danych.

Ostatnia możliwość to zwrócenie `false` przez metodę `TryGetTarget()` z typu `WeakReference`. Wtedy należy wczytać dane, przypisać referencję za pomocą wywołania `SetTarget()` i zwrócić nowy obiekt.

Porządkowanie zasobów

Odzyskiwanie pamięci to jedno z podstawowych zadań środowiska uruchomieniowego. Należy jednak pamiętać, że mechanizm odzyskiwania pamięci dotyczy głównie pamięci wykorzystywanej przez kod, nie służy natomiast do porządkowania uchwytów do plików, połączeniowych łańcuchów znaków, portów i innych zasobów o ograniczonej dostępności.

Finalizatory

Finalizatory umożliwiają programistom pisanie kodu, który porząduje zasoby klasy. Finalizatory (w odróżnieniu od konstruktorów, wywoływanych jawnie za pomocą operatora `new`) nie mogą być wywoływanego jawnie. Nie istnieje odpowiednik operatora `new` (na przykład `delete`). To mechanizm odzyskiwania pamięci odpowiada za wywołanie finalizatora dla obiektu. Dlatego programista nie może na etapie komplikacji precyzyjnie określić momentu wywołania finalizatora. Wie tylko, że finalizator zostanie uruchomiony między momentem ostatniego użycia obiektu a — zwykle — normalnym zamknięciem aplikacji. Słowo „zwykle” jest tu użyte celowo, aby podkreślić niepewność tego, czy finalizator zostanie uruchomiony. Jeśli proces zostanie zakończony w niestandardowy sposób, finalizator może nie zostać wywołany. Na przykład wyłączenie komputera lub wymuszone zamknięcie procesu uniemożliwiają uruchomienie finalizatora. Jednak w .NET Core nawet w standardowych warunkach finalizatory mogą nie zostać wykonane przed zamknięciem aplikacji. W kolejnym punkcie dowiesz się, że konieczne może być wykonanie dodatkowych operacji w celu zarejestrowania finalizacji w innych mechanizmach.

Uwaga

W czasie komplikacji nie można precyzyjnie określić, kiedy finalizator zostanie uruchomiony.

Deklaracja finalizatora wygląda identycznie jak deklaracja destruktora w języku C++, czyli w poprzedniku języka C#. Na listingu 10.20 pokazano, że w deklaracji finalizatora nazwa klasy jest poprzedzana tyldą.

Listing 10.20. Definiowanie finalizatora

```
using System.IO;

public class TemporaryFileStream
{
    public TemporaryFileStream(string fileName)
    {
        File = new FileInfo(fileName);
        // Lepszym rozwiązaniem jest użycie wywołania FileMode.DeleteOnClose.
        Stream = new FileStream(
            File.FullName, FileMode.OpenOrCreate,
            FileAccess.ReadWrite);
    }

    public TemporaryFileStream()
        : this(Path.GetTempFileName()) { }

    // Finalizator.
    ~TemporaryFileStream()
    {
        try
        {
            Close();
        }
        catch(Exception exception)
        {
            // Zapisywanie zdarzeń w dzienniku lub interfejsie użytkownika.
            // ...
        }
    }

    public FileStream? Stream { get; private set; }
    public FileInfo? File { get; private set; }

    public void Close()
    {
        Stream?.Dispose();
        try
        {
            File?.Delete();
        }
        catch(IOException exception)
        {
            Console.WriteLine(exception);
        }
        Stream = null;
        File = null;
    }
}
```

Do finalizatorów nie można przekazywać żadnych parametrów, dlatego nie da się tworzyć przeciążonych wersji. Ponadto finalizatory nie mogą być wywoływane jawnie. Tylko mechanizm odzyskiwania pamięci może wywołać finalizator. Modyfikatory dostępu nie mają więc zastosowania do finalizatorów, dlatego nie są dla nich dozwolone. Finalizatory z klas bazowych są wywoływanie automatycznie w ramach wywołania finalizatora danego obiektu.

Uwaga

Finalizatorów nie można wywoływać jawnie. Tylko mechanizm odzyskiwania pamięci może wywołać finalizator.

Ponieważ za zarządzanie pamięcią odpowiada mechanizm odzyskiwania pamięci, finalizatory nie mają za zadanie zwalniać pamięci. Ich funkcją jest zwalnianie takich zasobów jak połączenia z bazą danych i uchwyty do plików. Są to zasoby wymagające jawniej obsługi, ale nieznane mechanizmowi odzyskiwania pamięci.

W finalizatorze z listingu 10.20 program najpierw usuwa strumień typu `FileStream`. Ten krok jest opcjonalny, ponieważ ten typ ma własny finalizator działający tak samo jak metoda `Dispose()`. Wywołanie `Dispose()` ma tu gwarantować, że strumień `FileStream` zostanie usunięty w momencie finalizacji strumienia `TemporaryFileStream`, ponieważ ten ostatni odpowiada za utworzenie tego pierwszego. Jeśli pominiesz wywołanie `Stream?.Dispose()`, mechanizm przywracania pamięci usunie strumień `FileStream` niezależnie od strumienia `TemporaryFileStream`, ale dopiero po tym, jak strumień `TemporaryFileStream` zostanie usunięty i zwolniona zostanie zapisana w nim referencja do strumienia `FileStream`. Ale jeśli nie potrzebujesz finalizatora do zwalniania zasobów, to nie ma sensu definiować go tylko po to, aby wywołać metodę `FileStream.Dispose()`.

Warto zauważyc, że zalecenie stosowania finalizatorów tylko dla obiektów wymagających zwolnienia zasobów, których środowisko uruchomieniowe nie jest świadome (chodzi tu o zasoby niemające finalizatorów), to ważna wskazówka, która znacznie zmniejsza liczbę scenariuszy wymagających implementacji finalizatora.

Na listingu 10.20 finalizator ma usuwać plik⁸, który jest tu niezarządzanym zasobem. Dlatego trzeba wywołać metodę `File?.Delete()`. Po wykonaniu finalizatorów plik zostanie usunięty.

Finalizatory działają we własnym wątku, co dodatkowo sprawia, że ich działanie jest mniej deterministyczne. Niedeterministyczny charakter pracy finalizatorów powoduje, że trudno jest diagnozować zgłoszone w nich (poza debugerem) nieobsłużone wyjątki. Dzieje się tak, ponieważ nie są jednoznacznie określone warunki prowadzące do wyjątku. Z perspektywy użytkownika nieobsłużony wyjątek z finalizatora jest zgłoszany niemal losowo i bez bezpośredniego powiązania z działaniami wykonywanymi przez daną osobę. Dlatego trzeba zapobiegać występowaniu wyjątków w finalizatorach. Należy stosować techniki programowania defensywnego, takie jak wykrywanie wartości `null` (zobacz listing 10.20). Zalecane jest przechwytywanie wszystkich wyjątków w finalizatorze i zgłoszanie ich za pomocą innych technik (na przykład rejestrowanie w dzienniku lub wyświetlanie w interfejsie użytkownika) zamiast pozostawiania ich jako nieobsłużonych wyjątków. Z tego powodu wywołanie metody `Delete()` zostało umieszczone w bloku `try-catch`.

⁸ Przykład z listingu 10.20 jest nieco naciągany, ponieważ w trakcie tworzenia strumienia `FileStream` można użyć opcji `FileOptions.DeleteOnClose`, która powoduje usunięcie pliku po zamknięciu strumienia.

Innym sposobem na wymuszenie wykonania finalizatorów jest wywołanie metody `System.GC.WaitForPendingFinalizers()`. Jej uruchomienie powoduje, że bieżący wątek zostanie wstrzymany do czasu wykonania finalizatorów wszystkich obiektów, do których nie istnieją już referencje.

Deterministyczna finalizacja z wykorzystaniem instrukcji using

Problem z finalizatorami polega na tym, że nie obsługują one **finalizacji deterministycznej** (umożliwiającej określenie, kiedy finalizator zadziała). Finalizatory odgrywają ważną rolę jako rezerwowy mechanizm porządkowania zasobów, stosowany, gdy programista korzystający z danej klasy zapomni jawnie wywołać potrzebny kod zwalniający te zasoby.

Na przykład klasa `TemporaryFileStream` obejmuje nie tylko finalizator, ale też metodę `Close()`. Ta klasa korzysta z zasobu plikowego, który może zajmować dużo miejsca na dysku. Programista posługujący się klasą `TemporaryFileStream` może jawnie wywołać metodę `Close()`, by przywrócić zajmowaną przez zasób pamięć.

Udostępnienie metody umożliwiającej deterministyczną finalizację jest ważne, ponieważ pozwala wyeliminować zależność od niedeterministycznego finalizatora. Nawet jeśli programista nie wywoła jawnie metody `Close()`, zadba o to finalizator. W takiej sytuacji finalizator zadziała później niż przy jawnym wywołaniu, ale w końcu zostanie uruchomiony.

Z powodu znaczenia deterministycznej finalizacji podstawowa biblioteka klas udostępnia interfejs obsługujący ten wzorzec. W C# ten wzorzec jest zintegrowany z językiem. Szczegóły działania wzorca są zdefiniowane w metodzie `Dispose()` interfejsu `IDisposable`. Programista może wywołać tę metodę dla klasy zasobu, by zwolnić zajmowane zasoby. Na listingu 10.21 przedstawiono interfejs `IDisposable` i korzystający z niego kod.

Listing 10.21. Porządkowanie zasobów z wykorzystaniem interfejsu `IDisposable`

```
using System;
using System.IO;

public class Program
{
    // ...
    static void Search()
    {
        TemporaryFileStream fileStream =
            new TemporaryFileStream();

        // Używanie obiektu typu TemporaryFileStream.
        // ...

        fileStream.Dispose();

        // ...
    }
}

class TemporaryFileStream : IDisposable
{
    public TemporaryFileStream(string fileName)
    {
```

```
File = new FileInfo(fileName);
Stream = new FileStream(
    File.FullName, FileMode.OpenOrCreate,
    FileAccess.ReadWrite);
}

public TemporaryFileStream()
    : this(Path.GetTempFileName()) { }

~TemporaryFileStream()
{
    Dispose(false);
}

public FileStream? Stream { get; private set; }
public FileInfo? File { get; private set; }

#region Składowe z interfejsu IDisposable
public void Dispose()
{
    Dispose(true);
    // Wyłączenie wywołań finalizatora.
    System.GC.SuppressFinalize(this);
}
#endregion
public void Dispose(bool disposing)
{
    // Nie trzeba usuwać własnego zarządzanego obiektu
    // (finalizatorem), jeśli metoda została wywołana przez finalizator.
    // Powodem jest to, że finalizator takich obiektów zostanie
    // (lub już został) wywołany w przetwarzanej kolejce finalizacji.
    if (disposing)
    {
        Stream?.Close();
    }
    try
    {
        File?.Delete();
    }
    catch(IOException exception)
    {
        Console.WriteLine(exception);
    }
    Stream = null;
    File = null;
}
}
```

W metodzie Program.Search() znajduje się jawnie wywołanie metody Dispose() po zakończeniu korzystania z obiektu typu `TemporaryFileStream`. `Dispose()` to metoda odpowiedzialna za porządkowanie zasobów (tu jest nim plik) niepowiązanych z pamięcią, które nie są obsługiwane przez mechanizm odzyskiwania pamięci. Jednak w przedstawionym kodzie kryje się luka, która może uniemożliwić wykonanie metody `Dispose()`. Chodzi o możliwość wystąpienia wyjątku między momentem utworzenia obiektu typu `TemporaryFileStream` a czasem wywołania metody `Dispose()`. W takiej sytuacji metoda `Dispose()` nie zostanie

uruchomiona, a za porządkowanie zasobów będzie odpowiadał finalizator. Aby uniknąć tego problemu, w jednostce wywołującej należy umieścić blok `try-finally`. Zamiast wymagać od programistów jawnego pisania takiego bloku, w C# udostępniono instrukcję `using`, która wykonuje potrzebne zadania (zobacz listing 10.22).

Listing 10.22. Wywoływanie instrukcji `using`

```
public class Program
{
    // ...

    static void Search()
    {
        using (TemporaryFileStream fileStream2 =
            new TemporaryFileStream(),
            fileStream3 = new TemporaryFileStream())
        {
            // Korzystanie z obiektu typu TemporaryFileStream.
        }

        // Od wersji C# 8.0.
        using TemporaryFileStream fileStream1 =
            new TemporaryFileStream();
    }
}
```

W pierwszym wyróżnionym fragmencie wynikowy kod CIL jest taki sam jak kod generowany wtedy, gdy programista jawnie tworzy blok `try-finally` i umieszcza w bloku `finally` wywołanie `fileStream.Dispose()`. Instrukcja `using` to składniowy skrót odpowiadający blokowi `try-finally`.

W instrukcji `using` można utworzyć więcej niż jedną zmienną. Poszczególne zmienne należy oddzielić od siebie przecinkami. Ważne jest, że wszystkie podane zmienne muszą być tego samego typu, a ten typ musi obejmować implementację interfejsu `IDisposable`; należy też inicjować zmienne w miejscu deklaracji. Aby wymusić używanie tego samego typu, jest on podawany tylko raz, a nie przed deklaracją każdej zmiennej.

Od wersji C# 8.0 można stosować uproszczone zwalnianie zasobów. W drugim wyróżnionym fragmencie na listingu 10.22 pokazano, że można poprzedzić deklarację zwalnianego zasobu (implementującego interfejs `IDisposable`) słowem kluczowym `using`. Podobnie jak instrukcja `using`, powoduje to wygenerowanie bloku `try-finally`. Blok `finally` jest umieszczany tuż przed wyjściem zmiennej z zasięgu (tu: przed zamkającym nawiasem klamrowym metody `Search()`). Dodatkowym ograniczeniem związanym z deklaracją `using` jest to, że zmienna jest przeznaczona tylko do odczytu, dlatego nie można jej przypisać innej wartości.

Odzyskiwanie pamięci, finalizacja i interfejs `IDisposable`

Na listingu 10.21 warto zwrócić uwagę na kilka aspektów. Pierwszy z nich dotyczy tego, że metoda `IDisposable.Dispose()` zawiera istotne wywołanie `System.GC.SuppressFinalize()`. Sprawia ono, że obiekt typu `TemporaryFileStream` jest usuwany z kolejki finalizacji (ang. *finalization queue* lub *f-reachable queue*). Jest to dopuszczalne, ponieważ wszystkie operacje porządkujące wykonano w metodzie `Dispose()`. Nie trzeba więc czekać na wykonanie finalizatora.

Bez wywołania metody `SuppressFinalize()` obiekt znajdzie się w kolejce finalizacji. Zawiera ona listę wszystkich obiektów prawie gotowych do zwolnienia pamięci, ale zawierających implementację finalizacji. Środowisko uruchomieniowe nie może odzyskać pamięci obiektów z finalizatorami, jeśli metody finalizacji tych obiektów nie zostały jeszcze wykonane. Jednak sam mechanizm odzyskiwania pamięci nie wywołuje takich metod. Referencje do obiektów z finalizacją są umieszczane w kolejce finalizacji i przetwarzane przez dodatkowy wątek w czasie uznawanym za odpowiedni na podstawie kontekstu wykonywania programu. Na ironię zakrawa fakt, że to podejście opóźnia odzyskiwanie pamięci zajmowanej przez zasoby zarządzane, choć zwykle te właśnie zasoby powinny zostać zwolnione wcześniej. Opóźnienie wynika z tego, że kolejka finalizacji to lista referencji. Dlatego powiązane z nimi obiekty nie są uznawane za „śmieci” do czasu wykonania metod finalizacji i usunięcia referencji z kolejki.

Uwaga

Obiekty z finalizatorami, które nie są jawnie zwalniane, mają przedłużony czas życia. Nawet po usunięciu wszystkich jawnych referencji z zasięgu referencje znajdują się w kolejce finalizacji. Dlatego obiekt pozostaje dostępny do czasu zakończenia przetwarzania jego referencji w kolejce finalizacji.

Ze względu na opisane powody w metodzie `Dispose()` warto umieścić wywołanie `System.GC.SuppressFinalize`. Wywołanie tej metody jest dla środowiska uruchomieniowego informacją, żeby nie dodawać danego obiektu do kolejki finalizacji. Mechanizm odzyskiwania pamięci może wtedy zwolnić pamięć obiektu w momencie, w którym nie prowadzą do niego żadne referencje (łącznie z referencjami w kolejce finalizacji).

Drugi aspekt to wywołanie metody `Dispose(bool disposing)` z argumentem `true`. W efekcie wywoływana jest metoda `Dispose()` obiektu typu `Stream` (co prowadzi do zwolnienia zasobów i zablokowania finalizacji). Następnie bezpośrednio po wywołaniu metody `Dispose()` usuwany jest sam plik tymczasowy. Ważne wywołanie wspomniane na początku akapitu eliminuje konieczność oczekiwania na przetworzenie kolejki finalizacji przed zwolnieniem potencjalnie kosztownych zasobów.

Trzecia kwestia dotyczy tego, że finalizator, zamiast wywoływać metodę `Close()`, uruchamia metodę `Dispose(bool disposing)` z argumentem `false`. W efekcie obiekt typu `Stream` nie jest zamknięty (zwalniany), choć sam plik zostaje usunięty. Warunek powiązany z operacją zamknięcia obiektu typu `Stream` sprawia, że jeśli w finalizatorze nastąpi wywołanie `Dispose(bool disposing)`, obiekt typu `Stream` też trafi do kolejki finalizacji (w zależności od kolejności wykonywania instrukcji może się też zdarzyć, że obiekt został już przetworzony). Dlatego w trakcie działania finalizatora obiekty należące do zarządzanego zasobu nie powinny być zwalniane, ponieważ za to zadanie odpowiada kolejka finalizacji.

Czwarty punkt dotyczy tego, że należy zachować ostrożność w trakcie tworzenia metod `Close()` i `Dispose()`. Na podstawie samego interfejsu API trudno stwierdzić, czy metoda `Close()` wywołuje metodę `Dispose()`. Dlatego programiści mogą mieć wątpliwości, czy muszą jawnie wywoływać obie te metody.

Piąte zagadnienie dotyczy zwiększenia prawdopodobieństwa tego, że kod zdefiniowany w finalizatorze zostanie wykonany przed zamknięciem procesu nawet w platformie .NET Core. W tym celu należy zarejestrować kod w metodzie obsługi zdarzenia AppDomain.CurrentDomain.ProcessExit. Tak zarejestrowany kod finalizacji jest wykonywany zawsze, chyba że proces zostanie zamknięty w niestandardowy sposób (co jest opisane w następnym punkcie).

Wskazówki

IMPLEMENTUJ finalizator w obiektach korzystających z ograniczonych lub kosztownych zasobów.

IMPLEMENTUJ interfejs IDisposable, aby dodać obsługę deterministycznej finalizacji w klasach z finalizatorami.

IMPLEMENTUJ finalizator tylko w klasach, które mają zasoby nieposiadające finalizatora, ale wymagające zwolnienia.

PRZEPROWADŹ refaktoryzację finalizatora, by wywoływał ten sam kod co metody z interfejsu IDisposable (można na przykład wywołać w finalizatorze metodę Dispose()).

NIE zgłaszaj wyjątków w finalizatorach.

ROZWAŻ zarejestrowanie kodu finalizacji w metodzie obsługi zdarzenia AppDomain.ProcessExit, by zwiększyć prawdopodobieństwo zwolnienia zasobów przed zamknięciem procesu.

WYREJESTRUJ metody obsługi zdarzenia AppDomain.ProcessExit w trakcie zwalniania zasobów.

WYWOLUJ metodę System.GC.SuppressFinalize() w metodzie Dispose(), by uniknąć powtórnego zwalniania zasobów i opóźniania odzyskiwania pamięci obiektu.

UPEWNIJ SIĘ, że metoda Dispose() jest idempotentna (wielokrotne wywołanie Dispose() nie powinno prowadzić do błędów).

DBAJ o to, by metoda Dispose() była prosta. Należy się w niej skoncentrować na niezbędnym w ramach finalizacji porządkowaniu zasobów.

UNIKAJ wywoływania metody Dispose() dla własnych obiektów z finalizatorem. Za porządkowanie obiektów powinna odpowiadać kolejka finalizacji.

UNIKAJ w finalizatorze referencji do innych obiektów, które nie są finalizowane.

WYWOLUJ metodę Dispose() klasy bazowej, jeśli przesyłasz tę metodę.

ROZWAŻ zapewnienie tego, że obiekt stanie się nieużywalny po wywołaniu metody Dispose(). Po usunięciu obiektu wywołanie metody innej niż Dispose() (która potencjalnie można uruchamiać wielokrotnie) powinno kończyć się zgłoszeniem wyjątku ObjectDisposedException.

IMPLEMENTUJ interfejs IDisposable w typach mających pola (lub właściwości) z zasobami, które należy zwolnić, i zwalniaj te zasoby.

WYWOLUJ w metodzie Dispose(bool disposing) metodę Dispose() klasy bazowej (jeśli taka metoda istnieje).

Porównanie języków — deterministyczny destruktor w języku C++

Choć finalizatory są podobne do destruktorów z języka C++, istotna różnica między tymi konstrukcjami polega na tym, że na etapie komplikacji nie da się określić momentu wykonania finalizatorów. Mechanizm odzyskiwania pamięci w języku C# wywołuje finalizatory w okresie między ostatnim użyciem obiektu a zamknięciem programu. Destruktory w języku C++ są wywoływane automatycznie w momencie, gdy obiekt (a nie wskaźnik) wychodzi z zasięgu.

Choć działanie mechanizmu odzyskiwania pamięci bywa kosztowne, mechanizm ten jest na tyle „inteligentny”, że opóźnia swoje uruchomienie do momentu, w którym obciążenie procesora spada. Jest to zaleta w porównaniu z deterministycznymi destruktorami, które zawsze działają w miejscach zdefiniowanych na etapie komplikacji — także w momentach, gdy obciążenie procesora jest wysokie.

Początek
7.0

ZAGADNIENIE DLA ZAAWANSOWANYCH

Wymuszanie porządkowania zasobów przed zamknięciem procesu

Od czasu wprowadzenia platformy .NET Core finalizatory mogą w momencie zamknięcia procesu nie zostać uruchomione. Aby zwiększyć prawdopodobieństwo ich wykonania, kod finalizacji należy zarejestrować, aby był uruchamiany w czasie końca procesu⁹. Zwróć uwagę na instrukcję ze zdarzeniem ProcessExit w konstruktorze SampleUnmanagedResource na listingu 10.23. W tym kodzie używane są technologia LINQ, rejestrowanie zdarzeń i specjalne atrybuty¹⁰; zagadnienia te są opisane w rozdziałach 12., 14. i 18.

Listing 10.23. Rejestrowanie kodu obsługi zdarzenia ProcessExit

```
using System.IO;
using System.Linq;
using System.Runtime.CompilerServices;
using static ConsoleLogger;

public static class Program
{
    public static void Main(string[] args)
    {
        WriteLine("Starting...");
        DoStuff();
        if (args.Any(arg => arg.ToLower() == "-gc"))
        {
            GC.Collect();
            GC.WaitForPendingFinalizers();
        }
        WriteLine("Kończenie pracy...");
    }

    public static void DoStuff()
    {
```

⁹ Technicznie jest to domena aplikacji (przynajmniej w projektach z platformą .NET Framework).

¹⁰ Zobacz kod źródłowy powiązany z książką.

```
// ...
WriteLine("Rozpoczynanie pracy...");
SampleUnmanagedResource sampleUnmanagedResource = null;

try
{
    sampleUnmanagedResource =
        new SampleUnmanagedResource();
    // Użycie niezarządzanych zasobów.
    // ...
}
finally
{
    if (Environment.GetCommandLineArgs().Any(
        arg => arg.ToLower() == "-dispose"))
    {
        sampleUnmanagedResource?.Dispose();
    }
}
WriteLine("Kończenie pracy...");
// ...
}
```

7.0

```
class SampleUnmanagedResource : IDisposable
{
    public SampleUnmanagedResource(string fileName)
    {
        WriteLine("Rozpoczynanie pracy...",
            $"{nameof(SampleUnmanagedResource)}.ctor");

        WriteLine("Tworzenie zasobów zarządzanych...",
            $"{nameof(SampleUnmanagedResource)}.ctor");
        WriteLine("Tworzenie zasobów niezarządzanych...",
            $"{nameof(SampleUnmanagedResource)}.ctor");

        var weakReferenceToSelf =
            new WeakReference<IDisposable>(this);
        ProcessExitHandler = (_, __) =>
        {
            WriteLine("Rozpoczynanie pracy...", "ProcessExitHandler");
            if (weakReferenceToSelf.TryGetTarget(
                out IDisposable? self))
            {
                self.Dispose();
            }
            WriteLine("Kończenie pracy...", "ProcessExitHandler");
        };
        AppDomain.CurrentDomain.ProcessExit
            += ProcessExitHandler;
        WriteLine("Kończenie pracy...",
            $"{nameof(SampleUnmanagedResource)}.ctor");
    }
}
```

```
// Zapisywanie delegata do obsługi zdarzenia ProcessExit, dzięki czemu można
// go usunąć, jeśli metoda Dispose() lub Finalize() została już wywołana.
private EventHandler ProcessExitHandler { get; }

public SampleUnmanagedResource()
: this(Path.GetTempFileName()) { }

~SampleUnmanagedResource()
{
    WriteLine("Rozpoczynanie pracy...");
    Dispose(false);
    WriteLine("Kończenie pracy...");
}

public void Dispose()
{
    Dispose(true);
}

public void Dispose(bool disposing)
{
    WriteLine("Rozpoczynanie pracy...");

    // Nie trzeba usuwać własnego zarządzanego obiektu
    // (z finalizatorem), jeśli metoda została wywołana przez finalizator.
    // Powodem jest to, że finalizator takich obiektów zostanie
    // (lub już został) wywołany w przetwarzanej kolejce finalizacji.
    if (disposing)
    {
        WriteLine("Usuwanie zasobów zarządzanych...");

        // Wyrejestrowanie z kolejki finalizacji.
        System.GC.SuppressFinalize(this);
    }

    AppDomain.CurrentDomain.ProcessExit -=
        ProcessExitHandler;

    WriteLine("Usuwanie zasobów niezarządzanych...");

    WriteLine("Kończenie pracy...");
}
}
```

Na listingu 10.5 pokazane są dane wyjściowe generowane, gdy do programu nie są przekazywane żadne argumenty.

DANE WYJŚCIOWE 10.5.

```
Main: Rozpoczynanie pracy...
DoStuff: Rozpoczynanie pracy...
SampleUnmanagedResource.ctor: Rozpoczynanie pracy...
SampleUnmanagedResource.ctor: Tworzenie zasobów zarządzanych...
SampleUnmanagedResource.ctor: Tworzenie zasobów niezarządzanych...
SampleUnmanagedResource.ctor: Kończenie pracy...
DoStuff: Kończenie pracy...
```

```
Main: Kończenie pracy...
ProcessExitHandler: Rozpoczynanie pracy...
Dispose: Rozpoczynanie pracy...
Dispose: Usuwanie zasobów zarządzanych...
Dispose: Uswanie zasobów niezarządzanych...
Dispose: Kończenie pracy...
ProcessExitHandler: Kończenie pracy...
```

Jeśli pominąć instrukcję `WriteLine()`, kod rozpoczyna się od wywołania metody `DoStuff()`, która tworzy zasób `SampleUnmanagedResource`.

Gdy przedstawiony kod tworzy instancję typu `SampleUnmanagedResource`, symuluje za pomocą prostych wywołań `WriteLine()` tworzenie zasobów zarządzanych i niezarządzanych. Następnie deklarowany jest delegat (metoda obsługi zdarzeń) wykonywany, gdy proces kończy pracę. Delegat wykorzystuje referencję `WeakReference` prowadzącą do instancji typu `SampleUnmanagedResource` i wywołuje metodę `Dispose()` dla tej instancji (jeśli ta jeszcze istnieje). Słaba referencja typu `WeakReference` jest potrzebna, aby zagwarantować, że po zdarzeniu `ProcessExit` nie zostanie zachowana referencja do zasobu (która uniemożliwiłaby mechanizmowi przywracania pamięci usunięcie niezarządzanego zasobu po jego wyjściu z zasięgu i wywołaniu finalizatora). Delegat jest rejestrowany dla zdarzenia `AppDomain.CurrentDomain.ProcessExit` i zapisywany we właściwości `ProcessExitHandler`. Ten ostatni krok jest potrzebny, aby można było odkĄczyć delegata od zdarzenia `AppDomain.CurrentDomain.ProcessExit` po wywołaniu metody `Dispose()`; dzięki temu metoda `Dispose()` nie jest niepotrzebnie wykonywana po raz wtóry.

W metodzie `DoStuff()` program sprawdza, czy przy uruchamianiu programu w wierszu poleceń podany został argument `-Dispose`. Jeśli tak, należy wywołać metodę `Dispose()`, a finalizator i delegat dla zdarzenia `ProcessExit` nie są uruchamiane. Po zakończeniu działania metody `DoStuff()` nie istnieje już referencja podstawaowa prowadząca do instancji typu `SampleUnmanagedResource`. Jednak gdy mechanizm przywracania pamięci zacznie działać, wykryje finalizator i doda zasób do kolejki finalizacji.

Gdy proces zacznie kończyć pracę, a dla instancji typu `SampleUnmanagedResource` nie zostały wykonane metoda `Dispose()` ani finalizator, zgłoszane jest zdarzenie `AppDomain.CurrentDomain.ProcessExit` i uruchamiana jest metoda obsługi zdarzenia, która z kolei wywołuje metodę `Dispose()`. Główna różnica w porównaniu z metodą `Dispose()` z listingu 10.21 polega na tym, że tu metoda obsługi zdarzenia `AppDomain.CurrentDomain.ProcessExit` jest wyrejestrowywana, dlatego metoda `Dispose()` nie jest ponownie wywoływana w trakcie końca pracy procesu, jeśli została już wykonana wcześniej.

Zauważ, że choć można wywołać metody `GC.Collect()` i `GC.WaitForPendingFinalizers()`, aby wymusić uruchomienie wszystkich finalizatorów obiektów niemających podstawowych referencji (i teoretycznie można to nawet zrobić tuż przed zakończeniem pracy procesu), jest to niedoskonałe rozwiązanie. Pierwsza wada dotyczy tego, że projekty biblioteczne nie mogą wywoływać takich metod bezpośrednio przed zakończeniem procesu, ponieważ nie występuje w nich metoda `Main()`. Drugie zastrzeżenie związane jest z tym, że nawet proste konstrukcje, takie jak referencje statyczne, są referencjami podstawowymi, dlatego statyczne obiekty nie zostaną usunięte. Z tego powodu zalecanym podejściem jest używanie metody obsługi zdarzenia `ProcessExit`.

ZAGADNIENIE DLA ZAAWANSOWANYCH

Przekazywanie wyjątków z konstruktorów

Nawet jeśli z konstruktora zostanie przekazany dalej wyjątek, obiekt zostanie utworzony, choć operator new go nie zwróci. Jeśli w typie jest zdefiniowany finalizator, zostanie on uruchomiony w momencie, gdy obiekt stanie się gotowy do przetworzenia przez mechanizm odzyskiwania pamięci (jest to dodatkowa motywacja do tego, by umożliwić działanie finalizatora dla częściowo utworzonych obiektów). Zauważ, że jeśli konstruktor przedwcześnie udostępnii referencję this, będzie ona dostępna nawet po zgłoszeniu wyjątku w konstruktorze. Nie pozwól, by doszło do takiej sytuacji.

ZAGADNIENIE DLA ZAAWANSOWANYCH

Ponowne tworzenie usuwanych obiektów

W momencie wywołania finalizatora obiektu żadne referencje do obiektu już nie istnieją. Ostatnim krokiem przed odzyskaniem pamięci jest właśnie uruchomienie kodu finalizacji. Może się jednak zdarzyć, że do grafu z podstawowymi referencjami przypadkowo zostanie dodana referencja do finalizowanego obiektu. W takiej sytuacji obiekt z ponownie dodaną referencją nie będzie już niedostępny i nie będzie gotowy do zwolnienia pamięci. Jeśli jednak metoda finalizacji została już wykonana, niekoniecznie zostanie uruchomiona ponownie, chyba że zostanie jawnie oznaczona (za pomocą metody GC.ReRegisterFinalize()) jako przeznaczona do finalizacji.

Oczywiście ponowne tworzenie usuwanych obiektów w ten sposób to osobliwe rozwiązanie i zwykle należy tego unikać. Kod finalizacji powinien być prosty i zwalniać tylko używane przez obiekt zasoby.

Koniec
7.0

Leniwe inicjowanie

Początek
4.0

W poprzednim podrozdziale wyjaśniono, jak w deterministyczny sposób usunąć obiekt za pomocą instrukcji using. Dowiedziałeś się też, że kolejka finalizacji zwalnia zasoby w sytuacjach, gdy nie jest stosowane podejście deterministyczne.

Istnieje powiązany wzorzec o nazwie **leniwe inicjowanie** (lub **leniwe ładowanie**). Za pomocą leniwego inicjowania można tworzyć (lub pobierać) obiekty, gdy są one potrzebne, a nie wcześniej. Wcześniejsze tworzenie obiektów jest problematyczne zwłaszcza wtedy, gdy dany obiekt nie jest potem nigdy używany. Przyjrzyj się właściwości FileStream na listingu 10.24.

Listing 10.24. Leniwe inicjowanie właściwości

```
using System.IO;  
  
class DataCache  
{  
    // ...  
    public TemporaryFileStream FileStream =>  
        InternalFileStream??(InternalFileStream =
```

```

new TemporaryFileStream());
private TemporaryFileStream? InternalFileStream
{ get; set; } = null;
} // ...
}

```

We właściwości `FileStream` (z ciałem w postaci wyrażenia) kod sprawdza, czy strumień `InternalFileStream` ma wartość różną od `null`. Jeśli tak jest, bezpośrednio zwracana jest wartość tego strumienia. Gdy strumień `InternalFileStream` ma wartość `null`, kod najpierw tworzy obiekt typu `TemporaryFileStream` i przypisuje go do obiektu typu `InternalFileStream`, a następnie zwraca nowy obiekt. Tak więc obiekt typu `TemporaryFileStream` potrzebny we właściwości `FileStream` jest tworzony tylko po wywołaniu gettera tej właściwości. Jeśli getter nigdy nie zostanie użyty, obiekt typu `TemporaryFileStream` nie będzie tworzony. Pozwala to uniknąć poświęcania czasu na tworzenie tego obiektu. Oczywiście jeśli ten czas jest nieodczuwalny lub gdy utworzenia obiektu nie da się uniknąć (odkładanie nieuniknionego nie jest pożądane), uzasadnione jest utworzenie potrzebnego obiektu w deklaracji lub w konstruktorze.

■ ZAGADNIENIE DLA ZAAWANSOWANYCH

Leniwe inicjowanie za pomocą typów generycznych i wyrażeń lambda

Od wersji 4.0 platformy Microsoft .NET Framework i języka C# 4.0 w środowisku CLR dostępna jest nowa klasa pomocna przy leniwym inicjowaniu. Jest to klasa `System.Lazy<T>`. Na listingu 10.25 pokazano, jak się nią posługiwać.

Listing 10.25. Leniwe inicjowanie właściwości za pomocą klasy `System.Lazy<T>`

```

using System.IO;

class DataCache
{
    // ...

    public TemporaryFileStream FileStream =>
        InternalFileStream.Value;
    private Lazy<TemporaryFileStream> InternalFileStream { get; } =
        new Lazy<TemporaryFileStream>(
            () => new TemporaryFileStream() );
}

// ...
}

```

Klasa `System.Lazy<T>` przyjmuje parametr `T`, który określa, jakiego typu wartość ma zwracać właściwość `Value` tej klasy. Zamiast przypisywać do pola `_FileStream` gotowy obiekt typu `TemporaryFileStream`, kod przypisuje obiekt typu `Lazy<TemporaryFileStream>` (jest to mało kosztowne wywołanie), co opóźnia tworzenie obiektu typu `TemporaryFileStream` do momentu, w którym następuje dostęp do właściwości `Value` (a tym samym i właściwości `FileStream`).

Jeśli oprócz parametrów określających typ (i typów generycznych) stosujesz delegaty, możesz nawet udostępnić funkcję definiującą, jak zainicjować obiekt w momencie dostępu do właściwości `Value`. Na listingu 10.25 pokazano, jak przekazać delegat (tu jest nim wyrażenie lambda) do konstruktora typu `System.Lazy<T>`.

Zauważ, że samo wyrażenie lambda ((() => new `TemporaryFileStream(FileStreamName)`) jest wykonywane dopiero po wywołaniu właściwości `Value`. Wyrażenie lambda umożliwia przekazanie instrukcji określających, co się stanie. Te instrukcje są wykonywane dopiero wtedy, gdy kod tego zażąda.

Naturalne jest pytanie o to, kiedy należy stosować klasę `System.Lazy<T>` zamiast podejścia przedstawionego na listingu 10.24. Różnica między tymi rozwiązaniami jest niewielka. Podejście z listingu 10.24 może się okazać prostsze (zwykle jest ono takie, chyba że używanych jest wiele wątków, przez co warunek wyścigu może wystąpić niezależnie od sposobu tworzenia obiektów). Na listingu przed utworzeniem obiektu kod może sprawdzać równość z wartością `null` więcej niż raz. Może to skutkować utworzeniem kilku obiektów. Klasa `System.Lazy<T>` zapewnia mechanizm bezpieczny ze względu na wątki, gwarantujący, że powstanie tylko jeden obiekt.

Podsumowanie

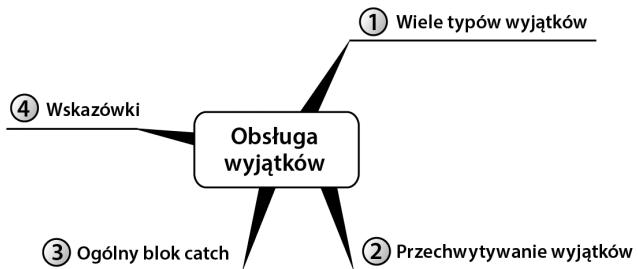
W tym rozdziale przedstawiono krótki przegląd wielu zagadnień związanych z budowaniem dobrych bibliotek klas. Wszystkie opisane tematy dotyczą także wewnętrznych mechanizmów, ale są znacznie ważniejsze w kontekście budowania wysokiej jakości klas. Nacisk położono tu na rozwijanie niezawodnych i programowalnych interfejsów API. Z niezawodnością związane są przestrzenie nazw i odzyskiwanie pamięci. Oba te zagadnienia dotyczą także programowalności, z którą łączą się też przesłanianie składowych wirtualnych klasy `object`, przeciążanie operatorów i komentarze XML-owe pełniące funkcję dokumentacji.

W obsłudze wyjątków w dużym stopniu wykorzystywane jest dziedziczenie. Zdefiniowana jest hierarchia wyjątków, a niestandardowe wyjątki należy umieszczać w jej odpowiednim miejscu. Ponadto kompilator języka C# wykorzystuje dziedziczenie do sprawdzania, który blok `catch` należy uruchomić. Z rozdziału 11. dowiesz się, dlaczego dziedziczenie jest tak ważnym aspektem obsługi wyjątków.

11

Obsługa wyjątków

ROZDZIAŁE 5. OPISANO UŻYWANIE bloków try/catch/finally do standardowej obsługi wyjątków. We wspomnianym rozdziale blok catch zawsze przechwytywał wyjątki typu System.Exception. Tu zapoznasz się z dodatkowymi informacjami na temat obsługi wyjątków i przeczytasz o szczegółach związanych z innymi typami wyjątków, definiowaniem niestandardowych wyjątków i wieloma blokami catch przeznaczonymi do przechwytywania wyjątków poszczególnych typów. Ten rozdział zawiera też dokładny opis dziedziczenia w kontekście wyjątków.



Wiele typów wyjątków

Kod na listingu 11.1 zgłasza wyjątek typu System.ArgumentException, a nie opisany w rozdziale 5. wyjątek typu System.Exception. Język C# umożliwia zgłoszenie wyjątków dowolnego typu pochodnego (często pośrednio) od typu System.Exception. Aby zgłosić wyjątek, należy poprzedzić nazwę reprezentującego wyjątek obiektu słowem kluczowym throw. Jako typ wyjątku należy zastosować typ najlepiej opisujący warunki związane z błędem, który był przyczyną wystąpienia tego wyjątku. Przyjrzyj się metodzie TextNumberParser.Parse() z listingu 11.1.

Listing 11.1. Zgłaszanie wyjątku

```
public sealed class TextNumberParser
{
    public static int Parse(string textDigit)
    {
        string[] digitTexts =
```

```

    { "zero", "jeden", "dwa", "trzy", "cztery",
      "pięć", "sześć", "siedem", "osiem", "dziewięć" };

  int result = Array.IndexOf(
    digitTexts,
    // Wykorzystanie wprowadzonego w wersji C# 2.0 operatora ??
    (textDigit??
     // Wykorzystanie wprowadzonego w wersji C# 7.0 wyrażenia throw.
     throw new ArgumentNullException(nameof(textDigit))
    ).ToLower());
  if (result < 0)
  {
    // Wykorzystanie wprowadzonego w wersji C# 6.0 operatora nameof.
    throw new ArgumentException(
      "Argument nie reprezentuje cyfry",
      nameof(textDigit));
  }
  return result;
}
}

```

W wywoaniu `Array.IndexOf()` w sytuacji, gdy argument `textDigit` ma wartość `null`, używane jest wprowadzone w wersji C# 7.0 wyrażenie `throw`. Do wersji C# 7.0 wyrażenia `throw` były niedozwolone, można było posługiwać się tylko instrukcjami `throw`. Dlatego konieczne były dwie instrukcje — jedna sprawdzająca wartość `null` i druga zgłaszająca wyjątek. Nie można było na przykład umieścić wyrażenia `throw` w tej samej instrukcji, w której używano operatora `??`.

Zamiast zgłaszać wyjątek typu `System.Exception`, lepiej wybrać typ `ArgumentException`, ponieważ określa on, co się stało, i zawiera specjalne parametry pozwalające ustalić, który argument był błędny.

Dwa podobne typy wyjątków to `ArgumentNullException` i `NullReferenceException`. Wyjątek typu `ArgumentNullException` należy zgłaszać, gdy argument błędnie ma wartość `null`. Jest to specjalna odmiana bardziej ogólnego wyjątku dotyczącego niewłaściwego parametru. Gdy błędny argument ma wartość różną od `null`, należy wybrać typ `ArgumentException` lub `ArgumentOutOfRangeException`. Wyjątku typu `NullReferenceException` zwykle są zgłaszane przez środowisko uruchomieniowe po wykryciu próby dereferencji wartości `null` (czyli w sytuacji, gdy programista próbuje wywołać składową obiektu o wartości `null`). Zamiast zgłaszać wyjątek `NullReferenceException`, programista powinien przed dostępem do parametrów sprawdzać, czy ich wartość jest różna od `null`. Jeśli parametr to `null`, należy zgłosić wyjątek typu `ArgumentNullException`, który zapewni więcej informacji kontekstowych (na przykład nazwę parametru). Jeśli istnieje bezpieczny sposób na kontynuowanie pracy nawet wtedy, gdy argument to `null`, należy zastosować wprowadzony w wersji C# 6.0 operator `?.` (ang. *null conditional operator*). Pozwala to uniknąć zgłaszania przez środowisko uruchomieniowe wyjątku typu `NullReferenceException`.

Ważną cechą typów wyjątków dotyczących argumentów (w tym typów `ArgumentNullException`, `ArgumentException` i `ArgumentOutOfRangeException`) jest to, że konstruktor każdego z tych typów przyjmuje parametr pozwalający podać nazwę argumentu jako łańcuch znaków. W wersjach starszych niż C# 6.0 określenie nazwy parametru wymagało zapisania na stałe „magicznego” łańcucha znaków (na przykład `textDigit`). Problem z tym podejściem polega na tym, że jeśli nazwa parametru się zmieni, programista będzie musiał pamiętać

o zmodyfikowaniu „magicznego” łańcucha znaków. Na szczeble C# 6.0 udostępnia nowy operator, `nameof`, który przyjmuje identyfikator parametru i generuje na etapie komplikacji odpowiednią nazwę (zobacz wywołanie `nameof(textDigit)` na listingu 11.1). Zaletą tego podejścia jest to, że w środowisku IDE można zastosować narzędzia do przeprowadzania refaktoryzacji (na przykład do automatycznej zmiany nazw), by zmienić wszystkie wystąpienia danego identyfikatora — także w miejscu użycia go w operatorze `nameof`. Ponadto jeśli nazwa parametru się zmieni, a programista nie zastosuje narzędzia do refaktoryzacji, kompilator wygeneruje błąd, w przypadku gdy identyfikator przekazany do operatora `nameof` nie będzie istniał. Dlatego od wersji C# 6.0 zaleca się, by zawsze korzystać z operatora `nameof` do podawania nazwy parametru w wyjątkach dotyczących argumentów. W rozdziale 18. znajdziesz dokładne omówienie operatora `nameof`. Na razie zapamiętaj, że zwraca on nazwę wskazanego argumentu.

Istnieje też kilka innych wyjątków przeznaczonych tylko dla środowiska uruchomieniowego i pochodnych (czasem pośrednio) od typu `System.SystemException`. Te typy to między innymi `System.StackOverflowException`, `System.OutOfMemoryException`, `System.Runtime.InteropServices.COMException`, `System.ExecutionEngineException` i `System.Runtime.InteropServices.SEHException`. Nie zgłaszać wyjątków tych typów. Należy też unikać zgłaszania wyjątków typów `System.Exception` i `System.ApplicationException`, ponieważ są one na tyle ogólne, że zapewniają niewiele informacji na temat powodu problemu i nie ułatwiają rozwiązania go. Zgłaszać wyjątek najbardziej pochodnego typu pasującego do danego scenariusza. Oczywiście programiści powinni unikać tworzenia interfejsów API, które mogą prowadzić do awarii systemu. Jeśli jednak program znajdzie się w stanie, w którym dalsza praca jest niebezpieczna lub poradzenie sobie z błędem jest niemożliwe, należy wywołać metodę `System.Environment.FailFast()`. Powoduje ona natychmiastowe zakończenie procesu po zapisaniu komunikatu w dzienniku zdarzeń aplikacji systemu Windows (komunikat można też przekazać do mechanizmu zgłaszania błędów systemu Windows, jeśli użytkownik tak zdecyduje).

Wskazówki

ZGŁASZAJ wyjątki typu `ArgumentException` lub jednego z typów pochodnych, jeśli do składowej przekazane zostały nieprawidłowe argumenty. Preferuj używanie wyjątków najbardziej pochodnego typu (na przykład `ArgumentNullException`), jeśli odpowiedni typ istnieje.

NIE zgłaszać wyjątków typu `System.SystemException` i typów od niego pochodnych.

NIE zgłaszać wyjątków typów `System.Exception`, `System.NullReferenceException` i `System.ApplicationException`.

ROZWAŻ kończenie pracy procesu za pomocą wywołania `System.Environment.FailFast()`, gdy dalsze wykonywanie programu jest niebezpieczne.

STOSUJ operator `nameof` do przekazywania argumentu `paramName` w wyjątkach typów przyjmujących taki argument (na przykład `ArgumentException`, `ArgumentOutOfRangeException` i `ArgumentNullException`).

Przechwytywanie wyjątków

Zgłoszenie określonego wyjątku umożliwia jednostce przechwytyjącej go wykorzystanie typu wyjątku do zidentyfikowania problemu. Nie trzeba więc przechwytywać wyjątku, a następnie korzystać z instrukcji switch do komunikatu, by ustalić, jakie operacje wykonać w reakcji na dany wyjątek. W języku C# można utworzyć wiele bloków catch przeznaczonych dla określonych typów wyjątków. Ilustruje to listing 11.2.

Listing 11.2. Przechwytywanie wyjątków różnych typów

```
using System;

public sealed class Program
{
    public static void Main(string[] args)
    {
        try
        {
            // ...
            throw new InvalidOperationException(
                "Dowolny wyjątek");
            // ...
        }
        catch (Win32Exception exception)
        when (exception.NativeErrorCode == 42)
        {
            // Obsługa wyjątku typu Win32Exception, gdy
            // właściwość NativeErrorCode ma wartość 42.
        }
        catch (ArgumentException exception)
        {
            // Obsługa wyjątków typu ArgumentException.
        }
        catch (InvalidOperationException exception)
        {
            bool exceptionHandled=false;
            // Obsługa wyjątków typu InvalidOperationException.
            // ...
            if (!exceptionHandled)
            {
                throw;
            }
        }
        catch (Exception exception)
        {
            // Obsługa wyjątków typu Exception.
        }
        finally
        {
            // Tu umieść kod porządkujący zasoby. Ten kod
            // jest wykonywany niezależnie od tego, czy wystąpił wyjątek.
        }
    }
}
```

Na listingu 11.2 znajduje się pięć bloków catch. Każdy z nich obsługuje wyjątki innego typu. Po wystąpieniu wyjątku sterowanie przechodzi do bloku catch przeznaczonego dla typu najbliższego typowi zgłoszonego wyjątku. Poziom dopasowania jest wyznaczany na podstawie łańcucha dziedziczenia. Na przykład typ `System.InvalidOperationException` pozostaje w relacji „jest odmianą” z typem `System.Exception`, ponieważ pośrednio po nim dziedziczy. Dlatego gdy zgłoszony wyjątek jest obu tych typów, typ `InvalidOperationException` jest bardziej dopasowany, tak więc wyjątek zostanie przechwycony przez blok `catch(InvalidOperationException...)`, a nie przez blok `catch(Exception...)`.

Od wersji C# 6.0 dostępne są dodatkowe wyrażenia warunkowe w blokach catch. Dzięki temu w C# 6.0 dopasowywanie bloków catch odbywa się nie tylko na podstawie typu wyjątku, ale też na podstawie warunkowej klauzuli. W klauzuli `when` można podać wyrażenie logiczne, a blok `catch` będzie obsługiwał dany wyjątek tylko wtedy, gdy warunek ma wartość `true`. Na listingu 11.2 w warunku używanym jest operator równości. Aby sprawdzić, czy warunek jest spełniony, można też na przykład wywołać metodę.

Oczywiście możesz też umieścić wyrażenie warunkowe jako blok `if` w bloku `catch`. Jednak to sprawia, że dany blok jest uznawany za blok obsługi wyjątku jeszcze przed sprawdzeniem warunku. Trudno napisać kod, który umożliwia innemu blokowi `catch` obsługi wyjątku w sytuacji, gdy warunek nie jest spełniony. Jednak dzięki **wyrażeniu warunkowemu wyjątku** możliwe jest sprawdzenie stanu programu (w tym wyjątku) bez konieczności przechwytywania i ponownego zgłaszania wyjątku.

Klauzule warunkowe należy jednak stosować z zachowaniem ostrożności. Jeśli w samym wyrażeniu warunkowym wyjątku wystąpi wyjątek, zostanie on zignorowany, a warunek nie będzie spełniony. Dlatego unikaj zgłaszania wyjątków w wyrażeniach warunkowych wyjątku.

Bloki `catch` należy umieścić w odpowiedniej kolejności — od najbardziej specyficzniego do najbardziej ogólnego. W przeciwnym razie wystąpi błąd czasu komplikacji. Na przykład przeniesienie bloku `catch(Exception...)` przed inne wyjątki skutkuje błędem komplikacji, ponieważ wszystkie wyjątki pośrednio lub bezpośrednio dziedziczą po typie `System.Exception`.

Blok `catch (SystemException){}` ilustruje, że nazwany parametr w blokach `catch` nie jest wymagany. W ostatnim bloku `catch` można w ogóle pominąć parametr określający typ. Przekonasz się o tym w następnym podrozdziale.

Ponowne zgłaszanie przetwarzanego wyjątku

W bloku `catch` dla wyjątków typu `InvalidOperationException` występuje instrukcja `throw`, w której nie podano typu zgłoszanego wyjątku (używane jest samo polecenie `throw`), choć obiekt wyjątku (`exception`) jest dostępny w zasięgu bloku i można go zgłosić ponownie. Zgłoszenie określonego wyjątku prowadzi do aktualizacji wszystkich informacji na stosie, tak by pasowały do nowej lokalizacji zgłoszenia. W efekcie wszystkie informacje o miejscu pierwotnego wystąpienia wyjątku zostają utracone, co znacznie utrudnia diagnozę problemu. Dlatego język C# umożliwia wywołanie instrukcji lub (od wersji C# 7.0) wyrażenia `throw` bez jawnie

Początek
6.0

Koniec
6.0

wskazanego wyjątku, jeśli instrukcja ta znajduje się w bloku catch. Dzięki temu kod może zbadać wyjątek i ustalić, czy możliwa jest jego pełna obsługa. Jeśli jest ona niemożliwa, można ponownie zgłosić wyjątek (bez jawnego podawania go), tak jakby nigdy wcześniej nie został przechwycony. Nie skutkuje to zastąpieniem informacji na stosie.

Początek
5.0

ZAGADNIENIE DLA POCZĄTKUJĄCYCH I ZAAWANSOWANYCH

Zgłaszanie istniejącego już wyjątku bez zastępowania informacji na stosie

W wersji C# 5.0 dodano mechanizm umożliwiający zgłaszanie wcześniej zgłoszonego wyjątku bez utraty zapisanych na stosie informacji o pierwotnym wyjątku. Dzięki temu można ponownie zgłaszać wyjątki nawet spoza bloku catch bez używania pustej instrukcji throw;. Choć jest to potrzebne stosunkowo rzadko, w niektórych sytuacjach wyjątki są opakowywane lub zapisywane przed wyjściem programu poza blok catch. Na przykład w kodzie wielowątkowym można opakować wyjątki w obiekt typu AggregateException. Klasa System.Runtime.ExceptionServices.ExceptionDispatchInfo zawiera metody Capture() (statyczna) i Throw() (instancji) przeznaczone do obsługi tego właśnie scenariusza. Na listingu 11.3 pokazano, jak ponownie zgłosić wyjątek bez zerowania informacji na stosie i bez używania pustej instrukcji throw.

Listing 11.3. Używanie klasy ExceptionDispatchInfo do ponownego zgłoszenia wyjątku

```
using System;
using System.Runtime.ExceptionServices;
using System.Threading.Tasks;
Task task = WriteWebRequestSizeAsync(url);
try
{
    while (!task.Wait(100))
    {
        Console.Write(".");
    }
}
catch(AggregateException exception)
{
    exception = exception.Flatten();
    ExceptionDispatchInfo.Capture(
        exception.InnerException??exception).Throw();
}
```

Gdy używana jest metoda ExceptionDispatchInfo.Throw(), kompilator (inaczej niż w przypadku wywołania normalnej instrukcji throw) nie traktuje jej jak instrukcji return. Na przykład jeśli według sygnatury metoda ma zwracać wartość, ale ścieżka kodu z wywołaniem ExceptionDispatchInfo.Throw() nie zwraca danych, kompilator zgłasza błąd informujący o braku zwróconej wartości. Dlatego czasem programista musi po wywołaniu ExceptionDispatchInfo.Throw() dodać instrukcję return, nawet jeśli nie zostanie ona uruchomiona w trakcie wykonywania programu, ponieważ zamiast niej nastąpi zgłoszenie wyjątku.

Koniec
5.0

Porównanie języków — specyfikatory wyjątków w Javie

C# nie udostępnia odpowiedników specyfikatorów wyjątków z Javy. Dzięki specyfikatorom wyjątków kompilator Javy potrafi wykryć, że wszystkie wyjątki zgłoszane w funkcji (lub w hierarchii wywołań funkcji) są albo przechwytywane, albo deklarowane w taki sposób, że mogą zostać ponownie zgłoszone. Zespół pracujący nad językiem C# rozważał dodanie podobnego mechanizmu, ale doszedł do wniosku, że potencjalne korzyści nie równoważą kosztów. Dlatego nie trzeba tworzyć listy wszystkich wyjątków, jakie mogą wystąpić na stosie wywołań, ale też nie da się łatwo ustalić potencjalnych wyjątków dla danego wywołania. Okazuje się jednak, że nie jest to możliwe także w Javie. Wywołania metod wirtualnych lub używanie późnego wiązania (na przykład z wykorzystaniem mechanizmu refleksji) sprawiają, że na etapie komplikacji nie da się ustalić pełnej listy wyjątków, które metoda może zgłosić.

Początek
2.0

Ogólny blok catch

Język C# wymaga, by każdy obiekt wyjątku zgłoszanego w kodzie był typu pochodnego od `System.Exception`. Jednak nie we wszystkich językach obowiązuje ten wymóg. Na przykład w C++ zgłaszać można obiekty wyjątków dowolnego typu (w tym zarządzane wyjątki typów, które nie są pochodne od `System.Exception`). W C# wszystkie wyjątki¹ niezależnie od tego, czy są pochodne od klasy `System.Exception`, są przekazywane do podzespołów języka C# tak, jakby były pochodne od tej klasy. W efekcie bloki `catch` przeznaczone dla wyjątków typu `System.Exception` przechwytyują wszystkie wyjątki, które nie zostały przechwycone przez wcześniejsze bloki.

Język C# udostępnia też **ogólny blok catch** (`catch{}`), działający tak jak `block catch(System.Exception exception)`. Blok ogólny nie wymaga podawania typu ani nazwy zmiennej. Ogólny blok `catch` musi się znajdować na końcu listy bloków. Ponieważ blok ogólny jest identyczny z blokiem `catch(System.Exception exception)` i musi występować jako ostatni, kompilator zgłasza ostrzeżenie, jeśli oba wymienione bloki pojawiają się w tej samej instrukcji `try/catch` (w takiej sytuacji blok ogólny nigdy nie zostanie wykonany).

Koniec
2.0

ZAGADNIENIE DLA ZAAWANSOWANYCH

Wewnętrzne mechanizmy ogólnego bloku catch

Kod CIL odpowiadający ogólnemu blokowi `catch` to blok `catch(object)`. Dlatego niezależnie od typu zgłoszanego wyjątku pusty blok `catch` na pewno go przechwyci. Co ciekawe, w języku C# nie można jawnie zadeklarować bloku `catch(object)`. Dlatego nie ma możliwości przechwytywania wyjątków typów niepochodnych od `System.Exception` i uzyskania obiektu wyjątku na potrzeby analiz.

Niezarządzane wyjątki z języków takich jak C++ zwykle prowadzą do zgłaszania wyjątków typu `System.Runtime.InteropServices.SEHException`, który jest pochodny od `System.Exception`. Dlatego za pomocą ogólnego bloku `catch` przechwytywane są nie tylko wyjątki typów niezarządzanych, ale też typów zarządzanych, które nie są pochodne od typu `System.Exception` (na przykład typu `string`).

¹ Jest tak od wersji C# 2.0.

Wskazówki związane z obsługą wyjątków

Mechanizmy obsługi wyjątków zapewniają potrzebną strukturę dla starszych technik zarządzania błędami. Jeśli jednak obsługa wyjątków jest stosowana chaotycznie, może prowadzić do nieoczekiwanych efektów. Przedstawione poniżej wskazówki obejmują najlepsze praktyki z zakresu obsługi wyjątków.

- *Przechwytyuj tylko te wyjątki, które potrafisz obsłużyć.*

Zwykle niektóre typy wyjątków można obsłużyć, natomiast w przypadku innych nie jest to wykonalne. Na przykład próba otwarcia pliku na wyłączność do odczytu i zapisu może skutkować wyjątkiem typu `System.IO.IOException`, jeśli plik jest już używany. Po przechwyceniu wyjątku tego rodzaju kod może poinformować użytkownika, że plik jest już otwarty, i umożliwić anulowanie operacji lub ponowienie próby. Przechwytywać należy tylko te wyjątki, dla których znany jest tok postępowania. Obsługę wyjątków pozostałych typów należy pozostawić jednostkom wywołującym z wyższych poziomów stosu wywołań.

- *Nie ukrywaj wyjątków, które nie są w pełni obsługiwane.*

Początkujący programiści często czują pokusę, by przechwytywać wszystkie wyjątki, a następnie kontynuować wykonywanie programu, zamiast informować użytkownika o nieobsłużonym wyjątku. Jednak takie postępowanie może prowadzić do niewykrycia krytycznych problemów w systemie. Jeśli kod nie wykonał jawnie określonych działań, by obsługiwać wyjątek (lub nie wykrył, że dany wyjątek jest nieszkodliwy), w blokach `catch` należy ponownie zgłaszać wyjątki, zamiast przechwytywać je i ukrywać przed jednostką wywołującą. W większości sytuacji blok `catch(System.Exception)` i ogólny blok `catch` powinny się znajdować wyżej na stosie wywołań, chyba że taki blok ponownie zgłasza wyjątek.

- *Rzadko korzystaj z bloków dla typu System.Exception i z ogólnych bloków catch.*

Prawie wszystkie typy wyjątków są pochodne od typu `System.Exception`.

Jednak najlepszym sposobem obsługi niektórych wyjątków pochodnych od tego typu jest rezygnacja z ich obsługi lub bezpieczne zamknięcie aplikacji stosunkowo szybko po wykryciu problemu. Dotyczy to na przykład wyjątków typów `System.OutOfMemoryException` i `System.StackOverflowException`.

Na szczęście domyślnie przyjmuje się², że nie da się przywrócić poprawnego stanu systemu po wystąpieniu takich wyjątków. Dlatego nawet jeśli kod nie zgłasza ich ponownie po przechwyceniu, i tak są one powtórnie zgłaszane przez środowisko CLR. Są to wyjątki środowiska uruchomieniowego i programista nie może napisać kodu, który pozwala przywrócić poprawny stan po ich wystąpieniu. Dlatego po zgłoszeniu takich wyjątków najlepiej zamknąć aplikację.

Początek
4.0

Koniec
4.0

² Jest tak od wersji CLR 4. Wcześniej kod powinien przechwytywać takie wyjątki tylko po to, aby ureguliować kod porządkujący zasoby lub awaryjnie wykonujący potrzebne zadania (na przykład zapisywanie nietrwałych danych); następnie aplikacja była zamknięta lub kod ponownie zgłaszał wyjątek za pomocą instrukcji `throw`;

■ Unikaj informowania o wyjątku i rejestrowania go na niskich poziomach stosu wywołań.

Programiści często czują pukusę, by rejestrować wyjątki w dzienniku lub informować o nich na jak najniższym poziomie stosu wywołań. Jednak w takim miejscu rzadko można w pełni obsłużyć wyjątek. Częściej wyjątki są wtedy zgłaszane ponownie. W blokach catch na niskich poziomach stosu wywołań nie należy rejestrować wyjątków w dzienniku ani informować o nich użytkowników. Jeśli wyjątek zostanie zarejestrowany i ponownie zgłoszony, jednostki wywołujące z wyższych poziomów stosu wywołań mogą wykonać podobne operacje, co spowoduje powtarzające się wpisy o wyjątku w dzienniku. Co gorsza, informacje o wyjątku wyświetlane użytkownikowi mogą być nieodpowiednie do typu aplikacji. Na przykład komunikaty z instrukcji `System.Console.WriteLine()` w aplikacji dla systemu Windows nigdy nie będą widoczne dla użytkowników, a wyświetlenie okna dialogowego w nienadzorowanym procesie uruchamianym w wierszu poleceń może zostać niezauważone i skutkować zatrzymaniem programu. Operacje związane z rejestrowaniem informacji w dzienniku i wyświetlaniem ich użytkownikom należy dodawać tylko na wyższych poziomach stosu wywołań.

■ W blokach catch stosuj instrukcję `throw`; zamiast `throw <obiekt_wyjątku>`.

W bloku catch można ponownie zgłosić wyjątek. Na przykład w bloku catch (`ArgumentNullException exception`) możesz umieścić wywołanie `throw exception`. Jednak ponowne zgłoszenie wyjątku w ten sposób powoduje zapisanie na stosie informacji związanych z lokalizacją nowego zgłoszenia. Nie jest wtedy uwzględniana lokalizacja pierwotnego zgłoszenia. Dlatego jeśli nie chcesz celowo ukryć pierwotnego stosu wywołań lub zmienić typu wyjątku, stosuj instrukcję `throw`. Dzięki temu w górze stosu wywołań przekazany zostanie pierwotny wyjątek.

■ Stosuj warunki wyjątków, aby uniknąć ponownego zgłaszania wyjątków w blokach catch.

W sytuacjach, gdy przechwytyujesz wyjątek, którego nie potrafisz poprawnie obsłużyć, przez co musisz zgłosić go ponownie, staraj się stosować warunki wyjątków, aby uniknąć przechwytywania takich wyjątków.

■ Unikaj zgłaszania wyjątków w warunkach wyjątków.

Gdy tworzysz warunek wyjątku, unikaj w nim kodu, który sam może skutkować zgłoszeniem wyjątku. Wyjątek w warunku powoduje, że warunek zostaje uznany za niespełniony, a dane wystąpienie wyjątku jest wtedy ignorowane. Dlatego gdy kod warunku jest skomplikowany, rozważ umieszczenie go w odrębnej metodzie z blokiem `try/catch`, który jawnie przechwytuje ewentualne wyjątki.

■ Unikaj stosowania wyrażeń warunkowych wyjątku, których wartość z czasem może się zmienić.

Jeśli w wyrażeniu warunkowym wyjątku sprawdzane są na przykład używane w wyjątkach komunikaty, które mogą się zmienić ze względu na tłumaczenie programu (lub w wyniku modyfikacji samego komunikatu), oczekiwany wyjątek może nie zostać przechwycony, co niespodziewanie zmienia logikę działania kodu. Dlatego dbaj o to, by wyrażenia warunkowe wyjątku w blokach były zawsze poprawne.

Początek
6.0Koniec
6.0

■ *Zachowaj ostrożność, gdy w ponownym zgłoszeniu podajesz inny wyjątek.*

W bloku catch zgłoszenie innego wyjątku nie tylko zmienia lokalizację zgłoszenia, ale też ukrywa pierwotny wyjątek. Aby zachować pierwotny wyjątek, ustaw właściwość InnerException nowego wyjątku (zwykle można to zrobić w konstruktorze wyjątku). Ponowne zgłaszanie innego wyjątku należy ograniczyć do następujących sytuacji:

1. *Gdy zmiana typu wyjątku pozwala dokładniej przedstawić problem.*

Na przykład w wywołaniu metody Logon(User user) ponowne zgłoszenie wyjątku innego typu w reakcji na niedostępność pliku z listą użytkowników jest prawdopodobnie lepszym rozwiązaniem niż przekazywanie dalej wyjątku System.IO.IOException.

2. *Gdy w pierwotnym wyjątku znajdują się poufne dane.*

Jeśli w opisany wyżej scenariuszu pierwotny wyjątek System.IO.IOException obejmuje ścieżkę do pliku i ujawnia w ten sposób poufne informacje związane z bezpieczeństwem systemu, wyjątek należy opakować. Oczywiście nie należy wtedy przypisywać pierwotnego wyjątku do właściwości InnerException. Co zabawne, w jednej z pierwszych wersji środowiska CLR v1 (nie była to nawet wersja alfa) dostępny był wyjątek wyświetlający informacje o następującej postaci: „Wyjątek bezpieczeństwa: nie masz uprawnień do poznania ścieżki do pliku C:\temp\foo.txt”.

3. *Gdy typ wyjątku jest zbyt specyficzny, by jednostka wywołująca mogła go poprawnie obsłużyć.*

Na przykład zamiast zgłaszać wyjątek dotyczący konkretnego systemu bazodanowego, można zastosować ogólniejszy wyjątek. Pozwala to uniknąć pisania kodu specyficznego dla danej bazy danych na wyższych poziomach stosu wywołań.

Wskazówki

UNIKAJ informowania o wyjątku i rejestrowania go na niskich poziomach stosu wywołań.

NIE przesadzaj z przechwytywaniem wyjątków. Jeśli na niższych poziomach stosu wywołań nie wiadomo, jak programowo rozwiązać dany błąd, wyjątek należy przekazać wyżej.

ROZWAŻ przechwytywanie specyficznych wyjątków, jeśli rozumiesz, dlaczego zostały zgłoszone w danym kontekście, i wiesz, jak programowo zareagować na błąd.

UNIKAJ przechwytywania wyjątków typów System.Exception i System.SystemException. Przechwytyj je tylko w blokach obsługi wyjątków na najwyższym poziomie stosu wywołań, by wykonać końcowe operacje porządkujące zasoby przed ponownym zgłoszeniem wyjątku.

WBLOKACH catch posługuj się instrukcją throw zamiast throw <obiekt wyjątku>.

STOSUJ warunki wyjątków, aby unikać ponownego zgłaszania wyjątków w blokach catch.

ZACHOWAJ ostrożność, gdy ponownie zgłaszasz nowe wyjątki.

UNIKAJ zgłaszania wyjątków w wyrażeniach warunkowych wyjątku.

UNIKAJ tworzenia wyrażeń warunkowych wyjątku, których działanie z czasem może się zmienić.

Definiowanie niestandardowych wyjątków

Gdy zgłoszenie wyjątku jest najlepszym rozwiązańiem, warto stosować wyjątki z platformy, ponieważ są dobrze znane i rozumiane. Na przykład zamiast zgłaszać niestandardowy wyjątek informujący o błędny argumencie, wykorzystaj typ `System.ArgumentException`. Jeśli jednak programista stosujący określony interfejs API wykonuje specjalne operacje (i logika obsługi wyjątków zmieni się po dodaniu niestandardowego wyjątku), warto zdefiniować niestandardowy wyjątek. Na przykład jeśli interfejs API do obsługi map otrzyma adres z błędny kodem pocztowym, zamiast zgłaszać wyjątek `System.ArgumentException`, lepiej będzie zgłosić niestandardowy wyjątek `InvalidAddressException`. Ważne jest, czy w jednostce wywołującej prawdopodobne jest użycie specyficznego bloku `catch` dla wyjątków typu `InvalidAddressException` (i ze specjalną ich obsługą) zamiast standardowego bloku `catch` dla wyjątków `System.ArgumentException`.

Aby zdefiniować niestandardowy wyjątek, wystarczy utworzyć klasę pochodną od `System.Exception` lub innego typu wyjątków. Przykładowy kod znajdziesz na listingu 11.4.

Listing 11.4. Tworzenie niestandardowego wyjątku

```
class DatabaseException : System.Exception
{
    public DatabaseException(
        string? message,
        System.Data.SqlClient.SqlException? exception)
        : base(message, innerException: exception)
    {
        // ...
    }

    public DatabaseException(
        string? message,
        System.Data.OracleClient.OracleException? exception)
        : base(message, innerException: exception)
    {
        // ...
    }

    public DatabaseException()
    {
        // ...
    }

    public DatabaseException(string message)
    {
        // ...
    }

    public DatabaseException(
        string? message, Exception? exception)
        : base(message, innerException: exception)
    {
        // ...
    }
}
```

Ten niestandardowy wyjątek można utworzyć na potrzeby opakowywania specyficznych wyjątków dotyczących baz danych. Ponieważ bazy Oracle i SQL Server w reakcji na podobne błędy zgłaszą odmienne wyjątki, w aplikacji można zdefiniować niestandardowy wyjątek, aby ustąpić specyficzne dla bazy danych wyjątki przez opakowanie ich we wspólny typ, który aplikacja może obsługiwać w standardowy sposób. Dzięki temu niezależnie od tego, czy aplikacja używa na zapleczu bazy Oracle, czy SQL Server, na wyższym poziomie stosu wywołań do obsługi zgłaszanych błędów można wykorzystać ten sam blok catch.

Jednym wymogiem dotyczącym niestandardowych wyjątków jest to, że muszą być pochodne od typu System.Exception lub jednego z jego typów pochodnych. Jednak w trakcie rozwijania niestandardowych wyjątków warto też stosować się do kilku dobrych praktyk.

- Wszystkie typy wyjątków powinny mieć przyrostek Exception. Pozwala to łatwo określić przeznaczenie typu na podstawie jego nazwy.
- Prawie wszystkie wyjątki powinny udostępniać konstruktor bezparametryczny, konstruktor z parametrem w postaci łańcucha znaków oraz konstruktor ze zbiorem parametrów obejmujących łańcuch znaków i wewnętrzny wyjątek. Ponadto ponieważ wyjątki zwykle są tworzone w tej samej instrukcji, w której są zgłasiane, warto udostępnić przekazywanie do konstruktora dodatkowych danych. Oczywiście nie trzeba tworzyć wszystkich wymienionych konstruktorów, jeśli niezbędne są pewne dane, a konkretny rodzaj konstruktora nie umożliwia ich podania.
- Łańcuch dziedziczenia powinien być stosunkowo płytki (i obejmować do pięciu poziomów).

Wewnętrzny wyjątek jest ważny, gdy kod ponownie zgłasza wyjątek inny od pierwotnie zgłoszonego. Na przykład jeśli baza danych zgłosiła wyjątek typu System.Data.SqlClient.SqlException, który został przechwycony w warstwie dostępu do danych, gdzie jest ponownie zgłaszanego jako wyjątek typu DatabaseException, konstruktor typu DatabaseException przyjmujący wyjątek wewnętrzny (na przykład typu SqlException) pozwala zachować pierwotny wyjątek typu SqlException we właściwości InnerException. Dzięki temu gdy potrzebne są dodatkowe informacje na temat pierwotnego wyjątku, programista może pobrać wyjątek z właściwości InnerException (na przykład za pomocą polecenia exception.InnerException).

Wskazówka

UNIKAJ tworzenia głębokich hierarchii wyjątków.

NIE twórz nowego typu wyjątku, jeśli nie będzie on obsługiwany inaczej niż istniejące wyjątki ze środowiska CLR. Wtedy zgłaszasz istniejące wyjątki z platformy.

TWÓRZ nowy typ wyjątku, by informować o niestandardowym błędzie w aplikacji, którego nie da się zgłosić za pomocą istniejącego wyjątku ze środowiska CLR i który można programowo obsługiwać w sposób inny niż istniejące wyjątki.

UDOSTĘPNIJ konstruktor bezparametryczny we wszystkich niestandardowych typach wyjątków. Udostępnij też konstruktory, które przyjmują komunikat i wewnętrzny wyjątek.

DODAWAJ przyrostek Exception do nazw klas wyjątków.

TWÓRZ wyjątki w taki sposób, by mogły być szeregowane przez środowisko uruchomieniowe.

ROZWAŻ dodanie do wyjątków właściwości pozwalających na programowy dostęp do dodatkowych informacji na temat wyjątku.

ZAGADNIENIE DLA ZAAWANSOWANYCH

Wyjątki z obsługą serializacji

Obiekty z obsługą serializacji to obiekty, które środowisko uruchomieniowe może zapisać w strumieniu (na przykład w strumieniu plikowym) i odtworzyć na podstawie danych ze strumienia. W przypadku wyjątków ten mechanizm może być potrzebny w niektórych technologiach komunikacji w środowisku rozproszonym. Aby umożliwić obsługę serializacji, w deklaracji wyjątku trzeba albo dodać atrybut System.SerializableAttribute, albo zaimplementować interfejs ISerializable. Ponadto należy udostępnić konstruktor przyjmujący parametry typów System.Runtime.Serialization.SerializationInfo i System.Runtime.Serialization.StreamingContext. Na listingu 11.5 znajdziesz przykładowy kod używający atrybutu System.SerializableAttribute.

Listing 11.5. Definiowanie wyjątku z obsługą serializacji

```
// Dodawanie obsługi serializacji za pomocą atrybutu.  
[Serializable]  
class DatabaseException : System.Exception  
{  
    // ...  
  
    // Konstruktor używany do deserializowania wyjątków.  
    public DatabaseException(  
        SerializationInfo serializationInfo,  
        StreamingContext context)  
        : base(serializationInfo, context)  
    {  
        //...  
    }  
}
```

W klasie DatabaseException pokazano zarówno wspomniany wcześniej atrybut, jak i konstruktor potrzebny do tego, by wyjątki można było serializować.

Warto zauważyć, że w platformie .NET Core typ System.SerializableAttribute został udostępniony dopiero w specyfikacji .NET Standard 2.0. Jeśli piszesz kod kompilowany w różnych platformach, w tym w narzędziach zgodnych ze starszymi wersjami specyfikacji .NET Standard, rozważ zdefiniowanie własnego typu System.SerializableAttribute działającego jako **polyfill**. Polyfill to kod, który uzupełnia lukę w określonej wersji technologii, a tym samym dodaje brakującą funkcję lub przynajmniej jej zastępnik.

Ponowne zgłoszanie opakowanego wyjątku

Zdarza się, że wyjątek zgłoszony na niższym poziomie stosu wywołań nie ma sensu, gdy zostanie przechwycony na wyższym poziomie. Pomyśl na przykład o wyjątku typu System.IO.IOException, który wystąpił, ponieważ na serwerze zabrakło wolnej przestrzeni dyskowej. Maszyna kliencka po przechwyceniu takiego wyjątku może nie znać kontekstu i nie móc ustalić, dlaczego w ogóle wykonywano operacje wejścia-wyjścia. Inny przykład to interfejs API do przetwarzania żądań

współrzędnych geograficznych, który zgłasza wyjątek typu `System.UnauthorizedAccessException` zupełnie niepowiązany z wywołanym interfejsem API. W tym przykładzie jednostka wywołująca nie zna kontekstu i nie potrafi ustalić, co wywołanie kierowane do używanego interfejsu API ma wspólnego z bezpieczeństwem. Z perspektywy kodu, który wywołał dany interfejs API, wyjątki wprowadzają konfuzję, zamiast pomagać zdiagnozować problem. Dlatego zamiast przekazywać takie wyjątki klientowi, warto czasem najpierw złechwycić dany wyjątek, a następnie zgłosić inny, na przykład `InvalidOperationException` lub nawet niestandardowy. Jest to informacja, że system znajduje się w nieprawidłowym stanie. W takich sytuacjach należy ustawić właściwość `InnerException` w nowym wyjątku (zwykle odbywa się to w wywołaniu konstruktora, na przykład `new InvalidOperationException(String, Exception)`). Dzięki temu uzyskiwany jest dodatkowy kontekst, który może zostać wykorzystany do celów diagnostycznych przez osobę lepiej znającą używaną platformę.

Ważnym szczegółem, o którym warto pamiętać w trakcie podejmowania decyzji o opakowywaniu i ponownym zgłaszaniu wyjątku, jest to, że pierwotny ślad stosu, określający kontekst zgłoszenia wyjątku, jest zastępowany nowym śladem stosu z momentu zgłoszenia nowego wyjątku (jeśli programista nie użył obiektu typu `ExceptionDispatchInfo`). Na szczęście gdy pierwotny wyjątek jest umieszczony w opakowującym go wyjątku, nadal można uzyskać dostęp do pierwotnego śladu stosu.

Docelowym odbiorcą wyjątku jest programista, który napisał (zapewne błędnie) kod wywołujący dany interfejs API. Dlatego powinieneś zapewnić jak najwięcej informacji określających, co programista zrobił źle, a także — co jest prawdopodobnie ważniejsze — jak może rozwiązać problem. Typ wyjątku jest bardzo ważnym elementem mechanizmu przekazywania takich informacji. Dlatego powinieneś starannie wybrać używany typ.

Wskazówka

ROZWAŻ opakowanie specyficznych wyjątków zgłaszanych na niskich poziomach w bardziej odpowiednie wyjątki, jeśli wyjątek z niskiego poziomu nie ma sensu w kontekście operacji na wyższym poziomie.

USTAWIAJ właściwość `InnerException`, gdy opakujesz wyjątki.

ZA DOCELOWYCH odbiorców wyjątków uważaj programistów. Jeśli to możliwe, określ zarówno problem, jak i jego rozwiązanie.

ZAGADNIENIE DLA POCZĄTKUJĄCYCH

Konwersje z kontrolą i bez kontroli

W ZAGADNIENIU DLA ZAAWANSOWANYCH w rozdziale 2. wyjaśniono, że C# udostępnia specjalne słowa kluczowe pozwalające dodać do bloku z kodem instrukcje dla środowiska uruchomieniowego, które określają, co należy zrobić, jeśli docelowy typ danych jest za mały na przypisywanie dane. Domyslnie jest tak, że jeśli docelowy typ danych nie mieści przypisywanych danych, są one przycinane. Przyjrzyj się na przykład kodowi z listingu 11.6.

Listing 11.6. Przepelenie typu całkowitoliczbowego

```
using System;

public class Program
{
    public static void Main()
    {
        // Wartość int.MaxValue jest równa 2147483647.
        int n = int.MaxValue;
        n = n + 1 ;
        System.Console.WriteLine(n);
    }
}
```

Efekt wykonania kodu z listingu 11.6 pokazano w danych wyjściowych 11.1.

DANE WYJŚCIOWE 11.1.

```
-2147483648
```

Kod z listingu 11.6 wyświetla w konsoli wartość -2147483648. Jeśli jednak umieścisz kod w bloku z kontrolą lub zastosujesz opcję checked w momencie uruchamiania kompilatora, środowisko uruchomieniowe zgłosi wyjątek typu System.OverflowException. Aby utworzyć blok z kontrolą konwersji, należy zastosować słowo kluczowe checked, co pokazano na listingu 11.7.

Listing 11.7. Przykładowy blok z kontrolą konwersji

```
using System;

public class Program
{
    public static void Main()
    {
        checked
        {
            // Wartość int.MaxValue jest równa 2147483647.
            int n = int.MaxValue;
            n = n + 1 ;
            System.Console.WriteLine(n);
        }
    }
}
```

Jeśli obliczenia dotyczą tylko stałych, domyślnie są kontrolowane. Wynik działania kodu z listingu 11.7 pokazano w danych wyjściowych 11.2.

DANE WYJŚCIOWE 11.2.

```
Unhandled Exception: System.OverflowException: Arithmetic operation
resulted in an overflow. at Program.Main() in ...Program.cs:line 12
```

W danych wyjściowych 11.2 pokazano, że wyjątek jest zgłoszany, jeśli w czasie wykonywania programu w kontrolowanym bloku nastąpi przypisanie prowadzące do przepelnienia³. Zauważ, że informacja o lokalizacji błędu z danych wyjściowych 11.2 (Program.cs:line X) pojawia się tylko w komplikacjach diagnostycznych (z opcją /Debug w kompilatorze).

Jeśli chcesz, aby konwersje w całym projekcie były kontrolowane lub niekontrolowane, ustaw właściwość CheckForOverflowUnderflow na wartość true lub false:

```
<PropertyGroup>
  <CheckForOverflowUnderflow>true</CheckForOverflowUnderflow>
</PropertyGroup>
```

Ponadto C# pozwala utworzyć blok niekontrolowany, w którym przypisania skutkujące przepelnieniem prowadzą do przycięcia danych, a nie do zgłoszenia wyjątku (zobacz listing 11.8).

Listing 11.8. Przykładowy blok niekontrolowany

```
using System;

public class Program
{
    public static void Main()
    {
        unchecked
        {
            // Wartość int.MaxValue jest równa 2147483647.
            int n = int.MaxValue;
            n = n + 1 ;
            System.Console.WriteLine(n);
        }
    }
}
```

Efekt działania kodu z listingu 11.8 znajdziesz w danych wyjściowych 11.3.

DANE WYJŚCIOWE 11.3.

```
-2147483648
```

Nawet jeśli w trakcie komplikacji zastosowana zostanie opcja checked, słowo kluczowe unchecked z kodu z listingu 11.8 sprawi, że środowisko uruchomieniowe w trakcie wykonywania tego kodu nie zgłosi wyjątku.

Analogiczne wyrażenia z kontrolą i bez kontroli można stosować w miejscach, gdzie nie można używać instrukcji. Na przykład w inicjatorze pola można podać wyrażenie zamiast instrukcji:

```
int _Number = unchecked(int.MaxValue + 1);
```



³ Oprócz danych wyjściowych 11.2 może — w zależności od tego, czy zainstalowany jest debugger platformy .NET — pojawić się okno dialogowe umożliwiające użytkownikowi przesłanie do Microsoftu komunikatu o błędzie, wyszukanie rozwiązania lub rozpoczęcie debugowania aplikacji.

Podsumowanie

Zgłoszenie wyjątku negatywnie wpływa na wydajność. Jeden wyjątek powoduje wczytanie i przetworzenie wielu informacji ze stosu wywołań (w innych sytuacjach te dane nie są pobierane). Obsługa wyjątku też zajmuje wiele czasu. W rozdziale 5. wspomniano, że wyjątki należy stosować tylko do obsługi nietypowych sytuacji. Interfejsy API powinny zapewniać mechanizmy pozwalające sprawdzić, czy wystąpi wyjątek. Nie należy zmuszać programistów do wywołania konkretnego interfejsu API w celu ustalenia, czy wyjątek zostanie zgłoszony.

W następnym rozdziale znajdziesz wprowadzenie do typów generycznych. Jest to mechanizm z wersji C# 2.0, który znacznie wzbogaca kod napisany w wersji C# 1.0. Typy generyczne sprawiają, że korzystanie z przestrzeni nazw `System.Collections` (używanej wcześniej w prawie każdym projekcie) stało się niemal całkowicie zbędne.

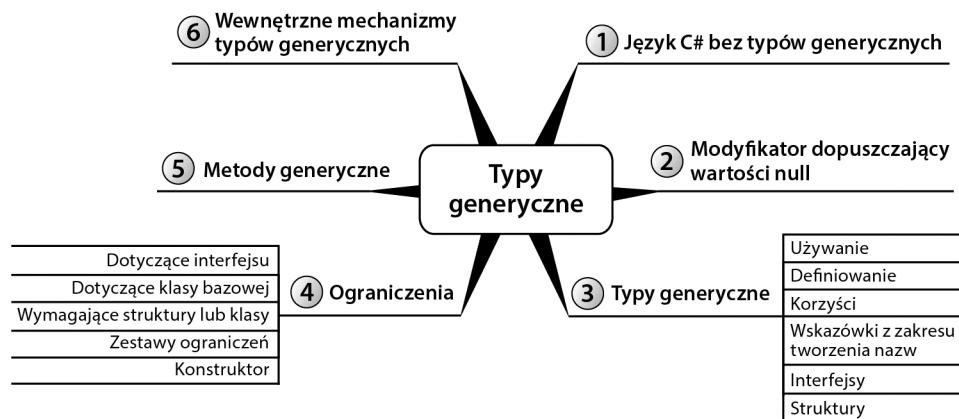
■ ■ ■ 12 ■ ■ ■

Typy generyczne

Początek
2.0

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

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 szczególne 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.

Język C# bez typów generycznych

W ramach omawiania typów generycznych najpierw przeanalizujemy klasę, w której takie typy nie są używane. Ta klasa, System.Collections.Stack, reprezentuje stos, czyli kolekcję obiektów, w której ostatni element dodawany do kolekcji jest pierwszym elementem z niej pobieranym (jest to kolekcja typu „ostatni na wejściu, pierwszy na wyjściu”; ang. *last in, first out* — LIFO). Dwie główne metody klasy Stack, czyli Push() i Pop(), dodają elementy do stosu i usuwają je z niego. Deklaracje tych metod z klasy Stack znajdują się na listingu 12.1.

Listing 12.1. Sygnatury metod klasy System.Collections.Stack

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

2.0

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

Listing 12.2. Obsługa wycofywania operacji w programie symulującym działanie znikopisu

```
using System.Collections;
class Program
{
    // ...

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

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

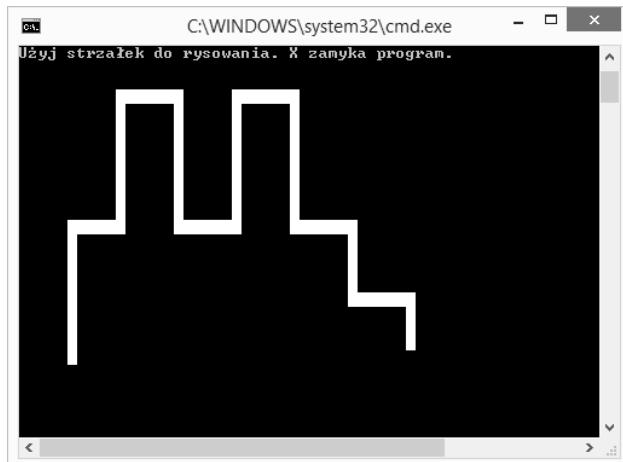
            switch (key.Key)
            {
                case ConsoleKey.Z:
                    // Wymazanie ostatnio narysowanego elementu.
                    if (path.Count >= 1)
                    {
                        currentPosition = (Cell)path.Pop();
                        Console.SetCursorPosition(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

```
case ConsoleKey.DownArrow:  
case ConsoleKey.UpArrow:  
case ConsoleKey.LeftArrow:  
case ConsoleKey.RightArrow:  
    // SaveState()  
    currentPosition = new Cell(  
        Console.CursorLeft, Console.CursorTop);  
    path.Push(currentPosition);  
    break;  
  
default:  
    Console.Beep(); // Dodane w wersji C# 2.0.  
    break;  
}  
}  
while (key.Key != ConsoleKey.X); // Klawisz X pozwala zamknąć program.  
}  
}  
  
public struct Cell  
{  
    // W wersjach starszych niż C# 6.0 należy użyć pola tylko do odczytu.  
    public int X { get; }  
    public int Y { get; }  
    public Cell(int x, int y)  
    {  
        X = x;  
        Y = y;  
    }  
}
```

2.0

Efekt uruchomienia kodu z listingu 12.2 znajdziesz w danych wyjściowych 12.1

DANE WYJŚCIOWE 12.1.



W zmiennej path typu `System.Collections.Stack` program zachowuje wcześniejsze ruchy pędzla, przekazując element niestandardowego typu `Cell` do metody `Stack.Push()` (w wyrażeniu `path.Push(currentPosition)`). Jeśli użytkownik wpisze literę Z (lub wybierze kombinację `Ctrl+Z`), poprzedni ruch pędzla zostaje anulowany. Anulowanie odbywa się przez zdjęcie poprzedniego ruchu pędzla ze stosu za pomocą metody `Pop()`, przeniesienie pozycji kurSORA na wcześniejszą pozycję i wywołanie metody `Undo()`.

Choć ten kod działa, klasa `System.Collections.Stack` ma ważną wadę. Na listingu 12.1 pokazano, że klasa `Stack` przechowuje wartości typu `object`. Ponieważ każdy obiekt w środowisku CLR jest typu pochodnego od klasy `object`, klasa `Stack` nie sprawdza, czy elementy umieszczane w kolekcji są tego samego i odpowiedniego typu. Na przykład zamiast przekazywać zmienną `currentPosition`, możesz przekazać łańcuch znaków zawierający współrzędne X i Y połączone kropką. Kompilator musi zezwalać na zapis wartości niespójnych typów danych, ponieważ klasa `Stack` przyjmuje dowolny obiekt pochodny od klasy `object`. Specyficzny typ obiektu nie ma tu znaczenia.

2.0

Ponadto po pobraniu (za pomocą metody `Pop()`) danych ze stosu należy zrzutować zwróconą wartość na typ `Cell`. Jeśli jednak typ wartości zwrotnej 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, skopiać 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 komplikacji. Problem polega na tym, że klasa `Stack` nie jest tak bezpieczna ze względu na typ, jak można tego oczekiwac po programach w języku C#. Aby zmodyfikować tę klasę i wymusić, by elementy stosu były określonego typu (jednak bez stosowania typów generycznych), należy utworzyć wyspecjalizowaną wersję klasy, przedstawioną na listingu 12.3.

Listing 12.3. Definicja wyspecjalizowanej wersji klasy `Stack`

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

Ponieważ klasa `CellStack` może przechowywać tylko obiekty typu `Cell`, to rozwiązanie wymaga dodania niestandardowej implementacji metod potrzebnych do obsługi stosu, co nie jest wygodne. Utworzenie bezpiecznego ze względu na typ stosu liczb całkowitych wymaga następnej niestandardowej implementacji, a każda z nich jest bardzo podobna do wszystkich pozostałych. To skutkuje powstaniem dużej ilości powtarzającego się nadmiarowego kodu.

ZAGADNIENIE DLA POCZĄTKUJĄCYCH

Inny przykład — typy bezpośrednie z możliwą wartością null

W rozdziale 3. opisano możliwość zadeklarowania zmiennych, które mogą zawierać wartość `null`. Wymaga to użycia modyfikatora `?` w deklaracji zmiennej typu bezpośredniego. Ta możliwość pojawiła się w wersji C# 2.0, ponieważ potrzebne były do tego typy generyczne. Przed ich wprowadzeniem programiści mieli do wyboru dwa rozwiązania.

Pierwsze z nich polegało na zadeklarowaniu typów danych 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.

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; }

    ...
}
```

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.

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 wcisniętą przez użytkownika.
            key = Move();

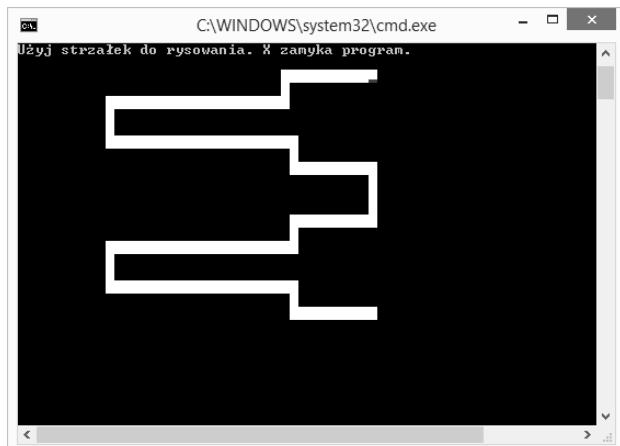
            switch (key.Key)
            {
                case ConsoleKey.Z:
                    // Cofnięcie poprzedniego ruchu pędzla.
                    if (path.Count >= 1)
                    {
                        // Rzutowanie nie jest potrzebne.
                        currentPosition = path.Pop();
                        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);
                    // Wywołaniu Push() można używać tylko zmiennych typu Cell.
                    path.Push(currentPosition);
                    break;
            }
        }
    }
}
```

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

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

DANE WYJŚCIOWE 12.2.



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().

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()
    {
        ...
    }
}
```

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 komplikacji informujący, że nie istnieje wersja przesiązonej 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 komplikacji zmniejsza prawdopodobieństwo wystąpienia wyjątków typu `InvalidOperationException` w czasie wykonywania programu.
3. Używanie typów bezpośrednich w składowych klasy generycznej nie powoduje opakowywania wartości tych typów w typ `object`. Na przykład metody `path.Pop()` i `path.Push()` nie wymagają opakowania elementu w momencie dodawania go i wypakowywania w trakcie usuwania.
4. Typy generyczne w języku C# zmniejszają ilość kodu. Pozwalają zachować korzyści, jakie dają specyficzne wersje klasy, ale nie powodują analogicznych kosztów. Nie trzeba na przykład definiować nowej klasy `CellStack`.
5. Wydajność kodu rośnie, ponieważ nie jest potrzebne rzutowanie z typu `object`. Eliminuje to operację sprawdzania typu. Inną przyczyną wzrostu wydajności jest to, że nie trzeba opakowywać wartości typów bezpośrednich.
6. Typy generyczne zmniejszają ilość zajmowanej pamięci, ponieważ nie trzeba opakowywać wartości. Dzięki temu program zużywa mniej pamięci na stercie.

7. Kod staje się bardziej czytelny, ponieważ jest w nim mniej operacji sprawdzania typów przy rzutowaniu i mniej implementacji specyficznych typów.
8. Edytory wspomagające pisanie kodu za pomocą jednej z odmian mechanizmu IntelliSense bezpośrednio obsługują wartości zwarcane 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; }
}
```

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; }
}
```

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
```

```

{
    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 umożliwiałby jawnie zaimplementowanie tylko jednej właściwości `IContainer.Items`.

2.0

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

Wskazówka

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

Definiowanie konstruktora i finalizatora

Zaskoczeniem może się okazać to, że konstruktory (i finalizator) klasy lub struktury generycznej nie wymagają parametrów określających typ. Oznacza to, że zapis `Pair<T>(){}...` nie jest konieczny. W klasie `Pair` na listingu 12.11 konstruktor jest zadeklarowany z sygnaturą `public Pair(T first, T second)`.

Listing 12.11. Deklarowanie konstruktora typu generycznego

```

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

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

```

Określanie wartości domyślnej 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 komplikacji, 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, wzystąpi błąd komplikacji

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

    // ...
}
```

2.0

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.

Początek
7.0

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

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)
```

```
{  
    First = first;  
    Second = second;  
}  
  
public TFirst First { get; set; }  
public TSecond Second { get; set; }  
}
```

Gdy używasz klasy `Pair<TFirst, TSecond>`, powinieneś podać oba parametry określające typ w nawiasie ostrym w deklaracji i przy inicjowaniu obiektu. Następnie należy stosować właściwe typy dla parametrów metod w ich wywołaniach. To podejście pokazano na listingu 12.15.

Listing 12.15. Używanie typu z kilkoma parametrami określającymi typ

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

2.0

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

Wskazówka

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

ZAGADNIENIE DLA POCZĄTKUJĄCYCH

Początek
7.0

Krotki — typy o różnej arności

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

Listing 12.16. Przesłanianie definicji typów na podstawie arności

```
public class ValueTuple { ... }  
public class ValueTuple<T1>:  
    IStructuralEquatable, IStructuralComparable, IComparable {...}  
public class ValueTuple<T1, T2>: ... {...}  
public class ValueTuple<T1, T2, T3>: ... {...}
```

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

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

2.0

Inną ciekawą składową rodziny klas reprezentujących krotki jest niegeneryczna klasa `ValueTuple`. Ta klasa ma osiem statycznych metod fabrycznych, które służą do tworzenia obiektów różnych generycznych typów `Tuple`. Choć każdy typ generyczny umożliwia bezpośrednie tworzenie obiektów za pomocą konstruktora, metody fabryczne z klasy `Tuple` automatycznie wykrywają typy argumentów, gdy wywoływana jest metoda `Create()`. W C# 7.0 nie ma to znaczenia, ponieważ kod jest bardzo prosty: `var keyValuePair = ("555-55-5555", new Contact("Inigo Montoya"))` (przy założeniu, że nie są używane elementy nazwane). Jednak, co pokazano na listingu 12.17, stosowanie metody `Create()` w połączeniu z inferencją typów upraszcza pracę w C# 6.0.

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

```
#if !PRECSHARP7
(string, Contact) keyValuePair;
keyValuePair =
    "555-55-5555", new Contact("Inigo Montoya"));
#else // W wersjach starszych niż C# 7.0 należy stosować składnię System.ValueTupe<string,Contact>.
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 powszechnie 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 dotyczyć parametru właśnie z tego typu. Na szczeście ponowne użycie w typie zagnieżdżonym określającego typ parametru o wykorzystanej już nazwie powoduje, że kompilator wyświetla ostrzeżenie. Zapobiega to przypadkowemu użyciu parametrów o tej samej nazwie (zobacz listing 12.18).

Listing 12.18. Zagnieżdżone typy generyczne

```
class Container<T1, T2>
{
    // Klasa zagnieżdżona dziedzicząca 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.

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.

2.0

Załóżmy, że chcesz, by drzewo sortowało wartości w obiekcie typu `Pair<T>` przypisywanym do właściwości `SubItems`. Aby posortować dane, akcesor `set` właściwości `SubItems` używa metody `CompareTo()` z podanego klucza. Ilustruje to listing 12.20.

Listing 12.20. Do działania interfejsu potrzebny jest parametr określający typ

```
public class BinaryTree<T>
{
    public 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 niejawnnej konwersji.
            first = value.First; // Konieczne jest jawnie 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;
}
```

W trakcie komplikacji 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 komplikacji, 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

```
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;
}
```

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 (`InvalidOperationException`). 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 komplikacji, 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.

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 oczekiwąć, 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.

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

delegata (bez używania metody `DynamicInvoke()`), to można wywoływać delegaty za pomocą bezpośredniej referencji do typu `T` na etapie komplikacji. 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`.

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`.

Początek
8.0Koniec
7.3

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`.

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

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

    public TKey Key { get; set; }
}

2.0

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

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;
}
```

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);
}
...
```

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 listingu 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

```
// '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 komplikacji. 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).

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.

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, który 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.

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 uniemożliiłoby kompilatorowi określenie na etapie komplikacji, 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)
```

```

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 użyтыmi 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.

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 zgłoszony jest błąd. Po zastanowieniu można zauważyc, 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.

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:

```
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. Założmy, 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.

Listing 12.40. Konwersja między typami generycznymi z różnymi parametrami określającymi typ

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

Jednak dlaczego jest to niedozwolone? Dlaczego typy `List<T>` i `Pair<T>` nie są kowariantne? Na listingu 12.41 pokazano, co by się stało, gdyby język C# zapewniał nieograniczoną kowariancję.

Listing 12.41. Rezygnacja z kowariancji pozwala zachować jednolitość typów

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

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

2.0

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 zauważysz, że oba opisane wcześniej problemy związane z nieograniczoną kowariancją wynikają z tego, że generyczna para i generyczna lista umożliwiają zapis przechowywanych w nich danych. Założmy, 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.

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 Buttercup"),
                new Contact("Inigo Montoya") );
        IReadOnlyPair<PdaItem> pair = contacts;
        PdaItem pdaItem1 = pair.First;
        PdaItem pdaItem2 = pair.Second;
    }
}

```

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

Mimo to kod z listingu 12.42 także się nie skompiluje. W wersji C# 4 dodano jednak obsługę bezpiecznej kowariancji. Aby określić, że generyczny interfejs ma umożliwiać kowariancję względem jednego z parametrów określających typ, ten parametr trzeba zadeklarować z modyfikatorem out. Na listingu 12.43 pokazano, jak zmodyfikować deklarację interfejsu, by informowała, że należy umożliwić kowariancję.

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

```

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

```

Dodanie modyfikatora `out` do parametru określającego typ w interfejsie `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 założmy, że dwa typy, `X` i `Y`, są powiązane w taki sposób, że każdą wartość typu `X` można przekształcić na wartość typu `Y`. Jeśli typy `I<X>` i `I<Y>` zawsze spełniają tę samą relację „w drugą stronę”, czyli każdą wartość typu `I<Y>` można przekształcić na typ `I<X>`, to `I<T>` jest kontrawariantny względem `T`.

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

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

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

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 12.44².

Listing 12.44. Kontrawariancja dzięki zastosowaniu modyfikatora `in` do parametru określającego typ

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

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

class Program
{
    class FruitComparer : ICompareThings<Fruit>
    {
        ...
    }
    static void Main()
    {
        // Dzwolone od wersji C# 4.0.
        ICompareThings<Fruit> fc = new FruitComparer();
        Apple apple1 = new Apple();
        Apple apple2 = new Apple();
        Orange orange = new Orange();
        // Obiekty 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);
    }
}
```

2.0

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.

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);  
}  
// Założmy, że przedstawiony wyżej kod jest poprawny.  
// Zobacz, jakie problemy mogą wystąpić.  
class FruitPairInitializer : IPairInitializer<Fruit>  
{  
    // Kod inicjuje parę obiektów typu Fruit  
    // wartościami typów Orange i Apple.  
    public void Initialize(IPair<Fruit> pair)  
    {  
        pair.First = new Orange();  
        pair.Second = new Apple();  
    }  
}  
// Dalej w kodzie.  
var f = new FruitPairInitializer();  
// Gdyby kontrawariancja była tu dozwolona, ten kod byłby poprawny:  
IPairInitializer<Apple> a = f;  
// Poniższy kod zapisuje obiekt typu Orange w obiekcie z parą obiektów typu Apple.  
a.Initialize(new Pair<Apple>());
```

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

Obsługa niezabezpieczonej kowariancji w tablicach

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

Niestety, C# umożliwia kowariancję w tablicach, choć ta operacja nie jest bezpieczna ze względu na typ. Na przykład instrukcja `Fruit[] fruits = new Apple[10];` jest w języku C# w pełni poprawna. Jeśli potem wykonasz wyrażenie `fruits[0] = new Orange();`, środowisko uruchomieniowe zgłosi wyjątek informujący o naruszeniu bezpieczeństwa typu. Bardzo kłopotliwe jest to, że nie zawsze można poprawnie przypisać obiekt typu Orange do tablicy elementów typu Fruit, ponieważ w rzeczywistości może to być tablica elementów typu Apple. Problem ten dotyczy nie tylko języka C#, ale wszystkich języków ze środowiska CLR, w których używana jest implementacja tablic ze środowiska uruchomieniowego.

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

2.0

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 obsługiwany zarówno przez kompilator, jak i przez platformę. Na przykład aby uniknąć opakowywania obiektów, używana jest inna implementacja typów generycznych w zależności od tego, czy jako parametr określający typ podano typ bezpośredni, czy typ referencyjny.

ZAGADNIENIE DLA ZAAWANSOWANYCH

Reprezentacja typów generycznych w kodzie CIL

Po skompilowaniu klasa generyczna tylko nieznacznie różni się od klasy niegenerycznej. W wyniku komplikacji powstają metadane i kod CIL. Kod CIL jest sparametryzowany, by umożliwić zastosowanie typu podanego przez użytkownika w określonym miejscu kodu. Założymy, 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 wątkiem uwagą 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
    ...
}
```

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 „natynie” korzysta z typu bezpośredniego.

ZAGADNIENIE DLA ZAAWANSOWANYCH

Tworzenie obiektów typów generycznych opartych na typach referencyjnych

Typy generyczne oparte na typach referencyjnych działają nieco inaczej. Gdy po raz pierwszy tworzony jest obiekt typu generycznego opartego na typie referencyjnym, środowisko uruchomieniowe tworzy wyspecjalizowany typ generyczny, w którym w kodzie CIL za parametry określające typ podstawiany jest typ object (a nie typ określony w argumencie).

Później za każdym razem, gdy tworzony jest obiekt danego typu generycznego opartego na typie referencyjnym, środowisko uruchomieniowe ponownie wykorzystuje wcześniej wygenerowaną wersję tego typu generycznego — także wtedy, jeśli ten typ referencyjny jest inny niż wcześniej.

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.

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 aktualizowanej 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 komplikacji, 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.

■ ■ ■ 13 ■ ■ ■

Delegaty i wyrażenia lambda

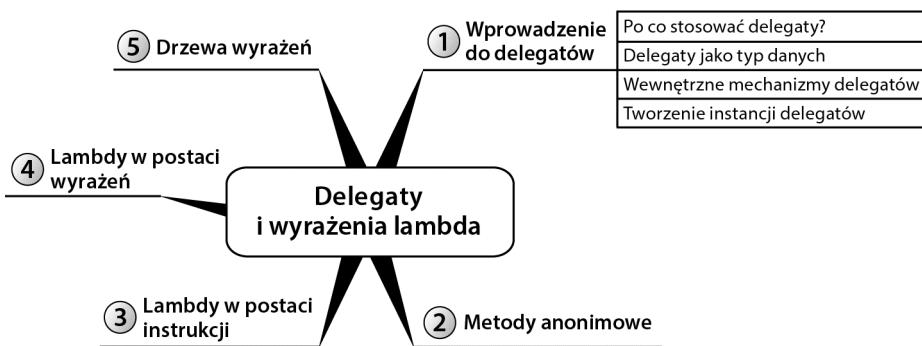
Początek
3.0

W POPRZEDNICH ROZDZIAŁACH DOKŁADNIE OPISANO, jak tworzyć klasy hermetyzujące dane i wykonywane na nich operacje. Gdy zaczniesz tworzyć więcej klas, dostrzeżesz wzorce w relacjach między nimi. Jednym z często występujących wzorców jest przekazywanie obiektu do metody tylko po to, by metoda mogła wywołać jedną z metod tego obiektu. Na przykład jeśli przekażesz do metody referencję do obiektu typu `IComparer<int>`, prawdopodobne jest, że ta metoda wywoła metodę `Compare()` przekazanego obiektu. W tym podejściu interfejs jest tylko narzędziem do przekazania referencji do jednej metody, którą należy wywołać. Drugim przykładem jest uruchamianie nowego procesu. Zamiast blokować dalsze operacje lub wielokrotnie sprawdzać (za pomocą odpytywania), czy proces zakończył pracę, najlepiej jest uruchamiać metodę asynchronicznie i używać funkcji **wywoływanej zwróciennie**, którą dana metoda uruchamia, aby powiadomić jednostkę wywołującą o zakończeniu asynchronicznej operacji.

Definiowanie nowego interfejsu za każdym razem, gdy trzeba przekazać metodę, wydaje się przesadą. W tym rozdziale opisano, jak tworzyć i stosować klasę specjalnego rodzaju — **delegat**. Umożliwia on traktowanie referencji do metod w taki sam sposób jak traktowane są dowolne inne dane. Następnie zobaczysz, jak za pomocą *wyrażeń lambda* szybko i łatwo tworzyć niestandardowe delegaty.

Wyrażenia lambda dodano w wersji C# 3.0. W C# 2.0 dostępna była mniej atrakcyjna składnia, pozwalająca tworzyć niestandardowe delegaty w postaci *metod anonimowych*. We wszystkich wersjach nowszych niż C# 2.0 metody anonimowe także są dostępne (w celu zachowania zgodności ze starszym kodem), jednak w nowym kodzie zamiast takich metod należy stosować wyrażenia lambda. W tym rozdziale w blokach z zagadnieniami dla zaawansowanych znajdziesz wyjaśnienie, jak korzystać z metod anonimowych. Jest to przydatne w trakcie pracy nad dawnym kodem zgodnym z wersją C# 2.0. Jeśli interesuje Cię tylko praca nad nowszym kodem, możesz pominąć te informacje.

Ten rozdział kończy się omówieniem *drzew wyrażeń*. Takie drzewa umożliwiają wykorzystanie w czasie wykonywania programu przygotowanych przez kompilator analiz wyrażeń lambda.



Wprowadzenie do delegatów

3.0

Doświadczeni programiści używający języków C i C++ od dawna stosują „wskaźniki do funkcji” jako mechanizm pozwalający przekazać referencję do jednej metody jako argument innej metody. W C# podobny efekt można uzyskać za pomocą **delegatów**. Delegaty umożliwiają zapisanie referencji do metody i przekazywanie jej jak dowolnego innego obiektu. Tak zapisaną metodę można wywoływać w ten sam sposób jak inne metody. Przyjrzyj się teraz przykładowej sytuacji, w której ta technika może być przydatna.

Opis scenariusza

Choć sortowanie bąbelkowe nie jest najbardziej wydajnym rozwiązaniem, stanowi jeden z najprostszych algorytmów sortowania. Na listingu 13.1 pokazano metodę BubbleSort().

Listing 13.1. Metoda BubbleSort()

```
static class SimpleSort1
{
    public static void BubbleSort(int[] items)
    {
        int i;
        int j;
        int temp;

        if (items==null)
        {
            return;
        }

        for (i = items.Length - 1; i >= 0; i--)
        {
            for (j = 1; j <= i; j++)
            {
                if (items[j - 1] > items[j])
                {
                    temp = items[j - 1];
                    items[j - 1] = items[j];
                    items[j] = temp;
                }
            }
        }
    }
}
```

```
        }  
    }  
    // ...  
}
```

Ta metoda sortuje tablicę liczb całkowitych w porządku rosnącym.

Założmy, że za pomocą kodu z listingu 13.1 chcesz mówić sortować liczby całkowite rosnąco lub malejąco. Możesz skopiować ten kod, a następnie zastąpić w nim operator „większy niż” operatorem „mniejszy niż”, ale powielanie kilkudziesięciu wierszy kodu tylko po to, by zmienić jeden operator, to zły pomysł. Bardziej zwięzła technika polega na przekazywaniu dodatkowego parametru, który określa, w jakiej kolejności program ma posortować dane. To rozwiązanie pokazano na listingu 13.2.

Listing 13.2. Metoda BubbleSort() obsługująca sortowanie w porządku rosnącym i malejącym

```
class SimpleSort2  
{  
    public enum SortType  
    {  
        Ascending,  
        Descending  
    }  
  
    public static void BubbleSort(int[] items, SortType sortOrder)  
    {  
        int i;  
        int j;  
        int temp;  
  
        if (items==null)  
        {  
            return;  
        }  
  
        for (i = items.Length - 1; i >= 0; i--)  
        {  
            for (j = 1; j <= i; j++)  
            {  
                bool swap = false;  
                switch (sortOrder)  
                {  
                    case SortType.Ascending :  
                        swap = items[j - 1] > items[j];  
                        break;  
  
                    case SortType.Descending :  
                        swap = items[j - 1] < items[j];  
                        break;  
                }  
                if (swap)  
                {  
                    temp = items[j - 1];  
                    items[j - 1] = items[j];  
                    items[j] = temp;  
                }  
            }  
        }  
    }  
}
```

3.0

```

        }
    }
}
// ...
}

```

Jednak ten kod obsługuje tylko dwa możliwe porządkie sortowania. Jeśli zechcesz posortować dane leksykograficznie (1, 10, 11, 12, 2, 20, ...) lub według innego kryterium, liczba wartości w wyliczeniu SortType i powiązanych przypadków w instrukcji switch szybko stanie się nie-wygodna w zarządzaniu.

Typ danych w postaci delegata

3.0

Aby zwiększyć swobodę programisty i zmniejszyć ilość powtarzającego się kodu z wcześniejszych listingów, metodę przeprowadzającą porównania można przekazywać jako parametr do metody BubbleSort(). W celu przekazywania metody jako argumentu potrzebny jest typ danych reprezentujący daną metodę. Ten typ danych jest nazywany *delegatem*, ponieważ deleguje wywołanie do metody wskazywanej przez obiekt tego typu. Jako instancję delegata można podawać nazwę metody. Od wersji C# 3.0 jako delegatów można też używać wyrażeń lambda, aby zapisywać fragment kodu „na miejscu”, zamiast tworzyć zawierającą go metodę. W C# 7.0 można ponadto utworzyć funkcję lokalną, a następnie posługiwać się nazwą tej funkcji jako delegatem. Listing 13.3 zawiera zmodyfikowaną wersję metody BubbleSort(), przyjmującą parametr w postaci wyrażenia lambda. Tu typem danych delegata jest Func<int, int, bool>.

Listing 13.3. Metoda BubbleSort() z parametrem w postaci delegata

```

class DelegateSample
{
// ...

public static void BubbleSort(
    int[] items, Func<int, int, bool> compare)
{
    int i;
    int j;
    int temp;

    if (compare == null)
    {
        throw new ArgumentNullException(nameof(compare));
    }

    if (items==null)
    {
        return;
    }

    for (i = items.Length - 1; i >= 0; i--)

```

```
{  
    for (j = 1; j <= i; j++)  
    {  
        if (compare(items[j - 1], items[j]))  
        {  
            temp = items[j - 1];  
            items[j - 1] = items[j];  
            items[j] = temp;  
        }  
    }  
}  
// ...  
}
```

Typ delegata, `Func<int, int, bool>`, reprezentuje metodę porównującą dwie liczby całkowite. W metodzie `BubbleSort()` można zastosować instancję typu `Func<int, int, bool>` (wskazywaną za pomocą parametru `compare`), by ustalić, która liczba całkowita jest większa. Ponieważ parametr `compare` reprezentuje metodę, składnia jej wywoływania jest identyczna jak dla innych metod. Tu delegat typu `Func<int, int, bool>` przyjmuje dwa parametry całkowitoliczbowe i zwraca wartość logiczną, określającą, czy pierwsza liczba całkowita jest większa od drugiej.

```
if (compare(items[j - 1], items[j])) { ... }
```

Zauważ, że dla delegata typu `Func<int, int, bool>` stosowana jest ścisła kontrola typu. Ten typ ma reprezentować metodę zwracającą wartość logiczną i przyjmującą dokładnie dwa parametry całkowitoliczbowe. Wywołanie kierowane do delegata (podobnie jak inne wywołania metod) jest zgłasiane ze ścisłą kontrolą typu. Jeśli typy danych argumentów nie są zgodne z typami parametrów, kompilator języka C# zgłosi błąd.

Deklarowanie typu delegata

Zobaczyłeś już, jak zdefiniować metodę korzystającą z delegata. Wiesz także, jak wywołać delegat — wystarczy potraktować reprezentującą go zmienną jak metodę. Musisz się jednak jeszcze nauczyć, jak zadeklarować typ delegata. W takiej deklaracji należy wpisać słowo kluczowe `delegate`, a następnie podać deklarację metody. Sygnatura tej metody określa, z jakimi metodami można powiązać delegat. Nazwa typu delegata występuje tam, gdzie w deklaracji metody pojawia się jej nazwa. Na przykład delegat `Func<...>` z listingu 13.3 jest zadeklarowany w następujący sposób:

```
public delegate TResult Func<in T1, in T2, out TResult>(  
    in T1 arg1, in T2 arg2)
```

Modyfikatory `in` i `out` dodano w wersji C# 4.0. Ich omówienie znajdziesz w dalszej części rozdziału.

Typy delegatów do ogólnego użytku — System.Func i System.Action

Na szczęście od wersji C# 3.0 rzadko (jeśli w ogóle) trzeba deklarować własne delegaty. Dzięki bibliotece uruchomieniowej z platformy .NET 3.5 (powiązanej z wersją C# 3.0) konieczność definiowania własnych niestandardowych typów delegatów została w praktyce wyeliminowana. W bibliotece tej znajduje się zestaw delegatów do ogólnego użytku, w większości są one generyczne. Rodzina delegatów System.Func służy do reprezentowania metod zwracających wartość. Delegaty z rodziny System.Action reprezentują metody zwracające void. Sygnatury takich delegatów przedstawiono na listingu 13.4.

Listing 13.4. Deklaracje delegatów typów Func i Action

3.0

```

public delegate void Action();
public delegate void Action<in T>(T arg)
public delegate void Action<in T1, in T2>(
    in T1 arg1, in T2 arg2)
public delegate void Action<in T1, in T2, in T3>(
    T1 arg1, T2 arg2, T3 arg3)
public delegate void Action<in T1, in T2, in T3, in T4>(
    T1 arg1, T2 arg2, T3 arg3, T4 arg4)
...
public delegate void Action<
    in T1, in T2, in T3, in T4, in T5, in T6, in T7, in T8,
    in T9, in T10, in T11, in T12, in T13, in T14, in T16(
        T1 arg1, T2 arg2, T3 arg3, T4 arg4,
        T5 arg5, T6 arg6, T7 arg7, T8 arg8,
        T9 arg9, T10 arg10, T11 arg11, T12 arg12,
        T13 arg13, T14 arg14, T15 arg15, T16 arg16)

public delegate TResult Func<out TResult>();
public delegate TResult Func<in T, out TResult>(T arg)
public delegate TResult Func<in T1, in T2, out TResult>(
    in T1 arg1, in T2 arg2)
public delegate TResult Func<in T1, in T2, in T3, out TResult>(
    T1 arg1, T2 arg2, T3 arg3)
public delegate TResult Func<in T1, in T2, in T3, in T4,
    out TResult>(T1 arg1, T2 arg2, T3 arg3, T4 arg4)
...
public delegate TResult Func<
    in T1, in T2, in T3, in T4, in T5, in T6, in T7, in T8,
    in T9, in T10, in T11, in T12, in T13, in T14, in T16,
    out TResult>(
        T1 arg1, T2 arg2, T3 arg3, T4 arg4,
        T5 arg5, T6 arg6, T7 arg7, T8 arg8,
        T9 arg9, T10 arg10, T11 arg11, T12 arg12,
        T13 arg13, T14 arg14, T15 arg15, T16 arg16)

public delegate bool Predicate<in T>( T obj)

```

Ponieważ delegaty z listingu 13.4 są generyczne, można stosować właśnie je zamiast definiować własne niestandardowe delegaty.

Pierwszy typ delegata z listingu 13.4 to Action<...>. Reprezentuje on metody, które nie zwracają wartości i mogą przyjmować do 16 parametrów. Do tworzenia delegatów zwracających wartości służą typy Func<...>. Ostatnim parametrem określającym typ w delegatach

Func<...> jest TResult, który określa typ zwracanej wartości. Pozostałe parametry delegatów Func<...> reprezentują sekwencję typów parametrów delegata. Metoda BubbleSort z listingu 13.3 wymaga delegata, który zwraca wartość typu bool i przyjmuje dwa parametry typu int.

Ostatnim z delegatów wymienionych na listingu 13.4 jest Predicate<int T>. Gdy wyrażenie lambda zwraca wartość logiczną, jest nazywane **predykatem**. Taki predykat zwykle służy do filtrowania lub identyfikowania elementów kolekcji. Należy przekazać element do predykatu, a predykat zwróci wartość true lub false informującą, czy odfiltrować dany element. W przykładowej metodzie BubbleSort() przyjmowane są dwa parametry w celu ich porównania, dlatego zamiast predykatu używany jest delegat Func<int, int, bool>.

Wskazówka

ROZWAŻ, czy poprawa czytelności zapewniana dzięki zdefiniowaniu własnego delegata przeważa nad wygodą wynikającą ze stosowania predefiniowanych generycznych typów delegatów.

3.0

ZAGADNIENIE DLA ZAAWANSOWANYCH

Deklarowanie typu delegata

Jak już wspomniałem, w wielu scenariuszach wprowadzone w platformie Microsoft .NET Framework 3.5 (i później w specyfikacji .NET Standard) delegaty Func i Action w praktyce eliminują konieczność definiowania własnych typów delegatów. Warto jedna rozważyć utworzenie takiego typu, jeśli pozwala to istotnie poprawić czytelność kodu. Na przykład delegat o nazwie Comparer jednoznacznie informuje, do czego służy. Z kolei nazwa Func<int, int, bool> określa tylko parametry i typ zwracany przez delegat. Na listingu 13.5 pokazano, jak zadeklarować typ delegata Comparer, który wymaga dwóch liczb całkowitych i zwraca wartość logiczną.

Listing 13.5. Deklarowanie typu delegata

```
public delegate bool Comparer (
    int first, int second);
```

Za pomocą nowego typu delegata możesz zmodyfikować listing 13.3, używając w sygnaturze typu Comparer zamiast Func<int, int, bool>:

```
public static void BubbleSort(int[] items, Comparer compare)
```

Podobnie jak w klasach można zagnieździć inne klasy, możliwe jest też zagnieźdzenie w nich delegata. Jeśli deklaracja delegata występuje w innej klasie, typ delegata jest typem zagnieżdzonym. Ilustruje to listing 13.6.

Listing 13.6. Deklarowanie zagnieźdzanego typu delegata

```
class DelegateSample
{
    public delegate bool ComparisonHandler (
        int first, int second);
}
```

W tym przykładzie typ delegata to `DelegateSample.ComparisonHandler`, ponieważ jest on typem zagnieżdżonym w klasie `DelegateSample`. Zagnieżdżanie warto rozważyć, gdy spo- dziewasz się, że delegat będzie używany tylko w zawierającej go klasie.

Tworzenie instancji delegata

W ostatnim kroku tworzenia metody `BubbleSort()` korzystającej z delegata dowiesz się, jak wywołać potrzebną metodę i przekazać instancję delegata (tu jest nią obiekt typu `Func<int, int, bool>`). Aby utworzyć instancję delegata, potrzebujesz metody z parametrami i zwracaną wartością, odpowiadającymi sygnaturze typu delegata. Nazwa metody nie musi pasować do nazwy delegata, natomiast pozostała część sygnatury musi być w obu miejscach taka sama. Na listingu 13.7 pokazano kod metody `GreaterThan` zgodnej z typem delegata.

Listing 13.7. Deklarowanie metody zgodnej z typem `Func<int, int, bool>`

3.0

```
class DelegateSample
{
    public static void BubbleSort(
        int[] items, Func<int, int, bool> compare)
    {
        // ...
    }

    public static bool GreaterThan(int first, int second)
    {
        return first > second;
    }
    // ...
}
```

Po zdefiniowaniu tej metody można wywołać metodę `BubbleSort()` i jako argument przekazać do niej nazwę metody wiązanej z delegatem. Przedstawiono to na listingu 13.8.

Listing 13.8. Używanie nazwy metody jako argumentu

```
class DelegateSample
{
    public static void BubbleSort(
        int[] items, Func<int, int, bool> compare)
    {
        // ...
    }

    public static bool GreaterThan(int first, int second)
    {
        return first > second;
    }
    static void Main()
    {
        int i;
        int[] items = new int[5];
```

```

for (i=0; i < items.Length; i++)
{
    Console.WriteLine("Wprowadź liczbę całkowitą: ");
    items[i] = int.Parse(Console.ReadLine());
}

BubbleSort(items, GreaterThan);

for (i = 0; i < items.Length; i++)
{
    Console.WriteLine(items[i]);
}
}

```

Zauważ, że typy delegatów to typy referencyjne, jednak nie trzeba używać słowa kluczowego new, by tworzyć ich instancje. Od wersji C# 2.0 konwersja z **grupy metod** (wyrażenia określającego nazwę metody) na typ delegata powoduje automatyczne utworzenie nowego obiektu typu delegata.

3.0

ZAGADNIENIE DLA ZAAWANSOWANYCH

Tworzenie instancji delegatów w wersji C# 1.0

Na listingu 13.8 instancję delegata utworzono w wyniku przekazania nazwy potrzebnej metody (GreaterThan) jako argumentu w wywołaniu metody BubbleSort(). W pierwszej wersji języka C# wymagane było utworzenie instancji delegata, przez co potrzebna była bardziej rozwlekła składnia przedstawiona na listingu 13.9.

Listing 13.9. Przekazywanie delegata jako parametru w wersji C# 1.0

```
BubbleSort(items,
    new Comparer(GreaterThan));
```

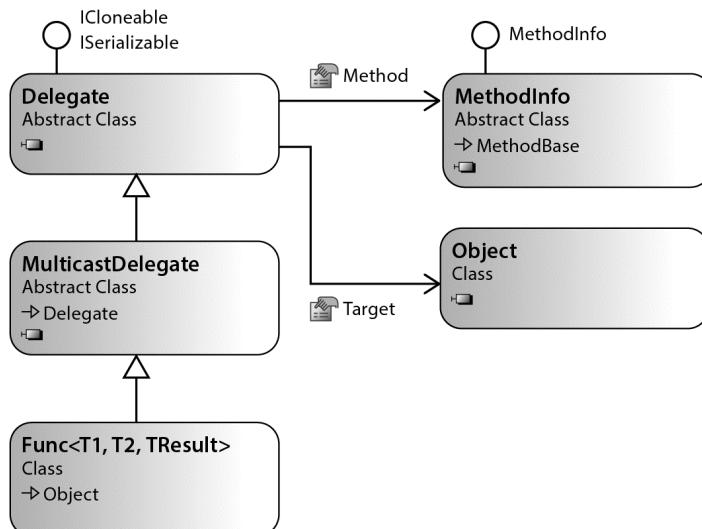
Tu zastosowano delegat Comparer zamiast Func<int, int, bool>, ponieważ ten ostatni był niedostępny w C# 1.0.

W nowszych wersjach języka obsługiwane są obie składnie, jednak w dalszej części książki stosowana jest tylko nowa, zwięzła składnia.

ZAGADNIENIE DLA ZAAWANSOWANYCH

Wewnętrzne mechanizmy delegatów

Delegat to specjalny rodzaj klasy. Choć w standardzie języka C# hierarchia klas delegatów nie jest precyzyjnie określona, wszystkie delegaty muszą bezpośrednio lub pośrednio dziedziczyć po typie System.Delegate. W platformie .NET delegaty zawsze dziedziczą po typie System.MulticastDelegate, który z kolei dziedziczy po typie System.Delegate, co pokazano na rysunku 13.1.



3.0

Rysunek 13.1. Obiektowy model typów delegatów

Pierwsza właściwość jest typu `System.Reflection.MethodInfo`. Właściwość typu `MethodInfo` opisuje sygnaturę danej metody (obejmuje nazwę metody, parametry i typ zwracanej wartości). Oprócz tej właściwości delegat wymaga instancji obiektu zawierającego wywoływaną metodę. Do tego służy druga właściwość, `Target`. W przypadku metod statycznych właściwość `Target` odpowiada samemu typowi. Przeznaczenie klasy `MulticastDelegate` wyjaśniono w rozdziale 14.

Zauważ, że wszystkie delegaty są niemodyfikowalne. Po utworzeniu delegata nie można go zmodyfikować. Jeśli istnieje zmienna zawierająca referencję do delegata i programista chce, by zmienna prowadziła do innej metody, musi utworzyć nowy delegat i przypisać go do zmiennej.

Choć wszystkie typy delegatów pośrednio dziedziczą po klasie `System.Delegate`, kompilator języka C# nie zezwala na deklarowanie klas bezpośrednio lub pośrednio pochodnych od `System.Delegate` lub `System.MulticastDelegate`. Dlatego kod z listingu 13.10 nie jest prawidłowy.

Listing 13.10. `System.Delegate` nie może być bezpośrednią klasą bazową

```
// BŁĄD: Func<T1, T2, TResult> nie może dziedziczyć
// po klasie specjalnej 'System.Delegate'.
public class Func<T1, T2, TResult>: System.Delegate
{
    // ...
}
```

Przekazywanie delegata w celu określenia порядку sortowania daje znacznie większą swobodę niż podejście opisane na początku rozdziału. Dzięki przekazaniu delegata można zmienić kolejność sortowania na alfabetyczną w wyniku dodania nowego delegata, który

w ramach porównania przekształca liczby całkowite nałańcuchy znaków. Na listingu 13.11 przedstawiono kompletny kod ilustrujący sortowanie w porządku alfabetycznym. Efekt działania kodu znajdziesz w danych wyjściowych 13.1.

Listing 13.11. Użycie innej metody zgodnej z typem Func<int, int, bool>

```
using System;
class DelegateSample
{
    public static void BubbleSort(
        int[] items, Func<int, int, bool> compare)
    {
        int i;
        int j;
        int temp;

        for (i = items.Length - 1; i >= 0; i--)
        {
            for (j = 1; j <= i; j++)
            {
                if (compare(items[j - 1], items[j]))
                {
                    temp = items[j - 1];
                    items[j - 1] = items[j];
                    items[j] = temp;
                }
            }
        }
    }

    public static bool GreaterThan(int first, int second)
    {
        return first > second;
    }

    public static bool AlphabeticalGreater Than(
        int first, int second)
    {
        int comparison;
        comparison = (first.ToString()).CompareTo(
            second.ToString()));

        return comparison > 0;
    }

    static void Main(string[] args)
    {
        int i;
        int[] items = new int[5];

        for (i=0; i<items.Length; i++)
        {
            Console.Write("Podaj liczbę całkowitą: ");
            items[i] = int.Parse(Console.ReadLine());
        }
    }
}
```

3.0

```
BubbleSort(items, AlphabeticalGreaterThan);  
  
    for (i = 0; i < items.Length; i++)  
    {  
        Console.WriteLine(items[i]);  
    }  
}
```

DANE WYJŚCIOWE 13.1.

```
Podaj liczbę całkowitą: 1  
Podaj liczbę całkowitą: 12  
Podaj liczbę całkowitą: 13  
Podaj liczbę całkowitą: 5  
Podaj liczbę całkowitą: 4  
1  
12  
13  
4  
5
```

3.0

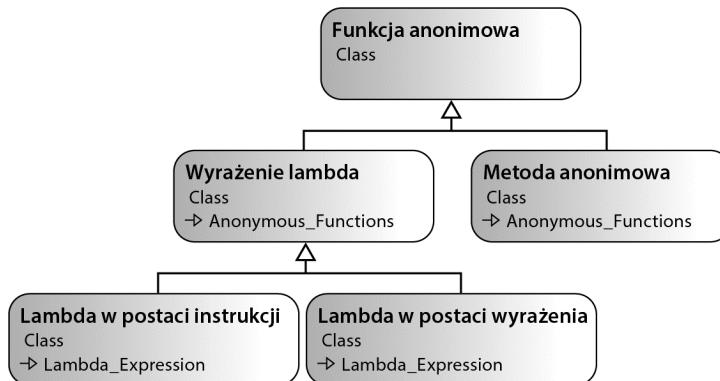
Porządek alfabetyczny jest inny od liczbowego. Zauważ jednak, jak łatwo udało się dodać nowy mechanizm sortowania w porównaniu z podejściem wykorzystanym na początku rozdziału. Jedyne zmiany potrzebne do rozpoczęcia sortowania w porządku alfabetycznym to dodanie metody `AlphabeticalGreaterThan` i przekazanie jej w wywołaniu metody `BubbleSort()`.

Wyrażenia lambda

Na listingach 13.7 i 13.10 zobaczyłeś, że można przekształcić wyrażenia `Greater Than` i `AlphabeticalGreater Than` na typ delegata zgodny z typami parametrów i typem zwracanej wartości danej metody. Może zauważysz, że deklaracja metody `Greater Than` (kod informujący, że jest to metoda publiczna i statyczna, która zwraca wartość typu `bool` oraz przyjmuje dwa parametry — `first` i `second` — typu `int`) jest wyraźnie dłuższa niż ciało tej metody. W cielesie wystarczy porównać oba parametry i zwrócić wynik. To niewygodne, że ta prosta metoda wymaga tak wiele „ceregieli”, by można ją było przekształcić na typ delegata.

Aby rozwiązać ten problem, w wersji C# 2.0 wprowadzono dużo bardziej zwięzłą składnię tworzenia delegatów. W C# 3.0 pojawiły się jeszcze bardziej zwięzłe techniki. Mechanizm z wersji C# 2.0 to **metody anonimowe**, natomiast w C# 3.0 wprowadzono **wyrażenia lambda**. Gdy opis dotyczy obu tych mechanizmów, nazywa się je **funkcjami anonimowymi**. Obecnie dopuszczalne są oba te rozwiązania, natomiast w nowym kodzie zalecane jest używanie wyrażeń lambda zamiast metod anonimowych. W tej książce w większości miejsc stosowane są wyrażenia lambda, chyba że dany fragment poświęcony jest metodom anonimowym z wersji C# 2.0.

Są dwa rodzaje wyrażeń lambda — **lambdy w postaci instrukcji** i **lambdy w postaci wyrażeń**. Na rysunku 13.2 przedstawiono hierarchiczne relacje między tymi mechanizmami.



Rysunek 13.2. Terminologia dotycząca funkcji anonimowych

Lambdy w postaci instrukcji

3.0

Wyrażenia lambda mają upraszczać deklarowanie zupełnie nowych składowych, gdy programista chce utworzyć delegat na podstawie bardzo prostej metody. Istnieje kilka rodzajów wyrażeń lambda. Lambda w postaci instrukcji obejmuje listę parametrów formalnych, po której następuje operator lambdy, =>, i blok kodu.

Na listingu 13.12 przedstawiono kod działający podobnie jak wywołanie metody BubbleSort na listingu 13.8. Różnica polega na tym, że na listingu 13.12 zamiast metody GreaterThan zastosowano lambdę w postaci instrukcji, aby podać metodę odpowiedzialną za porównania. Jak widać, w lambdzie w postaci instrukcji występują liczne informacje z deklaracji metody GreaterThan. Deklaracje parametrów formalnych i blok kodu są takie same, pominięto natomiast nazwę metody i modyfikator.

Listing 13.12. Tworzenie delegata za pomocą lambdy w postaci instrukcji

```
// ...
BubbleSort(items,
    (int first, int second) =>
{
    return first < second;
});
// ...
```

W trakcie czytania kodu obejmującego operator lambda można zastąpić ten operator słowami *jest kierowane do*. Na przykład na listingu 13.12 drugi parametr metody BubbleSort() można przeczytać tak: „liczby całkowite first i second są kierowane do instrukcji zwracającej informację o tym, czy first jest mniejsza niż second”.

Zauważ, że składnia z listingu 13.12 jest niemal identyczna z kodem z listingu 13.8. Jednak w nowej wersji metoda odpowiedzialna za porównania znajduje się leksykalnie w miejscu, gdzie jest przekształcana na typ delegata. Metoda nie jest teraz zapisana gdzie indziej i nie trzeba jej wyszukiwać na podstawie nazwy. Nazwa metody w ogóle nie istnieje, dlatego takie

metody określa się mianem *funkcji anonimowych*. Nie podano też typu zwracanej wartości, jednak kompilator wykrywa, że wyrażenie lambda jest przekształcane w delegat, który zgodnie z sygnaturą wymaga zwracania wartości typu bool. Dlatego kompilator sprawdza, czy wyrażenia w każdej instrukcji return z bloku kodu lambdy są zgodne z metodą zwracającą wartość typu bool. W nowym kodzie nie ma modyfikatora public. Ponieważ metoda nie jest już składową dostępną w zawierającej ją klasie, nie trzeba określać poziomu dostępu. Także modyfikator static nie jest już potrzebny. W ten sposób ilość „ceregieli” związanych z metodą jest już znacznie mniejsza.

Jednak składnia nadal jest niepotrzebnie rozwlekła. Z typu delegata wynika, że wyrażenie lambda musi zwracać wartość typu bool. Podobnie można wywnioskować, że oba parametry muszą być typu int. Ilustruje to listing 13.13.

Listing 13.13. Pomijanie typów parametrów w lambdach w postaci instrukcji

```
3.0 // ...
    BubbleSort(items,
        (first, second) =>
    {
        return first < second;
    });
// ...
```

Zwykle w wyrażeniach lambda jawne deklaracje typów parametrów są opcjonalne, jeśli kompilator może wywnioskować te typy na podstawie delegata, w który przekształcone jest dane wyrażenie. Jednak w sytuacjach, w których dzięki podaniu typu kod staje się bardziej czytelny, język C# pozwala to zrobić. Gdy inferencja typów nie jest możliwa, język C# wymaga jawnego określenia typów parametrów w lambdzie. Jeśli w lambdzie jawnie podany jest typ jednego parametru, trzeba to zrobić także dla wszystkich pozostałych parametrów. Wszystkie typy muszą wtedy dokładnie pasować do typów parametrów w delegacie.

Wskazówka

ROZWAŻ pominięcie typów na liście parametrów formalnych lambdy, gdy te typy są oczywiste dla czytelnika lub stanowią nieistotny szczegół.

Możliwe są też inne sposoby skrócenia składni, co ilustruje listing 13.14. Jeśli lambda ma dokładnie jeden parametr, a jego typ można wywnioskować, dozwolone jest pominięcie nawiasów wokół listy parametrów. Gdy parametry w ogóle nie występują lub gdy ich liczba jest większa niż jeden (a także w sytuacji, gdy używany jest jeden parametr, ale z jawnie podanym typem), lista parametrów musi być ujęta w nawias.

Listing 13.14. Lambda w postaci instrukcji mająca jeden parametr wejściowy

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

```
// ...
IEnumerable<Process> processes = Process.GetProcesses().Where(
    process => { return process.WorkingSet64 > 1000000000; });
// ...
```

Na listingu 13.14 metoda `Where()` zwraca wynik zapytania o procesy, które zajmują więcej niż miliard bajtów pamięci fizycznej. Porównaj ten kod z wersją z listingu 13.15, gdzie używana jest lambda w postaci instrukcji bez parametrów. Pusta lista parametrów wymaga nawiasu. Zauważ też, że na listingu 13.15 ciało lambdy zawiera kilka wyrażeń w bloku instrukcji (ograniczonym nawiasem klamrowym). Choć lambdy w postaci instrukcji mogą zawierać dowolną liczbę wyrażeń, zwykle ich liczba jest ograniczona do dwóch lub trzech.

Listing 13.15. Bezparametrowe lambdy w postaci instrukcji

```
// ...
Func<string> getUserInput =
() =>
{
    string input;
    do
    {
        input = Console.ReadLine();
    }
    while (input.Trim().Length == 0);
    return input;
};
```

3.0

Lambdy w postaci wyrażeń

Składnia lambd w postaci instrukcji jest dużo bardziej zwięzła niż deklaracje analogicznych metod. Nie trzeba deklarować nazwy metody, poziomu dostępności, typu zwracanej wartości ani typów parametrów. Jednak lambdy w postaci wyrażeń pozwalają jeszcze bardziej skrócić kod. Na listingach 13.12, 13.13 i 13.14 używane były lambdy w postaci instrukcji, gdzie w bloku znajdowała się jedna instrukcja `return`. Co się stanie, jeśli wyeliminuje się związane z tym „ceregiele”? Jedyną istotną informacją w takim bloku jest zwracane wyrażenie. Lambda w postaci wyrażenia obejmuje tylko zwracane wyrażenie — w ogóle nie występuje w niej blok instrukcji. Rozwiązań z listingu 13.16 działa tak samo jak kod z listingu 13.12, jednak wykorzystano tu lambdę w postaci wyrażenia zamiast lambdy w postaci instrukcji.

Listing 13.16. Przekazywanie delegata za pomocą lambdy w postaci wyrażenia

```
// ...
BubbleSort(items, (first, second) => first < second );
// ...
```

Operator lambdy, `=>`, w wyrażeniach czytany jest tak samo jak w lambdach w postaci instrukcji — jako *jest kierowany do lub staje się*. Gdy lambda jest predykatem, operator lambda często jest czytany jako *taki, że lub gdzie*. Tak więc lambdę z listingu 13.16 można przeczytać jako „*first i second takie, że first jest mniejsze niż second*”.

Funkcja anonimowa, podobnie jak literal `null`, nie ma określonego typu. Ten typ jest wyznaczany przez typ, na który następuje konwersja funkcji. Oznacza to, że wyrażenia lambda przedstawione do tej pory nie są same w sobie typu `Func<int, int, bool>` (lub `Comparer`), natomiast są zgodne z tym typem i mogą zostać na niego przekształcone. Dlatego do metody anonimowej nie można użyć operatora `typeof()`, a wywołanie metody `GetType()` dla metody anonimowej jest możliwe tylko po przekształceniu tej ostatniej na konkretny typ.

W tabeli 13.1 przedstawiono inne cechy wyrażeń lambda.

Tabela 13.1. Uwagi i przykłady dotyczące wyrażeń lambda

3.0

Instrukcja	Przykład
Same wyrażenia lambda nie mają typu. Dlatego nie istnieją składowe (nawet metody z typu <code>Object</code>), do których można uzyskać bezpośrednio dostęp za pomocą wyrażenia lambda.	// BŁĄD: Operatora '!' nie można zastosować // do operandu typu 'wyrażenie lambda'. string s = ((int x) => x).ToString();
Wyrażenia lambda nie mają typu, dlatego nie mogą występować po lewej stronie operatora <code>is</code> .	// BŁĄD: pierwszym operandem operatora // 'is' lub 'as' nie może być wyrażenie lambda // ani metoda anonimowa. bool b = ((int x) => x) is Func<int, int>;
Wyrażenie lambda można przekształcić tylko na zgodny typ delegata. Tu lambdy zwracającej wartość typu <code>int</code> nie można przekształcić na typ delegata reprezentujący metodę zwracającą wartość typu <code>bool</code> .	// BŁĄD: to wyrażenie lambda nie jest zgodne // z typem <code>Func<int, bool></code> . Func<int, bool> f = (int x) => x;
Wyrażenie lambda nie ma typu, dlatego nie może być użyte do inferencji typu zmiennej lokalnej.	// BŁĄD: nie można przypisać wyrażenia lambda do // zmiennej lokalnej o niejawnie określonym typie. var v = x => x;
Instrukcje skoku (<code>break</code> , <code>goto</code> , <code>continue</code>) w wyrażeniach lambda nie mogą być używane do przeskakiwania do lokalizacji poza lambdą (i spoza lambdy do niej). Tu instrukcja <code>break</code> w lambdzie powodowałaby skok na koniec instrukcji <code>switch</code> poza lambdą.	// BŁĄD: sterowanie nie może wyjść poza metodę // anonimową lub wyrażenie lambda. string[] args; Func<string> f; switch (args[0]) { case "/File": f = () => { if (!File.Exists(args[1])) break; return args[1]; }; // ... }

Tabela 13.1. Uwagi i przykłady dotyczące wyrażeń lambda — ciąg dalszy

Instrukcja	Przykład
Parametry i zmienne lokalne z wyrażenia lambda znajdują się w zasięgu wyłącznie w ciele lambdy.	// BŁĄD: nazwa 'first' nie istnieje // w kontekście jej użycia. Func<int, int, bool> expression = (first, second) => first > second; first++;
Kompilator w trakcie analizy przypisań nie potrafi wykryć inicjowania „zewnętrznych” zmiennych lokalnych w wyrażeniach lambda.	int number; Func<string, bool> f = text => int.TryParse(text, out number); if (f("1")) { // BŁĄD: użycie zmiennej lokalnej // o nieprzypisanej wartości. System.Console.WriteLine(number); } int number; Func<int, bool> isFortyTwo = x => 42 == (number = x); if (isFortyTwo(42)) { // BŁĄD: użycie zmiennej lokalnej // o nieprzypisanej wartości. System.Console.WriteLine(number); }

3.0

Początek
2.0

Metody anonimowe

W wersji C# 2.0 wyrażenia lambda nie są obsługiwane. Zamiast nich dostępna jest składnia do tworzenia *metod anonimowych*. Takie metody przypominają lambdy w postaci instrukcji, ale są pozbawione niektórych rozwiązań, dzięki którym lambdy są tak zwięzłe. W metodzie anonimowej trzeba jawnie określić typ każdego parametru, a także utworzyć blok instrukcji. Ponadto zamiast między listą parametrów a blokiem kodu wpisać operator lambda (`=>`), w metodzie anonimowej należy umieścić słowo kluczowe `delegate` przed listą parametrów. Jest to podkreślenie faktu, że metodę anonimową trzeba przekształcić na typ delegata. Na listingu 13.17 pokazano kod z listingów 13.7, 13.12 i 13.15, zmodyfikowany w wyniku zastosowania metody anonimowej.

Listing 13.17. Przekazywanie metody anonimowej w wersji C# 2.0

```
// ...
BubbleSort(items,
    delegate(int first, int second)
    {
        return first < second;
    }
);
// ...
```

Kłopotliwe jest to, że od wersji C# 3.0 dostępne są dwa bardzo podobne sposoby definiowania funkcji anonimowych.

Wskazówka

UNIKAJ w nowym kodzie stosowania składni metod anonimowych. Zamiast tego posługuj się bardziej zwięzłą składnią wyrażeń lambda.

Metody anonimowe zapewniają jednak pewne rozwiążanie niedostępne w wyrażeniach lambda — pozwalają w niektórych sytuacjach całkowicie pominąć listę parametrów.

■ ■ ■ ZAGADNIENIE DLA ZAAWANSOWANYCH

3.0

Bezparametrowe metody anonimowe

Metody anonimowe, w odróżnieniu od wyrażeń lambda, pozwalają całkowicie pominąć listę parametrów. Jest tak, jeśli w ciele metody anonimowej nie są używane żadne parametry, a typ delegata wymaga jedynie parametrów w postaci wartości (czyli nie wymaga oznaczania ich modyfikatorami out lub ref). Na przykład metoda anonimowa w wyrażeniu delegate { return Console.ReadLine() != ""; } może zostać przekształcona na dowolny typ delegata wymagający zwrócenia wartości typu bool. Nie jest przy tym istotne, ilu parametrów wymaga dany delegat. Ta technika jest stosowana rzadko, jednak możesz na nią natrafić w trakcie czytania starszego kodu.

Koniec
2.0

■ ■ ■ ZAGADNIENIE DLA ZAAWANSOWANYCH I POCZĄTKUJĄCYCH

Skąd się wzięła nazwa „lambda”?

Łatwo można zrozumieć, skąd wzięły swoją nazwę metody anonimowe. Wyglądają one bardzo podobnie jak deklaracje metod, ale nie mają określonej nazwy. Skąd jednak pochodzi człon *lambda* w nazwie „wyrażenia lambda”?

Pomysł na wyrażenia lambda wziął się z prac logika Alonzo Churcha, który w latach 30. ubiegłego wieku wymyślił technikę analizowania funkcji — *rachunek lambda*. W notacji zaproponowanej przez Churcha funkcja przyjmująca parametr *x* i dającą w efekcie wyrażenie *y* jest zapisywana z przedrostkiem w postaci małej greckiej litery lambda przed całym wyrażeniem. Ponadto parametr od wartości jest rozdzielany kropką. Tak więc wyrażenie lambda *x=>y* z języka C# przyjmuje postać *x.y* w notacji Churcha. Ponieważ używanie greckich liter w programach w języku C# jest niewygodne, a kropka ma już w nim wiele innych znaczeń, projektanci języka zdecydowali się zastosować notację z „grubą strzałką” zamiast pierwotnego zapisu. Nazwa „wyrażenie lambda” oznacza, że teoretyczne podstawy funkcji anonimowych są oparte na rachunku lambda, choć w tekście nie występuje litera lambda.

Delegaty nie zapewniają równości strukturalnej

Typy delegatów w platformie .NET nie zapewniają **równości strukturalnej**. Oznacza to, że nie można przekształcić referencji do obiektu jednego typu delegata na niepowiązany typ delegata, nawet jeśli parametry formalne i typ zwracanej wartości w obu delegatach są takie same. Na przykład nie można przypisać referencji prowadzącej do wartości typu `Comparer` do zmiennej typu `Func<int, int, bool>`, choć oba te typy delegata reprezentują metody przyjmujące dwa parametry typu `int` i zwracające wartość typu `bool`. Niestety, jedynym sposobem na użycie delegata innego typu, gdy delegaty są identyczne strukturalnie, ale niepowiązane, jest utworzenie nowego delegata korzystającego z metody `Invoke` pierwotnego delegata. Na przykład jeśli dostępna jest zmienna `c` typu `Comparer`, a programista chce przypisać jej wartość do zmiennej `f` typu `Func<int, int, bool>`, można napisać instrukcję `f = c.Invoke;`.

Dzięki dodanej w wersji C# 4.0 obsłudze wariancji można przeprowadzać konwersję między niektórymi typami delegatów. Przyjrzyj się przykładowi opartemu na kontrawariancji — ponieważ w delegacie `void Action<in T>(T arg)` w stosunku do parametru określającego typ używany jest modyfikator `in`, referencję do delegata typu `Action<object>` można przypisać do zmiennej typu `Action<string>`.

Dla wielu osób kontrawariancja w kontekście delegatów jest nieintuicyjna. Zapamiętaj, że operację, którą da się wykonać na każdym obiekcie, można też wykorzystać jako operację przeprowadzaną na łańcuchach znaków. Odwrotna zależność nie jest jednak prawdziwa. Operacja, którą można wykonać tylko na łańcuchach znaków, nie może zostać zastosowana do dowolnego obiektu. Podobnie każdy typ z rodziny delegatów `Func` jest kowariantny względem typu zwracanej wartości, na co wskazuje modyfikator `out` zastosowany do określającego typ parametru `TResult`. Dlatego referencję do delegata typu `Func<string>` można przypisać do zmiennej typu `Func<object>`.

Na listingu 13.18 przedstawiono przykłady zastosowania kowariancji i kontrawariancji do delegatów.

Listing 13.18. Wariancja zastosowana do delegatów

```
// Kontrawariancja.
Action<object> broadAction =
    (object data) =>
{
    Console.WriteLine(data);
};

Action<string> narrowAction = broadAction;

// Kowariancja.
Func<string> narrowFunction =
    () => Console.ReadLine();
Func<object> broadFunction = narrowFunction;

// Połączenie kontrawariancji i kowariancji.
Func<object, string?> func1 =
    (object data) => data.ToString();
Func<string, object?> func2 = func1;
```

W ostatnim fragmencie listingu połączono oba aspekty wariancji w jednym przykładzie. Ilustruje on, że kowariancja i kontrawariancja mogą występować jednocześnie, jeśli używane są określające typ parametry z modyfikatorami `in` i `out`.

Umożliwienie konwersji referencji do generycznych typów delegatów było jednym z głównych czynników motywujących do dodania obsługi konwersji kowariantnych i kontrawariantnych w wersji C# 4.0. Innym istotnym czynnikiem było dodanie obsługi kowariancji do typu `IEnumerable<out T>`.

Koniec
4.0

3.0

ZAGADNIENIE DLA ZAAWANSOWANYCH

Wewnętrzne mechanizmy wyrażeń lambda i metod anonimowych

Wyrażenia lambda (i metody anonimowe) nie są wbudowane w środowisko CLR. W rzeczywistości jest tak, że gdy kompilator natrafi na funkcję anonimową, przekształca ją na specjalne ukryte klasy, pola i metody z implementacją potrzebnych mechanizmów. Kompilator języka C# generuje kod z implementacją omawianego wzorca, dzięki czemu programiści nie muszą tego robić samodzielnie. Na podstawie kodu z listingów 13.12, 13.13, 13.16 i 13.17 kompilator języka C# generuje kod CIL odpowiadający kodowi podobnemu do tego przedstawionego na listingu 13.19.

Listing 13.19. Kod w języku C# odpowiadający kodowi CIL wygenerowanemu przez kompilator na podstawie wyrażeń lambda

```
class DelegateSample
{
    // ...
    static void Main(string[] args)
    {
        int i;
        int[] items = new int[5];

        for (i=0; i<items.Length; i++)
        {
            Console.Write("Podaj liczbę całkowitą:");
            items[i] = int.Parse(Console.ReadLine());
        }

        BubbleSort(items,
            DelegateSample._AnonymousMethod_00000000);

        for (i = 0; i < items.Length; i++)
        {
            Console.WriteLine(items[i]);
        }
    }

    private static bool _AnonymousMethod_00000000(
        int first, int second)
    {
        return first < second;
    }
}
```

W tym przykładzie kompilator przekształca funkcję anonimową w niezależnie zadeklarowaną metodę statyczną, tworzoną potem jako obiekt typu delegata przekazywany jako parametr. Nie jest zaskoczeniem, że kompilator generuje kod bardzo podobny do pierwotnej wersji z listingu 13.8. Składnię funkcji anonimowych dodano właśnie po to, by uprościć zapis takiego kodu. Jeśli jednak używane są zmienne zewnętrzne, transformacje wykonywane przez kompilator bywają dużo bardziej skomplikowane i nie ograniczają się do zapisania funkcji anonimowej jako metody statycznej.

Zmienne zewnętrzne

Zmienne lokalne (włącznie z parametrami zewnętrznej metody) zadeklarowane poza wyrażeniem lambda są nazywane **zmiennymi zewnętrznymi** lambdy. Referencja `this`, choć technicznie nie jest zmienną, też jest w tym kontekście traktowana jak zmienna zewnętrzna. Gdy w ciele lambdy używana jest zmienna zewnętrzna, można powiedzieć, że ta zmienna jest **przechwytywana** (ang. *captured* lub *closed over*) przez lambdę. Na listingu 13.20 zmienna zewnętrzna jest używana do zliczania porównań przeprowadzanych w metodzie `BubbleSort()`. Wynik działania kodu znajdziesz w danych wyjściowych 13.2.

Listing 13.20. Używanie zmiennej zewnętrznej w wyrażeniu lambda

```
class DelegateSample
{
    // ...
    static void Main(string[] args)
    {
        int i;
        int[] items = new int[5];
        int comparisonCount=0;

        for (i=0; i<items.Length; i++)
        {
            Console.Write("Podaj liczbę całkowitą:");
            items[i] = int.Parse(Console.ReadLine());
        }

        BubbleSort(items,
            (int first, int second) =>
            {
                comparisonCount++;
                return first < second;
            });
    }

    for (i = 0; i < items.Length; i++)
    {
        Console.WriteLine(items[i]);
    }

    Console.WriteLine("Liczba porównań to: {0}.",
        comparisonCount);
}
```

DANE WYJŚCIOWE 13.2.

```
Podaj liczbę całkowitą:5
Podaj liczbę całkowitą:1
Podaj liczbę całkowitą:4
Podaj liczbę całkowitą:2
Podaj liczbę całkowitą:3
5
4
3
2
1
```

Liczba porównań to: 10.

Zauważ, że zmienna `comparisonCount` pojawia się poza wyrażeniem lambda, ale jej wartość jest zwiększana w tym wyrażeniu. Po wykonaniu metody `BubbleSort()` wartość zmiennej `comparisonCount` jest wyświetlana w konsoli.

3.0

Zwykle czas życia zmiennej lokalnej jest ograniczony do jej zasięgu. Gdy sterowanie wychodzi poza ten zasięg, lokalizacja w pamięci powiązana z daną zmienną nie musi zawierać poprawnych danych. Jednak delegat utworzony na podstawie lambdy, która przechwytuje zmienną zewnętrzną, może mieć dłuższy (lub krótszy) czas życia niż dana zmienna lokalna. Taki delegat potrzebuje bezpiecznego dostępu do zmiennej zewnętrznej za każdym razem, gdy zostanie wywołany. Dlatego czas życia przechwyconej zmiennej jest wydłużany. Pewne jest, że będzie ona dostępna przynajmniej tak długo, jak długo istnieje choć jeden obiekt delegata, który ją przechwycił. Możliwe, że będzie dostępna dłużej. To, w jaki sposób kompilator generuje kod zapewniający wydłużenie czasu życia zmiennej zewnętrznej, jest zależne od implementacji i może się zmieniać.

Kompilator języka C# generuje kod CIL, który udostępnia zmienną `comparisonCount` w metodzie anonimowej oraz w metodzie, w której zmienną zadeklarowano.

ZAGADNIENIE DLA ZAAWANSOWANYCH**Implementacja zmiennych zewnętrznych w kodzie CIL**

Kod CIL wygenerowany przez kompilator języka C# dla funkcji anonimowych przechwytyjących zmienne zewnętrzne jest bardziej skomplikowany niż kod odpowiadający prostym funkcjom anonimowym, które nie przechwytują żadnych zmiennych. Na listingu 13.21 przedstawiono kod w języku C# odpowiadający kodowi CIL wygenerowanemu na podstawie przechwytyjącego zmienne zewnętrzne kodu z listingu 13.20.

Listing 13.21. Kod w języku C# odpowiadający wygenerowanemu przez kompilator kodowi CIL przetwarzającemu zmienne zewnętrzne

```
class DelegateSample
{
    // ...
    private sealed class __LocalsDisplayClass_00000001
    {
        public int comparisonCount;
        public bool __AnonymousMethod_00000000(
            int first, int second)
```

```

    {
        comparisonCount++;
        return first < second;
    }
}
// ...
static void Main(string[] args)
{
    int i;
    _LocalsDisplayClass_00000001 locals =
        new _LocalsDisplayClass_00000001();
    locals.comparisonCount=0;
    int[] items = new int[5];

    for (i=0; i<items.Length; i++)
    {
        Console.Write("Podaj liczbę całkowitą:");
        items[i] = int.Parse(Console.ReadLine());
    }

    BubbleSort(items, locals._AnonymousMethod_00000000);
    for (i = 0; i < items.Length; i++)
    {
        Console.WriteLine(items[i]);
    }

    Console.WriteLine("Liczba porównań to: {0}.",
        locals.comparisonCount);
}
}

```

3.0

Zauważ, że przechwytywana zmienna lokalna nigdy nie jest „przekazywana” ani „kopiowana”. Zamiast tego przechwytywana zmienna lokalna (`comparisonCount`) to jedna zmienna, której czas życia kompilator przedłużył w wyniku zaimplementowania jej jako pola instancji (a nie jako zmiennej lokalnej). Wszystkie przypadki użycia pierwotnej zmiennej lokalnej są więc zmieniane na przypadki użycia nowego pola.

Wygenerowana klasa, `_LocalsDisplayClass`, jest **domknietiem**, czyli strukturą danych (w języku C# jest nią klasa) zawierającą wyrażenie i zmienne (w języku C# są nimi pola publiczne) potrzebne do przetworzenia tego wyrażenia.

Początek
5.0

ZAGADNIENIE DLA ZAAWANSOWANYCH

Przypadkowe przechwytywanie zmiennych z pętli

Jak myślisz, jak powinny wyglądać dane wyjściowe z listingu 13.22?

Listing 13.22. Przechwytywanie zmiennych z pętli w wersji C# 5.0

```

class CaptureLoop
{
    static void Main()
    {
        var items = new string[] { "Bolek", "Lolek", "Tola" };
        var actions = new List<Action>();
    }
}

```

```

foreach (string item in items)
{
    actions.Add( ()=> { Console.WriteLine(item); } );
}
foreach (Action action in actions)
{
    action();
}
}
}

```

Większość osób oczekuje, że efekt będzie taki jak w danych wyjściowych 13.3. W C# 5.0 rzeczywiście tak jest. Jednak w starszych wersjach języka generowane były informacje przedstawione w danych wyjściowych 13.4.

DANE WYJŚCIOWE 13.3. DANE WYJŚCIOWE W WERSJI C# 5.0

3.0

Bolek
Lolek
Tola

DANE WYJŚCIOWE 13.4. DANE WYJŚCIOWE W WERSJI C# 4.0

Tola
Tola
Tola

Wyrażenie lambda przechwytuje zmienną i zawsze używa jej najnowszej wartości. Nie jest tak, że lambda przechwytuje zmienną i zachowuje jej wartość z momentu utworzenia delegata. Zwykle efekt jest zgodny z oczekiwaniami programistów. W końcu zmienna `comparisonCount` z listingu 13.20 jest przechwytywana właśnie po to, by otrzymać jej najnowszą wartość po inkrementacji. Dotyczy to także zmiennych z pętli. Każdy delegat przechwytuje te same zmienne. Gdy wartość zmiennej z pętli się zmieni, modyfikacje będą widoczne we wszystkich delegatach przechwytyujących daną zmienną. Tak więc działanie kodu z wersji C# 4.0 jest uzasadnione, jednak prawie zawsze niezgodne z oczekiwaniemi autora kodu.

W C# 5.0 wprowadzono zmianę, zgodnie z którą zmienna z pętli `foreach` jest teraz uznawana za „nową” w każdej iteracji tej pętli. Dlatego za każdym razem, gdy delegat jest tworzony, przechwytuje on inną zmienną. Nie jest tak, że we wszystkich iteracjach używana jest ta sama zmienna. Ta modyfikacja nie dotyczy jednak pętli `for`. Jeśli napiszesz podobny kod z użyciem pętli `for`, każda zmienna zadeklarowana w nagłówku pętli będzie po przechwyceniu uznawana za jedną zmienną zewnętrzną. Jeśli chcesz napisać kod, który działa tak samo w wersji C# 5.0 i w starszych odmianach języka, zastosuj wzorzec zaprezentowany na listingu 13.23.

Listing 13.23. Rozwiązywanie problemu przechwytywania zmiennych z pętli w wersjach starszych niż C# 5.0

```

class DoNotCaptureLoop
{
    static void Main()
    {
        var items = new string[] { "Bolek", "Lolek", "Tola" };
    }
}

```

```
var actions = new List<Action>();
foreach (string item in items)
{
    string _item = item;
    actions.Add(
        ()=> { Console.WriteLine(_item); } );
}
foreach (Action action in actions)
{
    action();
}
```

Teraz wyraźnie widać, że w każdej iteracji pętli używana jest nowa zmienna. Dlatego każdy delegat przechwytuje inną zmienną.

Wskazówka

UNIKAJ przechwytywania w funkcjach anonimowych zmiennych z pętli.

3.0

Koniec
5.0

Drzewo wyrażeń

Wiesz już, że wyrażenia lambda pozwalają za pomocą związkowej składni zadeklarować wewnętrzierszowo metodę, którą można przekształcić na typ delegata. Lambdy w postaci wyrażeń (ale już nie lambdy w postaci instrukcji lub metody anonimowej) można ponadto przekształcić w **drzewo wyrażeń**. Delegat to obiekt umożliwiający przekazywanie metody w taki sam sposób jak w przypadku dowolnego innego obiektu i wywoływanie jej w dowolnym momencie. Drzewo wyrażeń to obiekt pozwalający przekazywać wynik przeprowadzonej przez kompilator analizy ciała lambdy. Do czego jednak może być potrzebna taka możliwość? Analizy kompilatora są oczywiście przydatne kompilatorowi w trakcie analizowania kodu CIL, dla którego jednak programista ma potrzebować obiektu reprezentującego analizy w trakcie wykonywania programu? Przyjrzyj się przykładowi.

Używanie wyrażeń lambda jako danych

Przyjrzyj się wyrażeniu lambda w poniższym kodzie:

```
persons.Where(
    person => person.Name.ToUpper() == "INIGO MONTOYA");
```

Załóżmy, że `persons` to tablica obiektów typu `Person`, a parametr formalny metody `Where` (reprezentowany jako argument w postaci wyrażenia lambda) to delegat typu `Func<Person, bool>`. Kompilator generuje metodę zawierającą kod z ciała lambdy. Ponadto generuje kod tworzący delegat tej metody i przekazujący delegat do metody `Where`. Metoda `Where` zwraca obiekt z kwerendą, którego wykonanie powoduje uruchomienie delegata dla każdego elementu tablicy w celu ustalenia wyniku kwerendy.

Teraz przyjmij, że obiekt `persons` nie jest typu `Person[]`, natomiast reprezentuje tabelę ze zdalnej bazy, zawierającą dane milionów ludzi. Informacje o każdym wierszu tabeli można przekazać z serwera do klienta, który tworzy wtedy obiekty typu `Person` odpowiadające poszczególnym wierszom. Wywołanie metody `Where` powoduje zwrócenie obiektu reprezentującego kwerendę. Jak określane są wyniki kwerendy, gdy klient ich zażąda?

Jedna z technik polega na przesłaniu kilku milionów wierszy danych z serwera do klienta. Można utworzyć obiekty typu `Person` reprezentujące wszystkie wiersze, utworzyć delegat na podstawie lambdy i wykonać delegat dla każdego obiektu typu `Person`. Jest to rozwiązanie zblizone do scenariusza z tablicą, jednak dużo, dużo kosztowniejsze.

Druga, znacznie lepsza technika polega na przesłaniu znaczenia lambdy („odfiltrowanie wszystkich wierszy, w których imię i nazwisko jest inne niż Inigo Montoya”) na serwer. Serwery bazodanowe są zoptymalizowane pod kątem szybkiego przetwarzania takich operacji filtrowania. Serwer może następnie przesyłać do klienta tylko pasujące wiersze, których jest niewiele. Wtedy klient zamiast tworzyć miliony obiektów typu `Person` i odrzucać większość z nich, musi utworzyć jedynie obiekty z danymi pasującymi do kwerendy, co ustalił serwer. Jak jednak przesyłać znaczenie lambdy na serwer?

Ta sytuacja spowodowała dodanie do języka drzew wyrażeń. Wyrażenie lambda przekształcone na drzewo wyrażeń staje się obiektem z danymi opisującymi wyrażenie lambda (a nie obiektem ze skompilowanym kodem z implementacją danej funkcji anonimowej). Ponieważ drzewo wyrażeń reprezentuje dane, a nie skompilowany kod, lambdę można przeanalizować w trakcie wykonywania kodu i wykorzystać uzyskane informacje na przykład do zbudowania kwerendy wykonywanej w bazie danych. Drzewo wyrażeń otrzymywane przez metodę `Where()` może zostać przekształcone na SQL-ową kwerendę przekazywaną do bazy danych (zobacz listing 13.24).

Listing 13.24. Konwersja drzewa wyrażeń na SQL-ową klauzulę where

```
persons.Where( person => person.Name.ToUpper() == "INIGO MONTOYA");
           ^          ^
           |          |
select * from Person where upper(Name) = 'INIGO MONTOYA';
```

Drzewo wyrażeń przekazywane do metody `Where()` zawiera informacje, że argument w postaci lambdy składa się z następujących elementów:

- wczytania właściwości `Name` obiektu typu `Person`,
- wywołania metody `ToUpper()` typu `string`,
- stałej `"INIGO MONTOYA"`,
- operatora równości, `==`.

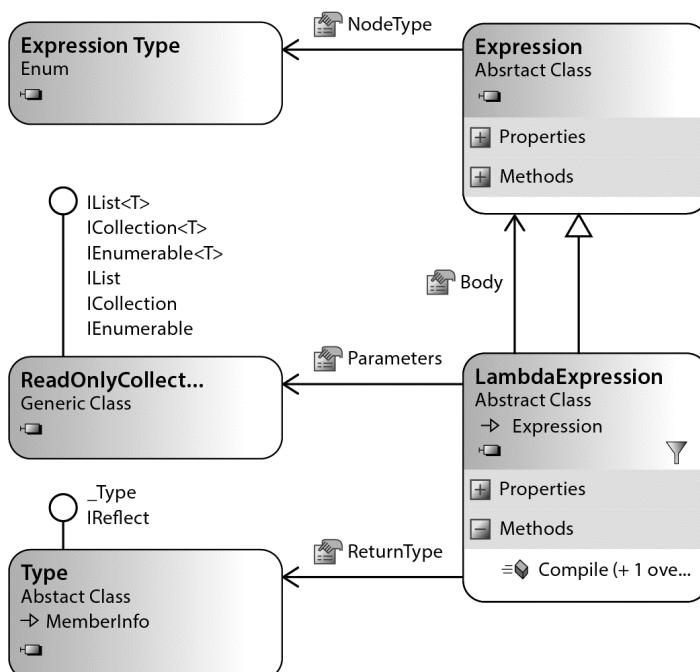
Metoda `Where()` przyjmuje dane i na podstawie ich analizy przekształca je na SQL-ową klauzulę `where` w utworzonym łańcuchu znaków z SQL-ową kwerendą. Jednak używanie SQL-a nie jest jedyną możliwością. Możesz zbudować mechanizm przetwarzania drzew, który przekształca wyrażenia na dowolny język.

Drzewa wyrażeń są grafami obiektów

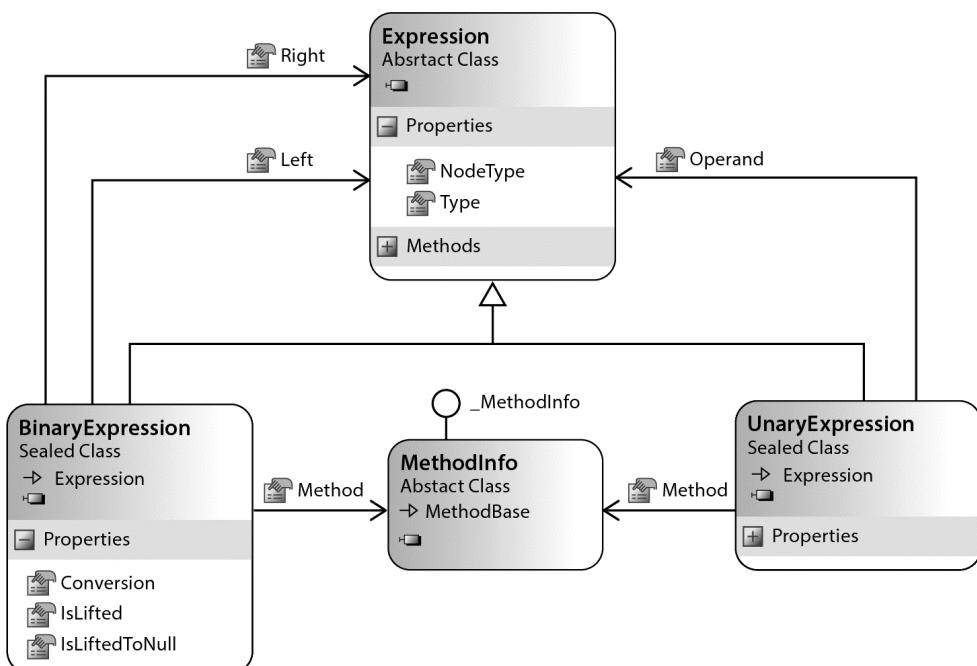
W czasie wykonywania programu lambda przekształcana na drzewo wyrażeń staje się grafem obejmującym obiekty z przestrzeni nazw `System.Linq.Expressions`. Główny obiekt w grafie („korzeń”) reprezentuje samą lambdę. Ten obiekt prowadzi do obiektów reprezentujących parametry, typ zwracanej wartości i ciało wyrażenia, co pokazano na rysunku 13.3. Graf obiektów obejmuje wszystkie informacje, które kompilator wywnioskował na podstawie lambdy. Te informacje można później wykorzystać w czasie wykonywania programu w celu utworzenia kwerendy. Główne wyrażenie lambda udostępnia też metodę `Compile`, generującą „w locie” kod CIL i tworzącą delegat odpowiadający opisanej lambdzie.

Na rysunku 13.4 pokazano typy używane w grafie obiektów do reprezentowania wyrażeń jedno- i dwuargumentowych z ciała lambdy.

Typ `UnaryExpression` reprezentuje wyrażenia takie jak `-count`. Ma on właściwość `Operand` typu `Expression` (reprezentuje ona jedno wyrażenie podrzędne). Typ `BinaryExpression` obejmuje właściwości `Left` i `Right`, reprezentujące dwa wyrażenia podrzędne. W obu typach występuje właściwość `NodeType`, określająca używany operator. Oba typy dziedziczą po klasie bazowej `Expression`. Istnieje około 30 typów reprezentujących wyrażenia (na przykład `NewExpression`, `ParameterExpression`, `MethodCallExpression` i `LoopExpression`). Odpowiadają one (prawie) wszystkim możliwym wyrażeniom z języków C# i Visual Basic.



Rysunek 13.3. Typy reprezentujące drzewo wyrażeń lambda



Rysunek 13.4. Typy reprezentujące drzewa wyrażeń jedno- i dwuargumentowych

Delegaty a drzewa wyrażeń

Poprawność wyrażeń lambda jest sprawdzana w czasie komplikacji na podstawie kompletnej analizy semantycznej, przeprowadzanej niezależnie od tego, czy lambda jest przekształcana na delegat, czy na drzewo wyrażeń. Lambda przekształcana na delegat skutkuje wygenerowaniem przez kompilator lambdy w postaci metody. Generowany jest wtedy kod, który w czasie wykonywania programu tworzy delegat dla tej metody. Jeśli lambda jest przekształcana na drzewo wyrażeń, kompilator generuje kod tworzący w czasie wykonywania programu obiekt typu **LambdaExpression**. Jednak jeśli używany jest interfejs API technologii LINQ (ang. *Language Integrated Query*), skąd kompilator ma wiedzieć, czy ma wygenerować delegat, wykonać kwerendę lokalnie, czy wygenerować drzewo wyrażeń, by przesyłać informacje o kwerendzie na zdalny serwer bazodanowy?

Metody używane do budowania kwerend w technologii LINQ, na przykład metoda **Where()**, są metodami rozszerzającymi. Wersje metod rozszerzających interfejs **IEnumerable<T>** przyjmują parametry w postaci delegatów. Metody rozszerzające interfejs **IQueryable<T>** przyjmują parametry w postaci drzew wyrażeń. Dlatego kompilator może wykorzystać typ kolekcji, której dotyczy kwerenda, w celu ustalenia, czy na podstawie lambdy przekazanej jako argument utworzyć delegat, czy drzewo wyrażeń.

Przyjrzyj się metodzie **Where()** w następującym kodzie:

```
persons.Where( person => person.Name.ToUpper() ==
    "INIGO MONTOYA");
```

Sygnatura metody rozszerzającej zadeklarowanej w klasie System.Linq.Enumerable wygląda tak:

```
public IEnumerable<TSource> Where<TSource>(
    this IEnumerable<TSource> collection,
    Func<TSource, bool> predicate);
```

Poniżej pokazano sygnaturę metody rozszerzającej zadeklarowanej w klasie System.Linq.
↳Queryable:

```
public IQueryable<TSource> Where<TSource>(
    this IQueryable<TSource> collection,
    Expression<Func<TSource, bool>> predicate);
```

Kompilator ustala, którą metodę rozszerzającą ma zastosować, na podstawie typu zmiennej persons w czasie komplikacji. Jeśli ten typ można przekształcić na typ IQueryable<Person>, wybierana jest metoda z typu System.Linq.Queryable. Skutkuje to przekształceniem lambdy na drzewo wyrażeń. W czasie wykonywania programu obiekt wskazywany przez zmienną persons otrzymuje dane z drzewa wyrażeń i może na podstawie tych danych zbudować SQL-ową kwerendę, przekazywaną następnie do bazy danych, gdy program potrzebuje wyniku kwerendy. Wynikiem wywołania metody Where jest w tym scenariuszu obiekt, który w reakcji na żądanie wyników kwerendy przesyła ją do bazy danych i generuje wynik.

Jeśli zmiennej persons nie można niejawnie przekształcić na typ IQueryable<Person>, natomiast możliwe jest jej niejawnie przekształcenie na typ IEnumerable<Person>, wybierana jest metoda z typu System.Linq.Enumerable, a lambda zostaje przekształcona na delegat. Wynikiem wywołania metody Where jest wtedy obiekt, który w reakcji na żądanie wyników kwerendy stosuje wygenerowany delegat jako predykat do każdego elementu kolekcji i zwraca wyniki pasujące do danego predykatu.

Badanie drzewa wyrażeń

Wiesz już, że przekształcenie wyrażenia lambda na typ Expression<TDelegate> prowadzi do utworzenia drzewa wyrażeń, a nie do zbudowania delegata. We wcześniejszej części rozdziału zobaczyłeś, jak przekształcić lambdę taką jak $(x,y) \Rightarrow x > y$ na typ delegata taki jak Func<int, int, bool>. W celu przekształcenia tej samej lambdy w drzewo wyrażeń wystarczy przeprowadzić konwersję na typ Expression<Func<int, int, bool>>, co pokazano na listingu 13.25. Następnie można zbadać wygenerowany obiekt i wyświetlić informacje o jego strukturze (dotyczy to także bardziej skomplikowanych drzew wyrażeń).

Listing 13.25. Badanie drzewa wyrażeń

```
using System;
using System.Linq.Expressions;
using static System.Linq.Expressions.ExpressionType;

public class Program
{
    public static void Main()
    {
```

```

Expression<Func<int, int, bool>> expression;
expression = (x, y) => x > y;
Console.WriteLine("----- {0} -----",
    expression);
PrintNode(expression.Body, 0);
Console.WriteLine();
Console.WriteLine();
expression = (x, y) => x * y > x + y;
Console.WriteLine("----- {0} -----",
    expression);
PrintNode(expression.Body, 0);
}
public static void PrintNode(Expression expression,
    int indent)
{
    if (expression is BinaryExpression)
        PrintNode(expression as BinaryExpression, indent);
    else
        PrintSingle(expression, indent);
}
private static void PrintNode(BinaryExpression expression,
    int indent)
{
    PrintNode(expression.Left, indent + 1);
    PrintSingle(expression, indent);
    PrintNode(expression.Right, indent + 1);
}
private static void PrintSingle(
    Expression expression, int indent) =>
Console.WriteLine("{0," + indent * 5 + "}{1}",
    "", NodeToString(expression));

private static string NodeToString(Expression expression) =>
    expression.NodeType switch
    {
        // using static ExpressionType
        Multiply => "*",
        Add => "+",
        Divide => "/",
        Subtract => "-",
        GreaterThan => ">",
        LessThan => "<",
        _ => expression.ToString() +
            " (" + expression.NodeType.ToString() + ")",
    };
}

```

3.0

Zauważ, że przekazanie instancji drzewa wyrażeń do metody `Console.WriteLine()` prowadzi do automatycznego przekształcenia drzewa wyrażeń na opisowy łańcuch znaków. Wszystkie obiekty generowane jako drzewa wyrażeń przesyłają metodę `ToString()`, dzięki czemu w trakcie diagnozowania można natychmiast zapoznać się z zawartością danego drzewa.

W danych wyjściowych 13.5 widać, że instrukcje `Console.WriteLine()` z metody `Main()` wyświetlają tekst reprezentujący ciało drzew wyrażeń.

DANE WYJŚCIOWE 13.5.

```
----- (x, y) => (x > y) -----
  x (Parameter)
>
  y (Parameter)

----- (x, y) => ((x * y) > (x + y)) -----
  x (Parameter)
*
  y (Parameter)
>
  x (Parameter)
+
  y (Parameter)
```

Warto zauważyć, że drzewo wyrażeń to kolekcja danych. W trakcie ich przetwarzania można je przekształcić na inny format. Tu następuje konwersja drzewa wyrażeń na opisowe łańcuchy znaków, jednak równie dobrze można przekształcić drzewo na wyrażenia w innym języku służącym do tworzenia kwerend.

Z pomocą rekurencji w funkcji `PrintNode()` pokazano, że węzły z drzew wyrażeń same są drzewami zawierającymi zero lub więcej podrzędnych drzew wyrażeń. „Korzeń” reprezentujący lambdę to wyrażenie będące ciałem lambdy (zapisane w jej właściwości `Body`). Każdy węzeł reprezentujący drzewo wyrażeń zawiera właściwość `NodeType` typu wyliczeniowego `ExpressionType`, który opisuje, jakiego rodzaju wyrażenie jest używane. Istnieje wiele typów wyrażeń (na przykład `BinaryExpression`, `ConditionalExpression`, `LambdaExpression`, `MethodCallExpression`, `ParameterExpression` i `ConstantExpression`). Wszystkie te typy są pochodne od typu `Expression`.

Zauważ, że choć biblioteka drzew wyrażeń obecnie zawiera obiekty reprezentujące większość instrukcji z języków C# i Visual Basic, żaden z tych języków nie obsługuje konwersji lambd w postaci instrukcji na drzewa wyrażeń. Możliwa jest tylko konwersja lambd w postaci wyrażeń.

Podsumowanie

Na początku tego rozdziału omówiono delegaty i ich wykorzystanie jako referencji do metod lub wywołań zwrotnych. Ten wartościowy mechanizm umożliwia przekazywanie w inne miejsce zbiorów instrukcji, które należy tam wykonać. Nie trzeba wtedy wykonywać ich bezpośrednio, poprzez wpisanie instrukcji.

Składnia wyrażeń lambda ma zastępować (choć nie eliminować) składnię metod anonimowych z wersji C# 2.0. Obie techniki umożliwiają programistom bezpośrednie przypisanie zbiorów instrukcji do zmiennej. Nie wymaga to definiowania odrębnej metody zawierającej dane instrukcje. To rozwiązanie zapewnia dużą swobodę w zakresie dynamicznego dodawania instrukcji w metodach. Jest to wartościowy mechanizm, który znacznie upraszcza programowanie z wykorzystaniem kolekcji i interfejsu API technologii LINQ.

W końcowej części rozdziału objaśniono drzewa wyrażeń. Opisano, że drzewa te są kompilowane do postaci obiektów reprezentujących semantyczną analizę wyrażeń lambda (a nie do postaci implementacji delegata). To ważne rozwiązywanie umożliwia działanie takim bibliotekom jak Entity Framework i LINQ to XML. Te biblioteki interpretują drzewa wyrażeń i wykorzystują je w kontekście innym niż kod CIL.

Wyrażenia lambda dzielą się na *lambdy w postaci instrukcji* i *lambdy w postaci wyrażeń*. Zarówno lambdy w postaci instrukcji, jak i lambdy w postaci wyrażeń są wyrażeniami lambda.

Jedną z kwestii, o których tylko побieżnie wspomniano w tym rozdziale, są delegaty związane z wieloma metodami (ang. *multicast delegates*). W następnym rozdziale szczegółowo opisano takie delegaty i wyjaśniono, że umożliwiają one stosowanie opartego na zdarzeniach wzorca publikuj-subskrybij.

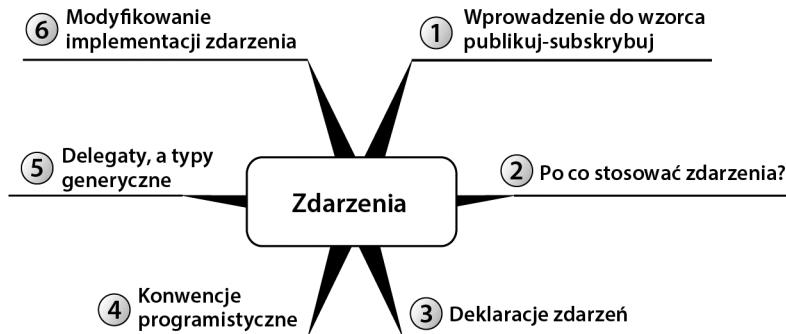
14

Zdarzenia

Początek
2.0

WRODZIALE 13. DOWIEDZIAŁEŚ SIĘ, jak wskazywać metodę za pomocą instancji typu delegata i wywoływać tę metodę przy użyciu delegata. Delegaty to cegiełki używane w większym wzorcu *publikuj-subskrybuj* (nazywanym też wzorcem *obserwator*). Tematem tego rozdziału jest właśnie wykorzystanie delegatów w tym wzorcu. Prawie wszystkie zadania omawiane w tym rozdziale można wykonać za pomocą samych delegatów. Jednak zdarzenia zapewniają dodatkową hermetyzację, co ułatwia implementację wzorca publikuj-subskrybuj i zmniejsza ryzyko wystąpienia w nim błędów.

W rozdziale 13. każdy delegat wskazywał jedną metodę. Jednak jeden delegat może wskazywać całą kolekcję wykonywanych sekwencyjnie metod. Tak działają **delegaty typu multicast** (ang. *multicast delegates*), czyli związane z więcej niż jedną metodą. Dzięki takim delegatom powiadomienie o jednym zdarzeniu (na przykład o zmianie stanu obiektu) jest publikowane do wielu subskrybentów.



Choć zdarzenia były dostępne już w wersji C# 1.0, wprowadzenie typów generycznych w C# 2.0 znacznie zmieniło konwencje programistyczne, ponieważ dzięki generycznym typom delegatów nie trzeba już było deklarować odrębnych delegatów odpowiadających każdej możliwej sygnaturze zdarzeń. Dlatego w tym rozdziale przyjęto, że używana jest przynajmniej wersja C# 2.0. Czytelnicy, którzy wciąż korzystają z C# 1.0, też mogą posługiwać się zdarzeniami, ale muszą w tym celu deklarować własne typy delegatów (w sposób opisany w rozdziale 13.).

Implementacja wzorca publikuj-subskrybij za pomocą delegatów typu multicast

Wyobraź sobie urządzenie do sterowania temperaturą, w którym grzejnik i chłodnica są podłączone do tego samego termostatu. Aby maszyna we właściwych momentach włączała się i wyłączała, trzeba powiadomić ją o zmianach temperatury. Jeden termostat publikuje wtedy zmiany temperatury odbierane przez wielu subskrybentów (grzejnik i chłodnicę). W następnym podrozdziale opisano kod takiego systemu¹.

Definiowanie metod subskrybujących

Zacznij od zdefiniowania typów Heater i Cooler (zobacz listing 14.1).

Listing 14.1. Implementacja typów subskrybujących zdarzenia — Heater i Cooler

2.0

```
class Cooler
{
    public Cooler(float temperature)
    {
        Temperature = temperature;
    }
    // Chłodnica jest uruchamiana, gdy temperatura przekracza określony poziom.
    public float Temperature { get; set; }

    // Powiadamianie o zmianie temperatury w danym obiekcie.
    public void OnTemperatureChanged(float newTemperature)
    {
        if (newTemperature > Temperature)
        {
            System.Console.WriteLine("Chłodnica: włączona");
        }
        else
        {
            System.Console.WriteLine("Chłodnica: wyłączona");
        }
    }
}

class Heater
{
    public Heater(float temperature)
    {
        Temperature = temperature;
    }

    public float Temperature { get; set; }

    public void OnTemperatureChanged(float newTemperature)
```

¹ W tym przykładzie używane jest pojęcie *termostat*, ponieważ wiele osób kojarzy to urządzenie z systemami chłodniczo-grzejnymi. Jednak technicznie lepszą nazwą byłby *termometr*.

```
{  
    if (newTemperature < Temperature)  
    {  
        System.Console.WriteLine("Grzejnik: włączony");  
    }  
    else  
    {  
        System.Console.WriteLine("Grzejnik: wyłączony");  
    }  
}
```

Te dwie klasy są niemal identyczne. Różnią się tylko porównaniami temperatury. Można nawet wyeliminować jedną z tych klas, jeśli w metodzie `OnTemperatureChanged` użyty zostanie delegat prowadzący do metody odpowiedzialnej za takie porównania. W obu klasach przechowywana jest temperatura, której osiągnięcie powoduje włączenie danej jednostki. Ponadto obie klasy udostępniają metodę `OnTemperatureChanged()`. Wywołanie tej metody pozwala poinformować obiekt klasy Heater lub Cooler o zmianie temperatury. W kodzie metody wartość parametru `newTemperature` jest porównywana z zapisaną temperaturą progową, by ustalić, czy należy włączyć urządzenie.

Metody `OnTemperatureChanged()` to metody subskrybujące (nazywane też *odbiornikami*). Muszą one mieć parametry i typ zwracanej wartości pasujące do delegata z opisanej dalej klasy `Thermostat`.

2.0

Definiowanie klasy publikującej zdarzenia

Klasa `Thermostat` odpowiada za zgłaszanie zmian temperatury obiektom typów `Heater` i `Cooler`. Kod klasy `Thermostat` znajdziesz na listingu 14.2.

Listing 14.2. Definiowanie klasy publikującej zdarzenia — `Thermostat`

```
public class Thermostat  
{  
    // Definicja składowej publikującej zdarzenia (początkowo bez parametru sender).  
    public Action<float>? OnTemperatureChange { get; set; }  
  
    public float CurrentTemperature { get; set; }  
}
```

Klasa `Thermostat` obejmuje właściwość `OnTemperatureChange` typu delegata `Action<float>`. W tej właściwości zapisana jest lista subskrybentów. Zauważ, że do przechowywania wszystkich subskrybentów potrzebne jest tylko jedno pole. Oznacza to, że obiekty typów `Cooler` i `Heater` będą otrzymywać powiadomienia o zmianie temperatury od jednego nadawcy.

Ostatnia składowa klasy `Thermostat` to właściwość `CurrentTemperature`. Ta właściwość służy do ustawiania i pobierania aktualnej temperatury, o której informuje klasa `Thermostat`.

Łączenie subskrybentów z nadawcą

W ostatnim kroku wszystkie opisane wcześniej elementy są łączone ze sobą w metodzie Main(). Na listingu 14.3 przedstawiono przykładową metodę Main().

Listing 14.3. Łączenie nadawcy z subskrybentami

```
2.0
class Program
{
    public static void Main()
    {
        Thermostat thermostat = new Thermostat();
        Heater heater = new Heater(60);
        Cooler cooler = new Cooler(80);
        string temperature;

        thermostat.OnTemperatureChange +=  
            heater.OnTemperatureChanged;  
        thermostat.OnTemperatureChange +=  
            cooler.OnTemperatureChanged;

        Console.WriteLine("Podaj temperaturę: ");
        temperature = Console.ReadLine();
        thermostat.CurrentTemperature = int.Parse(temperature);
    }
}
```

W kodzie na tym listingu zarejestrowano dwóch subskrybentów (heater.OnTemperatureChanged i cooler.OnTemperatureChanged) delegata OnTemperatureChange. Subskrybentów zarejestrowano bezpośrednio w wyniku przypisania za pomocą operatora +=.

Na podstawie wysokości temperatury wprowadzonej przez użytkownika jako dane wejściowe można ustawić właściwość CurrentTemperature obiektu thermostat. Na razie jednak nie napisano jeszcze kodu publikującego zdarzenie zmiany temperatury, co pozwala odebrać je przez subskrybentów.

Wywoływanie delegata

Każda zmiana wartości właściwości CurrentTemperature w obiekcie klasy Thermostat powinna prowadzić do **wywołania delegata** w celu powiadomienia subskrybentów (obiekty heater i cooler) o zmianie temperatury. Aby osiągnąć ten cel, należy zmodyfikować właściwość CurrentTemperature. Powinna ona zapisywać nową temperaturę i publikować powiadomienie przesybane do wszystkich subskrybentów. Zmodyfikowany kod przedstawiono na listingu 14.4.

Listing 14.4. Wywoływanie delegata (bez sprawdzania wartości null)

```
public class Thermostat
{
    ...
    public float CurrentTemperature
    {
        get { return _CurrentTemperature; }
```

```

set
{
    if (value != CurrentTemperature)
    {
        _CurrentTemperature = value;
        // NIEKOMPLETNE: trzeba sprawdzać, czy wartość jest różna od null.
        // Wywołanie kierowane do subskrybentów.
        OnTemperatureChange(value);
    }
}
private float _CurrentTemperature;
}

```

Teraz przypisywanie wartości do właściwości CurrentTemperature obejmuje specjalny kod powiadający subskrybentów o zmianie tej wartości. Wywołanie powiadające wszystkich subskrybentów to jedna instrukcja języka C# — OnTemperatureChange(value). Ta pojedyncza instrukcja publikuje zmianę temperatury oraz przekazuje informacje o niej do obiektów typów Cooler i Heater. Tu w praktyce pokazano, że możliwe jest powiadomianie wielu subskrybentów w jednym wywołaniu. Dlatego czasem stosuje się nazwę „delegaty typu multicast”.

W C# 8.0 bezpośrednie wywołanie właściwości CurrentTemperature powoduje zgłoszenie ostrzeżenia o dereferencji wartości dopuszczającej null, co oznacza, że potrzebne jest sprawdzanie wartości null.

2.0

 Początek
8.0
Koniec
8.0

 Początek
6.0

Sprawdzanie, czy wartość jest różna od null

Na listingu 14.4 brakuje ważnej części kodu publikującego zdarzenia. Jeśli nie zarejestrowano żadnego subskrybenta powiadomień, wartość właściwości OnTemperatureChange to null. Wtedy instrukcja OnTemperatureChange(value) powoduje zgłoszenie wyjątku typu NullReferenceException. Aby uniknąć takich sytuacji, należy przed zgłoszeniem zdarzenia sprawdzić, czy wartość właściwości jest różna od null. Na listingu 14.5 pokazano, jak zrobić to za pomocą wprowadzonego w wersji C# 6.0 operatora ?.. Dopiero po sprawdzeniu wartości następuje wywołanie metody Invoke().

Listing 14.5. Wywoływanie delegata

```

public class Thermostat
{
    // Definicja metody publikującej zdarzenia
    public Action<float>? OnTemperatureChange { get; set; }

    public float CurrentTemperature
    {
        get { return _CurrentTemperature; }
        set
        {
            if (value != CurrentTemperature)
            {
                _CurrentTemperature = value;
            }
        }
    }
}

```

```

    // Jeśli istnieją subskrybenci,
    // należy ich powiadomić o zmianach temperatury
    // za pomocą skierowanego do nich wywołania.
    OnTemperatureChange?.Invoke(value); // Z wersji C# 6.0.
}
}
}
private float _CurrentTemperature;
}

```

Zwróć uwagę na wywołanie metody `Invoke()` po użyciu operatora `?..`. Choć tę metodę można wywołać także za pomocą samego operatora kropki, nie ma to większego sensu, ponieważ wtedy kod działa jak bezpośrednie wywołanie delegata (zobacz wywołanie `OnTemperatureChange(value)` na listingu 14.4). Ważną zaletą używania operatora `?.` jest specjalny mechanizm gwarantujący, że po sprawdzeniu, czy wartość jest różna od `null`, nie ma możliwości, by subskrybent zrezygnował z subskrypcji (co mogłoby skutkować tym, że delegat znów przyjmie wartość `null`).

Koniec
6.0
2.0

■ ZAGADNIENIE DLA ZAAWANSOWANYCH

Wywoływanie delegatów w wersjach starszych niż C# 6.0

Niestety, w wersjach starszych niż C# 6.0 nie istnieje tego typu specjalny mechanizm, który pozwala bez ryzyka późniejszej zmiany wartości sprawdzić, czy wartość jest różna od `null`. Dlatego we wcześniejszych wersjach języka C# potrzebny jest dłuższy kod, co pokazano na listingu 14.6.

Listing 14.6. Wywoływanie delegata ze sprawdzaniem, czy jego wartość jest różna od `null` (kod dla wersji starszych niż C# 6.0)

```

public class Thermostat
{
    // Definicja metody publikującej zdarzenia.
    public Action<float>? OnTemperatureChange { get; set; }

    public float CurrentTemperature
    {
        get{return _CurrentTemperature;}
        set
        {
            if (value != CurrentTemperature)
            {
                CurrentTemperature = value;
                // Jeśli istnieją subskrybenci,
                // należy ich powiadomić o zmianie temperatury
                // za pomocą skierowanego do nich wywołania.
                Action<float>? localOnChange =
                    OnTemperatureChange;
                if (localOnChange != null)
                {
                    // Wywołanie skierowane do subskrybentów.
                    localOnChange(value);
                }
            }
        }
    }
}

```

```
        }  
    }  
    private float _CurrentTemperature;  
}
```

Zamiast bezpośrednio sprawdzać, czy wartość to null, ten kod najpierw przypisuje właściwość OnTemperatureChange do drugiej zmiennej typu delegata, localOnChange. Ta prosta modyfikacja gwarantuje, że nawet jeśli wszyscy subskrybenci powiązani z właściwością OnTemperatureChange zostaną usunięci (przez inny wątek) w czasie między sprawdzeniem wartości a przesłaniem powiadomienia, nie wystąpi wyjątek typu NullReferenceException.

W dalszej części książki we wszystkich przykładach delegaty są wywoływane z wykorzystaniem operatora ?. z wersji C# 6.0.

Wskazówki

SPRAWDZAJ, czy wartość delegata jest różna od null. Dopiero potem go wywołuj.

STOSUJ operator ?. przed wywołaniem metody Invoke() (od wersji C# 6.0).

2.0

ZAGADNIENIE DLA ZAAWANSOWANYCH

Operator == delegatów zwraca nową instancję

Ponieważ delegat jest typu referencyjnego, może się wydać zaskakujące, że przypisanie go do zmiennej lokalnej i późniejsze używanie tej zmiennej wystarcza do tego, by sprawdzanie wartości null było bezpieczne ze względu na wątki. Ponieważ zmienna localOnChange prowadzi do tej samej lokalizacji co delegat z właściwością OnTemperatureChange, możesz sądzić, że wszelkie zmiany wartości właściwości OnTemperatureChange są widoczne także w zmiennej localOnChange.

Jest jednak inaczej, ponieważ wywołania OnTemperatureChange == <odbiorca> nie usuwają delegata z właściwości OnTemperatureChange, tak by zawierał on o jeden delegat mniej niż wcześniej. Zamiast tego takie wywołanie przypisuje zupełnie nowy delegat typu multicast i nie wpływa na pierwotny delegat, do którego prowadzi zmienna localOnChange.

ZAGADNIENIE DLA ZAAWANSOWANYCH

Wywoływanie delegata z zachowaniem bezpieczeństwa ze względu na wątki

Jeśli subskrybenci delegata mogą być dodawani i usuwani w różnych wątkach, to przed sprawdzeniem, czy wartość delegata to null, warto (o czym wspomniano wcześniej) skopiować referencję do delegata do zmiennej lokalnej. Jednak choć to podejście zapobiega wywoływaniu delegata o wartości null, nie chroni przed wystąpieniem sytuacji wyścigu. Możliwe, że jeden wątek utworzy kopię, a inny ustawi delegat na wartość null, po czym pierwotny wątek uruchomi wywołanie dla delegata o wcześniejszej wartości i w ten sposób powiadomi subskrybenta, który nie znajduje się już na liście. W subskrybentach w programach wielowątkowych należy dbać o to, by kod w takiej sytuacji działał poprawnie (ponieważ może się zdarzyć, że „dawny” subskrybent otrzyma wywołanie).

Operatory używane do delegatów

Aby w przykładzie obejmującym klasę Thermostat połączyć dwóch subskrybentów, zastosowano operator `+=`. Ten operator przyjmuje pierwszy delegat, a następnie dodaje do łańcucha drugi. Po zwróceniu sterowania przez metodę powiązaną z pierwszym delegatem uruchamiana jest metoda powiązana z drugim. Aby usunąć delegat z łańcucha, należy zastosować operator `-=`, co pokazano na listingu 14.7.

Listing 14.7. Używanie operatorów `+= i -=` do delegatów

```
// ...
Thermostat thermostat = new Thermostat();
Heater heater = new Heater(60);
Cooler cooler = new Cooler(80);

Action<float> delegate1;
Action<float> delegate2;
Action<float>? delegate3;

2.0
delegate1 = heater.OnTemperatureChanged;
delegate2 = cooler.OnTemperatureChanged;

Console.WriteLine("Wywołanie obu delegatów");
delegate3 = delegate1;
delegate3 += delegate2;
delegate3(90);

Console.WriteLine("Wywołanie tylko delegata delegate2");
delegate3 -= delegate1;
delegate3(30);
// ...
```

Wynik działania kodu z listingu 14.7 pokazano w danych wyjściowych 14.1.

DANE WYJŚCIOWE 14.1.

```
Wywołanie obu delegatów
Grzejnik: wyłączony
Chłodnica: włączona
Wywołanie tylko delegata delegate2
Chłodnica: wyłączona
```

Początek → Ponadto można posługiwać się operatorami `+ i -` do łączenia delegatów, co pokazano na listingu 14.8.

Listing 14.8. Używanie operatorów `+ i -` do delegatów

```
// ...
Thermostat thermostat = new Thermostat();
Heater heater = new Heater(60);
Cooler cooler = new Cooler(80);

Action<float> delegate1;
Action<float> delegate2;
Action<float> delegate3;
```

```
// Uwaga: w wersji C# 1.0 należy zastosować składnię  
// new Action(cooler.OnTemperatureChanged);  
delegate1 = heater.OnTemperatureChanged;  
delegate2 = cooler.OnTemperatureChanged;  
  
Console.WriteLine("Łączenie delegatów za pomocą operatora +:");  
delegate3 = delegate1 + delegate2;  
delegate3(60);  
  
Console.WriteLine("Odłączanie delegatów za pomocą operatora -:");  
delegate3 = (delegate3 - delegate2)!;  
delegate3(60);  
// ...
```

Użycie operatora przypisania powoduje usunięcie wszystkich wcześniejszych subskrybentów i pozwala zastąpić ich nowymi. Jest to kłopotliwa cecha delegatów. Zbyt łatwo jest pomyłkowo wpisać operator przypisania, gdy w rzeczywistości potrzebny jest operator `+=`. Rozwiązaniem tego problemu są *zdarzenia*, opisane w podrozdziale „Zdarzenia” w dalszej części rozdziału.

2.0

Operatory `+ i -` oraz powiązane z nimi operatory `+= i -=` są wewnętrznie zaimplementowane za pomocą metod statycznych `System.Delegate.Combine()` i `System.Delegate.Remove()`. Te metody przyjmują dwa parametry typu `Delegate`. Pierwsza z nich, `Combine()`, łączy oba parametry w taki sposób, że pierwszy parametr na liście delegatów prowadzi do drugiego. Druga metoda, `Remove()`, przeszukuje łańcuch delegatów podany w pierwszym parametrze i usuwa z tego łańcucha delegat wskazany za pomocą drugiego parametru. Ponieważ metoda `Remove()` może zwrócić wartość `null`, używany jest tu wprowadzony w wersji C# 8.0 operator braku wartości `null`, aby kompilator przyjął, że nadal dostępna jest prawidłowa instancja delegata.

W metodzie `Combine()` warto zwrócić uwagę na to, że jeden lub oba jej parametry mogą mieć wartość `null`. Jeśli jeden z parametrów to `null`, metoda `Combine()` zwraca drugi parametr. Jeżeli oba parametry to `null`, metoda zwraca `null`. To wyjaśnia, dlaczego można wywołać instrukcję `thermostat.OnTemperatureChange += heater.OnTemperatureChanged;` i nie zgłosi ona wyjątku nawet w sytuacji, gdy wartość delegata `thermostat.OnTemperatureChange` to `null`.

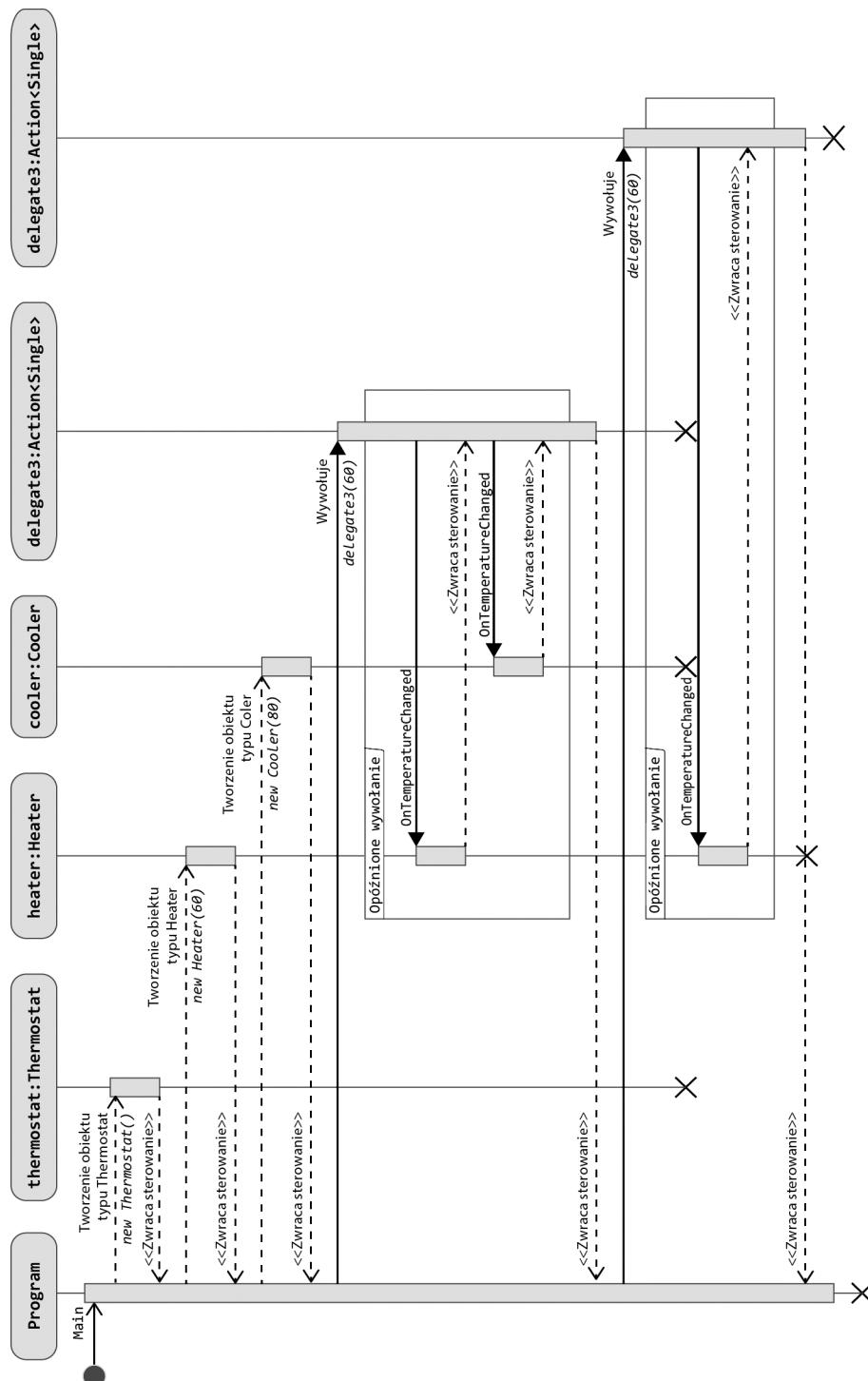
Koniec
8.0

Wywołania sekwencyjne

Na rysunku 14.1 przedstawiono sekwencyjne kierowanie powiadomień do obiektów `heater` i `cooler`.

Choć w kodzie znajduje się tylko jedno wywołanie `OnTemperatureChange()`, jest ono kierowane do obu subskrybentów. W ten sposób jedno wywołanie pozwala powiadomić obiekty `cooler` i `heater` o zmianie temperatury. Jeśli dodasz kolejnych subskrybentów, także oni zostaną powiadomieni o wywołaniu `OnTemperatureChange()`.

Choć powiadomienia są kierowane do wszystkich subskrybentów w wyniku jednego wywołania `OnTemperatureChange()`, subskrybenci są wywoływani sekwencyjnie, a nie jednocześnie. Dzieje się tak, ponieważ ich wywołania odbywają się w tym samym wątku.



Rysunek 14.1. Diagram z sekwencją wywołań delegatów

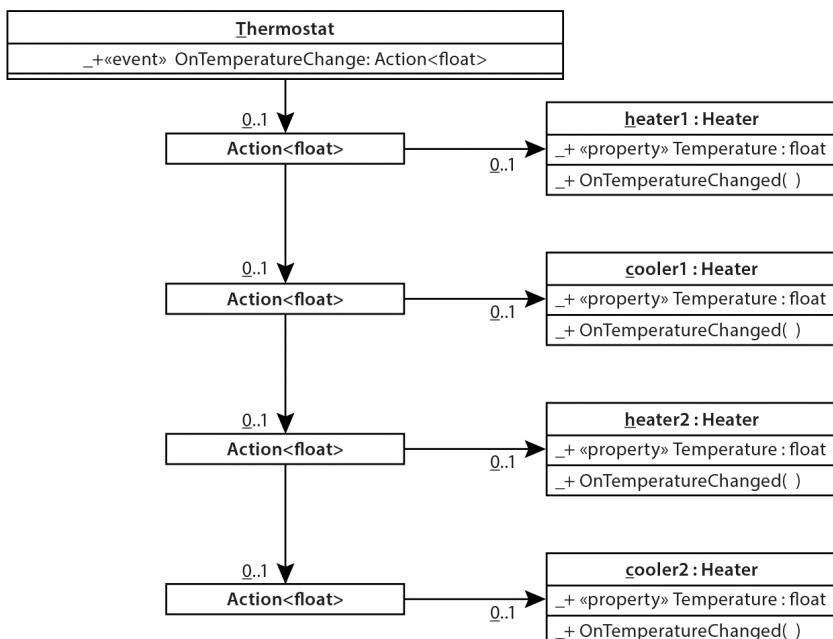
ZAGADNIENIE DLA ZAAWANSOWANYCH

Wewnętrzne mechanizmy delegatów typu multicast

Aby zrozumieć, jak działają zdarzenia, należy wrócić do pierwszego omówienia wewnętrznych mechanizmów typu System.Delegate. Warto przypomnieć, że słowo kluczowe delegate to alias reprezentujący typ pochodny od typu System.MulticastDelegate, który z kolei dziedziczy po klasie System.Delegate (obejmuje ona — potrzebną dla metod niestatycznych — referencję do obiektu, a także referencję do metody). Gdy tworzysz delegat, kompilator automatycznie stosuje typ System.MulticastDelegate, a nie typ System.Delegate. Klasa MulticastDelegate (podobnie jak jej klasa bazowa Delegate) obejmuje referencję do obiektu i referencję do metody, a ponadto zawiera referencję do innego obiektu typu MulticastDelegate.

Gdy dodajesz metodę do delegata typu multicast, obiekt klasy MulticastDelegate tworzy nową instancję typu delegata, zapisuje referencję do obiektu i referencję do dodawanej metody w tej nowej instancji, a następnie dodaje nową instancję jako następny element do listy instancji. W efekcie obiekt klasy MulticastDelegate zawiera listę powiązaną obiektów typu Delegate. W przykładowym programie z termostatem taka lista może wyglądać tak jak na rysunku 14.2.

2.0



Rysunek 14.2. Delegaty typu multicast połączone w łańcuch

Gdy wywoływany jest delegat typu multicast, sekwencyjnie uruchamiana jest każda instancja delegata z listy powiązanej. Zwykle delegaty są wywoływanie w kolejności, w jakiej je dodano, jednak w specyfikacji interfejsu CLI nie jest to wymagane. Ponadto kolejność delegatów może zostać zmieniona. Dlatego programiści nie powinni tworzyć kodu zależnego od porządku wywołań.

Obsługa błędów

Z powodu obsługi błędów sekwencyjny charakter zgłaszania powiadomień ma bardzo duże znaczenie. Jeśli jeden z subskrybentów zgłosi wyjątek, subskrybenci z dalszych pozycji w łańcuchu nie otrzymają powiadomienia. Pomyśl na przykład, co się stanie, jeśli zmodyfikujesz metodę `OnTemperatureChanged()` w klasie `Heater` w taki sposób, by zgłaszała wyjątek. Nowy kod znajdziesz na listingu 14.9.

Listing 14.9. Metoda `OnTemperatureChanged()` zgłaszająca wyjątek

```
2.0
class Program
{
    public static void Main()
    {
        Thermostat thermostat = new Thermostat();
        Heater heater = new Heater(60);
        Cooler cooler = new Cooler(80);
        string temperature;

        thermostat.OnTemperatureChange +=
            heater.OnTemperatureChanged;
        // Kod dla wersji C# 3.0. Jeśli używasz wersji C# 2.0,
        // zastosuj metodę anonimową.
        thermostat.OnTemperatureChange +=
            (newTemperature) =>
        {
            throw new InvalidOperationException();
        };
        thermostat.OnTemperatureChange +=
            cooler.OnTemperatureChanged;

        Console.Write("Podaj temperaturę: ");
        temperature = Console.ReadLine();
        thermostat.CurrentTemperature = int.Parse(temperature);
    }
}
```

Na rysunku 14.3 pokazano zmodyfikowany diagram sekwencji wywołań. Choć obiekty `cooler` i `heater` subskrybują komunikaty, wyjątek zgłoszony w wyrażeniu lambda przerywa łańcuch i uniemożliwia odbiór powiadomienia obiektowi `cooler`.

Aby uniknąć tego problemu i umożliwić odbiór powiadomień wszystkim subskrybentom (niezależnie od działania elementów z wcześniejszych pozycji w łańcuchu), trzeba ręcznie pobierać elementy z listy subskrybentów i wywoływać je niezależnie. Na listingu 14.10 pokazano zmiany potrzebne we właściwości `CurrentTemperature`. Wynik działania kodu znajdziesz w danych wyjściowych 14.2.


```

        "Wystąpiły wyjątki zgłoszone przez subskrybentów " +
        "zdarzenia OnTemperatureChange.",

    exceptionCollection);
}
}
}
}

private float _CurrentTemperature;
}

```

DANE WYJŚCIOWE 14.2.

```

Podaj temperaturę: 45
Grzejnik: włączony
Chłodnica: wyłączona
There were exceptions thrown...

```

2.0

Na listingu 14.10 pokazano, że można pobrać listę subskrybentów za pomocą metody GetInvocationList() delegata. Pobieranie kolejnych elementów listy pozwala otrzymać poszczególnych subskrybentów. Jeśli następnie umieścisz każde wywołanie subskrybenta w bloku try-catch, będziesz mógł obsługiwać błędy przed wznowieniem przetwarzania pętli z subskrybentami. W tym przykładzie delegat odbiorcy zgłasza wyjątek, jednak obiekt cooler mimo to otrzymuje powiadomienie o zmianie temperatury. Po przesłaniu kompletu powiadomień kod z listingu 14.10 informuje o wszystkich wyjątkach, zgłaszając wyjątek typu AggregateException. Ten typ to opakowanie kolekcji wyjątków (dostęp do nich można uzyskać za pomocą właściwości InnerExceptions). Dzięki temu każdy wyjątek jest zgłoszany, a wszyscy subskrybenci otrzymują powiadomienia.

Wartości zwarcane przez metodę i przekazywanie danych przez referencję

Przetwarzanie kolejnych pozycji z listy wywołań delegatów (zamiast bezpośredniego aktywowania powiadomienia) jest przydatne także w innym scenariuszu. Dotyczy on delegatów, które albo zwracają wartość inną niż void, albo mają parametry ref lub out. W przykładzie dotyczącym termostatu delegat OnTemperatureChange jest typu Action<float>, dlatego zwraca wartość void i nie ma parametrów ref lub out. W efekcie do nadawcy nie są zwarcane żadne dane. Ta kwestia jest istotna, ponieważ wywołanie delegata może prowadzić do przesyłania powiadomień do wielu subskrybentów. Jeśli każdy subskrybent zwraca wartość, nie wiadomo, która ze zwróconych wartości ma zostać użyta.

Jeśli zmodyfikujesz delegat OnTemperatureChange w taki sposób, by zwracał wartość z wyliczenia, opisującą, czy w wyniku zmiany temperatury urządzenie zostało włączone, nowy delegat będzie typu Func<float, Status>, gdzie Status to wyliczenie z elementami On i Off. Wszystkie metody subskrybujące muszą mieć tę samą sygnaturę co delegat, dlatego też muszą zwracać wartość informującą o stanie urządzenia. Ponieważ delegat OnTemperatureChange może reprezentować łańcuch delegatów, konieczne jest zastosowanie wzorca użytego wcześniej do obsługi błędów. Oznacza to, że trzeba po kolej po kolej pobrać każdy element z listy wywołań

delegatów (wywołując metodę `GetInvocationList()`), by otrzymać każdą ze zwróconych wartości. Także typy delegatów z parametrami `ref` i `out` wymagają specjalnej obsługi. Jednak choć w wyjątkowych sytuacjach możliwe jest zastosowanie opisanego podejścia, zalecanym rozwiązaniem jest unikanie problemu — wystarczy tworzyć delegaty zwracające `void`.

Zdarzenia

Delegaty w postaci używanej do tej pory powodowały dwa poważne problemy. Aby je przewyciężyć, w języku C# wprowadzono słowo kluczowe `event`. W tym podrozdziale zobaczymy, dlaczego warto stosować zdarzenia i jak one działają.

Po co stosować zdarzenia?

W rozdziałach tym i poprzednim opisano wszystko, co trzeba wiedzieć na temat działania delegatów. Niestety, słabe punkty w budowie delegatów mogą sprawić, że programista przypadkowo popełni błąd. Wynika to z tego, że za pomocą samych delegatów ani na poziomie subskrypcji, ani na poziomie publikacji nie można zapewnić wystarczającej kontroli hermetyzacji kodu. Zdarzenia pozwalają zewnętrznym klasom wyłącznie na dodawanie do nadawcy metod subskrybujących zdarzenia (za pomocą operatora `+=`) i rezygnowanie z subskrypcji (za pomocą operatora `-=`). Ponadto zgłaszanie zdarzenia jest możliwe tylko w zawierającej je klasie.

Hermetyzacja subskrypcji

Wcześniej wyjaśniono, że za pomocą operatora przypisania można przypisać jeden delegat do drugiego. Niestety, ten mechanizm często jest źródłem błędów. Przyjrzyj się listingowi 14.11.

Listing 14.11. Skutki użycia operatora przypisania = zamiast +=

```
class Program
{
    public static void Main()
    {
        Thermostat thermostat = new Thermostat();
        Heater heater = new Heater(60);
        Cooler cooler = new Cooler(80);
        string temperature;

        // Uwaga: w wersji C# 1.0 zastosuj kod
        // new Action(cooler.OnTemperatureChanged).
        thermostat.OnTemperatureChange =
            heater.OnTemperatureChanged;

        // Błąd: operator przypisania usuwa
        // wcześniej przypisane wartości.
        thermostat.OnTemperatureChange =
            cooler.OnTemperatureChanged;

        Console.WriteLine("Podaj temperaturę: ");
    }
}
```

```

        temperature = Console.ReadLine();
        thermostat.CurrentTemperature = int.Parse(temperature);
    }
}

```

Listing 14.11 jest prawie identyczny jak listing 14.7. Różnica polega na tym, że zamiast operatora `+=` zastosowano operator `=`. W efekcie gdy kod przypisuje do właściwości `OnTemperatureChange` wartość `cooler.OnTemperatureChanged`, z listy wartości usuwany jest element `heater.OnTemperatureChanged`, ponieważ przypisanie powoduje zastąpienie wcześniejszego łańcucha nowym. Prawdopodobieństwo błędnego użycia operatora `=` w miejscu, gdzie należy zastosować operator `+=`, jest tak wysokie, że byłoby lepiej, gdyby operator `=` nie był w tym kontekście obsługiwany (powinien on być dostępny tylko w kodzie klasy nadawcy). Słowo kluczowe `event` zapewnia dodatkowy poziom hermetyzacji, chroniący przed przypadkowym usunięciem innych subskrybentów.

2.0

Hermetyzacja procesu publikacji

Druga ważna różnica między delegatami i zdarzeniami polega na tym, że zdarzenia gwarantują, iż tylko klasa zawierająca dane zdarzenie może zgłaszać powiadomienia o nim. Przyjrzyj się kodowi z listingu 14.12.

Listing 14.12. Zgłaszanie zdarzenia poza zawierającą je klasą

```

class Program
{
    public static void Main()
    {
        Thermostat thermostat = new Thermostat();
        Heater heater = new Heater(60);
        Cooler cooler = new Cooler(80);
        string temperature;

        // Uwaga: jeśli korzystasz z wersji C# 1.0, zastosuj instrukcję
        // new Action(cooler.OnTemperatureChanged).
        thermostat.OnTemperatureChange +=
            heater.OnTemperatureChanged;

        thermostat.OnTemperatureChange +=
            cooler.OnTemperatureChanged;

        thermostat.OnTemperatureChange(42);
    }
}

```

Na listingu 14.12 klasa `Program` może wywołać delegat `OnTemperatureChange` nawet w sytuacji, gdy wartość `CurrentTemperature` w obiekcie `thermostat` się nie zmieniła. Tak więc klasa `Program` wysyła do wszystkich subskrybentów obiektu `thermostat` powiadomienie o zmianie temperatury, choć w rzeczywistości ta pozostaje taka sama. Podobnie jak we wcześniejszym opisie, problem z delegatami polega więc na braku wystarczającej hermetyzacji. Klasa `Thermostat` powinna uniemożliwić innym klasom wywoływanie delegata `OnTemperatureChange`.

Deklarowanie zdarzeń

Język C# udostępnia słowo kluczowe event, które rozwiązuje oba opisane problemy. Choć na pozór wygląda ono jak modyfikator pola, w rzeczywistości służy do tworzenia składowych nowego rodzaju (zobacz listing 14.13).

Listing 14.13. Używanie słowa kluczowego event w kodzie wzorca opartego na zdarzeniach

```
public class Thermostat
{
    public class TemperatureArgs: System.EventArgs
    {
        public TemperatureArgs(float newTemperature )
        {
            NewTemperature = newTemperature;
        }

        public float NewTemperature { get; set; }
    }

    // Definiowanie nadawcy zdarzeń.
    public event EventHandler<TemperatureArgs> OnTemperatureChange =
        delegate { };

    public float CurrentTemperature
    {
        ...
    }
    private float _CurrentTemperature;
}
```

2.0

W nowej klasie Thermostat wprowadzono cztery zmiany w porównaniu do pierwotnej wersji. Po pierwsze, usunięto właściwość OnTemperatureChange, a zamiast niej zadeklarowano pole publiczne o tej samej nazwie. Na pozór jest to sprzeczne z dążeniem do rozwiązania problemu braku hermetyzacji. W końcu celem jest zwiększenie hermetyzacji, a nie zmniejszanie jej przez tworzenie publicznego pola. Jednak druga zmiana polega na dodaniu słowa kluczowego event bezpośrednio przed deklaracją pola. Ta prosta modyfikacja zapewnia potrzebną hermetyzację. Dodanie słowa kluczowego event zapobiega użyciu operatora = w celu przypisania wartości do publicznego pola delegata (niedozwolona jest na przykład instrukcja thermostat.OnTemperatureChange = cooler.OnTemperatureChanged). Ponadto tylko klasa zawierająca dany delegat może go wywołać, by opublikować zdarzenie i przesłać je do wszystkich subskrybentów (nie można na przykład wywołać polecenia thermostat.OnTemperatureChange(42) poza klasą obejmującą ten delegat). Tak więc słowo kluczowe event zapewnia hermetyzację uniemożliwiającą zewnętrznym klasom publikowanie zdarzenia i usuwanie subskrypcji, których dana klasa nie dodała. To rozwiązuje dwa opisane wcześniej problemy dotyczące zwykłych delegatów. Zapewnienie tego rozwiązania było jedną z głównych przyczyn dodania słowa kluczowego event do języka C#.

Inną pułapką związaną ze zwykłymi delegatami jest wysokie ryzyko zapomnienia o sprawdzeniu, czy ich wartość jest różna od null (w wersji C# 6.0 najlepiej posługiwać się do tego operatorem ?.). Jeśli przed wywołaniem delegata zapomnisz o takim teście, może nieoczekiwane wystąpić wyjątek NullReferenceException. Na szczęście hermetyzacja zapewniana

przez słowo kluczowe event daje dodatkowe możliwości w miejscu deklaracji (lub w konstruktorze), co pokazano na listingu 14.13. Zauważ, że w deklaracji zdarzenia przypisywana jest do niego wartość delegate { }, czyli różny od null pusty delegat reprezentujący kolekcję z zerem odbiorców. Przypisanie pustego delegata pozwala zgłaszać zdarzenie bez sprawdzania, czy istnieją odbiorcy. Ta operacja przypomina przypisanie do zmiennej tablicy z zerem elementów. Dzięki temu można wywołać składową reprezentującą tablicę bez wcześniejszego sprawdzania, czy zmieniona jest różna od null. Oczywiście jeśli możliwe jest, że do delegata przypisana zostanie później wartość null, sprawdzanie tej wartości nadal jest konieczne. Jednak ponieważ słowo kluczowe event pozwala na przypisywanie wartości do delegata tylko w zawierającej go klasie, zmiany delegata mogą nastąpić tylko w jej kodzie. Jeśli wartość null nigdy nie jest przypisywana do delegata, przed jego wywołaniem nie trzeba sprawdzać, czy jest różny od null².

Konwencje programistyczne

2.0

Aby uzyskać pożądane rozwiązanie, wystarczy przekształcić pierwotną deklarację zmiennej delegata w pole i dodać słowo kluczowe event. Te dwie zmiany zapewniają potrzebną hermetyzację, a oprócz tego działanie kodu się nie zmienia. Jednak w kodzie na listingu 14.13 w deklaracji delegata pojawiła się też dodatkowa modyfikacja. Aby zachować zgodność ze standardowymi konwencjami programistycznymi dotyczącymi języka C#, należy zastąpić typ Action<float> nowym typem delegata — EventHandler<TemperatureArgs>. Jest to typ ze środowiska CLR. Deklarację tego typu znajdziesz na listingu 14.14.

Listing 14.14. Deklarowanie generycznego typu delegata

```
public delegate void EventHandler<TEventArgs>(
    object sender, TEventArgs e);
```

Zmiana typu skutkuje tym, że jeden określający temperaturę parametr z typu delegata Action<TEventArgs> zostaje zastąpiony dwoma nowymi parametrami. Jeden z nich reprezentuje nadawcę, a drugi — dane o zdarzeniu. Kompilator języka C# nie wymusza tej zmiany, jednak przekazywanie dwóch parametrów opisanego rodzaju jest standardem w delegatach używanych do zgłoszania zdarzeń.

Pierwszy parametr, sender, zawiera instancję klasy, która wywołała delegat. Jest to pomocne zwłaszcza w sytuacji, gdy ta sama metoda subskrybująca rejestruje chęć otrzymywania powiadomień o wielu zdarzeniach — na przykład wtedy, gdy metoda heater.OnTemperatureChanged subskrybuje powiadomienia od dwóch instancji typu Thermostat. W takiej sytuacji obie instancje klasy Thermostat mogą wywołać metodę heater.OnTemperatureChanged. Aby ustalić, która instancja klasy Thermostat zgłosiła zdarzenie, należy wykorzystać parametr sender z wywołania metody Heater.OnTemperatureChanged(). Jeśli zdarzenie jest statyczne, nie da się przekazać instancji. Wtedy jako wartość parametru sender należy podać null.

² Ten wzorzec nie zadziała, jeśli zdarzenie znajduje się w strukturze (choć jest to rzadko stosowana technika).

Drugi parametr, `TEventArgs e`, jest tu podawany jako wartość typu `Thermostat.TemperatureArgs`. Ważnym aspektem typu `TemperatureArgs` (przynajmniej w kontekście konwencji programistycznych) jest to, że dziedziczy on po typie `System.EventArgs`. Do wersji 4.5 platforma .NET wymuszała dziedziczenie po tym typie za pomocą ograniczenia w typie generycznym. Jedyną ważną właściwością typu `System.EventArgs` jest właściwość `Empty`. Służy ona do określania, że nie istnieją żadne dane powiązane ze zdarzeniem. Jednak tu pochodny od typu `System.EventArgs` typ `TemperatureArgs` obejmuje dodatkową właściwość `NewTemperature`. Pozwala ona przekazać temperaturę z termostatu do subskrybentów.

Oto podsumowanie konwencji programistycznych dotyczących zdarzeń — pierwszy argument, `sender`, jest typu `object` i zawiera referencję do obiektu, który wywołał dany delegat (jeśli zdarzenie jest statyczne, zamiast referencji używana jest wartość `null`). Drugi argument jest typu `System.EventArgs` lub pochodnego od niego typu, który zawiera dodatkowe dane na temat zdarzenia. Delegat może być wywoływany w prawie taki sam sposób jak wcześniej (różnica polega na użyciu dodatkowych parametrów). Przykładowy kod przedstawiono na listingu 14.15.

Listing 14.15. Zgłaszanie powiadomień o zdarzeniu

```
public class Thermostat
{
    ...
    public float CurrentTemperature
    {
        get { return _CurrentTemperature; }
        set
        {
            if (value != CurrentTemperature)
            {
                CurrentTemperature = value;
                // Jeśli istnieją subskrybenci,
                // należy ich powiadomić o zmianie temperatury
                // za pomocą skierowanego do nich wywołania.
                OnTemperatureChange?.Invoke( // W wersji C# 6.0.
                    this, new TemperatureArgs(value));
            }
        }
    }
    private float _CurrentTemperature;
}
```

Jako nadawcę zwykle podaje się (za pomocą słowa `this`) obiekt klasy zawierającej zdarzenie, ponieważ jest to jedyna klasa, która może wywoływać delegat powiązany ze zdarzeniem.

W tym przykładzie subskrybent może zrzutować wartość parametru `sender` na typ `Thermostat` i uzyskać w ten sposób dostęp do bieżącej temperatury. Temperaturę można też określić za pomocą instancji klasy `TemperatureArgs`. Jednak temperatura zapisana w instancji typu `Thermostat` może zostać zmodyfikowana przez inny wątek. Do zdarzeń zgłaszanych w wyniku zmiany stanu często stosowany jest wzorzec, który polega na przekazaniu nowej i poprzedniej wartości. Pozwala to sprawdzać, czy zmiana stanu, jaka nastąpiła, jest dozwolona.

Wskazówki

SPRAWDZAJ, czy wartość delegata jest różna od null; dopiero potem go wywołuj (w wersji C# 6.0 można wykorzystać do tego operator ?.).

PRZEKAZUJ instancję klasy jako wartość nadawcy w zdarzeniach niestatycznych.

PRZEKAZUJ wartość null jako nadawcę w zdarzeniach statycznych.

NIE przekazuj wartości null jako wartości argumentu eventArgs.

STOSUJ typ System.EventArgs (lub typ pochodny od niego) dla parametru TEventArgs.

ROZWAŻ stosowanie podklasy klasy System.EventArgs jako typu argumentu zdarzenia (TEventArgs), chyba że jesteś przekonany, że w zdarzeniu nigdy nie będą potrzebne żadne dane.

2.0

Typy generyczne a delegaty

W poprzednim podrozdziale wyjaśniono, że w trakcie definiowania typu zdarzenia należy wykorzystać typ delegata EventHandler<TEventArgs>. Teoretycznie można zastosować dowolny typ delegata, ale zwyczajowo pierwszy parametr, sender, jest typu object, a drugi parametr, e, powinien być typu pochodnego od System.EventArgs. Jedną z niewygodnych cech delegatów w wersji C# 1.0 było to, że trzeba było zadeklarować nowy typ delegata za każdym razem, gdy zmieniły się parametry w metodzie obsługującej zdarzenie. Utworzenie każdego nowego typu pochodnego od System.EventArgs (co zdarza się dość często) wymagało zadeklarowania nowego typu delegata, korzystającego z nowego typu pochodnego od EventArgs. Na przykład aby wykorzystać typ TemperatureArgs w kodzie zgłaszającym powiadomienia o zdarzeniach na listingu 14.15, konieczne byłoby zadeklarowanie typu delegata TemperatureChangeHandler z parametrem typu TemperatureArgs (zobacz listing 14.16).

Listing 14.16. Używanie niestandardowego typu delegata

```
public class Thermostat
{
    public class TemperatureArgs : System.EventArgs
    {
        public TemperatureArgs( float newTemperature )
        {
            NewTemperature = newTemperature;
        }

        public float NewTemperature { get; set; }
    }

    public delegate void TemperatureChangeHandler(
        object sender, TemperatureArgs newTemperature);

    public event TemperatureChangeHandler?
        OnTemperatureChange;

    public float CurrentTemperature
    {
```

```

    ...
}

private float _CurrentTemperature;
}

```

Choć zwykle zalecane jest używanie typu `EventHandler<EventArgs>` zamiast tworzenia niestandardowego typu delegata, takiego jak `TemperatureChangeHandler`, niestandardowy typ też ma pewną zaletę — pozwala utworzyć parametry o nazwach dostosowanych do zdarzenia. Na listingu 14.16 w wywołaniu delegata (czego celem jest zgłoszenie zdarzenia) nazwa drugiego parametru to `newTemperature` zamiast `e`.

Inny powód, dla którego może być używany niestandardowy typ delegata, związany jest z fragmentami interfejsu API środowiska CLR opracowanymi przed wersją C# 2.0. Ponieważ te fragmenty stanowią istotną część wielu często używanych typów platformy, w zdarzeniach we wspomnianym interfejsie API w wielu miejscach używane są konkretne typy delegatów zamiast typów generycznych. Mimo to w większości sytuacji, gdy w C# 2.0 (i nowszych wersjach) używane są zdarzenia, nie trzeba deklarować niestandardowych typów delegatów.

2.0

Wskazówka

STOSUJ typ `System.EventHandler<T>`, zamiast ręcznie tworzyć nowe typy delegatów na potrzeby obsługi zdarzeń. Wyjątkiem jest sytuacja, gdy nazwy parametrów w niestandardowym typie pomagają jednoznacznie opisać ich znaczenie.

ZAGADNIENIE DLA ZAAWANSOWANYCH

Wewnętrzne mechanizmy zdarzeń

Wcześniej wspomniano już, że zdarzenia pozwalają zewnętrznym klasom wyłącznie na dodawanie do nadawcy metod subskrybujących zdarzenia (za pomocą operatora `+=`) i rezygnowanie z subskrypcji (za pomocą operatora `-=`). Ponadto zgłaszanie zdarzenia jest możliwe tylko w zawierającej je klasie. Kompilator języka C# przyjmuje publiczną zmienną delegata z modyfikatorem `event` i deklaruje ten delegat jako prywatny. Poza tym dodaje do niego kilka metod i dwa specjalne bloki. Słowo kluczowe `event` jest więc dostępnym w języku C# skrótem pozwalającym wygenerować kod zapewniający hermetyzację. Przyjrzyj się przykładowej deklaracji zdarzenia z listingu 14.17.

Listing 14.17. Deklaracja zdarzenia `OnTemperatureChange`

```

public class Thermostat
{
    public event EventHandler<TemperatureArgs>? OnTemperatureChange;

    ...
}

```

Gdy kompilator języka C# natrafi na słowo kluczowe `event`, generuje kod CIL będący odpowiednikiem kodu w języku C# przedstawionego na listingu 14.18.

Listing 14.18. Kod w języku C# będący odpowiednikiem kodu CIL wygenerowanego przez kompilator na podstawie zdarzenia

```
public class Thermostat
{
    // ...
    // Deklarowanie pola typu delegata w celu
    // zapisania listy subskrybentów.
    private EventHandler<TemperatureArgs> _OnTemperatureChange;

    public void add_OnTemperatureChange(
        EventHandler<TemperatureArgs> handler)
    {
        System.Delegate.Combine(_OnTemperatureChange, handler);
    }

    public void remove_OnTemperatureChange(
        EventHandler<TemperatureArgs> handler)
    {
        System.Delegate.Remove(_OnTemperatureChange, handler);
    }

    public event EventHandler<TemperatureArgs> OnTemperatureChange
    {
        add
        {
            add_OnTemperatureChange(value)
        }
        remove
        {
            remove_OnTemperatureChange(value)
        }
    }
}
```

2.0

Kod z listingu 14.17 jest więc (w przybliżeniu) dostępnym w języku C# skrótem, na podstawie którego kompilator generuje odpowiednik dłuższego kodu przedstawionego na listingu 14.18. Dookreślenie „w przybliżeniu” jest konieczne, ponieważ aby opis był bardziej przejrzysty, pominięto pewne szczegóły związane z synchronizacją wątków.

Kompilator języka C# najpierw bierze pierwotną definicję zdarzenia i definiuje zamiast niej prywatną zmienną delegata. Dlatego delegat staje się niedostępny dla zewnętrznych klas (nawet klas pochodnych od tej, która zawiera ten delegat).

Następnie kompilator definiuje dwie metody, `add_OnTemperatureChange()` i `remove_OnTemperatureChange()`. Przyrostek `OnTemperatureChange` pochodzi tu od pierwotnej nazwy zdarzenia. Te metody określają działanie operatorów `+=` i `-=`. Na listingu 14.18 pokazano, że te metody są zaimplementowane z wykorzystaniem statycznych metod `System.Delegate.Combine()` i `System.Delegate.Remove()` opisanych we wcześniejszej części rozdziału. Pierwszym parametrem przekazywanym do każdej z tych metod jest `_OnTemperatureChange` — prywatna instancja typu delegata `EventHandler<TemperatureArgs>`.

Prawdopodobnie najciekawszą częścią kodu generowanego na podstawie słowa kluczo-wego event jest ostatni fragment listingu. Składnia wygląda tu bardzo podobnie jak w gette-rach i setterach właściwości. Blok add odpowiada za obsługę operatora += zdarzeń, przekazując wywołanie do metody `add_OnTemperatureChange()`. Podobnie blok remove obsługuje operator -=, przekazując wywołanie do metody `remove_OnTemperatureChange()`.

Zwróć uwagę na podobieństwa między tym kodem a kodem generowanym dla właściwości. Warto przypomnieć, że w języku C# na potrzeby właściwości tworzone są metody `get_<nazwa_właściwości>` i `set_<nazwa_właściwości>`, a następnie wywołania tych metod są przekazywane do bloków get i set tych metod. Nawet składnia w obu sytuacjach wygląda bardzo podobnie.

Inną ważną cechą wygenerowanego kodu CIL jest to, że występuje w nim odpowiednik słowa kluczowego event. Oznacza to, że zdarzenia są jawnie określone w kodzie CIL. Nie jest to konstrukcja występująca tylko w języku C#. Dzięki temu, że w kodzie CIL pojawia się odpowiednik słowa kluczowego event, wszystkie języki i edytory mogą udostępniać specjalne funkcje, ponieważ wykrywają zdarzenie jako specjalną składową klasy.

Modyfikowanie implementacji zdarzeń

Możesz zmodyfikować kod operatorów += i -= generowany przez kompilator. Pomyśl na przykład o zmianie zasięgu delegata `OnTemperatureChange` z prywatnego na chroniony. To oczywiście umożliwia klasom pochodnym od `Thermostat` bezpośredni dostęp do delegata. Klasy pochodne nie podlegają wtedy tym samym ograniczeniom co klasy zewnętrzne. Aby umożliwić zastosowanie tego rozwiązania, w języku C# można je dodać za pomocą zaprezentowanej na listingu 14.16 składni właściwości. Oznacza to, że C# pozwala zdefiniować niestandardowe bloki `add` i `remove`, w których można umieścić niestandardową implementację każdego aspektu hermetyzacji związanego ze zdarzeniami. Przykładowy kod tego rodzaju znajdziesz na listingu 14.19.

Listing 14.19. Niestandardowe metody obsługi zdarzeń — add i remove

```
public class Thermostat
{
    public class TemperatureArgs : System.EventArgs
    {
        ...
    }

    // Definicja nadawcy zdarzeń.
    public event EventHandler<TemperatureArgs> OnTemperatureChange
    {
        add
        {
            _OnTemperatureChange = (TemperatureChangeHandler)
                System.Delegate.Combine(value, _OnTemperatureChange);
        }
        remove
        {
            _OnTemperatureChange = (TemperatureChangeHandler?)
                System.Delegate.Remove(_OnTemperatureChange, value);
        }
    }
}
```

```
        }  
    }  
protected EventHandler<TemperatureArgs>? _OnTemperatureChange;  
  
public float CurrentTemperature  
{  
    ...  
}  
private float _CurrentTemperature;  
}
```

Tu delegat przechowujący subskrybentów, `_OnTemperatureChange`, ma poziom dostępu zmieniony na `protected`. Ponadto w implementacji bloku `add` przedstawiono parametry, dlatego ostatni delegat dodawany do łańcucha jest pierwszym delegatem otrzymującym powiadomienie. Kod nie powinien jednak być zależny od tej implementacji.

Podsumowanie

Po omówieniu zdarzeń warto wspomnieć, że zwykle referencje do metod są jedynym miejscem, w którym warto korzystać ze zmiennej delegata bez posługiwania się zdarzeniami. Ponieważ zdarzenia zapewniają dodatkową hermetyzację i umożliwiają (gdy jest to potrzebne) modyfikację implementacji, najlepszym rozwiązaniem jest tworzenie wzorca `publikuj-subskrybuj` zawsze za pomocą zdarzeń.

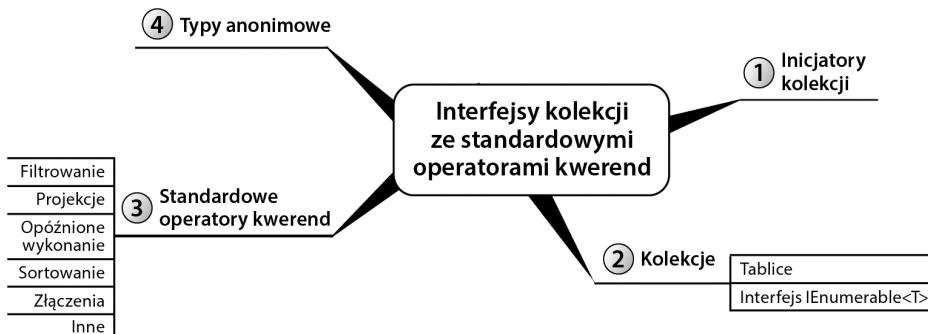
Możliwe, że będziesz potrzebował nieco praktyki, zanim nauczysz się pisać zdarzenia samodzielnie, bez zaglądania do przykładowego kodu. Jednak zdarzenia są niezbędną podstawą do pisania asynchronicznego wielowątkowego kodu omawianego w dalszych rozdziałach.

■ 15 ■

Interfejsy kolekcji ze standardowymi operatorami kwerend

Najważniejsze mechanizmy dodane w wersji C# 3.0 dotyczą kolekcji. Mechanizmy te związane są z interfejsem API LINQ (ang. *Language Integrated Query*). Metody rozszerzające i wyrażenia lambda sprawiają, że LINQ jest znacznie lepszym interfejsem API przeznaczonym do pracy z kolekcjami. We wcześniejszych wydaniach książki rozdział o kolekcjach znajdował się bezpośrednio po rozdziale poświęconym typom generycznym, ale przed rozdziałem dotyczącym delegatów. Jednak wyrażenia lambda miały tak duży wpływ na interfejsy API kolekcji, że nie da się już opisywać kolekcji bez wcześniejszego omówienia delegatów (które są podstawą wyrażeń lambda). Teraz, gdy dzięki poprzedniemu rozdziałowi masz już solidne podstawy z zakresu wyrażeń lambda, możesz przejść do szczegółowego zapoznania się z kolekcjami. Temu zagadnieniu poświęcone są trzy rozdziały. W tym rozdziale zostały omówione **standardowe operatory kwerend**. Pozwalają one wykorzystać interfejs LINQ dzięki bezpośredniemu wywoływaniu metod rozszerzających.

Na początku rozdziału znajduje się wprowadzenie inicjatorów kolekcji. Dalej w rozdziale omówiono różne interfejsy kolekcji, a także wyjaśniono, jak są one ze sobą powiązane. To zapewni podstawy potrzebne do zrozumienia kolekcji, dlatego powinieneś starannie opanować ten materiał. W części dotyczącej interfejsów kolekcji zaprezentowano metody rozszerzające interfejs `IEnumerable<T>`, dodane w wersji C# 3.0, by zaimplementować standardowe operatory kwerend.



Są dwie kategorie klas i interfejsów związanych z kolekcjami. Jedna grupa klas i interfejsów obsługuje typy generyczne, a druga tego nie robi. Ten rozdział dotyczy głównie generycznych interfejsów kolekcji. Z klas kolekcji nieobsługujących typów generycznych powinienieś korzystać tylko wtedy, gdy tworzysz komponenty, które muszą współpracować ze starszymi wersjami środowiska uruchomieniowego. Jest tak, ponieważ wszystkie kolekcje dostępne w formie niegenerycznej mają obecnie generyczne odpowiedniki ze ścisłą kontrolą typów. Choć opisane tu zagadnienia dotyczą obu postaci kolekcji, nie omawiamy tu bezpośrednio wersji niegenerycznych¹.

Rozdział kończy się szczegółowym omówieniem typów anonimowych. Zagadnienie to zostało opisane tylko skrótnie w kilku sekcjach „Zagadnienie dla zaawansowanych” w rozdziale 3. Ciekawą rzeczą związaną z typami anonimowymi jest to, że obecnie częściej zamiat nich używa się wprowadzonych w C# 7.0 krotek. Temat ten omówiono w końcowej części rozdziału.

Iinicjatory kolekcji

Iinicjatory kolekcji pozwalają programistom tworzyć kolekcje z początkowym zbiorem elementów określonym w czasie tworzenia instancji (podobnie jak w deklaracji tablicy). Przed wprowadzeniem inicjatorów kolekcji elementy trzeba było dodawać jawnie po utworzeniu instancji. Służyły do tego metody takie jak `Add()` z interfejsu `System.Collections.Generic.ICollection<T>`. Gdy korzystasz z inicjatora kolekcji, wywołania metody `Add()` są generowane przez kompilator języka C#, a nie jawnie podawane przez programistę. Na listingu 15.1 pokazano, jak zainicjować kolekcję za pomocą inicjatora.

Listing 15.1. Inicjowanie kolekcji

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

class Program
{
    static void Main()
    {
        List<string> sevenWorldBlunders;
        sevenWorldBlunders = new List<string>()
        {
            // Cytaty Gandhiego.
            "Bogactwo bez pracy",
            "Przyjemność bez sumienia",
            "Wiedza bez osobowości",
            "Handel bez moralności",
            "Nauka bez człowieczeństwa",
            "Wiara bez poświęcenia",
            "Polityka bez zasad"
        };

        Print(sevenWorldBlunders);
    }
}
```

¹ W specyfikacji .NET Standard i środowisku .NET Core kolekcje niegeneryczne nie występują.

```
private static void Print<T>(IEnumerable<T> items)
{
    foreach (T item in items)
    {
        Console.WriteLine(item);
    }
}
```

Ta składnia jest podobna nie tylko do składni inicjowania tablic, ale też do inicjatora obiektów, gdzie po konstruktorze znajduje się nawias klamrowy. Jeśli do konstruktora nie są przekazywane parametry, nawias po nazwie typu danych jest opcjonalny (podobnie jak w inicjatorze obiektu).

Aby inicjator kolekcji można było poprawnie skompilować, spełnionych musi być kilka podstawowych wymagań. Najlepiej jest, gdy typ, dla którego używany jest inicjator kolekcji, zawiera implementację interfejsu `System.Collections.Generic.ICollection<T>`. To daje gwarancję, że kolekcja zawiera metodę `Add()`, którą może wywołać wygenerowany przez kompilator kod. Dostępna jest też mniej restrykcyjna wersja tego wymagania, zgodnie z którą konieczna jest tylko dostępność jednej lub kilku metod `Add()` — albo w postaci metod rozszerzających (w wersji C# 6.0), albo w postaci metody instancji typu z implementacją interfejsu `IEnumerable<T>` (nawet jeśli dany typ nie zawiera implementacji interfejsu `ICollection<T>`). Dostępne metody `Add()` muszą przyjmować parametry zgodne z wartościami podanymi w inicjatorze kolekcji.

W słownikach składnia inicjatora kolekcji jest bardziej skomplikowana, ponieważ każdy element słownika wymaga zarówno klucza, jak i wartości. Tę składnię pokazano na listingu 15.2.

Listing 15.2. Inicjowanie słownika za pomocą inicjatora

```
using System;
using System.Collections.Generic;
#if !PRECSHARP6
    // Wersja C# 6.0 lub nowsza.
    Dictionary<string, ConsoleColor> colorMap =
        new Dictionary<string, ConsoleColor>
    {
        ["Error"] = ConsoleColor.Red,
        ["Warning"] = ConsoleColor.Yellow,
        ["Information"] = ConsoleColor.Green,
        ["Verbose"] = ConsoleColor.White
    };
#else
    // Wersje starsze niż C# 6.0.
    Dictionary<string, ConsoleColor> colorMap =
        new Dictionary<string, ConsoleColor>
    {
        { "Error", ConsoleColor.Red },
        { "Warning", ConsoleColor.Yellow },
        { "Information", ConsoleColor.Green },
        { "Verbose", ConsoleColor.White}
    };
#endif
```

Na listingu 15.2 pokazano dwie różne wersje procesu inicjowania. W pierwszej wykorzystano nową składnię wprowadzoną w wersji C# 6.0. Pary nazwa-wartość są tu tworzone za pomocą operatora przypisania, określającego, jakie wartości mają być powiązane z poszczególnymi kluczami. Druga składnia (nadal dostępna w wersji C# 6.0 i nowszych) polega na łączeniu nazw z wartościami w nawiasie klamrowym.

Umożliwienie stosowania inicjatorów do kolekcji, które nie zawierają implementacji interfejsu `ICollection<T>`, było ważne z dwóch powodów. Po pierwsze, większość kolekcji (typów z implementacją interfejsu `IEnumerable<T>`) nie zawiera implementacji interfejsu `ICollection<T>`, co znacznie zmniejszałoby przydatność inicjatorów kolekcji. Po drugie, dopasowywanie wyrażeń z inicjatora kolekcji do metod na podstawie nazw i sygnatur pozwala na większą różnorodność elementów inicjowanych w kolekcji. Inicjator może teraz na przykład przetworzyć wyrażenie `new DataStore() { a, {b, c}}`, o ile występuje jedna metoda `Add()` o sygnaturze zgodnej z parametrem `a` i druga metoda `Add()` o sygnaturze zgodnej z parą `b, c`.

Interfejs `IEnumerable<T>` sprawia, że klasa staje się kolekcją

Kolekcja w platformie .NET z definicji jest klasą, która zawiera przynajmniej implementację interfejsu `IEnumerable`. Ten interfejs jest kluczowy, ponieważ implementacja metod interfejsu `IEnumerable` to minimum niezbędne do tego, by możliwa była iteracja po kolekcji.

W rozdziale 4. pokazano, jak zastosować instrukcję `foreach` do iterowania po elementach tablicy. Składnia oparta na tej instrukcji jest prosta i pozwala uniknąć komplikacji związanych z ustalaniem, ile elementów znajduje się w kolekcji. Jednak środowisko uruchomieniowe nie obsługuje bezpośrednio instrukcji `foreach`. To kompilator języka C# przekształca kod w sposób opisany w tym podrozdziale.

Instrukcja `foreach` dla tablic

Na listingu 15.3 pokazano prostą pętlę `foreach` służącą do iterowania po tablicy liczb całkowitych i wyświetlania każdej z tych liczb w konsoli.

Listing 15.3. Instrukcja `foreach` dla tablic

```
int[] array = new int[]{1, 2, 3, 4, 5, 6};

foreach (int item in array)
{
    Console.WriteLine(item);
}
```

Na podstawie tego kodu kompilator języka C# tworzy w kodzie CIL odpowiednik pętli `for` pokazanej na listingu 15.4.

Listing 15.4. Kod odpowiadający skompilowanej instrukcji `foreach` dla tablic

```
int[] tempArray;
int[] array = new int[]{1, 2, 3, 4, 5, 6};

tempArray = array;
for (int counter = 0; (counter < tempArray.Length); counter++)
{
    int item = tempArray[counter];

    Console.WriteLine(item);
}
```

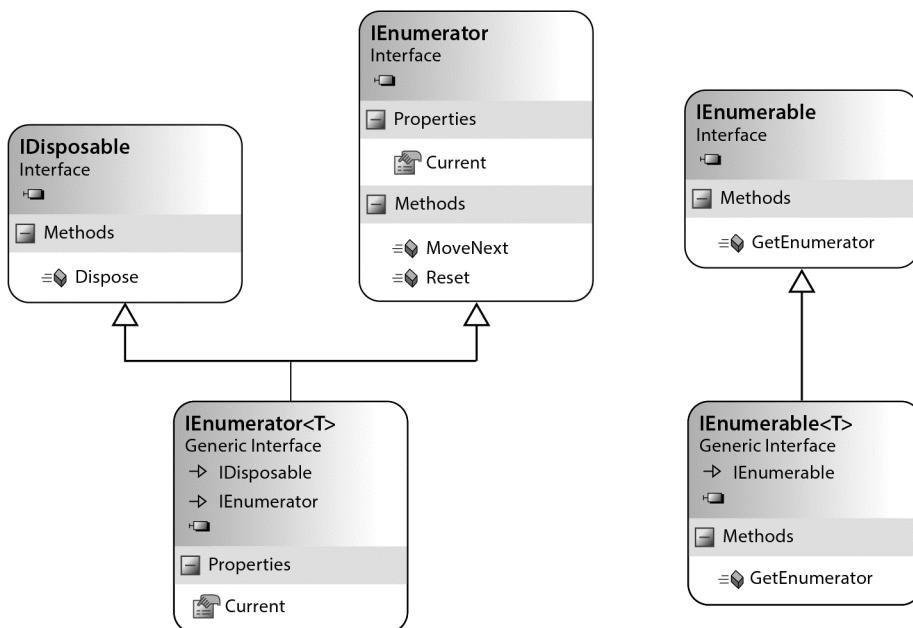
Zauważ, że w tym przykładzie instrukcja `foreach` wymaga wykorzystania właściwości `Length` i operatora indeksowania (`[]`). Dzięki właściwości `Length` kompilator języka C# może wykorzystać instrukcję `for` do iteracyjnego pobrania każdego elementu tablicy.

Instrukcja `foreach` dla interfejsu `IEnumerable<T>`

Choć kod z listingu 15.4 działa prawidłowo dla tablic, gdzie długość jest stała i operator indeksowania zawsze jest dostępny, nie wszystkie typy kolekcji mają ustaloną liczbę elementów. Ponadto wiele klas kolekcji, w tym `Stack<T>`, `Queue<T>` i `Dictionary< TKey, TValue >`, nie obsługuje pobierania elementów za pomocą indeksu. Dlatego potrzebny jest bardziej ogólny mechanizm iterowania po kolekcjach. Zapewnia go wzorzec „iterotor”. Jeśli można określić pierwszy, następny i ostatni element, to znajomość liczby elementów i możliwość pobierania ich za pomocą indeksu nie są potrzebne.

Interfejs `System.Collections.Generic.IEnumerator<T>` i niegeneryczny interfejs `System.Collections_IEnumerator` są zaprojektowane w taki sposób, by umożliwić iterowanie po elementach kolekcji za pomocą wzorca „iterotor”. Nie trzeba wtedy korzystać z wzorca opartego na długości kolekcji i indeksach, zastosowanego na listingu 15.4. Diagram klas ilustrujący relacje między używanymi w tych wzorcach interfejsami przedstawiono na rysunku 15.1.

Interfejs `IEnumerator`, po którym dziedziczy `IEnumerator<T>`, zawiera trzy składowe. Pierwsza z nich to metoda `bool MoveNext()`. Za pomocą tej metody można przejść od jednego elementu kolekcji do następnego, przy czym obiekt sprawdza, czy metoda dotarła do wszystkich elementów. Druga składowa to przeznaczona tylko do odczytu właściwość `Current`, zwierająca obecnie przetwarzany element. W interfejsie `IEnumerator<T>` składowa `Current` jest przeciążana i pozwala utworzyć wersję specyfczną dla typu. Za pomocą dwóch wymienionych składowych klas kolekcji można przeprowadzić iterację po kolekcji przy użyciu pętli `while`, co pokazano na listingu 15.5. Metoda `Reset()` zwykle zgłasza wyjątek typu `NotSupportedException`, dlatego nigdy nie należy jej wywoływać. Jeśli chcesz zacząć od początku iterację po kolekcji, utwórz nowy enumerator.



Rysunek 15.1. Diagram klas przedstawiający interfejsy **IEnumarator<T>** i **IEnumarator**

Listing 15.5. Iterowanie po kolekcji za pomocą pętli while

```

System.Collections.Generic.Stack<int> stack =
    new System.Collections.Generic.Stack<int>();
int number;
// ...

// Jest to kod „teoretyczny”, a nie używany w praktyce.
while (stack.MoveNext())
{
    number = stack.Current;
    Console.WriteLine(number);
}
  
```

Na listingu 15.5 metoda **MoveNext()** zwraca wartość **false**, gdy wykracza poza koniec kolekcji. Dzięki temu nie trzeba zliczać elementów w pętli.

Na listingu 15.5 typ kolekcji to **System.Collections.Generic.Stack<T>**. Istnieje też wiele innych typów kolekcji. Ważnym aspektem typu **Stack<T>** jest to, że działa on zgodnie z zasadą „pierwszy na wejściu, pierwszy na wyjściu” (ang. *last in, first out* — LIFO). Zauważ, że określający typ parametr **T** wyznacza typ wszystkich elementów przechowywanych w kolekcji. Cechą charakterystyczną kolekcji generycznych jest możliwość przechowywania w nich obiektów jednego konkretnego typu. Programista musi znać ten typ, gdy dodaje i usuwa elementy kolekcji oraz uzyskuje do nich dostęp.

W poprzednim przykładzie pokazano istotę kodu generowanego przez kompilator języka C#, jednak w rzeczywistości skompilowany kod wygląda nieco inaczej; w przykładzie pominięto dwa ważne szczegóły — przeplatanie się instrukcji i usług błędów.

Współdzielony stan

Problem z kodem pokazanym na listingu 15.5 polega na tym, że jeśli dwie podobne pętle przeplatają się (jedna pętla `foreach` działa wewnątrz drugiej i obie przetwarzają tę samą kolekcję), kolekcja musi udostępniać wskaźnik stanu określający bieżący element. Pozwala to ustalić następny element w momencie wywołania metody `MoveNext()`. W takiej sytuacji działanie jednej pętli może wpływać na pracę drugiej. To samo dotyczy pętli uruchamianych przez różne wątki.

Aby rozwiązać ten problem, klasy kolekcji nie obsługują bezpośrednio interfejsów `IEnumerator<T>` i `IEnumerator`. Na rysunku 15.1 pokazano, że istnieje inny interfejs, `IEnumerable<T>`, którego jedną metodą jest `GetEnumerator()`. Ta metoda ma zwracać obiekt z implementacją interfejsu `IEnumerator<T>`. Dlatego klasa kolekcji nie musi zarządzać stanem. Można wykorzystać inną klasę (zwykle jest to klasa zagnieżdzona mająca dostęp do wewnętrznych mechanizmów kolekcji), która zawiera implementację interfejsu `IEnumerator<T>` i zarządza stanem pętli odpowiedzialnej za iterowanie. Enumerator działa jak „kursor” lub „zakładka” w sekwencji wartości. Możesz utworzyć wiele zakładek, a każdą z nich można przesuwać niezależnie w trakcie poruszania się po kolekcji. Napisany w języku C# oparty na tym wzorcu odpowiednik pętli `foreach` wygląda tak jak kod z listingu 15.6.

Listing 15.6. Odrębny enumerator przechowujący stan w trakcie iteracji

```
System.Collections.Generic.Stack<int> stack =
    new System.Collections.Generic.Stack<int>();
int number;
System.Collections.Generic.Stack<int>.Enumerator
    enumerator;
// ...
// Jeśli interfejs IEnumerable<T> jest zaimplementowany jawnie,
// potrzebne jest rzutowanie.
// ((IEnumerable<int>)stack).GetEnumerator();
enumerator = stack.GetEnumerator();
while (enumerator.MoveNext())
{
    number = enumerator.Current;
    Console.WriteLine(number);
}
```

ZAGADNIENIE DLA POCZĄTKUJĄCYCH I ZAAWANSOWANYCH

Porządkowanie zasobów po iteracji

Ponieważ klasa z implementacją interfejsu `IEnumerator<T>` przechowuje stan, czasem po zakończeniu pracy pętli (w wyniku ukończenia wszystkich iteracji lub zgłoszenia wyjątku) trzeba zwolnić związane ze stanem zasoby. Aby było to możliwe, interfejs `IEnumerator<T>` dziedziczy po interfejsie `IDisposable`. Enumeratory z implementacją interfejsu `IEnumerator` nie muszą zawierać implementacji interfejsu `IDisposable`, natomiast jeśli już implementują ten ostatni, dostępna jest metoda `Dispose()`. To pozwala wywołać metodę `Dispose()` po wyjściu z pętli `foreach`. Dlatego kod w języku C# odpowiadający ostatecznej wersji kodu CIL wygląda tak jak na listingu 15.7.

Listing 15.7. Kod odpowiadający skompilowanemu kodowi pętli foreach dla kolekcji

```
System.Collections.Generic.Stack<int> stack =
    new System.Collections.Generic.Stack<int>();
System.Collections.Generic.Stack<int>.Enumerator
    enumerator;
IDisposable disposable;
enumerator = stack.GetEnumerator();
try
{
    int number;
    while (enumerator.MoveNext())
    {
        number = enumerator.Current;
        Console.WriteLine(number);
    }
}
finally
{
    // Jawne rzutowanie obiektu typu IEnumarator<T>.
    disposable = (IDisposable) enumerator;
    disposable.Dispose();
    // Dla obiektu typu IEnumarator należy zastosować operator as, chyba że
    // już w czasie komplikacji wiadomo, że obiekt obsługuje interfejs IDisposable.
    // disposable = (enumerator as IDisposable);
    // if (disposable != null)
    // {
    //     disposable.Dispose();
    // }
}
```

Zauważ, że dzięki temu, iż interfejs IDisposable jest obsługiwany w interfejsie IEnumerator<T>, kod z listingu 15.7 można uprościć za pomocą instrukcji using. Nową wersję pokazano na listingu 15.8.

Listing 15.8. Obsługa błędów i porządkowanie zasobów za pomocą instrukcji using

```
System.Collections.Generic.Stack<int> stack =
    new System.Collections.Generic.Stack<int>();
int number;

using(
    System.Collections.Generic.Stack<int>.Enumerator
    enumerator = stack.GetEnumerator())
{
    while (enumerator.MoveNext())
    {
        number = enumerator.Current;
        Console.WriteLine(number);
    }
}
```

Pamiętaj jednak, że kod CIL nie obsługuje bezpośrednio słowa kluczowego using. Dlatego kod z listingu 15.7 jest bardziej precyzyjną reprezentacją kodu w języku C#, będącego odpowiednikiem kodu CIL wygenerowanego na podstawie pętli foreach.

ZAGADNIENIE DLA ZAAWANSOWANYCH

Pętla foreach bez interfejsu IEnumerable

Język C# nie wymaga implementacji interfejsu `IEnumerable` (lub `IEnumerable<T>`), aby możliwe było iterowanie po elementach za pomocą pętli `foreach`. Kompilator do obsługi iterowania wykorzystuje technikę **duck typing** — szuka metod `GetEnumerator()` (zwieraczącej wartość typu obejmującego właściwość `Current`) i `MoveNext()`. Technika duck typing polega na wyszukiwaniu metod na podstawie nazw. Nie wymaga korzystania z interfejsu ani jawnych wywołań metod. Nazwa *duck typing* (dosłownie „kaczka typowanie”) pochodzi od żartobliwego założenia, że jeśli obiekt ma być traktowany jako kaczka, wystarczy, że będzie udostępniał metodę `Kwak()`; nie musi zawierać implementacji interfejsu `IKaczka`. Jeżeli kompilator za pomocą techniki duck typing nie znajdzie odpowiedniej implementacji wzorca iteracji, sprawdza, czy kolekcja zawiera implementację określonych interfejsów.

Nie modyfikuj kolekcji w trakcie iterowania z wykorzystaniem pętli foreach

W rozdziale 4. była mowa o tym, że kompilator uniemożliwia przypisywanie wartości do zmiennej pętli `foreach` (tam była to zmienna `number`). Na listingu 15.7 pokazano, że przyznanie wartości do zmiennej `number` nie powodowałoby modyfikacji elementu kolekcji, dlatego kompilator języka C# zapobiega takiemu przypisaniu.

W pętli `foreach` nie można zwykle zmieniać liczbę elementów kolekcji ani wartości tych elementów. Jeśli w pętli `foreach` wywołasz na przykład instrukcję `stack.Push(42)`, nie będzie wiadomo, czy iterator powinien zignorować, czy uwzględnić zmianę w kolekcji `stack` — czy iterator ma uwzględnić w trakcie iteracji nowo dodany element, czy pominąć go i założyć, że obowiązuje stan z momentu utworzenia iteratora.

Z powodu tej niejednoznaczności próba dostępu do enumeratatora po zmodyfikowaniu kolekcji w pętli `foreach` skutkuje zwykle zgłoszeniem wyjątku typu `System.InvalidOperationException`. Jest to informacja, że kolekcję zmodyfikowano po utworzeniu enumeratatora.

Standardowe operatory kwerend

Każdy typ z implementacją interfejsu `IEnumerable<T>` musi zawierać implementację jednej metody (oprócz metod typu `System.Object`) — `GetEnumerator()`. To daje jednak ponad 50 metod dostępnych we wszystkich typach z implementacją interfejsu `IEnumerable<T>`, i to bez uwzględniania przeciążonych wersji metod. Dzieje się tak bez konieczności jawnego implementowania jakichkolwiek metod innych niż `GetEnumerator()`. Dodatkowe mechanizmy są udostępniane za pomocą metod rozszerzających (wprowadzonych w wersji C# 3.0) i znajdują się w klasie `System.Linq.Enumerable`. Dlatego by uzyskać dostęp do tych metod, wystarczy dodać deklarację `using System.Linq`.

Każda metoda z interfejsu `IEnumerable<T>` to **standardowy operator kwerend**. Te metody umożliwiają kierowanie kwerend do kolekcji, na których działają. W dalszych podrozdziałach poznasz najważniejsze z tych operatorów. W licznych przykładach wykorzystywane są klasy `Inventor` i `Patent`, zdefiniowane na listingu 15.9.

Listing 15.9. Przykładowe klasy używane razem ze standardowymi operatorami kwerend

```

using System;
using System.Collections.Generic;
using System.Linq;

public class Patent
{
    // Tytuł opublikowanego wniosku.
    public string Title { get; }

    // Data oficjalnego opublikowania wniosku.
    public string YearOfPublication { get; }

    // Unikatowy numer przypisany do opublikowanego wniosku.
    public string? ApplicationNumber { get; set; }

    public long[] InventorIds { get; }

    public Patent(
        string title, string yearOfPublication, long[] inventorIds)
    {
        Title = title ?? throw new ArgumentNullException(nameof(title));
        YearOfPublication = yearOfPublication ?? 
            throw new ArgumentNullException(nameof(yearOfPublication));
        InventorIds = inventorIds ?? 
            throw new ArgumentNullException(nameof(inventorIds));
    }

    public override string ToString()
    {
        return $"{{ Title }} ({ { YearOfPublication } })";
    }
}

public class Inventor
{
    public long Id { get; }
    public string Name { get; }
    public string City { get; }
    public string State { get; }
    public string Country { get; }

    public Inventor(
        string name, string city, string state, string country, int id)
    {
        Name = name ?? throw new ArgumentNullException(nameof(name));
        City = city ?? throw new ArgumentNullException(nameof(city));
        State = state ?? throw new ArgumentNullException(nameof(state));
        Country = country ?? throw new ArgumentNullException(nameof(country));
        Id = id;
    }

    public override string ToString()
    {
        return $"{{ Name }} ({ { City } }, { { State } })";
    }
}

```

```
}

class Program
{
    static void Main()
    {
        IEnumerable<Patent> patents = PatentData.Patents;
        Print(patents);

        Console.WriteLine();

        IEnumerable<Inventor> inventors = PatentData.Inventors;
        Print(inventors);
    }

    private static void Print<T>(IEnumerable<T> items)
    {
        foreach (T item in items)
        {
            Console.WriteLine(item);
        }
    }
}

public static class PatentData
{
    public static readonly Inventor[] Inventors = new Inventor[]
    {
        new Inventor(
            "Benjamin Franklin", "Filadelfia",
            "PA", "USA", 1 ),
        new Inventor(
            "Orville Wright", "Kitty Hawk",
            "NC", "USA", 2),
        new Inventor(
            "Wilbur Wright", "Kitty Hawk",
            "NC", "USA", 3),
        new Inventor(
            "Samuel Morse", "Nowy Jork",
            "NY", "USA", 4),
        new Inventor(
            "George Stephenson", "Wylam",
            "Northumberland", "UK", 5),
        new Inventor(
            "John Michaelis", "Chicago",
            "IL", "USA", 6),
        new Inventor(
            "Mary Phelps Jacob", "Nowy Jork",
            "NY", "USA", 7),
    };
}

public static readonly Patent[] Patents = new Patent[]
{
    new Patent("Okulary dwuogniskowe", "1784",
        InventorIds: new long[] {1}),
    new Patent("Fonograf", "1877",
        InventorIds: new long[] {1}),
}
```

```

new Patent("Kinetoskop", "1888",
    InventorIds: new long[] {1}),
new Patent("Telegraf elektryczny", "1837",
    InventorIds: new long[] {4}),
new Patent("Maszyna latająca", "1903",
    InventorIds: new long[] {2,3}),
new Patent("Lokomotywa parowa", "1815",
    InventorIds: new long[] {5}),
new Patent("Mechanizm nakładania kropelek", "1989",
    InventorIds: new long[] {6}),
new Patent("Biustonosz bez pleców", "1914",
    InventorIds: new long[] {7}),
};

}

```

Na listingu 15.9 przedstawiono fragment przykładowych danych. W danych wyjściowych 15.1 znajdziesz efekt uruchomienia kodu z tego listingu.

DANE WYJŚCIOWE 15.1.

```

Okulary dwuogniskowe (1784)
Fonograf (1877)
Kinetoskop (1888)
Telegraf elektryczny (1837)
Maszyna latająca (1903)
Lokomotywa parowa (1815)
Mechanizm nakładania kropelek (1989)
Biustonosz bez pleców (1914)

Benjamin Franklin (Filadelfia, PA)
Orville Wright (Kitty Hawk, NC)
Wilbur Wright (Kitty Hawk, NC)
Samuel Morse (Nowy Jork, NY)
George Stephenson (Wylam, Northumberland)
John Michaelis (Chicago, IL)
Mary Phelps Jacob (Nowy Jork, NY)

```

3.0

Filtrowanie danych za pomocą metody Where()

Aby przefiltrować dane z kolekcji, należy udostępnić metodę filtrującą, która zwraca wartość `true` lub `false`. Ta wartość określa, czy dany element należy uwzględnić, czy wykluczyć. Wyrażenie delegata przyjmujące argument i zwracające wartość logiczną jest nazywane **predykatem**. W metodzie `Where()` kolekcji predykaty służą do określania kryteriów filtrowania, co pokazano na listingu 15.10. W ujęciu technicznym wynik wykonania metody `Where()` to **obiekt** ukrywający operację filtrowania danej sekwencji wartości za pomocą określonego predykatu. Wynik działania kodu pokazano w danych wyjściowych 15.2.

Listing 15.10. Filtrowanie danych za pomocą metody System.Linq.Enumerable.Where()

```

using System;
using System.Collections.Generic;
using System.Linq;

```

```

class Program
{
    static void Main()
    {
        IEnumerable<Patent> patents = PatentData.Patents;
        patents = patents.Where(
            patent => patent.YearOfPublication.StartsWith("18"));
        Print(patents);
    }
    // ...
}

```

DANE WYJŚCIOWE 15.2.

Fonograf (1877)
 Kinetoskop (1888)
 Telegraf elektryczny (1837)
 Lokomotywa parowa (1815)

Zauważ, że kod przypisuje dane wyjściowe wywołania `Where()` do obiektu typu `IEnumerable<T>`. Oznacza to, że danymi wyjściowymi wywołania `IEnumerable<T>.Where()` jest nowa kolekcja typu `IEnumerable<T>` (na listingu 15.10 jest to kolekcja typu `IEnumerable<Patent>`).

Mniej oczywiste jest, że wyrażenie podane jako argument metody `Where()` nie musi być wykonywane w momencie przypisywania wyniku uruchomienia tej metody do nowej kolekcji. Dotyczy to wielu standardowych operatorów kwerend. Na przykład w metodzie `Where()` wyrażenie jest przekazywane do kolekcji i „zapisywane”, ale nie zostaje od razu wykonane. Wykonanie wyrażenia ma miejsce dopiero wtedy, gdy trzeba rozpoczęć iterowanie po elementach kolekcji. Na przykład pętla `foreach`, taka jak w metodzie `Print()` z listingu 15.9, powoduje przetworzenie wyrażenia dla każdego elementu kolekcji. Dlatego metodę `Where()` należy traktować tak, jakby służyła do określania kwerendy zależnej od wartości elementów kolekcji. Metoda ta nie iteruje po kolekcji w celu wygenerowania nowej kolekcji o potencjalnie mniejszej liczbie elementów.

Projekcje z wykorzystaniem metody `Select()`

3.0

Ponieważ danymi wyjściowymi metody `IEnumerable<T>.Where()` jest nowa kolekcja typu `IEnumerable<T>`, możliwe jest ponowne wywołanie standardowego operatora kwerend dla tej samej kolekcji. Na przykład zamiast tylko filtrować dane z pierwotnej kolekcji, można je przekształcić (zobacz listingu 15.11).

Listing 15.11. Projekcja z wykorzystaniem metody `System.Linq.Enumerable.Select()`

```

using System;
using System.Collections.Generic;
using System.Linq;

class Program
{
    static void Main()
    {

```

```

{
    IEnumerable<Patent> patents = PatentData.Patents;
    IEnumerable<Patent> patentsOf1800 = patents.Where(
        patent => patent.YearOfPublication.StartsWith("18"));
    IEnumerable<string> items = patentsOf1800.Select(
        patent => patent.ToString());
    Print(items);
}

// ...
}

```

Kod z listingu 15.11 tworzy nową kolekcję typu `IEnumerable<string>`. W tym przykładzie dodanie wywołania `Select()` nie zmienia danych wyjściowych. Dzieje się tak jednak tylko dlatego, że wywołanie `Console.WriteLine()` w metodzie `Print()` i tak korzysta z metody `ToString()`. W kodzie nastąpiła jednak transformacja każdego elementu typu `Patent` z pierwotnej kolekcji na element typu `string` w kolekcji `items`.

Przyjrzyj się teraz przykładowi, w którym wykorzystano typ `System.IO.FileInfo` (zobacz listing 15.12).

Listing 15.12. Projekcja z wykorzystaniem metody `System.Linq.Enumerable.Select()` i operatora new

```

// ...
IEnumerable<string> fileList = Directory.GetFiles(
    rootDirectory, searchPattern);
IEnumerable<FileInfo> files = fileList.Select(
    file => new FileInfo(file));
// ...

```

Tu kolekcja `fileList` jest typu `IEnumerable<string>`. Jednak za pomocą projekcji opartej na metodzie `Select` można przekształcić każdy element z tej kolekcji na obiekt typu `System.IO.FileInfo`.

Ponadto, wykorzystując krotki, można utworzyć kolekcję typu `IEnumerable<T>` z krotką podaną jako parametr `T` (zobacz listing 15.13 i dane wyjściowe 15.3).

3.0

Listing 15.13. Projekcja na krotkę

```

// ...
IEnumerable<string> fileList = Directory.EnumerateFiles(
    rootDirectory, searchPattern);
IEnumerable<(string FileName, long Size)> items = fileList.Select(
    file =>
{
    FileInfo fileInfo = new FileInfo(file);
    return (
        FileName: fileInfo.Name,
        Size: fileInfo.Length
    );
});
// ...

```

DANE WYJŚCIOWE 15.3.

```
FileName = AssemblyInfo.cs, Size = 1704
FileName = CodeAnalysisRules.xml, Size = 735
FileName = CustomDictionary.xml, Size = 199
FileName = EssentialCSharp.sln, Size = 40415
FileName = EssentialCSharp.suo, Size = 454656
FileName = EssentialCSharp.vsmdi, Size = 499
FileName = EssentialCSharp.vssscc, Size = 256
FileName = intelliTechture.ConsoleTester.dll, Size = 24576
FileName = intelliTechture.ConsoleTester.pdb, Size = 30208
```

Dane wyjściowe reprezentujące typ anonimowy automatycznie obejmują nazwy i wartości właściwości. Te informacje zwraca metoda `ToString()` generowana na potrzeby typu anonimowego.

Projekcja oparta na metodzie `Select()` daje bardzo duże możliwości. Zobaczyłeś już, jak przeprowadzić pionowe filtrowanie kolekcji (polegające na zmniejszeniu liczby elementów w kolekcji) za pomocą standardowego operatora kwerend `Where()`. Teraz wiesz już też, że przy użyciu standardowego operatora kwerend `Select()` możesz również przeprowadzić filtrowanie poziome (zmniejszyć liczbę kolumn) lub całkowicie przekształcić dane. Tak więc metody `Where()` i `Select()` umożliwiają pobranie z pierwotnej kolekcji tylko tych danych, które są potrzebne w wykonywanym algorytmie. Już same te dwie metody tworzą wartościowy interfejs API do manipulowania kolekcjami, a samodzielne uzyskanie podobnego efektu wymagałoby napisania znacznie większej ilości kodu, który w dodatku byłby mniej czytelny.

Początek
4.0

ZAGADNIENIE DLA ZAAWANSOWANYCH**Równoległe wykonywanie kwerend w technologii LINQ**

Ponieważ wiele komputerów ma kilka procesorów i po kilka rdzeni w każdym procesorze, coraz ważniejsza staje się możliwość wykorzystania tej dodatkowej mocy obliczeniowej. Wymaga to zmodyfikowania programów o obsługę kilku wątków, co pozwala jednocześnie wykonywać zadania w różnych procesorach komputera. Na listingu 15.14 pokazano jeden ze sposobów na osiągnięcie tego celu za pomocą technologii PLINQ (ang. *Parallel LINQ*).

Listing 15.14. Równoległe wykonywanie kwerend w technologii LINQ

3.0

```
// ...
IEnumerable<string> fileList = Directory.EnumerateFiles(
    rootDirectory, searchPattern);
var items = fileList.AsParallel().Select(
    file =>
{
    FileInfo fileInfo = new FileInfo(file);
    return new
    {
        FileName = fileInfo.Name,
        Size = fileInfo.Length
    };
});
// ...
```

Na listingu 15.14 pokazano, że dodanie obsługi przetwarzania równoległego wymaga jedynie drobnych zmian w kodzie. Tu używany jest tylko wprowadzony w platformie .NET 4 standardowy operator kwerend `AsParallel()` ze statycznej klasy `System.Linq.ParallelEnumerable`. Zastosowanie tej prostej metody rozszerzającej sprawia, że środowisko uruchomieniowe zaczyna równolegle przetwarzać elementy z kolekcji `fileList` i zwracać wynikowe obiekty. W tym kodzie wykonywane równolegle operacje nie są kosztowne (choć należy to oceniać względem innych czynności), pomyśl jednak o zadaniach wymagających dużo czasu pracy procesora, takich jak szyfrowanie lub kompresja. Równoległe przetwarzanie kwerendy za pomocą wielu procesorów pozwala skrócić czas wykonywania programu tyle razy, ile jest procesorów.

Wałą kwestią, o której należy wiedzieć (i która powoduje, że metodę `AsParallel()` opisano w zagadnieniu dla zaawansowanych), jest to, że przetwarzanie równolegle może doprowadzić do sytuacji wyścigu. Polega to na tym, że operacje z jednego wątku mogą się przeplatać z operacjami z innego wątku i skutkować uszkodzeniem danych. Aby można było uniknąć tego problemu, potrzebne są mechanizmy synchronizacji. Należy je stosować do danych współużytkowanych przez wiele wątków, aby w odpowiednich miejscach wymusić atomowość operacji. Jednak sama synchronizacja może prowadzić do zakleszczenia, co blokuje wykonywanie programu. To dodatkowo utrudnia skuteczne pisanie programów równoległych.

Więcej szczegółowych informacji o tym zagadnieniu i o innych tematach związanych z wielowątkowością znajdziesz w rozdziałach od 19. do 22.

Koniec
4.0

Zliczanie elementów za pomocą metody Count()

Inna kwerenda często wykonywana na kolekcjach polega na pobieraniu liczby elementów. Na potrzeby tej kwerendy w technologii LINQ udostępniono metodę rozszerzającą `Count()`.

Na listingu 15.15 pokazano, że metoda `Count()` jest przeciążona w ten sposób, by zliczała wszystkie elementy (wersja bez parametrów) lub przyjmowała predykat pozwalający ustalić liczbę zgodnych z nim elementów.

Listing 15.15. Zliczanie elementów za pomocą metody `Count()`

3.0

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

class Program
{
    static void Main()
    {
        IEnumerable<Patent> patents = PatentData.Patents;
        Console.WriteLine($"Liczba patentów: { patents.Count() }");
        Console.WriteLine($"Liczba patentów z XIX wieku: { 
            patents.Count(patent =>
                patent.YearOfPublication.StartsWith("18"))
        }");
    }
    // ...
}
```

Choć instrukcja `Count()` na pozór upraszcza kolekcję, w rzeczywistości kolekcja typu `IEnumerable<T>` się nie zmienia, dlatego wykonywany kod iteruje po wszystkich elementach kolekcji. Gdy w kolekcji bezpośrednio dostępna jest właściwość `Count`, lepiej korzystać z niej zamiast z metody `Count()` z technologii LINQ (to subtelna różnica). Interfejs `ICollection<T>` obejmuje właściwość `Count`, dlatego kod wywołujący metodę `Count()` na kolekcji zawierającej implementację tego interfejsu rzutuje kolekcję na ten interfejs i bezpośrednio wywołuje właściwość `Count`. Jeśli jednak kolekcja nie obejmuje implementacji interfejsu `ICollection<T>`, metoda `Enumerable.Count()` iteruje po wszystkich elementach kolekcji, zamiast wywoływać wbudowaną właściwość `Count`. Jeżeli celem sprawdzania liczby elementów jest ustalenie, czy różni się ona od zera (`if (patents.Count() > 0) {...}`), lepiej zastosować operator `Any()` (`if (patents.Any()) {...}`). Ten operator sprawdza tylko jeden z elementów kolekcji, aby zwrócić wartość `true` — nie musi iterować po całej kolekcji.

Wskazówki

STOSUJ operator `System.Linq.Enumerable.Any()` zamiast wywołania metody `patents.Count()`, gdy chcesz sprawdzić, czy kolekcja ma więcej niż zero elementów.

STOSUJ właściwość `Count` kolekcji (jeśli jest dostępna) zamiast metody `System.Linq.Enumerable.Count()`.

Opóźnione wykonanie

Jednym z najważniejszych aspektów, o których należy pamiętać w trakcie korzystania z technologii LINQ, jest opóźnione wykonanie. Przyjrzyj się kodowi z listingu 15.16 i odpowiadającym mu danym wyjściowym 15.4.

Listing 15.16. Filtrowanie z wykorzystaniem metody `System.Linq.Enumerable.Where()`

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

// ...

IQueryable<Patent> patents = PatentData.Patents;
bool result;
patents = patents.Where(
    patent =>
{
    if (result =
        patent.YearOfPublication.StartsWith("18"))
    {
        // Efekty uboczne w predykatce
        // służą tu do zademonstrowania reguł języka.
        // Zwykle należy unikać pisania takiego kodu.
        Console.WriteLine("\t" + patent);
    }
    return result;
});
```

```
Console.WriteLine("1. Patenty sprzed 1900 roku:");
foreach (Patent patent in patents)
{
}

Console.WriteLine();
Console.WriteLine(
    "2. Druga lista patentów sprzed 1900 roku:");
Console.WriteLine(
    $" Liczba patentów sprzed 1900 roku to: {patents.Count()}.");

Console.WriteLine();
Console.WriteLine(
    "3. Trzecia lista patentów sprzed 1900 roku:");
patents = patents.ToArray();
Console.Write($" Liczba patentów sprzed 1900 roku to: ");
Console.WriteLine(
    $"{ patents.Count() }");

// ...
```

DANE WYJŚCIOWE 15.4.

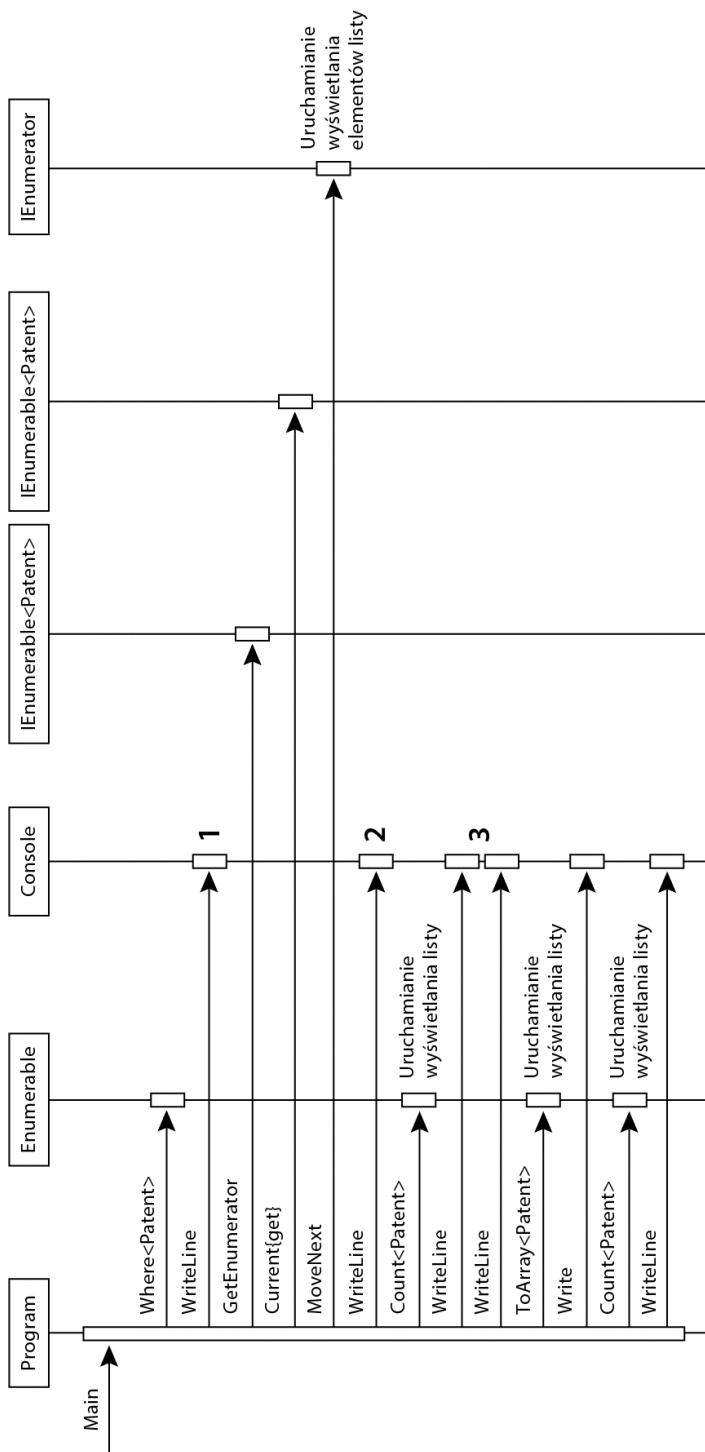
1. Patenty sprzed 1900 roku:
 - Fonograf (1877)
 - Kinetoskop (1888)
 - Telegraf elektryczny (1837)
 - Lokomotywa parowa (1815)
2. Druga lista patentów sprzed 1900 roku:
 - Fonograf (1877)
 - Kinetoskop (1888)
 - Telegraf elektryczny (1837)
 - Lokomotywa parowa (1815)Liczba patentów sprzed 1900 roku to: 4.
3. Trzecia lista patentów sprzed 1900 roku:
 - Fonograf (1877)
 - Kinetoskop (1888)
 - Telegraf elektryczny (1837)
 - Lokomotywa parowa (1815)Liczba patentów sprzed 1900 roku to: 4.

3.0

Zauważ, że wywołanie `Console.WriteLine("1. Patenty sprzed...")` jest wykonywane przed wyrażeniem lambda. Jest to bardzo ważna cecha, na którą należy zwrócić uwagę. Nie jest ona oczywista dla osób, które nie są świadome jej znaczenia. Zwykle predykaty powinny robić dokładnie jedną rzecz — sprawdzać warunek. Nie powinny mieć żadnych efektów ubocznych (nawet takich jak wyświetlanie tekstu w konsoli, co ma miejsce w tym przykładzie).

Aby zrozumieć, co dzieje się w kodzie, przypomnij sobie, że wyrażenia lambda to delegaty (czyli referencje do metod), dlatego można je przekazywać. W kontekście technologii LINQ i standardowych operatorów kwerend każde wyrażenie lambda stanowi część ogólnej wykonywanej kwerendy.

W miejscu deklaracji wyrażenia lambda nie są wykonywane. Dopiero w momencie wywołania wyrażenia lambda następuje rozpoczęcie wykonywania jego kodu. Sekwencję operacji przedstawiono na rysunku 15.2.



Rysunek 15.2. Sekwencja operacji wywołujących wyrażenia lambda

Zgodnie z rysunkiem 15.2 wyrażenie lambda jest uruchamiane przez trzy wywołania z listingu 15.14. Za każdym razem odbywa się to pośrednio. Gdyby używane wyrażenie było kosztowne (tak jak wywołania kierowane do bazy danych), ważne byłoby zminimalizowanie liczby jego wykonień.

Po raz pierwszy wyrażenie lambda jest uruchamiane w pętli `foreach`. Zgodnie z opisem z wcześniejszej części rozdziału pętla `foreach` jest przekształcana na wywołania `MoveNext()`, a każde takie wywołanie skutkuje uruchomieniem wyrażenia lambda dla każdego elementu z pierwotnej kolekcji. W trakcie iteracji środowisko uruchomieniowe wywołuje wyrażenie lambda dla każdego elementu, by ustalić, czy dany element jest zgodny z predykatem.

Drugi raz wyrażenie lambda jest uruchamiane dla każdego elementu w wyniku wywołania funkcji `Count()` z typu `Enumerable`. Może to być trudne do zauważenia, ponieważ w kolekcjach, dla których nie są używane standardowe operatory kwerend, bardzo często występuje właściwość o tej samej nazwie (`Count`).

Trzeci raz wyrażenie lambda jest wykonywane dla każdego elementu z powodu wywołania metody `ToArrayList()` (może to być też wywołanie `ToDictionary()` lub `ToLookup()`). Przekształcenie kolekcji za pomocą jednej z metod z rodziny `To...` jest bardzo pomocne. Zwarcana jest wtedy kolekcja, dla której standardowy operator kwerend został już wykonany. Na listingu 15.14 konwersja kolekcji na tablicę powoduje, że gdy w ostatniej instrukcji `Console.WriteLine()` wywoływana jest właściwość `Length`, zmienna `patients` jest powiązana z tablicą (która, co oczywiste, zawiera implementację interfejsu `IEnumerable<T>`). Dlatego wywoływana jest właściwość `Length` z klasy `System.Array`, a nie odpowiednik tej właściwości z typu `System.Linq.Enumerable`. Z tego względu po konwersji danych na jeden z typów kolekcji zwracanych przez metody `To...` zwykle można bezpiecznie posługiwać się kolekcją (do czasu wywołania następnego standardowego operatora kwerend). Zauważ jednak, że po takiej konwersji cały zbiór wyników jest zapisywany w pamięci (wcześniej może zostać zapisany w bazie lub w pliku). Ponadto metody `To...` tworzą snapshot danych, dlatego późniejsze kwerendy dotyczące wyników zwróconych przez te metody nie powodują pobierania nowych danych.

Warto szczegółowo dokładnie przeanalizować diagram sekwencyjny z rysunku 15.2 oraz powiązany kod. Opóźnione wykonywanie standardowych operatorów kwerend może prowadzić do bardzo subtelnych przesunięć momentu uruchamiania takich operatorów. Dlatego programiści powinni zachować ostrożność i starać się unikać nieoczekiwanych wywołań. Obiekt kwerendy reprezentuje kwerendę, a nie wyniki. Po zażądaniu wyników z kwerendy zostanie ona wykonana (możliwe, że ponownie), ponieważ nie wie, że wyniki będą takie same jak przy jej poprzednim uruchomieniu (jeśli miało już ono miejsce).

3.0

Uwaga

Aby uniknąć ponownego wykonywania kodu, musisz zapisać w pamięci podręcznej dane pobrane przez wykonaną kwerendę. W tym celu należy przypisać dane do lokalnej kolekcji za pomocą jednej z metod `To...`. Wywołanie takiej metody w momencie przypisywania danych oczywiście wymaga uruchomienia kwerendy. Jednak późniejsze iterowanie po zapisanej kolekcji nie wymaga już kolejnych wykonień kwerendy. Zwykle jeśli chcesz, aby kolekcja działała jak zapisany w pamięci snapshot, najlepiej przypisać wynik kwerendy do kolekcji zapisanej w pamięci podręcznej. Pozwala to uniknąć zbędnego powtarzania kwerend.

Sortowanie z wykorzystaniem metod OrderBy() i ThenBy()

Inna często wykonywana operacja na kolekcjach to sortowanie. Proces ten obejmuje wywołanie metody System.Linq.Enumerable.OrderBy() (zobacz listing 15.17 i dane wyjściowe 15.5).

Listing 15.17. Porządkowanie elementów za pomocą metod System.Linq.Enumerable.OrderBy() i ThenBy()

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

// ...
IQueryable<Patent> items;
Patent[] patents = PatentData.Patents;
items = patents.OrderBy(
    patent => patent.YearOfPublication).ThenBy(
    patent => patent.Title);
Print(items);
Console.WriteLine();

items = patents.OrderByDescending(
    patent => patent.YearOfPublication).ThenByDescending(
    patent => patent.Title);
Print(items);
// ...
```

DANE WYJŚCIOWE 15.5.

Okulary dwuogniskowe (1784)
Lokomotywa parowa (1815)
Telegraf elektryczny (1837)
Fonograf (1877)
Kinetoskop (1888)
Maszyna latająca (1903)
Biustonosz bez pleców (1914)
Mechanizm nakładania kropelek (1989)

Mechanizm nakładania kropelek (1989)
Biustonosz bez pleców (1914)
Maszyna latająca (1903)
Kinetoskop (1888)
Fonograf (1877)
Telegraf elektryczny (1837)
Lokomotywa parowa (1815)
Okulary dwuogniskowe (1784)

3.0

Metoda OrderBy() przyjmuje wyrażenie lambda określające klucz używany do sortowania. Na listingu 15.17 najpierw patenty są sortowane na podstawie roku ich publikacji.

Zauważ, że metoda OrderBy() przyjmuje tylko jeden parametr określający sposób sortowania. Jest to parametr o nazwie keySelector. Aby dodatkowo posortować dane według drugiej kolumny, trzeba zastosować inną metodę — ThenBy(). Ta sama metoda pozwala przeprowadzać dodatkowe sortowanie według kolejnych kolumn.

Metoda `OrderBy()` zwraca obiekt typu `IOrderedEnumerable<T>`, a nie `IEnumerable<T>`. Typ `IOrderedEnumerable<T>` jest pochodny od `IEnumerable<T>`, dlatego wartość zwrócona przez metodę `OrderBy()` zapewnia dostęp do wszystkich standardowych operatorów kwerend (w tym do metody `OrderBy()`). Jednak ponowne wywołanie metody `OrderBy()` powoduje anulowanie efektów wcześniejszego jej uruchomienia. Wtedy końcowe dane są posortowane tylko na podstawie argumentu `keySelector` z ostatniego wywołania metody `OrderBy()`. Dlatego powinieneś dbać o to, by nie wywoływać metody `OrderBy()` na wynikach zwróconych przez wcześniejsze wywołanie tej metody.

Dodatkowe kryteria sortowania należy podawać w metodzie `ThenBy()`. Jest ona metodą rozszerzającą, nie jest jednak dodawana do interfejsu `IEnumerable<T>`, a do interfejsu `IOrderedEnumerable<T>`. Ta metoda, także zdefiniowana w typie `System.Linq.Enumerable`, jest zadeklarowana w następujący sposób:

```
public static IOrderedEnumerable<TSource>
    ThenBy<TSource, TKey>(
        this IOrderedEnumerable<TSource> source,
        Func<TSource, TKey> keySelector)
```

Najpierw należy więc wywołać metodę `OrderBy()`, a następnie zero lub więcej razy uruchomić metodę `ThenBy()`, aby określić dodatkowe „kolumny” używane do sortowania. Metody `OrderByDescending()` i `ThenByDescending()` zapewniają podobne mechanizmy, ale porządkują dane malejąco. Łączenie metod sortujących dane rosnąco i malejąco jest dozwolone, natomiast jeśli sortujesz dane według dodatkowych kryteriów, korzystaj z metod `ThenBy()` (z sortowaniem danych rosnąco lub malejąco).

Warto wspomnieć o dwóch innych ważnych kwestiach dotyczących sortowania. Po pierwsze, sortowanie ma miejsce dopiero wtedy, gdy kod zacznie posługiwać się elementami kolekcji. Na tym etapie cała kwerenda jest już wykonana. Nie można sortować danych, jeśli któreś elementy są niedostępne (uniemożliwia to ustalenie pierwszego elementu). Odraczanie sortowania do czasu, gdy w kodzie potrzebny jest dostęp do elementów, wynika z opisanego we wcześniejszej części rozdziału opóźnionego wykonywania kodu. Po drugie, każde kolejne wywołanie metody sortującej dane (na przykład w sekwencji `OrderBy()`, `ThenBy()` i `ThenByDescending()`) skutkuje dodatkowymi wywołaniami wyrażenia lambda podanego w parametrze `keySelector` we wcześniejszych wywołaniach. Tak więc metoda `OrderBy()` wywołuje w trakcie iterowania po kolekcji wyrażenie lambda podane jako jej parametr `keySelector`. Ponadto późniejsze uruchomienie metody `ThenBy()` także spowoduje wywołanie wyrażenia lambda z parametru `keySelector` metody `OrderBy()`.

3.0

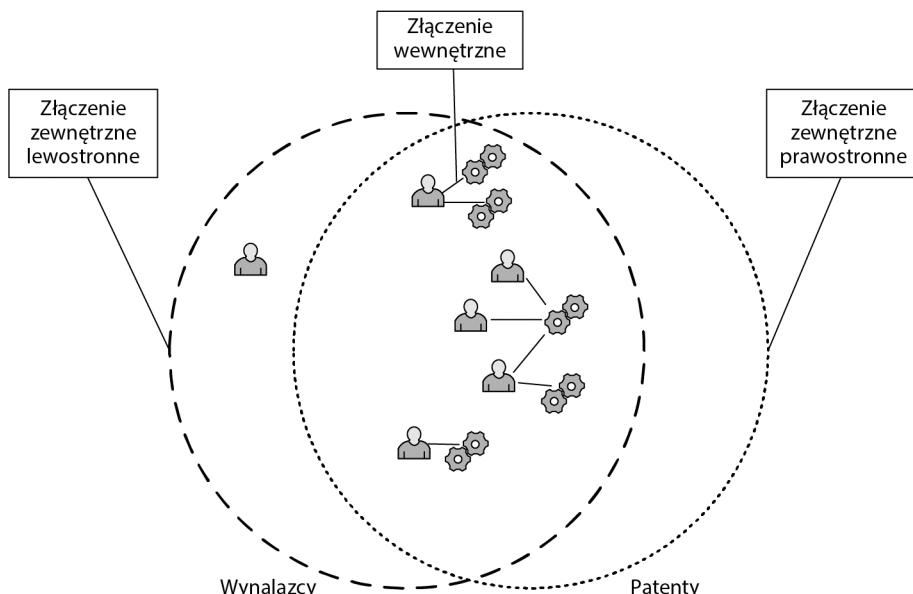
Wskazówka

NIE wywołuj ponownie metody `OrderBy()`. Jeśli chcesz uporządkować elementy na podstawie więcej niż jednej wartości, zastosuj metodę `ThenBy()`.

ZAGADNIENIE DLA POCZĄTKUJĄCYCH

Złączanie

Przyjrzyj się dwóm kolekcjom obiektów przedstawionym na diagramie Venna na rysunku 15.3. Lewe koło na diagramie obejmuje wszystkich wynalazców, a prawe — wszystkie patenty. Część wspólna zawiera zarówno wynalazców, jak i patenty, a linie łączą wynalazców z patentami, jakie uzyskali. Na diagramie widać, że każdy wynalazca może zgłosić wiele patentów, a każdy patent może być powiązany z jednym lub kilkoma wynalazcami. Na diagramie każdy patent ma określonego wynalazcę, natomiast niektórzy wynalazcy nie zgłosili jeszcze żadnych patentów.



Rysunek 15.3. Diagram Venna dotyczący kolekcji reprezentujących wynalazców i patenty

Dopasowanie wynalazców do patentów z części wspólnej to **złączenie wewnętrzne**. Wynik tego złączenia to kolekcja par wynalazca-patent, w których istnieją zarówno wynalazcy, jak i patenty z par. **Złączenie zewnętrzne lewostronne** obejmuje wszystkie elementy z lewego koła niezależnie od tego, czy istnieje odpowiadający im patent. W tym przykładzie **złączenie zewnętrzne prawostronne** jest identyczne ze złączeniem wewnętrzny, ponieważ nie występują patenty bez wynalazców. Ponadto podział na lewą i prawą stronę jest arbitralny, dlatego nie istnieje różnica między złączeniami lewo- i prawostronnymi. **Pełne złączenie zewnętrzne** obejmuje elementy występujące po obu stronach i jest przeprowadzane stosunkowo rzadko.

Ważną cechą relacji między wynalazcami i patentami jest to, że jest to relacja **wiele do wielu**. Każdy patent może być zgłoszony przez jednego lub więcej wynalazców (na przykład maszyna latająca jest wynalazkiem Orville'a i Wilbura Wrightów). Ponadto każdy wynalazca może zgłosić jeden lub więcej patentów (na przykład Benjamin Franklin jest twórcą zarówno okularów dwusoczekowych, jak i fonografu).

Często występują też relacje **jeden do wielu**. Na przykład w jednym dziale firmy może pracować wiele osób. Jednak każdy pracownik w danym momencie jest zatrudniony w tylko jednym dziale. Jednak, jak często się zdarza w przypadku relacji jeden do wielu, dodanie czynnika czasu może przekształcić ją w relację wiele do wielu. Dana osoba może przejść z jednego działu do innego, dlatego z czasem może zostać powiązana z wieloma działami. W ten sposób powstaje następująca relacja wiele do wielu.

Na listingu 15.18 znajduje się przykładowy kod z danymi pracowników i działów. Wykonanie tego kodu znajdziesz w danych wyjściowych 15.6.

Listing 15.18. Przykładowe dane pracowników i działów

```
public class Department
{
    public long Id { get; }
    public string Name { get; }
    public Department(string name, long id)
    {
        Id = id;
        Name = name ?? throw new ArgumentNullException(nameof(name));
    }
    public override string ToString()
    {
        return Name;
    }
}
public class Employee
{
    public int Id { get; }
    public string Name { get; }
    public string Title { get; }
    public int DepartmentId { get; }
    public Employee(
        string name, string title, int departmentId)
    {
        Name = name ?? throw new ArgumentNullException(nameof(name));
        Title = title ?? throw new ArgumentNullException(nameof(title));
        DepartmentId = departmentId;
    }
    public override string ToString()
    {
        return $"{ Name } ({ Title })";
    }
}
public static class CorporateData
{
    public static readonly Department[] Departments =
        new Department[]
    {
        new Department("Dział korporacyjny", 0),
        new Department("Dział finansów", 1),
        new Department("Dział inżynierii", 2),
        new Department("Dział IT", 3),
        new Department("Dział filantropii", 4),
        new Department("Dział marketingu", 5),
    };
}
```

```
public static readonly Employee[] Employees = new Employee[]
{
    new Employee("Mark Michaelis", "Główny nerd", 0),
    new Employee("Michael Stokesbary", "Starszy geniusz komputerowy", 2),
    new Employee("Brian Jones", "Guru od integracji systemów", 2),
    new Employee("Anne Beard", "Dyrektor działu kadr", 1),
    new Employee("Pat Dever", "Architekt systemów korporacyjnych", 3),
    new Employee("Kevin Bost", "Programista nadzwyczajny", 2),
    new Employee("Thomas Heavey", "Architekt oprogramowania", 2),
    new Employee("Eric Edmonds", "Koordynator działań charytatywnych", 4)
};

class Program
{
    static void Main()
    {
        IEnumerable<Department> departments =
            CorporateData.Departments;
        Print(departments);

        Console.WriteLine();

        IEnumerable<Employee> employees =
            CorporateData.Employees;
        Print(employees);
    }

    private static void Print<T>(IEnumerable<T> items)
    {
        foreach (T item in items)
        {
            Console.WriteLine(item);
        }
    }
}
```

DANE WYJŚCIOWE 15.6.

Dział korporacyjny
Dział finansów
Dział inżynierii
Dział IT
Dział filantropii
Dział marketingu

Mark Michaelis (Główny nerd)
Michael Stokesbary (Starszy geniusz komputerowy)
Brian Jones (Guru od integracji systemów)
Anne Beard (Dyrektor działu kadr)
Pat Dever (Architekt systemów korporacyjnych)
Kevin Bost (Programista nadzwyczajny)
Thomas Heavey (Architekt oprogramowania)
Eric Edmonds (Koordynator działań charytatywnych)

3.0

Te dane będą używane w przykładowym kodzie w dalszych podrozdziałach poświęco-nych złączaniu danych.

Przeprowadzanie łączenia wewnętrznego za pomocą instrukcji Join()

Po stronie klienta relacje między obiektami są już zwykle ustalone. Na przykład relacja między plikami i zawierającymi je katalogami jest wstępnie określona za pomocą metod `DirectoryInfo.GetFiles()` i `FileInfo.Directory`. Jednak w przypadku danych wczytywanych z magazynów nieobiektowych często jest inaczej. Dane trzeba wtedy złączyć, by móc w dostosowany do nich sposób przejść od obiektu jednego typu do obiektu innego typu.

Pomyśl o pracownikach i działach firmy. Na listingu 15.19 kod łączy ka dego pracownika z jego dzia em, a nast pnie wyświetla list  wszystkich pracowników i ich dzia ów. Poniewa  ka dy pracownik jest zatrudniony w (dok adnie) jednym dziale,  czna liczba elementów na liście jest równa  cznej liczbie pracowników. Ka dy pracownik pojawia si  tylko raz (dane pracowników s  **znormalizowane**). Efekt dzia ania kodu pokazano w danych wyj ciowych 15.7.

Listing 15.19. Wewnętrzne   czenie uzyskane za pomoc  metody `System.Linq.Enumerable.Join()`

```
using System;
using System.Linq;

// ...
Department[] departments = CorporateData.Departments;
Employee[] employees = CorporateData.Employees;

IQueryable<(int Id, string Name, string Title,
    Department Department)> items =
employees.Join(
    departments,
    employee => employee.DepartmentId,
    department => department.Id,
    (employee, department) => (
        employee.Id,
        employee.Name,
        employee.Title,
        department
));
3.0
foreach (var item in items)
{
    Console.WriteLine(
        $"{ item.Name } ({ item.Title })");
    Console.WriteLine("\t" + item.Department);
}

// ...
```

DANE WYJ CIOWE 15.7.

```
Mark Michaelis (G owyner nerd)
Dzia  korporacyjny
Michael Stokesbary (Starszy geniusz komputerowy)
Dzia  in enierii
Brian Jones (Guru od integracji systemów)
```

Dział inżynierii
Anne Beard (Dyrektor działu kadr)
Dział kadr
Pat Dever (Architekt systemów korporacyjnych)
Dział IT
Kevin Bost (Programista nadzwyczajny)
Dział inżynierii
Thomas Heavey (Architekt oprogramowania)
Dział inżynierii
Eric Edmonds (Koordynator działań charytatywnych)
Dział filantropii

Pierwszy parametr metody `Join()` ma nazwę `inner`. Określa on kolekcję (`departments`), z którą złączana jest kolekcja `employees`. Dwa następne parametry to wyrażenia lambda określające, jak kolekcje będą ze sobą powiązane. Wyrażenie `employee => employee.DepartmentId` (odpowiadające parametrowi `outerKeySelector`) oznacza, że dla każdego pracownika kluczem ma być właściwość `DepartmentId`. Następne wyrażenie lambda (`department => department.Id`) określa, że drugim kluczem ma być właściwość `Id` z typu `Department`. Oznacza to, że z każdym pracownikiem należy złączyć dział, gdy wartość `employee.DepartmentId` jest równa `department.Id`. Ostatni parametr określa typ pobieranych wynikowych elementów. Tu tworzona jest krotka z właściwościami `Id`, `Name` i `Title` z klasy `Employee` oraz właściwością `Department` odpowiadającą złączanemu obiektowi `department`.

Zauważ, że w danych wyjściowych tekst `Dział inżynierii` pojawia się wielokrotnie — jeden raz dla każdego zatrudnionego w tym dziale pracownika zapisanego w klasie `CorporateData`. Tu metoda `Join()` tworzy **iloczyn kartezjański** uwzględniający wszystkie działy i wszystkich pracowników w taki sposób, że nowy rekord jest tworzony dla każdego przypadku, gdy w obu kolekcjach występują rekordy o identycznych identyfikatorach działu. Jest to złączenie wewnętrzne.

Dane można złączać również w odwrotną stronę. Można na przykład złączyć działy z pracownikami, tak aby utworzyć listę działów i zatrudnionych w nich pracowników. Zauważ, że dane wyjściowe zawierają wtedy więcej rekordów, niż jest działów. W każdym dziale zatrudnionych może być wielu pracowników, a w danych wyjściowych znajduje się jeden rekord dla każdej pasującej pary dział-pracownik. Tak jak wcześniej, dział inżynierii pojawia się wielokrotnie (raz dla każdej zatrudnionej w nim osoby).

Kod z listingu 15.20 (generujący dane wyjściowe 15.8) jest podobny do kodu z listingu 15.19, jednak tym razem obiekty `departments` i `employees` są zamienione miejscami. Pierwszym parametrem metody `Join()` jest kolekcja `employees`. Ten parametr określa, z czym złączana jest kolekcja `departments`. Dwa następne parametry to wyrażenia lambda wyznaczające sposób złączania tych kolekcji. Klucze z kolekcji `departments` określa wyrażenie `department => department.Id`, a klucze z kolekcji `employees` odpowiadają wyrażeniu `employee => employee.DepartmentId`. Tak jak wcześniej, złączenie następuje, gdy wartość `department.Id` jest równa `employee.DepartmentId`. Ostatni parametr w postaci krotki tworzy klasę z właściwościami `int Id, string Name i Employee Employee`. Nazwy w wyrażeniu tworzącym krotkę są opcjonalne, tu zastosowano je, aby zwiększyć czytelność.

Listing 15.20. Następne złączenie wewnętrzne uzyskane za pomocą metody System.Linq.Enumerable.Join()

```
using System;
using System.Linq;

// ...
Department[] departments = CorporateData.Departments;
Employee[] employees = CorporateData.Employees;

IQueryable<(long Id, string Name, Employee Employee)> items =
    departments.Join(
        employees,
        department => department.Id,
        employee => employee.DepartmentId,
        (department, employee) => (
            department.Id,
            department.Name,
            employee
        ));
        
foreach (var item in items)
{
    Console.WriteLine(item.Name);
    Console.WriteLine("\t" + item.Employee);
}
// ...
```

DANE WYJŚCIOWE 15.8.

3.0

Dział korporacyjny	Mark Michaelis (Główny nerd)
Dział kadr	Anne Beard (Dyrektor działu kadr)
Dział inżynierii	Michael Stokesbary (Starszy geniusz komputerowy)
Dział inżynierii	Brian Jones (Guru od integracji systemów)
Dział inżynierii	Kevin Bost (Programista nadzwyczajny)
Dział inżynierii	Thomas Heavey (Architekt oprogramowania)
Dział IT	Pat Dever (Architekt systemów korporacyjnych)
Dział filantropii	Eric Edmonds (Koordynator działań charytatywnych)

Oprócz sortowania i złączania kolekcji obiektów często przydatne jest grupowanie obiektów o podobnych cechach. W danych dotyczących pracowników można grupować rekordy na podstawie działu, regionu, stanowiska itd. Na listingu 15.21 pokazano, jak zrobić to za pomocą standardowego operatora kwerend GroupBy(). Efekt działania kodu znajdziesz w danych wyjściowych 15.9.

Listing 15.21. Grupowanie elementów za pomocą metody System.Linq.Enumerable.GroupBy()

```
using System;
using System.Linq;

// ...

IEnumerable<Employee> employees = CorporateData.Employees;

IEnumerable<IGrouping<int, Employee>> groupedEmployees =
    employees.GroupBy((employee) => employee.DepartmentId);

foreach(IGrouping<int, Employee> employeeGroup in
    groupedEmployees)
{
    Console.WriteLine();
    foreach(Employee employee in employeeGroup)
    {
        Console.WriteLine("\t" + employee);
    }
    Console.WriteLine(
        "\tLiczba pracowników: " + employeeGroup.Count());
}
// ...
```

DANE WYJŚCIOWE 15.9.

```
Mark Michaelis (Główny nerd)
Liczba pracowników: 1
```

```
Michael Stokesbary (Starszy geniusz komputerowy)
Brian Jones (Guru od integracji systemów)
Kevin Bost (Programista nadzwyczajny)
Thomas Heavey (Architekt oprogramowania)
Liczba pracowników: 4
```

```
Anne Beard (Dyrektor działu kadr)
Liczba pracowników: 1
```

```
Pat Dever (Architekt systemów korporacyjnych)
Liczba pracowników: 1
```

```
Eric Edmonds (Koordynator działań charytatywnych)
Liczba pracowników: 1
```

Zauważ, że elementy zwracane przez metodę GroupBy() są typu `IGrouping<TKey, TElement>`. Ten typ obejmuje właściwość pełniąą funkcję klucza (`employee.DepartmentId`), który jest używany do grupowania danych w kwerendzie. W typie nie ma jednak właściwości przeznaczonej na elementy z grupy. Typ `IGrouping<TKey, TElement>` jest pochodny od typu `IEnumerable<T>`, dlatego umożliwia iterowanie po elementach w grupie za pomocą instrukcji `foreach` oraz agregowanie danych (na przykład w celu zliczania elementów za pomocą instrukcji `employeeGroup.Count()`).

Implementowanie relacji jeden do wielu za pomocą metody GroupJoin()

Kod z listingów 15.19 i 15.20 jest prawie identyczny. Oba wywołania `Join()` z tych listingów mogą zwracać te same dane wyjściowe — wystarczy zmodyfikować definicję krotki. Gdy potrzebna jest lista pracowników, odpowiednie wyniki zwraca kod z listingu 15.19. Obiekt `department` jest tam zapisywany jako element z obu krotek reprezentującychłączanych pracowników. Jednak kod z listingu 15.20 nie jest idealny. Ponieważ dostępne są kolekcje, lepszą reprezentacją działu jest kolekcja pracowników, a nie jedna krotka dla każdej pary dział-pracownik. To rozwiązanie przedstawiono na listingu 15.22, a lepszą wersję zwracanych informacji znajdziesz w danych wyjściowych 15.10.

Listing 15.22. Tworzenie kolekcji podrzędnej za pomocą metody `System.Linq.Enumerable.GroupJoin()`

```
using System;
using System.Linq;

// ...
Department[] departments = CorporateData.Departments;
Employee[] employees = CorporateData.Employees;

IQueryable<(long Id, string Name, IQueryable<Employee> Employees)>
items =
departments.GroupJoin(
    employees,
    department => department.Id,
    employee => employee.DepartmentId,
    (department, departmentEmployees) => (
        department.Id,
        department.Name,
        departmentEmployees
    ));
}

foreach (
    _, string name, IQueryable<Employee> employeeCollection) in items)
{
    Console.WriteLine(name);
    foreach (Employee employee in employeeCollection)
    {
        Console.WriteLine("\t" + employee);
    }
}
// ...
```

DANE WYJŚCIOWE 15.10.

Dział korporacyjny

Mark Michaelis (Główny nerd)

Dział kadr

Anne Beard (Dyrektor działu kadr)

Dział inżynierii

Michael Stokesbary (Starszy geniusz komputerowy)

Brian Jones (Guru od integracji systemów)

Kevin Bost (Programista nadzwyczajny)
Thomas Heavey (Architekt oprogramowania)

Dział IT

Pat Dever (Architekt systemów korporacyjnych)

Dział filantropii

Eric Edmonds (Koordynator działań charytatywnych)

Aby osiągnąć oczekiwany efekt, zastosowano metodę `GroupJoin()` z klasy `System.Linq.Enumerable`. Ta metoda przyjmuje parametry opisane w omówieniu listingu 15.19. Inną na obu listingach jest krotka używana do pobierania wartości. Na listingu 15.19 wyrażenie lambda jest typu `Func<Department, IEnumerable<Employee>, (long Id, string Name, IEnumerable<Employee> Employees)`. Zauważ, że w kodzie zastosowano drugi argument określający typ (`IEnumerable<Employee>`), aby przeprowadzić projekcję kolekcji pracowników każdego działu na wynikową krotkę reprezentującą dział. Dlatego każdy dział w wynikowej kolekcji obejmuje listę pracowników.

Osoby znające język SQL zauważą, że metoda `GroupJoin()` (w odróżnieniu od metody `Join()`) nie ma odpowiednika w SQL-u. Jest tak, ponieważ dane zwarcane w SQL-u są oparte na rekordach i nie są hierarchiczne.

ZAGADNIENIE DLA ZAAWANSOWANYCH

Implementowanie złączeń zewnętrznych za pomocą metody `GroupJoin()`

Przedstawiane wcześniej złączenia zewnętrzne to *złączenia równościowe*, ponieważ są dworzone dla kluczy o równych wartościach. Rekordy pojawiają się w wynikowej kolekcji tylko wtedy, gdy w obu kolekcjach występują pasujące do siebie obiekty. Czasem jednak rekord należy także utworzyć, gdy dla danej wartości nie istnieje pasujący obiekt w drugiej kolekcji. Na przykład zamiast pomijać na wynikowej liście dział marketingu, ponieważ nikt nie jest w nim zatrudniony, lepiej dodać ten dział razem z pustą listą pracowników. Aby osiągnąć ten cel, należy przeprowadzić złączenie zewnętrzne lewostronne, korzystając z metod `GroupJoin()`, `SelectMany()` i `DefaultIfEmpty()` (zobacz listing 15.23 i dane wyjściowe 15.11).

Listing 15.23. Implementowanie złączenia zewnętrzne za pomocą metod `GroupJoin()` i `SelectMany()`

```
using System;
using System.Linq;

// ...

Department[] departments = CorporateData.Departments;
Employee[] employees = CorporateData.Employees;

var items = departments.GroupJoin(
    employees,
    department => department.Id,
    employee => employee.DepartmentId,
    (department, departmentEmployees) => new
    {
        department.Id,
        department.Name,
```

7.0

```

Employees = departmentEmployees
}).SelectMany(
    departmentRecord =>
        departmentRecord.Employees.DefaultIfEmpty(),
        (departmentRecord, employee) => new
    {
        departmentRecord.Id,
        departmentRecord.Name,
        departmentRecord.Employees
    }).Distinct();

foreach (var item in items)
{
    Console.WriteLine(item.Name);
    foreach (Employee employee in item.Employees)
    {
        Console.WriteLine("\t" + employee);
    }
}
// ...

```

DANE WYJŚCIOWE 15.11.

```

Dział korporacyjny
    Mark Michaelis (Główny nerd)
Dział kadr
    Anne Beard (Dyrektor działu kadr)
Dział inżynierii
    Michael Stokesbary (Starszy geniusz komputerowy)
    Brian Jones (Guru od integracji systemów)
    Kevin Bost (Programista nadzwyczajny)
    Thomas Heavey (Architekt oprogramowania)
Dział IT
    Pat Dever (Architekt systemów korporacyjnych)
Dział filantropii
    Eric Edmonds (Koordynator działań charytatywnych)
Dział marketingu

```

3.0

Wywoływanie metody SelectMany()

Czasem używane są kolekcje kolekcji. Taką sytuację przedstawiono na listingu 15.24. Tablica teams zawiera dwie drużyny, z których każda obejmuje tablicę łańcuchów znaków reprezentujących graczy.

Listing 15.24. Wywoływanie metody SelectMany()

```

using System;
using System.Collections.Generic;
using System.Linq;

// ...
(string Team, string[] Players)[] worldCup2006Finalists = new[]
{
    (
        TeamName: "France",
        Players: new string[]

```

```

{
    "Fabien Barthez", "Gregory Coupet",
    "Mickael Landreau", "Eric Abidal",
    "Jean-Alain Boumsong", "Pascal Chimbonda",
    "William Gallas", "Gael Givet",
    "Willy Sagnol", "Mikael Silvestre",
    "Lilian Thuram", "Vikash Dhorasoo",
    "Alou Diarra", "Claude Makelele",
    "Florent Malouda", "Patrick Vieira",
    "Zinedine Zidane", "Djibril Cisse",
    "Thierry Henry", "Francck Ribery",
    "Louis Saha", "David Trezeguet",
    "Sylvain Wiltord",
}
),
(
    TeamName: "Italy",
    Players: new string[]
    {
        "Gianluigi Buffon", "Angelo Peruzzi",
        "Marco Amelia", "Cristian Zaccardo",
        "Alessandro Nesta", "Gianluca Zambrotta",
        "Fabio Cannavaro", "Marco Materazzi",
        "Fabio Grosso", "Massimo Oddo",
        "Andrea Barzaghi", "Andrea Pirlo",
        "Gennaro Gattuso", "Daniele De Rossi",
        "Mauro Camoranesi", "Simone Perrotta",
        "Simone Barone", "Luca Toni",
        "Alessandro Del Piero", "Francesco Totti",
        "Alberto Gilardino", "Filippo Inzaghi",
        "Vincenzo Iaquinta",
    }
)
};

IEnumerable<string> players =
    worldCup2006Finalists.SelectMany(
        team => team.Players);

Print(players);
// ...

```

3.0

W danych wyjściowych generowanych przez kod z tego listingu imiona i nazwiska wszystkich graczy są wyświetlane w odrębnych wierszach w kolejności ich występowania w kodzie. Różnica między metodami `Select()` i `SelectMany()` polega na tym, że metoda `Select()` zwróciłaby tu dwa elementy (po jednym odpowiadającym każdemu elementowi z pierwotnej kolekcji). Metoda `Select()` może przeprowadzić projekcję i przekształcić wartości z pierwotnego typu na inny, ale nie zmienia liczby elementów. Na przykład wyrażenie `teams.Select(team => team.Players)` zwróci obiekt typu `IEnumerable<string[]>`.

Natomiast metoda `SelectMany()` iteruje po każdym elemencie określonym za pomocą wyrażenia lambda (czyli po tablicach pobranych wcześniej przy użyciu metody `Select()`) i pobiera każdy element do nowej kolekcji, zawierającej sumę wszystkich elementów z kolekcji podrzędnej. Zamiast dwóch tablic z graczami metoda `SelectMany()` generuje jedną kolekcję z wszystkimi elementami utworzoną w wyniku połączenia wszystkich pobranych tablic.

Inne standardowe operatory kwerend

Na listingu 15.25 przedstawiono kod, w którym wykorzystano prostsze interfejsy API dostępne w klasie `Enumerable`. Wynik działania kodu znajdziesz w danych wyjściowych 15.12.

Listing 15.25. Wywołania innych metod z klasy `System.Linq.Enumerable`

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

class Program
{
    static void Main()
    {
        I Enumerable<object> stuff =
            new object[] { new object(), 1, 3, 5, 7, 9,
                           "\rzecz\"", Guid.NewGuid() };
        Print("Zmienna stuff: {0}", stuff);
        I Enumerable<int> even = new int[] { 0, 2, 4, 6, 8 };
        Print("Parzyste liczby całkowite: {0}", even);

        I Enumerable<int> odd = stuff.OfType<int>();
        Print("Nieparzyste liczby całkowite: {0}", odd);

        I Enumerable<int> numbers = even.Union(odd);
        Print("Unia liczb parzystych i nieparzystych: {0}", numbers);

        Print("Unia z liczbami parzystymi: {0}", numbers.Union(even));
        Print("Dołaczanie liczb nieparzystych: {0}", numbers.Concat(odd));
        Print("Część wspólna z parzystymi: {0}",
              numbers.Intersect(even));
        Print("Niepowtarzające się wartości: {0}", numbers.Concat(odd).Distinct());
        if (!numbers.SequenceEqual(
            numbers.Concat(odd).Distinct()))
        {
            throw new Exception("Nieoczekiwana nierówność.");
        }
        else
        {
            Console.WriteLine(
                @"Kolekcja ""SequenceEquals"" +
                  " numbers.Concat(odd).Distinct())");
        }
        Print("Odwrócenie: {0}", numbers.Reverse());
        Print("Średnia: {0}", numbers.Average());
        Print("Suma: {0}", numbers.Sum());
        Print("Maksimum: {0}", numbers.Max());
        Print("Minimum: {0}", numbers.Min());
    }

    private static void Print<T>(
        string format, I Enumerable<T> items) =>
        where T: notnull =>
    Console.WriteLine(format, string.Join(
        ", ", items));
}
```

DANE WYJŚCIOWE 15.12.

```
Zmienna stuff: System.Object, 1, 3, 5, 7, 9, "rzecz"
24c24a41-ee05-41b9-958e-50dd12e3981e
Parzyste liczby całkowite: 0, 2, 4, 6, 8
Nieparzyste liczby całkowite: 1, 3, 5, 7, 9
Unia liczb parzystych i nieparzystych: 0, 2, 4, 6, 8, 1, 3, 5, 7, 9
Unia z liczbami parzystymi: 0, 2, 4, 6, 8, 1, 3, 5, 7, 9
Dołączenie liczb nieparzystych: 0, 2, 4, 6, 8, 1, 3, 5, 7, 9, 1, 3, 5, 7, 9
Część wspólna z liczbami parzystymi: 0, 2, 4, 6, 8
Niepowtarzające się wartości: 0, 2, 4, 6, 8, 1, 3, 5, 7, 9
Kolekcja "SequenceEquals" numbers.Concat(odd).Distinct()
Odwrócenie: 9, 7, 5, 3, 1, 8, 6, 4, 2, 0
Średnia: 4.5
Suma: 45
Maksimum: 9
Minimum: 0
```

Żaden z interfejsów API używanych na listingu 15.25 nie wymaga stosowania wyrażeń lambda. W tabelach 15.1 i 15.2 opisano wszystkie wywołane tu metody wraz z przykładami. Klasa `System.Linq.Enumerable` zawiera zestaw funkcji agregujących, które przetwarzają kolekcje i wyznaczają wyniki (te funkcje opisano w tabeli 15.2). Przykładową funkcją agregującą przedstawioną już w tym rozdziale jest `Count`.

Zauważ, że wszystkie metody wymienione w tabelach 15.1 i 15.2 są wykonywane z opóźnieniem.

Tabela 15.1. Proste standardowe operatory kwerend

Metoda	Opis
<code>OfType<T>()</code>	Tworzy dotyczącą kolekcji kwerendę, która zwraca tylko elementy wskazanego typu, podanego w parametrze w wywołaniu <code>OfType<T>()</code> .
<code>Union()</code>	Łączy dwie kolekcje, aby utworzyć nadzbiór wszystkich elementów z obu kolekcji. Końcowa kolekcja nie zawiera powtarzających się elementów, nawet jeśli początkowo dany element znajduje się w obu kolekcjach.
<code>Concat()</code>	Łączy dwie kolekcje, aby utworzyć nadzbiór obu kolekcji. Powtarzające się elementy nie są usuwane z wynikowej kolekcji. Metoda <code>Concat()</code> zachowuje kolejność elementów, dlatego scalenie kolekcji {A, B} i {C, D} daje w wyniku {A, B, C, D}.
<code>Intersect()</code>	Zwraca kolekcję elementów występujących w obu pierwotnych kolekcjach.
<code>Distinct()</code>	Odfiltrowuje z kolekcji powtarzające się elementy. W efekcie każdy element w wynikowej kolekcji jest unikatowy.
<code>SequenceEquals()</code>	Porównuje dwie kolekcje i zwraca wartość logiczną określającą, czy są one identyczne (z uwzględnieniem kolejności elementów). Jest to bardzo pomocne do sprawdzania, czy wyniki są zgodne z oczekiwaniemi.
<code>Reverse()</code>	Odwraca kolejność elementów, tak aby występowały w odwrotnym porządku w trakcie iterowania po kolekcji.

Tabela 15.2. Funkcje agregujące z klasy System.Linq.Enumerable

Metoda	Opis
Count()	Zwraca łączną liczbę elementów kolekcji.
Average()	Oblicza średnią wartość na podstawie liczbowego klucza.
Sum()	Oblicza sumę wartości z kolekcji liczb.
Max()	Okręsła maksymalną wartość w kolekcji liczb.
Min()	Okręsła minimalną wartość w kolekcji liczb.

ZAGADNIENIE DLA ZAAWANSOWANYCH

Klasa Queryable z metodami rozszerzającymi interfejs IQueryable<T>

IQueryable<T> to interfejs niemal identyczny z interfejsem IEnumerable<T>. Ponieważ IQueryable<T> dziedziczy po IEnumerable<T>, zawiera wszystkie bezpośrednio zadeklarowane składowe interfejsu IEnumerable<T> (na przykład metodę GetEnumator()). Metody rozszerzające nie są jednak dziedziczone, dlatego w interfejsie IQueryable<T> nie są dostępne żadne metody rozszerzające z klasy Enumerable. Istnieje jednak podobna klasa rozszerzająca System.Linq.Queryable, która dodaje do interfejsu IQueryable<T> prawie wszystkie metody dodawane przez klasę Enumerable do interfejsu IEnumerable<T>. Dlatego w obu przypadkach dostępny interfejs programistyczny jest bardzo podobny.

Wyjątkową cechą interfejsu IQueryable<T> jest to, że umożliwia stosowanie niestandardowych dostawców danych z technologii LINQ. Taki dostawca dzieli wyrażenia na ich części. Po podziale wyrażenie można przekształcić na inny język, zserializować na potrzeby zdalnego wykonania, wstrzyknąć za pomocą wzorca asynchronicznego wykonywania itd. Dostawcy z technologii LINQ umożliwiają dodanie mechanizmu przechwytywania instrukcji do standardowego interfejsu API używanego dla kolekcji. Dzięki temu można wstrzykiwać niemal dowolne funkcje związane z kwerendami i kolekcjami.

Dostawcy z technologii LINQ umożliwiają na przykład przekształcenie wyrażenia z kwerendą z języka C# na SQL i późniejsze wykonanie kwerendy w zdalnej bazie danych. Programista używający języka C# może przy tym korzystać z podstawowego obiektowego języka programowania i pozostawić przekształcanie instrukcji na SQL dostawcy z technologii LINQ. Dzięki tej technice w językach programowania można rozwiązać problem niezgodności impedancji między światem obiektowym a relacyjnymi bazami danych.

W interfejsie IQueryable<T> świadomość opóźnionego wykonywania instrukcji jest jeszcze ważniejsza niż gdzie indziej. Wyobraź sobie na przykład dostawcę z technologii LINQ, który zwraca dane z bazy. Zamiast pobierać dane z bazy niezależnie od kryteriów, można wykorzystać w wyrażeniu lambda implementację interfejsu IQueryable<T> zawierającą informacje kontekstowe (na przykład łańcuch znaków połączenia), ale nie same dane. Pobieranie danych następuje wtedy dopiero po wywołaniu metody GetEnumator() lub nawet po uruchomieniu metody MoveNext(). Jednak metoda GetEnumator() jest zwykle wywoływana niejawnie, na przykład w trakcie iterowania po kolekcji za pomocą pętli foreach lub w wyniku wywołania metody Count<T>(), Cast<T>() bądź innej metody z klasy Enumerable. Oczywiście wymaga to od programistów uważania na trudne do wykrycia i powtarzające się

wywołania kosztownych operacji, które mogą się pojawiać z powodu opóźnionego wykonywania instrukcji. Na przykład jeśli wywołanie metody `GetEnumerator()` jest związane z wywołaniem kierowanym przez sieć do bazy w środowisku rozproszonym, warto unikać przypadkowych powtarzających się wywołań w trakcie iteracji związanych z instrukcjami `Count()` i `foreach`.

Typy anonimowe w technologii LINQ

W języku C# 3.0 znacznie poprawiono możliwości w zakresie obsługi kolekcji elementów za pomocą technologii LINQ. Fantastyczne jest to, że w celu dodania obsługi nowego zaawansowanego interfejsu API wystarczyło wprowadzić w języku osiem usprawnień. Jednak te poprawki są bardzo ważnym czynnikiem sprawiającym, że wersja C# 3.0 okazała się tak znacznym usprawnieniem języka. Dwie spośród wspomnianych nowości to typy anonimowe i zmienne lokalne o niejawnie określonym typie. Jednak w wersji C# 7.0 typy anonimowe zostały „przyćmione” przez składnię dla krotek. W szóstym wydaniu tej książki wszystkie przykłady użycia technologii LINQ, w których wcześniej stosowano typy anonimowe, zostały zmodyfikowane i zamiast tych typów użyto krotek.

Co jednak zrobić, jeśli nie masz dostępu do wersji C# 7.0 (lub nowszej) albo pracujesz nad kodem napisanym w starszych wersjach? W dalszej części tego rozdziału omawiam typy anonimowe, dzięki czemu będziesz mógł je poznać. Jeżeli jednak nie zamierzasz pisać kodu w C# 6.0 ani pracować nad starszymi aplikacjami, możesz pominąć ten podrozdział.

Typy anonimowe

Typy anonimowe to typy danych deklarowane przez kompilator, a nie za pomocą opisanych w rozdziale 6. bezpośrednich definicji klas. Gdy kompilator natrafi na typ anonimowy, to (analogicznie jak po wykryciu funkcji anonimowych) stara się utworzyć na jego podstawie klasę i umożliwia korzystanie z niej tak, jakby była zadeklarowana jawnie. Deklarację takiego typu znajdziesz na listingu 15.26.

Listing 15.26. Zmienna lokalna o niejawnie określonym typie anonimowym

3.0

```
using System;

class Program
{
    static void Main()
    {
        var patent1 =
            new
            {
                Title = "Okulary dwuogniskowe",
                YearOfPublication = "1784"
            };
        var patent2 =
            new
            {
                Title = "Fonograf",
```

```

        YearOfPublication = "1877"
    };
var patent3 =
new
{
    patent1.Title,
    // Zmiana nazwy, aby określala znaczenie właściwości.
    Year = patent1.YearOfPublication
};

Console.WriteLine(
    $"{ patent1.Title } ({ patent1.YearOfPublication })");
Console.WriteLine(
    $"{ patent2.Title } ({ patent2.YearOfPublication })");

Console.WriteLine();
Console.WriteLine(patent1);
Console.WriteLine(patent2);

Console.WriteLine();
Console.WriteLine(patent3);
}
}

```

Wynik działania tego kodu pokazano w danych wyjściowych 15.13.

DANE WYJŚCIOWE 15.13.

```

Okulary dwuogniskowe (1784)
Fonograf (1784)

{ Title = Okulary dwuogniskowe, YearOfPublication = 1784 }
{ Title = Fonograf, YearOfPublication = 1877 }

{ Title = Okulary dwuogniskowe, Year = 1784 }

```

Typy anonimowe są mechanizmem języka C#; nie są one nowym rodzajem typów ze środowiska uruchomieniowego. Gdy kompilator natrafi na składnię typu anonimowego, generuje w kodzie CIL klasę o właściwościach odpowiadających nazwanym wartościami i typom danych z deklaracji typu anonimowego.

3.0

■ ■ ■ ZAGADNIENIE DLA POCZĄTKUJĄCYCH

Zmienne lokalne o niejawnie określonym typie (zmienne var)

Ponieważ typ anonimowy z definicji nie ma nazwy, nie da się zadeklarować zmiennej lokalnej z jawnie podanym typem anonimowym. Zamiast tego jako typ zmiennej lokalnej należy podać var. Jednak nie oznacza to, że zmienne o niejawnie określonym typie nie mają typu. Mają one ściśle określony typ wartości, które do nich przypisano. Jeśli do zmiennej z niejawnie określonym typem przypisana zostanie wartość typu anonimowego, w kodzie CIL w deklaracji tej zmiennej lokalnej użyty zostanie typ wygenerowany przez kompilator. Podobnie jeśli do zmiennej przypisana zostanie wartość typu string, w kodzie CIL typem danych zmiennej

będzie `string`. W wynikowym kodzie CIL nie ma znaczenia, czy używana jest zmienna z niejawnie określonym typem, do której przypisano wartość typu nianonimowego (na przykład `string`), czy też w deklaracji określono, że zmienna jest tego typu. Jeśli deklaracja to `string text = "To test...",` wynikowy kod CIL będzie identyczny jak w deklaracji z niejawnie określonym typem — `var text = "To test..."`.

Kompilator ustala typ zmiennych z niejawnie określonym typem na podstawie wyrażenia przypisywanego do zmiennej. Dla zmiennych lokalnych z jawnie określonym typem i inicjatorem (na przykład `string s = "witaj";`) kompilator najpierw ustala typ zmiennej `s` na podstawie typu zadeklarowanego po lewej stronie, a następnie sprawdza wyrażenie podane po prawej stronie i bada, czy można je przypisać do zmiennej danego typu. W zmiennej lokalnej o niejawnie określonym typie ten proces jest w pewnym sensie odwrócony. Najpierw kompilator analizuje wyrażenie podane po prawej stronie, by ustalić jego typ, a następnie następuje słowo `var` wykrytym typem.

Choć w języku C# typy anonimowe nie mają nazwy, są silnie typowane. Na przykład dostępne są właściwości z takich typów. Na listingu 15.26 w instrukcji `Console.WriteLine` znajdują się wywołania `patent1.Title` i `patent2.YearOfPublication`. Próby wywołania nieistniejących zmiennych spowodują błąd kompilatora. Nawet mechanizm IntelliSense w środowiskach IDE takich jak Visual Studio obsługuje typy anonimowe.

Staraj się rzadko stosować zmienne o niejawnie określonym typie. Oczywiście, gdy używasz typu anonimowego, nie możesz podać typu danych, dlatego potrzebna jest zmienna `var`. Jednak w sytuacjach, gdy używany jest nianonimowy typ danych, często lepiej podać go jawnie.

Należy się skoncentrować na tym, by można było łatwo zrozumieć działanie kodu, a także by kompilator mógł sprawdzać, czy zmienna jest oczekiwany typu. Jeśli chcesz osiągnąć te cele, zmienne lokalne o niejawnie określonym typie stosuj tylko wtedy, gdy typ wartości przypisywanej do takiej zmiennej jest oczywisty. Na przykład instrukcja `var items = new Dictionary<string, List<Account>>();` to zwięzły i czytelny kod. Natomiast gdy typ nie jest oczywisty (na przykład gdy do zmiennej przypisywana jest wartość zwracana przez metodę), programiści powinni preferować deklaracje zmiennych o jawnie podanym typie:

```
Dictionary<string, List<Account>> dictionary = GetAccounts();
```

3.0

Pobieranie danych do zmiennych typu anonimowego za pomocą technologii LINQ

Korzystając z typów anonimowych, można utworzyć kolekcję typu `IEnumerable<T>`, gdzie `T` to typ anonimowy (zobacz listing 15.27 i dane wyjściowe 15.14).

Listing 15.27. Projekcja z przekształceniem wartości na typ anonimowy

```
// ...
IEnumerable<string> fileList = Directory.EnumerateFiles(
    rootDirectory, searchPattern);
var items = fileList.Select(
    file =>
{
    FileInfo fileInfo = new FileInfo(file);
    return new
```

```
{
    FileName = fileInfo.Name,
    Size = fileInfo.Length
};

});

// ...
}
}
```

DANE WYJŚCIOWE 15.14.

```
{ FileName = AssemblyInfo.cs, Size = 1704 }
{ FileName = CodeAnalysisRules.xml, Size = 735 }
{ FileName = CustomDictionary.xml, Size = 199 }
{ FileName = EssentialCSharp.sln, Size = 40415 }
{ FileName = EssentialCSharp.suo, Size = 454656 }
{ FileName = EssentialCSharp.vsmdi, Size = 499 }
{ FileName = EssentialCSharp.vssscc, Size = 256 }
{ FileName = intelliTechture.ConsoleTester.dll, Size = 24576 }
{ FileName = intelliTechture.ConsoleTester.pdb, Size = 30208 }
```

W tych danych wyjściowych dotyczących zmiennych typu anonimowego automatycznie wyświetlane są nazwy i wartości właściwości. Odpowiada za to wygenerowana metoda `ToString()` powiązana z typem anonimowym.

Projekcje z użyciem metody `Select()` zapewniają bardzo dużo możliwości. Zobaczyłeś już, jak pionowo filtrować kolekcję (zmniejszając przy tym liczbę elementów w kolekcji) za pomocą standardowego operatora kwerend `Where()`. Przy użyciu standardowego operatora kwerend `Select()` można filtrować kolekcję poziomo (zmniejszając liczbę kolumn) lub całkowicie przekształcać dane. Dzięki obsłudze typów anonimowych można wywołać metodę `Select()` dla dowolnego obiektu i pobrać tylko te fragmenty pierwotnej kolekcji, które są potrzebne w bieżącym algorytmie. Nie trzeba nawet deklarować klasy do przechowywania tych fragmentów.

Więcej o typach anonimowych i zmiennych lokalnych o niejawnie określonym typie

3.0

Na listingu 15.26 nazwy składowych w typach anonimowych są identyfikowane jawnie na podstawie przypisania wartości do nazw (na przykład `Title = "Fonograf"`) w zmiennych `patent1` i `patent2`. Jednak jeśli przypisywana wartość to właściwość lub pole, można domyślnie wykorzystać nazwę przypisywanego pola lub właściwości, zamiast określać nazwę jawnie. Na przykład zmienna `patent3` jest zdefiniowana z wykorzystaniem właściwości o nazwie `Title`; nie ma tu przypisania do właściwości o jawnie określonej nazwie. W danych wyjściowych 15.3 widać, że nazwa wynikowej właściwości jest określana przez kompilator na podstawie właściwości, z której pochodzi wartość.

W zmiennych `patent1` i `patent2` używane są właściwości o tych samych nazwach i typach danych. Dlatego kompilator języka C# generuje tylko jeden typ danych dla dwóch deklaracji typów anonimowych użytych dla tych zmiennych. Zmienna `patent3` sprawia, że kompilator musi utworzyć drugi typ anonimowy, ponieważ nazwa właściwości reprezentującej rok przyznania patentu jest tu inna niż w zmiennych `patent1` i `patent2`. Ponadto jeśli kolejność właściwości w zmiennych `patent1` i `patent2` będzie różna, typy anonimowe użyte dla tych zmiennych nie będą ze sobą zgodne. Tak więc jeśli dwa typy anonimowe mają być ze sobą zgodne

w ramach danego podzespołu, muszą mieć właściwości o tych samych nazwach i typach danych, a właściwości te muszą być podane w tej samej kolejności. Jeśli te kryteria są spełnione, typy są zgodne nawet wtedy, jeśli utworzono je w różnych metodach lub klasach. Na listingu 15.28 przedstawiono typy niezgodne ze sobą.

Listing 15.28. Bezpieczeństwo ze względu na typ i niemodyfikowalność typów anonimowych

```
class Program
{
    static void Main()
    {
        var patent1 =
            new
            {
                Title = "Okulary dwuogniskowe",
                YearOfPublication = "1784"
            };

        var patent2 =
            new
            {
                YearOfPublication = "1877",
                Title = "Fonograf"
            };

        var patent3 =
            new
            {
                patent1.Title,
                Year = patent1.YearOfPublication
            };

        // BŁĄD: nie można niejawnie przeprowadzić konwersji
        // 'TypAnonimowyNr2' na 'TypAnonimowyNr1'.
        patent1 = patent2;
        // BŁĄD: nie można niejawnie przeprowadzić konwersji
        // 'TypAnonimowyNr3' na 'TypAnonimowyNr1'.
        patent1 = patent3;

        // BŁĄD: nie można przypisać wartości do właściwości lub indeksera
        // 'TypAnonimowyNr1.Title'; jest to składowa tylko do odczytu.
        patent1.Title = "Ser szwajcarski";
    }
}
```

3.0

Dwa pierwsze błędy kompilatora dowodzą tego, że typy nie są ze sobą zgodne, dlatego nie można dokonać konwersji z jednego z nich na drugi. Trzeci błąd kompilatora jest spowodowany próbą ponownego przypisania wartości do właściwości `Title`. Typy anonimowe są niemodyfikowalne, dlatego zmiana wartości właściwości w typie anonimowym skutkuje błędem kompilatora.

Choć na listingu 15.28 nie jest to pokazane, nie da się zadeklarować metody z parametrem o niejawnie określonym typie (`var`). Dlatego instancje typów anonimowych można przekazać poza metodę, w której je utworzono, tylko na dwa sposoby. Pierwszy dostępny

jest wtedy, jeśli parametr metody jest typu `object`. Instancję typu anonimowego można wtedy przekazać poza metodę, ponieważ nastąpi niejawna konwersja typu anonimowego. Druga technika polega na inferencji typu metody. Instancja typu anonimowego jest wówczas przekazywana jako parametr określający typ, który kompilator może ustalić w wyniku inferencji. Dlatego wywołanie `void Method<T>(T parameter)` w postaci `Function(patient1)` jest dozwolone, przy czym w metodzie `Function()` na parametrze `parameter` można wykonywać wyłącznie operacje dostępne w typie `object`.

Choć język C# umożliwia tworzenie typów anonimowych takich jak te pokazane na listingu 15.26, zwykle nie powinno się definiować ich w ten sposób. Typy anonimowe są niezwykle ważne ze względu na wprowadzoną w wersji C# 3.0 obsługę projekcji (na przykład przy łączeniu kolekcji). To zagadnienie opisano w dalszej części rozdziału. Zwykle jednak typy anonimowe należy definiować tam, gdzie są niezbędne, na przykład w celu złączania danych z wielu typów.

W czasie, gdy wprowadzono typy anonimowe, były one przełomową techniką, która rozwiązywała ważny problem — pozwalała szybko deklarować tymczasowe typy bez „ceregieli” związanych z koniecznością deklarowania kompletnego typu. Technika ta miała jednak kilka opisanych wcześniej wad. Na szczęście wprowadzone w wersji C# 7.0 krotki są pozbawione tych wad i pozwalają zrezygnować z typów anonimowych. Oto zalety krotek w porównaniu z typami anonimowymi:

- Krotki zapewniają nazwany typ, który można stosować wszędzie tam, gdzie inne typy, w tym w deklaracjach i jako parametry określające typ.
- Krotki są dostępne także poza metodą, w której zostały utworzone.
- Krotki pozwalają uniknąć zaśmiecania programu typami, które są generowane, ale rzadko używane.

Jedną z różnic między krotkami a typami anonimowymi jest to, że typy anonimowe są referencyjne, a krotki są typami bezpośrednimi. To, które z tych podejść jest korzystniejsze, zależy od sytuacji. Jeśli krotki są często kopowane i zajmują więcej niż 128 bitów, zapewne lepiej byłoby użyć typu referencyjnego. W innych scenariuszach krotki są zwykle bardziej wydajne od typów anonimowych i stanowią lepszy domyślny wybór.

Początek
7.0Koniec
7.0

3.0

■ ZAGADNIENIE DLA ZAAWANSOWANYCH

Generowanie typów anonimowych

Choć metoda `Console.WriteLine()` korzysta z metody `ToString()`, zauważ, że na listingu 15.26 dane wyjściowe z metody `Console.WriteLine()` nie powstały za pomocą domyślnej metody `ToString()` (która wyświetla pełną nazwę typu danych). Zamiast tego dane wyjściowe to lista par `NazwaWłaściwości = wartość`. Wyświetlana jest jedna taka para dla każdej właściwości typu anonimowego. Dzieje się tak, ponieważ w trakcie generowania kodu typu anonimowego kompilator przesyła metodę `ToString()`, aby sformatować dane wyjściowe w przedstawiony sposób. W wygenerowanym typie znajdują się też nowe wersje przesyłanych metod `Equals()` i `GetHashCode()`.

Sama implementacja metody `ToString()` to ważny powód, dla którego zmiana kolejności właściwości powoduje wygenerowanie nowego typu danych. Jeśli dwa odrębne typy anonimowe, które mogą się znajdować w zupełnie odmiennych typach nadrzędnych i przestrzeniach nazw, zostałyby ujednolicone, a następnie programista zmieniłby kolejność właściwości, modyfikacja tej kolejności w jednym typie miałaby zauważalny i prawdopodobnie nieakceptowalny wpływ na wyniki zwracane przez metodę `ToString()` z drugiego typu. Ponadto w czasie wykonywania programu można za pomocą mechanizmu refleksji sprawdzić składowe typu, a nawet dynamicznie wywołać jedną z jego składowych (określając ją właśnie w czasie wykonywania programu). Zmiana kolejności składowych w dwóch pozornie identycznych typach może wtedy skutkować nieoczekiwany wynikami. Aby uniknąć takich problemów, projektanci języka C# zdecydowali się generować dwa odmienne typy, gdy kolejność właściwości jest różna.

ZAGADNIENIE DLA ZAAWANSOWANYCH

Inicjatory kolekcji a typy anonimowe

Nie można utworzyć inicjatora kolekcji dla typu anonimowego, ponieważ taki inicjator wymaga wywołania konstruktora, a w typie anonimowym nie można podać nazwy konstruktora. Rozwiązaniem jest zdefiniowanie metody takiej jak `static List<T> CreateList<T>(T t) { return new List<T>(); }`. Wnioskowanie typów pozwala wywnioskować parametr określający typ zamiast go podawać. Dlatego ta technika umożliwia tworzenie kolekcji typów anonimowych.

Inny sposób inicjowania kolekcji obiektów typów anonimowych polega na użyciu inicjatora tablicy. Nie można tu podać typu danych w konstruktorze, jednak składnia inicjowania tablic pozwala zastosować inicjator tablicy obiektów typu anonimowego z użyciem operatora `new[]` (zobacz listing 15.29).

Listing 15.29. Inicjowanie tablic obiektów typu anonimowego

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

class Program
{
    static void Main()
    {
        new
        {
            TeamName = "France",
            Players = new string[]
            {
                "Fabien Barthez", "Gregory Coupet",
                "Mickael Landreau", "Eric Abidal",
                // ...
            },
        new
        {
            TeamName = "Italy",
            Players = new string[]
```

```
        {
            "Gianluigi Buffon", "Angelo Peruzzi",
            "Marco Amelia", "Cristian Zaccardo",
            // ...
        }
    };

    Print(worldCup2006Finalists);
}

private static void Print<T>(IEnumerable<T> items)
{
    foreach (T item in items)
    {
        Console.WriteLine(item);
    }
}
}
```

Wynikową zmienną jest tablica obiektów typu anonimowego; musi być ona jednorodna, ponieważ jest to cecha tablic.

Podsumowanie

W tym rozdziale opisano wewnętrzne mechanizmy działania pętli `foreach` oraz wyjaśniono, które interfejsy są potrzebne do korzystania z tej pętli. Programiści często chcą filtrować kolekcje, tak aby zmniejszyć liczbę elementów, i przeprowadzać projekcję, polegającą na przekształcaniu elementów. W tym kontekście opisano szczegółowo, jak posługiwać się standardowymi operatorami kwerend (w technologii LINQ wprowadzono metody rozszerzające kolekcje; są one dostępne w klasie `System.Linq.Enumerable`) do manipulowania kolekcjami.

We wprowadzeniu do standardowych operatorów kwerend szczegółowo opisano proces opóźnionego wykonywania instrukcji i zwróciło uwagę na to, że programiści powinni stać się unikać przypadkowego ponownego wykonywania wyrażenia z powodu trudnych do wykrycia wywołań związanych z iteracją po kolekcji. Opóźnione wykonywanie instrukcji i wynikające z tego niejawne wykonywanie standardowych operatorów kwerend mają istotny wpływ na wydajność kodu — zwłaszcza gdy wykonywanie kwerendy jest kosztowne. Programiści powinni traktować obiekt reprezentujący kwerendę jak kwerendę, a nie jak wyniki. Dlatego należy się spodziewać, że kwerenda zostanie ponownie wykonana, nawet jeśli została już wcześniej przetworzona. Obiekt kwerendy nie potrafi określić, że wyniki będą takie same jak po wcześniejszym jej wykonaniu.

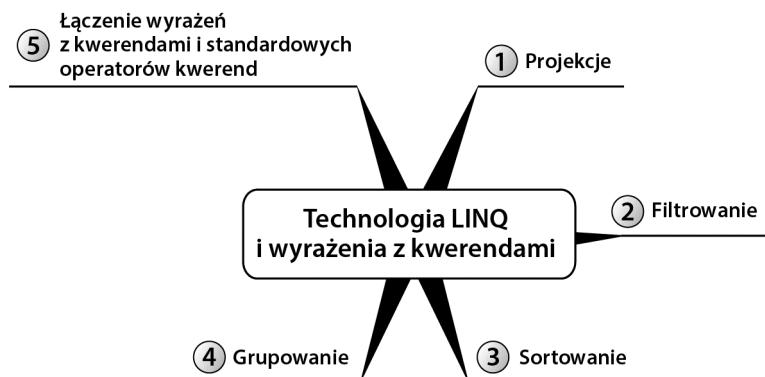
Listing 15.23 umieszczono w zagadnienniu dla zaawansowanych, ponieważ wywoływanie wielu standardowych operatorów kwerend jeden po drugim jest skomplikowane. Choć często potrzebne jest wykonywanie podobnych operacji, nie trzeba bezpośrednio polegać na standardowych operatorach kwerend. Od wersji C# 3.0 dostępne są wyrażenia z kwerendami, czyli podobna do SQL-a składnia pozwalająca manipulować kolekcjami za pomocą łatwego do napisania i czytelnego kodu. Ten mechanizm opisano w następnym rozdziale.

W końcowej części rozdziału omówiono szczegółowo typy anonimowe i wyjaśniono, dlaczego w wersji C# 7.0 i nowszych zaleca się stosowanie krotek zamiast takich typów.

■ ■ ■ 16 ■ ■ ■

Technologia LINQ i wyrażenia z kwerendami

W KOŃCOWEJ CZĘŚCI ROZDZIAŁU 15. pokazano kwerendę opartą na standardowych operatorach kwerend (`GroupJoin()`, `SelectMany()` i `Distinct()`). W efekcie powstała instrukcja zajmująca wiele wierszy. Jest ona bardziej skomplikowana i trudniejsza do zrozumienia niż instrukcje pisane za pomocą mechanizmów z wcześniejszych wersji języka C#. We współczesnych językach, które manipulują bogatymi zbiorami danych, często takie złożone kwerendy są potrzebne. Dlatego byłoby wygodnie, gdyby język ułatwiał czytanie takich instrukcji. Języki służące do tworzenia kwerend, na przykład SQL, ułatwiają czytanie i rozumienie kwerend, ale nie mają tak bogatych możliwości jak C#. To dlatego projektanci języka C# dodali w wersji C# 3.0 składnię **wyrażeń z kwerendami**. Za jej pomocą wiele wyrażeń opartych na standardowych operatorach kwerend można przekształcić na bardziej czytelny kod, przypominający kod w SQL-u.



3.0

Ten rozdział zawiera wprowadzenie do wyrażeń z kwerendami. Zobaczysz, jak przy ich użyciu zapisać wiele kwerend z rozdziału 15.

Wprowadzenie do wyrażeń z kwerendami

Dwie spośród najczęściej wykonywanych przez programistów operacji to **filtrowanie kolekcji** (w celu wyeliminowania niepotrzebnych elementów) i **przeprowadzanie projekcji** (aby przekształcić elementy na inną postać). Na przykład jeśli istnieje kolekcja plików, można ją przefiltrować, by otrzymać nową kolekcję obejmującą tylko pliki o rozszerzeniu „.cs” lub tylko pliki mające więcej niż milion bajtów. Można też przeprowadzić projekcję kolekcji plików, by uzyskać nową kolekcję obejmującą ścieżki do katalogów, w których znajdują się pliki, i ilość miejsca zajmowanego przez te katalogi. Wyrażenia z kwerendami zapewniają prostą składnię do wykonywania obu tych standardowych operacji. Na listingu 16.1 przedstawiono wyrażenie z kwerendą filtrujące kolekcję łańcuchów znaków. Wynik działania tego kodu znajdziesz w danych wyjściowych 16.1.

Listing 16.1. Proste wyrażenie z kwerendą

```
static public class CSharp
{
    static public readonly string[] Keywords =
        ReadOnlyCollection<string> Keywords = new ReadOnlyCollection<string>(
    new string[]
    {
        "abstract", "add*", "alias*", "as", "ascending*",
        "async*", "await*", "base", "bool", "break",
        "by*", "byte", "case", "catch", "char", "checked",
        "class", "const", "continue", "decimal", "default",
        "delegate", "descending*", "do", "double",
        "dynamic*", "else", "enum", "event", "equals*",
        "explicit", "extern", "false", "finally", "fixed",
        "from*", "float", "for", "foreach", "get*", "global*",
        "group*", "goto", "if", "implicit", "in", "int",
        "into*", "interface", "internal", "is", "lock", "long",
        "join*", "let*", "nameof*", "namespace", "new", "nonnull*",
        "null", "object", "on*", "operator", "orderby*", "out",
        "override", "params", "partial*", "private", "protected",
        "public", "readonly", "ref", "remove*", "return", "sbyte",
        "sealed", "select*", "set*", "short", "sizeof",
        "stackalloc", "static", "string", "struct", "switch",
        "this", "throw", "true", "try", "typeof", "uint", "ulong",
        "unsafe", "ushort", "using", "value*", "var*", "virtual",
        "unchecked", "void", "volatile", "where*", "while", "yield*");
    });

    using System;
    using System.Collections.Generic;
    using System.Linq;

    private static void ShowContextualKeywords1()
    {
        IEnumerable<string> selection =
            from word in CSharp.Keywords
            where !word.Contains('*')
            select word;

        foreach (string keyword in selection)
```

```

    {
        Console.Write(keyword + " ");
    }
}

// ...

```

DANE WYJŚCIOWE 16.1.

```

abstract as base bool break byte case catch char checked class const
continue decimal default delegate do double else enum event explicit
extern false finally fixed float for foreach goto if implicit in int
interface internal is lock long namespace new null object operator out
override params private protected public readonly ref return sbyte
sealed short sizeof stackalloc static string struct switch this throw
true try typeof uint ulong unchecked ushort using virtual void
volatile while

```

W tym wyrażeniu do zmiennej `selection` jest przypisywana kolekcja słów zarezerwowanych z języka C#. Wyrażenie z kwerendą z tego przykładu obejmuje klauzulę `where`, która odfiltrowuje kontekstowe słowa kluczowe.

Wyrażenie z kwerendą zwykle rozpoczyna się od klauzuli `from`, a kończy klauzulą `select` lub `group`. Te klauzule są tworzone za pomocą kontekstowych słów kluczowych `from`, `select` i `group`. Identyfikator `word` w klauzuli `from` to **zmienna zakresowa** (ang. *range variable*). Reprezentuje ona każdy element z kolekcji, podobnie jak robi to zmienna pętli w pętlach `foreach`.

Programiści znający język SQL zauważą, że wyrażenia z kwerendą mają składnię podobną do języka SQL. Jest to celowy efekt. Technologia LINQ ma być łatwa do opanowania dla programistów, którzy znają już SQL. Jednak między składniami występują pewne oczywiste różnice. Pierwszą z nich, którą zauważą doświadczeni użytkownicy języka SQL, jest to, że w przedstawionym wcześniej wyrażeniu z kwerendą z języka C# klauzule są podane w następującej kolejności: `from`, `where`, `select`. W analogicznej kwerendzie w SQL-u najpierw znajdziesz się klauzula `SELECT`, potem klauzula `FROM`, a na końcu klauzula `WHERE`.

Jednym z powodów zmiany kolejności klauzul jest umożliwienie używania mechanizmu IntelliSense. Jest to mechanizm środowiska IDE, polegający na tym, że edytor wyświetla przydatne elementy interfejsu użytkownika, na przykład listy rozwijane z opisem składowych danego obiektu. Ponieważ klauzula `from` występuje jako pierwsza i określa tablicę łańcuchów znaków `Keywords` jako źródło danych, edytor kodu może wywnioskować, że zmienna zakresowa `word` jest typu `string`. Dlatego gdy w trakcie wprowadzania kodu w edytorze dotrzesz do kropki po słowie `word`, edytor wyświetli tylko składowe typu `string`.

Gdyby klauzula `from` występowała po słowie `select` (jak ma to miejsce w SQL-u), w trakcie wpisywania kwerendy w edytorze nie byłoby wiadomo, jaki jest typ danych zmiennej `word`. Dlatego wyświetlenie listy składowych tej zmiennej byłoby niemożliwe. Na przykład na lisingu 16.1 nie można było przewidzieć, że metoda `Contains()` to jedna z dostępnych składowych zmiennej `word`.

Kolejność klauzul w wyrażeniach z kwerendą w języku C# lepiej odpowiada też logicznemu porządkowi przeprowadzania operacji. Gdy kwerenda jest przetwarzana, najpierw określana jest kolekcja (opisana w klauzuli `from`), potem kod odfiltrowuje niepotrzebne elementy (w klauzuli `where`), a w ostatnim kroku wskazywane są pożądane elementy (za pomocą klauzuli `select`).

Ponadto kolejność klauzul w języku C# sprawia, że reguły określające, gdzie zmienne (zakresowe) znajdują się w zasięgu, są w dużej części spójne z regułami określania zasięgu zmiennych lokalnych. Na przykład zmienną zakresową trzeba zadeklarować w klauzuli (zwykle jest to klauzula `from`), a dopiero potem można jej używać. Podobnie zmienną lokalną przed użyciem należy zadeklarować.

Projekcja

Wynik zwracany przez wyrażenie z kwerendą to kolekcja typu `IEnumerable<T>` lub `IQueryable<T>`¹. Typ `T` jest ustalany na podstawie klauzuli `select` lub `group by`. Na listingu 16.1 kompilator potrafi określić, że kolekcja `Keywords` jest typu `string[]`, a ten można przekształcić na typ `IEnumerable<string>`. Z tego można wywnioskować, że zmienna `word` jest typu `string`. Kwerenda kończy się członem `select word`, co oznacza, że wynikiem wyrażenia z kwerendą musi być kolekcja łańcuchów znaków. Dlatego typ omawianego wyrażenia z kwerendą to `IEnumerable<string>`.

Tu dane wejściowe i wyjściowe kwerendy to kolekcje łańcuchów znaków. Jednak typ wyjściowy może być odmienny od wejściowego, jeśli wyrażenie w klauzuli `select` ma typ różny od pierwotnego. Przyjrzyj się wyrażeniu z kwerendą z listingu 16.2 i powiązanym danym wyjściowym 16.2.

Listing 16.2. Projekcja oparta na wyrażeniu z kwerendą

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

// ...

static void List1(string rootDirectory, string searchPattern)
{
    I Enumerable<string> fileNames = Directory.GetFiles(
        rootDirectory, searchPattern);
    I Enumerable<FileInfo> fileInfo =
        from fileName in fileNames
        select new FileInfo(fileName);

    foreach (FileInfo fileInfo in fileInfo)
    {
        Console.WriteLine(
            $"{fileInfo.Name} ({fileInfo.LastWriteTime})");
    }
}
```

3.0

¹ Wynikiem wyrażenia z kwerendą w praktyce jest prawie zawsze obiekt typu `IEnumerable<T>` lub typu pochodnego od niego. Jednak dozwolone, choć mało zrozumiałe, jest tworzenie wyrażeń z kwerendą zwracających wartość innego typu. W języku nie ma *wymogu*, by wynik wyrażenia z kwerendą można było przekształcić na typ `IEnumerable<T>`.

DANE WYJŚCIOWE 16.2.

```
Account.cs (11/22/2011 11:56:11 AM)
Bill.cs (8/10/2011 9:33:55 PM)
Contact.cs (8/19/2011 11:40:30 PM)
Customer.cs (11/17/2011 2:02:52 AM)
Employee.cs (8/17/2011 1:33:22 AM)
Person.cs (10/22/2011 10:00:03 PM)
```

To wyrażenie z kwerendą zwraca wartość typu `IEnumerable<FileInfo>`, a nie typu `IEnumerable<string>` (jest to typ wartości zwracanej przez metodę `Directory.GetFiles()`). Klauzula `select` w przedstawionym wyrażeniu z kwerendą umożliwia zmianę typu danych wartości pobranych w wyrażeniu z klauzuli `from`.

W tym kodzie wybrano typ `FileInfo`, ponieważ obejmuje on dwa pola potrzebne w danych wyjściowych — pole z nazwą pliku i pole z czasem ostatniego zapisu. Jeśli potrzebujesz informacji, które nie są zapisywane w obiektach typu `FileInfo`, możliwe, że nie znajdziesz tak dogodnego typu. Wtedy krotki (lub, w wersjach starszych niż C# 7.0, typy anonimowe) pozwalają na wygodne i zwięzłe przeprowadzanie projekcji w celu uzyskania potrzebnych danych. Nie trzeba przy tym wyszukiwać ani tworzyć jawnego typu. Ten scenariusz był głównym czynnikiem, który doprowadził do dodania do języka typów anonimowych. Kod z listingu 16.3 generuje podobne dane jak kod z listingu 16.2, ale tym razem zamiast typu `FileInfo` wykorzystano składnię dla krotek.

Listing 16.3. Krotki w wyrażeniach z kwerendą

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

// ...

static void List2(string rootDirectory, string searchPattern)
{
    var fileNames = Directory.EnumerateFiles(
        rootDirectory, searchPattern)
    var fileResults =
        from fileName in fileNames
        select
        (
            Name: fileName,
            LastWriteTime: File.GetLastWriteTime(fileName)
        );
    foreach (var fileResult in fileResults)
    {
        Console.WriteLine(
            $"{ fileResult.Name } ({ fileResult.LastWriteTime })");
    }
}
```

3.0

W tym przykładzie kwerenda za pomocą projekcji zwraca tylko nazwę pliku i czas ostatniego zapisu. Projekcja taka jak na listingu 16.3 nie sprawia wielkiej różnicy, gdy używane obiekty są niewielkie (na przykład typu `FileInfo`). Jednak pozioma projekcja, pozwalająca odfiltrować część danych powiązanych z każdym elementem kolekcji, daje bardzo duże możliwości, gdy ilość danych jest duża, a ich pobieranie (na przykład z innego komputera przez internet) — kosztowne. Zamiast w momencie wykonywania kwerendy pobierać wszystkie dane, można więc za pomocą krotek (lub, w wersjach starszych niż C# 7.0, typów anonimowych) zapisać i pobrać do kolekcji tylko potrzebne informacje.

Wyobraź sobie dużą bazę danych z tabelami mającymi po 30 lub więcej kolumn. Gdyby nie krotki, programiści musieliby korzystać z obiektów zawierających zbędne informacje lub definiować małe, wyspecjalizowane klasy przydatne tylko do przechowywania określonych danych. Krotki umożliwiają definiowanie za pomocą kompilatora typów zawierających tylko dane potrzebne w konkretnej sytuacji. W innych scenariuszach można przeprowadzić inną projekcję i pobrać jedynie potrzebne w danym momencie właściwości.

■ ZAGADNIENIE DLA POCZĄTKUJĄCYCH

Opóźnione wykonywanie wyrażeń z kwerendą

Kwerendy pisane w postaci wyrażeń z kwerendą są wykonywane z opóźnieniem (podobnie jak kwerendy z rozdziału 15.). Wróć teraz do przypisania obiektu kwerendy do zmiennej `selection` z listingu 16.1. Utworzenie kwerendy i przypisanie jej do zmiennej nie powoduje wykonania tej kwerendy. Na tym etapie budowany jest tylko obiekt reprezentujący kwerendę. Metoda `word.Contains("")` nie jest wywoływana w momencie tworzenia obiektu kwerendy. Wyrażenie z kwerendą zapisuje kryteria selekcji (określone za pomocą zmiennej `selection`) używane w trakcie iterowania po kolekcji.

Aby lepiej to zrozumieć, przyjrzyj się listingowi 16.4 i powiązanym z nim danym wyjaśnionym 16.3.

Listing 16.4. Opóźnione wykonywanie wyrażeń z kwerendą (przykład nr 1)

3.0

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

// ...

private static void ShowContextualKeywords2()
{
    IEnumerable<string> selection = from word in Keywords
                                      where IsKeyword(word)
                                      select word;
    Console.WriteLine("Kwerenda została utworzona.");
    foreach (string keyword in selection)
    {
        // Ten kod nie generuje spacji.
        Console.Write(keyword);
    }
}
```

```
// Efekt uboczny w postaci wyświetlania danych wyjściowych w konsoli
// jest generowany za pomocą predykatu. Pozwala to zilustrować opóźnione wykonywanie.
// W kodzie produkcyjnym należy unikać predykatów powodujących efekty uboczne.
private static bool IsKeyword(string word)
{
    if (word.Contains('*'))
    {
        Console.Write(" ");
        return true;
    }
    else
    {
        return false;
    }
}
// ...
```

DANE WYJŚCIOWE 16.3.

Kwerenda została utworzona.

```
add* alias* ascending* async* await* by* descending* dynamic*
equals* from* get* global* group* into* join* let* nameof* nonnull*
on* orderby* partial* remove* select* set* value* var* where* yield*
```

Na listingu 16.4 kod z pętli foreach nie wyświetla spacji. Efekt uboczny, czyli wyświetlanie spacji w trakcie sprawdzania predykatu IsKeyword(), zachodzi w trakcie przetwarzania kwerendy, a nie w momencie jej tworzenia. Dlatego choć zmienna selection reprezentuje kolekcję (jest przecież typu IEnumerable<T>), w momencie przypisania całego kod po klauzuli from wyznacza kryteria pobierania danych. Jednak kryteria te są uwzględniane dopiero po rozpoczęciu iterowania po zmiennej selection.

Przyjrzyj się teraz drugiemu przykładowi (listing 16.5 i dane wyjściowe 16.4).

Listing 16.5. Opóźnione wykonywanie wyrażeń z kwerendą (przykład nr 2)

```
using System;
using System.Collections.Generic;
using System.Linq;
// ...
private static void CountContextualKeywords()
{
    int delegateInvocations = 0;
    string func(string text)
    {
        delegateInvocations++;
        return text;
    };
    IEnumerable<string> selection =
        from keyword in CSharp.Keywords
        where keyword.Contains('*')
        select func(keyword);
```

```

Console.WriteLine(
    $"1. delegateInvocations={ delegateInvocations }");

// Wywołanie metody Count powinno spowodować wykonanie funkcji func raz
// dla każdego pobranego elementu.
Console.WriteLine(
    $"2. Kontekstowe słowo kluczowe Count={selection.Count() }");

Console.WriteLine(
    $"3. delegateInvocations={ delegateInvocations }");

// Wywołanie metody Count powinno spowodować wykonanie funkcji func raz
// dla każdego pobranego elementu.
Console.WriteLine(
    $"4. Kontekstowe słowo kluczowe Count={ selection.Count() }");

Console.WriteLine(
    $"5. delegateInvocations={ delegateInvocations }");

// Zapisanie wartości w pamięci podręcznej, tak by późniejsze wywołania Count
// nie skutkowały kolejnym uruchomieniem kwerendy.
List<string> selectionCache = selection.ToList();

Console.WriteLine(
    $"6. delegateInvocations={ delegateInvocations }");

// Pobranie liczby elementów z kolekcji zapisanej w pamięci podręcznej.
Console.WriteLine(
    $"7. selectionCache count={ selectionCache.Count() }");

Console.WriteLine(
    $"8. delegateInvocations={ delegateInvocations }");
}

// ...

```

DANE WYJŚCIOWE 16.4.

3.0

1. delegateInvocations=0
2. Kontekstowe słowo kluczowe Count=28
3. delegateInvocations=28
4. Kontekstowe słowo kluczowe Count=28
5. delegateInvocations=56
6. delegateInvocations=84
7. selectionCache count=28
8. delegateInvocations=84

Zamiast definiować odrębną metodę, na listingu 16.5 wykorzystano wyrażenie lambda, które zlicza wykonania metody.

Dwie rzeczy w tych danych wyjściowych są warte uwagi. Po pierwsze, zauważ, że po przypisaniu wyrażenia do zmiennej `selection` zmienna `delegateInvocations` jest równa zero. Tak więc na etapie przypisania wyrażenia do zmiennej `selection` nie jest wykonywana iteracja po kolekcji `Keywords`. Gdyby kolekcja `Keywords` była właściwością, zostałaby wywołana. Dzieje się tak, ponieważ klauzula `from` jest wykonywana na etapie przypisania. Jednak projekcja, filtrowanie, a także cały kod po klauzuli `from` są wykonywane dopiero wtedy, gdy

program zaczyna iterować po wartościach ze zmiennej `selection`. Dlatego na etapie przypisania zmiennej `selection` lepiej byłoby nazywać kwerendą.

Jednak po wywołaniu metody `Count()` określenia takie jak *selekcja* lub *elementy* (oznaczające kontener bądź kolekcję) stają się właściwsze, ponieważ program zaczyna zliczać elementy kolekcji. Zmienna `selection` pełni więc dwie funkcje — zachowuje informacje o kwerendzie i jest kontenerem, z którego można pobierać dane.

Drugą ważną cechą, którą warto zauważać, jest to, że dwukrotne wywołanie metody `Count()` sprawia, iż delegat `func` jest wykonywany raz dla każdego pobranego elementu. Ponieważ zmienna `selection` działa zarówno jak kwerenda, jak i jak kolekcja, zażądanie liczby elementów wymaga ponownego wykonania kwerendy w ramach iterowania po kolekcji `IEnumerable<string>`, do której zmienna `selection` prowadzi, i zliczenia elementów. Komplikator języka C# nie wie, czy ktoś zmodyfikował łańcuchy znaków w tablicy (przez co liczba elementów mogła się zmienić), dlatego za każdym razem zliczanie trzeba przeprowadzić od nowa, aby zagwarantować, że wynik jest prawidłowy i aktualny. Podobnie pętla `foreach` dotycząca zmiennej `selection` powoduje wywołanie delegata `func` dla każdego elementu. To samo jest prawdą dla wszystkich innych metod rozszerzających udostępnianych za pomocą klasy `System.Linq.Enumerable`.

ZAGADNIENIE DLA ZAAWANSOWANYCH

Implementacja opóźnionego wykonywania

Opóźnione wykonywanie jest zaimplementowane za pomocą delegatów i drzew wyrażeń. Delegat umożliwia utworzenie i stosowanie referencji do metody zawierającej wyrażenie, które później można wykonać. Drzewo wyrażeń umożliwia tworzenie i stosowanie informacji o wyrażeniu. Te informacje można później sprawdzić i manipulować nimi.

Na listingu 16.5 wyrażenie z predykatem w klauzuli `where` i wyrażenie z projekcją w klauzuli `select` są przekształcane przez kompilator na lambdy w postaci wyrażeń. Później te lambdy są przetwarzane w delegaty. Wynik wyrażenia z kwerendą to obiekt zawierający referencje do tych delegatów. Dopiero wtedy, gdy kod iteruje po wynikach kwerendy, obiekt kwerendy wykonuje kod delegatów.

Filtrowanie

Na listingu 16.1 znajduje się klauzula `where`, która filtryuje zarezerwowane słowa kluczowe i usuwa kontekstowe słowa kluczowe. Ta klauzula `where` filtryuje kolekcję pionowo. Jeśli wyobrażisz sobie, że kolekcja to pionowa lista elementów, klauzula `where` pozwala skrócić tę listę, tak by kolekcja zawierała mniej elementów. Kryteria filtrowania są zapisane za pomocą **predykatu** — wyrażenia lambda zwracającego wartość typu `bool`. Predykaty to na przykład `word.Contains()` (tak jak na listingu 16.1) lub `File.GetLastWriteTime(file) < DateTime.Now.AddMonths(-1)`. To ostatnie wyrażenie zastosowano na listingu 16.6. Wynik działania kodu z tego listingu znajdziesz w danych wyjściowych 16.5.

3.0

Listing 16.6. Wyrażenie z kwerendą filtrującą dane w klauzuli `where`

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

```
// ...

static void FindMonthOldFiles(
    string rootDirectory, string searchPattern)
{
    IEnumerable<FileInfo> files =
        from fileName in Directory.EnumerateFiles(
            rootDirectory, searchPattern)
        where File.GetLastWriteTime(fileName) <
            DateTime.Now.AddMonths(-1)
        select new FileInfo(fileName);

    foreach (FileInfo file in files)
    {
        // W celu uproszczenia przyjmij, że bieżący katalog jest
        // podkatalogiem katalogu rootDirectory.
        string relativePath = file.FullName.Substring(
            Environment.CurrentDirectory.Length);
        Console.WriteLine(
            $"{relativePath} ({file.LastWriteTime})");
    }
}

// ...

```

DANE WYJŚCIOWE 16.5.

```
.\TestData\Bill.cs (8/10/2011 9:33:55 PM)
.\TestData>Contact.cs (8/19/2011 11:40:30 PM)
.\TestData\Employee.cs (8/17/2011 1:33:22 AM)
.\TestData\Person.cs (10/22/2011 10:00:03 PM)
```

Sortowanie

Aby uporządkować elementy za pomocą wyrażenia z kwerendą, zastosuj klauzulę orderby, przedstawioną na listingu 16.7.

3.0

Listing 16.7. Sortowanie w wyrażeniu z kwerendą za pomocą klauzuli orderby

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

// ...
static void ListByFileSize(
    string rootDirectory, string searchPattern)
{
    IEnumerable<string> fileNames =
        from fileName in Directory.EnumerateFiles(
            rootDirectory, searchPattern)
        orderby (new FileInfo(fileName)).Length descending,
            fileName
        select fileName;
```

```
foreach (string fileName in fileNames)
{
    Console.WriteLine(fileName);
}
// ...
```

Na listingu 16.7 zastosowano klauzulę `orderby` do posortowania plików zwróconych przez metodę `Directory.GetFiles()`. Pliki najpierw są sortowane malejąco według wielkości, a następnie rosnąco według nazw. Kryteria sortowania są rozdzielone przecinkami. W pierwszej kolejności elementy są porządkowane według wielkości, a jeśli ta jest równa, elementy są sortowane na podstawie nazw. Człony `ascending` i `descending` to kontekstowe słowa kluczowe określające porządek sortowania. Określanie porządku jest opcjonalne. Jeśli go pominiesz (tak jak tu przy polu `filename`), domyślnie używane jest słowo kluczowe `ascending`.

Klauzula `let`

Na listingu 16.8 znajdziesz kwerendę bardzo podobną do tej z listingu 16.7, przy czym tu określający typ argument w typie `IEnumerable<T>` ma wartość `FileInfo`. Zauważ, że nowa kwerenda prowadzi do problemu — obiekt typu `FileInfo` jest niepotrzebnie tworzony dwukrotnie (w klauzuli `orderby` i w klauzuli `select`).

Listing 16.8. Projekcja kolekcji `FileInfo` i sortowanie danych według wielkości plików

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

// ...
static void ListByFileSize2(
    string rootDirectory, string searchPattern)
{
    IEnumerable<FileInfo> files =
        from fileName in Directory.EnumerateFiles(
            rootDirectory, searchPattern)
        orderby new FileInfo(fileName).Length, fileName
        select new FileInfo(fileName);

    foreach (FileInfo file in files)
    {
        // Dla uproszczenia przyjmij, że bieżący katalog to
        // podkatalog katalogu rootDirectory.
        string relativePath = file.FullName.Substring(
            Environment.CurrentDirectory.Length);
        Console.WriteLine(
            $"{relativePath}({{ file.Length }})");
    }
}
// ...
```

Niestety, choć zwracany wynik jest prawidłowy, kod z listingu 16.8 tworzy obiekt typu `FileInfo` dwa razy dla każdego elementu ze źródłowej kolekcji. Jest to niepotrzebne marnowanie zasobów. Aby uniknąć tego rodzaju zbędnych i potencjalnie wysokich kosztów, możesz zastosować klauzulę `let`, co przedstawiono na listingu 16.9.

Listing 16.9. Porządkowanie wyników w wyrażeniu z kwerendą

```
// ...
IEnumerable<FileInfo> files =
    from fileName in Directory.EnumerateFiles(
        rootDirectory, searchPattern)
    let file = new FileInfo(fileName)
    orderby file.Length, fileName
    select file;
// ...
```

Klauzula `let` wprowadza nową zmienną zakresową, która może przechowywać wartość wyrażenia używaną w dalszych miejscach wyrażenia z kwerendą. Możesz utworzyć dowolną liczbę klauzul `let`. Wystarczy, że dodasz każdą z nich do kwerendy jako dodatkową klauzulę po pierwszej klauzuli `from`, ale przed ostatnią klauzulą `select` lub `group by`.

Grupowanie

W trakcie manipulowania danymi często trzeba pogrupować powiązane elementy. W SQL-u zwykle polega to na agregacji elementów w celu utworzenia sumy lub innej zagregowanej wartości. Technologia LINQ daje znacznie większe możliwości. Wyrażenia w technologii LINQ pozwalają grupować pojedyncze elementy w zestawy podkolekcji. Grupy można powiązać z elementami kolekcji, której dotyczy kwerenda. Na przykład na listingu 16.10 i w danych wyjściowych 16.6 pokazano, jak pogrupować kontekstowe i zwykłe słowa kluczowe.

Listing 16.10. Grupowanie wyników zwracanych przez kwerendę

3.0

```
using System;
using System.Collections.Generic;
using System.Linq;
// ...
private static void GroupKeywords1()
{
    IEnumerable<IGrouping<bool, string>> selection =
        from word in CSharp.Keywords
        group word by word.Contains('*');

    foreach (IGrouping<bool, string> wordGroup
        in selection)
    {
        Console.WriteLine(Environment.NewLine + "{0}:", wordGroup.Key ?
            "Kontekstowe słowa kluczowe": "Słowa kluczowe");
```

```

foreach (string keyword in wordGroup)
{
    Console.Write(" " +
        (wordGroup.Key ?
            keyword.Replace("*", null) : keyword));
}
}

// ...

```

DANE WYJŚCIOWE 16.6.

Słowa kluczowe:

```

abstract as base bool break byte case catch char checked class
const continue decimal default delegate do double else enum event
explicit extern false finally fixed float for foreach goto if
implicit in int interface internal is lock long namespace new null
operator out override object params private protected public
readonly ref return sbyte sealed short sizeof stackalloc static
string struct switch this throw true try typeof uint ulong unsafe
ushort using virtual unchecked void volatile while

```

Kontekstowe słowa kluczowe:

```

add alias ascending async await by descending dynamic equals from
get global group into join let nameof nonnull on orderby partial remove
select set value var where yield

```

Zwróć uwagę na kilka rzeczy na listingu 16.10. Przede wszystkim wynik zwrócony przez kwerendę to sekwencja elementów typu `IGrouping<bool, string>`. Pierwszy argument określający typ oznacza, że podane po członie `by` wyrażenie reprezentujące klucz grupy jest typu `bool`. Drugi argument określający typ oznacza, że podane po członie `group` wyrażenie reprezentujące elementy grupy jest typu `string`. Tak więc omawiana kwerenda generuje sekwencję grup, w których klucz typu logicznego ma tę samą wartość dla każdej wartości typu `string` z danej grupy.

Ponieważ kwerenda z klauzulą `group by` generuje sekwencję kolekcji, iterowanie po wynikach często odbywa się za pomocą zagnieżdzonej pętli `foreach`. Na listingu 16.10 zewnętrzna pętla iteruje po grupach i wyświetla jako nagłówek typ słów kluczowych z poszczególnych grup. Zagnieżdżona pętla `foreach` wyświetla każde słowo kluczowe z grupy jako element pod nagłówkiem.

3.0

Wynik omawianego wyrażenia z kwerendą to sekwencja, z której można pobierać dane w ten sam sposób jak z każdej innej sekwencji. Na listingu 16.11 i w danych wyjściowych 16.7 pokazano, jak utworzyć dodatkową kwerendę, która dodaje projekcję do kwerendy zwracającej sekwencję grup. W następnym punkcie w rozwinięciu kwerendy zobaczysz zalecaną składnię pozwalającą dodać kolejne klauzule w celu uzupełnienia kwerendy.

Listing 16.11. Pobieranie wartości krotki podanej po klauzuli group

```

using System;
using System.Collections.Generic;
using System.Linq;

```

```
// ...

private static void GroupKeywords1()
{
    I Enumerable<IGrouping<bool, string>> keywordGroups =
        from word in CSharp.Keywords
        group word by word.Contains('*');

    I Enumerable<(bool IsContextualKeyword, I Grouping<bool, string> Items)>
    ↪selection =
        from groups in keywordGroups
        select
        (
            IsContextualKeyword: groups.Key,
            Items: groups
        );

    foreach (
        (bool IsContextualKeyword, I Grouping<bool, string> Items)
        wordGroup in selection)
    {
        Console.WriteLine(Environment.NewLine + "{0}:", 
            wordGroup.IsContextualKeyword ?
                "Kontekstowe słowa kluczowe" : "Słowa kluczowe");
        foreach (string keyword in wordGroup.Items)
        {
            Console.Write(" " +
                keyword.Replace("*", null));
        }
    }
}
// ...

```

DANE WYJŚCIOWE 16.7.

3.0 Słowa kluczowe:

```
abstract as base bool break byte case catch char checked class
const continue decimal default delegate do double else enum
event explicit extern false finally fixed float for foreach goto if
implicit in int interface internal is lock long namespace new null
operator out override object params private protected public
readonly ref return sbyte sealed short sizeof stackalloc static
string struct switch this throw true try typeof uint ulong unsafe
ushort using virtual unchecked void volatile while
Kontekstowe słowa kluczowe:
add alias ascending async await by descending dynamic equals from
get global group into join let nameof nonnull on orderby partial remove
select set value var where yield
```

Klauzula `group` pozwala utworzyć kwerendę, która zwraca kolekcję obiektów typu `IGrouping<TKey, TElement>`. Podobnie działa standardowy operator kwerendy `GroupBy()` (zobacz rozdział 15.). Klauzula `select` w następnej kwerendzie za pomocą krotki zmienia nazwę właściwości `IGrouping<TKey, TElement>.Key` na `IsContextualKeyword`, a dla właściwości zawierającej podkolekcję tworzy nazwę `Items`. Po tej zmianie w zagnieżdżonej pętli `foreach` używana jest właściwość `wordGroup.Items` zamiast (jak na listingu 16.10) bezpośrednio zmienna `wordGroup`.

Inna wartość, którą można dodać do tworzonej krotki, to liczba elementów w podkolekcji. Ta liczba jest już dostępna za pomocą metody `wordGroup.Items.Count()` z technologii LINQ, dlatego dodawanie wspomnianej wartości bezpośrednio do krotki nie jest zbyt przydatne.

Kontynuowanie kwerendy za pomocą klauzuli `into`

Na listingu 16.11 pokazano, że można wykorzystać istniejącą kwerendę jako dane wejściowe w drugiej kwerendzie. Nie trzeba jednak pisać nowego wyrażenia z kwerendą, by zastosować wyniki z jednej kwerendy jako dane wejściowe innej. Można rozwinąć kwerendę za pomocą **klauzuli kontynuacji kwerendy**. Do tworzenia takich klauzul służą kontekstowe słowo kluczowe `into`. Kontynuacja kwerendy to nic więcej jak udogodnienie składniowe, ułatwiające utworzenie dwóch kwerend i wykorzystanie pierwszej z nich jako danych wejściowych drugiej. Zmienna zakresowa dodana w klauzuli `into` (na listingu 16.11 jest to zmienna `groups`) staje się zmienną zakresową w dalszym kodzie kwerendy. Wcześniej zmienne zakresowe są logicznie częścią poprzedniej kwerendy i nie mogą być używane w jej kontynuacji. Na listingu 16.12 pokazano, jak napisać nową wersję kodu z listingu 16.11 i zastosować kontynuację kwerendy zamiast dwóch osobnych kwerend.

Listing 16.12. Pobieranie danych z wykorzystaniem kontynuacji kwerendy

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

// ...

private static void GroupKeywords1()
{
    IEnumerable<(bool IsContextualKeyword, IGrouping<bool, string> Items)>
        ↪selection =
            from word in CSharp.Keywords
            group word by word.Contains('*')
            into groups
            select
            (
                IsContextualKeyword: groups.Key,
                Items: groups
            );
    // ...
}
// ...
```

3.0

Możliwość uruchamiania za pomocą klauzuli `into` dodatkowych kwerend dotyczących wyników istniejących kwerend nie ogranicza się do kwerend zakończonych klauzulą `group`. Tę technikę można stosować do wszystkich wyrażeń z kwerendami. Kontynuacja kwerendy to skrót pozwalający zapisać wyrażenie z kwerendą, które przetwarza wyniki innego takiego wyrażenia. Możesz traktować słowo kluczowe `into` jak operator potoku, ponieważ łączy w potok wyniki z pierwszej kwerendy z drugą kwerendą. W ten sposób możesz swobodnie połączyć wiele kwerend.

„Spłaszczenie” sekwencji składających się z sekwencji za pomocą kilku klauzul from

Często przydatne jest „spłaszczenie” sekwencji składających się z sekwencji w jedną. Na przykład każdy element sekwencji zawierającej rekordy klientów może być powiązany z sekwencją zamówień. Podobnie każdy element sekwencji komunikatów może być powiązany z sekwencją plików. Operator sekwencji `SelectMany` (omówiony w rozdziale 15.) złącza wszystkie podsekwencje. Aby uzyskać ten sam efekt za pomocą wyrażenia z kwerendą, można zastosować kilka klauzul `from`, co pokazano na listingu 16.13.

Listing 16.13. Kilka klauzul from

```
var selection =
    from word in CSharp.Keywords
    from character in word
    select character;
```

Ta kwerenda zwraca sekwencję znaków `a`, `b`, `s`, `t`, `r`, `a`, `c`, `t`, `a`, `d`, `d`, `*`, `a`, `l`, `i`, `a`, ...

Za pomocą kilku klauzul `from` można też utworzyć **iloczyn kartezjański** — zestaw wszystkich możliwych kombinacji elementów kilku sekwencji. Tę technikę pokazano na listingu 16.14.

Listing 16.14. Iloczyn kartezjański

```
var numbers = new[] { 1, 2, 3 };
IEnumerable<(string Word, int Number)> product =
    from word in CSharp.Keywords
    from number in numbers
    select (word, number);
```

Ta kwerenda generuje sekwencję par `(abstract, 1)`, `(abstract, 2)`, `(abstract, 3)`, `(as, 1)`, `(as, 2)`, ...

3.0

■ ZAGADNIENIE DLA POCZĄTKUJĄCYCH

Niepowtarzalne elementy

Często przydatne jest zwrócenie tylko niepowtarzalnych (unikatowych) elementów kolekcji i pominięcie powtarzających się wartości. Nie istnieje składnia, która pozwala bezpośrednio wykonać taką operację w wyrażeniach z kwerendą. Potrzebny efekt można uzyskać za pomocą operatora kwerend `Distinct()`, przedstawionego w rozdziale 15. Aby zastosować operator kwerend do wyrażenia z kwerendą, wyrażenie musi się znajdować w nawiasie, by kompilator nie uznał, iż wywołanie `Distinct()` to część klauzuli `select`. Przykładowy kod znajdziesz na listingu 16.15, a wyniki — w danych wyjściowych 16.8.

Listing 16.15. Pobieranie niepowtarzalnych elementów z wyrażenia z kwerendą

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

// ...

public static void ListMemberNames()
{
    IEnumerable<string> enumerableMethodNames = (
        from method in typeof(Enumerable).GetMembers(
            System.Reflection.BindingFlags.Static |
            System.Reflection.BindingFlags.Public)
        orderby method.Name
        select method.Name).Distinct();

    foreach(string method in enumerableMethodNames)
    {
        Console.WriteLine($"{ method }, ");
    }
}
// ...
```

DANE WYJŚCIOWE 16.8.

```
Aggregate, All, Any, AsEnumerable, Average, Cast, Concat, Contains,
Count, DefaultIfEmpty, Distinct, ElementAt, ElementOrDefault,
Empty, Except, First, FirstOrDefault, GroupBy, GroupJoin,
Intersect, Join, Last, LastOrDefault, LongCount, Max, Min, OfType,
OrderBy, OrderByDescending, Range, Repeat, Reverse, Select,
SelectMany, SequenceEqual, Single, SingleOrDefault, Skip,
SkipWhile, Sum, Take, TakeWhile, ThenBy, ThenByDescending, ToArray,
ToDictionary, ToList, ToLookup, Union, Where, Zip,
```

W tym przykładzie wywołanie `typeof(Enumerable).GetMembers()` zwraca listę wszystkich składowych (metod, właściwości itd.) z klasy `System.Linq.Enumerable`. Jednak wiele tych składowych jest przeciżonych (czasem więcej niż raz). Zamiast wielokrotnie wyświetlać te same składowe, w wyrażeniu z kwerendą wywołano metodę `Distinct()`. To eliminuje powtarzające się nazwy z listy. Szczegółowe omówienie operatora `typeof()` i mechanizmu refleksji (związanego z metodami takimi jak `GetMembers()`) znajdziesz w rozdziale 18.

3.0

Wyrażenia z kwerendą to tylko wywołania metod

Dodanie wyrażeń z kwerendami w wersji C# 3.0 nie wymagało zmian w środowisku CLR ani w języku CIL, co może się wydawać zaskakujące. Komplikator języka C# przekształca wyrażenia z kwerendami na serię wywołań metod. Przyjrzyj się wyrażeniu z kwerendą z listingu 16.1. Fragment tego kodu powtórzono na listingu 16.16.

Listing 16.16. Proste wyrażenie z kwerendą

```
private static void ShowContextualKeywords1()
{
    IEnumerable<string> selection =
        from word in CSharp.Keywords
        where word.Contains('*')
        select word;
    // ...
}
```

Po komplikacji wyrażenie z listingu 16.16 jest przekształcane na wywołanie dostępnej w klasie `System.Linq.Enumerable` metody rozszerzającej typ `IEnumerable<T>`. To wywołanie pokazano na listingu 16.17.

Listing 16.17. Wyrażenie z kwerendą przekształcone na składnię wykorzystującą standardowe operatory kwerend

```
private static void ShowContextualKeywords3()
{
    IEnumerable<string> selection =
        CSharp.Keywords.Where(word => word.Contains('*'));
    // ...
}
```

W rozdziale 15. wyjaśniono, że wyrażenie lambda jest przekształcane przez kompilator w metodę z kodem danej lambdy. Z tą metodą jest potem wiązany delegat.

Każde wyrażenie z kwerendą może (i musi) zostać przekształcone na wywołania metod. Jednak nie dla każdej sekwencji wywołań metod można utworzyć analogiczne wyrażenie z kwerendą. Na przykład nie istnieje wyrażenie z kwerendą odpowiadające metodzie rozszerzającej `TakeWhile<T>(Func<T, bool> predicate)`, która wielokrotnie zwraca elementy z kolekcji dopóki, dopóki predykat ma wartość `true`.

Jeśli kwerendę można zapisać zarówno w postaci wywołań metod, jak i wyrażenia z kwerendą, która postać jest lepsza? Musisz sam to ocenić. Niektóre kwerendy lepiej przedstawić w formie wyrażeń z kwerendą, inne są bardziej czytelne jako wywołania metod.

3.0

Wskazówki

STOSUJ składnię wyrażeń z kwerendą, by zwiększyć czytelność kwerend (zwłaszcza jeśli obejmują skomplikowane klauzule `from`, `let`, `join` lub `group`).

ROZWAŻ stosowanie standardowych operatorów kwerend (czyli kwerendy w postaci wywołań metod), jeśli kwerenda obejmuje operacje, dla których nie ma odpowiednika w składni wyrażeń z kwerendami. Są to na przykład operacje `Count()`, `TakeWhile()` i `Distinct()`.

Podsumowanie

W tym rozdziale przedstawiono nową składnię — wyrażenia z kwerendami. Czytelnicy znający SQL natychmiast dostrzegą podobieństwa między wyrażeniami z kwerendami a SQL-em. Jednak wyrażenia z kwerendami zapewniają też nowe mechanizmy (takie jak grupowanie danych w hierarchiczny zbiór nowych obiektów), niedostępne w SQL-u. Wszystkie funkcje wyrażeń z kwerendą były już dostępne w postaci standardowych operatorów kwerend. Jednak wyrażenia z kwerendą często pozwalają zapisać kwerendę w prostszej formie. Zarówno standardowe operatory kwerend, jak i wyrażenia z kwerendą są znacznym usprawnieniem ułatwiającym programistom pisanie kodu z wykorzystaniem interfejsów API kolekcji. To usprawnienie stanowi przełom w obszarze komunikowania się języków obiektowych z relacyjnymi bazami danych.

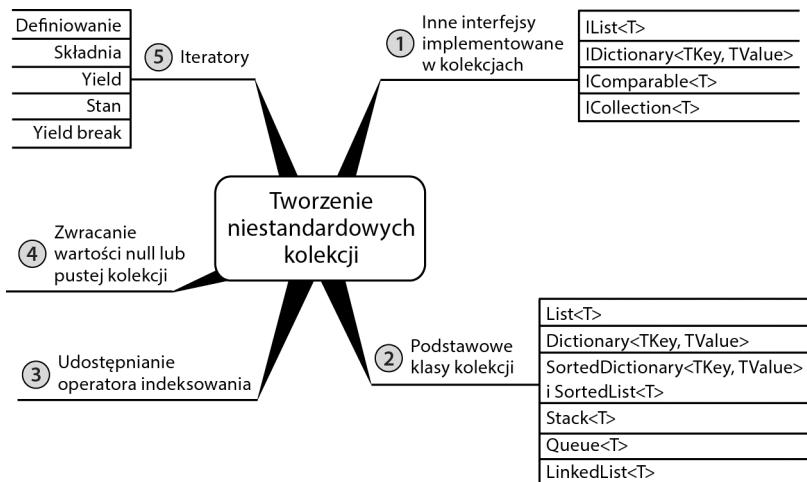
Następny rozdział zawiera ciąg dalszy omówienia kolekcji. Poznasz tam wybrane typy kolekcji platformy .NET i zobaczysz, jak definiować niestandardowe kolekcje.

17

Tworzenie niestandardowych kolekcji

Początek
2.0

ROZDZIAŁE 15. OPISANO STANDARDOWE operatory kwerend, czyli metody rozszerzające interfejs `IEnumerable<T>`, który zapewnia metody dostępne we wszystkich kolekcjach. Jednak te operatory nie sprawiają, że wszystkie kolekcje w równym stopniu nadają się do wykonywania każdego zadania. Nadal potrzebne są różne typy kolekcji. Niektóre kolekcje lepiej obsługują wyszukiwanie za pomocą klucza, inne są lepiej dostosowane do dostępu do elementów na podstawie pozycji. Pewne kolekcje działają jak kolejki; pierwszy dodany element jest w nich też pierwszym pobieranym. Inne przypominają stos — pierwszy dodany element jest pobierany jako ostatni. Niektóre kolekcje w ogóle nie są uporządkowane.



Platforma .NET udostępnia wiele typów kolekcji dostosowanych do wielu scenariuszy, w których kolekcje są potrzebne. W tym rozdziale znajdziesz wprowadzenie do wybranych typów kolekcji i implementowanych w nich interfejsów. Opisano tu też, jak tworzyć niestandardowe kolekcje z obsługą standardowych funkcji (na przykład indeksów). Ponadto omówiono stosowanie instrukcji `yield return` do tworzenia klas i metod z implementacją interfejsu `IEnumerable<T>`. Ten wprowadzony w wersji C# 2.0 mechanizm znacznie upraszcza implementowanie kolekcji, po których można iterować za pomocą instrukcji `foreach`.

W platformie .NET dostępnych jest wiele niegenerycznych klas i interfejsów kolekcji. Jednak obecnie istnieją one tylko w celu zapewniania zgodności z kodem napisanym przed wprowadzeniem typów generycznych. Generyczne typy kolekcji są bardziej wydajne (ponieważ pozwalają uniknąć kosztów związanych z opakowywaniem wartości), a także bezpieczniejsze ze względu na typ w porównaniu z kolekcjami niegenerycznymi. Dlatego w nowym kodzie prawie zawsze należy korzystać wyłącznie z generycznych typów kolekcji. W tej książce przyjęto, że posługujesz się przede wszystkim generycznymi typami kolekcji.

2.0

Inne interfejsy implementowane w kolekcjach

Wiesz już, jak w kolekcjach implementowany jest interfejs `IEnumerable<T>` — główny interfejs umożliwiający iterowanie po elementach kolekcji. Istnieje też wiele dodatkowych interfejsów implementowanych w bardziej skomplikowanych kolekcjach. Na rysunku 17.1 pokazano hierarchię interfejsów implementowanych w klasach kolekcji.

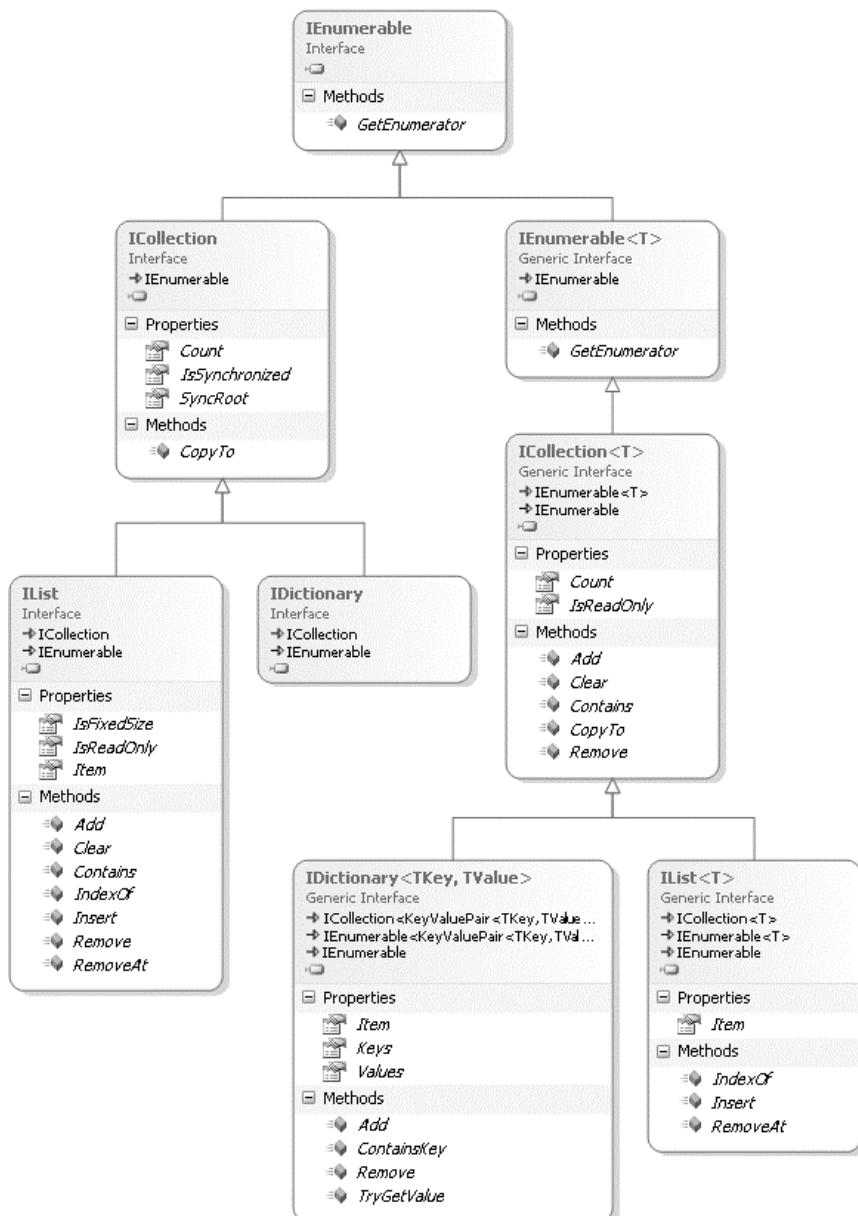
Wymienione interfejsy zapewniają standardowy sposób wykonywania typowych zadań, takich jak iterowanie, indeksowanie i zliczanie elementów kolekcji. W tym podrozdziale opisano wszystkie generyczne interfejsy, zaczynając od interfejsów z dolnej części rysunku 17.1 i przechodząc do interfejsów wymienionych wyżej.

Porównanie interfejsów `IList<T>` i `IDictionary<TKey, TValue>`

Słownik języka polskiego możesz sobie wyobrazić jako kolekcję definicji. Dostęp do konkretnej definicji można szybko uzyskać na podstawie powiązanego z nią klucza, czyli definiowanego słowa. Klasa kolekcji reprezentującej słownik też jest kolekcją wartości, w której dostęp do każdej wartości można szybko uzyskać na podstawie unikatowego klucza. Zauważ jednak, że w słowniku języka definicje zwykle są uporządkowane alfabetycznie według kluczy. W klasie słownika też może tak być, jednak przeważnie stosowane są inne rozwiązania.

Kolekcje w postaci słownika najlepiej traktować jak nieuporządkowane listy kluczy i powiązanych z nimi wartości, chyba że w dokumentacji opisano, iż wartości są uporządkowane. Poza tym użytkownik zwykle nie szuka na przykład „szóstej definicji w słowniku”. Dlatego klasy słownika najczęściej umożliwiają indeksowanie tylko na podstawie kluczy, a nie według pozycji.

Natomiast na liście wartości są przechowywane w określonej kolejności, a dostęp do nich odbywa się na podstawie pozycji. W pewnym sensie listy są specjalną odmianą słownika, gdzie klucz to zawsze liczba całkowita, a zbiór kluczy to zestaw kolejnych nieujemnych liczb całkowitych rozpoczynający się od zera. Jest to jednak wystarczająco istotna różnica, by warto było utworzyć zupełnie odmienne typy do reprezentowania słowników i list.



Rysunek 17.1. Hierarchia generycznych interfejsów implementowanych w kolekcjach

Dlatego gdy wybierasz klasę kolekcji w celu wykonania określonego zadania związanego z przechowywaniem lub pobieraniem danych, pierwsze dwa interfejsy, którym warto się przyjrzeć, to **IList<T>** i **IDictionary<TKey, TValue>**. Te interfejsy określają, czy typ kolekcji ma umożliwiać pobieranie wartości na podstawie indeksu określającego pozycję, czy na podstawie klucza.

Oba wymienione interfejsy wymagają udostępnienia indeksera w klasie z ich implementacją. Gdy używany jest interfejs `IList<T>`, operand indeksera odpowiada pozycji pobieranego elementu. Indekser przyjmuje wtedy liczbę całkowitą i zapewnia dostęp do n -tego elementu listy. Jeśli chodzi o interfejs `IDictionary< TKey, TValue >`, to operand indeksera jest powiązany z wartością kluczem i zapewnia dostęp do tej wartości.

Interfejs `ICollection<T>`

Interfejsy `IList<T>` i `IDictionary< TKey, TValue >` dziedziczą po interfejsie `ICollection<T>`. Kolekcja, w której nie zaimplementowano ani interfejsu `IList<T>`, ani interfejsu `IDictionary< TKey, TValue >`, prawie z pewnością zawiera implementację interfejsu `ICollection<T>` (choć nie jest to konieczne, ponieważ w kolekcjach można też zaimplementować mnóstwo wymagających interfejsów: `IEnumerable` lub `IEnumerable<T>`). Interfejs `ICollection<T>` jest pochodny od interfejsu `IEnumerable<T>` i obejmuje dwie składowe — `Count` i `CopyTo()`.

- Właściwość `Count` zwraca łączną liczbę elementów kolekcji. Początkowo może się wydawać, że wystarczy wykonać iterację po wszystkich elementach kolekcji za pomocą pętli `for`, jednak kolekcja musi wtedy obsługiwać pobieranie wartości za pomocą indeksu, czego interfejs `ICollection<T>` nie zapewnia (gwarantuje to natomiast interfejs `IList<T>`).
- Metoda `CopyTo()` umożliwia przekształcenie kolekcji w tablicę. Ta metoda ma parametr `index`, pozwalający określić, w którym miejscu docelowej tablicy elementy mają zostać wstawione. Aby zastosować tę metodę, trzeba zainicjować docelową tablicę o wystarczającej pojemności (od pozycji `index`), tak by zmieściły się w niej wszystkie elementy z kolekcji typu `ICollection<T>`.

2.0

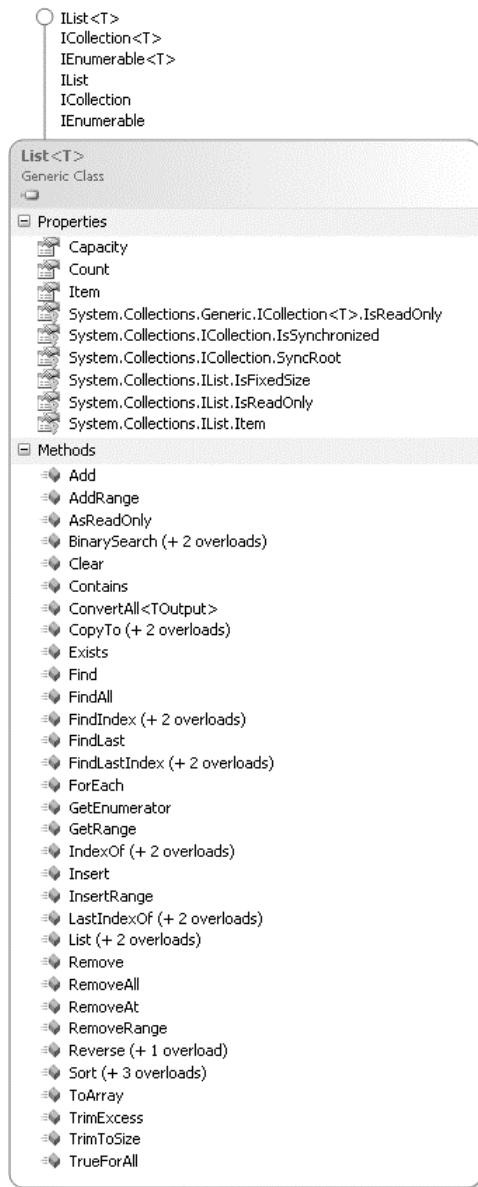
Podstawowe klasy kolekcji

Istnieje pięć rodzajów klas kolekcji. Różnią się one między sobą sposobem wstawiania, przechowywania i pobierania danych. Każda klasa generyczna znajduje się w przestrzeni nazw `System.Collections.Generic`, a ich niegeneryczne odpowiedniki są dostępne w przestrzeni nazw `System.Collections`.

Kolekcje w postaci list — `List<T>`

Klasa `List<T>` działa podobnie jak tablica. Główna różnica między nimi polega na tym, że listy są automatycznie wydłużane wraz ze wzrostem liczby elementów, natomiast rozmiar tablicy pozostaje stały. Ponadto listy można skracać za pomocą jawnych wywołań `TrimToSize()` i `Capacity` (zobacz rysunek 17.2).

Klasy z tej grupy to **kolekcje w postaci listy**. Ich charakterystyczną cechą jest to, że za pomocą indeksu można uzyskać dostęp do każdego elementu (podobnie jak w tablicach). Dlatego można ustawić i pobierać wartość elementów listy za pomocą operatora indeksowania. Wartość indeksu odpowiada pozycji elementu w kolekcji. Na listingu 17.1 pokazano przykładowy kod, a w danych wyjściowych 17.1 — efekty jego wykonania.



2.0

Rysunek 17.2. Diagram klas `List<T>`**Listing 17.1.** Używanie klasy `List<T>`

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

class Program
{
    static void Main()
    {
```

```

List<string> list = new List<string>()
{
    "Apsik", "Wesołek", "Gapcio", "Mędrak", "Śpioszek", "Nieśmiałek", "Gburek"};
    list.Sort();

    Console.WriteLine(
        $"W porządku alfabetycznym pierwszy krasnal to { list[0] }, a "
        + $"ostatnim jest { list[6] }.");
}

list.Remove("Gburek");
}
}

```

DANE WYJŚCIOWE 17.1.

W porządku alfabetycznym pierwszy krasnal to Apsik, a ostatnim jest Wesołek.

W języku C# pierwszy indeks ma wartość zero. Tak więc indeks 0 na listingu 17.1 odpowiada pierwszemu elementowi, a indeks 6 — siódmemu. Pobieranie elementów za pomocą indeksu nie obejmuje wyszukiwania. Wystarczy wtedy krótka i prosta operacja skoku do odpowiedniej lokalizacji w pamięci.

2.0

Kolekcja `List<T>` jest uporządkowana. Metoda `Add()` dodaje dany element na koniec listy. W kodzie z listingu 17.1 przed wywołaniem metody `Sort()` wartość "Apsik" była pierwszym elementem, a "Gburek" — ostatnim. Po wywołaniu odpowiedniej metody lista została posortowana alfabetycznie, dlatego elementy nie były już uporządkowane zgodnie z kolejnością ich dodawania. Niektóre kolekcje automatycznie sortują dodawane elementy, jednak `List<T>` nie jest jedną z nich. Posortowanie elementów listy wymaga jawnego wywołania metody `Sort()`.

Aby usunąć element, wystarczy wywołać metodę `Remove()` lub `RemoveAt()`. Powoduje to usunięcie podanego elementu lub elementu o podanym indeksie.

ZAGADNIENIE DLA ZAAWANSOWANYCH**Modyfikowanie procesu sortowania kolekcji**

Może się zastanawiasz, skąd metoda `List<T>.Sort()` z listingu 17.1 wiedziała, jak posortować elementy listy w porządku alfabetycznym. Typ `string` zawiera implementację interfejsu `IComparable<string>`, obejmującego jedną metodę — `CompareTo()`. Ta metoda zwraca liczbę całkowitą określającą, czy przekazany element jest większy, mniejszy, czy równy względem elementu, dla którego wywołano metodę. Jeśli w typie elementu zaimplementowany jest generyczny interfejs `IComparable<T>` (lub niegeneryczny interfejs `IComparable`), algorytm sortujący domyślnie zastosuje wspomnianą metodę do określenia sposobu sortowania.

Co się jednak dzieje, jeśli w typie elementu nie zaimplementowano interfejsu `IComparable<T>` lub domyślny mechanizm porównywania dwóch elementów nie jest zgodny z potrzebami programisty? Aby określić niestandardowy sposób sortowania, możesz wywołać przeciążoną metodę `List<T>.Sort()` przyjmującą jako argument obiekt z implementacją interfejsu `IComparer<T>`.

Różnica między interfejsami `IComparable<T>` i `IComparer<T>` jest subtelna, ale ważna. Pierwszy interfejs oznacza: „wiem, jak porównać samego siebie do innej instancji mojego typu”, a drugi: „wiem, jak porównać dwa elementy określonego typu”.

Interfejs `IComparer<T>` jest zwykle stosowany wtedy, gdy istnieje wiele możliwych sposobów sortowania wartości danego typu i żaden z nich nie jest wyraźnie najlepszy. Może na przykład istnieć kolekcja obiektów typu `Contact`, które programista czasem chce sortować według nazwisk, a innym razem według miejscowości, dat urodzenia, regionu lub dowolnych innych danych. Dlatego zamiast decydować się na jedną strategię sortowania i implementować w klasie `Contact` interfejs `IComparable<Contact>`, lepiej utworzyć kilka różnych klas z implementacją interfejsu `IComparer<Contact>`. Na listingu 17.2 przedstawiono przykładową implementację z porównywaniem właściwości `LastName` i `FirstName`.

Listing 17.2. Implementacja interfejsu `IComparer<T>`

```
class Contact
{
    public string FirstName { get; private set; }
    public string LastName { get; private set; }
    public Contact(string firstName, string lastName)
    {
        this.FirstName = firstName;
        this.LastName = lastName;
    }
}
using System;
using System.Collections.Generic;

class NameComparison : IComparer<Contact>
{
    public int Compare(Contact? x, Contact? y)
    {
        if (Object.ReferenceEquals(x, y))
            return 0;
        if (x == null)
            return 1;
        if (y == null)
            return -1;
        int result = StringCompare(x.LastName, y.LastName);
        if (result == 0)
            result = StringCompare(x.FirstName, y.FirstName);
        return result;
    }

    private static int StringCompare(string? x, string? y)
    {
        if (Object.ReferenceEquals(x, y))
            return 0;
        if (x == null)
            return 1;
        if (y == null)
            return -1;
        return x.CompareTo(y);
    }
}
```

2.0

Aby posortować kolekcję typu `List<Contact>` najpierw na podstawie nazwisk, a następnie dodatkowo według imion, wywołaj metodę `contactList.Sort(new NameComparer())`.

Porządkowanie całkowite

Gdy implementowane są interfejsy `IComparable<T>` lub `IComparer<T>`, trzeba zapewnić **porządek całkowity**. Implementacja metody `CompareTo` musi wtedy zapewniać w pełni spójne uporządkowanie dla dowolnej pary elementów. Takie uporządkowanie ma mieć szereg podstawowych cech. Na przykład każdy element musi być równy samemu sobie. Jeśli element X jest uznawany za równy elementowi Y, a element Y jest równy elementowi Z, wszystkie trzy elementy (X, Y i Z) muszą być uznawane za równe. Jeżeli element X jest uznawany za większy od elementu Y, element Y musi być uznawany za mniejszy od elementu X. Nie mogą też występować paradoksy przechodniości — nie może być tak, że X jest większe niż Y, Y większe niż Z, a Z większe niż X. Jeśli nie zapewnysz porządku całkowitego, działanie algorytmu sortującego będzie nieprzewidywalne. Możliwe, że algorytm uporządkuje elementy w niespodziewany sposób, spowoduje awarię programu, uruchomi pętlę nieskończoną itd.

Zauważ, że mechanizm porównywania z listingu 17.2 zapewnia porządek całkowity — nawet gdy argumenty to referencje `null`. Nie jest jednak dozwolone stwierdzenie, że „jeśli któryś element jest równy `null`, zwróć zero”, ponieważ wtedy dwie wartości różne od `null` będą równe wartości `null`, ale różne od siebie.

2.0

Wskazówka

UPEWNIJ SIĘ, że niestandardowy kod do obsługi porównań gwarantuje porządek całkowity.

Przeszukiwanie kolekcji typu `List<T>`

Aby znaleźć na liście `List<T>` konkretny element, można się posłużyć metodami `Contains()`, `IndexOf()`, `LastIndexOf()` i `BinarySearch()`. Pierwsze trzy z tych metod przeszukują tablicę, począwszy od pierwszego elementu (lub ostatniego w przypadku metody `LastIndexOf()`), i sprawdzają każdą wartość do momentu znalezienia tej szukanej. Czas działania tych metod jest proporcjonalny do liczby elementów, które trzeba sprawdzić do czasu natrafienia na szukaną wartość. Zauważ, że klasy kolekcji nie wymagają, by elementy były unikatowe. Jeśli dwa elementy kolekcji (lub większa ich liczba) są identyczne, metoda `IndexOf()` zwraca indeks pierwszego ze znalezionych elementów, a metoda `LastIndexOf()` — ostatniej znalezionej wartości.

Metoda `BinarySearch()` posługuje się znacznie szybszym algorytmem wyszukiwania binarnego, ale wymaga, by elementy były posortowane. Przydatną cechą metody `BinarySearch()` jest to, że jeśli element nie zostanie znaleziony, metoda zwraca ujemną liczbę całkowitą. Dopełnienie bitowe (`-`) tej liczby to indeks pierwszego elementu większego od szukanego lub, jeśli żadna wartość nie jest większa od szukanej, łączna liczba elementów. To zapewnia wygodny mechanizm wstawiania nowych wartości na listę w konkretnym miejscu, aby zachować porządek sortowania. Na listingu 17.3 przedstawiono przykładowy kod.

Listing 17.3. Używanie dopełnienia bitowego do wyniku zwróconego przez metodę BinarySearch()

```

using System;
using System.Collections.Generic;

class Program
{
    static void Main()
    {
        List<string> list = new List<string>();
        int search;

        list.Add("public");
        list.Add("protected");
        list.Add("private");

        list.Sort();

        search = list.BinarySearch("protected internal");
        if (search < 0)
        {
            list.Insert(~search, "protected internal");
        }

        foreach (string accessModifier in list)
        {
            Console.WriteLine(accessModifier);
        }
    }
}

```

2.0

Zauważ, że jeśli lista nie jest posortowana, przedstawiony kod może nie znaleźć danego elementu (nawet jeżeli dana wartość występuje na liście). Wynik działania kodu z listingu 17.3 pokazano w danych wyjściowych 17.2.

DANE WYJŚCIOWE 17.2.

```

private
protected
protected internal
public

```

ZAGADNIENIE DLA ZAAWANSOWANYCH**Wyszukiwanie wielu elementów za pomocą metody FindAll()**

Czasem trzeba znaleźć wiele elementów na liście, a kryteria są bardziej skomplikowane niż sprawdzanie konkretnej wartości. Na potrzeby tej sytuacji w klasie `System.Collections.Generic.List<T>` udostępniono metodę `FindAll()`. Przyjmuje ona parametr typu `Predicate<T>`, który określa delegat (referencję do metody). Na listingu 17.4 pokazano, jak posługiwać się metodą `FindAll()`.

Listing 17.4. Metoda FindAll() i jej parametr w postaci predykatu

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

class Program
{
    static void Main()
    {
        List<int> list = new List<int>();
        list.Add(1);
        list.Add(2);
        list.Add(3);
        list.Add(2);

        List<int> results = list.FindAll(Even);

        foreach (int number in results)
        {
            Console.WriteLine(number);
        }
    }

    public static bool Even(int value) =>
        (value % 2) == 0;
}
```

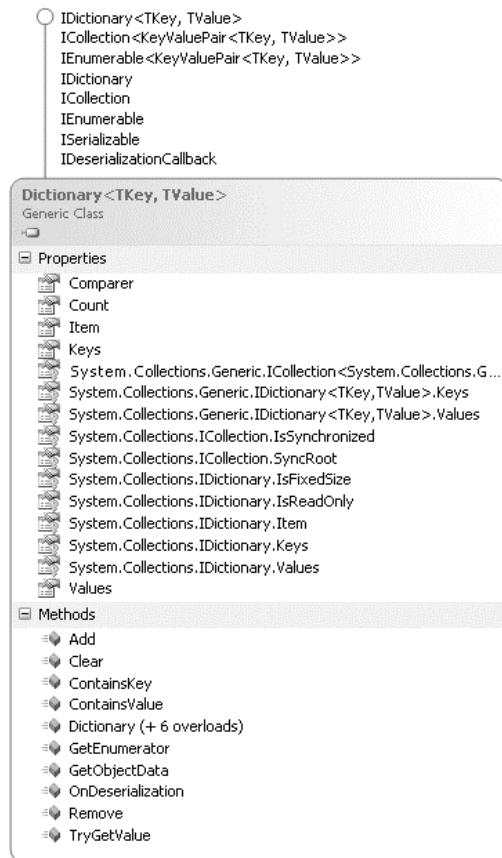
2.0

Na listingu 17.4 do metody `FindAll()` przekazywany jest delegat `Even()`. Zwraca on `true`, jeśli wartość całkowitoliczbowego argumentu jest parzysta. Metoda `FindAll()` przyjmuje delegat `Even()` i wywołuje go dla każdego elementu z listy (na przedstawionym listingu wykorzystano inferencję typu delegata; ten mechanizm wprowadzono w wersji C# 2.0). Za każdym razem, gdy zwrócona wartość to `true`, kod dodaje element do nowej instancji typu `List<T>`, a następnie zwraca tę instancję po sprawdzeniu wszystkich elementów z obiektu `list`. Kompletne omówienie delegatów znajdziesz w rozdziale 13.

Kolekcje w postaci słownika — `Dictionary< TKey, TValue >`

Inną kategorią klas kolekcji są klasy słownika — `Dictionary< TKey, TValue >` (zobacz rysunek 17.3). Klasy słownika, w odróżnieniu od klas listy, przechowują pary nazwa-wartość. Nazwy pełnią funkcję unikatowego klucza, który można wykorzystać do wyszukania odpowiedniego elementu w podobny sposób, jak klucz główny umożliwia dostęp do rekordu w bazie danych. Zwiększa to złożoność procesu dostępu do elementów słownika, jednak ponieważ wyszukiwanie na podstawie klucza to wydajna operacja, słowniki są przydatnymi kolekcjami. Zauważ, że klucz może być dowolnego typu danych; nie musi być łańcuchem znaków lub liczbą.

Jedną z technik wstawiania elementów do słownika jest wywoływanie metody `Add()` i przekazanie do niej zarówno klucza, jak i wartości. To rozwiązanie pokazano na listingu 17.5.



2.0

Rysunek 17.3. Diagram klasy `Dictionary`**Listing 17.5.** Dodawanie elementów do kolekcji typu `Dictionary<TKey, TValue>`

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

class Program
{
    static void Main()
    {
        // Kod dla wersji C# 6.0 (w starszych wersjach zastosuj polecenie {"Error", ConsoleColor.Red}).
        var colorMap = new Dictionary<string, ConsoleColor>
        {
            ["Error"] = ConsoleColor.Red,
            ["Warning"] = ConsoleColor.Yellow,
            ["Information"] = ConsoleColor.Green
        };

        colorMap.Add("Verbose", ConsoleColor.White);
        // ...
    }
}
```

Po zainicjowaniu słownika za pomocą inicjatora słowników z wersji C# 6.0 (zobacz podrozdział „Inicjatory kolekcji” w rozdziale 15.) kod z listingu 17.5 dodaje do kolekcji klucz "Verbose" o wartości ConsoleColor.White. Jeśli element o tym samym kluczu został już wcześniej dodany do kolekcji, kod zgłasza wyjątek.

Inny sposób dodawania elementów polega na użyciu indeksów, co pokazano na listingu 17.6.

Listing 17.6. Wstawianie elementów do kolekcji typu Dictionary< TKey, TValue > za pomocą operatora indeksowania

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

class Program
{
    static void Main()
    {
        // Kod dla wersji C# 6.0 (w starszych wersjach zastosuj składnię {"Error", ConsoleColor.Red}).
        var colorMap = new Dictionary<string, ConsoleColor>
        {
            ["Error"] = ConsoleColor.Red,
            ["Warning"] = ConsoleColor.Yellow,
            ["Information"] = ConsoleColor.Green
        };

        colorMap["Verbose"] = ConsoleColor.White;
        colorMap["Error"] = ConsoleColor.Cyan;

        // ...
    }
}
```

2.0

Pierwszą rzeczą, na jaką warto zwrócić uwagę na listingu 17.6, jest to, że operator indeksowania nie wymaga podawania liczby całkowitej. Typ operandu indeksu (tu jest to typ string) należy podać w pierwszym argumentem określającym typ. Typ wartości ustawianej lub pobieranej za pomocą indeksu (tu jest to typ ConsoleColor) jest podawany w drugim takim argumentem.

Drugą rzeczą wartą uwagi na listingu 17.6 jest dwukrotne użycie tego samego klucza ("Error"). W pierwszym przypisaniu z danym kluczem nie jest powiązana żadna wartość ze słownika. Wtedy klasa słownika wstawia nową wartość o podanym kluczu. W drugim przypisaniu element o podanym kluczu już istnieje. Wtedy kod nie wstawia dodatkowego elementu, ale zastępuje wcześniejszą wartość typu ConsoleColor powiązaną z kluczem "Error" nową wartością — ConsoleColor.Cyan.

Próba odczytu ze słownika wartości o nieistniejącym kluczu skutkuje zgłoszeniem wyjątku KeyNotFoundException. Metoda ContainsKey() umożliwia sprawdzenie przed dostępem do wartości, czy dany klucz jest używany. W ten sposób można uniknąć wyjątku.

Klasa `Dictionary<TKey, TValue>` jest zaimplementowana jako *tablica z haszowaniem*. Ta struktura danych zapewnia bardzo szybki dostęp, gdy dane są wyszukiwane na podstawie klucza. Nie ma przy tym znaczenia, ile wartości jest przechowywanych w słowniku. Jednak sprawdzanie, czy w słowniku występuje określona wartość, to czasochłonna operacja o złożoności liniowej (podobnie jak przeszukiwanie nieposortowanej listy). Do jej wykonywania służy metoda `ContainsValue()`, która po kolei sprawdza wszystkie elementy kolekcji.

Aby usunąć element ze słownika, wywołaj metodę `Remove()`. Należy przekazać do niej klucz, a nie wartość elementu.

Ponieważ dodanie wartości do słownika wymaga podania zarówno klucza, jak i wartości, zmienna pętli `foreach` iterującej po elementach słownika musi być typu `KeyValuePair<TKey, TValue>`. Na listingu 17.7 przedstawiono fragment kodu ilustrujący, jak za pomocą pętli `foreach` wyświetlić klucze i wartości ze słownika. Efekt wykonania kodu znajdziesz w danych wyjściowych 17.3.

Listing 17.7. Iterowanie po kolekcji typu `Dictionary<TKey, TValue>` za pomocą pętli `foreach`

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

class Program
{
    static void Main()
    {
        // Kod dla wersji C# 6.0 (we wcześniejszych wersjach zastosuj składnię {"Error", ConsoleColor.Red}).
        Dictionary<string, ConsoleColor> colorMap =
            new Dictionary<string, ConsoleColor>
        {
            ["Error"] = ConsoleColor.Red,
            ["Warning"] = ConsoleColor.Yellow,
            ["Information"] = ConsoleColor.Green,
            ["Verbose"] = ConsoleColor.White
        };

        Print(colorMap);
    }

    private static void Print(
        IEnumerable<KeyValuePair<string, ConsoleColor>> items)
    {
        foreach (KeyValuePair<string, ConsoleColor> item in items)
        {
            Console.ForegroundColor = item.Value;
            Console.WriteLine(item.Key);
        }
    }
}
```

DANE WYJŚCIOWE 17.3.

```
Error
Warning
Information
Verbose
```

Zauważ, że elementy pojawiają się tu w kolejności ich dodawania do słownika (tak jakby zostały dodane do listy). Wiele implementacji słowników iteruje po kluczach i wartościach zgodnie z kolejnością dodawania elementów, ale to podejście nie jest ani wymagane, ani opisane w dokumentacji, dlatego nie należy zakładać, że kod będzie tak działał.

Wskazówka

NIE przyjmuj niezasadnionych założeń dotyczących kolejności iterowania po elementach kolekcji. Jeśli w dokumentacji nie opisano, że iteracja powinna się odbywać w określonej kolejności, nie ma gwarancji, że elementy zostaną zwrócone w ustalonym porządku.

Jeśli interesują Cię tylko klucze lub tylko wartości z klasy słownika, możesz je pobrać za pomocą właściwości `Keys` lub `Values`. Zwracają one wartość typu `ICollection<T>` zawierającą referencje do danych z pierwotnej kolekcji, a nie kopię tych danych. Zmiany wprowadzone w słowniku są automatycznie odzwierciedlane w kolekcji zwracanej przez właściwości `Keys` i `Values`.

2.0

■ ZAGADNIENIE DLA ZAAWANSOWANYCH

Modyfikowanie sprawdzania równości w słowniku

Aby ustalić, czy dany klucz pasuje do któregoś z kluczy istniejących w słowniku, musi istnieć możliwość sprawdzania równości dwóch kluczy. Podobnie na listach potrzebna jest możliwość porównywania dwóch elementów w celu ustalenia ich kolejności (przykładowy kod znajdziesz w zagadnienniu dla zaawansowanych „Modyfikowanie procesu sortowania kolekcji” we wcześniejszej części rozdziału). Domyslnie dwie instancje typu bezpośredniego są porównywane na podstawie tego, czy obie zawierają dokładnie te same dane. W trakcie porównywania dwóch instancji typu referencyjnego domyslnie sprawdzane jest, czy prowadzą one do tego samego obiektu. Jednak czasem dwie instancje są uznawane za równe także wtedy, gdy nie mają dokładnie tej samej wartości lub nie zawierają referencji do tego samego obiektu.

Załóżmy na przykład, że chcesz utworzyć słownik typu `Dictionary<Contact, string>` na podstawie typu `Contact` z listingu 17.2. W takiej sytuacji dwa obiekty typu `Contact` można uznać za równe, jeśli zawierają to samo imię i to samo nazwisko. Referencje nie muszą być identyczne. Podobnie jak można udostępnić implementację interfejsu `IComparer<T>` na potrzeby sortowania listy, tak można utworzyć implementację interfejsu `IEqualityComparer<T>`, by umożliwić ustalenie, czy dwa klucze należy uznać za równe. Ten interfejs wymaga dwóch metod — jednej zwracającej informację o tym, czy dwa elementy są równe, i jednej zwracającej skrót, który słownik może wykorzystać do przyspieszenia indeksowania. Przykładowy kod pokazano na listingu 17.8.

Listing 17.8. Implementowanie interfejsu `IEqualityComparer<T>`

```

using System;
using System.Collections.Generic;

class ContactEquality : IEqualityComparer<Contact>
{
    public bool Equals(Contact? x, Contact? y)
    {
        if (Object.ReferenceEquals(x, y))
            return true;
        if (x == null || y == null)
            return false;
        return x.LastName == y.LastName &&
            x.FirstName == y.FirstName;
    }

    public int GetHashCode(Contact x)
    {
        if (x is null)
            return 0;
        int h1 = x.FirstName == null ? 0 : x.FirstName.GetHashCode();
        int h2 = x.LastName == null ? 0 : x.LastName.GetHashCode();
        return h1 * 23 + h2;
    }
}

```

2.0

Aby utworzyć słownik, który wykorzystuje ten kod do sprawdzania równości, wywołaj konstruktor `new Dictionary<Contact, string>(new ContactEquality)`.

ZAGADNIENIE DLA POCZĄTKUJĄCYCH

Wymogi związane ze sprawdzaniem równości

W rozdziale 10., „Dobrze uformowane typy”, wyjaśniono, że w algorytmach sprawdzających równość i generujących skróty obowiązuje kilka ważnych reguł. W kolekcjach przestrzeganie tych reguł jest bardzo ważne. Podobnie jak poprawne posortowanie listy wymaga tego, by niestandardowy kod do określania kolejności elementów zapewniał porządek całkowity, tak w tablicy z haszowaniem niestandardowy kod do sprawdzania równości musi gwarantować spełnienie określonych wymogów. Najważniejszy wymóg dotyczy tego, że jeśli metoda `Equals()` zwraca dla dwóch obiektów wartość `true`, metoda `GetHashCode()` musi zwracać na podstawie tych obiektów te same wartości. Zauważ, że w drugą stronę ten wymóg nie obowiązuje — dwa elementy uznane za różne mogą mieć ten sam skrót. W praktyce często muszą się pojawiać różne elementy o tych samych skrótach, ponieważ istnieją tylko 2^{32} możliwe skróty, a różnych obiektów może być o wiele więcej.

Drugi bardzo istotny wymóg dotyczy tego, że dwa wywołania metody `GetHashCode()` dla jednego elementu muszą dawać ten sam wynik dopóty, dopóki ten element znajduje się w tablicy z haszowaniem. Zauważ, że dwa obiekty „wyglądające na równe” nie muszą mieć tego samego skrótu w dwóch różnych przebiegach programu. Na przykład dopuszczalne jest, że dla danej osoby kontaktowej jednego dnia wygenerowany zostanie jeden skrót, a dwa

tygodnie później, gdy program zostanie ponownie uruchomiony, do tej samej osoby kontaktowej przypisany zostanie inny skrót. Nie należy utrzymać skrótów w bazie danych i oczekwać, że pozostaną stałe w różnych przebiegach programu.

Najlepiej jest, gdy wyniki generowane przez metodę `GetHashCode()` wyglądają na losowe. Oznacza to, że mała zmiana w danych wejściowych powinna prowadzić do istotnej zmiany w danych wyjściowych. Ponadto skróty powinny mieć rozkład równomierny (to znaczy, że wszystkie dostępne liczby całkowite powinny być używane równie często). Trudno jednak zaprojektować algorytm haszujący, który działa bardzo szybko i generuje dane wyjściowe o idealnym rozkładzie. Dlatego należy znaleźć dobry kompromis między tymi czynnikami.

Metody `GetHashCode()` i `Equals()` nie mogą zgłaszać wyjątków. Zwróć uwagę, że w kodzie z listingu 17.8 zadbanio na przykład o to, by nigdy nie wykonywać dereferencji dla referencji o wartości null.

Poniżej znajduje się podsumowanie najważniejszych reguł:

- Równe sobie obiekty muszą mieć równe skróty.
- Skrót obiektu nie powinien się zmieniać w czasie życia instancji (przynajmniej dopóty, dopóki znajduje się ona w tablicy z haszowaniem).
- Algorytm haszujący powinien szybko generować skróty o dobrym rozkładzie.
- Algorytm haszujący nie powinien zgłaszać wyjątków, gdy stan obiektu jest dopuszczalny.

2.0

Kolekcje posortowane

— `SortedDictionary< TKey, TValue >` i `SortedList< T >`

Klasy kolekcji posortowanych (zobacz rysunek 17.4) przechowują elementy posortowane według kluczy (klasa `SortedDictionary< TKey, TValue >`) lub według wartości (klasa `SortedList< T >`). Jeśli zmodyfikujesz kod na listingu 17.7 i zamiast klasy `Dictionary< string, string >` zastosujesz klasę `SortedDictionary< string, string >`, program wyświetli informacje przedstawione w danych wyjściowych 17.4.

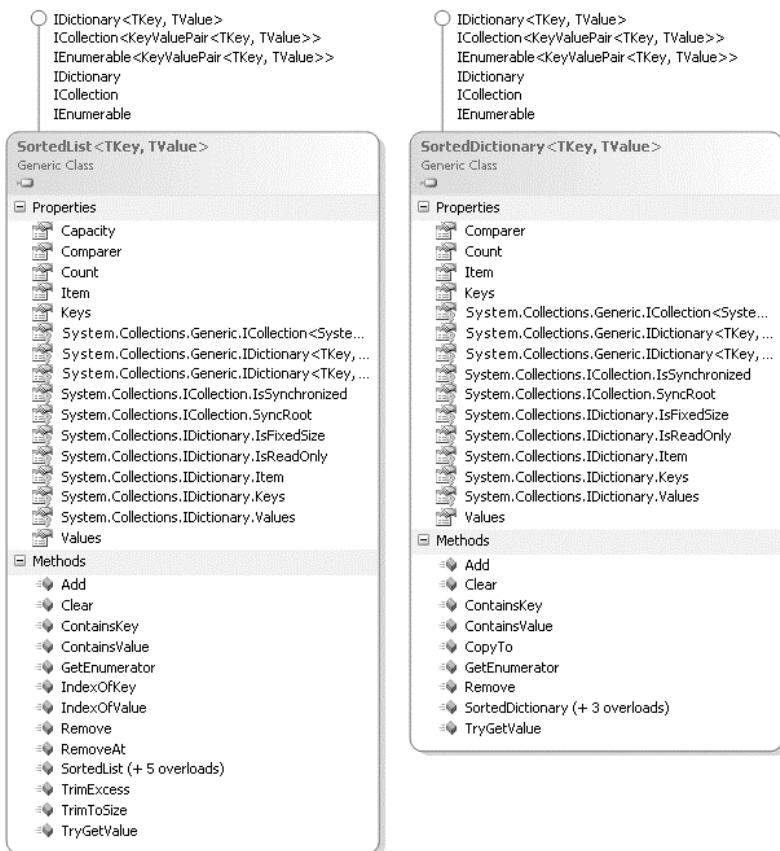
DANE WYJŚCIOWE 17.4.

Error
Information
Verbose
Warning

Zauważ, że elementy są sortowane na podstawie kluczy, a nie według wartości.

Ponieważ posortowane kolekcje muszą wykonywać dodatkowe operacje, by zapewnić odpowiednią kolejność elementów, wstawianie i usuwanie wartości odbywa się w nich zwykle wolniej niż wykonywanie analogicznych zadań w słowniku nieuporządkowanym.

Kolekcje posortowane muszą przechowywać elementy w określonym porządku, dlatego dostęp można uzyskać zarówno na podstawie kluczy, jak i za pomocą indeksu. Aby na posortowanej liście uzyskać dostęp do klucza lub wartości na podstawie indeksu, wykorzystaj właściwości `Keys` i `Values`. Zwracają one instancje typów `IList< TKey >` i `IList< TValue >`. W wynikowej kolekcji można używać indeksów w taki sam sposób jak w dowolnej innej liście.



Rysunek 17.4. Klasa sortowanych kolekcji

2.0

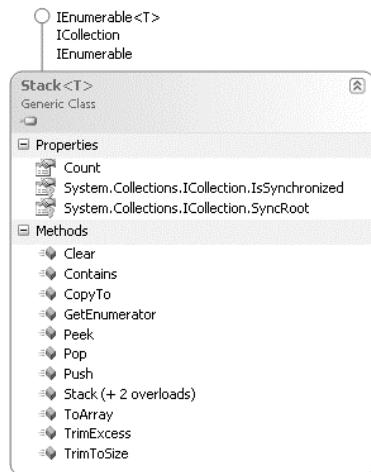
Kolekcje w postaci stosu — Stack<T>

W rozdziale 12. opisano klasy kolekcji w postaci stosu (zobacz rysunek 17.5). Takie klasy są projektowane jako kolekcje LIFO (ang. *last in, first out*, czyli „ostatni na wejściu, pierwszy na wyjściu”). Dwie najważniejsze metody tej klasy to Push() i Pop().

- Metoda Push() dodaje elementy do kolekcji. Te elementy nie muszą być unikatowe.
- Metoda Pop() usuwa elementy w kolejności odwrotnej do ich dodawania.

Aby uzyskać dostęp do elementów stosu bez modyfikowania go, należy zastosować metody Peek() i Contains(). Metoda Peek() zwraca następny element, który zostanie pobrany za pomocą metody Pop().

Metoda Contains() (podobnie jak w większości klas kolekcji) służy do ustalania, czy na stosie znajduje się określony element. Tak jak we wszystkich kolekcjach, można wykorzystać pętlę foreach, by iteracyjnie pobrać wszystkie elementy stosu. W ten sposób można uzyskać dostęp do wartości z dowolnego miejsca stosu. Zauważ jednak, że dostęp do wartości za pomocą pętli foreach nie powoduje usuwania elementów ze stosu. Elementy są usuwane tylko za pomocą metody Pop().

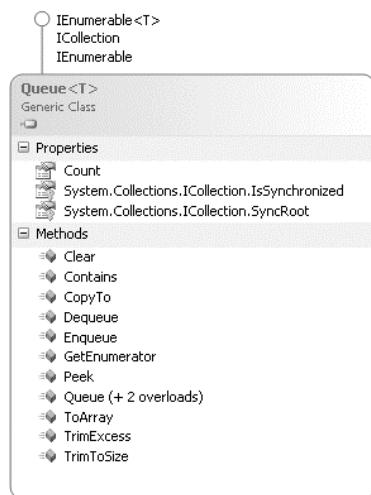


Rysunek 17.5. Diagram klasy Stack<T>

2.0

Kolekcje w postaci kolejek — Queue<T>

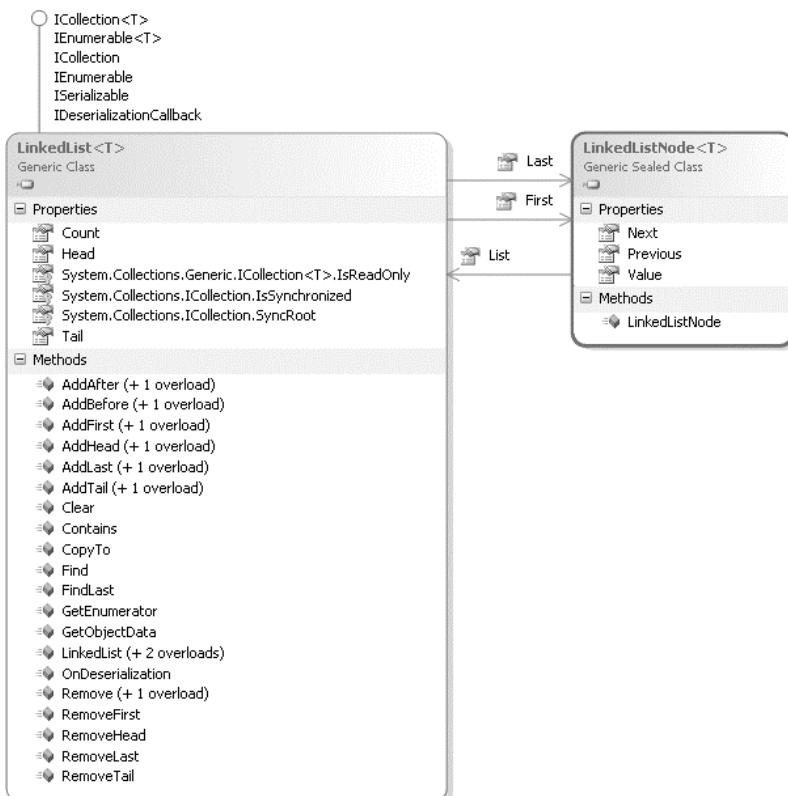
Klasy kolekcji w postaci kolejek (zobacz rysunek 17.6) są bardzo podobne do klas stosu, natomiast działając w modelu FIFO (ang. *first in, first out*, czyli „pierwszy na wejściu, pierwszy na wyjściu”). Zamiast metod Pop() i Push() w kolejkach używane są metody Enqueue() i Dequeue(). Kolejka działa jak rura — na jednym końcu umieszczasz obiekty w kolejce (za pomocą metody Enqueue()), a na drugim końcu je usuwasz (przy użyciu metody Dequeue()). Podobnie jak na stosie, obiekty w kolejce nie muszą być unikatowe. Pojemność kolejki automatycznie rośnie wraz ze wzrostem liczby elementów. Gdy liczba elementów się zmniejsza, nie trzeba odzyskiwać wcześniej zajmowanej pamięci, ponieważ zwiększa to potencjalnie koszt dodania nowego elementu. Jeśli jednak wiesz, że kolejka przez dłuższy czas będzie zawierała tę samą liczbę elementów, możesz za pomocą metody TrimToSize() przekazać kolejce wskazówkę informującą, że warto odzyskać pamięć.



Rysunek 17.6. Diagram klasy Queue<T>

Listy powiązane — `LinkedList<T>`

Przestrzeń nazw `System.Collections.Generic` obejmuje też kolekcję w postaci listy powiązanej, umożliwiającą poruszanie się po kolekcji w przód i w tył. Diagram klasy listy powiązanej pokazano na rysunku 17.7 (nie istnieje analogiczny typ niegeneryczny).



Rysunek 17.7. Diagramy klas `LinkedList<T>` i `LinkedListNode<T>`

2.0

Udostępnianie indeksera

Tablice, słowniki i listy udostępniają **indekser** jako wygodny sposób do pobierania lub ustawiania elementów kolekcji na podstawie klucza bądź indeksu. Widziałeś już, że aby zastosować indekser, należy podać indeks (lub indeksy) w nawiasie kwadratowym po nazwie kolekcji. Możesz też zdefiniować własny indekser. Na listingu 17.9 pokazano przykładowy indekser dla typu `Pair<T>`.

Listing 17.9. Definiowanie indeksera

```

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

```

```

    T this[PairItem index] { get; }
}
public enum PairItem
{
    First,
    Second
}
public struct Pair<T> : IPair<T>
{
    public Pair(T first, T second)
    {
        First = first;
        Second = second;
    }
    public T First { get; } // Automatycznie generowana właściwość z samym getterem (z wersji C# 6.0).
    public T Second { get; } // Automatycznie generowana właściwość z samym getterem (z wersji C# 6.0).

    public T this[PairItem index]
    {
        get
        {
            switch (index)
            {
                case PairItem.First:
                    return First;
                case PairItem.Second:
                    return Second;
                default :
                    throw new NotImplementedException(
                        $"Wyliczenie nie zawiera wartości { index.ToString() }.");
            }
        }
    }
}

```

2.0

Indekser jest deklarowany w podobny sposób jak właściwość. Różnica polega na tym, że zamiast nazwy właściwości należy podać słowo kluczowe `this` i listę parametrów w nawiasie kwadratowym. Ciało indeksera też wygląda jak właściwość — obejmuje bloki `get` i `set`. Na lisingu 17.9 pokazano, że parametr nie musi być typu `int`. Indeks może przyjmować wiele parametrów, a nawet być przeciążony. W tym przykładzie zastosowano typ wyliczeniowy (`enum`), by zmniejszyć prawdopodobieństwo podania w jednostce wywołującej indeksu nieistniejącego elementu.

Kod CIL generowany przez kompilator języka C# na podstawie operatora indeksowania to specjalna właściwość `Item` przyjmująca argument. W języku C# nie można bezpośrednio tworzyć właściwości przyjmujących argumenty, dlatego właściwość `Item` jest pod tym względem wyjątkowa. Wszelkie inne składowe o identyfikatorze `Item` (nawet jeśli mają zupełnie inną sygnaturę) powodują konflikt ze składową wygenerowaną przez kompilator, dlatego nie można ich tworzyć.

ZAGADNIENIE DLA ZAAWANSOWANYCH

Określanie nazwy indeksera za pomocą atrybutu IndexerName

Jak wspomniano wcześniej, w kodzie CIL indekser domyślnie ma nazwę Item. Za pomocą atrybutu IndexerName można podać inną nazwę. Na listingu 17.10 jest ona ustawiana na "Entry".

Listing 17.10. Zmienianie domyślnej nazwy indeksera

```
[System.Runtime.CompilerServices.IndexerName("Entry")]
public T this[params PairItem[] branches]
{
    // ...
}
```

W kodzie w języku C# w jednostkach wywołujących indeks ta zmiana nie ma znaczenia. Nową nazwę należy jednak stosować w językach, które nie obsługują bezpośrednio indeksów.

Atrybut IndexerName jest dla kompilatora instrukcją, że należy zastosować dla indeksera inną nazwę. Sam atrybut nie jest dodawany przez kompilator do metadanych, dlatego nie jest też dostępny za pomocą mechanizmu refleksji.

2.0

ZAGADNIENIE DLA ZAAWANSOWANYCH

Definiowanie operatora indeksowania przyjmującego różną liczbę parametrów

Operator indeksowania może przyjmować różną liczbę parametrów. Na listingu 17.11 zdefiniowano operator indeksowania dla typu BinaryTree<T> (opisanego w rozdziale 12. oraz ponownie w następnym podrozdziale).

Listing 17.11. Definiowanie operatora indeksowania przyjmującego różną liczbę parametrów

```
using System;

public class BinaryTree<T>
{
    // ...

    public BinaryTree<T> this[params PairItem[] ?branches]
    {
        get
        {
            BinaryTree<T> currentNode = this;

            // Umożliwia przedstawianie korzenia
            // za pomocą pustej tablicy lub wartości null.
            int totalLevels = branches?.Length ?? 0;
            int currentLevel = 0;

            while (currentLevel < totalLevels)
            {

```

```
System.Diagnostics.Debug.Assert(branches != null,
    $"'{ nameof(branches) } != null'");
currentNode = currentNode.SubItems[
    branches[currentLevel]];
if (currentNode == null)
{
    // W tym miejscu drzewo binarne ma wartość null.
    throw new IndexOutOfRangeException();
}
currentLevel++;
}
return currentNode;
}
}
```

Każdy element w parametrze branches to wartość typu `PairItem`, określająca, które odgałęzienie należy wybrać w trakcie poruszania się w dół drzewa binarnego. Spójrz na przykładowy kod:

2.0

```
tree[PairItem.Second, PairItem.First].Value
```

Pobiera on wartość elementu dostępnego po wybraniu najpierw drugiej gałęzi, a następnie pierwszej gałęzi (zaczynając od wartości wyjściowej).

Zwracanie wartości null lub pustej kolekcji

Gdy zwracasz tablicę lub kolekcję, to aby określić, że liczba elementów wynosi zero, należy zwrócić wartość `null` lub instancję kolekcji bez elementów. Zwykle lepszym rozwiązaniem jest zwrócenie pustej instancji kolekcji. Wtedy jednostka wywołująca przed rozpoczęciem iterowania po elementach kolekcji nie musi sprawdzać, czy jest ona różna od `null`. Na przykład gdy używana jest kolekcja typu `IEnumerable<T>` o zerowej długości, jednostka wywołująca może natychmiast i bezpiecznie zastosować pętlę `foreach` do tej kolekcji. Nie występuje wtedy ryzyko, że wygenerowane wywołanie metody `GetEnumerator()` spowoduje zgłoszenie wyjątku `NullReferenceException`. Rozważ zastosowanie metody `Enumerable.Empty<T>()` do łatwego generowania pustych kolekcji danego typu.

Jedna z nielicznych sytuacji, w których nie należy stosować się do opisanego zalecenia, dotyczy scenariusza, gdy wartość `null` oznacza coś innego niż zero elementów. Na przykład wartość `null` kolekcji nazw użytkowników w witrynie może oznaczać, że z jakichś przyczyn nie można pobrać aktualnej kolekcji. Nie oznacza to tego samego co pusta kolekcja.

Wskazówki

NIE reprezentuj pustych kolekcji za pomocą referencji `null`.

ROZWAŻ zastosowanie zamiast tego metody `Enumerable.Empty<T>()`.

Iteratory

W rozdziale 15. szczegółowo opisano wewnętrzne mechanizmy pętli `foreach`. W tym podrozdziale wyjaśniono, jak stosować **iteratory** do tworzenia własnych implementacji interfejsów `IEnumerator<T>`, `IEnumerable<T>` oraz analogicznych niegenerycznych interfejsów niesandardowych kolekcji. Iteratory zapewniają wygodną składnię pozwalającą określić, jak iterować po danych z klas kolekcji (przede wszystkim przy użyciu pętli `foreach`). Iterator umożliwia użytkownikom kolekcji poruszanie się po jej wewnętrznej strukturze bez jej znajomości.

ZAGADNIENIE DLA ZAAWANSOWANYCH

Początki iteratorów

W 1972 roku Barbara Liskov i zespół naukowców z uczelni MIT zaczęli badania nad metodami programowania, koncentrując się na definiowanych przez użytkowników abstrakcjach danych. Aby udowodnić wiele swoich odkryć, zespół opracował język CLU, w którym używane były klastry (*CLU* to trzy pierwsze litery angielskiej nazwy *clusters*, czyli właśnie klastry). Klastry były poprzednikiem podstawowych abstrakcji danych, z których programiści korzystają obecnie (czyli obiektów).

W trakcie badań zespół odkrył, że choć za pomocą języka CLU można przedstawić w formie abstrakcyjnej niektóre dane i ukryć je przed użytkownikami, często trzeba ujawnić wewnętrzną strukturę danych, by umożliwić innym właściwe posługiwanie się nimi. Ze względu na ten dylemat zespół opracował w języku konstrukcję nazwaną *iteratorem*. Język CLU udostępniał wiele nowinek, które w późniejszym czasie zostały spopularyzowane w ramach programowania obiektowego.

Jeśli klasy mają umożliwiać iterowanie za pomocą pętli `foreach`, trzeba w nich zaimplementować wzorzec „enumerator”. W rozdziale 15. opisano, że w języku C# pętla `foreach` jest rozwijana przez kompilator jako pętla `while` oparta na obiekcie z implementacją interfejsu `IEnumerator<T>`, który jest pobierany za pomocą interfejsu `IEnumerable<T>`.

Problem z wzorcem „enumerator” polega na tym, że ręczne implementowanie go jest kłopotliwe, ponieważ trzeba przechowywać cały stan niezbędnego do określenia bieżącej pozycji w kolekcji. Ten wewnętrzny stan może być prosty w klasach kolekcji — wystarczy wtedy zachować indeks określający bieżącą pozycję. Natomiast w rekurencyjnych strukturach danych, na przykład w drzewach binarnych, stan bywa skomplikowany. Aby ograniczyć trudności związane z implementowaniem tego wzorca, w języku C# 2.0 dodano konstrukcję, która ułatwia określanie w klasie sposobu iterowania po jej zawartości za pomocą pętli `foreach`.

Definiowanie iteratora

Iteratory służą do implementowania metod klasy i są składniowym skrótem umożliwiającym implementację bardziej skomplikowanego wzorca „enumerator”. Gdy kompilator języka C# natrafi na iterator, rozwija jego zawartość w kod CIL z implementacją wzorca „enumerator”. Dlatego w środowisku uruchomieniowym nie ma mechanizmów służących

do obsługi iteratorów. Ponieważ kompilator języka C# zapewnia potrzebną implementację w wyniku wygenerowania kodu CIL, stosowanie iteratorów nie zapewnia wzrostu wydajności w trakcie wykonywania kodu. Jednak wykorzystanie iteratorów zamiast ręcznego implementowania wzorca „enumerator” pozwala uzyskać znaczący wzrost wydajności programistów. Aby zrozumieć usprawnienia, najpierw przyjrzyj się temu, jak iterator jest zdefiniowany w kodzie.

Składnia iteratora

Iterator pozwala w skróconej formie zaimplementować interfejsy iteratora (czyli parę interfejsów `IEnumerable<T>` i `IEnumerator<T>`). Na listingu 17.12 zadeklarowano iterator dla generycznego typu `BinaryTree<T>`, tworząc metodę `GetEnumerator()`. Dalej zobaczysz, jak dodać obsługę interfejsów iteratora.

Listing 17.12. Wzorzec oparty na interfejsach iteratora

```
2.0
using System;
using System.Collections.Generic;

public class BinaryTree<T> : IEnumerable<T>
{
    public BinaryTree ( T value )
    {
        Value = value;
    }

    #region IEnumerable<T>
    public IEnumerator<T> GetEnumerator()
    {
        // ...
    }
    #endregion IEnumerable<T>

    public T Value { get; } // Automatycznie implementowana właściwość z samym getterem (kod dla wersji C# 6.0).
    public Pair<BinaryTree<T>> SubItems { get; set; }
}

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

    public T First { get; } // Automatycznie implementowana właściwość z samym getterem (kod dla wersji C# 6.0).
    public T Second { get; } // Automatycznie implementowana właściwość z samym getterem (kod dla wersji C# 6.0).
}
```

Na listingu 17.12 pokazano, że należy dodać implementację metody `GetEnumerator()`.

Zwracanie wartości przez iterator

Iteratory są jak funkcje, ale zamiast zwracać jedną wartość, generują sekwencję wartości (jedna po drugiej). W klasie `BinaryTree<T>` iterator zwraca sekwencję wartości o typie podanym w parametrze `T`. Jeśli używana jest niegeneryczna wersja interfejsu `IEnumerator`, zwarcane wartości są typu `object`.

Aby poprawnie zaimplementować wzorzec „iterator”, należy przechowywać wewnętrzny stan i śledzić pozycję w trakcie poruszania się po kolekcji. W klasie `BinaryTree<T>` należy rejestrować, które elementy drzewa zostały już pobrane, a które jeszcze nie. Iteratory są przekształcane przez kompilator na maszyny stanowe, które śledzą bieżącą pozycję i wiedzą, jak przejść do następnego elementu.

Instrukcja `yield return` zwraca wartość za każdym razem, gdy iterator napotka tę instrukcję. Wtedy sterowanie natychmiast jest zwracane do jednostki wywołującej, która zażądała danego elementu. Ciekawe jest to, że sterowanie rzeczywiście jest zwracane *natychmiast*, inaczej — paradoksalnie — niż w przypadku instrukcji `return`. Instrukcja `return` powoduje wcześniejsze wykonanie bloków `finally`, czego instrukcja `yield return` nie robi. Gdy jednostka wywołująca zażąda następnego elementu, wykonywany będzie kod znajdujący się bezpośrednio po wcześniej wykonanej instrukcji `yield return`. Kod z listingu 17.13 po kolej zwraca słowa kluczowe reprezentujące wbudowane typy języka C#.

Listing 17.13. Zwracanie po kolej wybranych słów kluczowych z języka C#

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

public class CSharpBuiltInTypes: IEnumerable<string>
{
    public IEnumerator<string> GetEnumerator()
    {
        yield return "object";
        yield return "byte";
        yield return "uint";
        yield return "ulong";
        yield return "float";
        yield return "char";
        yield return "bool";
        yield return "ushort";
        yield return "decimal";
        yield return "int";
        yield return "sbyte";
        yield return "short";
        yield return "long";
        yield return "void";
        yield return "double";
        yield return "string";
    }

    // Metoda IEnumerable.GetEnumerator jest potrzebna, ponieważ
    // interfejs IEnumerable<T> jest pochodny od interfejsu IEnumerable.
    System.Collections.IEnumerator
        System.Collections.IEnumerable.GetEnumerator()
    {
    }
}
```

```
// Wywołuje przedstawioną wcześniej metodę IEnumarator<string> GetEnumerator().
return GetEnumerator();
}

public class Program
{
    static void Main()
    {
        var keywords = new CSharpBuiltInTypes();
        foreach (string keyword in keywords)
        {
            Console.WriteLine(keyword);
        }
    }
}
```

Wyniki działania kodu z listingu 17.13 pokazano w danych wyjściowych 17.5.

DANE WYJŚCIOWE 17.5.

2.0

```
object
byte
uint
ulong
float
char
bool
ushort
decimal
int
sbyte
short
long
void
double
string
```

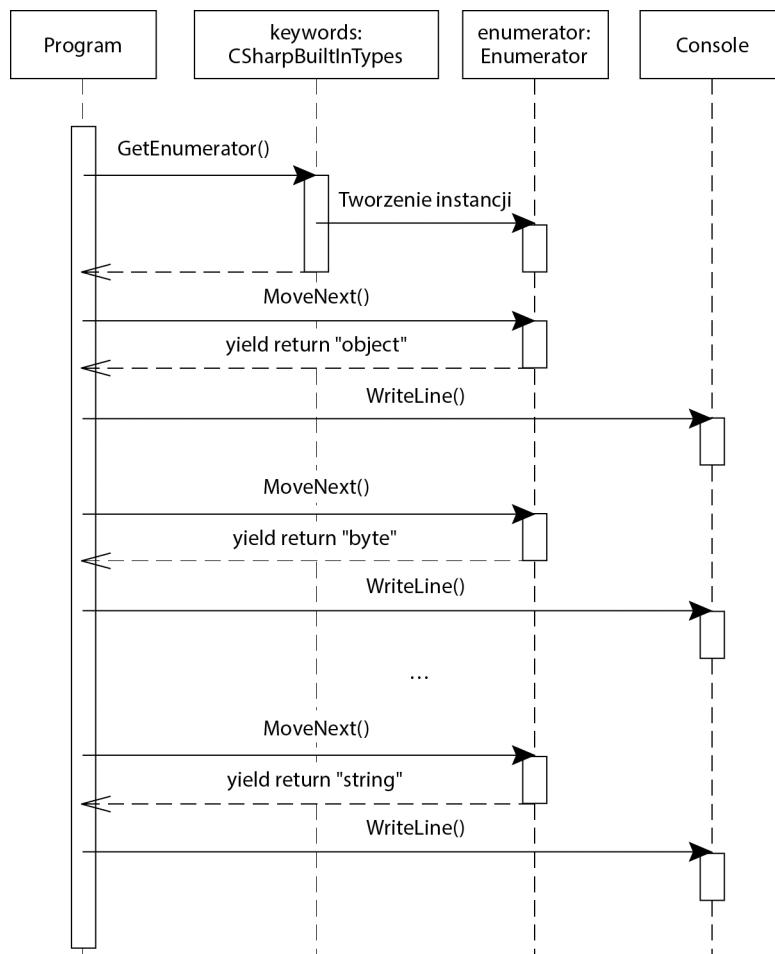
Dane wyjściowe z tego listingu to lista wbudowanych typów języka C#.

Iteratory i stan

Gdy w pętli foreach (na przykład foreach (string keyword in keywords) z listingu 17.13) po raz pierwszy wywoływana jest metoda GetEnumerator(), tworzony jest obiekt iteratora, a jego stan zostaje zainicjowany specjalną wartością początkową, informującą, że w iteratorze nie wykonano jeszcze żadnego kodu, a tym samym nie zwrócono żadnych wartości. Iterator przechowuje stan tak długo, jak długo wykonywana jest dana instrukcja foreach. Za każdym razem, gdy pętla żąda następnej wartości, sterowanie jest przekazywane do iteratora. Iterator wznowia wtedy pracę w miejscu, w którym ją zakończył w poprzedniej iteracji pętli. Informacje o stanie przechowywane w obiekcie iteratora pozwalają określić, gdzie należy wznowić pracę. Gdy instrukcja foreach zakończy działanie, stan iteratora nie jest już dłużej przechowywany.

Zawsze można bezpiecznie ponownie wywołać metodę `GetEnumerator()`. W razie potrzeby tworzone są nowe obiekty iteratora.

Na rysunku 17.8 pokazano ogólny diagram sekwencyjny ilustrujący wykonywane operacje. Pamiętaj, że metoda `MoveNext()` pochodzi z interfejsu `IEnumerator<T>`.



2.0

Rysunek 17.8. Diagram sekwencyjny z instrukcją `yield return`

Na listingu 17.13 instrukcja `foreach` w miejscu wywołania uruchamia metodę `GetEnumerator()` obiektu `keywords` typu `CSharpBuiltInTypes`. Gdy dostępny jest iterator (wskaazywany za pomocą nazwy `iterator`), pętla `foreach` rozpoczęta każdą iterację wywołaniem metody `MoveNext()`. W iteratorze wartość zwracana jest do miejsca wywołania instrukcji `foreach`. Po wykonaniu instrukcji `yield return` metoda `GetEnumerator()` wstrzymuje pracę do czasu otrzymania następnego żądania `MoveNext()`. W ciele pętli instrukcja `foreach` wyświetla zwróconą wartość na ekranie. Następnie kod wraca na początek pętli, co powoduje ponowne wywołanie metody `MoveNext()` iteratora. Zauważ, że w drugiej iteracji kod przechodzi do drugiej instrukcji `yield return`. Pętla `foreach` ponownie wyświetla wtedy na ekranie zwróconą wartość z klasy

CSharpBuiltInTypes i rozpoczyna następną iterację. Ten proces jest ponawiany do momentu, w którym w iteratorze nie ma już więcej instrukcji `yield return`. Na tym etapie pętla `foreach` kończy działanie, ponieważ metoda `MoveNext()` zwraca wartość `false`.

Inne przykładowe iteratory

Zanim zmodyfikujesz klasę `BinaryTree<T>`, musisz wprowadzić zmiany w typie `Pair<T>`, by za pomocą iteratora dodać obsługę interfejsu `IEnumerable<T>`. Na listingu 17.14 pokazano przykładowy kod zwracający każdy element z obiektu typu `Pair<T>`.

Listing 17.14. Używanie instrukcji `yield` do zaimplementowania typu `BinaryTree<T>`

```
2.0
public struct Pair<T>: IPair<T>,
    IEnumerable<T>
{
    public Pair(T first, T second) : this()
    {
        First = first;
        Second = second;
    }
    public T First { get; } // Automatycznie generowana właściwość z samym getterem (kod dla wersji C# 6.0).
    public T Second { get; } // Automatycznie generowana właściwość z samym getterem (kod dla wersji C# 6.0).

    #region IEnumerable<T>
    public IEnumerator<T> GetEnumerator()
    {
        yield return First;
        yield return Second;
    }
    #endregion IEnumerable<T>

    #region Składowe interfejsu IEnumerable
    System.Collections.IEnumerator
        System.Collections.IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }
    #endregion
}
```

Na listingu 17.14 iteracja po danych typu `Pair<T>` obejmuje dwa powtórzenia pętli — pierwsze powiązane z instrukcją `yield return First` i drugie powiązane z instrukcją `yield return Second`. Za każdym razem, gdy kod natrafi na instrukcję `yield return` w metodzie `GetEnumerator()`, stan jest zachowywany, a kod „przeskakuje” z metody `GetEnumerator()` do ciała pętli. Po rozpoczęciu drugiej iteracji pętli metoda `GetEnumerator()` wznowia działanie od instrukcji `yield return Second`.

Typ `System.Collections.Generic.IEnumerable<T>` dziedziczy po typie `System.Collections.IEnumerable`. Dlatego gdy implementowany jest interfejs `IEnumerable<T>`, trzeba też zaimplementować interfejs `IEnumerable`. Na listingu 17.14 odbywa się to jawnie, a implementacja polega na wywołaniu metody `GetEnumerator()` z interfejsu `IEnumerable<T>`. Wywołanie metody `IEnumerable<T>.GetEnumerator()` w metodzie `IEnumerator.GetEnumerator()`

zawsze jest możliwe. Dzieje się tak dzięki (wynikającej z dziedziczenia) zgodności typów `IEnumerable<T>` i `IEnumerable`. Ponieważ sygnatury obu metod `GetEnumerator()` są identyczne (inne typy zwracanych wartości nie różnią się sygnaturą), potrzebna jest jawnia implementacja jednej z tych metod (lub obu).

Na listingu 17.15 kod wywołuje metodę `Pair<T>.GetEnumerator()` i wyświetla w dwóch kolejnych wierszach słowa "Inigo" i "Montoya".

Listing 17.15. Używanie metody `Pair<T>.GetEnumerator()` za pomocą pętli `foreach`

```
var fullname = new Pair<string>("Inigo", "Montoya");
foreach (string name in fullname)
{
    Console.WriteLine(name);
}
```

Zauważ, że wywołanie metody `GetEnumerator()` odbywa się niejawnie, za pośrednictwem pętli `foreach`.

Umieszczanie instrukcji `yield return` w pętli

Nie trzeba zapisywać na stałe każdej instrukcji `yield return`, co miało miejsce w typach `CSharpBuiltInTypes` i `Pair<T>`. Za pomocą instrukcji `yield return` można zwracać wartości w pętli. Na listingu 17.16 używana jest pętla `foreach`. Pętla `foreach` za każdym razem, gdy jest wykonywana w metodzie `GetEnumerator()`, zwraca następną wartość.

Listing 17.16. Umieszczanie instrukcji `yield return` w pętli

```
public class BinaryTree<T>: IEnumerable<T>
{
    // ...

    #region IEnumerable<T>
    public IEnumerator<T> GetEnumerator()
    {
        // Zwracanie elementu z danego węzła.
        yield return Value;

        // Iterowanie po każdym elemencie z pary.
        foreach (BinaryTree<T>? tree in SubItems)
        {
            if (tree != null)
            {
                // Ponieważ każdy element z pary to drzewo,
                // należy przejść po tym drzewie i zwrócić każdy element.
                foreach (T item in tree)
                {
                    yield return item;
                }
            }
        }
    }
    #endregion IEnumerable<T>
```

```
#region Składowe interfejsu IEnumerable
System.Collections.IEnumerator
    System.Collections.IEnumerable.GetEnumerator()
{
    return GetEnumerator();
}
#endregion
}
```

Na listingu 17.16 pierwsza iteracja zwraca korzeń drzewa binarnego. W drugiej iteracji kod przetwarza parę podelementów. Jeśli zawiera ona wartość różną od null, należy przejść do danego węzła podzielnego i zwrócić jego elementy. Zauważ, że pętla foreach (`T item in tree`) zawiera rekurencyjne wywołanie skierowane do węzła podzielnego.

Teraz można iterować po zawartości obiektów typu `BinaryTree<T>` za pomocą pętli foreach (wcześniej podobne rozwiązania pokazano dla klas `CSharpBuiltInTypes` i `Pair<T>`). Proces ten zaprezentowano na listingu 17.17, a wynik działania kodu znajdziesz w danych wyjściowych 17.6.

Listing 17.17. Używanie pętli foreach dla obiektów typu `BinaryTree<string>`

2.0

```
// John Fitzgerald Kennedy
var jfkFamilyTree = new BinaryTree<string>(
    "John Fitzgerald Kennedy")
{
    SubItems = new Pair<BinaryTree<string>>(
        new BinaryTree<string>("Joseph Patrick Kennedy")
    {
        // Dziadkowie ze strony ojca.
        SubItems = new Pair<BinaryTree<string>>(
            new BinaryTree<string>("Patrick Joseph Kennedy"),
            new BinaryTree<string>("Mary Augusta Hickey"))
    },
    new BinaryTree<string>("Rose Elizabeth Fitzgerald")
    {
        // Dziadkowie ze strony matki.
        SubItems = new Pair<BinaryTree<string>>(
            new BinaryTree<string>("John Francis Fitzgerald"),
            new BinaryTree<string>("Mary Josephine Hannon"))
    })
};

foreach (string name in jfkFamilyTree)
{
    Console.WriteLine(name);
}
```

DANE WYJŚCIOWE 17.6.

```
John Fitzgerald Kennedy
Joseph Patrick Kennedy
Patrick Joseph Kennedy
Mary Augusta Hickey
Rose Elizabeth Fitzgerald
John Francis Fitzgerald
Mary Josephine Hannon
```

ZAGADNIENIE DLA ZAAWANSOWANYCH

Zagrożenia związane z iteratorami rekurencyjnymi

Kod z listingu 17.16 tworzy nowe iteratory zagnieżdżone, gdy porusza się po drzewie binarnym. W efekcie gdy wartość jest zwracana przez węzeł, zostaje zwrócona przez iterator danego węzła, a następnie przez iterator węzła nadzielnego, węzła nadzielnego od niego i tak dalej — aż do iteratora korzenia. Wartość zagnieżdżona na n -tym poziomie musi zostać przekazana w łańcuchu obejmującym n iteratorów. Jeśli dane drzewo binarne jest stosunkowo płytkie, przekazywanie wartości zwykle nie stanowi problemu. Jednak niezrównoważone drzewa binarne bywają bardzo głębokie, dlatego poruszanie się po nich za pomocą rekurencji bywa kosztowne.

Wskazówka

ROZWAŻ stosowanie nierekurencyjnych algorytmów, gdy poruszasz się po potencjalnie głębokich strukturach danych.

2.0

ZAGADNIENIE DLA POCZĄTKUJĄCYCH I ZAAWANSOWANYCH

Struktury a klasy

Ciekawym efektem ubocznym zdefiniowania typu `Pair<T>` jako struktury (`struct`), a nie klasy (`class`), jest to, że nie można bezpośrednio przypisywać wartości do właściwości `SubItems`. `→First` i `SubItems.Second`, i to nawet po utworzeniu publicznego settera. Jeśli utworzysz publiczny setter, poniższy kod spowoduje błąd komplikacji z informacją, że nie można zmodyfikować wartości `SubItems`, ponieważ nie jest zmienną:

```
jfkFamilyTree.SubItems.First =
    new BinaryTree<string>(
        "Joseph Patrick Kennedy");
```

Problem wynika z tego, że właściwość `SubItems` jest typu `Pair<T>`, który jest strukturą. Dlatego gdy właściwość `SubItems` zwraca wartość, tworzona jest kopia danych. Przypisanie właściwości `First` do kopii, która zostanie usunięta na końcu instrukcji, byłoby mylące. Na szczęście kompilator języka C# zapobiega temu błędu.

Aby rozwiązać ten problem, nie przypisuj wartości właściwości `First` (zobacz technikę zastosowaną na listingu 17.17), zadeklaruj typ `Pair<T>` jako klasę, a nie jako strukturę, nie twórz właściwości `SubItems` i zamiast niej zastosuj pole lub udostępnij w typie `BinaryTree<T>` właściwości zapewniające bezpośredni dostęp do składowych właściwości `SubItems`.

Anulowanie dalszych iteracji za pomocą instrukcji `yield break`

Czasem programista chce anulować dalsze iteracje. Można to zrobić, dodając instrukcję `if`, tak by kolejne instrukcje w kodzie nie były wykonywane. Inne rozwiązanie to wywołanie instrukcji `yield break`, by metoda `MoveNext()` zwracała wartość `false`. Wtedy sterowanie jest

natychmiast przekazywane do jednostki wywołującej, a pętla kończy pracę. Na listingu 17.18 pokazano przykładową metodę z taką instrukcją.

Listing 17.18. Unikanie iteracji za pomocą instrukcji yield break

```
public System.Collections.Generic.IEnumerable<T>
    GetNotNullEnumerator()
{
    if ((First == null) || (Second == null))
    {
        yield break;
    }
    yield return Second;
    yield return First;
}
```

Ta metoda anuluje iterowanie, jeśli któryś z elementów z klasy `Pair<T>` ma wartość `null`.

2.0

Instrukcja `yield break` działa podobnie jak wywołanie instrukcji `return` na początku funkcji, gdy wiadomo, że nie trzeba wykonywać już żadnych zadań. W ten sposób można pominąć dalsze iteracje bez umieszczenia całego pozostałoego kodu w bloku `if`. Pozwala to dodać wiele punktów wyjścia. Stosuj jednak tę technikę ostrożnie, ponieważ przypadkowa osoba czytająca kod może przeoczyć wcześniejsze zakończenie pracy metody.

■ ZAGADNIENIE DLA ZAAWANSOWANYCH

Jak działają iteratory

Gdy kompilator języka C# natrafi na iterator, rozwija go do postaci kodu CIL odpowiedniego dla wzorca „enumerator”. W wygenerowanym kodzie kompilator najpierw tworzy zagnieżdzoną klasę prywatną z implementacją interfejsu `IEnumerable<T>` (czyli z właściwością `Current` i metodą `MoveNext()`). Właściwość `Current` zwraca wartość typu odpowiadającego typowi wartości zwracanych przez iterator. Na listingu 17.14 typ `Pair<T>` zawiera iterator zwracający wartość typu `T`. Kompilator języka C# sprawdza kod iteratora i tworzy działający tak samo kod potrzebny w metodzie `MoveNext` i właściwości `Current`. Na potrzeby iteratora z typu `Pair<T>` kompilator języka C# generuje kod odpowiadający w przybliżeniu zawartości listingu 17.19.

Listing 17.19. Kod w języku C# będący odpowiednikiem kodu wygenerowanego przez kompilator na podstawie iteratora

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

[NullableContext(1)]
[Nullable(0)]
public class Pair<[Nullable(2)] T> : IPair<T>, IEnumerable<T>, IEnumerable
{
    public Pair(T first, T second)
    {
        First = first;
    }

    public void Dispose()
    {
    }

    public IEnumerator GetEnumerator()
    {
        return new Enumerator();
    }

    public T Current
    {
        get;
    }

    public void MoveNext()
    {
    }
}
```

```
        Second = second;
    }

    public T First { get; }

    public T Second { get; }

    public T this[PairItem index]
    {
        get
        {
            PairItem pairItem = index;
            PairItem pairItem2 = pairItem;
            T result;
            if (pairItem2 != PairItem.First)
            {
                if (pairItem2 != PairItem.Second)
                {
                    throw new NotImplementedException(
                        string.Format(
                            "Wartość {0} nie jest zaimplementowana",
                            index.ToString()));
                }
                result = Second;
            }
            else
            {
                result = First;
            }
            return result;
        }
    }

    public IEnumerator<T> GetEnumerator()
    {
        yield return First;
        yield return Second;
        yield break;
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }
}
```

2.0

Ponieważ kompilator na podstawie instrukcji `yield return` generuje klasy podobne do tych, które sam napisałbyś ręcznie, iteratory w języku C# działają z podobną wydajnością jak klasy z samodzielnie dodaną implementacją wzorca „enumerator”. Jednak choć iteratory nie zapewniają korzyści w zakresie szybkości działania, pozwalają znacznie zwiększyć produktywność programistów.

ZAGADNIENIE DLA ZAAWANSOWANYCH

Kontekstowe słowa kluczowe

Wiele słów kluczowych w języku C# jest zarezerwowanych. Nie można ich stosować jako identyfikatorów, chyba że poprzedzisz je znakiem @. Słowo kluczowe `yield` jest kontekstowe, a nie zarezerwowane. Dozwolone (choć mylące) jest na przykład zadeklarowanie zmiennej lokalnej o nazwie `yield`. Wszystkie słowa kluczowe dodane do języka C# po wersji 1.0 są kontekstowe. Pomaga to zapobiegać przypadkowemu uszkodzeniu kodu w trakcie dostosowywania istniejących programów do nowych wersji języka.

Gdyby projektanci języka C# zdecydowali się zwracać wartość w iteratorach za pomocą składni `yield value`; zamiast zapisu `yield return value`; mogłyby to prowadzić do wieloznaczności. Wtedy wyrażenie `yield(1+2)`; mogłyby zwracać wartość lub przekazywać wartość jako argument do metody o nazwie `yield`.

Ponieważ wcześniej identyfikator `yield` nigdy nie mógł pojawiać się bezpośrednio przed instrukcjami `return` lub `break`, kompilator języka C# potrafi ustalić, że `yield` przed tymi instrukcjami jest słowem kluczowym, a nie identyfikatorem.

2.0

Tworzenie wielu identyfikatorów w jednej klasie

We wcześniejszych przykładach ilustrujących iterator zaimplementowano metodę `Ienumerable<T>.GetEnumerator()`. Jest to metoda bezpośrednio wyszukiwana przez pętlę `foreach`. Czasem potrzebna jest możliwość iterowania w różny sposób, na przykład w odwrotnej kolejności, z filtrowaniem wyników lub z użyciem projekcji innej niż domyślna. W klasie możesz zadeklarować dodatkowe iteratory, ukrywając je we właściwościach lub metodach zwracających obiekt typu `IEnumerable<T>` lub `IEnumerable<T>e`. Jeśli na przykład chcesz iterować po elementach z obiektu typu `Pair<T>` w odwrotnej kolejności, możesz udostępnić metodę `GetReverseEnumerator()`, co pokazano na listingu 17.20.

Listing 17.20. Używanie instrukcji `yield return` w metodzie zwracającej obiekt typu `IEnumerable<T>`

```
public struct Pair<T>: IEnumerable<T>
{
    // ...

    public IEnumerable<T> GetReverseEnumerator()
    {
        yield return Second;
        yield return First;
    }
    // ...
}

public void Main()
{
    var game = new Pair<string>("Redskins", "Eagles");
    foreach (string name in game.GetReverseEnumerator())
    {
        Console.WriteLine(name);
    }
}
```

Zauważ, że kod zwraca obiekt typu `IEnumerable<T>`, a nie typu `IEnumerator<T>`. Metoda `IEnumerable<T>.GetEnumerator()` zwraca obiekt typu `IEnumerator<T>`. Kod w metodzie `Main()` pokazuje, jak wywołać metodę `GetReverseEnumerator()` za pomocą pętli `foreach`.

Wymagania związane z instrukcją `yield`

Instrukcję `yield return` możesz stosować tylko w składowych zwracających obiekt typu `IEnumerator<T>` lub `IEnumerable<T>` (lub ich niegenerycznych odpowiedników). Składowe, w których ciele znajduje się instrukcja `yield return`, nie mogą zawierać zwyknej instrukcji `return`. Jeśli w składowej używana jest instrukcja `yield return`, kompilator języka C# generuje kod potrzebny do zachowania stanu iteratora. Natomiast jeżeli w składowej występuje instrukcja `return` zamiast `yield return`, programista odpowiada za zarządzanie własną maszyną stanową i zwracanie obiektu z implementacją jednego z interfejsów iteratora. Ponadto podobnie jak wszystkie ścieżki w kodzie metody zwracającej wartość muszą zawierać instrukcję `return` z określona wartością (chyba że zgłaszają wyjątek), tak wszystkie ścieżki w iteratorze muszą prowadzić do instrukcji `yield return`, jeśli mają zwracać dane.

Naruszenie opisanych poniżej dodatkowych wymogów dotyczących instrukcji `yield` skutkuje zgłoszeniem błędu kompilatora.

- Instrukcja `yield` może występować tylko w metodzie, w operatorze zdefiniowanym przez użytkownika lub w akcesorze get indeksera lub właściwości. Składowa z tą instrukcją nie może przyjmować żadnych parametrów `ref` i `out`.
- Instrukcja `yield` nie może występować w metodzie anonimowej ani w wyrażeniu `lambda` (zobacz rozdział 13.).
- Instrukcja `yield` nie może występować w klaузach `catch` i `finally` instrukcji `try`. Ponadto może pojawiać się w bloku `try` tylko wtedy, jeśli nie jest on powiązany z blokiem `catch`.

Podsumowanie

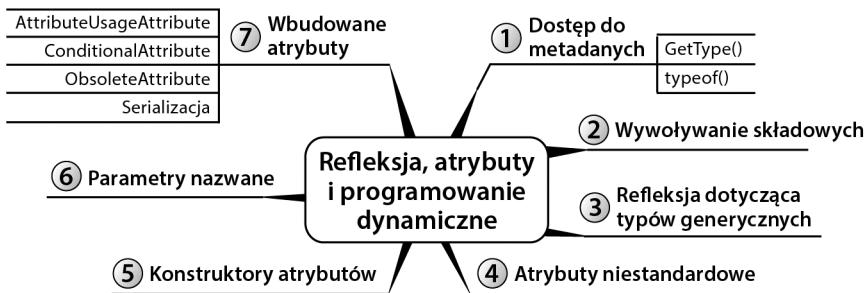
W tym rozdziale omówiono najważniejsze klasy kolekcji i ich podział na podstawie obsługiwanych interfejsów. Każda z tych klas służy do wstawiania elementów do kolekcji i ich pobierania. Odbywa się to na przykład przy użyciu kluczy, indeksów i kolejek FIFO lub LIFO. Opisano tu także iterowanie w kolekcjach. W rozdziale objaśniono też definiowanie własnych kolekcji z niestandardowymi iteratorami do pobierania elementów danej kolekcji. Na potrzeby iteracji używane jest kontekstowe słowo kluczowe `yield`, które w C# powoduje wygenerowanie kodu CIL z implementacją wzorca iteratora opartą na pętli `foreach`.

W rozdziale 18. opisano mechanizm refleksji. To zagadnienie pokróćce omówiono wcześniej, choć z bardzo ubogimi objaśnieniami. Mechanizm refleksji umożliwia sprawdzanie struktury typu w kodzie CIL w czasie wykonywania programu.

18

Refleksja, atrybuty i programowanie dynamiczne

TRYBUTY SŁUŻĄ DO wstawiania dodatkowych metadanych do podzespołu i wiążania metadanych z elementami języka programowania, takimi jak klasa, metoda lub właściwość. W tym rozdziale omówiono szczegóły związane z atrybutami wbudowanymi w platformę i opisano, jak definiować niestandardowe atrybuty. Aby móc wykorzystać niestandardowe atrybuty, trzeba je wykryć. Służy do tego mechanizm refleksji. Ten rozdział rozpoczyna się od omówienia refleksji. Dowiesz się między innymi, jak za pomocą tego mechanizmu na podstawie wywołań składowych przy użyciu nazwy (lub metadanych) na etapie komplikacji dynamicznie wiązać składowe w czasie wykonywania programu. Ta operacja jest często wykonywana w narzędziach takich jak generator kodu. Ponadto refleksja jest wykorzystywana w czasie wykonywania programu, gdy docelowa wywoływana jednostka jest nieznana.



Rozdział kończy się omówieniem programowania dynamicznego. Jest to mechanizm dodany w wersji C# 4.0, który znacznie upraszcza pracę z dynamicznymi danymi, wymagającymi wiązania w czasie wykonywania programu, a nie na etapie komplikacji.

Mechanizm refleksji

Za pomocą mechanizmu refleksji można wykonać następujące zadania:

- Uzyskać dostęp do metadanych dotyczących typów z podzespołu. Te metadane to na przykład pełna nazwa typu, nazwy składowych i atrybuty powiązane z daną jednostką.
- Dynamicznie wywoływać składowe typu w czasie wykonywania programu za pomocą metadanych, a nie na podstawie wiązania zdefiniowanego na etapie komplikacji.

Refleksja to proces analizowania metadanych z podzespołu. W tradycyjnym podejściu, gdy kod jest komplikowany do postaci języka maszynowego, wszystkie metadane (na przykład nazwy typów i metod) są usuwane. Jednak gdy kod w języku C# jest komplikowany do języka CIL, zachowywana jest większość metadanych dotyczących kodu. Ponadto za pomocą mechanizmu refleksji można sprawdzić wszystkie typy z podzespołu i znaleźć te, które spełniają określone kryteria. Dostęp do metadanych typu można uzyskać za pomocą obiektu typu `System.Type`. Ten obiekt udostępnia metody pozwalające pobrać składowe obiektu danego typu. Ponadto można wywoływać wykryte składowe w obiektach analizowanego typu.

Refleksja pozwala na stosowanie niedostępnych bez niej technik. Umożliwia na przykład pobrańe wszystkich typów z podzespołu (wraz z ich składowymi) i utworzenie szablonu dokumentacji interfejsu API podzespołu. Następnie można połączyć metadane pobrańe za pomocą refleksji z dokumentem XML wygenerowanym na podstawie komentarzy XML-owych (przy użyciu opcji `/doc`). W ten sposób można utworzyć dokumentację interfejsu API. Programiści wykorzystują też pobrańe za pomocą refleksji metadane do generowania kodu, który utrwała (w wyniku serializacji) obiekty biznesowe w bazie. Refleksję można też zastosować w kontrolce wyświetlającej listę obiektów z kolekcji. Taka kontrolka może przyjmować kolekcję i przy użyciu mechanizmu refleksji pobierać wszystkie właściwości obiektów z tej kolekcji, a następnie tworzyć na liście kolumny odpowiadające poszczególnym właściwościom. Ponadto dzięki wywołaniu każdej właściwości każdego obiektu w kontrolce można zapełnić wszystkie wiersze i kolumny danymi z obiektów, choć ich typ danych na etapie komplikacji jest nieznany.

`XmlSerializer`, `ValueType` i `DataBinder` z platformy Microsoft .NET Framework to kilka klas z platformy, w których we fragmentach kodu wykorzystano mechanizm refleksji.

Dostęp do metadanych za pomocą obiektu typu `System.Type`

Aby wczytać metadane określonego typu, należy utworzyć obiekt typu `System.Type` reprezentujący docelowy typ. Typ `System.Type` udostępnia wszystkie metody potrzebne do pobrania informacji na temat typu. Za pomocą obiektu typu `System.Type` można uzyskać odpowiedzi na następujące pytania:

- Jaka jest nazwa typu? (Informuje o tym właściwość `Type.Name`).
- Czy typ jest publiczny? (Informuje o tym właściwość `Type.IsPublic`).
- Jaki jest typ bazowy danego typu? (Informuje o tym właściwość `Type BaseType`).
- Czy dany typ obsługuje jakieś interfejsy? (Informuje o tym metoda `Type.GetInterfaces()`).
- W którym podzespołe zdefiniowany jest dany typ? (Informuje o tym właściwość `Type.Assembly`).

- Jakie są właściwości, metody, pola i inne składowe typu? (Informują o tym metody Type.GetProperties(), Type.GetMethods(), Type.GetFields() i inne).
- Jakimi atrybutami jest opatrzony dany typ?
(Informuje o tym metoda Type.GetCustomAttributes()).

Dostępne są też inne składowe, a wszystkie one zapewniają informacje na temat określonego typu. Ważne jest, by uzyskać referencję do obiektu Type reprezentującego określony typ. Dwie podstawowe techniki, które to umożliwiają, to wywołania object.GetType() i typeof().

Zauważ, że wywołanie GetMethods() nie zwraca metod rozszerzających. Są one dostępne tylko jako składowe statyczne typu, w którym są zaimplementowane.

Metoda GetType()

Klasa object udostępnia składową GetType(), dlatego znajduje się ona w każdym typie. Wywołanie GetType() pozwala uzyskać obiekt typu System.Type reprezentujący pierwotny obiekt. Na listingu 18.1 pokazano, jak za pomocą tego procesu utworzyć obiekt typu Type na podstawie typu DateTime. Wynik działania kodu znajdziesz w danych wyjściowych 18.1.

Listing 18.1. Używanie metody Type.GetProperties() do pobrania publicznych właściwości obiektu

```
DateTime dateTime = new DateTime();
Type type = dateTime.GetType();
foreach (
    System.Reflection.PropertyInfo property in
    type.GetProperties())
{
    Console.WriteLine(property.Name);
}
```

DANE WYJŚCIOWE 18.1.

```
Date
Day
DayOfWeek
DayOfYear
Hour
Kind
Millisecond
Minute
Month
Now
UtcNow
Second
Ticks
TimeOfDay
Today
Year
```

Po wywołaniu metody GetType() można pobrać każdy obiekt typu System.Reflection.PropertyInfo zwrocony przez metodę Type.GetProperties() i wyświetlić nazwy właściwości. W wywołaniu GetType() niezbędny jest obiekt docelowego typu. Czasem jednak taki obiekt nie jest dostępny. Nie można na przykład utworzyć obiektu klasy statycznej, dlatego dla takich klas nie da się wywołać metody GetType().

Operator `typeof()`

Inny sposób na pobranie obiektu typu Type polega na użyciu operatora `typeof`. Na etapie komplikacji jest on wiązany z konkretnym obiektem typu Type i jako parametr bezpośrednio przyjmuje nazwę typu. Na listingu 18.2 pokazano, jak zastosować operator `typeof` razem z metodą `Enum.Parse()`.

Listing 18.2. Używanie operatora `typeof()` do tworzenia obiektu typu `System.Type`

```
using System.Diagnostics;
// ...
ThreadPriorityLevel priority;
priority = (ThreadPriorityLevel)Enum.Parse(
    typeof(ThreadPriorityLevel), "Idle");
// ...
```

Na tym listingu metoda `Enum.Parse()` przyjmuje obiekt typu `Type` reprezentujący wyliczenie, a następnie przekształca łańcuch znaków na konkretną wartość tego wyliczenia. Tu łańcuch znaków "Idle" jest przekształcany w wartość `System.Diagnostics.ThreadPriorityLevel.Idle`.

Na listingu 18.3 w następnym punkcie operator `typeof` zastosowano w metodzie `CompareTo(object obj)`, by sprawdzić, czy typ parametru `obj` jest zgodny z oczekiwany:

```
if (obj.GetType() != typeof(Contact)) { ... }
```

Operator `typeof` jest przetwarzany w czasie komplikacji, dlatego za pomocą porównania (na przykład z typem zwróconym w wywoaniu `GetType()`) można ustalić, czy dany obiekt jest określonego typu.

Wywoływanie składowych

Możliwości, jakie daje refleksja, nie ograniczają się do pobierania metadanych. Następny krok polega na pobraniu metadanych i dynamicznym wywołaniu dostępnych w nich składowych. Pomyśl o definicji klasy reprezentującej wiersz poleceń aplikacji¹. Problem z tego rodzaju klasą, na przykład `CommandLineInfo`, związany jest z zapelnianiem klasy danymi podanymi w wierszu poleceń w momencie uruchamiania aplikacji. Za pomocą mechanizmu refleksji można powiązać opcje z wiersza poleceń z nazwami właściwości, a następnie dynamicznie ustawić wartości tych właściwości w czasie wykonywania programu. Ten proces przedstawiono na listingu 18.3.

Listing 18.3. Dynamiczne wywoływanie składowej

```
using System;
using System.Diagnostics;

public partial class Program
{
    public static void Main(string[] args)
    {
        CommandLineInfo commandLine = new CommandLineInfo();
```

¹ W specyfikacji .NET Standard 1.6 dodano pakiet NuGet o nazwie `CommandLineUtils`, w którym dostępny jest m.in. mechanizm przetwarzania instrukcji z wiersza poleceń. Więcej informacji znajdziesz w napisanym przez autora artykuły zamieszczonym w serwisie MSDN: <http://itl.tc/sept2016>.

```
if (!CommandLineHandler.TryParse(
    args, commandLine, out string? errorMessage))
{
    Console.WriteLine(errorMessage);
    DisplayHelp();
}

if (commandLine.Help)
{
    DisplayHelp();
}
else
{
    if (commandLine.Priority != 
        ProcessPriorityClass.Normal)
    {
        // Zmienianie priorytetu wątku.
    }
    // ...
}

private static void DisplayHelp()
{
    // Wyświetlanie pomocy w wierszu poleceń.
    Console.WriteLine(
        "Compress.exe / Out:< nazwa pliku > / Help \n"
        + "/ Priority:RealTime | High | "
        + "AboveNormal | Normal | BelowNormal | Idle");
}

using System;
using System.Diagnostics;

public partial class Program
{
    private class CommandLineInfo
    {
        public bool Help { get; set; }

        public string? Out { get; set; }

        public ProcessPriorityClass Priority { get; set; }
        = ProcessPriorityClass.Normal;
    }
}

using System;
using System.Diagnostics;
using System.IO;
using System.Reflection;

public class CommandLineHandler
{
    public static void Parse(string[] args, object commandLine)
    {
        if (!TryParse(args, commandLine, out string? errorMessage))
        {
            throw new InvalidOperationException(errorMessage);
        }
    }
}
```

```
public static bool TryParse(string[] args, object commandLine,
    out string? errorMessage)
{
    bool success = false;
    errorMessage = null;
    foreach (string arg in args)
    {
        string option;
        if (arg[0] == '/' || arg[0] == '-')
        {
            string[] optionParts = arg.Split(
                new char[] { ':' }, 2);

            // Usuwanie ukośnika lub kreski.
            option = optionParts[0].Remove(0, 1);
            PropertyInfo? property =
                commandLine.GetType().GetProperty(option,
                    BindingFlags.IgnoreCase |
                    BindingFlags.Instance |
                    BindingFlags.Public);
            if (property != null)
            {
                if (property.PropertyType == typeof(bool))
                {
                    // Ostatni parametr służy do obsługi indeksu.
                    property.SetValue(
                        commandLine, true, null);
                    success = true;
                }
                else if (
                    property.PropertyType == typeof(string))
                {
                    property.SetValue(
                        commandLine, optionParts[1], null);
                    success = true;
                }
                else if (
                    // Dostępna jest też wartość property.PropertyType.IsEnum
                    property.PropertyType ==
                        typeof(ProcessPriorityClass))
                {
                    try
                    {
                        property.SetValue(commandLine,
                            Enum.Parse(
                                typeof(ProcessPriorityClass),
                                optionParts[1], true),
                            null);
                        success = true;
                    }
                    catch (ArgumentException )
                    {
                        success = false;
                        errorMessage =
                            $"Opcja '{
                                optionParts[1]
                            }' jest nieprawidłowa dla '{
                                option }'";
                    }
                }
            }
        }
    }
}
```

```

        }
    else
    {
        success = false;
        errorMessage =
            $"Właściwość typu '{
                property.PropertyType
            }' nie jest obsługiwana w typie {
                commandLine.GetType()
            }."}
    }
else
{
    success = false;
    errorMessage =
        $"Opcja '{ option }' nie jest obsługiwana.";
}
}
return success;
}
}

```

Choć listing 18.3 jest długi, przedstawiony na nim kod jest stosunkowo prosty. Metoda Main() najpierw tworzy obiekt klasy CommandLineInfo. Ten typ jest zdefiniowany po to, by przechowywał wprowadzone w wierszu poleceń dane dla omawianego programu. Każda właściwość tego typu odpowiada opcji programu. W danych wyjściowych 18.2 pokazano zawartość wiersza poleceń.

DANE WYJŚCIOWE 18.2.

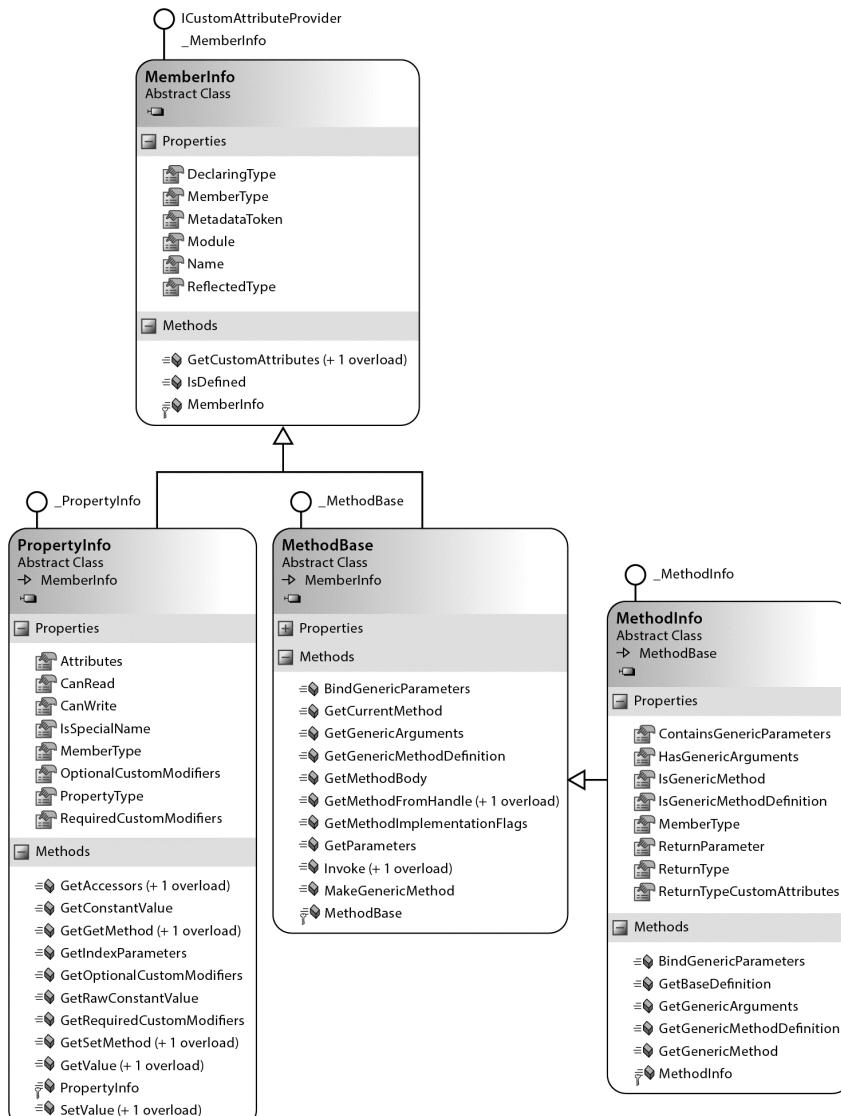
```
Compress.exe /Out:<nazwa pliku> /Help
/Priority:RealTime|High|AboveNormal|Normal|BelowNormal|Idle
```

Obiekt typu CommandLineInfo jest przekazywany do metody TryParse() typu CommandLineHandler. Ta metoda najpierw pobiera wszystkie opcje i określa ich nazwy (na przykład Help lub Out). Po ustaleniu nazwy kod stosuje refleksję do obiektu typu CommandLineInfo i próbuje znaleźć w nim właściwość o danej nazwie. Jeśli znajdzie taką właściwość, przypisuje do niej wartość za pomocą wywołania SetValue(), ustalając w trakcie wykonywania tej operacji, jakie dane są odpowiednie dla typu danej właściwości. Dla argumentów to wywołanie przyjmuje obiekt, w którym należy ustawić daną wartość, nową wartość, a także dodatkowy parametr index, którego wartość to null, chyba że właściwość to indeks. Przedstawiony kod obsługuje właściwości trzech typów: logiczne, tekstowe i wyliczeniowe. W przypadku wyliczeń kod przetwarza wartość opcji i przypisuje do właściwości wartość wyliczeniową będącą odpowiednikiem wprowadzonego tekstu. Jeśli wywołanie TryParse() zakończyło się powodzeniem, metoda kończy pracę, a obiekt typu CommandLineInfo jest zainicjowany danymi z wiersza poleceń.

Co ciekawe, choć CommandLineInfo to klasa prywatna zagnieźdzona w klasie Program, klasa CommandLineHandler bez problemów stosuje refleksję do obiektu typu CommandLineInfo, a nawet wywołuje jego składowe. Oznacza to, że refleksja pozwala pominąć reguły dostępności,

o ile zapewnione są odpowiednie uprawnienia. Gdyby na przykład właściwość `Out` była prywatna, nadal możliwe byłoby przypisanie do niej wartości za pomocą metody `TryParse()`. Z tego powodu można przenieść klasę `CommandLineHandler` do odrębnego podzespołu i korzystać z niej w wielu programach z odrębnymi klasami `CommandLineInfo`.

W przedstawionym przykładzie składowa klasa `CommandLineInfo` jest wywoływana za pomocą metody `PropertyInfo.SetValue()`. Nie jest zaskoczeniem, że typ `PropertyInfo` udostępnia też metodę `GetValue()`, pozwalającą pobrać dane z właściwości. Do wywoływania metod służy klasa `MethodInfo` i jej składowa `Invoke()`. Klasy `MethodInfo` i `PropertyInfo` dziedziczą (pośrednio) po klasie `MemberInfo`, co pokazano na rysunku 18.1.



Rysunek 18.1. Klasa pochodne od klasy `MemberInfo`

Refleksja dotycząca typów generycznych

Wprowadzenie typów generycznych w wersji 2.0 środowiska CLR wymusiło dodanie nowych funkcji w mechanizmie refleksji. Refleksja dotycząca typów generycznych pozwala w czasie wykonywania programu ustalić, czy klasa lub metoda jest generyczna, a także wykryć parametry określające typ.

Ustalanie typów podanych w parametrach określających typ

Podobnie jak za pomocą operatora `typeof` można generować obiekty typu `System.Type` reprezentujące typy niegeneryczne, tak można zastosować ten operator do parametrów określających typ w typie generycznym lub w metodzie generycznej. Na listingu 18.4 operator `typeof` jest używany do określającego typ parametru metody `Add` w klasie `Stack`.

Listing 18.4. Deklarowanie klasy `Stack<T>`

```
public class Stack<T>
{
    // ...
    public void Add(T i)
    {
        // ...
        Type t = typeof(T);
        // ...
    }
    // ...
}
```

Gdy już uzyskasz obiekt typu `Type`, który reprezentuje parametr określający typ, możesz zastosować refleksję do tego parametru. Pozwala to ustalić jego działanie i lepiej dostosować metodę `Add` do konkretnego typu.

Określanie, czy klasa lub metoda ma parametry generyczne

W wersji 2.0 środowiska CLR do klasy `System.Type` dodano kilka metod służących do określania, czy dany typ ma generyczne parametry lub argumenty. Argument generyczny to wartość parametru określającego typ podana w momencie tworzenia obiektu klasy generycznej. Aby określić, czy klasa lub metoda ma parametry generyczne, które nie zostały jeszcze podane, zastosuj właściwość `Type.ContainsGenericParameters` w sposób pokazany na listingu 18.5.

Listing 18.5. Refleksja stosowana do typów generycznych

```
using System;

public class Program
{
    static void Main()
    {
        Type type = typeof(System.Nullable<>);
        Console.WriteLine(type.ContainsGenericParameters);
        Console.WriteLine(type.IsGenericType);
```

```

        type = typeof(System.Nullable<DateTime>);
        Console.WriteLine(type.ContainsGenericParameters);
        Console.WriteLine(type.IsGenericType);
    }
}

```

Wynik działania kodu z listingu 18.5 pokazano w danych wyjściowych 18.3.

DANE WYJŚCIOWE 18.3.

```

True
True
False
True

```

Type.IsGenericType to właściwość zwracająca wartość logiczną, określającą, czy dany typ jest generyczny.

Pobieranie parametrów określających typ z klas lub metod generycznych

Listę argumentów generycznych klasy generycznej można uzyskać za pomocą metody GetGenericArguments(). Zwraca ona tablicę obiektów typu System.Type, które reprezentują kolejne typy odpowiadające deklaracjom parametrów klasy generycznej. Na listingu 18.6 przy użyciu refleksji zastosowanej do typu generycznego pobrano wartości wszystkich argumentów generycznych. Wyniki znajdziesz w danych wyjściowych 18.4.

Listing 18.6. Stosowanie refleksji do typów generycznych

```

using System;
using System.Collections.Generic;

public partial class Program
{
    public static void Main()
    {
        Stack<int> s = new Stack<int>();

        Type t = s.GetType();

        foreach (Type type in t.GetGenericArguments())
        {
            System.Console.WriteLine(
                "Argument określający typ: " + type.FullName);
        }
        // ...
    }
}

```

DANE WYJŚCIOWE 18.4.

```
Argument określający typ: System.Int32
```

Operator nameof

Operator nameof został pokrótko opisany w rozdziale 11., gdzie posłużył do podania nazwy parametru w argumencie wyjątku:

```
throw new ArgumentException(
    "Argument nie reprezentuje cyfry", nameof(textDigit));
```

Wprowadzone w wersji C# 6.0 kontekstowe słowo kluczowe nameof generuje stałą w postaci łańcucha znaków zawierającego krótką nazwę elementu programu podanego jako argument. Tu textDigit to parametr metody, dlatego wywołanie nameof(textDigit) zwraca wartość "textDigit". Ponieważ ta operacja ma miejsce w czasie komplikacji, operator nameof technicznie nie służy do refleksji. Opisano go w tym miejscu, ponieważ ostatecznie pobiera dane na temat podzespołu i jego struktury.

Możesz się zastanawiać, jakie zalety daje stosowanie wyrażenia nameof(textDigit) zamiast samego tekstu "textDigit" (ta druga wersja może niektórym programistom wydawać się prostsza). Operator nameof zapewnia dwojakie korzyści:

- Sprawia, że kompilator języka C# gwarantuje, iż argument operatora nameof jest poprawnym elementem programu. To pomaga unikać literówek, zapobiegając błędowi po zmianie nazwy elementu programu itd.
- Narzędzia środowiska IDE lepiej współpracują z operatorem nameof niż z dosłownie podanymi łańcuchami znaków. Na przykład narzędzie do wyszukiwania wszystkich referencji znajduje elementy programu podawane w operatorze nameof, natomiast nie obsługuje dosłownie podawanych łańcuchów znaków. Omawiany operator zapewnia także lepsze działanie automatycznej refaktoryzacji polegającej na zmianie nazw i innych mechanizmów.

W pokazanym wcześniej fragmencie wyrażenie nameof(textDigit) zwraca nazwę parametru. Jednak operator nameof działa do dowolnych elementów programu. Na przykład na listingu 18.7 operator nameof wykorzystano do przekazania nazwy właściwości do zdarzenia INotifyPropertyChanged.PropertyChanged.

Listing 18.7. Dynamiczne wywoływanie składowych

```
using System.ComponentModel;

public class Person : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;
    public Person(string name)
    {
        Name = name;
    }
    private string _Name = string.Empty;
    public string Name
    {
        get { return _Name; }
        set
        {
            if (_Name != value)
```

```

{
    Name = value;
    // Zastosowanie operatora ?. z wersji C# 6.0.
    PropertyChanged?.Invoke(
        this,
        new PropertyChangedEventArgs(
            nameof(Name)));
}
}
// ...
}

```

Zauważ, że niezależnie od tego, czy używana jest krótka nazwa Name (co jest możliwe, ponieważ właściwość znajduje się w zasięgu), czy pełny (lub częściowo kwalifikowany) zapis, na przykład Person.Name, wynikiem jest tylko końcowy identyfikator (ostatni element w nazwie z operatorami kropki).

W celu ustalenia nazwy właściwości nadal możesz korzystać z wprowadzonego w wersji C# 5.0 atrybutu `CallerMemberName`. Przykładowy kod znajdziesz pod adresem <http://itl.tc/CallerMemberName>.

Koniec
6.0

Atrybuty

Zanim przejdziesz do szczegółowego omówienia pisania atrybutów, powinieneś się zapoznać z sytuacją, która ilustruje ich przydatność. W kodzie klasy `CommandLineHandler` z listingu 18.3 dynamicznie ustawiono właściwości klasy na podstawie dopasowania opcji z wiersza poleceń do nazw właściwości. To podejście nie jest jednak wystarczające, gdy opcja z wiersza poleceń nie może być nazwą właściwości (nie można w ten sposób obsłużyć na przykład opcji `/?`). Ten mechanizm nie zapewnia też żadnego sposobu identyfikowania, które opcje są niezbędne, a które opcjonalne.

Zamiast więc polegać na dokładnym dopasowaniu nazw opcji do nazw właściwości, można za pomocą atrybutów wykrywać dodatkowe metadane dotyczące opatrzonej nimi jednostki (w omawianym przykładzie można powiązać jednostkę z opcją reprezentowaną przez atrybut). Przy użyciu atrybutu możesz opisać właściwość jako wymaganą (`Required`) i określić alias opcji — `/?`. Atrybuty pozwalają więc dodawać do właściwości (i innych jednostek) dodatkowe dane.

Atrybuty podawane są w nawiasie kwadratowym przed wiązaną z nimi jednostką. Możesz na przykład zmodyfikować klasę `CommandLineInfo` i dołączyć do niej atrybuty, co pokazano na listingu 18.8.

Listing 18.8. Dodawanie atrybutów do właściwości

```

class CommandLineInfo
{
    [CommandLineSwitchAlias("?"")]
    public bool Help { get; set; }

    [CommandLineSwitchRequired]
}

```

```

public string? Out { get; set; }

public System.Diagnostics.ProcessPriorityClass Priority
{ get; set; } =
    System.Diagnostics.ProcessPriorityClass.Normal;
}

```

Na listingu 18.8 właściwości `Help` i `Out` są opatrzone atrybutami. Te atrybuty pozwalają stosować alias `/?` dla opcji `/Help`, a także określają, że `/Out` to wymagany parametr. Dzięki temu w metodzie `CommandLineHandler.TryParse()` możliwa jest obsługa aliasów opcji, a jeśli przetwarzanie kończy się powodzeniem, można też sprawdzić, czy wszystkie wymagane opcje zostały podane.

Atrybuty dla tej samej jednostki można łączyć na dwa sposoby. Pierwszy polega na rozdzielaniu atrybutów przecinkami w jednym nawiasie kwadratowym. Inna możliwość to umieszczenie każdego atrybutu w odrębnym nawiasie kwadratowym. Przykładowy kod znajdziesz na listingu 18.9.

Listing 18.9. Dodawanie kilku atrybutów do właściwości

```

[CommandLineSwitchRequired]
[CommandLineSwitchAlias("FileName")]
public string? Out { get; set; }

[CommandLineSwitchRequired,
CommandLineSwitchAlias("FileName")]
public string Out { get; set; }

```

Programiści mogą stosować atrybuty nie tylko do właściwości, ale też do podzespołów, klas, konstruktorów, delegatów, wyliczeń, zdarzeń, pól, parametrów określających typ, interfejsów, metod, modułów, parametrów, zwracanych wartości i struktur. W większości sytuacji dodanie atrybutu wymaga zastosowania tej samej składni z nawiasem kwadratowym, którą zaprezentowano na listingu 18.9. Ta składnia nie działa jednak dla zwracanych wartości, podzespołów i modułów.

Atrybuty podzespołów często służą do dodawania metadanych dotyczących podzespołu. Na przykład w środowisku Visual Studio kreator Project Wizard dla projektów .NET Framework (choć już nie dla projektów wygenerowanych w .NET Core) generuje plik `AssemblyInfo.cs` zawierający liczne atrybuty opisujące podzespół. Listing 18.10 przedstawia przykładowy plik tego rodzaju.

Listing 18.10. Atrybuty podzespółu z pliku `AssemblyInfo.cs`

```

using System.Reflection;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;

// Ogólne informacje na temat podzespółu można podawać
// za pomocą przedstawionego poniżej zestawu atrybutów. Zmień
// wartości tych atrybutów, aby zmodyfikować informacje
// powiązane z podzespolem.
[assembly: AssemblyTitle("CompressionLibrary")]

```

```
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("IntelliTect")]
[assembly: AssemblyProduct("Compression Library")]
[assembly: AssemblyCopyright("Copyright® IntelliTect 2006-2018")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]

// Ustawienie atrybutu ComVisible na false sprawia, że typy z danego
// podzespołu nie są widoczne dla komponentów COM. Jeśli w komponentach COM
// potrzebujesz dostępu do jednego z typów podzespołu, ustaw atrybut
// ComVisible dla tego typu na wartość true.
[assembly: ComVisible(false)]

// Poniższy identyfikator GUID to identyfikator biblioteki typów.
// Jest on używany, jeśli projekt jest dostępny dla komponentów COM.
[assembly: Guid("417a9609-24ae-4323-b1d6-cef0f87a42c3")]

// Informacje o wersji podzespołu
// obejmują cztery następujące wartości:
//
// Główna wersja
// Podwersja
// Numer komplikacji
// Poprawka
//
// Możesz ustawić wszystkie te wartości lub zastosować domyślne
// wartości dla członów z poprawką i numerem komplikacji,
// używając symbolu '*' (tak jak w poniżej instrukcji):
// [assembly: AssemblyVersion("1.0.*")]
[assembly: AssemblyVersion("1.0.0.0")]
[assembly: AssemblyFileVersion("1.0.0.0")]
```

Atrybuty z rodziny assembly pozwalają podać firmę, produkt i numer wersji. Podobne do atrybutów assembly są atrybuty reprezentujące moduł, wymagające przedrostka module:. Atrybuty assembly i module muszą się znajdować po dyrektywie using, ale przed deklaracjami przestrzeni nazw i klas. Atrybuty na listingu 18.10 zostały wygenerowane przez kreator Project Wizard ze środowiska Visual Studio i należy je stosować we wszystkich projektach, by dodać do wynikowych plików binarnych informacje na temat zawartości pliku wykonywalnego lub biblioteki DLL.

Atrybuty z rodziny return, takie jak pokazany na listingu 18.11, pojawiają się przed deklaracją metody. Obowiązuje dla nich opisana wcześniej składnia.

Listing 18.11. Dodawanie atrybutu return

```
[return: Description(
    "Zwraca wartość true, jeśli obiekt znajduje się w poprawnym stanie.")]
public bool IsValid()
{
    // ...
    return true;
}
```

W języku C# docelową jednostkę wiązaną z atrybutem można wskazać za pomocą słów `assembly:` i `return:`, a także określeń `module:`, `class:` i `method:`, informujących, że atrybut dotyczy modułu, klasy lub metody. Słowa `class:` i `method:` są opcjonalne, co pokazano wcześniej.

Jedną z wygodnych cech atrybutów jest to, że język uwzględnia konwencje nazewnicze dotyczące atrybutów. Zgodnie z tymi konwencjami na końcu nazwy musi się znajdować słowo `Attribute`. Jednak we wszystkich miejscach *stosowania* atrybutów na wcześniejszych listingach taki przyrostek się nie pojawia, choć każdy z użytych atrybutów ma nazwę zgodną z konwencjami. Jest tak, ponieważ w języku C# podawanie przyrostka jest opcjonalne, choć można też stosować pełną nazwę (na przykład `DescriptionAttribute` lub `AssemblyVersionAttribute`). Zwykle w miejscu *stosowania* atrybutu przyrostek nie jest podawany. Pojawia się on tylko w definicji atrybutu lub wtedy, gdy atrybut jest stosowany wewnętrznie (na przykład w wyrażeniu `typeof(DescriptionAttribute)`).

Zauważ, że w projektach opartych na platformie .NET Core można — zamiast generowania pliku `AssemblyInfo.cs` — podać specyfikację podzespołu w pliku `*.CSPROJ`. Kod z listingu 18.12 wstrzymuje odpowiednie atrybuty do podzespołu na etapie komplikacji.

Listing 18.12. Definiowanie niestandardowego atrybutu

```
<Project>
  <PropertyGroup>
    <Company>Addison Wesley</Company>
    <Copyright>Copyright © Addison Wesley 2020</Copyright>
    <Product>C# 8.0. Kompletny przewodnik dla praktyków</Product>
    <Version>8.0</Version>
  </PropertyGroup>
</Project>
```

Ten kod jest przekształcany na kod CIL pokazany w danych wyjściowych 18.5.

DANE WYJŚCIOWE 18.5.

```
[assembly: AssemblyCompany("Addison Wesley")]
[assembly: AssemblyCopyright("Copyright © Addison Wesley 2020")]
[assembly: AssemblyFileVersion("8.0.0.0")]
[assembly: AssemblyInformationalVersion("8.0")]
[assembly: AssemblyProduct("C# 8.0. Kompletny przewodnik dla praktyków")]
[assembly: AssemblyVersion("8.0.0.0")]
```

Wskazówki

STOSUJ atrybut `AssemblyVersionAttribute` do podzespołów zawierających typy publiczne.

ROZWAŻ zastosowanie atrybutów `AssemblyFileVersionAttribute` i `AssemblyCopyrightAttribute`, by udostępnić dodatkowe informacje na temat podzespołu.

STOSUJ następujące atrybuty z informacjami o podzespołe:

`System.Reflection.AssemblyCompanyAttribute`,
`System.Reflection.AssemblyCopyrightAttribute`,
`System.Reflection.AssemblyDescriptionAttribute`
 i `System.Reflection.AssemblyProductAttribute`.

Niestandardowe atrybuty

Definiowanie niestandardowych atrybutów jest proste. Atrybuty są obiektami, dlatego aby zdefiniować atrybut, należy utworzyć klasę. Cechą, która przekształca zwykłą klasę w atrybut, jest dziedziczenie po typie `System.Attribute`. Możesz więc utworzyć na przykład klasę `CommandLineSwitchRequiredAttribute` przedstawioną na listingu 18.13.

Listing 18.13. Definiowanie niestandardowego atrybutu

```
public class CommandLineSwitchRequiredAttribute : Attribute
{
}
```

Po utworzeniu tej prostej definicji możesz zacząć stosować atrybut w sposób pokazany na listingu 18.8. Na razie żaden kod nie reaguje na ten atrybut, dlatego dodanie go do właściwości `Out` nie wpływa na przetwarzanie instrukcji z wiersza poleceń.

Wskazówka

STOSUJ przyrostek `Attribute` w nazwach klas niestandardowych atrybutów.

Wyszukiwanie atrybutów

Typ `Type` (obok właściwości służących do analizowania składowych danego typu) udostępnia metody pozwalające pobrać dodane do typu atrybuty. Wszystkie typy związane z refleksją (na przykład `PropertyInfo` i `MethodInfo`) zawierają składowe, które umożliwiają pobranie listy atrybutów powiązanych z danym typem. Na listingu 18.14 zdefiniowano metodę, która zwraca listę wymaganych opcji pominiętych w instrukcji z wiersza poleceń.

Listing 18.14. Pobieranie niestandardowych atrybutów

```
using System;
using System.Collections.Specialized;
using System.Reflection;

public class CommandLineSwitchRequiredAttribute : Attribute
{
    public static string[] GetMissingRequiredOptions(
        object commandLine)
    {
        List<string> missingOptions = new List<string>();
        PropertyInfo[] properties =
            commandLine.GetType().GetProperties();

        foreach (PropertyInfo property in properties)
        {
            Attribute[] attributes =
                (Attribute[])property.GetCustomAttributes(
                    typeof(CommandLineSwitchRequiredAttribute),
                    false);
        }
    }
}
```

```

        if ((attributes.Length > 0) &&
            (property.GetValue(commandLine, null) == null))
    {
        missingOptions.Add(property.Name);
    }
}
return missingOptions.ToArray();
}
}

```

Kod do wykrywania atrybutów jest stosunkowo prosty. Gdy dostępny jest obiekt typu `PropertyInfo` (otrzymany za pomocą mechanizmu refleksji), należy wywołać metodę `GetCustomAttributes()` i podać szukany atrybut, a następnie określić, czy mają być sprawdzane różne wersje przeciążonej metody. Inna możliwość to wywołanie metody `GetCustomAttributes()` bez podawania typów atrybutów. Wtedy zwracane są wszystkie atrybuty.

Choć kod do wyszukiwania atrybutu typu `CommandLineSwitchRequiredAttribute` można umieścić bezpośrednio w klasie `CommandLineHandler`, lepszą hermetyzację zapewnia umieszczenie potrzebnego kodu w samej klasie `CommandLineSwitchRequiredAttribute`. W atrybutach niestandardowych często stosuje się tę technikę. Jakie miejsce lepiej nadaje się na kod służący do wyszukiwania atrybutu niż statyczna metoda klasy tego atrybutu?

Iinicjowanie atrybutu za pomocą konstruktora

Wywołanie metody `GetCustomAttributes()` zwraca tablicę obiektów, którą można zrzutować na tablicę elementów typu `Attribute`. W przykładowym kodzie atrybut nie ma żadnych składowych instancji, dlatego jedyne zwracane metadane dotyczą tego, czy atrybut w ogóle się pojawił. W atrybutach można też jednak umieszczać dane. Na listingu 18.15 zdefiniowano atrybut `CommandLineSwitchAliasAttribute`. Jest to niestandardowy atrybut umożliwiający obsługę aliasów opcji z wiersza poleceń. Możesz na przykład zapewnić obsługę opcji `/Help` oraz jej skrótnego zapisu `/?`. Podobnie `/S` może być aliasem opcji `/Subfolders`, oznaczającej, że instrukcja powinna przetworzyć wszystkie podkatalogi.

Listing 18.15. Udostępnianie konstruktora atrybutu

```

public class CommandLineSwitchAliasAttribute : Attribute
{
    public CommandLineSwitchAliasAttribute(string alias)
    {
        Alias = alias;
    }

    public string Alias { get; }

}

class CommandLineInfo
{
    [CommandLineSwitchAlias("??")]
    public bool Help { get; set; }
    // ...
}

```

Aby dodać obsługę omawianej funkcji, należy udostępnić konstruktor atrybutu. Na potrzeby obsługi aliasów utwórz konstruktor przyjmujący argument w postaci łańcucha znaków. Jeśli chcesz umożliwić podanie kilku aliasów, zdefiniuj atrybut z parametrem w postaci tablicy params string.

Gdy dodajesz atrybut do wybranej jednostki, jako argumenty można podawać wyłącznie stałe i wyrażenia typeof(). To ograniczenie ma umożliwiać przekształcenie kodu na wynikowy kod CIL. Z tego wynika, że konstruktor atrybutu musi mieć parametry właściwego typu. Nie ma na przykład sensu tworzenie konstruktora przyjmującego argumenty typu System.DateTime, skoro w języku C# nie istnieją stałe tego typu.

Obiekty zwarcane przez metodę PropertyInfo.GetCustomAttributes() są inicjowane podanymi argumentami konstruktora, co pokazano na listingu 18.16.

Listing 18.16. Pobieranie konkretnego atrybutu i sprawdzanie wartości użytej do zainicjowania go

```
 PropertyInfo property =
    typeof(CommandLineInfo).GetProperty("Help")!;
CommandLineSwitchAliasAttribute attribute =
    (CommandLineSwitchAliasAttribute)
        property.GetCustomAttributes(
            typeof(CommandLineSwitchAliasAttribute), false).ElementAt(0);
if (attribute?.Alias == "?")
{
    Console.WriteLine("Help(?)");
};
```

Ponadto (co pokazano na listingach 18.17 i 18.18) można zastosować podobny kod w metodzie GetSwitches() klasy CommandLineSwitchAliasAttribute. Ta metoda może zwracać słownik zawierający wszystkie opcje (także te oparte na nazwach właściwości) i łączyć wszystkie nazwy z odpowiadającymi im wartościami z obiektu commandLine.

Listing 18.17. Pobieranie instancji niestandardowych atrybutów

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

public class CommandLineSwitchAliasAttribute : Attribute
{
    public CommandLineSwitchAliasAttribute(string alias)
    {
        Alias = alias;
    }

    public string Alias { get; set; }

    public static Dictionary<string, PropertyInfo> GetSwitches(
        object commandLine)
    {
        PropertyInfo[] properties;
        Dictionary<string, PropertyInfo> options =
            new Dictionary<string, PropertyInfo>();
```

```

properties = commandLine.GetType().GetProperties(
    BindingFlags.Public | BindingFlags.Instance);
foreach ( PropertyInfo property in properties)
{
    options.Add(property.Name, property);
    foreach (CommandLineSwitchAliasAttribute attribute in
        property.GetCustomAttributes(
            typeof(CommandLineSwitchAliasAttribute), false))
    {
        options.Add(attribute.Alias.ToLower(), property);
    }
}
return options;
}
}

```

Listing 18.18. Zmodyfikowana wersja metody CommandLineHandler.TryParse() obsługująca aliasy

```

using System;
using System.Reflection;
using System.Collections.Generic;

public class CommandLineHandler
{
    // ...

    public static bool TryParse(
        string[] args, object commandLine,
        out string? errorMessage)
    {
        bool success = false;
        errorMessage = null;

        Dictionary<string, PropertyInfo> options =
            CommandLineSwitchAliasAttribute.GetSwitches(
                commandLine);

        foreach (string arg in args)
        {
            string option;
            if (arg[0] == '/' || arg[0] == '-')
            {
                string[] optionParts = arg.Split(
                    new char[] { ':' }, 2);
                option = optionParts[0].Remove(0, 1).ToLower();

                if (options.TryGetValue(option, out PropertyInfo? property))
                {
                    success = SetOption(
                        commandLine, property,
                        optionParts, ref errorMessage);
                }
            }
            else
            {
                success = false;
                errorMessage =
                    $"Opcja '{ option }' nie jest obsługiwana.";
            }
        }
    }
}

```

```

        }
    }
    return success;
}

private static bool SetOption(
    object commandline, PropertyInfo property,
    string[] optionParts, ref string? errorMessage)
{
    bool success;

    if (property.PropertyType == typeof(bool))
    {
        // Ostatni parametr służy do obsługi indeksu.
        property.SetValue(
            commandLine, true, null);
        success = true;
    }
    else
    {
        if (optionParts.Length < 2
            || optionParts[1] == "")
        {
            // Nie podano wartości danej opcji.
            success = false;
            errorMessage =
                $"Należy podać wartość opcji { property.Name }.";;
        }
        else if (
            property.PropertyType == typeof(string))
        {
            property.SetValue(
                commandLine, optionParts[1], null);
            success = true;
        }
        else if (
            // Dostępna jest też wartość property.PropertyType.IsEnum.
            property.PropertyType ==
                typeof(ProcessPriorityClass))
        {
            success = TryParseEnumSwitch(
                commandLine, optionParts,
                property, ref errorMessage);
        }
        else
        {
            success = false;
            errorMessage =
                $"Właściwość typu '{ property.PropertyType.ToString() }' "
                + "nie jest dostępna w typie {"
                + commandLine.GetType().ToString() + "}";
        }
    }
    return success;
}

```

Wskazówki

UDOSTĘPNIJ właściwości tylko do odczytu (z prywatnym setterem) w atrybutach, które muszą mieć ustawioną wartość danych właściwości.

UDOSTĘPNIJ konstruktor z parametrami, by inicjować właściwości w atrybutach, które ich wymagają. Każdy parametr powinien mieć taką samą nazwę (choć z inną wielkością liter) jak odpowiadająca mu właściwość.

UNIKAJ dodawania w konstruktorze parametrów przeznaczonych do inicjowania właściwości atrybutu odpowiadających argumentom opcjonalnym (oznacza to, że należy unikać przeciążania konstruktorów w atrybutach niestandardowych).

Atrybut System.AttributeUsageAttribute

Większość atrybutów jest przeznaczona do dodawania do tylko określonych jednostek. Na przykład nie ma sensu dodawać atrybutu `CommandLineSwitchAliasAttribute` do klasy lub podzespołu. Dla tych jednostek ten atrybut nie ma znaczenia. Aby uniknąć niewłaściwego zastosowania atrybutu, do niestandardowych atrybutów można dodać atrybut `System.AttributeUsageAttribute` (tak, atrybut jest tu dodawany do deklaracji niestandardowego atrybutu). Na listingu 18.19 pokazano, jak to zrobić dla klasy `CommandLineSwitchAliasAttribute`.

Listing 18.19. Określanie, do jakich jednostek można dodawać dany atrybut

```
[AttributeUsage(AttributeTargets.Property)]
public class CommandLineSwitchAliasAttribute : Attribute
{
    // ...
}
```

Jeśli atrybut jest używany w niewłaściwy sposób, tak jak na listingu 18.20, zgłaszany jest błąd czasu komplikacji (pokazany w danych wyjściowych 18.6).

Listing 18.20. Atrybut AttributeUsageAttribute ogranicza miejsca, gdzie można zastosować dany atrybut

```
// BŁĄD: ten atrybut można stosować tylko do właściwości.
[CommandLineSwitchAlias("?")]
class CommandLineInfo
{
}
```

DANE WYJŚCIOWE 18.6.

```
...Program+CommandLineInfo.cs(24,17): error CS0592: Attribute
'CommandLineSwitchAlias' is not valid on this declaration type. It is
valid on 'property, indexer' declarations only.
```

Konstruktor klasy `AttributeUsageAttribute` przyjmuje opcję `AttributeTargets`. Jest to opcja typu wyliczeniowego, określająca listę jednostek, do których można dodać dany atrybut w środowisku uruchomieniowym. Jeśli chcesz umożliwić dodawanie atrybutu `CommandLineSwitchAliasAttribute` do pól, zmodyfikuj atrybut `AttributeUsageAttribute` w sposób pokazany na listingu 18.21.

Listing 18.21. Używanie atrybutu `AttributeUsageAttribute` do ograniczenia listy miejsc, gdzie można stosować dany atrybut

```
// Dany atrybut można stosować tylko do właściwości i metod.  
[AttributeUsage(  
    AttributeTargets.Field | AttributeTargets.Property)]  
public class CommandLineSwitchAliasAttribute : Attribute  
{  
    // ...  
}
```

Wskazówka

STOSUJ atrybut `AttributeUsageAttribute` do niestandardowych atrybutów.

Parametry nazwane

Za pomocą atrybutu `AttributeUsageAttribute` można nie tylko określić jednostki, dla których dozwolone jest stosowanie danego atrybutu. Można też umożliwić wielokrotne dodanie określonego atrybutu do tej samej jednostki. Potrzebną składnię przedstawiono na listingu 18.22.

Listing 18.22. Używanie parametru nazwanego

```
[AttributeUsage(AttributeTargets.Property, AllowMultiple=true)]  
public class CommandLineSwitchAliasAttribute : Attribute  
{  
    // ...  
}
```

Ta składnia różni się od omówionej wcześniej składni inicjowania wartości w konstruktorze. `AllowMultiple` to **parametr nazwany**. Podobna składnia używana jest do podawania opcjonalnych parametrów metod (została ona dodana w wersji C# 4.0). Parametry nazwane umożliwiają ustawianie wybranych publicznych właściwości i pól w wywołaniu konstruktora atrybutu, nawet jeśli konstruktor nie zawiera powiązanych parametrów. Parametry nazwane są opcjonalne i umożliwiają ustawienie dodatkowych danych instancji w atrybutach bez konieczności tworzenia kolejnych parametrów konstruktora. Klasa `AttributeUsageAttribute` z przykładowego kodu obejmuje składową publiczną o nazwie `AllowMultiple`. Dlatego gdy stosujesz ten atrybut, możesz ustawić wartość tej zmiennej za pomocą parametru nazwanego. Przypisywać wartości do parametrów nazwanych trzeba w końcowej części konstruktora, po jawnie zadeklarowanych parametrach konstruktora.

Parametry nazwane umożliwiają przypisywanie danych do atrybutu bez tworzenia oddzielnych konstruktorów dla każdej możliwej kombinacji podawanych właściwości. Ponieważ atrybut może zawierać liczne właściwości opcjonalne, opisana technika często się przydaje.

ZAGADNIENIE DLA POCZĄTKUJĄCYCH

Atrybut FlagsAttribute

W rozdziale 9. przedstawiono wyliczenia oraz zamieszczono zagadnienie dla zaawansowanych z omówieniem atrybutu FlagsAttribute. Ten zdefiniowany w platformie atrybut jest przeznaczony dla wycieźli reprezentujących wartości flag. W niniejszym zagadnieniu też opisano ten atrybut. Zacznię od zapoznania się z przykładowym kodem z listingu 18.23.

Listing 18.23. Używanie atrybutu FlagsAttribute

```
// Typ FileAttributes zdefiniowany w przestrzeni nazw System.IO.

[Flags] // Atrybut FlagsAttribute dodany do wyliczenia.
public enum FileAttributes
{
    ReadOnly = 1<<0,      // 0000000000000001
    Hidden =   1<<1,      // 0000000000000010
    // ...
}
using System;
using System.Diagnostics;
using System.IO;

class Program
{
    public static void Main()
    {
        // ...

        string fileName = @"enumtest.txt";
        FileInfo file = new FileInfo(fileName);

        file.Attributes = FileAttributes.Hidden |
            FileAttributes.ReadOnly;

        Console.WriteLine("\\"{0}" wyświetlone jako \"{1}\",
            file.Attributes.ToString().Replace(", ", " |"),
            file.Attributes);

        FileAttributes attributes =
            (FileAttributes)Enum.Parse(typeof(FileAttributes),
            file.Attributes.ToString());

        Console.WriteLine(attributes);
        // ...
    }
}
```

Wynik działania kodu z listingu 18.23 pokazano w danych wyjściowych 18.7.

DANE WYJŚCIOWE 18.7.

"ReadOnly | Hidden" wyświetlane jako "ReadOnly, Hidden"

Dodany atrybut informuje, że wartości wyliczenia można ze sobą łączyć, a ponadto zmienia działanie metod `ToString()` i `Parse()`. Na przykład wywołanie metody `ToString()` obiektu typu wyliczeniowego opatrzonego atrybutem `FlagsAttribute` sprawia, że wyświetlane są łańcuchy znaków reprezentujące wszystkie ustawione flagi. Na listingu 18.23 wywołanie `file.Attributes.ToString()` zwraca wartość "ReadOnly, Hidden", a nie liczbę 3 (zwracaną, gdy atrybut `FlagsAttribute` nie jest używany). Jeśli dana wartość wyliczeniowa pojawia się dwukrotnie, metoda `ToString()` zwraca pierwsze jej wystąpienie. Wcześniej wyjaśniono jednak, że nie należy bezkrytycznie polegać na wynikach zwracanych przez tę metodę, ponieważ nie obsługuje ona tłumaczeń.

Możliwe jest też przekształcanie łańcuchów znaków na wartość wyliczeniową. Poszczególne łańcuchy znaków muszą być wtedy rozdzielone przecinkami.

Zauważ, że atrybut `FlagsAttribute` nie powoduje automatycznego przypisania unikatowych wartości do flag ani sprawdzania, czy flagi mają unikatowe wartości. Wartość każdego elementu wyliczenia trzeba ustawić jawnie.

Predefiniowane atrybuty

Atrybut `AttributeUsageAttribute` ma specjalną cechę, z którą nie zetknąłeś się jeszcze w kontekście niestandardowych atrybutów rozwijanych w tej książce. Ten atrybut wpływa na działanie kompilatora i powoduje, że kompilator czasem zgłasza błąd. Atrybut `AttributeUsageAttribute` ma wbudowaną obsługę kompilatora i nie wymaga pisania kodu dla środowiska uruchomieniowego (atrzybyt `CommandLineRequiredAttribute` i `CommandLineSwitchAliasAttribute` muszą być pobierane za pomocą mechanizmu refleksji).

`AttributeUsageAttribute` to atrybut predefiniowany. Nie tylko zapewnia dodatkowe metadane na temat jednostki, do której został dodany, ale też powoduje zmiany w działaniu środowiska uruchomieniowego i kompilatora. Przykładowe atrybuty predefiniowane to: `AttributeUsageAttribute`, `FlagsAttribute`, `ObsoleteAttribute` i `ConditionalAttribute`. Tylko takie atrybuty mogą wpływać w specjalny sposób na działanie generatora lub kompilatora kodu CLI. Nie ma możliwości tworzenia dodatkowych atrybutów predefiniowanych, a niestandardowe atrybuty są w pełni pasywne. Na listingu 18.23 zastosowano kilka predefiniowanych atrybutów, a w rozdziale 19. poznasz inne atrybuty tego rodzaju.

Atrybut `System.ConditionalAttribute`

W ramach jednego podzespołu atrybut `System.Diagnostics.ConditionalAttribute` działa podobnie jak dyrektywy preprocessora `#if/#endif`. Jednak zamiast eliminować kod CIL z podzespołu, atrybut `System.Diagnostics.ConditionalAttribute` opcjonalnie powoduje, że wywołanie działa jak instrukcja `no-op`, czyli jak polecenie, które nic nie robi. Na listingu 18.24 przedstawiono tę technikę, a w danych wyjściowych 18.8 — wyniki.

Listing 18.24. Używanie atrybutów ConditionalAttribute do eliminowania wywołań

```
#define CONDITION_A

using System;
using System.Diagnostics;

public class Program
{
    public static void Main()
    {
        Console.WriteLine("Początek...");
        MethodA();
        MethodB();
        Console.WriteLine("Koniec...");
    }

    [Conditional("CONDITION_A")]
    static void MethodA()
    {
        Console.WriteLine("Wykonywanie metody MethodA()...");
    }

    [Conditional("CONDITION_B")]
    static void MethodB()
    {
        Console.WriteLine("Wykonywanie metody MethodB()...");
    }
}
```

DANE WYJŚCIOWE 18.8.

```
Początek...
Wykonywanie metody MethodA()...
Koniec...
```

W tym przykładzie zdefiniowano identyfikator `CONDITION_A`, dlatego metoda `MethodA()` jest wykonywana w standardowy sposób. Identyfikator `CONDITION_B` nie został jednak zdefiniowany (ani za pomocą dyrektywy `#define`, ani przy użyciu opcji `csc.exe /Define`). Dlatego wszystkie wywołania metody `Program.MethodB()` w danym podzespole nie prowadzą do wykonania żadnych operacji.

Jeśli chodzi o efekt, atrybut `ConditionalAttribute` działa podobnie jak dodanie dyrektywy `#if/#endif` wokół wywołania metody. Składnia jest jednak bardziej przejrzysta, ponieważ programiści mogą uzyskać pożądany skutek w wyniku dodania atrybutu `ConditionalAttribute` do docelowej metody bez wprowadzania jakichkolwiek zmian w jednostce wywołującej.

Kompilator języka C# wykrywa omawiany atrybut wywoływanej metody w trakcie komplikacji i jeśli potrzebna dyrektywa preprocesora z identyfikatorem nie została zdefiniowana, eliminuje wszystkie wywołania metody. Atrybut `ConditionalAttribute` nie wpływa jednak na skompilowany kod CIL docelowej metody (dodaje wyłącznie metadane atrybutu). Zamiast tego usuwa wywołania metody. Jest to następna różnica między atrybutem `ConditionalAttribute` a dyrektywami `#if/#endif`, gdy kod jest wywoływany w różnych podzespołach. Ponieważ metoda opatrzona opisywanym atrybutem jest kompilowana i dołączana do docelowego podzespołu,

to, czy należy ją wywoływać, zależy nie od zdefiniowania potrzebnego identyfikatora w podzespołe *wywoływanej metody*, ale od tego, czy identyfikator zdefiniowano w podzespołe *jednostki wywołującej*. Oznacza to, że jeśli utworzysz drugi podzespół z definicją warunku `CONDITION_B`, wywołania metody `Program.MethodB()` w tym podzespołe zostaną wykonane. Jest to przydatne w wielu scenariuszach związanych ze śledzeniem i testami. Opisywana cecha jest na przykład wykorzystywana do wywołań metod typów `System.Diagnostics.Trace` (zależnych od dyrektywy `TRACE`) i `System.Diagnostics.Debug` (zależnych od dyrektywy `DEBUG`).

Ponieważ jeśli odpowiedni identyfikator preprocesora nie jest zdefiniowany, metody nie są wykonywane, atrybutu `ConditionalAttribute` nie można stosować do metod z parametrem `out` lub z typem zwracanej wartości innym niż `void`. W przeciwnym razie wystąpi błąd czasu komilacji. Jest to uzasadnione, ponieważ kod metody opatrzonej tym atrybutem może nie zostać wykonany, dlatego nie wiadomo, jaką wartość należy zwrócić do jednostki wywołującej. Atrybutu `ConditionalAttribute` nie można dodawać także do właściwości. Atrybut `AttributeUsage` (zobacz podrozdział „Atrybut `System.AttributeUsageAttribute`” we wcześniejszej części rozdziału) dla atrybutu `ConditionalAttribute`² ma wartości `AttributeTargets.Class` i `AttributeTargets.Method`. Oznacza to, że atrybut `ConditionalAttribute` można stosować do metod lub klas. W przypadku klas obowiązuje jednak pewne ograniczenie — omawiany atrybut można stosować tylko do klas pochodnych od `System.Attribute`.

Gdy atrybut `ConditionalAttribute` jest dodany do niestandardowego atrybutu, ten niestandardowy atrybut można pobrać za pomocą mechanizmu refleksji tylko wtedy, gdy w wywołującym instrukcje podzespołe zdefiniowany jest warunkowy łańcuch znaków. Jeśli taki łańcuch znaków jest niedostępny, mechanizm refleksji nie znajdzie danego niestandardowego atrybutu.

Atrybut `System.ObsoleteAttribute`

Jak wcześniej wspomniano, predefiniowane atrybuty wpływają na działanie kompilatora i środowiska uruchomieniowego. `ObsoleteAttribute` to następny atrybut z tej grupy. Ma on pomagać w zarządzaniu wersjami kodu i umożliwia poinformowanie jednostki wywołującej o tym, że dane składowe lub typy są nieaktualne. Na listingu 18.25 pokazano, jak zastosować atrybut `ObsoleteAttribute`. W danych wyjściowych 18.9 zobaczysz, że jednostki wywołujące składowe opatrzone atrybutem `ObsoleteAttribute` powodują zgłoszenie ostrzeżenia (lub opcjonalnie błędu) w czasie komplikacji.

Listing 18.25. Stosowanie atrybutu `ObsoleteAttribute`

```
class Program
{
    public static void Main()
    {
        ObsoleteMethod();
    }

    [Obsolete]
    public static void ObsoleteMethod()
    {
    }
}
```

² Tę możliwość wprowadzono w platformie Microsoft .NET Framework 2.0.

DANE WYJŚCIOWE 18.9.

```
c:\SampleCode\ObsoleteAttributeTest.cs(24,17): warning CS0612:  
Program.ObsoleteMethod()' is obsolete
```

Tu atrybut `ObsoleteAttribute` prowadzi do wyświetlenia ostrzeżenia. Atrybut ma też jednak dwa inne konstruktory. Jeden z nich, `ObsoleteAttribute(string message)`, powoduje dodanie argumentu z komunikatem dołączanym do wyświetlonej przez kompilator informacji o nieaktualnym kodzie. Najlepiej umieścić w tym komunikacie wskazówki opisujące, jak zastąpić przestarzały kod. Drugi konstruktor ma parametr `bool error`, który pozwala przekształcić ostrzeżenie w błąd.

Atrybut `ObsoleteAttribute` umożliwia niezależnym jednostkom powiadamianie programistów o przestarzałych interfejsach API. Ostrzeżenie (ale już nie błąd) pozwala korzystać z pierwotnego interfejsu API do czasu zaktualizowania kodu przez autora jednostki wywołującej.

Początek
4.0

Programowanie z wykorzystaniem obiektów dynamicznych

Wprowadzenie obiektów dynamicznych w wersji C# 4.0 uprościło programowanie w wielu scenariuszach i umożliwiło stosowanie niedostępnych wcześniej rozwiązań. Istotą programowania z wykorzystaniem obiektów dynamicznych jest to, że programiści mogą pisać operacje oparte na dynamicznym mechanizmie wiązania wywołań. Środowisko uruchomieniowe wiąże wtedy wywołania w trakcie wykonywania programu. Jest to różnica w porównaniu z sytuacją, gdy kompilator sprawdza i wiąże wywołania na etapie komplikacji.

Dlaczego ta technika jest przydatna? Ponieważ obiekty często nie mają statycznie określonego typu. Pomyśl na przykład o wczytywaniu danych z plików XML i CSV, z tabel bazy danych, z modelu DOM przeglądarki Internet Explorer lub z interfejsu `IDispatch` komponentów COM, a także o wywoywaniu kodu w językach dynamicznych, takich jak IronPython. Wprowadzona w wersji C# 4.0 obsługa obiektów dynamicznych zapewnia standardowe rozwiązanie do komunikowania się ze środowiskami uruchomieniowymi, które nie zawsze mają strukturę definiowaną w czasie komplikacji. W początkowej implementacji obiektów dynamicznych (z wersji C# 4.0) dostępne były cztery techniki wiązania.

1. Użycie refleksji do stosowanego typu ze środowiska CLR.
2. Wywołanie niestandardowego obiektu typu `IDynamicMetaObjectProvider`, udostępniającego obiekt typu `DynamicMetaObject`.
3. Wywołania komponentów COM za pomocą interfejsów `IUnknown` i `IDispatch`.
4. Wywołania kierowane do typów zdefiniowanych w językach dynamicznych, takich jak IronPython.

Tu znajdziesz omówienie dwóch pierwszych z tych czterech podejść. Obowiązujące w nich zasady dotyczą też pozostałych technik związanych ze współdziałaniem z komponentami COM i z językami dynamicznymi.

Korzystanie z refleksji za pomocą instrukcji dynamic

Jedną z najważniejszych cech refleksji jest możliwość dynamicznego wyszukiwania i wywoływanie składowych określonego typu na podstawie wykrywania w czasie wykonywania programu nazw (lub innych aspektów, na przykład atrybutów) składowych. Pokazano to na listingu 18.3. Dodanie dynamicznych obiektów w wersji C# 4.0 pozwala wywoływać składowe za pomocą refleksji w prostszy sposób, pod warunkiem jednak, że sygnatura składowej jest znana na etapie komplikacji. To ograniczenie oznacza, że na etapie komplikacji trzeba znać nazwę składowej i sygnaturę (liczbę parametrów oraz to, czy ich typy będą zgodne z daną sygnaturą). Przykładowy kod pokazano na listingu 18.26, a wynik jego działania znajdziesz w danych wyjściowych 18.10.

Listing 18.26. Dynamiczne programowanie z wykorzystaniem „refleksji”

```
using System;
// ...
dynamic data =
    "Witaj! Nazywam się Inigo Montoya";
Console.WriteLine(data);
data = (double)data.Length;
data = data * 3.5 + 28.6;
if (data == 2.4 + 112 + 26.2)3
{
    Console.WriteLine(
        $"{ data } oznacza długi triathlon.");
}
else
{
    data.NonExistentMethodCallStillCompiles();
}
// ...
```

4.0

DANE WYJŚCIOWE 18.10.

```
Witaj! Nazywam się Inigo Montoya
140.6 oznacza długi triathlon.
```

W tym przykładzie nie pojawia się kod, który jawnie określa typ obiektu, wyszukuje konkretną instancję typu `MethodInfo`, a następnie kieruje do niej wywołanie. Zamiast tego dla zmiennej `data` zadeklarowany jest typ `dynamic`, po czym wywołania metod są kierowane bezpośrednio do niej. Na etapie komplikacji kompilator nie sprawdza, czy podane składowe są dostępne. Nie sprawdza nawet typu obiektu `dynamic`. Dlatego na tym etapie dozwolone jest zgłaszanie dowolnych wywołań, pod warunkiem że ich składnia jest prawidłowa. W czasie komplikacji nie ma znaczenia, czy podana składowa w ogóle istnieje.

³ Są to odległości (w milach) pokonywane w wodzie, na rowerze i biegiem w triathlonowych zawodach Ironman.

Nie oznacza to jednak całkowitej rezygnacji z bezpieczeństwa ze względu na typ. Jeśli programista posługuje się standardowymi typami środowiska CLR (takimi jak typ użyty na listingu 18.26), mechanizm kontroli stosowany dla typów niedynamicznych na etapie komplikacji jest uruchamiany dla typów dynamicznych w trakcie wykonywania programu. Dlatego jeśli w czasie wykonywania programu okaże się, że podana składowa jest niedostępna, wywołanie spowoduje wyjątek `Microsoft.CSharp.RuntimeBinder.RuntimeBinderException`.

Zauważ, że to rozwiązanie ma wprawdzie prostszy interfejs API, ale nie daje tyle swobody co opisany we wcześniejszej części rozdziału mechanizm refleksji. Najważniejsza różnica między refleksją a posługiwaniem się dynamicznymi obiektami polega na tym, że w tym drugim przypadku trzeba ustalić sygnaturę na etapie komplikacji, zamiast określić aspekty takie jak nazwa składowej w czasie wykonywania programu (jak miało to miejsce, gdy kod przetwarzał argumenty podawane w wierszu poleceń).

Zasady i operacje związane z typem `dynamic`

Z listingu 18.26 i dotyczącego tekstu można wywnioskować kilka rzeczy na temat typu `dynamic`.

4.0

- *Słowo `dynamic` jest dyrektywą kompilatora żądającą wygenerowania kodu.*

Typ `dynamic` jest powiązany z mechanizmem przechwytywania wywołań, dlatego gdy środowisko uruchomieniowe natrafi na dynamiczne wywołanie, może skompilować żądanie, przekształcając je na kod CIL, a następnie uruchomić nowo skompilowane wywołanie. Bardziej szczegółowy opis tego procesu znajdziesz w zagadnienniu dla zaawansowanych „Omówienie typu `dynamic`”.

Gdy obiekt jest przypisywany do zmiennej typu `dynamic`, pierwotny typ obiektu zostaje „ukryty”, dlatego na etapie komplikacji nie dochodzi do sprawdzenia typu. Ponadto gdy w czasie wykonywania programu składowa jest wywoływana, nakładka przechwytuje wywołanie i kieruje je do odpowiedniej składowej (lub odrzuca je). Wywołanie metody `GetType()` obiektu typu `dynamic` pozwala ustalić ukryty typ dynamicznego obiektu; to wywołanie nie zwraca wartości `dynamic` jako nazwy typu.

- *Dowolny typ⁴, który można przekształcić na typ `object`, można też przekształcić na typ `dynamic`.*

Na listingu 18.26 na typ `dynamic` z powodzeniem zrzutowano obiekty typu bezpośredniego (`double`) i obiekt typu referencyjnego (`string`). Na typ `dynamic` można przekształcać dowolne typy. Konwersja z dowolnego typu referencyjnego na typ `dynamic` odbywa się niejawnie. Ponadto obsługiwana jest niejawną konwersją (z opakowywaniem) z typów bezpośrednich na typ `dynamic`. Możliwa jest też niejawną konwersję z typu `dynamic` na ten sam typ. Jest to może oczywiste, ale w przypadku typu `dynamic` sytuacja jest bardziej skomplikowana, ponieważ nie wystarczy skopiować wskaźnika (adresu) z jednej lokalizacji do innej.

⁴ W ujęciu technicznym chodzi tu o dowolny typ, który można przekształcić na obiekt, co wyklucza niezabezpieczone wskaźniki, lambdy i grupy metod.

■ *Możliwość konwersji z typu dynamic na inny typ zależy od danego typu.*

Konwersja obiektu typu `dynamic` na standardowe typy środowiska CLR musi się odbywać jawnie (na przykład `(double)data.Length`). Nie jest zaskoczeniem, że jeśli docelowy typ jest bezpośredni, konieczna jest konwersja z wypakowywaniem. Jeżeli typ używanego obiektu obsługuje konwersję na typ docelowy, konwersja z typu `dynamic` też zakończy się powodzeniem.

■ *Rzeczywisty typ obiektu używanego jako obiekt typu dynamic może zmieniać się w kolejnych przypisaniach.*

Inaczej niż w przypadku zmiennych o niejawnie określonym typie (zmiennych `var`), gdzie po przypisaniu pierwotnej wartości nie można zmienić typu, typ `dynamic` udostępnia mechanizm przechwytywania i komplikacji stosowany przed wykonaniem kodu używanego obiektu. Dlatego możliwe jest zastąpienie wykorzystywanego obiektu obiektem zupełnie innego typu. Skutkuje to powstaniem następnego punktu przechwytywania wywołań, gdzie trzeba skompilować kod przed wykonaniem wywołania.

■ *Sprawdzanie, czy podana sygnatura jest dostępna w rzeczywistym typie obiektu, zachodzi dopiero na etapie wykonywania programu.*

Na podstawie wywołania metody `person.NonExistentMethodCallStillCompiles()` widać, że kompilator prawie nie sprawdza operacji obiektów typu `dynamic`. Ten krok jest pozostawiany środowisku uruchomieniowemu, które sprawdza operacje w trakcie wykonywania kodu. Ponadto jeśli dane wywołanie nigdy nie jest wykonywane (dotyczy to na przykład wywołania metody `person.NonExistentMethodCallStill ↴Compiles()`), to nawet jeżeli uruchamiany jest otaczający je kod, sprawdzanie i wiązanie danej składowej jest całkowicie pomijane.

■ *Wynik wywołania dowolnej składowej obiektu typu dynamic na etapie kompilacji też jest tego typu.*

Wywołanie dowolnej składowej obiektu typu `dynamic` powoduje zwrocenie obiektu tego samego typu. Dlatego wywołanie takie jak `data.ToString()` skutkuje zwroceniem obiektu typu `dynamic`, a nie rzeczywiście używanego typu `string`. Jednak w trakcie wykonywania programu, gdy dla obiektu typu `dynamic` wywoływana jest metoda `GetType()`, zwracany jest obiekt reprezentujący typ z czasu wykonania.

■ *Jeśli w czasie wykonywania programu podana składowa nie istnieje, środowisko uruchomieniowe zgłosi wyjątek Microsoft.CSharp.RuntimeBinder.RuntimeBinderException.*

Jeżeli w trakcie wykonywania programu nastąpi próba wywołania składowej, środowisko uruchomieniowe sprawdzi, czy dane wywołanie jest prawidłowe (gdy używana jest refleksja, środowisko zbada na przykład to, czy typy są zgodne z sygnaturami). Jeśli sygnatury metod nie są właściwe, środowisko uruchomieniowe zgłosi wyjątek `Microsoft.CSharp.RuntimeBinder.RuntimeBinderException`.

- Gdy typ dynamic jest stosowany razem z refleksją, metody rozszerzające nie są obsługiwane.

Podobnie jak refleksja oparta na typie System.Type, tak refleksja wykorzystująca typ dynamic nie obsługuje metod rozszerzających. Możliwe jest wywoływanie metod rozszerzających za pomocą typu z ich implementacją (na przykład typu System.Linq.Enumerable), natomiast nie można ich wywoływać bezpośrednio przy użyciu rozszerzanego typu.

- Typ dynamic jest w istocie typem System.Object.

Ponieważ każdy obiekt można z powodzeniem przekształcić na typ dynamic, a obiekty typu dynamic można jawnie przekształcać na inne typy, dynamic działa podobnie jak typ System.Object. Typ dynamic ma nawet tę samą wartość domyślną, null (zwracaną za pomocą instrukcji default(dynamic)), co typ System.Object. Oznacza to, że jest typem referencyjnym. Specjalne dynamiczne działanie typu dynamic, które odróżnia go od typu System.Object, jest widoczne tylko w czasie komilacji.

4.0

ZAGADNIENIE DLA ZAAWANSOWANYCH

Omówienie typu dynamic

Dezasembler kodu w języku CIL pozwala zobaczyć, że w kodzie CIL typ dynamic to typ System.Object. Gdyby w kodzie nie było żadnych wywołań, użycie typu dynamic niczym by się nie różniło od zastosowania typu System.Object. Różnica staje się jednak widoczna, gdy wywoływane są składowe.

By wywołać składową, kompilator deklaruje zmienną typu System.Runtime.CompilerServices.CallSite<T>. Wartość parametru T zależy od sygnatury składowej, jednak czasem tak prosta operacja jak wywołanie metody ToString() wymaga utworzenia obiektu typu CallSite<Func<CallSite, object, string>> oraz wywołania metody z parametrami CallSite site, object dynamicTarget i string result. Parametr site reprezentuje miejsce wywołania, parametr dynamicTarget to obiekt, którego metoda jest wywoływana, a result to wartość zwracana przez wywołanie metody ToString(). Zamiast bezpośrednio tworzyć obiekt typu CallSite<Func<CallSite _site, object dynamicTarget, string result>>, można zastosować metodę fabryczną Create(), przyjmującą parametr typu Microsoft.CSharp.RuntimeBinder.CSharpConvertBinder. Gdy już dostępny jest obiekt typu CallSite<T>, ostatni krok polega na wywołaniu metody CallSite<T>.Target(), co prowadzi do uruchomienia odpowiedniej składowej.

W czasie wykonania platforma „na zapleczu” wykorzystuje mechanizm refleksji do znalezienia składowych i sprawdzenia, czy mają one właściwe sygnatury. Następnie środowisko uruchomieniowe tworzy drzewo wyrażeń, które reprezentuje dynamiczne wyrażenie zdefiniowane w miejscu wywołania. Po skompilowaniu drzewa wyrażeń powstaje ciało metody w kodzie CIL, podobne do tego, jakie wygenerowałby kompilator, gdyby wywołanie nie było dynamiczne. Kod CIL jest następnie zachowywany w miejscu wywołania, a wywołanie jest wykonywane za pomocą delegata. Ponieważ kod CIL jest teraz zachowany w miejscu wywołania, w następnym wywołaniu nie trzeba ponownie ponosić kosztów refleksji i komilacji.

Po co stosować wiązanie dynamiczne?

Typy dynamiczne można stosować razem z mechanizmem refleksji, ale można też tworzyć niestandardowe typy i wywoływać je dynamicznie. Możesz na przykład rozważyć użycie wywołania dynamicznego do pobrania wartości elementu XML-owego. Zamiast posługiwać się składnią ze ścisłą kontrolą typów (taką jak na listingu 18.27), można dynamicznie uruchomić wywołania `person.FirstName` i `person.LastName`.

Listing 18.27. Wiązanie elementów XML-owych w czasie wykonywania programu bez używania typu dynamic

```
4.0
using System;
using System.Xml.Linq;

// ...
 XElement person = XElement.Parse(
    @"<Person>
        <FirstName>Inigo</FirstName>
        <LastName>Montoya</LastName>
    </Person>");
Console.WriteLine("{0} {1}",
    person.Descendants("FirstName").FirstOrDefault().Value,
    person.Descendants("LastName").FirstOrDefault().Value);
// ...
```

Choć kod z listingu 18.27 nie jest przesadnie skomplikowany, porównaj go teraz z zawartością listingu 18.28, gdzie pokazano inne rozwiązanie, oparte na obiekcie z dynamicznie określonym typem.

Listing 18.28. Wiązanie elementów XML-owych w czasie wykonywania programu z użyciem typu dynamic

```
using System;
// ...
// Typ DynamicXml jest pokazany na listingu 18.32.
dynamic person = DynamicXml.Parse(
    @"<Person>
        <FirstName>Inigo</FirstName>
        <LastName>Montoya</LastName>
    </Person>");
Console.WriteLine(
    $"{ person.FirstName } { person.LastName }");
// ...
```

Zalety są oczywiste, czy jednak oznacza to, że programowanie dynamiczne jest lepsze od statycznej kompilacji?

Statyczna komplikacja a programowanie dynamiczne

Kod z listingu 18.28 działa prawie tak samo jak rozwiązanie z listingu 18.27, jednak z jedną bardzo ważną różnicą — na listingu 18.27 kontrola typów jest statyczna. Dlatego w czasie komplikacji sprawdzane są wszystkie typy i sygnatury ich składowych. Nazwy metod muszą być właściwe. Sprawdzana jest także zgodność typów wszystkich parametrów. Jest to bardzo ważna cecha języka C# i coś, na co zwracamy uwagę w tej książce.

Natomiast w kodzie z listingu 18.28 nie ma statycznej kontroli typów. Zmienna person jest typu `dynamic`. Dlatego w czasie komplikacji nie jest sprawdzane, czy obiekt person ma właściwość `FirstName` lub `LastName` (lub czy udostępnia jakiekolwiek inne składowe). Ponadto gdy piszesz kod w środowisku IDE, mechanizm IntelliSense nie wyświetla listy składowych obiektu person.

Wydaje się zatem, że brak kontroli typów prowadzi do znacznego zubożenia możliwości języka. Po co więc w ogóle udostępniono w języku C# taką technikę (a nawet wprowadzono ją jako nową funkcję w wersji C# 4.0)?

Aby zrozumieć ten pozorny paradoks, warto ponownie zastanowić się nad kodem z listingu 18.28. Zwróć uwagę na wywołanie pobierające element "FirstName":

```
Element.Descendants("LastName").FirstOrDefault().Value
```

4.0

Na listingu do podania nazwy elementu posłużył łańcuch znaków ("LastName"), jednak w czasie komplikacji nie następuje sprawdzenie, czy ten łańcuch znaków jest poprawny. Gdyby wielkość liter była niezgodna z nazwą elementu lub gdyby w nazwie dodano spację, komplikacja też zakończyłaby się powodzeniem, natomiast w momencie wywołania właściwości `Value` wystąpiłby wyjątek `NullReferenceException`. Kompilator nie próbuje też sprawdzić, czy element "FirstName" w ogóle istnieje. Jeśli nie jest dostępny, wystąpi wyjątek `NullReferenceException`. Oznacza to, że choć język zapewnia korzyści wynikające z bezpieczeństwa ze względu na typ, nie są one dostępne, gdy kod korzysta na przykład z dynamicznych danych zapisanych w elemencie XML-owym.

Kod z listingu 18.28 nie jest lepszy od jego odpowiednika z listingu 18.27, jeśli chodzi o kontrolę procesu pobierania elementu na etapie komplikacji. Jeśli wielkość liter będzie błędna lub element `FirstName` będzie niedostępny, wystąpi wyjątek⁵. Porównaj jednak wywołanie dające dostęp do imienia na listingu 18.28 (`person.FirstName`) z wywołaniem z listingu 18.27. Rozwiązanie z listingu 18.28 jest wyraźnie prostsze.

Zdarzają się sytuacje, w których mechanizm zapewniania bezpieczeństwa ze względu na typ nie przeprowadza (i prawdopodobnie nie może przeprowadzić) niektórych testów. Wtedy kod zawierający wywołanie dynamiczne, sprawdzane tylko w czasie wykonywania programu, jest dużo bardziej czytelny i zwięzły. Oczywiście, jeśli sprawdzanie kodu na etapie komplikacji jest możliwe, lepiej stosować programy ze statyczną kontrolą typu (z uwagi na czytelne i zwięzłe interfejsy API). Jednak w sytuacjach, gdy nie jest to efektywne rozwiązanie, C# 4.0 umożliwia programistom pisanie prostszego kodu (co pozwala zrezygnować z nacisku na ścisłe bezpieczeństwo ze względu na typ).

⁵ W wywołaniu właściwości `FirstName` nie można używać spacji, jednak także XML nie obsługuje spacji w nazwach elementów, dlatego w tym kontekście kwestię spacji można pominąć.

Tworzenie niestandardowych obiektów dynamicznych

Na listingu 18.28 znajduje się wywołanie metody `DynamicXml.Parse(...)`. Jest to wywołanie metody fabrycznej typu `DynamicXml` (jest to typ niestandardowy, a nie typ wbudowany w platformę CLR). Jednak typ `DynamicXml` nie zawiera implementacji właściwości `FirstName` i `LastName`. Dodanie takich właściwości spowoduje, że nie da się dynamicznie pobrać danych z pliku XML w czasie wykonywania programu (trzeba wtedy dodawać elementy XML-owe na etapie komplikacji). Oznacza to, że w typie `DynamicXml` w celu uzyskania dostępu do składowych nie jest stosowana refleksja. Zamiast tego kod dynamicznie wiąże wartości na podstawie zawartości pliku XML.

Aby zdefiniować niestandardowy typ dynamiczny, należy zaimplementować w nim interfejs `System.Dynamic.IDynamicMetaObjectProvider`. Jednak zamiast implementować ten interfejs od podstaw, lepiej utworzyć typ pochodny od `System.Dynamic.DynamicObject`. W ten sposób uzyskasz domyślne implementacje zestawu składowych i będziesz mógł przesłonić te ich wersje, które nie pasują do nowej klasy. Na listingu 18.29 przedstawiono kompletną implementację takiego typu.

4.0 Listing 18.29. Implementacja niestandardowego typu dynamicznego

```
using System;
using System.Dynamic;
using System.Xml.Linq;

public class DynamicXml : DynamicObject
{
    private XElement Element { get; set; }

    public DynamicXml(System.Xml.Linq.XElement element)
    {
        Element = element;
    }

    public static DynamicXml Parse(string text)
    {
        return new DynamicXml(XElement.Parse(text));
    }

    public override bool TryGetMember(
        GetMemberBinder binder, out object? result)
    {
        bool success = false;
        result = null;
        XElement firstDescendant =
            Element.Descendants(binder.Name).FirstOrDefault();
        if (firstDescendant != null)
        {
            if (firstDescendant.Descendants().Any())
            {
                result = new DynamicXml(firstDescendant);
            }
            else
            {
                result = firstDescendant.Value;
            }
        }
        return success;
    }

    public void TrySetMember(
        SetMemberBinder binder, object? value)
    {
        if (Element != null)
        {
            XElement element =
                Element.Descendants(binder.Name).FirstOrDefault();
            if (element != null)
            {
                element.Value = value.ToString();
            }
        }
    }
}
```

```
        }
        success = true;
    }
    return success;
}

public override bool TrySetMember(
    SetMemberBinder binder, object value)
{
    bool success = false;
    XElement firstDescendant =
        Element.Descendants(binder.Name).FirstOrDefault();
    if (firstDescendant != null)
    {
        if (value.GetType() == typeof(XElement))
        {
            firstDescendant.ReplaceWith(value);
        }
        else
        {
            firstDescendant.Value = value.ToString();
        }
        success = true;
    }
    return success;
}
```

4.0

Najważniejsze metody dynamiczne w tym przykładzie to `TryGetMember()` i `TrySetMember()` (przy założeniu, że pobieranie i ustawianie wartości elementów ma być możliwe). Wystarczy zaimplementować te dwie metody, by umożliwić wywoływanie dynamicznych getterów i setterów właściwości. Implementacje w tym przykładzie są proste. Najpierw kod sprawdza przechowywany obiekt typu `XElement`, szukając w nim elementu na podstawie wartości `binder.Name`, określającej nazwę wywoływanej składowej. Jeśli odpowiedni element XML-owy istnieje, wartość jest pobierana (lub ustawiana). Zwracana wartość to `true`, jeśli element istnieje, i `false`, jeżeli jest on niedostępny. Zwrócenie wartości `false` powoduje natychmiastowe zgłoszenie przez środowisko uruchomieniowe wyjątku `Microsoft.CSharp.RuntimeBinder.RuntimeBinderException` w miejscu wywołania dynamicznie uruchamianej składowej.

Typ `System.Dynamic.DynamicObject` obsługuje też dodatkowe metody wirtualne, jeśli wymagane są inne wywołania dynamiczne. Na listingu 18.30 znajdziesz listę wszystkich składowych z tego typu, które można przesłonić.

Listing 18.30. Składowe z typu `System.Dynamic.DynamicObject`, które można przesłonić

```
using System.Dynamic;

public class DynamicObject : IDynamicMetaObjectProvider
{
    protected DynamicObject();

    public virtual IEnumerable<string> GetDynamicMemberNames();
    public virtual DynamicMetaObject GetMetaObject(
```

```

    Expression parameter);
public virtual bool TryBinaryOperation(
    BinaryOperationBinder binder, object arg,
    out object result);
public virtual bool TryConvert(
    ConvertBinder binder, out object result);
public virtual bool TryCreateInstance(
    CreateInstanceBinder binder, object[] args,
    out object result);
public virtual bool TryDeleteIndex(
    DeleteIndexBinder binder, object[] indexes);
public virtual bool TryDeleteMember(
    DeleteMemberBinder binder);
public virtual bool TryGetIndex(
    GetIndexBinder binder, object[] indexes,
    out object result);
public virtual bool TryGetMember(
    GetMemberBinder binder, out object result);
public virtual bool TryInvoke(
    InvokeBinder binder, object[] args, out object result);
public virtual bool TryInvokeMember(
    InvokeMemberBinder binder, object[] args,
    out object result);
public virtual bool TrySetIndex(
    SetIndexBinder binder, object[] indexes, object value);
public virtual bool TrySetMember(
    SetMemberBinder binder, object value);
public virtual bool TryUnaryOperation(
    UnaryOperationBinder binder, out object result);
}

```

Na listingu 18.30 pokazano, że dostępne są składowe powiązane z wszelkimi możliwymi zadaniami — od rzutowania i różnych operacji po korzystanie z indeksu. Ponadto dostępna jest metoda `GetDynamicMemberNames()` pobierająca wszystkie dostępne nazwy składowych.

Koniec
4.0

Podsumowanie

W tym rozdziale opisano, jak posługiwać się mechanizmem refleksji do wczytywania metadanych, które są kompilowane do postaci kodu CIL. Za pomocą refleksji można zapewnić późne wiązanie wywołań, polegające na tym, że wywoływany kod jest definiowany w czasie wykonywania programu, a nie na etapie komplikacji. Choć refleksja jest w pełni akceptowalna do tworzenia systemów dynamicznych, oparty na niej kod działa znacznie wolniej niż kod wiązany statycznie (w czasie komplikacji). Dlatego refleksja jest częściej stosowana w narzędziach programistycznych, gdzie wydajność nie jest najważniejsza.

Refleksja umożliwia też pobieranie dodatkowych metadanych, dołączanych do różnych jednostek za pomocą atrybutów. Niestandardowe atrybuty są zwykle wyszukiwane z wykorzystaniem refleksji. Możesz zdefiniować własne niestandardowe atrybuty, aby dołączać wybrane dodatkowe metadane do kodu CIL. Następnie, w czasie wykonywania programu, można pobrać te metadane i wykorzystać je w kodzie.

Wielu programistów traktuje atrybuty jak wstęp do programowania aspektowego, w którym funkcje są dodawane za pomocą konstrukcji takich jak atrybuty (nie trzeba wtedy ręcznie implementować potrzebnych mechanizmów). Musi minąć trochę czasu, zanim w języku C# pojawią się prawdziwe elementy programowania aspektowego (nie wiadomo nawet, czy kiedykolwiek to nastąpi). Jednak atrybuty stanowią dobry krok w tym kierunku i nie zagrażają przy tym stabilności języka.

W tym rozdziale opisano też mechanizm wprowadzony w wersji C# 4.0 — programowanie dynamiczne oparte na nowym typie, `dynamic`. W tym fragmencie wyjaśniono, dlaczego wiązanie statyczne (choć zalecane, gdy w interfejsie API stosowana jest ścisła kontrola typów) ma pewne ograniczenia, gdy używane są dane dynamiczne.

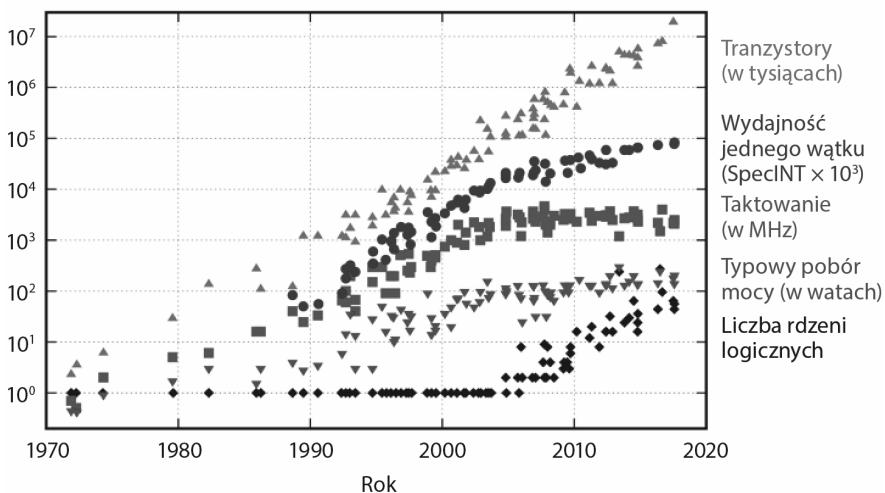
W następnym rozdziale zapoznasz się z wielowątkowością. W tej dziedzinie atrybuty służą do synchronizowania pracy kodu.

19

Wprowadzenie do wielowątkowości

DWA WAŻNE TRENDY MIAŁY w ostatniej dekadzie bardzo istotny wpływ na dziedzinę budowania oprogramowania. Po pierwsze, stały spadek kosztów wykonywania obliczeń nie wynika już ze wzrostu szybkości taktowania i z rosnącej gęstości upakowania tranzystorów (zobacz rysunek 19.1). Kost obliczeń wciąż spada, ponieważ opłaca się produkować sprzęt wieloprocesorowy.

Trendy w branży mikroprocesorów w ostatnich 42 latach



Rysunek 19.1. Zmiany szybkości taktowania w czasie

(Dane za okres do 2010 roku zebrane i przedstawione przez M. Horowitza, F. Labonte, K. Olukotuna, L. Hammonda i C. Battena. Nowy wykres i dane za okres 2010 – 2017 przygotowane przez K. Ruppa).

Po drugie, w obliczeniach często występuje obecnie bardzo duża **latencja**. W uproszczeniu można powiedzieć, że jest to ilość czasu potrzebna do otrzymania pożądanego wyniku. Istnieją dwie podstawowe przyczyny latencji. **Latencja związana z procesorem** ma miejsce, gdy zadanie obliczeniowe jest złożone. Jeśli obliczenia wymagają wykonania 12 miliardów operacji arytmetycznych, a łączna moc obliczeniowa to 6 miliardów operacji na sekundę,

od momentu zażądania wyniku do czasu otrzymania go upłyну przynajmniej dwie sekundy wynikające z latencji związanej z procesorem. Natomiast **latencja związana z operacjami wejścia-wyjścia** dotyczy opóźnienia wynikającego z konieczności pobrania danych z zewnętrznego źródła (na przykład dysku, serwera WWW itd.). Wszelkie obliczenia wymagające pobrania danych z serwera WWW zlokalizowanego fizycznie z dala od maszyny klienta powodują latencję odpowiadającą milionom cykli procesora.

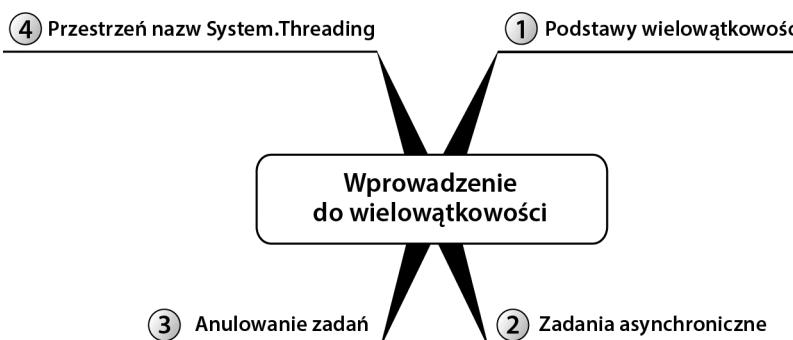
Dwa opisane trendy stanowią poważne wyzwanie dla twórców oprogramowania. Maszyny zapewniają dziś większą moc obliczeniową niż kiedykolwiek wcześniej, jak jednak efektywnie wykorzystać ją, by szybko dostarczać wyniki przy zachowaniu wysokiego komfortu pracy użytkowników? Jak uniknąć tworzenia budzących frustrację interfejsów użytkownika, które blokują się, gdy wykonywane są operacje skutkujące dużą latencją? Ponadto jak rozdzielić zadania między wiele procesorów, by zmniejszyć czas wykonywania obliczeń?

Standardową techniką stosowaną w inżynierii oprogramowania, by zapewnić możliwość reagowania interfejsu użytkownika i wysokie wykorzystanie mocy procesora, jest pisanie programów **wielowątkowych**, które równolegle wykonują wiele obliczeń. Niestety, pisanie poprawnego kodu wielowątkowego jest trudne. W następnych dwóch rozdziałach zobacysz, co jest tego przyczyną. Nauczysz się też stosować wysokopoziomowe abstrakcje i nowe mechanizmy języka, które ułatwiają pisanie programów wielowątkowych.

Pierwszą wysokopoziomową abstrakcją była biblioteka Parallel Extensions, wprowadzona w platformie .NET 4.0. Obejmuje ona komponenty **TPL** (ang. *Task Parallel Library*; omówiony w tym rozdziale) i **PLINQ** (ang. *Parallel LINQ*; opisany w rozdziale 21.). Drugą abstrakcją wysokiego poziomu jest wzorzec **TAP** (ang. *Task-based Asynchronous Pattern*) i mechanizmy jego obsługi wprowadzone w wersjach C# 5.0 i nowszych.

Choć gorąco zachęcamy do posługiwania się tymi wysokopoziomowymi abstrakcjami, w tym rozdziale omawiamy też niskopoziomowe interfejsy API związane z wątkami, stosowane w starszych wersjach środowiska uruchomieniowego platformy .NET. Dodatkowe wzorce z obszaru programowania wielowątkowego (z wersji starszych niż C# 5.0) znajdziesz na stronie <http://IntelliTect.com/EssentialCSharp>. Dostępne są tam także rozdziały z książki *Essential C# 3.0*. Dlatego jeśli chcesz w pełni opanować programowanie wielowątkowe bez nowszych rozwiązań, możesz skorzystać z tych materiałów.

Rozdział rozpoczyna się od omówienia kilku zagadnień dla początkujących (przydatnych dla osób, które dopiero poznają wielowątkowość).



Podstawy wielowątkowości

ZAGADNIENIE DLA POCZĄTKUJĄCYCH

Słownictwo związane z wielowątkowością

Z wielowątkością związanych jest wiele mało zrozumiałych pojęć. Warto więc zdefiniować wybrane określenia.

Procesor (ang. *CPU — central processing unit*) lub **rdzeń¹** to część sprzętu wykonująca dany program. Każdy komputer ma przynajmniej jeden procesor, choć obecnie powszechnie używane są maszyny wieloprocesorowe. Liczne nowoczesne procesory obsługują **wielowątkowość równoległą** (ang. *simultaneous multithreading*; firma Intel stosuje dla tej techniki zastrzeżoną nazwę **HyperThreading**). Jest to tryb, w którym jeden procesor może symulować pracę wielu wirtualnych procesorów.

Proces to obecnie wykonywana instancja danego programu. Głównym zadaniem systemu operacyjnego jest zarządzanie procesami. Każdy proces obejmuje jeden lub więcej wątków. Proces jest reprezentowany jako obiekt klasy `Process` z przestrzeni nazw `System.Diagnostics`.

Na poziomie instrukcji programowania w języku C# polega przede wszystkim na opisywaniu **przepływu sterowania**. Do tego miejsca książki przyjmowano ukryte założenie, że w danym programie znajduje się tylko jeden punkt sterowania. Możesz sobie wyobrazić, że punkt sterowania jest jak kurSOR. W momencie uruchomienia programu kurSOR rozpoczyna generowanie tekstu programu od metody `Main`, a następnie porusza się po programie w czasie wykonywania różnych warunków, pętli, wywołań metod itd. Takim punktem sterowania jest **wątek**. Jest on reprezentowany za pomocą obiektu klasy `System.Threading.Thread`. Interfejs API służący do zarządzania wątkami znajduje się w przestrzeni nazw `System.Threading`.

W programie **jednowątkowym** w procesie działa tylko jeden wątek. W programie **wielowątkowym** w procesach są dwa wątki lub większa ich liczba.

Fragment kodu jest **bezpieczny ze względu na wątki**, jeśli działa prawidłowo, gdy jest używany w programie wielowątkowym. **Model wątkowy** obowiązujący w danym fragmencie kodu to zestaw wymagań, jakie dany kod stawia jednostce wywołującej w zamian za gwarancje bezpieczeństwa ze względu na wątki. Na przykład model wątkowy w wielu klasach wygląda tak: „metody statyczne mogą być wywoływane w dowolnym wątku, natomiast metody instancji mogą być uruchamiane tylko w wątku, w którym utworzono daną instancję”.

Zadanie to jednostka pracy (często powodująca dużą latencję), która generuje wynikową wartość lub prowadzi do pożądanego efektu ubocznego. Różnica między zadaniami i wątkami polega na tym, że zadanie reprezentuje pracę, jaką należy wykonać, natomiast wątek to jednostka robocza, która tę pracę wykonuje. Zadanie jest przydatne tylko ze względu na efekty, do jakich prowadzi. Do reprezentowania zadań służy klasa `Task`. Zadanie używane do generowania wartości danego typu jest reprezentowane za pomocą klasy `Task<T>`, pochodnej od niegenerycznego typu `Task`. Te klasy znajdziesz w przestrzeni nazw `System.Threading.Tasks`.

¹ W ujęciu technicznym należałoby stwierdzić, że „procesor” zawsze oznacza fizyczny układ, a „rdzeń” może oznaczać procesor fizyczny lub wirtualny. W tej książce to rozróżnienie nie jest istotne, dlatego oba pojęcia są tu stosowane zamiennie.

Pula wątków to zestaw wątków wraz z mechanizmami określającymi, jak przydzielać zadania do tych wątków. Gdy program ma zadanie do wykonania, może pobrać wątek roboczy z puli, przydzielić mu wykonanie zadania, a następnie zwolnić wątek po zakończeniu pracy. Wątek staje się wtedy dostępny, gdy trzeba wykonać dodatkowe zadania.

ZAGADNIENIE DLA POCZĄTKUJĄCYCH

Co, jak i dlaczego w dziedzinie wielowątkowości

Wielowątkowość jest związana z dwoma podstawowymi scenariuszami — umożliwianiem wielozadaniowości i radzeniem sobie z latencją.

Użytkownicy często uruchamiają dziesiątki procesów jednocześnie. Możliwe, że otwierzyli do edycji prezentacje i arkusze kalkulacyjne, a jednocześnie przeglądają dokumenty w internecie, słuchają muzyki, otrzymują powiadomienia o wiadomościach z komunikatora i poczty elektronicznej, a także zerkają na mały zegar w rogu ekranu. Każdy z tych procesów ma wykonywać swoje zadania, choć komputer musi nadzorować przebieg wielu prac. Tego rodzaju wielozadaniowość jest zwykle implementowana na poziomie procesów, jednak w niektórych sytuacjach podobny efekt trzeba osiągnąć w jednym procesie.

W tej książce wielowątkowość jest omawiana głównie jako technika do radzenia sobie z latencją. Na przykład aby umożliwić użytkownikowi kliknięcie przycisku *Anuluj* w czasie, gdy pobierany jest duży plik, programista musi utworzyć dodatkowy wątek odpowiedzialny za ściąganie pliku. Ponieważ dane są pobierane w odrębnym wątku, użytkownik może zażądać anulowania tej operacji, a interfejs użytkownika nie jest blokowany na czas ściągania pliku.

Jeśli dostępna jest wystarczająca liczba rdzeni, by każdy wątek otrzymał rdzeń, wszystkie wątki mogą korzystać z własnych „małych komputerów”. Jednak częściej liczba wątków jest większa niż liczba rdzeni. Nawet powszechnie obecnie maszyny wielordzeniowe mają niewielką liczbę rdzeni, a każdy proces może uruchamiać dziesiątki wątków.

Aby poradzić sobie z rozbieżnością między dużą liczbą wątków a małą liczbą rdzeni, system operacyjny symuluje jednoczesną pracę wątków, **dzieląc czas na porcje**. System operacyjny przełącza wykonywane wątki tak szybko, że można odnieść wrażenie, iż działają one jednocześnie. Okres wykonywania przez procesor jednego wątku przed uruchomieniem następnego to **porcja** lub **kwant czasu**. Akt zmieniania wątku wykonywanego w danym rdzeniu to **przełączanie kontekstu**.

Cały proces wygląda podobnie jak korzystanie ze światłowodowej linii telefonicznej. Światłowód jest tu odpowiednikiem procesora, a każda rozmowa jest jak wątek. Światłowodowa linia telefoniczna (w trybie single) może w danym momencie przesyłać tylko jeden sygnał, jednak rozmowy może jednocześnie prowadzić wiele osób. Światłowód jest na tyle wydajny, że pozwala tak szybko przełączać rozmowy, by każda z nich wydawała się ciągła. Podobnie każdy wątek w procesie wielowątkowym sprawia wrażenie, że działa jednocześnie z innymi wątkami.

Jeśli dwie operacje są wykonywane równolegle (albo dzięki rzeczywistej równoległości opartej na wielu rdzeniach, albo w wyniku symulowania równoległości dzięki podziałowi czasu na porcje), są **współbieżne**. Aby taka współbieżność była możliwa, kod należy wywoływać **asynchronicznie**, tak by wykonywanie i ukończenie wywołanej operacji były niezależne od przepływu sterowania, w którym tę operację uruchomiono. Tak więc współbieżność ma miejsce, gdy prace uruchomione asynchronicznie są wykonywane równolegle względem

przepływu sterowania. **Programowanie równolegle** polega na podziale jednego problemu na części, tak by można było **asynchronicznie** uruchamiać każdą część i przetwarzać je wszystkie asynchronicznie.

ZAGADNIENIE DLA POCZĄTKUJĄCYCH

Uwagi dotyczące wydajności

Wątek wykonujący zadanie zależne od operacji wejścia-wyjścia może być ignorowany przez system operacyjny do momentu, w którym podsystem wejścia-wyjścia zwróci wynik. Przełączanie się między wątkiem zależnym od operacji wejścia-wyjścia a wątkiem zależnym od procesora pozwala na wydajniejsze wykorzystanie mocy procesora, ponieważ nie czeka on bezczynnie na zakończenie operacji wejścia-wyjścia.

Jednak przełączanie kontekstu związane jest z kosztami. Trzeba zapisać w pamięci aktualny wewnętrzny stan procesora i wczytać stan powiązany z nowym wątkiem. Jeśli wątek A wykonuje liczne operacje na jednym fragmencie pamięci, a wątek B wykonuje dużo zadań związanych z inną porcją pamięci, przełączenie kontekstu między tymi wątkami spowoduje zapewne, że wszystkie dane wczytane do pamięci podręcznej przez wątek A zostaną zastąpione danymi z wątku B (lub na odwrót). Jeśli liczba wątków jest zbyt duża, koszt przełączania może wywierać odczuwalny wpływ na wydajność. Dodanie kolejnych wątków może skutkować dalszym spadkiem wydajności. W pewnym momencie może się okazać, że procesor więcej czasu zużywa na przełączanie się między wątkami niż na wykonywanie zadań w poszczególnych wątkach.

Nawet jeśli pominąć koszt przełączania kontekstu, samo dzielenie czasu na porcje może mieć istotny wpływ na wydajność. Założymy na przykład, że wykonywane są dwa zależne od procesora zadania o dużej latencji. Każde z nich przetwarza średnio dwie listy zawierające po miliard liczb. Przyjmijmy też, że procesor może wykonywać miliard operacji na sekundę. Jeśli każde z tych zadań jest wykonywane przez odrębny wątek, a oba wątki mogą korzystać z własnych rdzeni, oba wyniki można otrzymać w sekundę.

Jeśli jednak oba wątki korzystają z tego samego procesora, z powodu podziału czasu na porcje jeden wątek wykona kilkaset tysięcy operacji, następnie system przełączy się do drugiego wątku, potem ponownie do pierwszego itd. Każde zadanie zajmie wtedy sekundę czasu procesora, dlatego oba wyniki będą dostępne po 2 sekundach. Średni czas wykonania zadania wyniesie wtedy 2 sekundy (koszt przełączania kontekstu jest tu ignorowany).

Jeżeli te dwa zadania wykonasz w jednym wątku, który najpierw ukończy pierwsze zadanie, a dopiero potem zacznie wykonywać drugie, wynik pierwszego zadania będzie dostępny po sekundzie, a wynik drugiego — po następnej sekundzie. Średni czas wykonania zadania wyniesie wtedy 1,5 sekundy (zadania są kończone po 1 lub po 2 sekundach, co średnio daje 1,5 sekundy).

Wskazówki

NIE popełniaj częstego błędu, zakładając, że większa liczba wątków zawsze prowadzi do szybszego wykonania kodu.

MIERZ starannie wydajność, gdy za pomocą wielowątkowości starasz się przyspieszyć wykonywanie zadań zależnych od procesora.

ZAGADNIENIE DLA POCZĄTKUJĄCYCH

Problemy z wątkami

Kilkakrotnie wspomniano, że pisanie programów wielowątkowych jest skomplikowane i trudne. Nie wyjaśniono jednak, dlaczego tak jest. W skrócie można stwierdzić, że problem wynika z tego, iż wiele założeń prawdziwych w programach jednowątkowych nie jest uzasadnionych w programach wielowątkowych. Trudności z wielowątkowością dotyczą braku atomowości, sytuacji wyścigu, złożonego modelu zarządzania pamięcią i zakleszczenia.

Większość operacji nie jest atomowa

Operacja atomowa to taka, która dla obserwatorów zawsze jest albo nierozpoczęta, albo już ukończona. Z zewnątrz jej stan nigdy nie jest odbierany jako „w toku”. Przyjrzyj się na przykład poniższemu fragmentowi kodu:

```
if (bankAccounts.Checking.Balance >= 1000.00m)
{
    bankAccounts.Checking.Balance -= 1000.00m;
    bankAccounts.Savings.Balance += 1000.00m;
}
```

Ta operacja (sprawdzanie dostępnych środków, a następnie warunkowe obciążanie jednego konta i uznawanie drugiego) musi być atomowa. Oznacza to, że aby ten kod został prawidłowo wykonany, trzeba się upewnić, że dla obserwatora operacja nigdy nie jest częściowo ukończona. Wyobraź sobie na przykład, że dwa wątki równolegle wykonują ten kod. Może się zdarzyć, że oba wątki uznają, iż na koncie jest wystarczająca ilość środków. Wtedy oba wątki prześlą środki, choć ilość pieniędzy na koncie wystarcza tylko na dokonanie jednego transferu. To jeszcze nie koniec złych wiadomości — w tym fragmencie kodu *nie ma atomowych operacji!* Nawet operacje takie jak dodawanie z przypisaniem lub odejmowanie z przypisaniem, a także odczyt i zapis wartości właściwości typu decimal są w języku C# nieatomowe. Dlatego w programach wielowątkowych mogą zostać uznane za częściowo ukończone (kod może wykonać inkrementację lub dekrementację w nieprawidłowy sposób). Zaobserwowanie niespójnego stanu wynikającego z częściowo ukończonych operacji nieatomowych jest jednym z objawów ogólnego problemu nazywanego **sytuacją wyścigu**.

Niepewność spowodowana sytuacją wyścigu

Wcześniej wyjaśniono, że współbieżność często jest symulowana za pomocą podziału czasu na porcje. Gdy brakuje specjalnych struktur zarządzania przepływem sterowania (opisanych szczegółowo w rozdziale 22.), system operacyjny może przełączać wątki w dowolnym wybranym przez siebie momencie. Dlatego gdy dwa wątki chcą uzyskać dostęp do tego samego obiektu, nie da się przewidzieć, który wątek „wygra wyścig” i zostanie uruchomiony jako pierwszy. Na przykład jeśli dwa wątki wykonują przedstawiony wcześniej fragment kodu, możliwe, że jeden z nich wygra wyścig i uruchomi wszystkie instrukcje przed rozpoczęciem pracy przez drugi wątek. Możliwe też, że przełączenie kontekstu nastąpi po sprawdzeniu stanu konta przez pierwszy wątek. Po tej operacji drugi wątek może wtedy wygrać wyścig i jako pierwszy wykonać wszystkie instrukcje.

Działanie kodu, który grozi wystąpieniem sytuacji wyścigu, zależy od momentu przełączenia kontekstu. Zależność od tej czynności wprowadza niepewność co do przebiegu programu. Kolejność wykonywania danej instrukcji w stosunku do instrukcji z innego wątku jest niemożliwa do ustalenia. Najgorsze jest to, że kod, w którym występuje sytuacja wyścigu, może działać prawidłowo w 99,9% przypadków, jednak raz na tysiąc prób z powodu nieszczęśliwej kolejności wykonania instrukcji wyścig zostanie wygrany przez inny wątek. To ta nieprzewidywalność sprawia, że programowanie wielowątkowe jest tak trudne.

Ponieważ sytuację wyścigu trudno zreplikować w laboratorium, zapewnianie jakości kodu wielowątkowego wymaga długich testów obciążeniowych, specjalnie zaprojektowanych narzędzi do analizy kodu i znacznych nakładów na analizy i przeglądy kodu przeprowadzane przez ekspertów. Prawdopodobnie jeszcze ważniejszy jest nawyk maksymalnego upraszczania rozwiązań. W pogoni za potencjalnym wzrostem wydajności programiści często rezygnują z prostego stosowania blokad i wybierają niskopoziomowe mechanizmy takie jak operacje z klasy Interlocked i pola typu volatile, co zwiększa ryzyko popełnienia błędów. „Zachowaj prostotę” to jedna z najważniejszych wskazówek z dziedziny pisania kodu wielowątkowego.

Rozdział 22. jest poświęcony technikom radzenia sobie z sytuacją wyścigu.

Modele zarządzania pamięcią są skomplikowane

Już samo występowanie sytuacji wyścigu, gdzie dwa punkty sterowania mogą się „ścigać” w danym fragmencie kodu z nieprzewidywalną i zmienną szybkością, jest wystarczająco problematyczne. Ale to nie koniec kłopotów. Wyobraź sobie dwa wątki używające dwóch różnych procesorów, ale uzyskujące dostęp do tych samych pól jednego obiektu. Nowoczesne procesory nie uzyskują dostępu do pamięci głównej za każdym razem, gdy kod korzysta z danej zmiennej. Zamiast tego tworzona jest lokalna kopia w pamięci podręcznej procesora. Ta pamięć podręczna jest okresowo synchronizowana z pamięcią główną. To oznacza, że dwa wątki z dwóch procesorów wczytujące i zapisujące dane w tej samej lokalizacji mogą nie zauważyc zmian wprowadzonych przez siebie nawzajem. Inne zagrożenie to zaobserwowanie niespójnych danych. Powstaje więc sytuacja wyścigu związana z tym, kiedy procesory zdecydują się zsynchronizować pamięć podręczną.

Blokady mogą skutkować zakleszczeniem

Niezbędne są mechanizmy do przekształcania operacji nieatomowych w atomowe, do nakażywania systemowi operacyjnemu szeregowania wątków w celu uniknięcia sytuacji wyścigu, a także do upewniania się, że pamięć podręczna procesora będzie synchronizowana w odpowiednich momentach. Głównym mechanizmem używanym do rozwiązywania wszystkich tych problemów w programach w języku C# jest instrukcja lock, która służy do tworzenia blokad. Pozwala ona programistom oznać fragment kodu jako „krytyczny”, który może być wykonywany w danym momencie przez tylko jeden wątek. Jeśli kilka wątków spróbuje rozpoczęć wykonywanie sekcji krytycznej, system operacyjny wstrzyma² wszystkie te wątki oprócz jednego. System operacyjny gwarantuje też prawidłowe synchronizowanie pamięci podręcznej procesora po wykryciu blokady.

² W tym celu albo usypia wątek, albo uruchamia go w pętli ze sprawdzaniem warunków, albo na zmianę usypia go i sprawdza warunki.

Jednak same blokady też mogą prowadzić do problemów. Jeśli kolejność zajmowania blokad w wątkach jest różna, może nastąpić **zakleszczenie** skutkujące zawieszeniem pracy wątków. Każdy wątek czeka wtedy na zwolnienie blokady przez inny wątek.

Na rysunku 19.2 przedstawiono przykładową sytuację.

	Wątek A	Wątek B
Czas ↓	Zajmuje blokadę w sekcji a	Zajmuje blokadę w sekcji b
	Żąda blokady w sekcji b	Żąda blokady w sekcji a
	Zakleszczenie w oczekiwaniu na b	Zakleszczenie w oczekiwaniu na a

Rysunek 19.2. Oś czasu w scenariuszu zakleszczenia

Na tym etapie każdy wątek oczekuje na drugi. Tak więc oba wątki są zablokowane, co prowadzi do ogólnego zakleszczenia wykonywania tego kodu.

Różne techniki tworzenia blokad są szczegółowo opisane w rozdziale 22.

Wskazówki

NIE przyjmuj nieuzasadnionego założenia, że każda operacja atomowa w zwykłym kodzie będzie atomowa także w kodzie wielowątkowym.

NIE zakładaj, że wszystkie wątki wykryją w tej samej kolejności wszystkie efekty uboczne operacji wpływające na współużytkowaną pamięć.

DBAJ o to, by kod jednocześnie wymagający kilku blokad zawsze zajmował je w tej samej kolejności.

UNIKAJ wszelkich sytuacji wyścigu, czyli kodu, w którym działanie programu zależy od tego, w jaki sposób system operacyjny zdecyduje się szeregować wątki.

Początek
4.0

Zadania asynchroniczne

Z programowaniem wielowątkowym związane są następujące komplikacje:

1. *Monitorowanie stanu asynchronicznej operacji pod kątem jej zakończenia.* Wymaga to ustalenia, kiedy asynchroniczna operacja została ukończona. Wskazane jest, by nie posługiwać się odpływaniem cyklicznym do sprawdzania stanu wątku oraz nie stosować techniki blokowania i oczekiwania.
2. *Korzystanie z puli wątków.* Pozwala to uniknąć znaczących kosztów związanych z uruchamianiem i usuwaniem wątków. Pula wątków chroni też przed utworzeniem zbyt dużej liczby wątków, skutkującej tym, że system więcej czasu poświęca na przełączanie wątków niż na wykonywanie ich kodu.
3. *Unikanie zakleszczenia.* Należy zapobiegać zakleszczeniu, a jednocześnie chronić dane przed jednoczesnym dostępem do nich przez dwa wątki.
4. *Zapewnianie atomowości operacji i synchronizacji dostępu do danych.* Zapewnienie synchronizacji grup operacji gwarantuje, że te operacje będą wykonywane jak odrębna jednostka i że będą przerywane przez inne wątki tylko we właściwy sposób. Blokady pozwalają chronić dane przed jednoczesnym dostępem do nich przez dwa różne wątki.

Ponadto wszędzie tam, gdzie używane są długotrwałe metody, programowanie wielowątkowe jest zwykle niezbędne (długotrwałe metody należy wykonywać asynchronicznie). Niestety, w wersjach starszych niż C# 5.0 w kodzie wielowątkowym trzeba było stosować niskopoziomową klasę `System.Threading.Thread`, która nie oferowała wzorców pozwalających uniknąć komplikacji.

Jednak ponieważ programiści piszą coraz więcej kodu wielowątkowego, pojawiły się standardowe scenariusze i służące do radzenia sobie z nimi wzorce programowania. Aby uprościć model programowania, w wersji C# 5.0 uwzględniono takie wzorce i wprowadzono nowy typ związany z wątkami: `System.Threading.Tasks.Task`. Znacznie uprościł on programowanie z użyciem jednego z wzorców, TAP³, wykorzystując bibliotekę TPL⁴ z platformy .NET 4.0 i wzbogacając język C# o nowe techniki związane z tym wzorcem. W tym i w następnych podrozdziałach szczegółowo opisano bibliotekę TPL, a także stosowanie jej razem z kontekstowymi słowami kluczowymi `async` i `await`, które upraszczają programowanie z wykorzystaniem wzorca TAP.

Po co stosować bibliotekę TPL?

Tworzenie wątku to stosunkowo kosztowna operacja. Ponadto każdy wątek zajmuje dużą ilość pamięci wirtualnej (w systemie Windows domyślnie jest to 1 megabajt). Wcześniej w tym rozdziale zobaczyłeś, że wydajniejsze może się okazać używanie puli wątków. Wtedy można zaalokować wątki wówczas, gdy są potrzebne, przypisać asynchroniczną pracę do wątku, wykonać tę pracę do końca, a następnie ponownie wykorzystać ten sam wątek do wykonania kolejnej asynchronicznej pracy, zamiast usuwać wątek po ukończeniu pracy i potem tworzyć nowy.

W platformie .NET 4 i jej nowszych wersjach zamiast za każdym razem, gdy rozpoczyna się praca asynchroniczna, tworzyć wątek systemu operacyjnego, biblioteka TPL tworzy obiekt typu `Task` i informuje **program szeregujący zadania** o asynchronicznej pracy do wykonania. Program szeregujący zadania może się posługiwać różnymi strategiami, jednak domyślnie żąda wątku roboczego z puli. Pula wątków, co pokazano wcześniej, może uznać, że bardziej wydajne będzie uruchomienie zadania później, po ukończeniu obecnie wykonywanego zadania. Może też przydzielić do konkretnego procesora wątek roboczy wykonujący dane zadanie. To pula wątków określa, czy wydajniej jest utworzyć zupełnie nowy wątek, czy ponownie wykorzystać istniejący wątek, który wcześniej zakończył pracę.

W wyniku przedstawienia pracy asynchronicznej w abstrakcyjnej postaci w klasie `Task` biblioteka TPL zapewnia obiekty reprezentujące pracę asynchroniczną i udostępnia obiektowy interfejs API służący do interakcji z tą pracą. Ponadto dzięki udostępnianiu obiektu, który reprezentuje jednostkę pracy, biblioteka TPL umożliwia programowe budowanie procesów przepływu pracy w wyniku łączenia mniejszych zadań w większe (dalej zapoznasz się z takim rozwiązaniem).

Zadanie to obiekt obejmujący asynchronicznie wykonywaną pracę. Powinno to brzmieć znajomo. *Delegat* także jest obiektem, który reprezentuje kod. Różnica między zadaniem

³ W ramach przypomnienia — jest to akronim od słów *Task-based Asynchronous Pattern* (czyli wzorzec asynchroniczności opartej na zadaniach).

⁴ Jest to akronim od słów *Task Parallel Library* (czyli biblioteka do obsługi zadań równoległych).

i delegatem polega na tym, że delegaty są **synchroniczne**, a zadania są **asynchroniczne**. Uruchomienie delegata (na przykład typu Action) skutkuje natychmiastowym przeniesieniem punktu sterowania z bieżącego wątku do kodu delegata. Do czasu zakończenia działania delegata sterowanie nie jest zwracane do jednostki wywołującej. Natomiast uruchomienie zadania powoduje prawie natychmiastowe zwrócenie sterowania do jednostki wywołującej. Nie ma znaczenia, jak dużo pracy zadanie ma wykonać. Zadanie działa asynchronicznie, zwykle w innym wątku (choć, jak zobaczysz w rozdziale 20., możliwe, a nawet korzystne jest asynchroniczne wykonywanie zadań w tylko jednym wątku). Zadanie powoduje przekształcenie wzorca wykonywania delegata z postaci synchronicznej w asynchroniczną.

Wprowadzenie do asynchronicznych zadań

Wiadomo, w jakim momencie delegat zakończył pracę w bieżącym wątku, ponieważ wcześniej jednostka wywołująca nie może wznowić działania. Jak jednak ustalić moment zakończenia pracy przez zadanie? Ponadto jak pobrać z niego wynik, jeśli jest zwracany? Pomyśl o tym, co się stanie po przekształceniu synchronicznego delegata w asynchroniczne zadanie. Wątek roboczy będzie wyświetlał w konsoli plusy, a wątek główny — minusy.

Uruchomienie zadania powoduje pobranie nowego wątku z puli, utworzenie drugiego punktu sterowania i wykonanie kodu delegata w nowym wątku. Na listingu 19.1 po wywołaniu uruchamiającym zadanie (Task.Run()) w wątku głównym kod jest normalnie wykonywany od punktu sterowania.

Listing 19.3. Wywoływanie asynchronicznego zadania

```
4.0
using System;
using System.Threading.Tasks;

public class Program
{
    public static void Main()
    {
        const int Repetitions = 10000;
        // Aby zastosować bibliotekę TPL w wersjach starszych niż .NET 4.5,
        // wywołaj metodę Task.Factory.StartNew<string>().
        Task task = Task.Run(() =>
        {
            for (int count = 0;
                count < repetitions; count++)
            {
                Console.Write('-');
            }
        });
        for (int count = 0; count < repetitions; count++)
        {
            Console.Write('+');
        }

        // Czekanie do momentu zakończenia zadania.
        task.Wait();
    }
}
```

Kod, który ma być uruchamiany w nowym wątku, jest definiowany w delegacie (tu jest on typu `Action`) przekazywanym do metody `Task.Run()`. Ten delegat (w postaci wyrażenia lambda) wyświetla w konsoli znaki minus. Pętla znajdująca się po kodzie uruchamiającym zadanie działa prawie tak samo, ale wyświetla znaki plus.

Zauważ, że po wywołaniu metody `Task.Run()` przekazany jako argument delegat typu `Action` natychmiast rozpoczyna pracę. Obiekt typu `Task` jest tu „aktywny”, co oznacza, że sam rozpoczyna pracę. Można też tworzyć „nieaktywne” zadania, które trzeba jawnie uruchomić przed rozpoczęciem wykonywania asynchronicznej pracy.

Choć korzystając z konstruktora klasy `Task`, obiekty typu `Task` można tworzyć jako „nieaktywne”, zwykle jest to właściwe tylko w implementacji w interfejsie API, który zwraca „aktywny” obiekt tego typu, uruchamiany w wyniku wywołania metody `Task.Start()`.

Zauważ, że stan „aktywnego” zadania bezpośrednio po wywołaniu metody `Run()` jest nieokreślony. Stan zależy od systemu operacyjnego, jego obciążenia i używanej biblioteki do obsługi zadań. Na tej podstawie ustalane jest, czy wywołanie `Run()` ma wykonać wątek roboczy natychmiast, czy opóźnić ten moment do czasu, gdy dostępne staną się dodatkowe zasoby. Może się też okazać, że „aktywny” wątek zakończy pracę do czasu, w którym kod z wątku wywołującego będzie miał okazję wznowić działanie. Wywołanie `Wait()` wymusza na wątku głównym oczekивание на zakończenie wykonywania całej pracy przypisanej do zadania.

W omawianym scenariuszu używane jest jedno zadanie, można jednak wykonywać asynchronicznie wiele zadań. Często tworzony jest zestaw zadań, a program ma czekać na ukończenie ich wszystkich (lub dowolnego z nich) przed wznowieniem działania bieżącego wątku. Używane są wtedy metody `Task.WaitAll()` i `Task.WaitAny()`.

Do tej pory zobaczyłeś, że zadanie może przyjąć delegat typu `Action` i uruchomić go asynchronicznie. Co jednak zrobić, jeśli prace wykonywane w zadaniu wymagają zwrócenia wyniku? Można wtedy wykorzystać typ `Task<T>`, by asynchronicznie uruchomić delegat typu `Func<T>`. Gdy delegat jest wykonywany synchronicznie, wiadomo, że sterowanie zostanie zwrócone dopiero po uzyskaniu wyniku. Jeśli asynchronicznie uruchamiasz obiekt typu `Task<T>`, możesz kierować do niego zapytania w jednym wątku, by sprawdzać, czy zakończył pracę, i pobierać wynik, jeśli jest już dostępny⁵. Na listingu 19.2 pokazano, jak uzyskać ten efekt w aplikacji konsolowej. Zauważ, że w tym przykładzie zastosowano metodę `PiCalculator.Calculate()`, opisaną szczegółowo w podrozdziale „Równolegle wykonywanie iteracji pętli” w rozdziale 21.

Listing 19.2. Odpytywanie dotyczące obiektu typu `Task<T>`

```
using System;
using System.Threading.Tasks;

public class Program
{
```

⁵ Zachowaj ostrożność, gdy stosujesz technikę odpytywania cyklicznego. Jeśli tworzysz zadanie na podstawie delegata (tak jak w omawianym przykładzie), zadanie jest uruchamiane w wątku roboczym z puli. W efekcie bieżący wątek będzie wykonywał pętlę do czasu zakończenia pracy przez wątek roboczy. Ta technika działa, ale może niepotrzebnie zużywać zasoby procesora. Może też prowadzić do błędów, jeśli programista zamiast zaszeregować zadania w wątku roboczym, zechce uruchomić je w przyszłości w bieżącym wątku. Ponieważ bieżący wątek będzie w pętli stosował odpytywanie cykliczne do zadania, nigdy z niej nie wyjdzie, gdyż zadania nie można ukończyć przed wyjściem bieżącego wątku z pętli.

```

public static void Main()
{
    // Aby zastosować bibliotekę TPL w wersjach starszych niż .NET 4.5,
    // wywołaj metodę Task.Factory.StartNew<string>().
    Task<string> task =
        Task.Run<string>(
            () => PiCalculator.Calculate(100));

    foreach (
        char busySymbol in Utility.BusySymbols())
    {
        if (task.IsCompleted)
        {
            Console.Write('\b');
            break;
        }
        Console.Write(busySymbol);
    }

    Console.WriteLine();

    Console.WriteLine(task.Result);
    System.Diagnostics.Trace.Assert(
        task.IsCompleted);
}
}

public class PiCalculator
{
    public static string Calculate(int digits = 100)
    {
        // ...
    }
}

public class Utility
{
    public static IEnumerable<char> BusySymbols()
    {
        string busySymbols = "-\\|/-\\|/";
        int next = 0;
        while(true)
        {
            yield return busySymbols[next];
            next = (next + 1) % busySymbols.Length;
            yield return '\b';
        }
    }
}

```

4.0

Na tym listingu pokazano, że typ danych zadania to `Task<string>`. Ten generyczny typ obejmuje właściwość `Result`, z której można pobrać wartość zwróconą przez delegat typu `Func<string>` wykonywany przez zadanie `Task<string>`.

Zauważ, że kod z listingu 19.2 nie wywołuje metody `Wait()`. Wczytywanie wartości z właściwości `Result` automatycznie powoduje, że bieżący wątek jest blokowany do momentu, w którym wynik będzie dostępny (jeśli jeszcze nie jest). Dzięki temu wiadomo, że gdy wynik zostanie pobrany, zadanie będzie już wykonane.

Oprócz właściwości `IsCompleted` i `Result` typ `Task<T>` udostępnia też kilka innych wertych uwagi właściwości:

- Właściwość `IsCompleted` jest ustawiana na wartość `true`, gdy zadanie zakończy pracę (niezależnie od tego, czy ukończy działanie normalnie, czy w wyniku błędu — z powodu zgłoszenia wyjątku). Bardziej szczegółowe informacje na temat stanu zadania można uzyskać za pomocą właściwości `Status`, która zwraca wartość typu `TaskStatus`. Dostępne wartości to: `Created`, `WaitingForActivation`, `WaitingToRun`, `Running`, `WaitingForChildrenToComplete`, `RanToCompletion`, `Canceled` i `Faulted`. Właściwość `IsCompleted` zwraca wartość `true`, gdy właściwość `Status` ma wartość `RanToCompletion`, `Canceled` lub `Faulted`. Oczywiście, jeśli zadanie działa w innym wątku i wczytany zostanie status informujący o tym, że zadanie wciąż działa, w dowolnym momencie (także bezpośrednio po ostatnim odczycie) status może się zmienić na wartość, która oznacza ukończenie pracy. To samo dotyczy wielu innych stanów. Nawet stan `Created` może się zmienić, jeśli inny wątek uruchomi zadanie. Tylko stany `RanToCompletion`, `Canceled` i `Faulted` są ostateczne i nie można z nich przejść do innych stanów.
- Zadanie można jednoznacznie zidentyfikować na podstawie wartości właściwości `Id`. Statyczna właściwość `Task.CurrentId` zwraca identyfikator aktualnie wykonywanego zadania (tego, które wykonuje wywołanie `Task.CurrentId`). Te właściwości są przydatne zwłaszcza w trakcie debugowania.
- Właściwość `AsyncState` możesz wykorzystać do powiązania z zadaniem dodatkowych danych. Wyobraź sobie na przykład obiekt typu `List<T>`, którego wartości są obliczane przez różne zadania. W każdym zadaniu można zapisać we właściwości `AsyncState` indeks wartości. Dzięki temu po zakończeniu wykonywania zadania kod na podstawie wartości właściwości `AsyncState` może znaleźć indeks elementu listy (po zrzutowaniu liczby na typ `int`)⁶.

Istnieją też inne przydatne właściwości. Ich omówienie znajdziesz w dalszej części rozdziału w podrozdziale „Anulowanie zadania”.

Kontynuacja zadania

Kilkakrotnie wspomniano w książce o przepływie sterowania w programie, nie wspominając jednak o najbardziej podstawowym aspekcie tego przepływu — *przepływ sterowania określa, co stanie się w następnym kroku*. Czasem przepływ sterowania jest prosty. Na przykład w wyrażeniu `Console.WriteLine(x.ToString())`; przepływ sterowania określa, że jeśli metoda `ToString` zakończy pracę w normalny sposób, następnie wywołana zostanie metoda `WriteLine` z argumentem w postaci wartości zwrotnej przez metodę `ToString`. **Kontynuacja** to właśnie to, co „stanie się w następnym kroku”. Każdy punkt w przepływie sterowania ma określoną kontynuację. W przykładowym wyrażeniu kontynuacją wywołania `ToString` jest

⁶ Zachowaj ostrożność, gdy posługujesz się zadaniami do asynchronicznego modyfikowania kolekcji. Zadania mogą działać w wątkach roboczych, a kolekcja może nie być bezpieczna ze względu na wątki. Bezpieczniej jest zapełnić kolekcję w wątku głównym, po zakończeniu wykonywania zadań.

wywołanie `WriteLine` (a kontynuacją wywołania `WriteLine` jest kod uruchamiany w następnej instrukcji). Zagadnienie kontynuacji jest tak podstawowe w programowaniu w języku C#, że większość programistów w ogóle się nad nim nie zastanawia. Jest ono jak niewidzialne powietrze, jakim oddychają. Proces programowania w języku C# polega na tworzeniu kolejnych kontynuacji do momentu, gdy przepływ sterowania całego programu się zakończy.

Zauważ, że kontynuacja danego fragmentu kodu w normalnym programie w języku C# jest wykonywana *natychmiast* po zakończeniu pracy tego fragmentu. Gdy metoda `ToString()` zwróci sterowanie, punkt sterowania w bieżącym wątku natychmiast synchronicznie wywoła metodę `WriteLine`. Zwróć uwagę, że dla danego fragmentu kodu istnieją dwie możliwe kontynuacje — *normalna* i *wyjątkowa* (realizowana, gdy dany fragment kodu zgłosi wyjątek).

Asynchroniczne wywołania metod, na przykład uruchomienie obiektu typu `Task`, dodają kolejny wymiar do przepływu sterowania. Asynchroniczne wywołanie obiektu typu `Task` sprawia, że przepływ sterowania może natychmiast przejść do instrukcji umieszczonej po wyrażeniu `Task.Start()`, a jednocześnie rozpoczęć wykonywanie kodu delegata powiązanego z obiektem typu `Task`. Tak więc w kodzie asynchronicznym to, co „stanie się w następnym kroku”, może mieć wiele wymiarów. Inaczej niż w przypadku wyjątków, gdzie kontynuacja to tylko inna ścieżka wykonania, w kodzie asynchronicznym powstaje dodatkowa, równoległa ścieżka.

Zadania asynchroniczne umożliwiają też łączenie mniejszych zadań w większe za pomocą asynchronicznych kontynuacji. W zadaniu (podobnie jak w zwykłym przepływie sterowania) można tworzyć różne kontynuacje na potrzeby obsługi błędów. Zadania można też łączyć ze sobą, manipulując kontynuacjami. Istnieje kilka służących do tego technik. Najbardziej bezpośrednia z nich polega na użyciu metody `ContinueWith()` (zobacz listing 19.3 i powiązane z nim dane wyjściowe 19.1).

Listing 19.3. Wywoływanie metody `Task.ContinueWith()`

```
4.0
using System;
using System.Threading.Tasks;

public class Program
{
    public static void Main()
    {
        Console.WriteLine("Przed");
        // Aby zastosować bibliotekę TPL w wersjach starszych niż .NET 4.5,
        // wywołaj metodę Task.Factory.StartNew<string>().
        Task taskA =
            Task.Run( () =>
                Console.WriteLine("Rozpoczynanie..."))
                    .ContinueWith(antecedent =>
                Console.WriteLine("Kontynuacja zadania A..."));
        Task taskB = taskA.ContinueWith( antecedent =>
            Console.WriteLine("Kontynuacja zadania B..."));
        Task taskC = taskA.ContinueWith( antecedent =>
            Console.WriteLine("Kontynuacja zadania C..."));
        Task.WaitAll(taskB, taskC);
        Console.WriteLine("Zakończono!");
    }
}
```

DANE WYJŚCIOWE 19.1.

```
Przed
Rozpoczynanie...
Kontynuacja zadania A...
Kontynuacja zadania C...
Kontynuacja zadania B...
Zakończono!
```

Metoda `ContinueWith()` umożliwia połączenie dwóch zadań razem w taki sposób, że gdy pierwsze zadanie (**zadanie poprzedzające**) zakończy pracę, drugie zadanie (**zadanie kontynuacyjne**) zostanie automatycznie uruchomione w trybie asynchronicznym. Na listingu 19.3 wyrażenie `Console.WriteLine("Rozpoczynanie...")` to ciało zadania poprzedzającego, a wyrażenie `Console.WriteLine("Kontynuacja zadania A...")` to ciało zadania kontynuacyjnego. Zadanie kontynuacyjne przyjmuje obiekt antecedent typu `Task` jako argument. Dzięki temu kod zadania kontynuacyjnego ma dostęp do stanu informującego o ukończeniu zadania poprzedzającego. Gdy zadanie poprzedzające zakończy pracę, zadanie kontynuacyjne natychmiast zostanie rozpoczęte i asynchronicznie uruchomi drugi delegat. Do tego delegata jako argument przekazywane jest właśnie ukończone zadanie poprzedzające. Ponadto ponieważ metoda `ContinueWith()` zwraca obiekt typu `Task`, ten obiekt można wykorzystać jako zadanie poprzedzające dla następnego zadania. W ten sposób można utworzyć dowolnie długie łańcuchów zadań.

Jeśli dwukrotnie wywołasz metodę `ContinueWith()` dla tego samego zadania poprzedzającego (na listingu 19.3 zadania `taskB` i `taskC` są zadaniami kontynuacyjnymi dla `taskA`), to zadanie poprzedzające będzie miało dwa zadania kontynuacyjne. Wtedy ukończenie zadania poprzedzającego prowadzi do asynchronicznego wykonania obu zadań kontynuacyjnych. Zauważ, że kolejność wykonywania zadań kontynuacyjnych dla jednego zadania poprzedzającego jest na etapie komplikacji nieokreślona. Z danych wyjściowych 19.1 wynika, że zadanie `taskC` zostało wykonane przed zadaniem `taskB`, ale w następnym przebiegu programu kolejność zadań może zostać odwrócona. Jednak zadanie `taskA` zawsze jest wywoływanie przed zadaniami `taskB` i `taskC`, ponieważ te dwa ostatnie są zadaniami kontynuacyjnymi zadania `taskA` i dlatego nie mogą zostać rozpoczęte przed ukończeniem `taskA`. Podobnie delegat z instrukcją `Console.WriteLine("Rozpoczynanie...")` zawsze jest wykonywany przed wyrażeniem (`Console.WriteLine("Kontynuacja zadania A...")`), ponieważ to drugie wyrażenie należy do zadania kontynuacyjnego względem zadania ze wspomnianą instrukcją. Tekst "Zakończono!" zawsze pojawi się na końcu. Wynika to z wywołania `Task.WaitAll(taskB, taskC)`, które blokuje przepływ sterowania do czasu zakończenia zadań `taskB` i `taskC`.

Dostępnych jest wiele przeciążonych wersji metody `ContinueWith()`. Niektóre z nich przyjmują wartości typu `TaskContinuationOptions`, które pozwalają zmienić działanie łańcucha z kontynuacjami. Te wartości to flagi, dlatego można je ze sobą łączyć za pomocą logicznego operatora OR (`|`). Krótki opis wybranych flag znajdziesz w tabeli 19.1. Więcej szczegółów zawiera dostępna w internecie dokumentacja MSDN⁷.

⁷ Zobacz stronę <https://docs.microsoft.com/dotnet/api/system.threading.tasks.taskcontinuationoptions> w witrynie Microsoft .NET Docs.

Tabela 19.1. Lista dostępnych wartości wyliczeniowych typu TaskContinuationOptions

Wartość wyliczeniowa	Opis
None	Jest to ustawienie domyślne. Zadanie kontynuacyjne jest wtedy wykonywane po zakończeniu zadania poprzedzającego niezależnie od statusu tego ostatniego.
PreferFairness	Domyślnie jest tak, że jeśli dwa zadania zostały zgłoszone asynchronicznie jedno przed drugim, nie ma gwarancji, że pierwsze rzeczywiście zostanie wcześniej uruchomione. Ta flaga informuje program szeregujący zadania, aby zwiększył prawdopodobieństwo wcześniejszego uruchomienia zadania zgłoszonego jako pierwsze. Jest to istotne zwłaszcza w sytuacji, gdy dwa zadania są tworzone z użyciem różnych wątków z puli.
LongRunning	Ta flaga informuje program szeregujący, że zadanie może trwać długo i być zależne od operacji wejścia-wyjścia. Program szeregujący może wtedy umożliwić przetworzenie innych zadań z kolejki, by nie ryzykować „zagłodzenia” ich przez długie zadanie. Nie należy nadużywać tej opcji.
AttachedToParent	Określa, że zadanie należy spróbować dołączyć do zadania nadziednego w hierarchii.
DenyChildAttach (.NET 4.5)	Powoduje zgłoszenie wyjątku przy próbie utworzenia zadania podzadnego. Jeśli w kodzie kontynuacji znajdzie się opcja AttachedToParent, program zadziała tak, jakby nie było żadnego zadania nadziednego.
NotOnRanToCompletion*	Określa, że zadania kontynuacyjnego nie należy wykonywać, jeśli zadanie poprzedzające zostało wykonane do końca. Ta opcja jest niedozwolona w kontynuacjach z wieloma zadaniami (ang. <i>multitask</i>).
NotOnFaulted*	Określa, że zadania kontynuacyjnego nie należy wykonywać, jeśli zadanie poprzedzające zgłosiło nieobsłużony wyjątek. Ta opcja jest niedozwolona w kontynuacjach z wieloma zadaniami.
OnlyOnCanceled*	Określa, że zadanie kontynuacyjne należy wykonać tylko w sytuacji, gdy jego zadanie poprzedzające zostało anulowane. Ta opcja jest niedozwolona w kontynuacjach z wieloma zadaniami.
NotOnCanceled*	Określa, że zadania kontynuacyjnego nie należy wykonywać, jeśli zadanie poprzedzające zostało anulowane. Ta opcja jest niedozwolona w kontynuacjach z wieloma zadaniami.
OnlyOnFaulted*	Określa, że zadanie kontynuacyjne należy wykonać tylko w sytuacji, gdy jego zadanie poprzedzające zgłosiło nieobsłużony wyjątek. Ta opcja jest niedozwolona w kontynuacjach z wieloma zadaniami.
OnlyOnRanToCompletion*	Określa, że zadanie kontynuacyjne należy wykonać tylko w sytuacji, gdy jego zadanie poprzedzające zostało wykonane do końca. Ta opcja jest niedozwolona w kontynuacjach z wieloma zadaniami.

Tabela 19.1. Lista dostępnych wartości wyliczeniowych typu TaskContinuationOptions — ciąg dalszy

Wartość wyliczeniowa	Opis
ExecuteSynchronously	Określa, że zadanie kontynuacyjne należy wykonać synchronicznie. Gdy używana jest ta opcja, kontynuacja uruchamiana przez program szeregujący będzie działała w tym samym wątku, który spowodował przejście zadania poprzedzającego do ostatecznego stanu. Jeśli w momencie tworzenia kontynuacji zadanie poprzedzające jest już ukończone, kontynuacja zostanie uruchomiona w wątku, w którym ją utworzono.
HideScheduler (.NET 4.5)	Zapobiega traktowaniu bieżącego programu szeregującego jako domyślnego programu szeregującego w tworzonym zadaniu. To oznacza, że dla operacji takich jak Run, StartNew i ContinueWith wykonywanych w utworzonym zadaniu bieżącym programem szeregującym będzie TaskScheduler.Default(null). Jest to przydatne, gdy kontynuacja powinna być zarządzana przez określony program szeregujący, natomiast wywoływany przez nią dodatkowy kod nie powinien być obsługiwany przez ten sam program szeregujący.
LazyCancellation (.NET 4.5)	Powoduje, że kontynuacja odracza sprawdzanie znacznika anulowania (uwzględnianego w żądaniach anulowania) do czasu zakończenia pracy zadania poprzedzającego. Rozważ zadania t1, t2 i t3, gdzie każde następne jest kontynuacją wcześniejszego. Jeśli t2 zostanie anulowane przed zakończeniem pracy przez t1, to t3 może zostać rozpoczęte przed ukończeniem pracy przez t1. Flaga LazyCancellation pozwala tego uniknąć.
RunContinuationAsynchronously (.NET 4.6)	Określa, że zadanie kontynuacyjne należy uruchomić asynchronicznie. Nawet jeśli dane zadanie samo jest kontynuacją, ta opcja nie wpływa na sposób jego wykonywania, tylko na pracę kontynuacji tego zadania. Zadania będące kontynuacją można tworzyć z jednociennym użyciem flag TaskContinuationOptions.ExecuteSynchronously i TaskContinuationOptions.RunContinuationsAsynchronously. Pierwsza powoduje, że kontynuacja jest uruchamiana synchronicznie po zakończeniu pracy zadania poprzedzającego. Druga sprawia, że kontynuacje kontynuacji będą uruchamiane asynchronicznie.

Pozycje opatrzone w tabeli 19.1 gwiazdką (*) określają, w jakich warunkach zadanie kontynuacyjne będzie wykonywane. Dlatego są bardzo przydatne do tworzenia kontynuacji, które działają jak metody obsługi zdarzeń związane z działaniem zadania poprzedzającego. Na listingu 19.4 pokazano, że z zadaniem poprzedzającym można połączyć wiele kontynuacji, uruchamianych warunkowo w zależności od tego, w jaki sposób zadanie poprzedzające zakończyło pracę.

Listing 19.4. Używanie metody ContinueWith() do rejestrowania zainteresowania powiadomieniami o działaniu zadania

```

using System;
using System.Threading.Tasks;
using System.Diagnostics;
using AddisonWesley.Michaelis.EssentialCSharp.Shared;

public class Program
{
    public static void Main()
    {
        // Aby zastosować bibliotekę TPL w wersjach starszych niż .NET 4.5,
// wywołaj metodę Task.Factory.StartNew<string>().
        Task<string> task =
            Task.Run<string>(
                () => PiCalculator.Calculate(10));

        Task faultedTask = task.ContinueWith(
            (antecedentTask) =>
        {
            Trace.Assert(antecedentTask.IsFaulted);
            Console.WriteLine(
                "Stan zadania: IsFaulted");
        },
        TaskContinuationOptions.OnlyOnFaulted);

        Task canceledTask = task.ContinueWith(
            (antecedentTask) =>
        {
            Trace.Assert(antecedentTask.IsCanceled);
            Console.WriteLine(
                "Stan zadania: IsCanceled");
        },
        TaskContinuationOptions.OnlyOnCanceled);

        4.0
        Task completedTask = task.ContinueWith(
            (antecedentTask) =>
        {
            Trace.Assert(antecedentTask.IsCompleted);
            Console.WriteLine(
                "Stan zadania: IsCompleted");
        },
        TaskContinuationOptions.
        OnlyOnRanToCompletion);

        completedTask.Wait();
    }
}

```

Kod z tego listingu rejestruje *odbiorniki zdarzeń* zadania poprzedzającego, dlatego gdy to zadanie zakończy pracę (w zwykły lub nietypowy sposób), rozpoczęcie się wykonywanie odpowiedniego zadania z odbiornikiem. Daje to bardzo duże możliwości, zwłaszcza jeśli pierwotne zadanie jest typu „uruchom i zapomnij” (program tylko uruchamia je, łączy z zadaniami kontynuacyjnymi, a następnie nigdy do niego nie wraca).

Zauważ, że na listingu 19.4 ostatnie wywołanie `Wait()` dotyczy zadania `completedTask`, a nie zadania `task` (czyli pierwotnego zadania poprzedzającego utworzonego za pomocą polecenia `Task.Run()`). Choć w każdym delegacie dostępna jest referencja `antecedentTask` prowadząca do nadrzednego (poprzedzającego) zadania `task`, poza odbiornikami można pominąć referencję do pierwotnego zadania `task`. Pozwala to polegać tylko na zadaniach kontynuacyjnych, które asynchronicznie rozpoczynają działanie. Dodatkowy kod, sprawdzający stan pierwotnego zadania `task`, nie jest potrzebny.

Na listingu wywoływana jest metoda `completedTask.Wait()`, dlatego wątek główny kończy pracę programu dopiero po pojawienniu się informacji o ukończeniu zadania (zobacz dane wyjściowe 19.2).

DANE WYJŚCIOWE 19.2.

Stan zadania: `IsCompleted`.

W tym przykładzie wywołanie metody `completedTask.Wait()` nie ma większego sensu, ponieważ wiadomo, że pierwotne zadanie z powodzeniem zakończy pracę. Jednak wywołanie metody `Wait()` dla zadań `canceledTask` lub `faultedTask` spowoduje zgłoszenie wyjątku. Te zadania kontynuacyjne są uruchamiane tylko wtedy, jeśli zadanie poprzedzające zostało anulowane lub zgłosiło wyjątek. Ponieważ w tym programie tak się nie stanie, wymienione zadania nigdy nie zostaną uruchomione, a oczekiwanie na ich ukończenie zakończy się zgłoszeniem wyjątku. Poszczególne kontynuacje na listingu 19.1 wzajemnie się wykluczają, dlatego gdy zadanie poprzedzające zostanie wykonane do końca oraz ukończone zostanie zadanie `completedTask`, program szeregujący automatycznie anuluje zadania `canceledTask` i `faultedTask`. Stan anulowanych zadań jest ustalany na `Canceled`. Dlatego wywołanie metody `Wait()` (lub innej instrukcji wymagającej od bieżącego wątku oczekiwania na ukończenie zadań) dla któregoś z tych zadań doprowadzi do zgłoszenia wyjątku z informacją o ich anulowaniu. Bardziej sensownym podejściem byłoby tu wywołanie metody `Task.WaitAny(completedTask, canceledTask, faultedTask)`, która zgłosi wymagający obsłużenia wyjątek `AggregateException`.

4.0

Używanie wyjątków `AggregateException` do obsługi nieobsłużonych wyjątków w zadaniach

Gdy wywołujesz metodę synchronicznie, możesz umieścić ją w bloku `try` i dodać klauzulę `catch`, aby poinformować kompilator, jaki kod należy wykonać po wystąpieniu wyjątku. Ta technika nie sprawdza się jednak dla wywołań asynchronicznych. Nie wystarczy umieścić wywołania `Start()` w bloku `try`, by przechwytywać wyjątki. Dzieje się tak, ponieważ sterowanie opuszcza blok `try` (często na długo przed zgłoszeniem wyjątku w wątku roboczym). Możliwym rozwiązaniem jest umieszczenie ciała delegata powiązanego z zadaniem w bloku `try/catch`. Wyjątki zgłasiane i przechwytywane przez wątek roboczy nie stanowią wtedy problemu, ponieważ blok `try` będzie działał w normalny sposób w wątku roboczym. Inaczej jest jednak w przypadku wyjątków nieobsłużonych, których wątek roboczy nie przechwycił.

Zwykle (od wersji 2.0⁸ środowiska CLR) nieobsłużone wyjątki w dowolnym wątku są uznawane za krytyczne, co skutkuje wyświetleniem okna dialogowego do zgłoszania błędów w systemie operacyjnym i nieprawidłowym zamknięciem aplikacji. Wszystkie wyjątki we wszystkich wątkach trzeba przechwytywać. Jeśli wyjątki nie zostaną przechwycone, aplikacja nie będzie mogła kontynuować pracy (złożone techniki radzenia sobie z nieobsłużonymi wyjątkami omówiono dalej w zagadnienu dla zaawansowanych „Radzenie sobie z nieobsłużonymi wyjątkami z wątków”). Na szczęście nie dotyczy to nieobsłużonych wyjątków z asynchronicznie działających zadań. Program szeregujący umieszcza delegat w uniwersalnym bloku obsługi wyjątków, dlatego gdy zadanie zgłosi nieobsługiwany gdzie indziej wyjątek, uniwersalny blok przechwyci go i zarejestruje szczegółowe informacje o wyjątku w zadaniu. Pozwala to uniknąć automatycznego zamknięcia procesu przez środowisko CLR.

Na listingu 19.4 pokazano, że jedną z technik radzenia sobie z zadaniami zakończonymi błędem jest jawnie utworzenie zadania kontynuacyjnego, które działa jak metoda obsługi błędów z pierwotnego zadania. Program szeregujący automatycznie uruchamia wtedy kontynuację, gdy wykryje, że zadanie poprzedzające zgłosiło nieobsłużony wyjątek. Jeśli takie zadanie kontynuacyjne nie jest dostępne, a kod wywoła dla błędного zadania metodę `Wait()` lub spróbuje pobrać z niego wartość właściwości `Result`, zostanie zgłoszony wyjątek `AggregateException` (zobacz listing 19.5 i dane wyjściowe 19.3).

Listing 19.5. Obsługa nieobsłużonego wyjątku zgłoszonego w zadaniu

```
using System;
using System.Threading.Tasks;

public class Program
{
    public static void Main()
    {
        // Aby zastosować bibliotekę TPL w wersjach starszych niż .NET 4.5,
        // wywołaj metodę Task.Factory.StartNew<string>().
        Task task = Task.Run(() =>
        {
            throw new InvalidOperationException();
        });

        try
        {
            task.Wait();
        }
        catch(AggregateException exception)
        {
            exception.Handle(eachException =>
            {
                Console.WriteLine(
                    $"BŁĄD: { eachException.Message }");
            });
        }
    }
}
```

4.0

⁸ W wersji 1.0 środowiska CLR nieobsłużony wyjątek z wątku roboczego kończył pracę wątku, ale nie aplikacji. Dlatego w błędnych programach wszystkie wątki robocze mogły zakończyć pracę, ale wątek główny nadal działał, choć program nie wykonywał już żadnych operacji. Jest to mylące dla użytkowników. Lepiej poinformować użytkownika o nieprawidłowym stanie aplikacji i zamknąć ją, zanim spowoduje więcej problemów.

```
        return true;
    });
}
}
```

DANE WYJŚCIOWE 19.3.

BŁĄD: Operation is not valid due to the current state of the object.

Wyjątek AggregateException (czyli wyjątek zagregowany) nazywa się tak, ponieważ może zawierać wiele wyjątków zebranych z różnych zadań zakończonych niepowodzeniem. Wyobraź sobie na przykład, że równolegle wykonywanych jest asynchronicznie dziesięć zadań, a pięć z nich zgłosiło wyjątki. Aby zgłosić wszystkich pięć wyjątków i obsłużyć je w jednym bloku catch, platforma wykorzystuje wyjątek AggregateException, aby zebrać wyjątki i zgłosić je jako jeden. Ponadto ponieważ na etapie komplikacji nie wiadomo, czy wyjątek roboczy zgłosi jeden wyjątek, czy większą ich liczbę, zadania z nieobsłużonymi błędami zawsze zgłaszą wyjątki typu AggregateException. Ten model zademonstrowano na listingu 19.5 i w danych wyjściowych 19.3. Choć nieobsłużony wyjątek zgłoszony w wątku roboczym był typu InvalidOperationException, typ wyjątku przechwyconego w wątku głównym to AggregateException. Ponadto, zgodnie z oczekiwaniemi, przechwycenie zgłoszanego w kodzie wyjątku wymaga bloku catch dla typu AggregateException.

Lista wyjątków zapisanych w wyjątku typu AggregateException jest dostępna we właściwości InnerExceptions. Dlatego za pomocą tej właściwości można sprawdzić każdy wyjątek i ustalić odpowiedni tok postępowania. Inna możliwość, co pokazano na listingu 19.5, to zastosowanie metody AggregateException.Handle() i podanie wyrażenia, które należy wykonać dla każdego wyjątku zapisanego w wyjątku AggregateException. Ważną cechą metody Handle() jest to, że jest ona predykatem. Dlatego podany w niej delegat powinien zwracać true, gdy wyjątek zostanie poprawnie obsłużony. Jeśli kod obsługujący któryś z wyjątków zwróci wartość false, metoda Handle() zgłosi nowy wyjątek typu AggregateException, zawierający listę wszystkich wyjątków, których nie udało się obsłużyć.

4.0

Aby sprawdzić stan błędnego zadania bez powodowania ponownego zgłoszenia wyjątku w bieżącym wątku, wystarczy pobrać wartość właściwości Exception danego zadania. To podejście przedstawiono na listingu 19.6. Kod z tego listingu oczekuje na ukończenie błędnej kontynuacji zadania⁹, o którym wiadomo, że zgłosi wyjątek.

Listing 19.6. Wykrywanie nieobsłużonych wyjątków w zadaniu za pomocą metody ContinueWith()

```
using System;
using System.Diagnostics;
using System.Threading.Tasks;

public class Program
{
```

⁹ Wcześniej wyjaśniono, że oczekiwanie na kontynuację błędnego kodu to dziwne rozwiązanie, ponieważ zwykle ta kontynuacja w ogóle nie zostanie uruchomiona. Kod na listingu przedstawiono tylko w celach ilustracyjnych.

```

public static void Main()
{
    bool parentTaskFaulted = false;
    Task task = new Task(() =>
    {
        throw new InvalidOperationException();
    });
    Task continuationTask = task.ContinueWith(
        (antecedentTask) =>
    {
        parentTaskFaulted =
            antecedentTask.IsFaulted;
    }, TaskContinuationOptions.OnlyOnFaulted);
    task.Start();
    continuationTask.Wait();
    Trace.Assert(parentTaskFaulted);
    Trace.Assert(task.IsFaulted);
    task.Exception!.Handle(eachException =>
    {
        Console.WriteLine(
            $"BŁĄD: { eachException.Message }");
        return true;
    });
}
}

```

Zauważ, że w celu pobrania nieobsłużonego wyjątku z pierwotnego zadania używana jest tutu właściwość `Exception` (w ramach dereferencji używany jest operator braku null, ponieważ wiadomo, że wartość jest różna od null). Wynik jest identyczny jak w danych wyjściowych 19.3.

Może się zdarzyć, że wyjątek, który wystąpił w zadaniu, pozostanie niezauważony. Dzieje się tak, gdy (1) wyjątek nie zostanie przechwycony w zadaniu, (2) zakończenie zadania nie zostanie wykryte (wykryć ukończenie zadania można na przykład za pomocą metody `Wait()`, właściwości `Result` lub dostępu do właściwości `Exception`) i (3) błąd w kodzie z wywołaniem `ContinueWith()` nie zostanie wykryty. W takiej sytuacji wyjątek prawdopodobnie w ogóle nie zostanie obsłużony, co skutkuje jego nieobsłużeniem na poziomie procesu. W platformie .NET 4.0 takie błędne zadanie prowadzi do ponownego zgłoszenia wyjątku w wątku finalizatora i zwykle skutkuje zamknięciem procesu. W platformie .NET 4.5 proces nie jest w takiej sytuacji zamknięty, choć środowisko CLR można skonfigurować w taki sposób, by platforma działała jak wcześniej.

W obu wersjach platformy za pomocą zdarzenia `TaskScheduler.UnobservedTaskException` można zarejestrować chcąc odbierania komunikatów o nieobsłużonych wyjątkach z zadań.

■ ZAGADNIENIE DLA ZAAWANSOWANYCH

Radzenie sobie z nieobsłużonymi wyjątkami w wątku

Wcześniej opisano, że nieobsłużony wyjątek w dowolnym wątku domyślnie powoduje zamknięcie aplikacji. Nieobsłużony wyjątek to krytyczny, nieoczekiwany błąd, który może wystąpić na przykład z powodu uszkodzenia niezbędnej struktury danych. Nie wiadomo wtedy, co program zrobił, dlatego najbezpieczniej jest go zamknąć.

W idealnych warunkach programy nigdy nie zgłaszą nieobsługiwanych wyjątków w żadnym wątku. Programy, które zgłaszą takie wyjątki, są błędne. Dlatego należy znaleźć i naprawić usterki przed udostępnieniem oprogramowania klientom. Jednak zamiast zamknąć aplikację od razu po wykryciu nieobsłużonego wyjątku, często warto zachować dane robocze i (lub) zapisać wyjątek na potrzeby zgłoszenia raportów o błędach oraz późniejszego debugowania. To wymaga mechanizmu rejestracji powiadomień o nieobsłużonych wyjątkach.

W platformach Microsoft .NET Framework i .NET Core 2.0 (oraz nowszych) każda domena aplikacji udostępnia taki mechanizm. By wykrywać nieobsłużone wyjątki zgłoszone w domenie aplikacji, trzeba dodać metodę obsługi zdarzenia `UnhandledException`. Jest ono zgłoszane dla wszystkich nieobsłużonych wyjątków z wątków (głównego i roboczych) z domeny aplikacji. Zauważ, że ten mechanizm służy do przesyłania powiadomień. Nie umożliwia on aplikacji przywrócenia poprawnego stanu po zgłoszeniu nieobsłużonego wyjątku i wznowienia pracy. Gdy metody obsługi zdarzeń zostaną uruchomione, aplikacja wyświetli okno dialogowe ze zgłoszeniem błędu od systemu operacyjnego, po czym aplikacja zakończy pracę. W aplikacjach konsolowych szczegółowe informacje o wyjątku pojawią się w konsoli.

Na listingu 19.7 pokazano, jak utworzyć drugi wątek, który zgłasza wyjątek przetwarzany następnie w domenie aplikacji przez metodę obsługi zdarzeń dla nieobsłużonych wyjątków. Na potrzeby przykładu (by zapewnić, że nie wystąpią problemy związane z czasem wykonywania wątków) dodano sztuczne opóźnienia za pomocą metody `Thread.Sleep`. Wyniki przedstawiono w danych wyjściowych 19.4.

Listing 19.7. Rejestrowanie w metodzie chęci otrzymywania powiadomień o nieobsłużonych wyjątkach

```
using System;
using System.Diagnostics;
using System.Threading;

public static class Program
{
    public static Stopwatch _Clock = new Stopwatch();
    public static void Main()
    {
        try
        {
            Clock.Start();
            // Rejestrowanie wywołania zwracającego w celu otrzymywania powiadomień
            // o nieobsłużonych wyjątkach.
            AppDomain.CurrentDomain.UnhandledException +=
                (s, e) =>
            {
                Message("Rozpoczynanie obsługi zdarzeń");
                Delay(4000);
            };
        }

        Thread thread = new Thread(() =>
        {
            Message("Zgłoszanie wyjątku.");
            throw new Exception();
        });
        thread.Start();
        Delay(2000);
    }
}
```

4.0

```

finally
{
    Message("Wykonywanie bloku finally.");
}
}

static void Delay(int i)
{
    Message($"Usypianie na {i} ms");
    Thread.Sleep(i);
    Message("Wzbudzono");
}

static void Message(string text)
{
    Console.WriteLine("{0}:{1:0000}:{2}",
        Thread.CurrentThread.ManagedThreadId,
        _Clock.ElapsedMilliseconds, text);
}
}

```

DANE WYJŚCIOWE 19.4.

```

3:0047:Zgłaszanie wyjątku.
3:0052:Rozpoczynanie obsługi zdarzeń.
3:0055:Usypianie na 4000 ms
1:0058:Usypianie na 2000 ms
1:2059:Wzbudzono
1:2060:Wykonywanie bloku finally.
3:4059:Wzbudzono
Unhandled Exception: System.Exception: Exception of type 'System.
Exception' was thrown.

```

4.0

W danych wyjściowych 19.4 widać, że dla nowego wątku ustawiany jest identyfikator 3, a wątek główny ma identyfikator 1. System operacyjny uruchamia na pewien czas wątek 3. Ten wątek zgłasza nieobsłużony wyjątek, po czym wywoływany jest kod obsługi zdarzeń, który zostaje uśpiony. Wkrótce po tym system operacyjny wykrywa, że można uruchomić wątek 1, jednak ten po wzbudzeniu zostaje natychmiast uśpiony. Wątek 1 zostaje wzbudzony jako pierwszy i wykonuje blok `finally`. Po 2 sekundach wątek 3 zostaje wzbudzony, a nieobsłużony wyjątek powoduje zamknięcie procesu.

Ta sekwencja zdarzeń (uruchomienie metody obsługi zdarzeń i zamknięcie procesu po zakończeniu przez nią pracy) jest typowa, ale nie można jej zagwarantować. Gdy w programie pojawi się nieobsłużony wyjątek, trudno cokolwiek przewidzieć. Program znajduje się wtedy w nieznanym i potencjalnie wysoce niestabilnym stanie. Dlatego jego działanie jest nieprzewidywalne. Po uruchomieniu przykładowego kodu środowisko CLR umożliwia wątkowi głównemu kontynuowanie pracy i wykonuje jego blok `finally`, choć w momencie przekazywania sterowania do tego bloku wiadomo, że inny wątek wykonuje metodę obsługi zdarzeń związaną z nieobsłużonymi wyjątkami z domeny aplikacji.

Aby lepiej to dostrzec, spróbuj zmienić opóźnienia w taki sposób, by wątek główny był usypanny na dłuższy czas niż metoda obsługi zdarzeń. W takiej sytuacji blok `finally` nigdy nie jest wykonywany! Proces zostaje zamknięty z powodu nieobsłużonego wyjątku przed wzbudzeniem wątku 1. W zależności od tego, czy wątek zgłaszający wyjątek został utworzony przez pulę wątków, czy nie, efekt działania programu może być inny. Dlatego najlepiej unikać nieobsłużonych wyjątków niezależnie od tego, czy występują one w wątkach roboczych, czy w wątku głównym.

Jak jest to związane z zadaniami? Co się stanie, jeśli w momencie zamykania aplikacji w systemie wciąż działają niezakończone zadania? W następnym podrozdziale zapoznasz się z procesem anulowania zadań.

Wskazówka

UNIKAJ pisania programów, które w dowolnym wątku zgłaszą nieobsługiwane wyjątki.

ROZWAŻ zarejestrowanie metody obsługi zdarzeń na potrzeby nieobsługiwanych wyjątków. Będzie ona przydatna do debugowania, rejestrowania zdarzeń i awaryjnego zamykania aplikacji.

ANULUJ niezakończone zadania w trakcie zamykania aplikacji, zamiast pozwalać im kontynuować działanie.

Anulowanie zadania

We wcześniejszej części rozdziału wyjaśniono, dlaczego siłowe zamykanie wątku w celu przerwania wykonywanych przez niego zadań to zły pomysł. W bibliotece TPL stosowane jest **anulowanie kooperatywne** (ang. *cooperative cancellation*). Jeśli chcesz mieć możliwość anulowania zadania, musi ono śledzić pracę obiektu typu `CancellationToken` (z przestrzeni nazw `System.Threading`) i stosować odpetywanie cykliczne, by sprawdzać, czy nie pojawiły się żądania anulowania. Na listingu 19.8 pokazano, jak zgłosić żądanie anulowania i jak na nie zareagować. Efekt działania kodu pokazano w danych wyjściowych 19.5.

Listing 19.8. Anulowanie zadania z wykorzystaniem klasy `CancellationToken`

```
using System;
using System.Threading;
using System.Threading.Tasks;
using AddisonWesley.Michaelis.EssentialCSharp.Shared;

public class Program
{
    public static void Main()
    {
        string stars =
            "*".PadRight(Console.WindowWidth-1, '*');
        Console.WriteLine("Wciśnij ENTER, aby zakończyć.");

        CancellationTokenSource cancellationTokenSource=
```

```

    new CancellationTokenSource();
    // Aby zastosować bibliotekę TPL w wersjach starszych niż .NET 4.5,
    // wywołaj metodę Task.Factory.StartNew<string>().
    Task task = Task.Run(
        () =>
        WritePi(cancellationTokenSource.Token),
        cancellationTokenSource.Token);
    // Oczekiwanie na dane wejściowe od użytkownika.
    Console.ReadLine();

    cancellationTokenSource.Cancel();
    Console.WriteLine(stars);
    task.Wait();
    Console.WriteLine();
}

private static void WritePi(
    CancellationToken cancellationToken)
{
    const int batchSize = 1;
    string piSection = string.Empty;
    int i = 0;

    while(!cancellationToken.IsCancellationRequested
        || i == int.MaxValue)
    {
        piSection = PiCalculator.Calculate(
            batchSize, (i++) * batchSize);
        Console.Write(piSection);
    }
}
}

```

DANE WYJŚCIOWE 19.5.

4.0

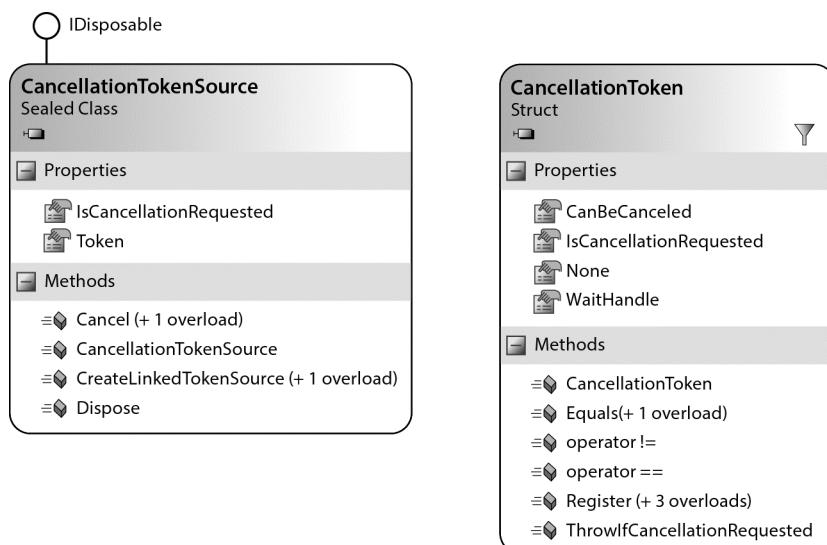
Wciśnij ENTER, aby zakończyć.
3.141592653589793238462643383279502884197169399375105820974944592307816
40628620899862803482534211706798214808651328230664709384460955058223172
5359408128481117450

2

Po uruchomieniu przez platformę zadania wywołanie `Console.Read()` blokuje wątek główny. Zadanie wciąż jednak działa — oblicza kolejne cyfry liczby pi oraz je wyświetla. Gdy użytkownik wciśnie klawisz *Enter*, program wywołuje metodę `CancellationTokenSource.Cancel()`. Na listingu 19.8 wywołanie `task.Cancel()` jest oddzielone od wywołania `task.Wait()`, a program wyświetla między nimi wiersz z gwiazdkami. Ten krok ma pokazać, że po wykryciu tokena anulowania może nastąpić dodatkowa iteracja. Stąd dodatkowa cyfra 2 po gwiazdkach w danych wyjściowych 19.5. Cyfra 2 pojawia się, ponieważ metoda `CancellationTokenSource.Cancel()` nie przerywa siłowo zadania. Zadanie do czasu sprawdzenia tokena wciąż działa, a gdy wykryje, że właściciel tokena żąda anulowania zadania, „uprzejmie” kończy pracę.

Metoda `Cancel()` powoduje ustawienie właściwości `IsCancellationRequested` we wszystkich tokenach anulowania skopiowanych z właściwości `CancellationTokenSource.Token`. Warto zwrócić uwagę na kilka kwestii:

- Dla zadań asynchronicznych używany jest obiekt typu `CancellationToken` (a nie typu `CancellationTokenSource`). Typ `CancellationToken` umożliwia stosowanie odpytywania cyklicznego w celu sprawdzania żądania anulowania. Typ `CancellationTokenSource` udostępnia token i sygnalizuje wystąpienie żądania anulowania (zobacz rysunek 19.3). Dzięki przekazaniu obiektu typu `CancellationToken` zamiast `CancellationTokenSource` nie trzeba się martwić o problemy z synchronizacją wątków w obiekcie typu `CancellationTokenSource`, ponieważ pozostaje on dostępny tylko w pierwotnym wątku.



Rysunek 19.3. Diagramy klas `CancellationTokenSource` i `CancellationToken`

- Typ `CancellationToken` jest strukturą, dlatego jest kopowany przez wartość. Wartość zwrócona przez właściwość `CancellationTokenSource.Token` to kopia tokena. Dlatego obiekt typu `CancellationToken` jest bezpieczny ze względu na wątki — dostęp do niego jest możliwy tylko w metodzie `WritePi()`.

Aby możliwe było monitorowanie wartości właściwości `IsCancellationRequested`, do zadania należy przekazać kopię obiektu typu `CancellationToken` (pobraną z właściwości `CancellationTokenSource.Token`). Na listingu 19.7 okresowo sprawdzana jest wartość właściwości `IsCancellationRequested` parametru typu `CancellationToken` (tu sprawdzanie ma miejsce po obliczeniu każdej cyfry). Jeśli właściwość `IsCancellationRequested` ma wartość `true`, pętla `while` kończy pracę. Inaczej niż w sytuacji zamknięcia wątku, co powoduje zgłoszenie wyjątku w losowym miejscu zadania, tu kod wychodzi z pętli na podstawie standardowego przepływu sterowania. Częste odpytywanie cykliczne dotyczące wspomnianej właściwości gwarantuje, że kod szybko zareaguje na żądania anulowania pracy.

Elementem, na który warto zwrócić uwagę w typie `CancellationToken`, jest przeciążona metoda `Register()`. Za pomocą tej metody można zarejestrować operację wykonywaną za każdym razem, gdy token otrzyma żądanie anulowania. Wywołanie metody `Register()` powoduje więc dodanie delegata działającego jak odbiornik subskrybujący wywołania metody `Cancel()` obiektu typu `CancellationTokenSource`.

Ponieważ w omawianym programie oczekiwane jest anulowanie zadania przed jego ukończeniem, kod z listingu 19.7 nie zgłasza wyjątku `System.Threading.Tasks.TaskCanceledException`. Dlatego właściwość `task.Status` zwraca wartość `TaskStatus.RanToCompletion` i nie informuje o tym, że prace w zadaniu zostały anulowane. W tym przykładzie takie informacje nie są potrzebne. Jednak biblioteka TPL umożliwia informowanie o anulowaniu prac. Jeśli spowodowały one jakieś problemy (na przykład uniemożliwiły zwrócenie prawidłowego wyniku), można poinformować o tym za pomocą zgłoszenia wyjątku `TaskCanceledException` (jest to typ pochodny od `System.OperationCanceledException`). Zamiast zgłaszać ten wyjątek jawnie, można wywołać wygodniejszą w użyciu metodę `ThrowIfCancellationRequested()` obiektu typu `CancellationToken` (jeśli jest on dostępny).

Próba wywołania metody `Wait()` (lub pobrania wartości właściwości `Result`) dla zadania, które zgłosiło wyjątek `TaskCanceledException`, ma te same skutki co po zgłoszeniu w zadaniu dowolnego innego wyjątku. Takie wywołanie prowadzi do zgłoszenia wyjątku `AggregateException`. Ten wyjątek pozwala poinformować, że zadanie mogło nie zostać wykonane do końca. Inaczej niż w zadaniach ukończonych z powodzeniem, gdzie wszystkie prace są wykonywane z sukcesem, w anulowanym zadaniu prace mogą być zakończone tylko częściowo. Dlatego stan nie jest pewny.

W przykładowym kodzie pokazano, że długotrwała operacja zależna od procesora (obliczanie liczby pi prawie w nieskończoność) może monitorować żądania anulowania i reagować na nie. W niektórych sytuacjach można anulować prace bez jawnego pisania kodu do obsługi tej operacji w docelowym zadaniu. Na przykład opisana w rozdziale 21. klasa `Parallel` domyślnie udostępnia taką możliwość.

4.0

Wskazówka

ANULUJ niedokończone zadania, zamiast pozwalać im kontynuować pracę w czasie zamknięcia aplikacji.

Początek
5.0

Task.Run() — skrócona i uproszczona wersja wywołania Task.Factory.StartNew()

W platformie .NET 4.0 tworzenie zadań odbywa się zwykle za pomocą wywołania `Task.Factory.StartNew()`. W platformie .NET 4.5 wprowadzono prostszą składnię w postaci metody `Task.Run()`. Metoda `Task.Factory.StartNew()` ma przeznaczenie podobne do metody `Task.Run()` i można ją stosować w języku C# 4.0 do wywoływanego metod znacznie obciążających procesor, wymagających utworzenia dodatkowego wątku.

W platformie .NET 4.5 domyślnie należy stosować metodę `Task.Run()`, chyba że okaże się niewystarczająca. Jeśli na przykład musisz zarządzać zadaniem za pomocą flag z wyliczenia `TaskCreationOptions`, zamierzysz zastosować konkretny program szeregujący lub ze względu na wydajność chcesz przekazać stan obiektu, rozważ użycie metody `Task.Factory.StartNew()`. Tylko w rzadkich sytuacjach, kiedy musisz oddzielić instrukcje tworzące zadanie i rozpoczęjące jego pracę, powinieneś dodawać zadania za pomocą konstruktora i uruchamiać je przy użyciu wywołania `Start()`.

Na listingu 19.9 pokazano, jak zastosować metodę `Task.Factory.StartNew()`.

Listing 19.9. Używanie metody `Task.Factory.StartNew()`

```
public Task<string> CalculatePiAsync(int digits)
{
    return Task.Factory.StartNew<string>(
        () => CalculatePi(digits));
}

private string CalculatePi(int digits)
{
    // ...
}
```

Koniec
5.0

Długotrwałe zadania

Pula wątków działa zgodnie z założeniami, iż jednostki pracy są zależne od procesora i stosunkowo krótkie. Te założenia skutkują ograniczaniem liczby tworzonych wątków. To zapobiega alokacji zbyt wielu kosztownych zasobów dla wątków i przeciążeniu procesorów (co skutkowałoby podziałem czasu na zbyt małe porcje i zbyt częstym przełączaniem kontekstu).

Co jednak zrobić, jeśli programista wie, że zadanie będzie działać długo i przez długi czas zajmować wątek? W takiej sytuacji programista może powiadomić program szeregujący, że zadanie prawdopodobnie nie zakończy szybko pracy. Ma to dwojakiego efektu. Po pierwsze, stanowi to wskazówkę dla programu szeregującego, że może lepiej będzie utworzyć specjalny wątek na potrzeby danego zadania, zamiast pobierać wątek z puli. Po drugie, jest to informacja dla programu szeregującego, że sensowne może się okazać uruchomienie większej liczby zadań niż liczba procesorów. To prowadzi do częstszeego przełączania wątków, co w omawianej sytuacji jest korzystne. Niepożądane jest, by jedno długotrwałe zadanie zajmowało cały czas procesora i blokowało dostęp do niego krótszym zadaniom. Krótkie zadania będą mogły wykorzystać przydzielone im porcje czasu do wykonania dużej części swojej pracy, a dla długotrwałego zadania stosunkowo niewielkie opóźnienia spowodowane współużytkowaniem procesora z innymi zadaniami będą mało odczuwalne. Aby uzyskać pożądany efekt, zastosuj w wywołaniu `StartNew()` opcję `TaskCreationOptions.LongRunning`, co pokazano na listingu 19.10. Metoda `Task.Run()` nie przyjmuje parametru typu `TaskCreationOptions`.

4.0

Listing 19.10. Wykonywanie długotrwałych zadań w trybie kooperacji

```
using System.Threading.Tasks;  
  
// ...  
Task task = Task.Factory.StartNew(  
    () =>  
        WritePi(cancellationTokenSource.Token),  
        TaskCreationOptions.LongRunning);  
// ...
```

Wskazówka

INFORMUJ fabrykę zadań o tym, że nowo tworzone zadanie prawdopodobnie będzie długo wykonywane. Pozwoli to odpowiednio zarządzać tym zadaniem.

OSZCZĘDΝIE posługuj się opcją `TaskCreationOptions.LongRunning`.

Zadania pozwalają zwalniać zasoby

Zauważ, że typ `Task` zawiera implementację interfejsu `IDisposable`. Jest to konieczne, ponieważ obiekt typu `Task` może zaalokować uchwyt `WaitHandle` i oczekiwać na ukończenie pracy. Ponieważ typ `WaitHandle` obsługuje interfejs `IDisposable`, zgodnie z dobrymi praktykami to samo powinien robić typ `Task`. Zauważ jednak, że we wcześniejszym przykładowym kodzie nie używano wywołań `Dispose()` — ani jawnie, ani niejawnie za pomocą instrukcji `using`. Na wcześniejszych listingach polegano na automatycznych wywołaniach finalizatora typu `WaitHandle` w momencie zamknięcia programu.

4.0

Stosowane wcześniej podejście ma dwa ważne skutki. Po pierwsze, uchwyty istnieją wtedy dłużej i zużywają więcej zasobów, niż jest to konieczne. Po drugie, mechanizm odzyskiwania pamięci jest mniej wydajny, ponieważ finalizowane obiekty są zachowywane do następnego cyklu pracy tego mechanizmu. Jednak gdy używane są obiekty typu `Task`, obie te kwestie nie mają znaczenia (chyba że finalizacji wymaga bardzo duża liczba zadań). Dlatego choć teoretycznie w programach zawsze powinno się usuwać zasoby zadań, nie musisz tego robić — chyba że wskaźniki wydajności tego wymagają, a dodanie potrzebnego kodu jest łatwe (jest tak, gdy wiadomo, że zadania zakończyły pracę i żaden inny kod z nich nie korzysta).

Początek
5.0**Używanie przestrzeni nazw `System.Threading`**

Biblioteka Parallel Extensions jest bardzo przydatna, ponieważ umożliwia zarządzanie wysokopoziomowymi abstrakcjami w postaci zadań, dzięki czemu nie trzeba bezpośrednio korzystać z wątków. Możliwe jednak, że będziesz musiał pracować nad kodem napisanym przed wprowadzeniem technologii TPL i PLINQ (czyli w wersjach platformy .NET starszych niż 4.0). Możliwe też, że wspomniane technologie nie pozwalają bezpośrednio rozwiązać problemu, z którym się borykasz. Wtedy możesz użyć klasy `Thread` i powiązanego API klasy `System.Threading`. Klasa `System.Threading.Thread` reprezentuje punkt sterowania

w programie i zapewnia nakładkę na wątki systemu operacyjnego. Przestrzeń nazw wątków udostępnia też inne zarządzane API do sterowania wątkami.

Jedną z często używanych metod klasy Thread jest `Sleep()`. Jest ona wygodna, jednak warto jej unikać. Metoda `Thread.Sleep()` powoduje uśpienie bieżącego wątku. Jest to informacja dla systemu operacyjnego, aby nie przydzielał danemu wątkowi żadnych porcji czasu, zanim nie upłynie określony okres. Na pozór jest to sensowna technika, jednak jej stosowanie traktowane jest jak „zły zapaszek kodu”, wskazujący na to, że projekt programu mógłby być lepszy. W ogólnym ujęciu usypianie wątku to zła praktyka programistyczna, ponieważ cały sens przydzielania kosztownych zasobów (takich jak wątek) polega na wykonywaniu przy ich użyciu pracy. Nie płaciłbyś przecież pracownikowi za spanie, dlatego nie ponoś też kosztów alokowania wątków tylko po to, by usypiać je na miliony lub miliardy cykli procesora. Istnieje jednak kilka sytuacji, w których usypianie jest uzasadnione.

Po pierwsze uśpienie wątku z czasem oczekiwania do wzbudzenia równym zero to informacja dla systemu operacyjnego: „bieżący wątek uprzejmie rezygnuje z reszty swojego kwantu czasu na rzecz innego wątku, jeśli istnieje inny wątek, który może wykorzystać ten czas”. Później „uprzejmy” wątek jest szeregowany w standardowy sposób, bez dalszych opóźnień. Po drugie, instrukcja `Thread.Sleep()` jest często stosowana w testach w celu zasymulowania, że dany wątek wykonuje operację o dużej latencji. Nie trzeba wtedy niepotrzebnie obciążać procesora wykonywaniem bezcelowych operacji arytmetycznych. Inne zastosowania tej instrukcji w kodzie produkcyjnym należy poddać uważnej analizie, by ustalić, czy nie ma lepszego sposobu na uzyskanie pożądanego efektu.

Następnym typem w przestrzeni nazw `System.Threading` jest `ThreadPool`. Ma ona ograniczać liczbę wątków, ponieważ ich nadmiar mógłby negatywnie wpływać na wydajność systemu. Wątki są kosztownym zasobem, przełączanie kontekstu między wątkami także generuje koszty, a przetwarzanie dwóch zadań w środowisku z symulowaną równoległością i przedziałami czasu jest znacznie wolniejsze niż wykonywanie ich jedno po drugim. Choć pula wątków dobrze spełnia swoje zadanie, nie umożliwia obsługi długich zadań oraz zadań wymagających synchronizacji z wątkiem głównym (lub z innymi wątkami). Takie scenariusze wymagają abstrakcji wyższego poziomu, w których wątki i ich pule są szczegółem implementacji. Ponieważ biblioteka TPL zapewnia właśnie taką abstrakcję, można całkowicie zrezygnować z klasy `ThreadPool` na rzecz API opartych na wspomnianej bibliotece.

Więcej informacji o technikach zarządzania wątkami roboczymi powszechnie stosowanymi w wersjach starszych niż .NET 4.0 znajdziesz w rozdziałach poświęconych wielowątkowości w książce *Essential C# 3.0* (<https://IntelliTect.com/EssentialCSharp>).

4.0

Wskazówki

UNIKAJ wywoływania metody `Thread.Sleep()` w kodzie produkcyjnym.

STOSUJ zadania i powiązane API zamiast klas z przestrzeni nazw `System`.
→ `Threading`, takich jak `Thread` i `ThreadPool`.

Więcej o metodach `Sleep()` z klas `System.Threading.ThreadPool` i `System.Threading.Thread` dowiesz się ze strony <https://IntelliTect.com/legacy-system-threading>.

Podsumowanie

Na początku rozdziału pokrótkę wspomniano o pewnych trudnościach, z którymi programiści stykają się w trakcie pisania programów wielowątkowych. Chodzi tu o problemy z atomowością, zakleszczeniami i sytuacją wyścigu. Wprowadzają one niepewność i powodują błędne działanie kodu w programach wielowątkowych. Dalej opisano technologie TPL (ang. *Task Parallel Library*) i PLINQ (ang. *Parallel LINQ*). Są to nowe API do tworzenia i szeregowania jednostek pracy reprezentowanych przez obiekty typu Task. Zobaczyłeś, że ten typ pozwala uprościć programowanie wielowątkowe dzięki automatycznemu modyfikowaniu programów z wykorzystaniem kontynuacji pozwalających łączyć mniejsze zadania w większe. W rozdziałach 20. i 21. poznasz kolejne wysokopoziomowe abstrakcje, które dodatkowo upraszczają korzystanie ze wzorca TAP.

W rozdziale 22. dowiesz się, jak uniknąć problemów z atomowością dzięki synchronizacji dostępu do współużytkowanych zasobów bez powodowania zakleszczenia.

Koniec
5.0
Koniec
4.0

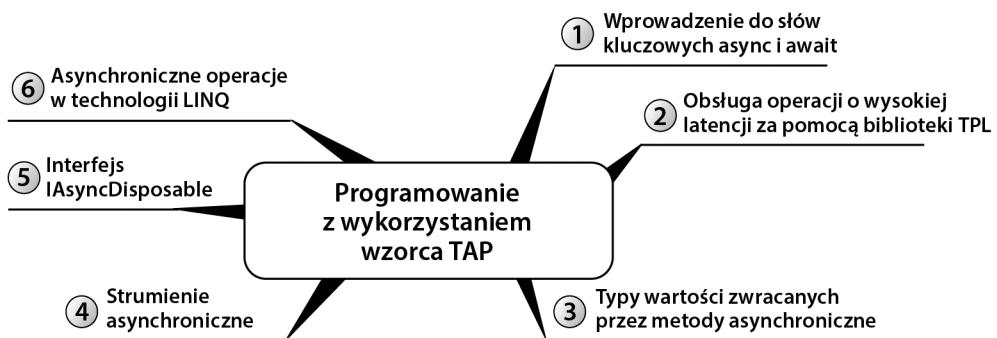
■ 20 ■

Programowanie z wykorzystaniem wzorca TAP

Początek
4.0
Początek
5.0

W ROZDZIALE 19. DOWIEDZIAŁEŚ SIĘ, ŻE zadania są abstrakcją służącą do operowania asynchronicznymi operacjami. Zadania są automatycznie szeregowane w odpowiedniej liczbie wątków, a duże zadania można podzielić na łańcuch mniejszych zadań (podobnie jak rozbudowane programy mogą składać się z wielu małych metod).

Jednak zadania mają też wady. Podstawowa trudność związana jest z tym, że powodują odwrócenie logiki programu „do góry nogami”. Aby to zilustrować, na początku rozdziału przedstawiona jest metoda synchroniczna blokowana przez zależną od wejścia-wyjścia operację o wysokiej latencji — żądanie sieciowe. Dalej zobaczysz, jak zmodyfikować tę metodę za pomocą wprowadzonych w wersji C# 5.0 kontekstowych słów kluczowych `async` i `await`. Pozwalają one znacznie uprościć pisanie i zwiększyć czytelność kodu asynchronicznego.



W końcowej części rozdziału poznasz strumienie asynchroniczne. Jest to wprowadzony w wersji C# 8.0 mechanizm definiowania i stosowania asynchronicznych iteratorów.

Synchroniczne wykonywanie operacji o wysokiej latencji

Na listingu 20.1 program używa klasy `WebClient` do pobrania strony internetowej i wyszukuje liczbę wystąpień określonego tekstu. Wyniki przedstawione są w danych wyjściowych 20.1.

Listing 20.1. Synchroniczne żądanie sieciowe

```
using System;
using System.IO;
using System.Net;

public class Program
{
    public const string DefaultUrl =
        "https://IntelliTect.com";

    public static void Main(string[] args)
    {
        if (args.Length == 0)
        {
            Console.WriteLine("BŁĄD: brak wartości argumentu findText.");
            return;
        }
        string findText = args[0];

        string url = DefaultUrl;
        if (args.Length > 1)
        {
            url = args[1];
            // Ignorowanie dodatkowych parametrów
        }
        Console.Write(
            $"Szukanie tekstu '{findText}' na stronie '{url}'.");
        Console.WriteLine("Pobieranie....");
        using WebClient webClient = new WebClient();
        byte[] downloadData =
            webClient.DownloadData(url);

        Console.WriteLine("Wyszukiwanie....");
        int textOccurrenceCount = CountOccurrences(
            downloadData, findText);

        Console.WriteLine(
            $"{Environment.NewLine}Liczba wystąpień tekstu '{findText}': {textOccurrenceCount}. Strona: '{url}'.");
    }

    private static int CountOccurrences(byte[] downloadData, string findText)
    {
        int textOccurrenceCount = 0;

        using MemoryStream stream = new MemoryStream(downloadData);
        using StreamReader reader = new StreamReader(stream);

        int findIndex = 0;
```

4.0

5.0

```
int length = 0;
do
{
    char[] data = new char[reader.BaseStream.Length];
    length = reader.Read(data);
    for (int i = 0; i < length; i++)
    {
        if (findText[findIndex] == data[i])
        {
            findIndex++;
            if (findIndex == findText.Length)
            {
                // Tekst został znaleziony
                textOccurrenceCount++;
                findIndex = 0;
            }
        }
        else
        {
            findIndex = 0;
        }
    }
}
while (length != 0);

return textOccurrenceCount;
}
```

4.0

5.0

DANE WYJŚCIOWE 20.1.

```
Szukanie tekstu 'IntelliTect' na stronie 'https://IntelliTect.com'.
Pobieranie...
Wyszukiwanie...
Liczba wystąpień tekstu 'IntelliTect': 35. Strona: 'https://IntelliTect.com'.
```

Kod z listingu 20.1 jest dość prosty. Używane są tu standardowe idiomy języka C#. Po określeniu wartości zmiennych url i findText metoda Main() tworzy obiekt typu WebClient i wywołuje synchroniczną metodę DownloadData(), aby pobrać dane. Następnie przekazuje pobrane dane do metody CountOccurrences(), która wczytuje je do strumienia MemoryStream i używa metody Read() z klasy StreamReader do pobrania bloku danych, w którym szuka wartości findText. W tym kodzie używana jest metoda DownloadData() zamiast prostszej metody DownloadString(), co pozwala zademonstrować dodatkowe asynchroniczne wywołania przy odczytce danych ze strumienia na listingach 20.2 i 20.3.

Problem z tym podejściem polega naturalnie na tym, że wywołujący wątek jest blokowany do czasu zakończenia wykonywania operacji wejścia-wyjścia. Oznacza to marnowanie zasobów wątku, który mógłby wykonywać przydatną pracę w trakcie działania operacji. W wątku nie można więc wykonywać innego kodu, na przykład w celu asynchronicznego informowania o postępie prac. Wyświetlanie informacji „Pobieranie...” i „Wyszukiwanie...” ma miejsce *przed* uruchomieniem danej operacji, a nie w *trakcie* jej wykonywania. Tu nie ma to znaczenia, jednak wyobraź sobie, że chcesz równolegle wykonywać dodatkowe zadania lub choćby wyświetlać animowany kursor zajętości.

Asynchroniczne wywoływanie operacji o dużej latencji za pomocą biblioteki TPL

Aby rozwiązać opisany problem, na listingu 20.2 zastosowano podobne podejście, jednak tym razem za pomocą biblioteki TPL zapewniono asynchronicznosć wywołań.

Listing 20.2. Asynchroniczne żądanie sieciowe

```
using System;
using System.IO;
using System.Net;
using System.Threading.Tasks;
using System.Runtime.ExceptionServices;

public class Program
{
    public const string DefaultUrl =
        "https://IntelliTect.com";

    public static void Main(string[] args)
    {
        if (args.Length == 0)
        {
            Console.WriteLine("BŁĄD: nie podano argumentu findText.");
            return;
        }
        string findText = args[0];

        string url = DefaultUrl;
        if (args.Length > 1)
        {
            url = args[1];
            // Ignorowanie pozostałych parametrów.
        }
        Console.Write(
            $"Wyszukiwanie tekstu '{findText}' na stronie '{url}'.");

        using WebClient webClient = new WebClient();
        Console.WriteLine("\nPobieranie...");
        Task task = webClient.DownloadDataTaskAsync(url)
            .ContinueWith(antecedent =>
        {
            byte[] downloadData = antecedent.Result;
            Console.WriteLine("\nWyszukiwanie...");
            return CountOccurrencesAsync(
                downloadData, findText);
        })
            .Unwrap()
            .ContinueWith(antecedent =>
        {
            int textOccurrenceCount = antecedent.Result;
            Console.WriteLine(
                $"{Environment.NewLine}Liczba wystąpień tekstu '{findText}': {textOccurrenceCount}. Strona: '{url}'.");
        });
    }
}
```

4.0

5.0

```
try
{
    while(!task.Wait(100))
    {
        Console.Write(".");
    }
}
catch(AggregateException exception)
{
    exception = exception.Flatten();
    try
    {
        exception.Handle(innerException =>
        {
            // Ponowne zgłoszanie wyjątku zamiast
            // używania warunku if dotyczącego typu.
            ExceptionDispatchInfo.Capture(
                innerException)
            .Throw();
        return true;
    });
}
catch(WebException)
{
    // ...
}
catch(IOException )
{
    // ...
}
catchNotSupportedException )
{
    // ...
}
}
```

4.0

5.0

Dane wyjściowe wyglądają tu prawie tak samo jak w danych wyjściowych 20.1, przy czym po tekście „Pobieranie...” i „Wyszukiwanie...” powinny pojawić się dodatkowe kropki (ich liczba zależy od czasu wykonywania operacji).

Kod z listingu 20.2 w trakcie pobierania strony wyświetla w konsoli tekst „Pobieranie...” z dodatkowymi kropkami. Podobnie dzieje się w trakcie wyświetlania tekstu „Wyszukiwanie...”. Dlatego kod z listingu 20.2 zamiast wyświetlać w konsoli tylko trzy kropki (...), może dodawać je tak długo, jak długo trwa pobieranie pliku i przeszukiwanie tekstu.

Niestety, asynchroniczność jest uzyskiwana kosztem wyższej złożoności. W programie znajdują się instrukcje wykorzystujące bibliotekę TPL, które zmieniają przepływ sterowania. Nie wystarczy po wywoaniu `WebClient.DownloadDataTaskAsync(url)` wywołać instrukcji zliczającej wystąpienia tekstu (asynchronicznej wersji metody `CountOccurrences()`). W asynchronicznej wersji kodu niezbędne są też instrukcje `ContinueWith()`, upraszczające wywołania `Unwrap()` i skomplikowane bloki try-catch do obsługi błędów. Szczegóły znajdziesz w zagadnienu dla zaawansowanych „Skomplikowane żądania asynchroniczne z użyciem biblioteki TPL”. Tu wystarczy stwierdzić, że będziesz wdzięczny za dodanie w C# 5.0 wzorca wykonywania asynchronicznych zadań za pomocą instrukcji `async` i `await`.

ZAGADNIENIE DLA ZAAWANSOWANYCH

Skomplikowane żądania asynchroniczne z użyciem biblioteki TPL

Pierwsza instrukcja `ContinueWith()` określa, jaki kod należy wykonać po metodzie `WebClient.DownloadDataTaskAsync(url)`. Zauważ, że instrukcja `return` w pierwszym wyrażeniu `ContinueWith()` zwraca wynik wywołania `CountOccurrencesAsync(downloadData, findText)`, czyli następny obiekt zadania (typu `Task<int>`). Tak więc typ wartości zwracanej przez pierwszą instrukcję `ContinueWith()` to `Task<Task<byte[]>>`.

Dlatego jeśli pominiesz wywołanie `Unwrap()`, zadanie poprzedzające w drugiej instrukcji `ContinueWith()` będzie typu `Task<Task<byte[]>>`, co już samo w sobie wskazuje na złożoność rozwiązania. Dlatego wtedy właściwość `Result` trzeba pobrać dwukrotnie — raz bezpośrednio z obiektu antecedent i raz w wyrażeniu `Task<byte[]>.Result` zwróconym przez wywołanie `antecedent.Result` (ta druga instrukcja blokuje dalszy kod do czasu wykonania operacji `DownloadDataTaskAsync()`). Aby uniknąć tworzenia struktury typu `Task<Task<TResult>>`, wywołanie `ContinueWith()` poprzedzono wywołaniem `Unwrap()`, co pozwala pominąć zewnętrzny typ `Task` oraz zapewnić poprawną obsługę błędów i żądań anulowania.

4.0

5.0

Jednak złożoność nie ogranicza się do typu `Task` i instrukcji `ContinueWith()`. Obsługa błędów dodaje zupełnie nowy wymiar złożoności. Wcześniej wspomniano, że biblioteka TPL zwykle zgłasza wyjątek `AggregateException`, ponieważ asynchroniczna operacja może spowodować wiele wyjątków. Jednak ponieważ w blokach `ContinueWith()` pobierana jest wartość właściwości `Result`, także wątek roboczy może zgłosić wyjątek `AggregateException`.

We wcześniejszej części rozdziału dowiedziałeś się, że istnieje kilka sposobów na obsługę takich wyjątków:

1. Można za pomocą wywołania `ContinueWith()` dodać zadania kontynuacyjne dla wszystkich metod `*Async`, które zwracają zadanie. Jednak to uniemożliwia stosowanie płynnego interfejsu API, w którym instrukcje `ContinueWith()` są łączone w łańcuch. Ponadto kod obsługi błędów trzeba wtedy umieścić w kodzie przepływu sterowania (zamiast w blokach obsługi wyjątków).
2. Można umieścić ciało każdego delegata w blokach `try-catch`, tak by nieobsłużone wyjątki nie były przekazywane poza zadanie. Jednak także to rozwiązanie nie jest idealne. Po pierwsze, niektóre wyjątki (na przykład spowodowane wywołaniem `antecedent.Result`) prowadzą do zgłoszenia wyjątku `AggregateException`, co wymaga pobrania wyjątków z właściwości `InnerException` w celu obsłużenia każdego z nich. Po pobraniu wyjątków można je ponownie zgłosić, by przechwycić wyjątek konkretnego typu, lub sprawdzać typy wyjątków niezależnie od innych bloków `catch` (nawet tych przeznaczonych dla wyjątków danego typu). Po drugie, dla każdego ciała delegata trzeba utworzyć odrębne bloki `try-catch`, nawet jeśli wyjątki zgłaszane w różnych blokach są tego samego typu. Po trzecie, wywołanie `task.Wait()` w metodzie `Main` może spowodować zgłoszenie wyjątku, ponieważ wywołanie metody `webClient.DownloadDataTaskAsync()` lub `CountOccurrencesAsync()` może zgłosić wyjątek, a wykonywanego przez nią kodu nie da się umieścić w bloku `try-catch`. Nie da się więc wyeliminować w metodzie `Main` bloku `try-catch` wokół wywołania `task.Wait()`.

3. Na listingu 20.2 kod nie przechwytuje wyjątków zgłaszanych przez metodę `DownloadDataTaskAsync()`; zamiast tego polega całkowicie na bloku `try-catch` wokół wywołania `task.Wait()` w metodzie `Main`. Ponieważ wiadomo, że zgłaszaną wyjątek będzie typu `AggregateException`, można utworzyć blok `catch` tylko dla tego typu. W bloku `catch` można obsługiwać wyjątek za pomocą wywołania `AggregateException.Handle()` i zgłaszać każdy wyjątek przy użyciu obiektu `ExceptionDispatchInfo`. Zapobiega to utracie pierwotnych informacji ze stosu. Zgłasiane wyjątki są później przechwytywane w odpowiednich blokach obsługi i we właściwy sposób przetwarzane. Zauważ jednak, że przed obsługą wyjątków z właściwością `InnerException` obiektu `AggregateException` najpierw wywoływana jest metoda `AggregateException.Flatten()`. Ten krok rozwiązuje problem związany z tym, że w wyjątku `AggregateException` mogą się znajdować inne wyjątki tego typu (i tak dalej). Wywołanie `Flatten()` gwarantuje, że wszystkie wyjątki są przenoszone na pierwszy poziom, a wszystkie wewnętrzne wyjątki `AggregateException` zostają usunięte.

Rozwiązanie nr 3 (zastosowane na listingu 20.2) jest prawdopodobnie najlepsze, ponieważ w większości miejsc pozwala oddzielić obsługę wyjątków od przepływu sterowania. Nie eliminuje to w pełni złożoności związanej z obsługą błędów, pozwala jednak zminimalizować sytuacje, w których obsługa wyjątków miesza się ze zwykłym przepływem sterowania.

Wersja asynchroniczna z listingu 20.2 ma prawie ten sam logiczny przepływ sterowania co synchroniczny kod z listingu 20.1. Obie wersje próbują pobrać zasób z serwera, a jeśli zakończy się to powodzeniem, zwracają wynik. Gdy pobieranie kończy się niepowodzeniem, kod sprawdza typ wyjątku, by ustalić właściwy tok postępowania. Wyraźnie widać jednak, że wersja asynchroniczna z listingu 20.2 jest dużo mniej czytelna, mniej zrozumiała i trudniejsza do modyfikowania niż analogiczny synchroniczny kod z listingu 20.1. W wersji asynchronicznej (inaczej niż w synchronicznej, gdzie używane są standardowe instrukcje związane z przepływem sterowania) konieczne było utworzenie wielu wyrażeń lambda, by opisać kontynuacje w postaci delegatów.

A omawiany program jest stosunkowo prosty (tym bardziej że pominięta została implementacja metody `CountOccurrencesAsync()`!). Wyobraź sobie, jak kod asynchroniczny będzie wyglądał, jeśli synchroniczny kod w pętli ma trzykrotnie ponawiać próbę wykonania operacji po jej niepowodzeniu, próbować kontaktować się z wieloma różnymi serwerami, pobierać kilka zasobów zamiast jednego lub wykonywać wszystkie te czynności. Dodanie takich mechanizmów do wersji synchronicznej jest proste, nie jest jednak oczywiste, jak zrobić to w wersji asynchronicznej. Przekształcanie metod synchronicznych w asynchroniczne w wyniku jawnego dodania kontynuacji dla każdego zadania szybko prowadzi do bardzo dużych komplikacji (choć przepływ sterowania z synchronicznymi kontynuacjami wydaje się bardzo prosty).

Początek
7.0

Asynchroniczność oparta na zadaniach oraz instrukcjach `async` i `await`

Aby rozwiązać problem złożoności, na listingu 20.3 pokazany jest asynchroniczny kod, w którym jednak używane są zadania i wprowadzony w wersji C# 5.0 mechanizm `async/await`. Ten mechanizm sprawia, że skomplikowane operacje może wykonywać kompilator, co pozwala programistom skupić się na logice biznesowej. Zamiast tworzenia łańcuchów instrukcji `ContinueWith()`, pobierania wyników za pomocą wywołań `antecedent.Result`, używanego wywołań `Unwrap()`, pisania skomplikowanej obsługi błędów itd. mechanizm `async/await` pozwala dodać do kodu prostą składnię, która informuje kompilator, że powinien zadbać o skomplikowane aspekty programu. Ponadto gdy zadanie zakończy pracę i należy wykonać dodatkowy kod, kompilator automatycznie uruchamia w odpowiednim wątku pozostały kod. Standardowo nie możesz na przykład używać dwóch wątków do interakcji z systemem z jednowątkowym interfejsem użytkownika. Mechanizm `async/await` rozwiązuje ten problem (więcej dowiesz się z podrozdziału o używaniu mechanizmu `async/await` w systemie Windows).

4.0

5.0

Składnia `async/await` nakazuje kompilatorowi reorganizację kodu (jest on stosunkowo prosty) na etapie kompilacji i zadbanie o wiele skomplikowanych kwestii, którymi w innym scenariuszu programista musiałby zająć się samodzielnie (zobacz listng 20.3).

Listing 20.3. Asynchroniczne wywołania o wysokiej latencji oparte na wzorcu TAP

```
using System;
using System.IO;
using System.Net;
using System.Threading.Tasks;

public class Program
{
    public const string DefaultUrl =
        "https://IntelliTect.com";

    public static async Task Main(string[] args)
    {
        if (args.Length == 0)
        {
            Console.WriteLine("BŁĄD: nie podano argumentu findText.");
            return;
        }
        string findText = args[0];

        string url = DefaultUrl;
        if (args.Length > 1)
        {
            url = args[1];
            // Ignorowanie dodatkowych parametrów.
        }
        Console.Write(
            $"Wyszukiwanie tekstu '{findText}' na stronie '{url}'.");

        using WebClient webClient = new WebClient();
        Task<byte[]> taskDownload =
            webClient.DownloadDataTaskAsync(url);
```

```
Console.WriteLine("Pobieranie...");  
byte[] downloadData = await taskDownload;  
Task<int> taskSearch = CountOccurrencesAsync(  
    downloadData, findText);  
  
Console.WriteLine("Wyszukiwanie...");  
int textOccurrenceCount = await taskSearch;  
  
Console.WriteLine(  
    $"{Environment.NewLine}Liczba wystąpień tekstu '{findText}': {  
        textOccurrenceCount}. Strona: '{url}'.");  
}  
  
private static async Task<int> CountOccurrencesAsync(  
    byte[] downloadData, string findText)  
{  
    int textOccurrenceCount = 0;  
  
    using MemoryStream stream = new MemoryStream(downloadData);  
    using StreamReader reader = new StreamReader(stream);  
  
    int findIndex = 0;  
    int length = 0;  
    do  
    {  
        char[] data = new char[reader.BaseStream.Length];  
        length = await reader.ReadAsync(data);  
        for (int i = 0; i < length; i++)  
        {  
            if (findText[findIndex] == data[i])  
            {  
                findIndex++;  
                if (findIndex == findText.Length)  
                {  
                    // Tekst został znaleziony.  
                    textOccurrenceCount++;  
                    findIndex = 0;  
                }  
            }  
        }  
        else  
        {  
            findIndex = 0;  
        }  
    }  
    while (length != 0);  
  
    return textOccurrenceCount;  
}
```

Zwróć uwagę na to, że różnice między kodem z listingów 20.1 i 20.3 są dość niewielkie. Ponadto dane wyjściowe powinny wyglądać teraz prawie tak samo jak na listingu 20.2 (w poszczególnych wywołaniach zmieniać może się liczba kropek). Jest to jedna z najważniejszych cech mechanizmu `async/await` — kod wymaga tylko niewielkich zmian w porównaniu z wersją synchroniczną.

7.0
Początek
8.0Koniec
8.0
4.0

5.0

Aby zrozumieć ten wzorzec, skup się najpierw na metodzie `CountOccurrencesAsync()` i różnicach względem listingu 20.1. Po pierwsze zmieniono sygnaturę metody `CountOccurrencesAsync()`, używając modyfikatora w postaci nowego kontekstowego słowa kluczowego `async`. Po drugie metoda zwraca wartość typu `Task<int>`, a nie typu `int`. Każda metoda opatrzona słowem kluczowym `async` musi zwracać wartość **typu odpowiedniego dla metod asynchronicznych**. Od wersji C# 7.0 dozwolone są tu typy `void`, `Task`, `Task<T>`, `ValueTask<T>`, a od wersji C# 8.0 dodatkowo typy `IAsyncEnumerable<T>` i `IAsyncEnumerator<T>`¹. Tu metoda nie zwraca żadnych danych, ale przydatna jest możliwość zwracania informacji o asynchronicznych działaniach do jednostki wywołującej, dlatego metoda `CountOccurrencesAsync()` zwraca wartość typu `Task`. Dzięki zwracanemu zadaniu jednostka wywołującą ma dostęp do stanu asynchronicznego wywołania, a także — po jego wykonaniu — wyniku (jest nim wartość typu `int`). Nazwa metody ma zwyczajowo dodawany przyrostek `Async`. To informuje, że można ją wywoływać z wykorzystaniem operatora `await`. Ponadto wszędzie tam, gdzie kod ma asynchronicznie oczekiwany na zadanie z asynchronicznego wywołania w metodzie `CountOccurrencesAsync()`, używany jest operator `await`. Tu dotyczy to tylko wywołania `reader.ReadAsync()`. Metoda `StreamReader.ReadAsync()` jest — podobnie jak `CountOccurrencesAsync()` — metodą asynchroniczną działającą w analogiczny sposób jak jej standar-dowy odpowiednik.

W metodzie `Main()` występują różnice podobnego rodzaju. Wywołanie nowej metody `CountOccurrencesAsync()` odbywa się z użyciem kontekstowego słowa kluczowego `await`. W ten sposób można pisać kod z pominięciem skomplikowanych aspektów bezpośredniego zarządzania zadaniami. Gdy stosujesz wywołanie z operatorem `await`, możesz przypisać wynik do zmiennej typu `int` (a nie `Task<int>`) lub — jeśli wywoływana metoda zwraca zadanie — przyjąć, że nie ma zwracanej wartości.

Przypomnij sobie sygnaturę metody `CountOccurrencesAsync()`:

```
private static async Task<int> CountOccurrencesAsync(
    byte[] downloadData, string findText)
```

Choć zwracana wartość jest typu `Task<int>`, wywołanie `await CountOccurrencesAsync()` zwraca wartość typu `int`:

```
int textOccurrenceCount = await CountOccurrencesAsync(
    downloadData, findText);
```

Ten przykład ilustruje „magię” związaną z wywołaniem `await` — wypakowuje ono wynik z zadania i zwraca go.

Jeśli chcesz wykonywać kod w trakcie działania asynchronicznej operacji, możesz odro-czyć wywołanie `await` do czasu zakończenia równoległego zadania (wyświetlania tekstu w konsoli). Kod przed wywołaniem `CountOccurrencesAsync()` uruchamia w taki sposób wywołanie `webClient.DownloadDataTaskAsync(url)`. Zamiast od razu przypisywać do zmiennej wartość typu `byte[]` i stosować operator `await`, wywołanie `await` jest odraczane do czasu równoległego zapisania kropek w konsoli:

¹ Można też zwrócić obiekt dowolnego typu z implementacją metody `GetAwaiter()`. Zobacz zagad-nienie dla zaawansowanych „Inne typy zwracanych wartości odpowiednie dla metod asynchronicznych” w dalszej części rozdziału.

```
using WebClient webClient = new WebClient();
Task<byte[]> taskDownload =
    webClient.DownloadDataTaskAsync(url);
while (!taskSearch.Wait(100)){Console.Write(".");
byte[] downloadData = await taskDownload;
```

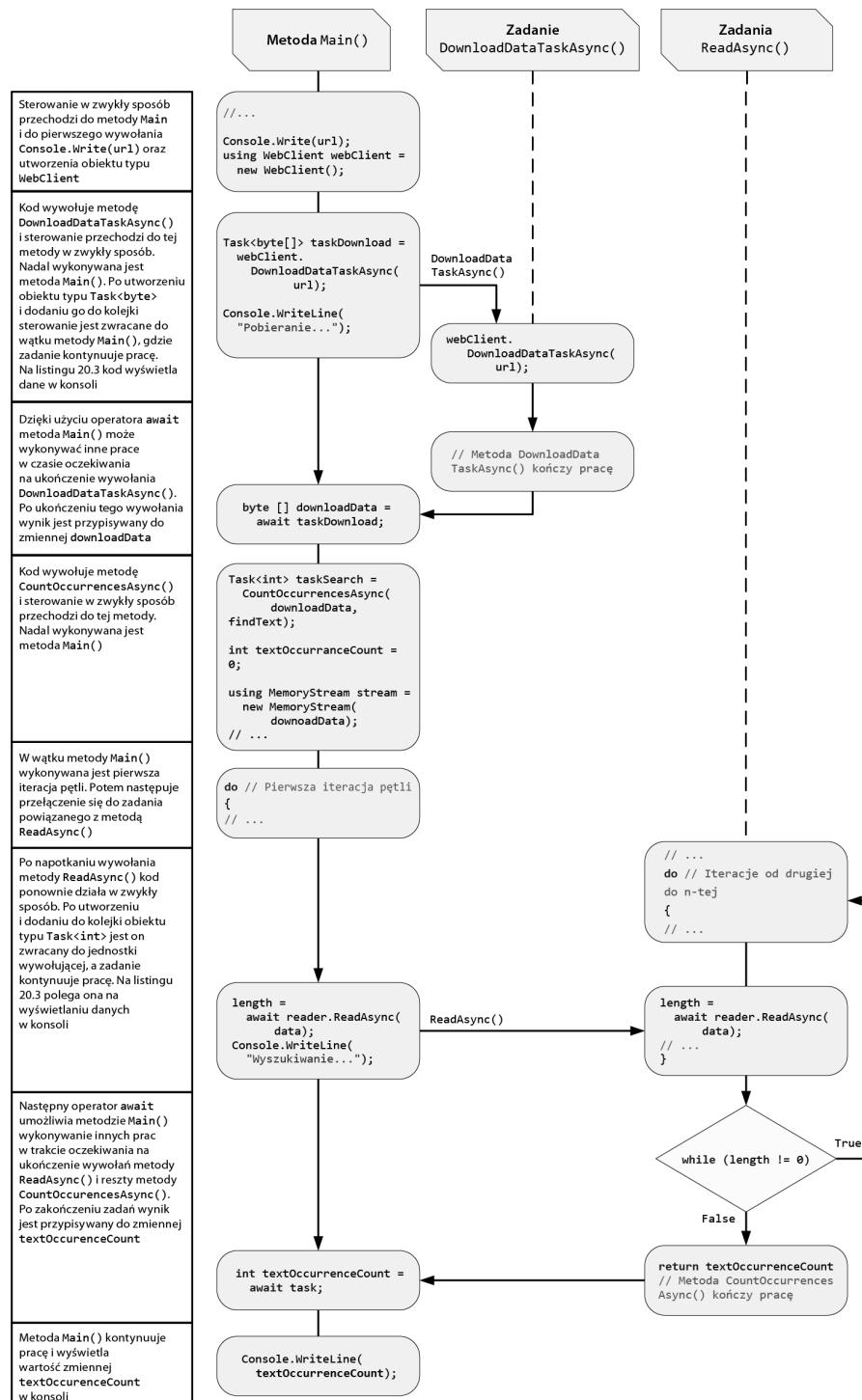
Oprócz wypakowania wartości z obiektu typu Task operator await informuje kompilator języka C#, aby wygenerował kod wykonujący w odpowiednim wątku instrukcje po wywołaniu await. Jest to ważna korzyść, pozwalająca uniknąć trudnych do wykrycia usterek.

Aby pomóc Ci lepiej zrozumieć przepływ sterowania, na rysunku 20.1 opisano w odrębnych kolumnach wszystkie zadania wraz z wykonywanym w nich kodem.

Rysunek 20.1 pomaga wyeliminować kilka ważnych błędnych założeń. Oto one:

- **Błędne przekonanie nr 1. Metoda opatrzona słowem kluczowym async jest automatycznie wykonywana w wątku roboczym po jej wywołaniu.** To nieprawda. Metoda jest wykonywana w normalny sposób, w wątku, który ją wywołał. Jeśli kod nie oczekuje na nieukończone zadania (które umożliwiają oczekiwanie), metoda kończy pracę synchronicznie w tym samym wątku. To w kodzie danej metody należy uruchamiać wszelkie asynchronousne prace. Samo zastosowanie słowa kluczowego async nie zmienia miejsca wykonywania kodu metody. Ponadto z perspektywy jednostki wywołującej w uruchomieniu metody z modyfikatorem async nie ma nic niezwykłego. Na przykład w metodzie Main() wartość zwracana przez metodę CountOccurrencesAsync() jest typu Task<int>, tak jak w wersji synchronicznej. Następnie można oczekwać na zadanie.
- **Błędne przekonanie nr 2. Słowo kluczowe await powoduje zablokowanie bieżącego wątku do momentu ukończenia zadania, którego to słowo dotyczy.** To także nie jest prawdą. Jeśli chcesz, by bieżący wątek został zablokowany do czasu ukończenia zadania, wywołaj metodę Wait() w sposób opisany w rozdziale 19. (uważaj przy tym, by nie spowodować zakleszczenia). Słowo kluczowe await powoduje przetworzenie podanego po nim wyrażenia (zwykle typu Task, Task<T> lub ValueTask<T>), dodanie kontynuacji do wynikowego zadania i natychmiastowe zwrócenie sterowania do jednostki wywołującej. Utworzenie zadania rozpoczyna wykonywanie asynchronousnej pracy. Słowo kluczowe await oznacza, że programista chce, by jednostka wywołująca daną metodę kontynuowała działanie w bieżącym wątku w trakcie wykonywania pracy asynchronousnej. W pewnym momencie po ukończeniu pracy asynchronousnej wykonywanie programu zostanie wznowione w punkcie sterowania po wyrażeniu ze słowem kluczowym await.

Są dwa podstawowe powody, dla których utworzono słowo kluczowe async. Po pierwsze, pozwala ono poinformować czytelnika kodu, że następująca po tym słowie metoda zostanie przekształcona przez kompilator. Po drugie, informuje kompilator o tym, że wystąpienia kontekstowego słowa kluczowego await w metodzie należy traktować jak instrukcje związane z asynchronousnym przepływem sterowania, a nie jak zwykły identyfikator.



Rysunek 20.1. Przepływ sterowania w każdym zadaniu

Od wersji C# 7.1 można tworzyć asynchroniczne metody Main. Dlatego sygnaturą metody Main z listingu 20.3 może być `private static async Task Main(string[] args)`. Dzięki temu użytkownik operatora await może wywoływać metody asynchroniczne. Bez asynchronicznej metody Main() konieczne byłoby bezpośrednie używanie dozwolonych typów wartości zwracanych przez metody asynchroniczne, a także jawne oczekiwanie na ukończenie zadania przed zamknięciem programu, aby uniknąć nieoczekiwanego działania kodu.

Przy okazji warto wspomnieć, że w wersjach C# 5.0 i 6.0 obowiązywało ograniczenie, zgodnie z którym instrukcja await nie mogła występować w związanych z obsługą wyjątków blokach catch lub finally. Od wersji C# 7.0 to ograniczenie jest zniesione. Jest to pomocne usprawnienie, ponieważ możesz chcieć rejestrować wyjątek z zewnętrznego bloku obsługi wyjątków ze stosu wywołań, a rejestrowanie to stosunkowo kosztowna operacja, dlatego warto ją wykonywać z użyciem asynchronicznej instrukcji await.

Dodanie możliwości zwracania typu ValueTask<T> w metodach asynchronicznych

4.0

5.0

Metody asynchroniczne są używane dla długich operacji z wysokim opóźnieniem. Ponieważ zwracane są obiekty typu Task lub Task<T>, oczywiste jest to, że zawsze trzeba otrzymać obiekt tego rodzaju. Inna możliwość, zwracanie wartości null, zmuszałaby jednostki wywołujące, by zawsze sprawdzać taką wartość przed wywołaniem metody. W kontekście użyteczności taki interfejs API byłby niedorzeczny i frustrujący. Zwykle koszt tworzenia obiektu typu Task lub Task<T> jest niewielki w porównaniu z kosztami wykonywania długich operacji o wysokim opóźnieniu.

Co się jednak stanie, jeśli operację można skrócić i natychmiast zwrócić wynik? Zastanów się na przykład nad kompresją bufora. Gdy ilość danych jest duża, asynchroniczne wykonywanie tej operacji ma sens. Jeżeli jednak dane mają zerową długość, operacja może natychmiast zwrócić sterowanie, a otrzymywanie (z pamięci podręcznej lub w wyniku utworzenia nowej instancji) obiektu typu Task lub Task<T> nie ma sensu, ponieważ gdy operacja jestkończona natychmiast, nie trzeba tworzyć zadania. Potrzebny jest obiekt przypominający zadanie, który obsługuje operacje asynchroniczne, ale nie wymaga kosztów tworzenia kompletnych obiektów typu Task lub Task<T>, gdy nie są one potrzebne. W momencie wprowadzenia w C# 5.0 słów kluczowych `async` i `await` nie istniało takie rozwiązanie. Jednak w C# 7.0 dodano obsługę wszystkich typów spełniających określony warunek — udostępnianie metody `GetAwaiter`, co zostało opisane w zagadnieniu dla zaawansowanych „Inne typy zwracanych wartości odpowiednie dla metod asynchronicznych”.

Platformy .NET zgodne z C# 7.0 obejmują typ bezpośredni `ValueTask<T>`, który pozwala uniknąć kosztów, gdy długą operację może skrócić, a w innych scenariuszach umożliwia obsługę wszystkich mechanizmów zadań za pomocą typu Task. Na listingu 20.4 pokazano kod do archiwizowania plików, gdzie używany jest typ `ValueTask<T>`, jeśli kompresję można skrócić.

Listing 20.4. Zwracanie wartości typu ValueTask<T> w metodach asynchronicznych

```
using System.IO;
using System.Text;
using System.Threading.Tasks;

public static class Program
{
    private static async ValueTask<byte[]> CompressAsync(byte[] buffer)
    {
        if (buffer.Length == 0)
        {
            return buffer;
        }
        using MemoryStream memoryStream = new MemoryStream();
        using System.IO.Compression.GZipStream gZipStream =
            new System.IO.Compression.GZipStream(
                memoryStream,
                System.IO.Compression.CompressionMode.Compress);

        await gZipStream.WriteAsync(buffer, 0, buffer.Length);
        return memoryStream.ToArray();
    }
    // ...
}
```

4.0

5.0

Warto zauważyć, że choć metoda asynchroniczna taka jak `GZipStream.WriteAsync()` może zwracać wartość typu `Task<T>`, wywołanie `await` będzie działać także dla metod zwracających wartość typu `ValueTask<T>`. Na przykład na listingu 20.4 zmiana zwracanej wartości z `ValueTask<T>` na `Task<T>` nie wymaga innych modyfikacji w kodzie.

Kiedy używać typu `ValueTask<T>`, a kiedy typów `Task` lub `Task<T>`? Jeśli dana operacja nie zwraca wartości, użąd typu `Task` (nie istnieje niegeneryczna wersja typu `ValueTask<T>`, ponieważ nie miałaby żadnych zalet). Typ `Task<T>` jest preferowany także wtedy, gdy prawdopodobne jest asynchroniczne zakończenie operacji lub gdy da się zapisywać typowe wyniki zadań w pamięci podręcznej. Zwykle zwracanie wartości typu `ValueTask<bool>` zamiast `Task<bool>` nie daje korzyści, ponieważ można łatwo zapisać w pamięci podręcznej obiekty `Task<bool>` dla wartości `true` i `false`. Co więcej, infrastruktura wywołania `async` robi to automatycznie. Oznacza to, że gdy asynchroniczna metoda zwracająca wartość typu `Task<bool>` kończy pracę synchronicznie, i tak zwracany jest wynik typu `Task<bool>` z pamięci podręcznej. Jeżeli jednak prawdopodobne jest synchroniczne zakończenie pracy i niepraktyczne jest zapisywanie w pamięci podręcznej wszystkich często zwracanych wartości, sensowne może być użycie typu `ValueTask<T>`.

ZAGADNIENIE DLA POCZĄTKUJĄCYCH**Typy wartości zwracanych przez metody asynchroniczne**

Wyrażenie podane po słowie kluczowym `await` zwykle zwraca wartość typu `Task`, `Task<T>` lub `ValueTask<T>`. Jeśli chodzi o składnię, instrukcja `await` wywołana dla obiektu typu `Task` to odpowiednik wyrażenia zwracającego `void`. Ponieważ kompilator nie wie, czy dane zadanie

w ogóle zwraca wynik (a tym bardziej jakiego jest on typu), takie wyrażenie jest traktowane tak samo jak wywołanie metody zwracającej void. Dlatego można je stosować tylko jako instrukcję. Na listingu 20.5 pokazano kilka wyrażeń z modyfikatorem await używanych jako instrukcje.

Listing 20.5. Wyrażenie z modyfikatorem await może być używane jako instrukcja

```
async Task<int> DoStuffAsync()
{
    await DoSomethingAsync();
    await DoSomethingElseAsync();
    return await GetAnIntegerAsync() + 1;
}
```

Tu przyjęto założenie, że pierwsza metoda zwraca obiekt typu Task, a nie Task<T> lub ValueTask<T>. Ponieważ pierwsze dwa zadania nie zwracają wyniku, oczekiwanie na nie prowadzi do otrzymania wartości. Dlatego te wyrażenia trzeba podać jako instrukcje. Trzecie zadanie jest prawdopodobnie typu Task<int>, a zwróconą przez nie wartość można wykorzystać w obliczeniach wartości zadania zwracanego przez metodę DoStuffAsync().

Strumienie asynchroniczne

W wersji C# 8.0 wprowadzono możliwość programowania strumieni asynchronicznych. Pozwalają one stosować wzorzec programowania asynchronicznego razem z iteratormi. W rozdziale 15. dowiedziałeś się, że w C# kolekcje wymagają implementowania interfejsów I Enumerable<T> i I Enumerator<T>. Ten pierwszy zawiera jedną funkcję GetEnumerator<T>(), która zwraca obiekt typu I Enumerator<T> umożliwiający iterację. Gdy tworzysz iterator oparty na wywołaniu yield return, metoda musi zwracać obiekt typu I Enumerable<T> lub I Enumerator<T>. Z kolei typy zwracanych wartości odpowiednie dla metod asynchronicznych muszą udostępniać metodę GetAwaiter()², tak jak robią to typy Task, Task<T> i ValueTask<T>. Problem polega więc na tym, że nie można otrzymać jednocześnie metody async i iteratora. Na przykład gdy wywołujesz metodę async w trakcie używania iteratora kolekcji, nie możesz zwrócić wyników do funkcji wywołującej przed ukończeniem wszystkich oczekiwanych iteracji.

Aby rozwiązać ten problem, w C# 8.0 dodane zostały strumienie asynchroniczne. Ten mechanizm został zaprojektowany po to, aby umożliwiać asynchroniczne iterowanie oraz tworzenie asynchronicznych kolekcji i metod z wywołaniami yield return zwracającymi obiekty z rodziny I Enumerable.

Wyobraź sobie szyfrowanie danych w metodzie asynchronicznej EncryptFilesAsync(), gdy dany jest katalog (domyślnie używany jest katalog bieżący). Kod przedstawiony jest na listingu 20.6.

Koniec
7.1
4.0
5.0
Początek
8.0

² Dopuszczalny jest też „typ” void.

Listing 20.6. Strumienie asynchroniczne

```
using System.IO;
using System.Linq;
using System.Threading.Tasks;
using System.Collections.Generic;
using System.Threading;
using System.Runtime.CompilerServices;
using AddisonWesley.Michaelis.EssentialCSharp.Shared;

public static class Program
{
    public static async void Main(params string[] args)
    {
        string directoryPath = Directory.GetCurrentDirectory();
        const string searchPattern = "*";
        // ...
        using Cryptographer cryptographer = new Cryptographer();

        IEnumerable<string> files = Directory.EnumerateFiles(
            directoryPath, searchPattern);
        // Tworzenie tokenu anulowania, co umożliwia
        // anulowanie pracy, gdy operacja trwa dłużej niż minutę.
        using CancellationTokenSource cancellationTokenSource =
            new CancellationTokenSource(1000*60);

        await foreach ((string fileName, string encryptedFileName)
            in EncryptFilesAsync(files, cryptographer)
            .Zip(files.ToAsyncEnumerable()
            .WithCancellation(cancellationTokenSource.Token))
        {
            Console.WriteLine($"{fileName}=>{encryptedFileName}");
        }
    }

    public static async IAsyncEnumerable<string> EncryptFilesAsync(
        IEnumerable<string> files, Cryptographer cryptographer,
        [EnumeratorCancellation] CancellationToken cancellationToken = default)
    {
        foreach (string fileName in files)
        {
            yield return await EncryptFileAsync(fileName, cryptographer);
            cancellationToken.ThrowIfCancellationRequested();
        }
    }

    private static async Task<string> EncryptFileAsync(
        string fileName, Cryptographer cryptographer)
    {
        string encryptedFileName = $"{fileName}.encrypt";
        await using FileStream outputFileStream =
            new FileStream(encryptedFileName, FileMode.Create);

        string data = await File.ReadAllTextAsync(fileName);
        await cryptographer.EncryptAsync(data, outputFileStream);
        return encryptedFileName;
    }
}
```

4.0

5.0

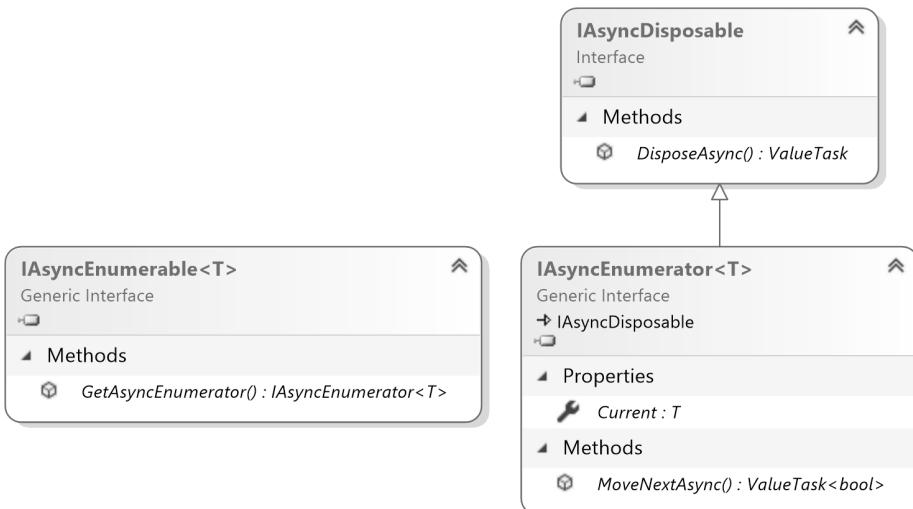
8.0

Listing 20.6 rozpoczyna się metodą Main(), w której używana jest wprowadzona w C# 8.0 instrukcja `async foreach` iteracyjnie przetwarzająca wywołanie metody asynchronicznej `EncryptFilesAsync()` (omówienie wywołania `WithCancellation()` znajdziesz dalej). Metoda `EncryptFilesAsync()` iteracyjnie pobiera każdy plik z pętli `foreach`. W pętli `foreach` znajdują się tu dwa wywołania metod asynchronicznych. Pierwsza z nich to `File.ReadAllTextAsync()`, która wczytuje całą zawartość pliku. Gdy dane są już dostępne w pamięci, kod wywołuje metodę `EncryptAsync()`, która szyfruje dane przed zwróceniem zaszyfrowanego pliku w instrukcji `yield return`. Ta metoda ilustruje więc potrzebę udostępnienia asynchronicznego iteratora jednostce wywołującej. Aby było to możliwe, trzeba opatrzyć metodę `EncryptFilesAsync()` słowem `async` i zwracać wartości typu `IAsyncEnumerable<T>` (tu jako `T` używany jest typ `string`).

Gdy dostępna jest metoda zwracająca wartości typu `IAsyncEnumerable<T>`, można pobić ją za pomocą instrukcji `await foreach`, tak jak w metodzie Main na listingu 20.6. Tak więc ten kod zarówno generuje, jak i pobiera strumień asynchroniczny.

W sygnaturze metody `GetAsyncEnumerator()` znajduje się parametr `CancellationToken`. Ponieważ pętla `await foreach` generuje tu kod wywołujący metodę `GetAsyncEnumerator()`, można wstrzymać token anulowania i umożliwić anulowanie prac, używając metod rozszerzającej `WithCancellation()` (na rysunku 20.2 widać, że typ `IAsyncEnumerable<T>` nie udostępnia bezpośrednio metody `WithCancellation()`). Aby umożliwić anulowanie prac w metodzie generującej strumień asynchroniczny, dodaj opcjonalny parametr `CancellationToken`, używając atrybutu `EnumeratorCancellation`. Ilustruje to deklaracja metody `EncryptFilesAsync`:

```
static public async IAsyncEnumerable<string>
EncryptFilesAsync(
    string directoryPath = null,
    string searchPattern = "*",
    [EnumeratorCancellation] CancellationToken
        cancellationToken = default)
{ ... }
```



Rysunek 20.2. `IAsyncEnumerable<T>` i powiązane interfejsy

Na listingu 20.6 pokazana jest metoda generująca strumień asynchroniczny, która zwraca obiekt typu `IAsyncEnumerable<T>`. Jednak, podobnie jak w przypadku iteratorów bez obsługi asynchroniczności, można zaimplementować interfejs `IAsyncEnumerable<T>` także w inny sposób — pisząc metodę `GetAsyncEnumerator()` z tego interfejsu. Każda klasa z implementacją tego interfejsu umożliwia iterację z wykorzystaniem instrukcji `await foreach`. Ilustruje to listing 20.7.

Listing 20.7. Używanie strumieni asynchronicznych za pomocą wywołania await foreach

```
class AsyncEncryptionCollection : IAsyncEnumerable<string?>
{
    public async IAsyncEnumerator<string> GetAsyncEnumerator(
        CancellationToken cancellationToken = default)
    {
        // ...
    }

    static public async void Main()
    {
        AsyncEncryptionCollection collection =
            new AsyncEncryptionCollection();
        // ...

        await foreach (string fileName in collection)
        {
            Console.WriteLine(fileName);
        }
    }
}
```

4.0

5.0

8.0

Uważaj jednak — pamiętaj, że zadeklarowanie metody jako asynchronicznej nie powoduje automatycznego równoległego jej wykonywania. Samo to, że metoda `EncryptFilesAsync()` jest asynchroniczna, nie oznacza jeszcze, że będzie równolegle iteracyjnie pobierać wszystkie pliki oraz wywoływać metody `File.ReadAllTextAsync()` i `Cryptographer.EncryptAsync()`. Aby zagwarantować równoległe wykonywanie operacji, trzeba zastosować zadania lub wywołania takie jak `System.Threading.Tasks.Parallel.ForEach()` (zobacz rozdział 21.).

Interfejs `IAsyncEnumerable<T>` i powiązany z nim interfejs `IAsyncEnumerator<T>` to wprowadzone w C# 8.0 dodatki (zobacz rysunek 20.2) przypominające ich synchroniczne odpowiedniki. Zauważ, że metody `IAsyncDisposable.DisposeAsync()` i `IAsyncEnumerator<T>.MoveNextAsync()` to asynchroniczne wersje analogicznych metod interfejsu `IEnumerator<T>`. Właściwość `Current` nie jest asynchroniczna. Ponadto w implementacjach asynchronicznych nie ma metody `Reset()`.

Interfejs `IAsyncDisposable` a deklaracje i instrukcje `await using`

`IAsyncDisposable` jest asynchronicznym odpowiednikiem interfejsu `IDisposable`, dlatego można go wywoływać z użyciem nowych **instrukcji** lub **deklaracji** `await using` z wersji C# 8.0. Na listingu 20.6 w deklaracji strumienia `outputFileStream` używana jest druga z tych możliwości, ponieważ typ `FileStream` obok implementacji interfejsu `IAsyncEnumerable<T>`

zawiera też implementację interfejsu `IAsyncDisposable`. Podobnie jak wtedy, gdy stosujesz deklarację `using`, nie możesz ponownie przypisać wartości do zmiennej zadeklarowanej za pomocą składni `async using`.

Nie jest zaskoczeniem, że w instrukcji `await using` używana jest ta sama składnia, co w standardowej instrukcji `using`:

```
await using FileStream outputFileStream =
    new FileStream(encryptedFileName, FileMode.Create);
{ ... }
```

Obie wersje składni można stosować zawsze, gdy typ implementuje interfejs `IAsyncDisposable` lub udostępnia metodę `DisposeAsync()`. Kompilator języka C# dodaje wtedy blok `try-finally` od miejsca deklaracji do wyjścia zmiennej z zasięgu, a następnie wywołuje `await DisposeAsync()` w bloku `finally`³. To podejście gwarantuje, że wszystkie zasoby zostaną zwolnione.

Zauważ, że interfejsy `IAsyncDisposable` i `IDisposable` nie są powiązane ze sobą relacją dziedziczenia. Dlatego ich implementacje też nie są zależne od siebie — można zaimplementować jeden z nich, ignorując drugi.

Używanie technologii LINQ razem z interfejsem IAsyncEnumerable

4.0

5.0

8.0

W instrukcji `await foreach` na listingu 20.6 wywoływana jest metoda `AsyncEnumerable.Zip()` z technologii LINQ, aby połączyć pierwotną nazwę pliku z nazwą zaszyfrowanego pliku.

```
await foreach (
    (string fileName, string encryptedFileName) in
        EncryptFilesAsync(files)
            .Zip(files.ToAsyncEnumerable())
{
    Console.WriteLine($"{fileName}=>{encryptedFileName}");
}
```

Typ `AsyncEnumerable`, jak można się domyślić, udostępnia mechanizmy technologii LINQ zgodne z interfejsem `IAsyncEnumerable<T>`. Jednak biblioteka z asynchronicznymi metodami technologii LINQ nie jest dostępna w bibliotece BCL. Dlatego⁴ aby uzyskać dostęp do asynchronicznych możliwości technologii LINQ, trzeba dodać referencję do pakietu NuGet `System.Linq.Async`.

Typ `AsyncEnumerable` jest zdefiniowany w przestrzeni nazw `System.Linq` (a nie w odrębnej unikatowej przestrzeni nazw z typami asynchronicznymi). Nie jest zaskoczeniem, że zawiera on asynchroniczne wersje standardowych operatorów technologii LINQ, takich jak metody `Where()`, `Select()` i użycia na omawianym listingu metoda `Zip()`. Można je nazwać „wersjami asynchronicznymi”, ponieważ są metodami rozszerzającymi interfejsu `IAsyncEnumerable` (a nie interfejsu `IEnumerable<T>`). Typ `AsyncEnumerable` obejmuje też zestaw metod `*Async()`, `*AwaitAsync()` i `*AwaitWithCancellationAsync()`. Wersje `Select*`() każdej z tych metod są pokazane na listingu 20.8.

³ Wywołania `await` w bloku `finally` można stosować od wersji C# 6.0.

⁴ Przynajmniej w czasie, gdy powstaje ta książka (w platformach .NET Core 3.0 i 3.1).

Listing 20.8. Sygnatury metod Select*()⁵ z typu AsyncEnumerable

```

namespace System.Linq
{
    public static class AsyncEnumerable
    {
        // ...
        public static IAsyncEnumerable<TResult> Select<TSource?, TResult?>(
            this IAsyncEnumerable<TSource> source,
            Func<TSource, TResult> selector);
        public static IAsyncEnumerable<TResult> SelectAwait<TSource?, TResult?>(
            this IAsyncEnumerable<TSource> source,
            Func<TSource, ValueTask<TResult>>? selector);
        public static
            IAsyncEnumerable<TResult> SelectAwaitWithCancellation<
                TSource?, TResult?>(
                    this IAsyncEnumerable<TSource> source,
                    Func<TSource, CancellationToken,
                        ValueTask<TResult>> selector);
        // ...
    }
}

```

4.0

5.0

8.0

Metody pasujące do ich odpowiedników z typu Enumerable (tu są to metody z rodziny Select()) mają podobne sygnatury, przy czym parametry TResult i TSource są w obu wersjach inne. Metody z członem Await w nazwie mają sygnatury asynchroniczne z selektorem, który zwraca wartość typu ValueTask<T>. Możesz na przykład wywołać w SelectAwait() metodę EncryptFileAsync() z listingu 20.6:

```

IAsyncEnumerable<string> items = files.ToAsyncEnumerable();
items = items.SelectAwait(
    (text, id) => EncryptFileAsync(text));

```

Należy zauważyć, że metoda EncryptionFileAsync() zwraca wartość typu ValueTask<T>, potrzebną zarówno w metodach *Await(), jak i w metodach *AwaitWithCancellationAsync(). W tych ostatnich można podać też token anulowania.

Nastecną wątką wzmianki metodą asynchroniczną z technologii LINQ jest użyta na listingu 20.6 metoda ToAsyncEnumerable(). Ponieważ asynchroniczne metody z technologii LINQ współdziałyają z interfejsem IAsyncEnumerable<T>, metoda ToAsyncEnumerable() przekształca typ IEnumerable<T> na IAsyncEnumerable<T>. Metoda ToEnumerable() przeprowadza konwersję w odwrotną stronę. Trzeba w tym miejscu przyznać, że wywołanie files.ToAsyncEnumerable() w omawianym fragmencie jest dość naciąganym przykładem obrazującym pobieranie obiektu typu IAsyncEnumerable<string>.

Skalarne wersje asynchronicznych metod z technologii LINQ (zestawy składowych *Await(), *AwaitAsync() i *AwaitWithCancellation()) też odpowiadają metodom z interfejsu IEnumerable<T>. Najważniejszą różnicą jest to, że metody z technologii LINQ zwracają wartości typu ValueTask<T>. Poniższy fragment pokazuje, jak używać metodę AverageAsync():

⁵ Z pominięciem metod SelectMany().

```
double average = await AsyncEnumerable.Range(0, 999).AverageAsync();
```

Można więc zastosować wywołanie `await`, aby potraktować zwracaną wartość jako liczbę typu `double` zamiast `ValueTask<double>`.

Koniec
8.0

Zwracanie wartości void w metodach asynchronous

Poznałeś już kilka dozwolonych typów wartości zwracanych przez metody asynchronousne, w tym `Task`, `Task<T>`, `ValueTask<T>` i `IAsyncEnumerable<T>`. Wszystkie one udostępniają metodę `GetAwaiter()`. Istnieje też dozwolony „typ” (tak naprawdę jest to brak typu) bez obsługi metody `GetAwaiter()`. Ta ostatnia możliwa zwracana wartość w metodach asynchronousnych to `void`. Metody zwracające taką wartość są dalej nazywane **metodami async void**. Zwykle należy unikać metod tego rodzaju. Inaczej niż przy zwracaniu wartości typów z metodą `GetAwaiter()` tu nie da się określić, kiedy metoda zakończyła pracę. Ponadto gdy następuje wyjątek, zwracanie `void` sprawia, że nie ma kontenera, który pozwoliłby poinformować o danym wyjątku. Każdy wyjątek zgłoszany w metodach `async void` przeważnie trafia do obiektu typu `UISynchronizationContext` i w praktyce staje się nieobsłużonym wyjątkiem (zobacz zagadnienie dla zaawansowanych „Radzenie sobie z nieobsłużonymi wyjątkami w wątku” w rozdziale 19.).

4.0

5.0

Skoro zwykle należy unikać metod `async void`, dlaczego w ogóle są one dozwolone? Wykrywa to stąd, że umożliwiają obsługę zdarzeń asynchronousnych. W rozdziale 14. wyjaśnimy, że zdarzenie należy deklarować za pomocą typu `EventHandler<T>` o następującej sygnaturze:

```
void EventHandler<TEventArgs>(object sender, TEventArgs e)
```

Dlatego aby dostosować się do konwencji dopasowywania zdarzeń do sygnatury `EventHandler<T>`, asynchronousne zdarzenie musi zwracać wartość `void`. Można zasugerować zmianę konwencji, jednak jak wiesz z rozdziału 14., może istnieć wielu subskrybentów, a pobieranie wartości zwracanych przez różnych subskrybentów jest nieintuicyjne i niewygodne. Dlatego zaleca się unikanie metod `async void`, chyba że służą one do obsługi zdarzeń. W takim scenariuszu nie powinny zgłaszać wyjątków. Inna możliwość to udostępnić kontekst synchronizacji, aby otrzymywać powiadomienia o zdarzeniach związanych z synchronizacją, na przykład z sregowaniem pracy (m.in. wywołania `Task.Run()`), a także — co prawdopodobnie ważniejsze — przyjmować nieobsługiwane wyjątki. Na listingu 20.9 i w powiązanych danych wyjściowych 20.2 pokazano, jak to zrobić.

Listing 20.9. Przechwytywanie wyjątków z metody `async void`

```
using System;
using System.Threading;
using System.Threading.Tasks;

public class AsyncSynchronizationContext : SynchronizationContext
{
    public Exception? Exception { get; set; }
    public ManualResetEventSlim ResetEvent { get; } = new
        ManualResetEventSlim();
```

```

public override void Send(SendOrPostCallback callback, object? state)
{
    try
    {
        Console.WriteLine($"@\"Powiadomienie w metodzie Send...(ID wątku: { Thread.CurrentThread.ManagedThreadId})\"");
        callback(state);
    }
    catch (Exception exception)
    {
        Exception = exception;
#if !WithOutUsingResetEvent
        ResetEvent.Set();
#endif
    }
}

public override void Post(SendOrPostCallback callback, object? state)
{
    try
    {
        Console.WriteLine($"@\"Powiadomienie w metodzie Post...(ID wątku: { Thread.CurrentThread.ManagedThreadId})\"");
        callback(state);
    }
    catch (Exception exception)
    {
        Exception = exception;
#if !WithOutUsingResetEvent
        ResetEvent.Set();
#endif
    }
}

public static class Program
{
    static bool EventTriggered { get; set; }

    public const string ExpectedExceptionMessage = "Oczekiwany wyjątek";
    public static void Main()
    {
        SynchronizationContext? originalSynchronizationContext =
            SynchronizationContext.Current;
        try
        {
            AsyncSynchronizationContext synchronizationContext =
                new AsyncSynchronizationContext();
            SynchronizationContext.SetSynchronizationContext(
                synchronizationContext);

            await OnEvent(typeof(Program), EventArgs.Empty);

#if WithOutUsingResetEvent
            Task.Delay(1000).Wait(); //
#else
            synchronizationContext.ResetEvent.Wait();
#endif
        }
    }
}

```

4.0

5.0

```

if(synchronizationContext.Exception != null)
{
    Console.WriteLine($"Zgłaszanie oczekiwanego wyjątku....(ID wątku: {
        Thread.CurrentThread.ManagedThreadId})");
    System.Runtime.ExceptionServices.ExceptionDispatchInfo.
        →Capture(synchronizationContext.Exception).Throw();
}
}
catch(Exception exception)
{
    Console.WriteLine($"{exception} zgłoszony zgodnie z oczekiwaniami. (ID wątku: {
        Thread.CurrentThread.ManagedThreadId})");
}
finally
{
    SynchronizationContext.SetSynchronizationContext(
        originalSynchronizationContext);
}
}

private static async void OnEvent(object sender, EventArgs eventArgs)
{
    Console.WriteLine($"Wywołanie Task.Run...(ID wątku: {
        Thread.CurrentThread.ManagedThreadId})");
    await Task.Run(()=>
    {
        EventTriggered = true;
        Console.WriteLine($"Wykonywanie zadania... (ID wątku: {
            Thread.CurrentThread.ManagedThreadId})");
        throw new Exception(ExpectedExceptionMessage);
    });
}
}

```

4.0

5.0

DANE WYJŚCIOWE 20.2

```

Wywołanie Task.Run...(ID wątku: 8)
Wykonywanie zadania... (ID wątku: 9)
Powiadomienie w metodzie Post...(ID wątku: 8)
Powiadomienie w metodzie Post...(ID wątku: 8)
Zgłaszanie oczekiwanej wyjątku....(ID wątku: 8)
System.Exception: Expected Exception
    at AddisonWesley.Michaelis.EssentialCSharp.Chapter20.
        →Listing20_09.Program.Main() in
...Listing20.09.AsyncVoidReturn.cs:line 80 thrown as expected.(ID wątku: 8)

```

Kod z listingu 20.9 działa proceduralnie do uruchomienia instrukcji `await Task.Run()` w metodzie `OnEvent()`. Po zakończeniu wykonywania tej instrukcji sterowanie jest przekazywane do metody `Post()` klasy `AsyncSynchronizationContext`. Po zakończeniu wykonywania metody `Post()` uruchamiana jest instrukcja `Console.WriteLine("Zgłaszanie wyjątku...")`, po czym zgłoszany jest wyjątek. Zostaje on przechwycony przez metodę `AsyncSynchronizationContext.Post()` i przekazany z powrotem do metody `Main()`.

W tym przykładzie używane jest wywołanie `Task.Delay()`, aby zagwarantować, że program nie zakończy pracy przed wywołaniem `Task.Run()`. Zalecanym rozwiązaniem jest jednak stosowanie klasy `ManualResetEventSlim`, co opisano w rozdziale 22.

Początek
8.0

ZAGADNIENIE DLA ZAAWANSOWANYCH

Inne typy zwracanych wartości odpowiednie dla metod asynchronouszych

Wyrażenie podane po słowie kluczowym await zazwyczaj zwraca wartość typu Task, Task<T> lub ValueTask<T>, czasem void, a czasem — od wersji C# 8.0 — IAsyncEnumerable<T>/IAsyncEnumerator<string>. Słowo *zazwyczaj* celowo ma sygnalizować brak pewności. W rzeczywistości reguła dotycząca typu zwracanej wartości, na jaką oczekuje wywołanie await, jest bardziej ogólna — dopuszczalne są typy udostępniające metodę GetAwaiter(). Ta metoda generuje obiekt o określonych właściwościach i metodach potrzebnych kompilatorowi do zmiany struktury kodu. Chodzi tu o obiekt implementujący interfejs INotifyCompletion i udostępniający metodę GetResult(). Dopuszczanie takich typów umożliwia rozszerzanie systemu przez niezależnych programistów. Jeśli chcesz zaprojektować własny asynchronousny system, który nie jest oparty na typie Task, a wykorzystuje inny typ do reprezentowania asynchronousnej pracy, możesz to zrobić i nadal stosować słowo kluczowe await.

Do wersji C# 8.0 metody asynchronousne mogły zwracać wyłącznie obiekty typu Task, Task<T>, ValueTask<T> oraz void. Nie miało przy tym znaczenia, jaki typ był używany w instrukcjach await w metodzie. Dzięki wprowadzeniu w C# 8.0 ogólniejszej zasady opartej na dostępności metody GetAwaiter() można zwracać także wartości typów IAsyncEnumerable<T> i IAsyncEnumerator<string>.

4.0

5.0

Koniec
8.0

Asynchronousne lambdy i funkcje lokalne

Wyrażenie lambda przekształcone w delegat można zastosować w celu zwięzłego zadeklarowania zwykłej metody. Od wersji C# 5.0 w delegaty można przekształcać także lambdy z wyrażeniami await. W tym celu wystarczy poprzedzić wyrażenie lambda słowem kluczowym async. Na listingu 20.10 asynchronousna lambda jest najpierw przypisywana do zmiennej Func<string, Task> writeWebRequestSizeAsync. Następnie do jej wywołania używany jest operator await.

Listing 20.10 Asynchronousna interakcja klienta z serwerem oparta na wyrażeniu lambda

```
using System;
using System.IO;
using System.Net;
using System.Linq;
using System.Threading.Tasks;

public class Program
{
    public static void Main(string[] args)
    {
        string url = "http://www.IntelliTect.com";
        if (args.Length > 0)
        {
            url = args[0];
        }
    }
}
```

```

Console.WriteLine(url);

Func<string, Task> writeWebRequestSizeAsync =
    async (string webRequestUrl) =>
{
    // Obsługę błędów pominięto, aby
    // kod był bardziej zrozumiały.
    WebRequest webRequest =
        WebRequest.Create(url);

    WebResponse response =
        await webRequest.GetResponseAsync();

    // Jawne zliczanie zamiast wywołania
    // webRequest.ContentLength, aby
    // zilustrować użycie kilku operatorów await
    using(StreamReader reader =
        new StreamReader(
            response.GetResponseStream()))
    {
        string text =
            (await reader.ReadToEndAsync());
        Console.WriteLine(
            FormatBytes(text.Length));
    }
};

Task task = writeWebRequestSizeAsync(url);

while (!task.Wait(100))
{
    Console.Write(".");
}
// ...
}

```

4.0

5.0

Od wersji C# 7.0 ten sam efekt można uzyskać za pomocą funkcji lokalnej. Na przykład na listingu 20.10 można zmodyfikować nagłówek wyrażenia lambda (cały kod do operatora => włącznie) na następującą postać:

```
async Task WriteWebRequestSizeAsync(string webRequestUrl)
```

Cały kod w ciele lambdy, w tym nawiasy klamrowe, nie wymaga zmian.

Zauważ, że wyrażenie lambda z modyfikatorem **async** musi spełniać te same warunki co metoda z tym modyfikatorem.

- Wyrażenie lambda z modyfikatorem **async** musi być przekształcane w delegat zwracający wartość odpowiedniego typu.
- Lambda jest przekształcana w taki sposób, że instrukcje **return** stają się sygnałami, iż zadanie zwrócone przez lambdę zakończyło pracę i zwróciło określony wynik.
- Kod w wyrażeniu lambda jest wykonywany synchronicznie do momentu napotkania pierwszej instrukcji **await** dotyczącej nieukończonego kodu z obsługą oczekiwania (ang. *awaitable*).

Początek
7.0

- Wszystkie instrukcje po słowie `await` są wykonywane jako kontynuacje zadania zwróconego przez wywołaną metodę asynchroniczną (jeśli jednak kod z obsługi oczekiwania został już ukończony, instrukcje są wykonywane synchronicznie, a nie jako kontynuacje).
- Wyrażenie lambda z modyfikatorem `async` można wywoływać z wykorzystaniem słowa `await` (nie pokazano tego na listingu 20.10).

Koniec
7.0

4.0

5.0

ZAGADNIENIE DLA POCZĄTKUJĄCYCH I ZAAWANSOWANYCH

Tworzenie niestandardowej metody asynchronicznej

Tworzenie za pomocą słowa kluczowego `await` metody asynchronicznej, która korzysta z innej metody asynchronicznej (wywołującej jeszcze inną metodę asynchroniczną), jest stosunkowo proste. Jednak w pewnym miejscu w hierarchii wywołań niezbędna jest „ostateczna” asynchroniczna metoda zwracająca obiekt typu `Task`. Pomyśl na przykład o asynchronicznej metodzie służącej do uruchamiania programu działającego w wierszu poleceń, przy czym możliwy ma być dostęp do danych wyjściowych z tego programu. Taką metodę można zadeklarować w następujący sposób:

```
public static Task<Process> RunProcessAsync(string filename)
```

Najprostsze rozwiązywanie to wykorzystanie metody `Task.Run()` oraz wywołanie metod `Start()` i `WaitForExit()` z typu `System.Diagnostics.Process`. Jednak tworzenie dodatkowego wątku w bieżącym procesie jest zbędne, gdy wywołany proces ma własny zestaw wątków. Aby zaimplementować metodę `RunProcessAsync()` i zwrócić sterowanie do kontekstu synchronizacji jednostki wywołującej po zakończeniu działania przez wywołany proces, można wykorzystać obiekt `TaskCompletionSource<T>`, co pokazano na listingu 20.11.

Listing 20.11. Implementowanie niestandardowej metody asynchronicznej

```
using System.Diagnostics;
using System.Threading;
using System.Threading.Tasks;
class Program
{
    static public Task<Process> RunProcessAsync(
        string fileName,
        string arguments = "",
        CancellationToken cancellationToken = default)
    {
        TaskCompletionSource<Process> taskCS =
            new TaskCompletionSource<Process>();

        Process process = new Process()
        {
            StartInfo = new ProcessStartInfo(fileName)
            {
                UseShellExecute = false,
                Arguments = arguments
            },
            EnableRaisingEvents = true
        };
    }
}
```

```

process.Exited += (sender, localEventArgs) =>
{
    taskCS.SetResult(process);
};

cancellationToken
    .ThrowIfCancellationRequested();

process.Start();

cancellationToken.Register(() =>
{
    process.CloseMainWindow();
});

return taskCS.Task;
}
// ...
}

```

Na razie zignoruj wyróżnione fragmenty i skoncentruj się na wzorcu polegającym na używaniu zdarzenia do powiadamiania o zakończeniu pracy przez proces. Ponieważ typ `System.Diagnostics.Process` zgłasza powiadomienie o ukończeniu działania, program rejestruje chęć otrzymywania tych powiadomień i wykorzystuje je do uruchamiania wywołania zwrotnego, w którym można wywołać metodę `TaskCompletionSource.SetResult()`. W kodzie z listingu 20.11 zastosowano dość często używany wzorzec. Możesz wykorzystać go do tworzenia asynchronicznych metod bez wywoływania metody `Task.Run()`.

4.0

5.0

Innym ważnym mechanizmem, którego obsługa może być potrzebna w metodzie z modyfikatorem `async`, jest anulowanie. We wzorcu TAP anulowanie jest obsługiwane za pomocą tych metod co w bibliotece TPL, czyli przy użyciu typu `System.Threading.CancellationToken`. Na listingu 20.11 wyróżniony jest kod potrzebny do obsługi anulowania. W przykładowym programie możliwe jest anulowanie pracy jeszcze przed uruchomieniem procesu. Można też spróbować zamknąć główne okno aplikacji (jeśli istnieje). Bardziej agresywne rozwiązanie polega na wywołaniu metody `Process.Kill()`, co jednak może prowadzić do problemów w wykonywanym programie.

Zauważ, że program rejestruje chęć otrzymywania zdarzenia anulowania dopiero po rozpoczęciu pracy przez proces. To pozwala uniknąć sytuacji wyścigu, która mogłaby wystąpić, gdyby prace anulowano jeszcze przed rozpoczęciem wykonywania procesu.

Ostatnim mechanizmem, który warto dodać, jest aktualizowanie informacji o postępie. Listing 20.12 zawiera pełną wersję metody `RunProcessAsync()` z dodaną obsługą tego mechanizmu.

Listing 20.12. Implementowanie niestandardowej metody asynchronicznej z obsługą informowania o postępie prac

```

using System;
using System.Diagnostics;
using System.Threading;
using System.Threading.Tasks;
class Program

```

```

{
    static public Task<Process> RunProcessAsync(
        string fileName,
        string arguments = "",
        CancellationToken cancellationToken =
            default(CancellationToken),
        IProgress<ProcessProgressEventArgs>? progress =
            null,
        object? objectState = null)
    {
        TaskCompletionSource<Process> taskCS =
            new TaskCompletionSource<Process>();

        Process process = new Process()
        {
            StartInfo = new ProcessStartInfo(fileName)
            {
                UseShellExecute = false,
                Arguments = arguments,
                RedirectStandardOutput =
                    progress != null
                },
                EnableRaisingEvents = true
            };

        process.Exited += (sender, localEventArgs) =>
        {
            taskCS.SetResult(process);
        };

        if (progress != null)
        {
            process.OutputDataReceived +=
                (sender, localEventArgs) =>
            {
                progress.Report(
                    new ProcessProgressEventArgs(
                        localEventArgs.Data,
                        objectState));
            };
        }

        if (cancellationToken.IsCancellationRequested)
        {
            cancellationToken
                .ThrowIfCancellationRequested();
        }

        process.Start();

        if (progress != null)
        {
            process.BeginOutputReadLine();
        }

        cancellationToken.Register(() =>
        {
            process.CloseMainWindow();
        });
    }
}

```

4.0

5.0

```

    cancellationToken
        .ThrowIfCancellationRequested();
    });

    return taskCS.Task;
}
// ...
}
class ProcessProgressEventArgs
{
    // ...
}

```

Dokładne zrozumienie tego, co dzieje się w metodzie z modyfikatorem `async`, może się okazać trudne. Jednak i tak jest łatwiejsze niż ustalenie tego, co robi asynchronouszny kod oparty na jawnych kontynuacjach w postaci lambd. Oto najważniejsze kwestie, o jakich trzeba pamiętać.

- Gdy sterowanie napotyka słowo kluczowe `await`, znajdujące się po tym słowie wyrażenie tworzy zadanie⁶. Wtedy sterowanie wraca do jednostki wywołującej, która może kontynuować pracę w czasie, gdy zadanie jest wykonywane asynchronousznie.
- Po pewnym czasie po ukończeniu zadania sterowanie wznowia pracę w punkcie po wyrażeniu ze słowem kluczowym `await`. Jeśli zadanie powiązane z tym słowem generuje wynik, jest on pobierany. Jeżeli to zadanie spowodowało błąd, zgłoszony jest wyjątek.
- Instrukcja `return` w metodzie z modyfikatorem `async` powoduje, że zadanie powiązane z wywołaniem tej metody zostaje ukończone. Jeśli dana instrukcja `return` jest powiązana z wartością, zwracana wartość staje się wynikiem zadania.

4.0

5.0

Programy szeregujące zadania i kontekst synchronizacji

W kilku miejscach tego rozdziału wspomniano o programie szeregującym i jego roli w określaniu tego, jak wydajnie przydzielić pracę wątkom. W kodzie program szeregujący zadania to obiekt typu `System.Threading.Tasks.TaskScheduler`. Ta klasa domyślnie wykorzystuje pulę wątków do szeregowania zadań i ustala, w jaki sposób bezpiecznie i wydajnie je wykonać — kiedy powtórnie je wykorzystać, kiedy je usunąć, a kiedy utworzyć dodatkowe zadania.

Możesz utworzyć własny program szeregujący zadania, który planuje ich wykonywanie w inny sposób. W tym celu utwórz nowy typ pochodny od klasy `TaskScheduler`. Za pomocą statycznej metody `FromCurrentSynchronizationContext()` możesz pobrać obiekt typu `TaskScheduler`, który zaplanuje wykonanie zadania w bieżącym wątku (a dokładniej w powiązanym z nim **kontekście synchronizacji**) zamiast w innym wątku roboczym⁷.

⁶ Technicznie chodzi tu o typ z obsługą oczekiwania, co opisano w zagadnieniu dla zaawansowanych „Inne typy zwracanych wartości odpowiednie w metodach asynchronousznych”.

⁷ Zobacz na przykład listing C.8 w książce *Multithreading Patterns Prior to C# 5.0* dostępnej w witrynie <http://IntelliTect.com/EssentialCSharp5>.

Kontekst synchronizacji, w którym wykonywane są dane zadanie i jego zadania kontynuacyjne, jest ważny, ponieważ oczekujące zadanie komunikuje się z kontekstem synchronizacji (jeśli ten istnieje), by możliwe było wydajne i bezpieczne wykonywanie zadań. Listing 20.13 (powiązany z danymi wyjściowymi 20.3) zawiera kod podobny do tego z listingu 19.3, ale w momencie wyświetlanego komunikatu podaje też identyfikator wątku.

Listing 20.13. Wywoływanie metody Task.ContinueWith()

```
using System;
using System.Threading;
using System.Threading.Tasks;

public class Program
{
    public static void Main()
    {
        DisplayStatus("Przed");
        Task taskA =
            Task.Run(() =>
                DisplayStatus("Rozpoczynanie pracy..."))
                .ContinueWith( antecedent =>
                    DisplayStatus("Kontynuacja dla zadania A..."));
        Task taskB = taskA.ContinueWith( antecedent =>
            DisplayStatus("Kontynuacja dla zadania B..."));
        Task taskC = taskA.ContinueWith( antecedent =>
            DisplayStatus("Kontynuacja dla zadania C..."));
        Task.WaitAll(taskB, taskC);
        DisplayStatus("Ukończono!");
    }

    private static void DisplayStatus(string message)
    {
        string text = string.Format(
            ${ Thread.CurrentThread.ManagedThreadId
            }: { message });
        Console.WriteLine(text);
    }
}
```

4.0

5.0

DANE WYJŚCIOWE 20.3.

```
1: Przed
3: Rozpoczynanie pracy...
4: Kontynuacja dla zadania A...
3: Kontynuacja dla zadania C...
4: Kontynuacja dla zadania B...
1: Ukończono!
```

W danych wyjściowych warto zwrócić uwagę na to, że identyfikator wątku czasem się zmienia, a czasem jest powtarzany. W tego rodzaju prostych aplikacjach konsolowych kontekst synchronizacji (dostępny za pomocą właściwości `SynchronizationContext.Current`) to `null`. Jest to domyślny kontekst synchronizacji, powodujący, że to pula wątków zarządza ich alokacją. To wyjaśnia, dlaczego identyfikator wątku zmienia się dla poszczególnych zadań. Czasem pula wątków ustala, że wydajniejszym rozwiązaniem jest utworzenie nowego wątku, a czasem stwierdza, że najlepsze podejście polega na ponownym wykorzystaniu istniejącego wątku.

Na szczęście w aplikacjach, w których kontekst synchronizacji jest niezbędny, zostaje on ustawiony automatycznie. Na przykład jeśli kod tworzący zadania działa w wątku utworzonym przez technologię ASP.NET, ten wątek jest powiązany z kontekstem synchronizacji typu `AspNetSynchronizationContext`. Natomiast jeśli kod działa w wątku utworzonym przez aplikację z interfejsem użytkownika z systemu Windows (czyli aplikację typu WPF lub Windows Forms), z wątkiem jest powiązany kontekst synchronizacji typu `DispatcherSynchronizationContext`. Dla aplikacji konsolowych domyślnie nie jest tworzony kontekst synchronizacji. Ponieważ biblioteka TPL komunikuje się z kontekstem synchronizacji, a ten kontekst zmienia się w zależności od warunków wykonywania programu, biblioteka TPL może zaplanować wykonywanie kontynuacji w kontekstach, które są wydajne i bezpieczne.

Aby zmodyfikować kod w celu wykorzystania określonego kontekstu synchronizacji musisz (1) ustawić kontekst synchronizacji i (2) zastosować modyfikatory `async` i `await`, by program komunikował się z takim kontekstem⁸.

Można zdefiniować niestandardowe konteksty synchronizacji i korzystać z istniejących kontekstów, by poprawić wydajność kodu w specyficznych sytuacjach. Jednak te zagadnienia wykraczają poza zakres tej książki.

4.0

5.0

Modyfikatory `async` i `await` w programach z interfejsem użytkownika z systemu Windows

Synchronizacja ma wyjątkowo duże znaczenie w programach z okienkowym interfejsem użytkownika i w programach sieciowych. Na przykład w programach z interfejsem użytkownika z systemu Windows przetwarzane są komunikaty związane z kliknięciem lub przesunięciem myszy. Ponadto interfejs użytkownika jest jednowątkowy, dlatego interakcja z jego komponentami (na przykład z polem tekstowym) zawsze musi się odbywać w jednym wątku interfejsu użytkownika. Jedną z ważnych zalet stosowania modyfikatorów `async` i `await` jest to, że pozwalają one wykorzystać kontekst synchronizacji w celu upewnienia się, że kontynuacja (praca opisywana po instrukcji `await`) zawsze będzie wykonywana w tym samym zadaniu, które wywołało daną instrukcję `await`. To podejście jest przydatne, ponieważ eliminuje konieczność jawnego przełączania się z powrotem do wątku interfejsu użytkownika w celu zarządzania przepływem sterowania.

Aby lepiej docenić przydatność tej techniki, przyjrzyj się zdarzeniu interfejsu użytkownika związanemu z kliknięciem przycisku w aplikacji WPF (zobacz listing 20.14).

Listing 20.14. Synchroniczne wywołanie operacji o dużej latencji w aplikacji WPF

```
using System;

private void PingButton_Click(
    object sender, RoutedEventArgs e)
{
    StatusLabel.Content = "Przesyłanie sygnału ping...";
    UpdateLayout();
```

⁸ Prosty przykład ilustrujący ustawianie kontekstu synchronizacji wątku i korzystanie z programu szeregującego do szeregowania zadania z użyciem danego wątku znajdziesz na listingu C.8 w książce *Multithreading Patterns Prior to C# 5.0* dostępnej w witrynie <http://IntelliTect.com/EssentialCSharp>.

```
Ping ping = new Ping();
PingReply pingReply =
    ping.Send("www.IntelliTect.com");
StatusLabel.Text = pingReply.Status.ToString();
}
```

4.0

5.0

Ponieważ `StatusLabel` to kontrolka typu `System.Windows.Controls.TextBlock` z platformy WPF, a właściwość `Content` jest dwukrotnie aktualizowana w subskrybcie zdarzenia (`PingButton_Click()`), na pozór można założyć, że pierwszy napis, „Przesyłanie sygnału ping...”, będzie wyświetlany do czasu zwrócenia sterowania przez instrukcję `Ping.Send()`, po czym etykieta zmieni się na status odpowiedzi podanej w metodzie `Send()`. Jednak doświadczeni użytkownicy platform do tworzenia interfejsów użytkownika w systemie Windows wiedzą, że nie jest to poprawny opis. Komunikat jest przekazywany do pompy komunikatów, aby zaktualizować etykietę za pomocą tekstu „Przesyłanie sygnału ping...”, jednak ponieważ wątek interfejsu użytkownika jest zajęty wykonywaniem metody `PingButton_Click()`, pomba komunikatów systemu Windows nie jest przetwarzana. Gdy wątek interfejsu użytkownika wykona metodę i będzie mógł sprawdzić pompę komunikatów systemu Windows, w kolejce znajdzie się już drugie żądanie, aktualizujące wartość właściwości `Text`, dlatego jedyną komunikat, jaki zobaczy użytkownik, to końcowa informacja o stanie.

Aby rozwiązać ten problem za pomocą wzorca TAP, należy wprowadzić w kodzie zmiany wyróżnione na listingu 20.15.

Listing 20.15. Synchroniczne wywoływanie operacji o dużej latencji w aplikacji WPF z wykorzystaniem słowa kluczowego `await`

```
using System;
async private void PingButton_Click(
    object sender, RoutedEventArgs e)
{
    StatusLabel.Content = "Przesyłanie sygnału ping...";
    UpdateLayout();
    Ping ping = new Ping();
    PingReply pingReply =
        await ping.SendPingAsync("www.IntelliTect.com");
    StatusLabel.Text = pingReply.Status.ToString();
}
```

Ta zmiana ma dwie zalety. Po pierwsze, asynchroniczna natura przesyłania sygnału ping pozwala wątkowi jednostki wywołującej zwrócić sterowanie do kontekstu synchronizacji pętli komunikatów systemu Windows. Pozwala to przetworzyć aktualizację właściwości `StatusLabel.Content`, dzięki czemu użytkownik zobaczy tekst „Przesyłanie sygnału ping...”. Po drugie, gdy oczekujące wywołanie `ping.SendTaskAsync()` zakończy pracę, używany jest ten sam kontekst synchronizacji co w jednostce wywołującej. Wykorzystywany tu kontekst synchronizacji jest wyjątkowo przydatny w interfejsie użytkownika w systemie Windows. Używany jest jeden wątek, dlatego sterowanie zawsze jest zwracane do tego samego wątku, czyli do wątku interfejsu użytkownika. Oznacza to, że biblioteka TPL nie wykonuje natychmiast zadania kontynuacyjnego, ale komunikuje się z kontekstem synchronizacji, który przekazuje komunikat związany z kontynuacją do pompy komunikatów. Wątek interfejsu

użytkownika monitoruje pętlę komunikatów, dlatego po otrzymaniu komunikatu związanego z kontynuacją wywołuje kod podany po wywołaniu `await`. W efekcie kod kontynuacji jest wykonywany w tym samym wątku, w którym działa jednostka przetwarzająca pompę komunikatów.

Z wzorcem TAP związaną jest ważna cecha poprawiająca czytelność kodu. Zwróć uwagę na to, że na listingu 20.15 wywołanie pobierające wartość właściwości `pingReply.Status` znajduje się po instrukcji `await`, co wygląda naturalnie i wyraźnie pokazuje, że wywołanie zostało wykonane natychmiast po poprzednim wierszu. Jednak samodzielnie utworzony kod opisujący, co się naprawdę dzieje, byłby z wielu powodów znacznie mniej zrozumiały.

ZAGADNIENIE DLA POCZĄTKUJĄCYCH

Operatory await

Nie ma ograniczenia określającego, ile razy instrukcje `await` mogą się pojawić w jednej metodzie. Ponadto nie trzeba ich umieszczać jedna po drugiej. Możesz je umieścić w pętlach i przetwarzać jedna po drugiej, zgodnie z naturalnym przepływem sterowania przedstawionym w kodzie. Przyjrzyj się przykładowi z listingu 20.16.

4.0

5.0

Listing 20.16. Wykorzystanie instrukcji `await` w iteracjach

```
async private void PingButton_Click(
    object sender, RoutedEventArgs e)
{
    List<string> urls = new List<string>()
    {
        "www.habitat-spokane.org",
        "www.partnersintl.org",
        "www.iasssist.org",
        "www.fh.org",
        "www.worldvision.org"
    };
    IPStatus status;

    Func<string, Task<IPStatus>> func =
        async (localUrl) =>
    {
        Ping ping = new Ping();
        PingReply pingReply =
            await ping.SendPingAsync(localUrl);
        return pingReply.Status;
    };

    StatusLabel.Content = "Zgłaszanie sygnału ping...";

    foreach (string url in urls)
    {
        status = await func(url);
        StatusLabel.Text =
            $"{url}: {status.ToString() } ({Thread.CurrentThread.ManagedThreadId})";
    }
}
```

Niezależnie od tego, czy instrukcje `await` pojawiają się w pętli, czy w odrębnych wyrażeniach, są wykonywane sekwencyjnie (jedna po drugiej) w tej samej kolejności, w jakiej zostały uruchomione w wątku wywołującym. Na zapleczu takie instrukcje są łączone w łańcuch w podobny sposób jak w wyniku zastosowania instrukcji `Task.ContinueWith()`. Różnica polega na tym, że cały kod między operatorami `await` jest wykonywany w kontekście synchronizacji jednostki wywołującej.

Dodawanie obsługi wzorca TAP w interfejsie użytkownika to jeden z najważniejszych scenariuszy, na potrzeby których utworzono ten wzorzec. Drugi scenariusz związany jest z serwerem, a dokładniej z sytuacją, gdy od klienta przychodzi żądanie z kwerendą dotyczącą całej tabeli danych z bazy. Ponieważ przetwarzanie kwerend dotyczących danych bywa czasochłonne, należy wtedy utworzyć nowy wątek, zamiast wykorzystywać jeden z ograniczonej liczby wątków zaalokowanych w puli. Problem z tym podejściem polega na tym, że kwerenda jest wykonywana na innej maszynie. Nie ma więc powodu blokować całego wątku, który i tak przez większość czasu nie jest aktywny.

Oto podsumowanie — wzorzec TAP opracowano, by poradzić sobie z następującymi ważnymi problemami:

- Potrzebna jest możliwość wykonywania długotrwałych operacji bez blokowania wątku interfejsu użytkownika.
- Tworzenie nowego wątku (lub zadania) dla prac, które nie obciążają w dużym stopniu procesora, jest stosunkowo kosztowne, jeśli uwzględnić, że wątek czeka tylko na zakończenie prac.
- Gdy prace zostaną ukończone (albo za pomocą nowego wątku, albo z wykorzystaniem wywołania zwrotnego), często trzeba przełączyć kontekst synchronizacji wątku z powrotem do pierwotnej jednostki wywołującej, która zainicjowała dane prace.

TAP to nowy wzorzec odpowiedni do asynchronicznego wywoływania różnych prac — obciążających procesor zarówno w wysokim, jak i w niewielkim stopniu. Wszystkie języki platformy .NET bezpośrednio obsługują ten wzorzec.

Początek
8.0

Podsumowanie

Większość tego rozdziału dotyczy wprowadzonego w C# 5.0 wzorca TAP. Jest on powiązany ze składnią `async/await`. Szczegółowo pokazano tu, o ile prostsze jest używanie wzorca TAP w porównaniu ze stosowaniem technologii TPL — zwłaszcza gdy przekształcasz kod synchroniczny w wersję asynchroniczną. Opisano też wymogi dotyczące typów wartości zwracanych przez metody asynchroniczne. Składnia `async/await` znacznie upraszcza programowanie złożonych procesów pracy z użyciem obiektów `Task`, ponieważ kompilator automatycznie przekształca programy i zarządza kontynuacjami, tworząc większe zadania z mniejszych.

Dalej w rozdziale omówiono strumienie asynchroniczne i wprowadzony w C# 8.0 typ danych `IAsyncEnumerable<T>`. Wyjaśniono, jak korzystać z tych mechanizmów do tworzenia asynchronicznych iteratorów i jak stosować te iteratory w asynchronicznych instrukcjach `foreach`.

Na tym etapie czytelnicy powinni mieć już solidne podstawy z zakresu pisania kodu asynchronicznego z wyłączeniem równoległych iteracji (jest to temat rozdziału 21.) i synchronizowania wątków (opisanego w rozdziale 22.).

Koniec
8.0

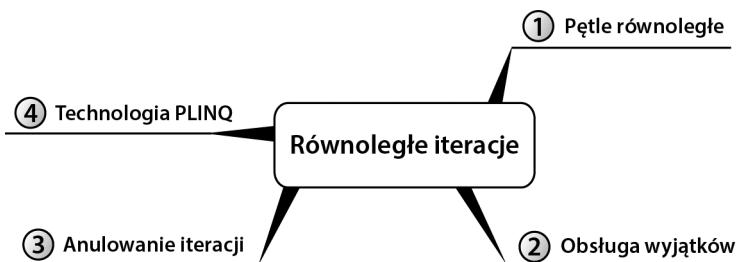
Koniec
5.0

Koniec
4.0

21

Równoległe iteracje

ROZDZIAŁE 19. WSPOMNIAНО, że koszty przetwarzania spadają. Dostępne są komputery z szybszymi procesorami, większą liczbą procesorów i większą liczbą rdzeni. Te trendy powodują, że coraz bardziej ekonomicznie zasadne jest intensyfikowanie równoległego przetwarzania, aby wykorzystać rosnącą moc obliczeniową. W tym rozdziale opisane jest równoległe wykonywanie pętli. Jest to jeden z najłatwiejszych sposobów na wykorzystanie większej mocy obliczeniowej. Duża część tego rozdziału to zagadnienia dla początkujących i zaawansowanych.



Równoległe wykonywanie iteracji pętli

Przyjrzyj się przedstawionej dalej instrukcji for i powiązanemu z nią kodowi (listing 21.1 i powiązane dane wyjściowe 21.1). Kod z listingu wywołuje metodę, która oblicza fragment rozwinięcia dziesiętnego liczby pi. Parametrami tej metody są liczba cyfr i cyfra, od której należy zacząć pracę. Same obliczenia nie są tu istotne. Ciekawe jest to, że obliczenia są „zawstydzająco” współbieżne — zawstydzające jest to, jak łatwo można podzielić duże zadanie (na przykład obliczenie liczby pi do milionowej cyfry po przecinku) na odpowiednią liczbę mniejszych zadań, które można wykonać równolegle. To właśnie tego rodzaju obliczenia najlepiej przyspieszyć za pomocą współbieżności.

Listing 21.1. Pętla for synchronicznie obliczająca liczbę pi w porcjach

```

using System;
using AddisonWesley.Michaelis.EssentialCSharp.Shared;

class Program
{
    const int TotalDigits = 100;
    const int BatchSize = 10;

    static void Main()
    {
        string pi = "";
        const int iterations = TotalDigits / BatchSize;
        for (int i = 0; i < iterations; i++)
        {
            pi += PiCalculator.Calculate(
                BatchSize, i * BatchSize);
        }
        Console.WriteLine(pi);
    }
}
using System;

class PiCalculator
{
    public static string Calculate(
        int digits, int startingAt)
    {
        // ...
    }
    // ...
}

```

DANE WYJŚCIOWE 19.9.

>3.14159265358979323846264338327950288419716939937510582097494459230781640
62862089986280348253421170679821480865132823066470938446095505822317253594
08128481117450284102701938521105559644622948954930381964428810975665933446
12847564823378678316527120190914564856692346034861045432664821339360726024
91412737245870066063155881748815209209628292540917153643678925903600113305
30548820466521384146951941511609433057270365759591953092186117381932611793
10511854807446237996274956735188575272489122793818301194912

4.0

Pętla for z tego kodu wykonuje każdą iterację synchronicznie i sekwencyjnie. Jednak ponieważ algorytm obliczania liczby pi dzieli przetwarzanie na niezależne porcje, nie trzeba obliczać poszczególnych fragmentów po kolejno (przy czym wyniki muszą być dołączane w odpowiedniej kolejności). Wyobraź sobie, co się stanie, jeśli wszystkie iteracje pętli będą wykonywane jednocześnie. Każdy procesor będzie wtedy mógł wykonywać jedną iterację równolegle z innymi procesorami przetwarzającymi inne iteracje. Dzięki jednoczesnemu wykonywaniu iteracji można coraz bardziej skracić czas wykonywania kodu, dodając kolejne procesory.

Biblioteka TPL udostępnia wygodną metodę `Parallel.For()`, która umożliwia uzyskanie tego efektu. Na listingu 21.2 pokazano, jak zmodyfikować sekwencyjny, jednowątkowy program z listingu 21.1 za pomocą tej metody pomocniczej.

Listing 21.2. Pętla for równolegle obliczająca fragmenty liczby pi

```

using System;
using System.Threading.Tasks;
using AddisonWesley.Michaelis.EssentialCSharp.Shared;

// ...
class Program
{
    static void Main()
    {
        string pi = "";
        const int iterations = TotalDigits / BatchSize;
        string[] sections = new string[iterations];
        Parallel.For(0, iterations, i =>
        {
            sections[i] = PiCalculator.Calculate(
                BatchSize, i * BatchSize);
        });
        pi = string.Join("", sections);
        Console.WriteLine(pi);
    }
}

```

Kod z listingu 21.2 wyświetla te same cyfry, które pokazano w danych wyjściowych 21.1. Jednak jeśli komputer ma kilka procesorów, nowa wersja kodu działa szybciej niż starsza (choć na maszynach jednoprocesorowych może się okazać wolniejsza). Interfejs API w postaci metody `Parallel.For()` jest tak zaprojektowany, by wyglądał podobnie do standardowej pętli `for`. Pierwszym parametrem tej metody jest `fromExclusive`, drugim `toExclusive`, a ostatnim — obiekt typu `Action<int>`, zawierający kod wykonywany w ciele pętli. Gdy jako ten obiekt podane jest wyrażenie lambda, kod wygląda podobnie jak w pętli `for`, przy czym wszystkie iteracje można teraz wykonywać równolegle. Podobnie jak pętla `for`, wywołanie metody `Parallel.For()` kończy się dopiero po wykonaniu wszystkich iteracji. Oznacza to, że do momentu dojścia sterowania do instrukcji `string.Join()` wszystkie fragmenty liczby pi są już obliczone.

Zauważ, że na listingu 21.2 kod łączący poszczególne fragmenty liczby pi nie znajduje się już w iteracji (w obiekcie typu `Action`). Ponieważ obliczenia fragmentów liczby pi z dużym prawdopodobieństwem nie będą kończone po kolejnej iteracji, dołączanie wyników po zakończeniu poszczególnych iteracji doprowadzi do błędnej kolejności cyfr. Nawet gdyby kolejność końca pracy nie była problemem, może wystąpić sytuacja wyścigu, ponieważ operator `+=` nie jest atomowy. Aby rozwiązać oba te problemy, wszystkie fragmenty liczby pi są zapisywane w tablicy, a do każdego jej elementu dostęp jednocześnie może uzyskać tylko jedna iteracja. Po obliczeniu wszystkich fragmentów liczby pi instrukcja `string.Join()` łączy je ze sobą. Oznacza to, że łącząc fragmenty liczby jest odraczane do momentu zakończenia pracy pętli `Parallel.For()`. To pozwala uniknąć sytuacji wyścigu spowodowanej przez fragmenty, które nie zostały jeszcze obliczone, i łączenia cyfr w niewłaściwej kolejności.

4.0

Aby zapewnić wysoką wydajność wykonywanej równolegle pętli, biblioteka TPL wykorzystuje te same techniki kierowania zapytań do wątku, które stosowane są przy szeregowaniu zadań. Biblioteka stara się zapewnić, że procesory nie zostaną przeciążone itd.

Wskazówka

STOSUJ pętle równoległe, gdy wykonywane obliczenia można łatwo rozbić na wiele wzajemnie niezależnych obliczeń zależnych od procesora, które można przetwarzać w dowolnej kolejności w dowolnych wątkach.

Biblioteka TPL udostępnia też równoległą wersję instrukcji `foreach`, pokazaną na lisingu 21.3.

Listing 21.3. Równoległe wykonywanie pętli `foreach`

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Threading.Tasks;

class Program
{
    // ...
    static void EncryptFiles(
        string directoryPath, string searchPattern)
    {
        IEnumerable<string> files = Directory.EnumerateFiles(
            directoryPath, searchPattern,
            SearchOption.AllDirectories);

        Parallel.ForEach(files, fileName =>
        {
            Encrypt(fileName);
        });
    }
    // ...
}
```

W tym przykładzie wywoływana jest metoda, która szyfruje każdy plik z kolekcji `files`. Szyfrowanie odbywa się równolegle, za pomocą tylu wątków, ile według biblioteki TPL zapewni wydajną pracę.

ZAGADNIENIE DLA ZAAWANSOWANYCH

W jaki sposób biblioteka TPL dostraja wydajność

Domyślny program szeregujący z biblioteki TPL wykorzystuje pulę wątków. Stosowane są przy tym rozmaite strategie, które mają zapewniać, że w każdym momencie wykonywana jest odpowiednia liczba wątków. Dwie z tych strategii to **wspinaczka** (ang. *hill climbing*) i **podkradanie pracy** (ang. *work stealing*).

Algorytm wspinaczki polega na tworzeniu wątków uruchamiających zadania i monitorowaniu wydajności tych zadań w celu eksperymentalnego ustalenia punktu, w którym dodanie kolejnych wątków prowadzi do spadku wydajności. Po dojściu do tego punktu liczbę wątków można zmniejszyć do poziomu, który zapewniał najwyższą wydajność.

Biblioteka TPL nie łączy oczekujących na wykonanie zadań z najwyższego poziomu z żadnym konkretnym wątkiem. Jeśli jednak zadanie działające w wątku tworzy inne zadanie, nowe zadanie jest automatycznie wiązane z tym samym wątkiem. Gdy nowe podrzędne zadanie jest ostatecznie szeregowane do wykonania, zwykle zostaje uruchomione w wątku zadania nadrzednego. Algorytm podkradania pracy wykrywa wątki o bardzo dużej lub bardzo małej ilości zaległej pracy. Wątek, z którym powiązana jest zbyt mała liczba zadań, czasem „podkrada” nieuruchomione jeszcze zadania z wątków, w których na wykonanie oczekuje zbyt wiele wątków.

Najważniejszą cechą tych algorytmów jest to, że umożliwiają bibliotece TPL dynamiczne dostrajanie wydajności w celu uniknięcia przeciążenia lub niepełnego wykorzystania procesorów. Te algorytmy pozwalają też równoważyć obciążenie procesorów.

Biblioteka TPL zwykle dobrze sobie radzi z dostrajaniem wydajności. Można jednak ułatwić jej działanie, podając wskazówki sugerujące najlepsze rozwiązanie. Taką wskazówką jest na przykład opcja `TaskCreationOptions.LongRunning` opisana wcześniej w podrozdziale „Długotrwałe zadania” w rozdziale 19. Możesz też jawnie poinformować program szeregujący zadania o tym, ile wątków najlepiej będzie zastosować dla pętli równoległej. Szczegółowe informacje na ten temat znajdziesz w zagadnieniu dla zaawansowanych „Opcje pętli równoległych”.

ZAGADNIENIE DLA POCZĄTKUJĄCYCH

Używanie typu `AggregateException` do obsługi wyjątków w pętlach równoległych

Wiesz już, że biblioteka TPL przechwytuje powiązane z zadaniami wyjątki i zapisuje je w wyjątku `AggregateException`. Dzieje się tak, ponieważ w danym zadaniu może wystąpić kilka wyjątków zgłoszonych w podzadaniach. Podobna sytuacja ma miejsce przy równoległym wykonywaniu pętli. Każda iteracja może zgłosić wyjątek, dlatego wyjątki trzeba zebrać w jeden wyjątek zbiorczy. Przyjrzyj się kodowi z listingu 21.4 i danym wyjściowym 21.2.

Listing 21.4. Obsługa nieobsłużonych wyjątków zgłoszonych w równolegle wykonywanych iteracjach

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Threading;
using System.Threading.Tasks;

public static class Program
{
    // ...
    static void EncryptFiles(
        string directoryPath, string searchPattern)
    {
        IEnumerable<string> files = Directory.EnumerateFiles(
            directoryPath, searchPattern,
            SearchOption.AllDirectories);
        try
        {
            Parallel.ForEach(files, fileName =>
            {

```

4.0

```

        Encrypt(fileName);
    });
}
catch(AggregateException exception)
{
    Console.WriteLine(
        "BŁĄD: {0}:", 
        exception.GetType().Name);
    foreach (Exception item in
        exception.InnerExceptions)
    {
        Console.WriteLine(" {0} - {1}",
            item.GetType().Name, item.Message);
    }
}
// ...
}

```

DANE WYJŚCIOWE 21.2.

BŁĄD: AggregateException:
 UnauthorizedAccessException - Attempted to perform an unauthorized operation.
 UnauthorizedAccessException - Attempted to perform an unauthorized operation.
 UnauthorizedAccessException - Attempted to perform an unauthorized operation.

Z danych wyjściowych 21.2 wynika, że w trakcie wykonywania pętli `Parallel.ForEach<T>(...)` wystąpiły trzy wyjątki. Jednak w kodzie znajduje się tylko jeden blok `catch`, przeznaczony dla wyjątków `System.AggregateException`. Wyjątki `UnauthorizedAccessException` są pobierane z właściwości `InnerExceptions` wyjątku `AggregateException`. W pętli wykonywanej przez metodę `Parallel.ForEach<T>()` każda iteracja może zgłosić wyjątek, a wyjątek `System.AggregateException` zgłoszony przez tę metodę zawiera we właściwości `InnerExceptions` wszystkie te wyjątki.

Anulowanie wykonywania pętli równoległej

4.0

W odróżnieniu od zadań, które wymagają jawnego wywołania metody, jeśli mają blokować dalsze instrukcje do czasu zakończenia pracy, pętle równoległe wykonują iteracje jednocześnie, ale zwracają sterowanie dopiero po zakończeniu działania całej pętli. Anulowanie pętli równoległej zwykle polega na przesłaniu żądania anulowania z wątku innego niż ten, w którym działa dana pętla równoległa. Na listingu 21.5 metoda `Parallel.ForEach<T>()` jest wywoływana za pomocą instrukcji `Task.Run()`. Dzięki temu kwerenda jest przetwarzana równolegle, a także asynchronicznie. Dlatego kod może wyświetlić użytkownikowi komunikat: **Wciśnij ENTER, aby zakończyć.**

Listing 21.5. Anulowanie pętli równoległej

```

using System;
using System.Collections.Generic;
using System.IO;
using System.Threading;
using System.Threading.Tasks;

```

```

public class Program
{
    // ...

    static void EncryptFiles(
        string directoryPath, string searchPattern)
    {
        string stars =
            "*".PadRight(Console.WindowWidth-1, '*');

        IEnumerable<string> files = Directory.GetFiles(
            directoryPath, searchPattern,
            SearchOption.AllDirectories);

        CancellationTokenSource cts =
            new CancellationTokenSource();
        ParallelOptions parallelOptions =
            new ParallelOptions
            { CancellationToken = cts.Token };
        cts.Token.Register(
            () => Console.WriteLine("Anulowanie..."));

        Console.WriteLine("Wciśnij ENTER, aby zakończyć.");

        Task task = Task.Run(() =>
        {
            try
            {
                Parallel.ForEach(
                    files, parallelOptions,
                    (fileName, loopState) =>
                {
                    Encrypt(fileName);
                });
            }
            catch(OperationCanceledException){}
        });

        // Oczekивание на данные выходные от пользователя.
        Console.Read();

        // Anulowanie kwerendy.
        cts.Cancel();
        Console.Write(stars);
        task.Wait();
    }
}

```

Dla pętli równoległych używany jest ten sam wzorzec z tokenem anulowania, który stosowany jest do zadań. Token pobrany z obiektu typu `CancellationTokenSource` jest wiązany z pętlą równoległą w wyniku wywołania przeciążonej metody `ForEach()` z parametrem typu `ParallelOptions`. To obiekt tego ostatniego typu zawiera token anulowania.

Zauważ, że jeśli anulujesz wykonywanie pętli równoległej, iteracje, które nie zostały jeszcze rozpoczęte, w ogóle nie zostaną uruchomione. Jest to efekt sprawdzenia właściwości `IsCancellationRequested`. Już uruchomione iteracje są wykonywane do momentu zakończenia.

Ponadto wywołanie metody `Cancel()` nawet po zakończeniu wszystkich iteracji też powoduje zgłoszenie zarejestrowanego (za pomocą instrukcji `cts.Token.Register()`) zdarzenia anulowania.

Jedyny sposób, za pomocą którego metoda `ForEach()` może stwierdzić, że pętla została anulowana, to zgłoszenie wyjątku `OperationCanceledException`. Ponieważ w tym kodzie anulowanie pętli jest oczekiwane, wyjątek zostaje przechwycony i zignorowany. To pozwala aplikacji wyświetlić tekst „Anulowanie...” i wiersz z gwiazdkami, a następnie zakończyć pracę.

■ ZAGADNIENIE DLA ZAAWANSOWANYCH

Opcje pętli równoległych

Choć zwykle nie jest to konieczne, można sterować maksymalnym poziomem równoległości (czyli liczbą wątków jednocześnie szeregowanych do wykonania). Służy do tego parametr `ParallelOptions` dostępny w przeciążonych wersjach pętli `Parallel.For()` i `Parallel.ForEach<T>()`. W niektórych sytuacjach programista może dobrze znać dany algorytm lub warunki sprawiające, że zmiana maksymalnego poziomu równoległości ma sens. Oto przykładowe warunki tego rodzaju:

- Sytuacje, gdy programista chce wyłączyć przetwarzanie równolegle, by ułatwić debugowanie lub analizy. Ustawienie maksymalnego poziomu równoległości na 1 gwarantuje, że iteracje pętli nie będą wykonywane współbieżnie.
- Sytuacje, w których z góry wiadomo, że poziom równoległości będzie obniżany z powodu czynników zewnętrznych (na przykład ograniczeń sprzętowych). Na przykład jeśli operacja równoległa wymaga użycia wielu portów USB, możliwe, że nie ma sensu tworzyć więcej wątków niż liczba dostępnych portów.
- Sytuacje, gdy iteracje pętli są bardzo długie (mierzone w minutach lub nawet godzinach). Pula wątków nie potrafi odróżnić długotrwałych iteracji od zablokowanych operacji, dlatego może utworzyć wiele nowych wątków, które wszystkie zostaną zajęte przez pętlę `for`. To może prowadzić do stopniowego dodawania kolejnych wątków i powstania bardzo dużej ich liczby w procesie.

4.0

To tylko przykładowe scenariusze. Aby ustawić maksymalny poziom równoległości, zastosuj właściwość `MaxDegreeOfParallelism` typu `ParallelOptions`.

Za pomocą właściwości `TaskScheduler` obiektu typu `ParallelOptions` możesz też wskazać niestandardowy program szeregujący zadania, używany do szeregowania zadań z każdej iteracji. Możliwe, że w programie działa asynchroniczna metoda obsługi zdarzeń, reagująca na kliknięcie przez użytkownika przycisku *Dalej*. Jeśli użytkownik kliknie ten przycisk kilkakrotnie, można zastosować niestandardowy program szeregujący, który priorytetowo traktuje ostatnio utworzone zadanie (a nie zadanie o najdłuższym czasie oczekiwania). Program szeregujący umożliwia określenie, w jaki sposób zadania będą wykonywane względem innych zadań.

Typ `ParallelOptions` udostępnia też właściwość `CancellationToken`, która pozwala poinformować pętlę o tym, że nie należy uruchamiać następnych iteracji. Ponadto w ciele iteracji można sprawdzać token anulowania, by ustalić, czy należy wcześniej zakończyć daną iterację.

ZAGADNIENIE DLA ZAAWANSOWANYCH

Wychodzenie z pętli równoległej

Pętla `Parallel.For()` (podobnie jak standardowa pętla `for`) umożliwia przerwanie pracy w celu wyjścia z pętli i anulowania jej dalszych iteracji. Jednak w kontekście równoległego wykonywania pętli przerwanie pracy oznacza, że nie należy uruchamiać kolejnych iteracji. Wszystkie obecnie działające iteracje są wykonywane do końca.

Aby przerwać pracę pętli równoległej, możesz utworzyć token anulowania i anulować pętlę w innym wątku, co opisano w zagadnienniu dla zaawansowanych „Opcje pętli równoległych”. Możesz też wykorzystać przeciążoną wersję metody `Parallel.For()`, w której delegat przyjmuje dwa parametry — indeks i obiekt typu `ParallelLoopState`. W iteracji, która ma przerwać pracę pętli, można wtedy wywołać metodę `Break()` lub `Stop()` przekazanego do delegata obiektu typu `ParallelLoopState`. Metoda `Break()` oznacza, że nie trzeba wykonywać iteracji o indeksach wyższych niż bieżący. Metoda `Stop()` informuje, że nie trzeba wykonywać żadnych dalszych iteracji.

Załóżmy na przykład, że pętla `Parallel.For()` wykonuje równolegle 10 iteracji. Niektóre z nich działają szybciej niż inne, a program szeregujący nie gwarantuje kolejności wykonywania iteracji. Przyjmijmy, że 1. iteracja została ukończona, iteracje 3., 5., 7. i 9. są przetwarzane w czterech różnych wątkach, a w iteracjach 5. i 7. wywołana została metoda `Break()`. W tym scenariuszu iteracje 6. i 8. nigdy nie zostaną uruchomione, natomiast iteracje 2. i 4. zostaną zaszeregowane do wykonania. Iteracje 3. i 9. zostaną wykonane do końca, ponieważ zostały już rozpoczęte w momencie zażądania przerwania pracy.

Metody `Parallel.For()` i `Parallel.ForEach<T>()` zwracają referencję do obiektu typu `ParallelLoopResult`, który zawiera przydatne informacje na temat tego, co stało się w trakcie pracy pętli. Ten obiekt z wynikiem udostępnia następujące właściwości:

- Właściwość `IsCompleted`, zwracającą wartość logiczną określającą, czy wszystkie iteracje rozpoczęły działanie.
- Właściwość `LowestBreakIteration`, określającą najwcześniejszą iterację, w której zażądano przerwania pracy. Ta właściwość zwraca wartość typu `long?`, gdzie wartość `null` oznacza, że w ogóle nie zażądano przerwania pętli.

Wróćmy do przykładu z 10 iteracjami. Właściwość `IsCompleted` zwróci w nim wartość `false`, a właściwość `LowestBreakIteration` będzie miała wartość 5.

4.0

Równoległe wykonywanie kwerend LINQ

Podobnie jak można wykonywać równolegle pętle za pomocą metody `Parallel.For()`, tak możliwe jest równoległe wykonywanie kwerend LINQ za pomocą interfejsu API PLINQ (ang. *Parallel LINQ*). Przykładowe proste wyrażenie LINQ bez przetwarzania równoległego przedstawiono na listingu 21.6. Na listingu 21.7 to wyrażenie zostało zmodyfikowane w taki sposób, by działało równolegle.

Listing 21.6. Metoda Select() w technologii LINQ

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

class Cryptographer
{
    // ...
    public List<string> Encrypt(IEnumerable<string> data)
    {
        return data.Select(
            item => Encrypt(item)).ToList();
    }
    // ...
}
```

Na listingu 21.6 kwerenda LINQ wykorzystuje standardowy operator kwerend `Select()`, by zaszyfrować każdy łańcuch znaków z sekwencji. Wynikowa sekwencja jest przekształcana w listę. Wygląda to na operację „zawstydzającą” współbieżną. Każde szyfrowanie to zależna od procesora operacja o dużej latencji, którą można przekazać do wątku roboczego z innego procesora.

Na listingu 21.7 pokazano, jak zmodyfikować kod z listingu 21.6, tak by kod szyfруjący łańcuchy znaków był wykonywany równolegle.

Listing 21.7. Równolegle wykonywana metoda Select() z technologii LINQ

```
using System.Linq;

class Cryptographer
{
    // ...
    public List<string> Encrypt (IEnumerable<string> data)
    {
        return data.AsParallel().Select(
            item => Encrypt(item)).ToList();
    }
    // ...
}
```

4.0

Na listingu 21.7 pokazano, że zmiana potrzebna do dodania obsługi równoległości jest bardzo drobna! Wystarczy dodać standardowy operator kwerend `AsParallel()`, dostępny w klasie `System.Linq.ParallelEnumerable`. Ten operator to prosta metoda rozszerzająca, informująca środowisko uruchomieniowe o tym, że kwerendę można przetwarzać równolegle. Na komputerze z dostępnymi wieloma procesorami pozwala to znacznie skrócić łączny czas wykonywania kwerendy.

Klasa `System.Linq.ParallelEnumerable` (wprowadzono ją w platformie Microsoft .NET Framework 4.0, aby umożliwić stosowanie technologii PLINQ) obejmuje między innymi równoległe wersje operatorów kwerend dostępnych w klasie `System.Linq.Enumerable`. W ten sposób powstaje interfejs API, który pozwala poprawić wydajność wszystkich często używanych operatorów kwerend, w tym tych używanych do sortowania, filtrowania (`Where()`), projekcji (`Select()`), złączania, grupowania i agregowania danych. Na listingu 21.8 pokazano, jak równolegle posortować dane.

Listing 21.8. Równoległe kwerendy LINQ oparte na standardowych operatorach kwerend

```
// ...
OrderedParallelQuery<string> parallelGroups =
    data.AsParallel().OrderBy(item => item);

// Dowodzi, że łączna liczba elementów nadal
// jest taka sama jak na początku.
System.Diagnostics.Trace.Assert(
    data.Count == parallelGroups.Sum(
        item => item.Count()));

// ...
```

Na listingu 21.8 pokazano, że wywołanie równoległej wersji wymaga tylko użycia metody rozszerzającej `AsParallel()`. Zauważ, że typ wyniku zwracanego przez równoległe standar-dowe operatory kwerend to `ParallelQuery<T>` lub `OrderedParallelQuery<T>`. Oba te typy informują kompilator o tym, że powinien stosować równoległe wersje standardowych operato-rów kwerend.

Ponieważ wyrażenie z kwerendą to tylko składniowe uproszczenie zapisu kwerend w postaci wywołań metod (takich jak na listingach 21.5 i 21.6), metodę `AsParallel()` można zastosowa-ć także do takiego wyrażenia. Na listingu 21.9 pokazano przykład ilustrujący równoległe grupowanie danych za pomocą wyrażenia z kwerendą.

Listing 21.9. Równoległa kwerenda LINQ w postaci wyrażenia z kwerendą

```
// ...
ParallelQuery<IGrouping<char, string>> parallelGroups;
parallelGroups =
    from text in data.AsParallel()
    orderby text
    group text by text[0];

// Dowodzi, że łączna liczba elementów nadal
// pozostaje taka sama jak na początku.
System.Diagnostics.Trace.Assert(
    data.Count == parallelGroups.Sum(
        item => item.Count()));

// ...
```

W poprzednich przykładach pokazano, że przekształcenie kwerendy lub pętli w wersję równoległą jest proste. Należy jednak pamiętać o pewnym zastrzeżeniu — trzeba uważać, by nie umożliwiał wielu wątkom nieprawidłowego dostępu do tej samej pamięci i modyfikowa-nia jej (szczegółowe omówienie tego zagadnienia znajdziesz w rozdziale 22.). Jeśli o tym zapomnisz, może wystąpić sytuacja wyścigu.

Wcześniej w rozdziale pokazano, że metody `Parallel.For()` i `Parallel.ForEach<T>()` zbie-rają wyjątki zgłoszone w trakcie równolegle wykonywanych iteracji, a następnie zgłaszą jeden zagregowany wyjątek zawierający wszystkie pierwotne wyjątki. W technologii PLINQ wygląda to podobnie. Równoległe operacje też mogą tu zgłaszać wiele wyjątków. Wynika to z tego samego powodu — gdy kod kwerendy jest równolegle wykonywany dla każdego elementu, za każdym razem może niezależnie zgłosić wyjątek. Nie jest zaskoczeniem, że technologia

PLINQ zarządza tą sytuacją w dokładnie ten sam sposób co pętle równoległe i biblioteka TPL — wyjątki zgłoszone w trakcie równoległego przetwarzania kwerend są dostępne we właściwości InnerExceptions wyjątku AggregateException. Dlatego umieszczenie kwerendy PLINQ w bloku try-catch powiązanym z wyjątkiem System.AggregateException pozwala z powodzeniem obsłużyć nieobsłużone wyjątki z wszystkich iteracji.

Anulowanie kwerendy PLINQ

Wzorzec polegający na żądaniach anulowania jest dostępny także dla kwerend PLINQ. Przedstawiono to na listingu 21.10 (wynik działania kodu znajdziesz w danych wyjściowych 21.3). Anulowane kwerendy PLINQ, podobnie jak pętle równoległe, zgłaszą wyjątek System.OperationCanceledException. Ponadto kwerendy PLINQ są wykonywane synchronicznie w wątku, w którym je wywołano (pod tym względem też przypominają pętle równoległe). Dlatego często stosowaną techniką jest umieszczanie kwerend równoległych w zadaniu uruchamianym w innym wątku, który w razie potrzeby można anulować z poziomu bieżącego wątku (podobne rozwiążanie zastosowano na listingu 21.5).

Listing 21.10. Anulowanie kwerendy PLINQ

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;

public class Program
{
    public static List<string> ParallelEncrypt(
        List<string> data,
        CancellationToken cancellationToken)
    {
        int goverer = 0;
        return data.AsParallel().WithCancellation(
            cancellationToken).Select(
                (item) =>
            {
                if (Interlocked.CompareExchange(
                    ref goverer, 0, 100) % 100 == 0)
                {
                    Console.Write('.');
                }
                Interlocked.Increment(ref goverer);
                return Encrypt(item);
            }).ToList();
    }

    public static async Task Main()
    {
        ConsoleColor originalColor = Console.ForegroundColor;
        List<string> data = Utility.GetData(100000).ToList();

        using CancellationTokenSource cts =

```

```
new CancellationTokenSource();

Task task = Task.Run(() =>
{
    data = ParallelEncrypt(data, cts.Token);
}, cts.Token);

Console.WriteLine("Wciśnij dowolny klawisz, aby zakończyć.");
Task<int> cancelTask = ConsoleReadAsync(cts.Token);

try
{
    Task.WaitAny(task, cancelTask);
    // Anulowanie nieukończonych zadań.
    cts.Cancel();
    await task;

    Console.ForegroundColor = ConsoleColor.Green;
    Console.WriteLine("\nUkończono z powodzeniem");
}
catch (OperationCanceledException taskCanceledException)
{
    Console.ForegroundColor = ConsoleColor.Red;
    Console.WriteLine(
        $"\nAnulowano: { taskCanceledException.Message }");
}
finally
{
    Console.ForegroundColor = originalColor;
}
}

private static async Task<int> ConsoleReadAsync(
    CancellationToken cancellationToken = default)
{
    return await Task.Run(async () =>
    {
        const int maxDelay = 1025;
        int delay = 0;
        while (!cancellationToken.IsCancellationRequested)
        {
            if (Console.KeyAvailable)
            {
                return Console.Read();
            }
            else
            {
                await Task.Delay(delay, cancellationToken);
                if (delay < maxDelay) delay *= 2 + 1;
            }
        }
        cancellationToken.ThrowIfCancellationRequested();
        throw new InvalidOperationException(
            "Wcześniej wiersz powinien zapobiegać wyświetleniu tego tekstu");
    }, cancellationToken);
}

private static string Encrypt(string item)
```

```
{
    Cryptographer cryptographer = new Cryptographer();
    return System.Text.Encoding.UTF8.GetString(cryptographer.Encrypt(item));
}
```

DANE WYJŚCIOWE 21.3.

Wciśnij dowolny klawisz, aby zakończyć.

.....
.....
.....
.....

Anulowano: The query has been canceled via the token supplied to WithCancellation.

Anulowanie kwerendy PLINQ (podobnie jak pętli równoległej lub zadania) wymaga użycia obiektu typu `CancellationToken`, dostępnego w obiekcie typu `CancellationTokenSource`. Zamiast przeciąjać każdą kwerendę PLINQ, by obsługiwała tokeny anulowania, zastosowano inne podejście. Obiekt typu `ParallelQuery<T>` zwracany przez metodę `AsParallel()` z interfejsu `IEnumerable` obejmuje metodę rozszerzającą `WithCancellation()`, która przyjmuje obiekt typu `CancellationToken`. Dlatego wywołanie metody `Cancel()` dla obiektu typu `CancellationTokenSource` powoduje zgłoszenie żądania anulowania równoległej kwerendy (ponieważ prowadzi to do sprawdzenia wartości właściwości `IsCancellationRequested` obiektu typu `CancellationToken`).

WCześniej wspomniano już, że anulowanie kwerendy PLINQ prowadzi do zgłoszenia wyjątku. Kwerenda nie zwraca wtedy kompletnego wyniku. Często stosowana technika zarządzania kwerendami PLINQ, które mogą zostać anulowane, polega na umieszczaniu ich w bloku `try` i przechwytywaniu wyjątków `OperationCanceledException`. Druga popularna technika, zastosowana na listingu 21.10, to przekazanie obiektu typu `CancellationToken` zarówno do metody `ParallelEncrypt()`, jak i jako drugiego parametru metody `Run()`. To powoduje, że metoda `task.Wait()` zgłosi wyjątek typu `AggregateException`, w którym właściwość `InnerException` będzie zawierała wyjątek `TaskCanceledException`. Wyjątek ze zagregowanego obiektu można następnie przechwycić (podobnie jak inne wyjątki z równoległych operacji).

Koniec
4.0

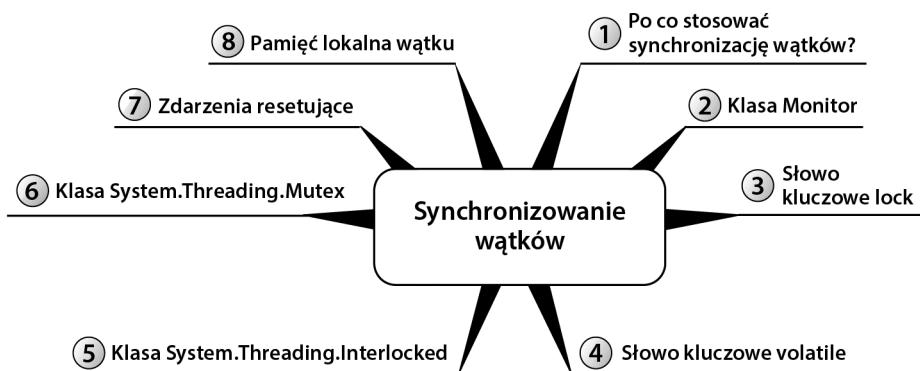
Podsumowanie

W tym rozdziale opisano, jak używać klasy `Parallel` z biblioteki TPL do iterowania za pomocą instrukcji `for` i `foreach`. Zobaczyłeś też, jak za pomocą metody rozszerzającej `AsParallel()` z przestrzeni nazw `System.Linq` równolegle wykonywać kwerendy LINQ. Łatwość, z jaką można wykonywać omawiane operacje, powinna pomóc Ci zrozumieć, że równoległe iteracje są jednym z najłatwiejszych sposobów na równoległe wykonywanie kodu. Nadal trzeba uważać na możliwe warunki wyścigu i zakleszczenie, jednak takie problemy są znacznie rzadsze, gdy stosujesz pętle równoległe i nie współdzielisz danych, niż gdy bezpośrednio używasz zadań. Wystarczy tylko zidentyfikować bloki kodu zależne od procesora, w których równoległe wykonywanie kodu przyniesie korzyści.

22

Synchronizowanie wątków

ROZDZIAŁE 21. OMÓWIONO szczegóły programowania wielowątkowego opartego na bibliotece TPL i technologii PLINQ. Nie wyjaśniono tam jednak synchronizowania pracy wątków, tak by zapobiegać sytuacji wyścigu, a jednocześnie uniknąć zakleszczenia. Synchronizowanie wątków jest tematem tego rozdziału.



Na początku przedstawiono przykładowy program wielowątkowy bez synchronizacji korzystania ze współużytkowanych danych przez wątki. W efekcie występuje sytuacja wyścigu, a integralność danych zostaje utracona. To omówienie ma pokazać, dlaczego należy synchronizować wątki. Dalej opisano liczne związane z tym mechanizmy i najlepsze praktyki.

We wcześniejszych wydaniach tej książki znajdował się duży fragment poświęcony dodatkowym wzorcom wielowątkowym, a także punkt dotyczący różnych mechanizmów wywołań zwrotnych opartych na zegarze. Jednak po wprowadzeniu wzorca opartego na słowach kluczowych `async` i `await` dawne techniki zostały zastąpione nowszymi (chyba że używasz języka starszego niż C# 5.0 i platformy .NET wcześniejszej niż 4.5). Materiały dotyczące wersji starszych niż C# 5.0 są jednak wciąż dostępne w poświęconej tej książce witrynie — <http://www.IntelliTect.com/EssentialCSharp>.

W tym rozdziale używana jest biblioteka TPL, dlatego przykładowy kod nie skompiluje się w wersjach platformy .NET starszych niż 4. Jednak jedyny powód, dla którego trzeba używać wersji 4 platformy, to korzystanie z klasy `System.Threading.Tasks.Task` do wykonywania operacji asynchronicznych (dotyczy to miejsc, w których nie napisano bezpośrednio, że dany interfejs API pochodzi z platformy Microsoft .NET Framework 4). Zmodyfikowanie kodu w taki sposób, by tworzył obiekt typu `System.Threading.Thread` i wywoływał instrukcję `Thread.Join()` w celu oczekiwania na wykonanie wątku, pozwala skompilować większość przykładów także w starszych wersjach platformy .NET.

Interfejsem API używanym w tym rozdziale do uruchamiania zadań jest pochodząca z platformy .NET 4.5 metoda `System.Threading.Tasks.Task.Run()`. W rozdziale 19. wyjaśniono, że warto stosować ją zamiast metody `System.Threading.Tasks.Task.Factory.StartNew()`, ponieważ jest prostsza i w wielu sytuacjach wystarczająca. Osoby, które muszą korzystać z wersji .NET 4, mogą zastąpić wywołania `Task.Run()` instrukcjami `Task.Factory.StartNew()` bez wprowadzania żadnych dodatkowych zmian. Dlatego w tym rozdziale kod, w którym z nowych technik używana jest tylko metoda `Run()`, nie jest oznaczany jako specyficzny dla platformy .NET 4.5.

Po co stosować synchronizację?

Uruchamianie nowego wątku to stosunkowo proste zadanie programistyczne. Tym, co sprawia, że programowanie wielowątkowe jest trudne, jest określanie, do których danych wiele wątków może jednocześnie uzyskać dostęp. Program musi synchronizować korzystanie z niektórych danych, aby zapobiegać jednoczesnemu dostępowi do nich i zapewniać w ten sposób bezpieczeństwo. Przyjrzyj się kodowi z listingu 22.1.

Listing 22.1. Zarządzanie stanem bez synchronizacji

```
using System;
using System.Threading.Tasks;

public class Program
{
    const int _Total = int.MaxValue;
    static long _Count = 0;

    public static void Main()
    {
        // W wersji .NET 4.0 zastosuj metodę Task.Factory.StartNew.
        Task task = Task.Run(() => Decrement());

        // Inkrementacja.
        for(int i = 0; i < _Total; i++)
        {
            _Count++;
        }

        task.Wait();
        Console.WriteLine("Count = {0}", _Count);
    }
}
```

```
static void Decrement()
{
    // Dekrementacja.
    for(int i = 0; i < _Total; i++)
    {
        _Count--;
    }
}
```

W danych wyjściowych 22.1 pokazano możliwy efekt wykonania kodu z listingu 22.1.

DANE WYJŚCIOWE 22.1.

Count = 113449949

Ważną rzeczą na listingu 22.1 jest to, że dane wyjściowe są różne od 0. Bezpośrednie (sekwencyjne) wywołania metody Decrement() dałyby właśnie 0. Jednak gdy ta metoda jest wywoływana asynchronicznie, występuje sytuacja wyścigu, ponieważ poszczególne kroki w instrukcjach `_Count++` i `_Count--` przepłatają się ze sobą. W zagadnieniu dla początkujących „Słownictwo związane z wielowątkowością” w rozdziale 19. wyjaśniono, że pojedyncze instrukcje w języku C# często obejmują wiele kroków. Przyjrzyj się opisanemu w tabeli 22.1 przykładowemu wykonaniu omawianego kodu.

Tabela 22.1. Przykładowe wykonanie programu opisane za pomocą pseudokodu

Wątek główny	Wątek z dekrementacją	Liczba
...
Kopiowanie wartości 0 z pola <code>_Count</code> .		0
Inkrementacja skopowanej wartości (0), co daje 1.		0
Kopiowanie wynikowej wartości (1) do pola <code>_Count</code> .		1
Kopiowanie wartości 1 z pola <code>_Count</code> .	Kopiowanie wartości 1 z pola <code>_Count</code> .	1
Inkrementacja skopowanej wartości (1), co daje 2.		1
Kopiowanie wynikowej wartości (2) do pola <code>_Count</code> .	Dekrementacja skopowanej wartości (1), co daje 0.	2
	Kopiowanie wynikowej wartości (0) do pola <code>_Count</code> .	0
...

W tabeli 22.1 równoległe wykonywanie kodu (przełączanie kontekstu wątków) przedstawiono za pomocą instrukcji umieszczonych w jednej lub drugiej kolumnie. Wartość pola `_Count` po wykonaniu każdego wiersza znajduje się w ostatniej kolumnie. W tym przykładowym fragmencie polecenie `_Count++` jest wykonywane dwukrotnie, natomiast polecenie `_Count--` tylko raz. Jednak końcowa wartość pola `_Count` to 0, a nie 1. Skopiowanie wyniku z powrotem do pola `_Count` skutkuje zignorowaniem zmian wartości tego pola wprowadzonych od momentu wczytania tego pola przez wątek, który kopiuje wynik.

Problemem na listingu 22.1 jest sytuacja wyścigu związana z jednoczesnym dostępem wielu wątków do tych samych danych. Przykładowy fragment wykonania programu pokazuje, że umożliwienie wielu wątkom dostępu do tych samych danych prawdopodobnie spowoduje naruszenie integralności danych (nawet na komputerze jednoprocesorowym). Aby zapobiec temu problemowi, w kodzie trzeba zapewnić synchronizację dostępu do danych. Dane (lub kod) z synchronizacją jednoczesnego dostępu ze strony wielu wątków są **bezpieczne ze względu na wątki**.

Należy zwrócić uwagę na jeden ważny punkt dotyczący atomowości operacji odczytu i zapisu wartości zmiennych. Środowisko uruchomieniowe gwarantuje, że wartości typów, których pojemność jest nie większa niż natywnych liczb całkowitych (jest to wielkość wskaźnika), nie będą wczytywane lub zapisywane częściowo. Tak więc w 64-bitowym systemie operacyjnym odczyt i zapis wartości typu `long` (zajmują one 64 bity) będzie odbywał się atomowo. Jednak odczyt i zapis zmiennych 128-bitowych (na przykład typu `decimal`) może nie być atomowy. Dlatego operacja modyfikująca zmienną typu `decimal` może zostać przerwana po skopiowaniu tylko 32 bitów, co skutkuje odczytem nieprawidłowej wartości (ten proces jest czasem nazywany *torn read*, czyli **odczytem przerwanym**).

■ ZAGADNIENIE DLA POCZĄTKUJĄCYCH

Wiele wątków i zmienne lokalne

Zauważ, że zmienne lokalne nie wymagają synchronizowania. Zmienne lokalne są wczytywane na stos, a każdy wątek korzysta z własnego stosu logicznego. Dlatego w każdym wywołaniu metody tworzona jest odrębna instancja każdej zmiennej lokalnej. Zmienne lokalne domyślnie nie są współużytkowane w wywołaniach metod. Nie są także współużytkowane w różnych wątkach.

To jednak nie oznacza, że zmienne lokalne nie powodują żadnych problemów z równoległością. W końcu kod może udostępniać zmienną lokalną wielu wątkom¹. Równoległa pętla `for`, w której zmienna lokalna jest współużytkowana w iteracjach, umożliwia jednoczesny dostęp do tej zmiennej, co grozi sytuacją wyścigu (zobacz listing 22.2).

Listing 22.2. Zmienne lokalne bez synchronizacji dostępu

```
using System;
using System.Threading.Tasks;

public class Program
```

¹ Choć na poziomie języka C# używana jest zmienna lokalna, w kodzie IL odpowiada jej pole, a dostęp do pola można uzyskać w wielu wątkach.

```

{
    public static void Main()
    {
        int x = 0;
        Parallel.For(0, int.MaxValue, i =>
        {
            x++;
            x--;
        });
        Console.WriteLine("Count = {0}", x);
    }
}

```

W tym przykładzie zmienna lokalna `x` jest używana w równoległej pętli `for`. Dlatego wiele wątków może jednocześnie modyfikować tę zmienną, co prowadzi do sytuacji wyścigu (bardzo podobnej do tej z listingu 22.1). Wartość końcowa zmiennej `x` prawdopodobnie nie będzie równa 0, choć jest zwiększana i zmniejszana tą samą liczbą razy.

Synchronizacja z wykorzystaniem klasy Monitor

Początek
4.0

Aby zapewnić synchronizację kilku wątków, tak by nie mogły jednocześnie wykonywać określonej sekcji kodu, można zastosować **monitor**. Pozwala on zablokować drugi wątek przed rozpoczęciem wykonywania chronionej sekcji kodu, dopóki pierwszy wątek nie wyjdzie z tej sekcji. Klasa `System.Threading.Monitor` zapewnia między innymi obsługę tego mechanizmu. Do oznaczania początku i końca chronionej sekcji kodu służą wywołania statycznych metod `Monitor.Enter()` i `Monitor.Exit()`.

Na listingu 22.3 pokazano jawną synchronizację z wykorzystaniem klasy `Monitor`. Ważne jest, by cały kod między wywołaniami `Monitor.Enter()` i `Monitor.Exit()` znajdował się w bloku `try-finally` (tak jak na listingu). Jeśli nie dodasz tego bloku, w sekcji chronionej może wystąpić wyjątek. W takiej sytuacji metoda `Monitor.Exit()` może nigdy nie zostać wywołana, co spowoduje trwałe zablokowanie innych wątków.

Listing 22.3. Jawną synchronizację z wykorzystaniem monitora

```

using System;
using System.Threading;
using System.Threading.Tasks;

public class Program
{
    readonly static object _Sync = new object();
    const int _Total = int.MaxValue;
    static long _Count = 0;

    public static void Main()
    {
        // W wersji .NET 4.0 zastosuj metodę Task.Factory.StartNew.
        Task task = Task.Run(() => Decrement());

        // Inkrementacja.
        for(int i = 0; i < _Total; i++)
    }
}

```

```

    {
        bool lockTaken = false;
        try
        {
            Monitor.Enter(_Sync, ref lockTaken);
            _Count++;
        }
        finally
        {
            if (lockTaken)
            {
                Monitor.Exit(_Sync);
            }
        }
    }

    task.Wait();
    Console.WriteLine($"Count = {_Count}");
}

static void Decrement()
{
    for(int i = 0; i < _Total; i++)
    {
        bool lockTaken = false;
        try
        {
            Monitor.Enter(_Sync, ref lockTaken);
            _Count--;
        }
        finally
        {
            if (lockTaken)
            {
                Monitor.Exit(_Sync);
            }
        }
    }
}

```

4.0

Efekt działania kodu z listingu 22.3 pokazano w danych wyjściowych 22.2.

DANE WYJŚCIOWE 22.2.

```
Count = 0
```

Zauważ, że wywołania `Monitor.Enter()` i `Monitor.Exit()` są powiązane ze sobą, ponieważ jako parametr przekazywana jest do nich ta sama referencja (`_Sync`). Wersja przeciążonej metody `Monitor.Enter()` przyjmująca parametr `lockTaken` została dodana do platformy .NET dopiero w wersji 4.0. Wcześniej parametr `lockTaken` nie był dostępny, dlatego nie istniała technika gwarantująca przechwycenie wyjątku, który wystąpił między wywołaniem `Monitor.Enter()` a blokiem `try`. Umieszczenie bloku `try` bezpośrednio po wywołaniu `Monitor.Enter()` było dobrym rozwiązaniem w kodzie produkcyjnym, ponieważ mechanizm

JIT zapobiegał wystąpieniu asynchronicznych wyjątków. Jednak jeśli bezpośrednio po instrukcji `Monitor.Enter()` znajdował się inny kod (nie blok `try`), to nawet gdy były to instrukcje, które kompilator dodał w kodzie diagnostycznym, mechanizm JIT mógł mieć trudności z poprawnym zwróceniem sterowania w bloku `try`. Dlatego wystąpienie wyjątku mogło skutkować wyciekaniem blokad (czyli zajęciem ich na trwale) zamiast wykonaniem bloku `finally` i zwolnieniem blokady. To z kolei mogło prowadzić do zakleszczenia, w sytuacji gdy inny wątek próbował zająć blokadę. Dlatego w wersjach platformy .NET starszych niż 4.0 po instrukcji `Monitor.Enter()` zawsze należy umieszczać blok `try-finally` z instrukcją `{Monitor.Exit(_Sync)}`.

Klasa `Monitor` udostępnia też metodę `Pulse()`, umożliwiającą wątkowi zajęcie miejsca w kolejce gotowych wątków. W ten sposób wątek informuje, że jest gotowy do pracy. Jest to często stosowany mechanizm synchronizacji we wzorcu producent-konsument, chroniący przed uruchomieniem konsumpcji, jeśli nie zakończono jeszcze produkcji. Wątek producenta, który zajął monitor (w wyniku wywołania `Monitor.Enter()`), może wywołać metodę `Monitor.Pulse()`, by poinformować wątek konsumenta (który też mógł już wywołać metodę `Monitor.Enter()`) o tym, że dostępny jest element przeznaczony do „konsumpcji”, dlatego wątek konsumenta powinien przygotować się do pracy. Po jednym wywołaniu `Pulse()` tylko jeden wątek (tu jest nim wątek konsumenta) może zająć miejsce w kolejce gotowych wątków. Gdy wątek producenta wywoła metodę `Monitor.Exit()`, wątek konsumenta zajmie blokadę (wykonane zostanie wywołanie `Monitor.Enter()`) i wejdzie do sekcji krytycznej, by rozpocząć „konsumpcję” elementu. Gdy konsument zakończy przetwarzanie oczekującego elementu, wywoła metodę `Exit()`, co umożliwia producentowi (zablokowanemu w wyniku wywołania metody `Monitor.Enter()`) produkcję następnego elementu. W tym przykładzie w kolejce gotowych wątków może się znajdować tylko jeden wątek. To gwarantuje, że nie nastąpi „konsumpcja” bez „produkcji” i na odwrót.

Koniec
4.0

Stosowanie słowa kluczowego lock

Ponieważ w kodzie wielowątkowym często trzeba synchronizować prace za pomocą klasy `Monitor`, a ponadto łatwo zapomnieć o bloku `try-finally`, język C# udostępnia specjalne słowo kluczowe do zarządzania wzorcem synchronizacji opartej na blokadach. Na listingu 22.4 pokazano, jak stosować słowo kluczowe `lock`, a w danych wyjściowych 22.3 znajdziesz wynik działania kodu.

Listing 22.4. Synchronizacja z wykorzystaniem słowa kluczowego lock

```
using System;
using System.Threading;
using System.Threading.Tasks;

public class Program
{
    readonly static object _Sync = new object();
    const int _Total = int.MaxValue;
    static long _Count = 0;
```

```
public static void Main()
{
    // W wersji 4.0 platformy .NET zastosuj metodę Task.Factory.StartNew.
    Task task = Task.Run(() => Decrement());

    // Inkrementacja.
    for (int i = 0; i < _Total; i++)
    {
        lock(_Sync)
        {
            _Count++;
        }
    }

    task.Wait();
    Console.WriteLine($"Count = {_Count}");
}

static void Decrement()
{
    for (int i = 0; i < _Total; i++)
    {
        lock(_Sync)
        {
            _Count--;
        }
    }
}
```

DANE WYJŚCIOWE 22.3.

```
Count = 0
```

Dzięki dodaniu (za pomocą instrukcji `lock` lub `Monitor`) blokady do sekcji kodu z dostępem do pola `_Count` metody `Main()` i `Decrement()` są bezpieczne ze względu na wątki. To oznacza, że można je bezpiecznie wywoływać jednocześnie w wielu wątkach. W wersjach starszych niż C# 4.0 technika ta wyglądała podobnie, jednak w kodzie wygenerowanym przez kompilator korzystano z metody `Monitor.Enter()` bez parametru `lockTaken` (ta metoda musiała się znajdować przed blokiem `try`).

Ceną za synchronizację jest spadek wydajności. Kod z listingu 22.4 działa o rząd wielkości wolniej niż kod z listingu 22.1. To dowodzi, że wykonywanie polecenia `lock` trwa dłużej w porównaniu do operacji inkrementacji i dekrementacji licznika.

Nawet gdy czas wykonywania polecenia `lock` jest niewielki w porównaniu z synchronizowaną operacją, programiści powinni unikać bezrefleksyjnego stosowania synchronizacji. Należy chronić się przed możliwością wystąpienia zakleszczenia i niepotrzebnym synchronizowaniem prac w komputerach wieloprocesorowych, gdy możliwe jest równoległe wykonywanie kodu. Zgodnie z ogólnym zaleceniem z obszaru projektowania obiektów należy synchronizować *zmienisty stan statyczny*, a nie dane instancji. Nie trzeba synchronizować dostępu do danych, które nigdy się nie zmieniają. Programiści, którzy umożliwiają wielu

wątkom dostęp do jednego obiektu, muszą zapewnić synchronizację tego dostępu. W każdej klasie, w której bezpośrednio używane są wątki, warto zapewnić bezpieczeństwo instancji ze względu na wątki.

Początek
7.1

ZAGADNIENIE DLA POCZĄTKUJĄCYCH

Zwracanie obiektu typu Task bez operatora await

Na listingu 22.1 instrukcja `Task.Run(() => Decrement())` zwraca obiekt typu Task, jednak operator await nie jest tam używany. Wynika to stąd, że do wersji C# 7.1 nie można było stosować modyfikatora `async` do metody `Main()`. W wersji C# 7.1 kod można zmodyfikować i użyć wzorca ze słowami kluczowymi `async` i `await`, co ilustruje listing 22.5.

Listing 22.5. Asynchroniczna metoda Main() w języku C# 7.1

```
using System;
using System.Threading.Tasks;

public class Program
{
    readonly static object _Sync = new object();
    const int _Total = int.MaxValue;
    static long _Count = 0;

    public static async Task Main()
    {
        // W wersji .NET 4.0 należy użyć metody Task.Factory.StartNew.
        Task task = Task.Run(() => Decrement());

        // Inkrementacja.
        for(int i = 0; i < _Total; i++)
        {
            lock(_Sync)
            {
                _Count++;
            }
        }
        await task;
        Console.WriteLine($"Count = {_Count}");
    }

    static void Decrement()
    {
        for(int i = 0; i < _Total; i++)
        {
            lock(_Sync)
            {
                _Count--;
            }
        }
    }
}
```

Koniec
7.1

Wybieranie obiektu reprezentującego blokadę

Niezależnie od tego, czy programista stosuje słowo kluczowe `lock`, czy bezpośrednio używa klasy `Monitor`, bardzo ważne jest, by starannie wybrał obiekt reprezentujący blokadę.

W przedstawionych wcześniej przykładach zmienna synchronizacyjna, `_Sync`, jest zadeklarowana jako prywatna i przeznaczona tylko do odczytu. Zadeklarowanie jej jako tylko do odczytu ma zapewniać, że wartość zmiennej nie zmieni się między wywołaniami `Monitor.Enter()` i `Monitor.Exit()`. To pozwala zachować spójność w operacjach wchodzenia do bloku z synchronizacją i wychodzenia z niego. Ponadto zmienna `_Sync` jest zadeklarowana jako prywatna, aby żaden blok synchronizacji spoza danej klasy nie mógł używać tej samej zmiennej i zablokować przez to pracy bloku z tej klasy.

Jeśli dane są publiczne, obiekt używany do synchronizacji może być publiczny. Wtedy inne klasy mogą synchronizować swoją pracę z wykorzystaniem tego samego obiektu. To jednak zwiększa ryzyko wystąpienia zakleszczenia. Na szczęście to rozwiązanie rzadko jest potrzebne. Gdy dane są publiczne, lepiej umieścić zarządzanie synchronizacją poza klasą i umożliwić zajmowanie blokad w kodzie wywołującym z wykorzystaniem odrębnego obiektu używanego do synchronizacji.

Ważne jest, by obiekt używany do synchronizacji nie był typu bezpośredniego. Jeśli słowo kluczowe `lock` zostanie zastosowane do wartości typu bezpośredniego, kompilator zgłosi błąd. W przypadku jawnego zastosowania klasy `System.Threading.Monitor` (nie w wyniku użycia słowa kluczowego `lock`) taki błąd na etapie komplikacji się nie pojawi. Jednak kod zgłosi wyjątek w momencie wywołania metody `Monitor.Exit()`, informując, że nie istnieje odpowiednie wywołanie `Monitor.Enter()`. Problem polega na tym, że gdy używany jest typ bezpośredni, środowisko uruchomieniowe tworzy kopię wartości, umieszcza ją na stercie (co związane jest z opakowywaniem wartości) i przekazuje opakowaną wartość do metody `Monitor.Enter()`. Metoda `Monitor.Exit()` też otrzymuje opakowaną kopię pierwotnej zmiennej. To oznacza, że metody `Monitor.Enter()` i `Monitor.Exit()` otrzymują inne obiekty używane do synchronizacji, dlatego nie można powiązać obu wywołań.

Dlaczego należy unikać blokad opartych na słowie kluczowym `this`, wyrażeniu `typeof(typ)` lub typie string?

Często stosowany wzorzec polega na dodawaniu blokad z użyciem słowa kluczowego `this` dla danych instancji i z użyciem nazwy typu pobranej z wyrażenia `typeof(typ)` (na przykład `typeof(MojTyp)`) dla danych statycznych. Ten wzorzec zapewnia docelowy obiekt synchronizacji (ang. *synchronization target*) dla wszystkich stanów powiązanych z konkretnym obiektem (gdy używane jest słowo kluczowe `this`) oraz dla wszystkich danych statycznych określonego typu (gdy stosowane jest wyrażenie `typeof(typ)`). Problem polega na tym, że docelowy obiekt synchronizacji wskazywany za pomocą słowa kluczowego `this` (lub wyrażenia `typeof(type)`) może być używany także w zupełnie innym synchronizowanym bloku utworzonym w niepowiązanym fragmencie kodu. To oznacza, że choć tylko kod w danym obiekcie może blokować prace za pomocą słowa kluczowego `this`, jednostka wywołującą, która utworzyła ten obiekt, może przekazać go do blokady.

W takiej sytuacji dwa różne synchronizowane bloki używane do synchronizowania dostępu do dwóch zupełnie innych zbiorów danych mogą się wzajemnie zablokować. Ponadto (choć jest to mało prawdopodobne) współużytkowanie docelowego obiektu synchronizacji może mieć niepożądany wpływ na wydajność, a w skrajnych przypadkach prowadzić nawet do zakleszczenia. Dlatego zamiast stosować w blokadach słowo kluczowe `this` lub nawet wyrażenie `typeof(type)`, lepiej zdefiniować prywatne pole tylko do odczytu, używane w blokadach wyłącznie przez klasę, która ma do niego dostęp.

Inny typ blokad, którego warto unikać, to `string`. Wynika to z mechanizmu internowania łańcuchów znaków (ang. *string interning*). Jeśli ta sama stała typu `string` występuje w wielu miejscach, prawdopodobne jest, że wszystkie te lokalizacje będą korzystały z tego samego obiektu. Wtedy zakres blokady staje się nieoczekiwanie duży.

Dlatego jako docelowy obiekt blokady należy stosować obiekt typu `object` utworzony dla określonego kontekstu synchronizacji.

Wskazówki

UNIKAJ tworzenia blokad opartych na słowie kluczowym `this`, wyrażeniu `typeof()` lub wartościach typu `string`.

Jako docelowy obiekt synchronizacji **DEKLARUJ** przeznaczoną tylko do odczytu odrębną zmienną synchronizacyjną typu `object`.

Unikaj synchronizacji z wykorzystaniem atrybutu `MethodImplAttribute`

Jednym z mechanizmów synchronizacji wprowadzonych w platformie .NET 1.0 był atrybut `MethodImplAttribute`. W połączeniu z parametrem `MethodImplOptions.Synchronized` ten atrybut powoduje oznaczenie metody jako synchronizowanej. Dzięki temu taką metodę w danym momencie może wykonywać tylko jeden wątek. Aby uzyskać ten efekt, kompilator JIT traktuje metodę w taki sposób, jakby znajdowała się w blokadzie `lock(this)` (lub, w przypadku metod statycznych, w blokadzie opartej na typie). To sprawia, że dana metoda i wszystkie inne metody z tej samej klasy opatrzone opisywanym atrybutem i odpowiednim parametrem typu wyliczeniowego są synchronizowane, choć każda metoda powinna być synchronizowana niezależnie. Oznacza to, że jeśli dwie metody danej klasy (lub większa ich liczba) są opatrzone omawianym atrybutem, w danym momencie wykonywana może być tylko jedna z nich. Wykonywana metoda blokuje wtedy wszystkie wywołania innych wątków skierowane do siebie, a także do innych metod klasy opatrzonych omawianym atrybutem. Ponadto ponieważ synchronizacja oparta jest na słowie kluczowym `this` (lub, co jeszcze gorsze, na typie), występują tu te same opisane wcześniej problemy co przy stosowaniu blokad `lock(this)` (gdy blokada dotyczy metod statycznych, sytuacja jest jeszcze gorsza). Dlatego najlepiej w ogóle unikać atrybutu `MethodImplAttribute`.

Wskazówka

UNIKAJ synchronizowania kodu za pomocą atrybutu `MethodImplAttribute`.

Deklarowanie pól jako zmiennych (volatile)

W niektórych sytuacjach kompilator lub procesor mogą zoptymalizować kod w taki sposób, że instrukcje nie są wykonywane dokładnie w kolejności ich występowania, a niektóre polecenia są pomijane. Takie zmiany są nieszkodliwe, gdy kod działa w jednym wątku. Jednak gdy wątków jest wiele, optymalizacja może mieć nieoczekiwane konsekwencje, jeśli zmienia kolejność operacji odczytu lub zapisu wartości pola względem dostępu do tego samego pola w innych wątkach.

Jednym ze sposobów na zapewnienie stabilnej pracy kodu jest zadeklarowanie pola z użyciem słowa kluczowego `volatile`. To słowo kluczowe wymusza odczyt i zapis wartości pola dokładnie w określonych miejscach kodu, a nie w lokalizacjach wybranych w wyniku optymalizacji. Modyfikator `volatile` informuje, że pole może zostać zmodyfikowane przez sprzęt, system operacyjny lub inny wątek. To oznacza, że dane są zmienne. Słowo kluczowe `volatile` nakazuje więc kompilatorom i środowisku uruchomieniowemu precyzyjne zarządzanie takimi danymi. Więcej informacji znajdziesz pod adresem <http://bit.ly/CSharpReorderingOptimizations>.

Modyfikator `volatile` jest stosowany rzadko i wiąże się z komplikacjami, które mogą prowadzić do błędów w posługiwaniu się nim. Dlatego zalecamy korzystanie z instrukcji `lock` zamiast z modyfikatora `volatile`, chyba że jesteś przekonany, iż w danym miejscu użycie tego modyfikatora jest właściwe.

Stosowanie klasy System.Threading.Interlocked

Opisywany do tego miejsca wzorzec wzajemnego wykluczania związany jest z minimalnym zestawem narzędzi potrzebnych do obsługi synchronizacji w procesie (w domenie aplikacji). Jednak synchronizacja oparta na klasie `System.Threading.Monitor` jest stosunkowo kosztowna. Inne rozwiązanie obsługiwane przez procesor jest bezpośrednio dostosowane do konkretnych wzorców synchronizacji.

Kod z listingu 22.6 przypisuje do pola `_Data` nową wartość, jeśli wcześniejsza wartość to `null`. Jak wskazuje na to nazwa zastosowanej w kodzie metody, używany jest tu wzorzec porównania i zmiany (ang. *compare/exchange*). Nie musisz tu ręcznie dodawać blokady wokół kodu sprawdzającego i zmieniającego wartość, ponieważ metoda `Interlocked.CompareExchange()` zapewnia wbudowaną obsługę synchronicznej operacji obejmującej sprawdzanie wartości (tu jest nią `null`) i aktualizację pierwszego parametru, jeśli wartość jest równa drugiemu parametrowi. W tabeli 22.2 opisano inne metody synchronizacji dostępne w klasie `Interlocked`.

Listing 22.6. Synchronizacja z wykorzystaniem klasy `System.Threading.Interlocked`

```
public class SynchronizationUsingInterlocked
{
    private static object? _Data;

    // Inicjalizowanie danych, jeśli jeszcze nie przypisano do nich wartości.
    static void Initialize(object newValue)
    {
        // Jeśli pole _Data ma wartość null, należy przypisać do niego newValue.
        Interlocked.CompareExchange(
            ref _Data, newValue, null);
    }

    // ...
}
```

Tabela 22.2. Metody klasy Interlocked związane z synchronizacją

Sygnatura metody	Opis
<code>public static T CompareExchange<T>(T location, T value, T comparand)</code>	Sprawdza, czy w location zapisana jest wartość comparand. Jeśli tak jest, przypisuje do location wartość value i zwraca pierwotne dane zapisane w location.
<code>public static T Exchange<T>(T location, T value)</code>	Przypisuje do location wartość value i zwraca poprzednią wartość.
<code>public static int Decrement(ref int location)</code>	Zmniejsza zapisaną w location wartość o 1. Metoda Decrement() działa podobnie jak przedrostkowy operator --, ale jest bezpieczna ze względu na wątki.
<code>public static int Increment(ref int location)</code>	Powiększa zapisaną w location wartość o 1. Metoda Increment() działa podobnie jak przedrostkowy operator ++, ale jest bezpieczna ze względu na wątki.
<code>public static int Add(ref int location, int value)</code>	Dodaje wartość value do location i przypisuje do location wynik tej operacji. Działa podobnie jak operator +=.
<code>public static long Read(ref long location)</code>	Zwraca 64-bitową wartość w jednej atomowej operacji.

Większość spośród tych metod jest przeciążona i ma dodatkowe sygnatury (na przykład z parametrami typu long). W tabeli 22.2 znajdują się tylko ogólne sygnatury i opisy.

Zauważ, że metody Increment() i Decrement() można zastosować zamiast synchronizowanych operatorów ++ i -- z listingu 22.5, co pozwala poprawić wydajność kodu. Zauważ też, że jeśli inny wątek uzyska dostęp do pola _Count za pomocą metody spoza klasy Interlocked, działanie obu wątków nie będzie poprawnie synchronizowane.

Powiadomienia o zdarzeniach kierowane do wielu wątków

Jednym z obszarów, w których programiści często zapominają o synchronizacji, jest zgłaszanie zdarzeń. Na listingu 22.7 pokazano publikujący informacje o zdarzeniach kod, który nie jest bezpieczny ze względu na wątki.

Listing 22.7. Zgłaszanie powiadomień o zdarzeniach

```
// Ten kod nie jest bezpieczny ze względu na wątki.  
if (OnTemperatureChanged != null)  
{  
    // Wywołanie kierowane do subskrybentów.  
    OnTemperatureChanged(  
        this, new TemperatureEventArgs(value) );  
}
```

Ten kod jest prawidłowy, jeśli nie występuje sytuacja wyścigu między przedstawioną metodą a subskrybentami zdarzenia. Jednak ten kod nie jest atomowy, dlatego większa liczba wątków może skutkować sytuacją wyścigu. Możliwe, że między momentem sprawdzenia, czy wartość `OnTemperatureChanged` jest równa `null`, a zgłoszeniem zdarzenia wartość ta zostanie ustawiona na `null`, co doprowadzi do wyjątku `NullReferenceException`. Oznacza to, że jeśli wiele wątków może jednocześnie uzyskać dostęp do delegata, konieczne jest zsynchronizowanie operacji przypisywania wartości i wywoływanie delegata.

W wersji C# 6.0 rozwiązanie jest bardzo proste — wystarczy zastosować operator `?.`:

```
OnTemperature?.Invoke(
    this, new TemperatureEventArgs( value ) );
```

Operator `?.` zaprojektowano tak, by był atomowy. Dlatego gdy jest używany, wywołanie delegata odbywa się atomowo. Oczywiście ważne jest, by pamiętać o zastosowaniu tego operatora.

W wersjach starszych niż C# 6.0 wywoływanie delegata w sposób bezpieczny ze względu na wątki też nie jest trudne, choć wymaga więcej kodu. Przedstawione tu podejście działa, ponieważ operatory służące do dodawania i usuwania odbiorników są bezpieczne ze względu na wątki i statyczne (przeciążanie operatorów oparte jest na metodach statycznych). Aby poprawić kod z listingu 22.7 i zapewnić jego bezpieczeństwo ze względu na wątki, skopiuj delegat, sprawdź, czy jest ona różny od `null`, a następnie wywołaj tę kopię (zobacz listing 22.8).

Listing 22.8. Powiadamianie o zdarzeniach bezpieczne ze względu na wątki

```
// ...
TemperatureChangedHandler localOnChange =
    OnTemperatureChanged;
if (localOnChange != null)
{
    // Wywołanie kierowane do subskrybentów.
    localOnChange(
        this, new TemperatureEventArgs(value) );
}
// ...
```

Ponieważ delegat jest typu referencyjnego, zaskakujące może być stwierdzenie, że przyisanie wartości do zmiennej lokalnej i wywołanie kodu za pomocą tej zmiennej wystarczy, by sprawdzanie równości z wartością `null` było bezpieczne ze względu na wątki. Ponieważ zmienna `localOnChange` prowadzi do tej samej lokalizacji co delegat `OnTemperatureChange`, możesz sądzić, że zmiany w delegacie `OnTemperatureChange` będą odzwierciedlane także w zmiennej `localOnChange`.

Jest jednak *inaczej*. Wywołania `OnTemperatureChange += <odbiornik>` nie powodują dodania nowego delegata do `OnTemperatureChange`. Zamiast tego do `OnTemperatureChange` przypisany jest nowy delegat typu multicast, co nie wpływa na pierwotny delegat tego rodzaju, do którego prowadzi zmienna `localOnChange`. Dzięki temu kod jest bezpieczny ze względu na wątki, ponieważ tylko jeden wątek ma dostęp do obiektu `localOnChange`, a `OnTemperatureChange` po dodaniu lub usunięciu odbiorników prowadzi do zupełnie nowego obiektu.

Najlepsze praktyki z obszaru projektowania synchronizacji

Ponieważ programowanie wielowątkowe związane jest z komplikacjami, opracowano kilka najlepszych praktyk, które pomagają sobie radzić z problemami.

Unikanie zakleszczenia

Synchronizacja może prowadzić do zakleszczenia. Zakleszczenie następuje, gdy dwa wątki (lub większa ich liczba) oczekują od siebie nawzajem zwolnienia blokady. Założmy, że wątek 1 żąda blokady `_Sync1`, a następnie, przed jej zwolnieniem, żąda innej blokady, `_Sync2`. Natomiast wątek 2 w tym samym czasie żąda blokady `_Sync2`, a następnie `_Sync1` (bez zwalniania `_Sync2`). To może skutkować zakleszczeniem. Następuje ono, jeśli wątek 1 i wątek 2 zajmą jeden po drugim pierwsze blokady (`_Sync1` i `_Sync2`) przed zajęciem drugich blokad.

Aby nastąpiło zakleszczenie, muszą wystąpić cztery podstawowe warunki:

1. *Wzajemne wykluczanie.* Jeden wątek (ThreadA) zajmuje na wyłączność określony zasób, dlatego żaden inny wątek (na przykład ThreadB) nie może uzyskać dostępu do tego zasobu.
2. *Utrzymywanie blokady i oczekiwanie.* Jeden wątek (ThreadA) działający w trybie wzajemnego wykluczania oczekuje na zasób zajęty przez inny wątek (ThreadB).
3. *Brak wywłaszczenia.* Zasób zajmowany przez wątek (ThreadA) nie może zostać siłowo odzyskany; wątek ThreadA musi sam zwolnić blokowany zasób.
4. *Cykliczne oczekiwanie.* Dwa wątki (lub większa ich liczba) tworzą cykliczny łańcuch w taki sposób, że blokują (co najmniej) dwa zasoby, a każdy wątek oczekuje na zasób blokowany przez następny wątek w łańcuchu.

Wyeliminowanie choć jednego z tych warunków pozwala zapobiec zakleszczeniu.

Do zakleszczenia może łatwo dojść w sytuacji, gdy (co najmniej) dwa wątki żądają dostępu na wyłączność do (co najmniej) dwóch docelowych obiektów synchronizacji (zasobów), a blokady są zajmowane w odmiennej kolejności. Można tego uniknąć, jeśli programista dba o to, by zajmowanie kilku blokad zawsze odbywało się w tym samym porządku. Inną przyczyną zakleszczeń są blokady, które nie są **wielowejściowe**. Gdy blokada z jednego wątku może zablokować ten sam wątek (w momencie, gdy ponownie zażąda on tej samej blokady), nie jest wielowejściowa. Na przykład jeśli wątek ThreadA zajmie blokadę, a następnie ponownie jej zażąda, jednak zostanie zablokowany, ponieważ blokada jest już zajęta (przez ten właśnie wątek), blokada nie jest wielowejściowa. Wtedy kolejne zażądanie zajętej blokady prowadzi do zakleszczenia. Dlatego blokady, które nie są wielowejściowe, mogą występować tylko w jednym wątku.

Kod generowany na podstawie słowa kluczowego `lock` (z wykorzystaniem klasy `Monitor`) jest wielowejściowy. Jednak, co opisano w punkcie „Inne typy związane z synchronizacją”, niektóre rodzaje blokad nie są wielowejściowe.

Kiedy należy zapewniać synchronizację?

Jak wcześniej opisano, wszystkie dane statyczne powinny być bezpieczne ze względu na wątki. Dlatego synchronizować trzeba dostęp do wszystkich danych statycznych, które mogą się zmieniać. Programiści powinni zwykle deklarować prywatne zmienne statyczne i udostępniać

publiczne metody służące do modyfikowania tych danych. W takich metodach należy wewnętrznie zarządzać synchronizacją, jeśli możliwy jest dostęp wielu wątków do danych.

Natomiast stan instancji nie musi być synchronizowany. Synchronizacja może powodować znaczny spadek wydajności oraz zwiększa prawdopodobieństwo konkurowania o blokadę i zakleszczenia. Jeśli klasy nie są bezpośrednio projektowane z myślą o dostępie wielowątkowym, to programiści korzystający z obiektów w wielu wątkach powinni samodzielnie zarządzać synchronizacją współużytkowanych danych.

Unikanie zbędnych blokad

Programiści, dbając o zachowanie integralności danych, powinni unikać niepotrzebnej synchronizacji. Należy na przykład korzystać z niemodyfikowalnych typów w wątkach, aby synchronizacja nie była konieczna. To podejście okazało się nieocenione w językach funkcyjnych, takich jak F#. Należy też unikać blokowania prac bezpiecznych ze względu na wątki, na przykład prostych odczytów i zapisów wartości mniejszych niż natywny typ całkowitoliczbowy (o wielkości wskaźnika), które to operacje są automatycznie atomowe.

Wskazówki

NIE żądaj w różnej kolejności dostępu na wyłączność do tych samych dwóch (lub większej liczby) docelowych obiektów synchronizacji.

ZAPEWNIAJ, że kod, który jednocześnie utrzymuje kilka blokad, zawsze zajmuje je w tej samej kolejności.

HERMETYZUJ modyfikowalne dane statyczne za pomocą publicznych interfejsów API z obsługą synchronizacji.

UNIKAJ synchronizowania prostych odczytów lub zapisów wartości nie większych niż natywny typ całkowitoliczbowy (o wielkości wskaźnika), ponieważ takie operacje są automatycznie atomowe.

Inne typy związane z synchronizacją

Oprócz stosowania klas System.Threading.Monitor i System.Threading.Interlocked dostępnych jest kilka innych technik synchronizacji.

Początek
2.0

Korzystanie z klasy System.Threading.Mutex

Klasa System.Threading.Mutex przypomina klasę System.Threading.Monitor (choć nie udostępnia metody Pulse()), jednak nie jest generowana na podstawie słowa kluczowego lock, a obiektom tej klasy można przypisywać nazwy, co pozwala zapewnić synchronizację w ramach kilku procesów. Klasa Mutex umożliwia synchronizację dostępu do pliku lub innego zasobu używanego w różnych procesach. Ponieważ obiekt typu Mutex to zasób międzyprocesowy, w platformie .NET 2.0 dodano możliwość kontrolowania dostępu do niego. Służy do tego typ System.Security.AccessControl.MutexSecurity. Jednym z zastosowań klasy Mutex jest dodawanie ograniczenia pozwalającego na uruchomienie tylko jednej instancji aplikacji. Takie rozwiązanie pokazano na listingu 22.9.

Listing 22.9. Aplikacja umożliwiająca utworzenie tylko jednej instancji

```
using System;
using System.Threading;
using System.Reflection;

public class Program
{
    public static void Main()
    {
        // Tworzenie nazwy muteksu na podstawie
        // pełnej nazwy podzespołu.
        string mutexName =
            Assembly.GetEntryAssembly()!.FullName;

        // Parametr firstApplicationInstance określa, czy
        // dana instancja aplikacji jest pierwszą instancją
        using Mutex mutex = new Mutex(false, mutexName,
            out bool firstApplicationInstance);

        if (!firstApplicationInstance)
        {
            Console.WriteLine(
                "Ta aplikacja już działa.");
        }
        else
        {
            Console.WriteLine("Wciśnij ENTER, aby zakończyć");
            Console.ReadLine();
        }
    }
}
```

Efekt uruchomienia pierwszej instancji aplikacji pokazano w danych wyjściowych 22.4.

DANE WYJŚCIOWE 22.4.

Wciśnij ENTER, aby zakończyć

W danych wyjściowych 22.5 pokazano wynik próby uruchomienia drugiej instancji aplikacji w czasie, gdy pierwsza wciąż działa.

DANE WYJŚCIOWE 22.5.

Ta aplikacja już działa.

Ta aplikacja może zostać uruchomiona na danym komputerze tylko raz, nawet jeśli próbują włączyć ją różni użytkownicy. Aby ograniczyć liczbę instancji do jednej na użytkownika, dodaj do nazwy muteksu (`mutexName`) przyrostek `System.Environment.UserName`; wymaga to zastosowania platformy Microsoft .NET Framework lub specyfikacji .NET Standard 2.0.

Klasa `Mutex` dziedziczy po typie `System.Threading.WaitHandle`, dlatego zawiera metody `WaitAll()`, `WaitAny()` i `SignalAndWait()`. Te metody umożliwiają automatyczne zajęcie wielu blokad, na co klasa `Monitor` nie pozwala.

Klasa WaitHandle

System.Threading.WaitHandle to klasa bazowa dla klasy Mutex. Jest to podstawowa klasa do obsługi synchronizacji, używana w klasach Mutex, EventWaitHandle i Semaphore. Najważniejsze metody klasy WaitHandle to metody z rodziny WaitOne(). Blokują one wykonywanie kodu do momentu, w którym obiekt typu WaitHandle otrzyma sygnał lub zostanie ustawiona jego wartość. Istnieje kilka wersji przeciążonej metody WaitOne() — void WaitOne() (oczekuje w nieskończoność), bool WaitOne(int milliseconds) (przyjmuje czas oczekiwania podany w milisekundach) i bool WaitOne(TimeSpan timeout) (przyjmuje czas oczekiwania podany jako obiekt typu TimeSpan). Wersje, które zwracają wartość logiczną, zwracają true, gdy obiekt typu WaitHandle otrzyma sygnał przed upływem limitu czasu.

Klasa WaitHandle oprócz metod instancji udostępnia też dwie ważne składowe statyczne — WaitAll() i WaitAny(). Pozwalają one (podobnie jak metody instancji) podać limit czasu oczekiwania. Ponadto przyjmują tablicę obiektów typu WaitHandle, dzięki czemu mogą reagować na sygnały zgłasiane dla obiektów z tej kolekcji.

Zauważ, że obiekt typu WaitHandle obejmuje uchwyt (typu SafeWaitHandle) z implementacją interfejsu IDisposable. Dlatego trzeba zadbać o to, by usuwać obiekty typu WaitHandle, gdy nie są już potrzebne.

Początek
4.0

Zdarzenia resetujące — ManualResetEvent i ManualResetEventSlim

Jednym ze sposobów kontrolowania niepewności co do tego, kiedy określone instrukcje z wątku będą wykonywane względem instrukcji z innego wątku, jest wykorzystanie zdarzeń resetujących. Mimo występowania słowa *zdarzenia* w nazwie, zdarzenia resetujące nie mają nic wspólnego z delegatami i zdarzeniami języka C#. Takie zdarzenia służą do wymuszania na kodzie oczekiwania na wykonanie innego wątku do czasu otrzymania od tego wątku sygnału. Zdarzenia resetujące są przydatne zwłaszcza do testowania kodu wielowątkowego, ponieważ pozwalają oczekiwania na wystąpienie określonego stanu przed sprawdzeniem wyników.

Zdarzenia resetujące są typów System.Threading.ManualResetEvent i System.Threading.ManualResetEventSlim. Ten drugi typ został dodany w platformie Microsoft .NET Framework 4 i wymaga mniej zasobów. W przedstawionym dalej zagadnieniu dla zaawansowanych opisano też trzeci typ, System.Threading.AutoResetEvent, jednak programiści powinni zamiast niego stosować dwa pozostałe typy (zobacz zagadnienie „Przedkładaj typ ManualResetEvent i semafory nad typ AutoResetEvent”). Najważniejsze metody typów zdarzeń resetujących to Set() i Wait() (w klasie ManualResetEvent druga z tych metod nosi nazwę WaitOne()). Wywołanie metody Wait() sprawia, że wątek zostaje zablokowany do momentu wywołania w innym wątku metody Set() lub upłynięcia limitu czasu. Na listingu 22.10 pokazano działanie tej techniki, a efekt działania kodu znajdziesz w danych wyjściowych 22.6.

Listing 22.10. Oczekивание на здание ManualResetEventSlim

```
using System;
using System.Threading;
using System.Threading.Tasks;

public class Program
{
```

```
static ManualResetEventSlim _MainSignaledResetEvent;
static ManualResetEventSlim _DoWorkSignaledResetEvent;

public static void DoWork()
{
    Console.WriteLine("Początek metody DoWork()...");
    _DoWorkSignaledResetEvent.Set();
    _MainSignaledResetEvent.Wait();
    Console.WriteLine("Koniec metody DoWork()...");
}

public static void Main()
{
    using (_MainSignaledResetEvent =
        new ManualResetEventSlim())
    using (_DoWorkSignaledResetEvent =
        new ManualResetEventSlim())
    {
        Console.WriteLine(
            "Uruchomiono aplikację...");
        Console.WriteLine("Uruchamianie wątku...");

        // W platformie .NET 4.0 zastosuj metodę Task.Factory.StartNew.
        Task task = Task.Run(() => DoWork());

        // Blokowanie prac do czasu uruchomienia metody DoWork().
        _DoWorkSignaledResetEvent.Wait();
        Console.WriteLine(
            "Oczekiwanie na wykonanie pracy przez wątek...");
        _MainSignaledResetEvent.Set();
        task.Wait();
        Console.WriteLine("Zakończono pracę wątku");
        Console.WriteLine(
            "Zamykanie aplikacji...");
    }
}
```

4.0

DANE WYJŚCIOWE 22.6.

```
Uruchomiono aplikację...
Uruchamianie wątku...
Początek metody DoWork()...
Oczekiwanie na wykonanie pracy przez wątek...
Koniec metody DoWork()...
Zakończono pracę wątku
Zamykanie aplikacji...
```

Listing 22.10 rozpoczyna się od utworzenia nowego obiektu typu Task i uruchomienia go. W tabeli 22.3 poniżej pokazano ścieżkę wykonania, w której każda kolumna reprezentuje wątek. Gdy różne fragmenty kodu znajdują się w tym samym wierszu, nie jest określone, który fragment zostanie wykonany jako pierwszy.

Tabela 22.3. Ścieżka wykonania w kodzie z synchronizacją opartą na typie ManualResetEvent

Main()	Dowork()
...	
Console.WriteLine("Uruchomiono aplikację...");	
Task task = new Task(Dowork);	
Console.WriteLine("Uruchamianie wątku...");	
task.Start();	
_DoworkSignaledResetEvent.Wait();	Console.WriteLine("Uruchomiono metodę Dowork()...");
	_DoworkSignaledResetEvent.Set();
Console.WriteLine("Oczekiwanie na wykonanie pracy przez wątek...");	_MainSignaledResetEvent.Wait();
_MainSignaledResetEvent.Set();	
task.Wait();	Console.WriteLine("Koniec metody Dowork()...");
Console.WriteLine("Zakończono pracę wątku");	
Console.WriteLine("Zamykanie aplikacji...");	

4.0

Wywołanie metody `Wait()` zdarzenia resetującego (dla typu `ManualResetEvent` używana jest metoda `WaitOne()`) powoduje zablokowanie wątku wywołującego do czasu przesłania pozwalającego mu wznowić pracę sygnału przez inny wątek. Istnieją wersje metod `Wait()` i `WaitOne()`, które nie blokują pracy wątku w nieskończoność, ale przyjmują parametr (liczbę sekund lub obiekt typu `TimeSpan`) określający maksymalny czas zablokowania działania wątku. Gdy określony jest limit czasu, metoda `WaitOne()` zwraca wartość `false`, jeśli limit czasu zostanie przekroczony przed otrzymaniem sygnału przez zdarzenie resetujące. Metoda `ManualResetEvent.Wait()` ma także wersję przyjmującą token anulowania, co pozwala zgłaszać opisane w rozdziale 19. żądania anulowania.

Różnica między typami `ManualResetEventSlim` i `ManualResetEvent` polega na tym, że typ `ManualResetEvent` domyślnie wykorzystuje synchronizację z udziałem jądra, natomiast typ `ManualResetEventSlim` jest zoptymalizowany w taki sposób, by korzystać z jądra tylko w ostatczności. Dlatego typ `ManualResetEventSlim` jest wydajniejszy, choć może się zdarzyć, że zużyje więcej cykli procesora. Z tego powodu należy stosować właściwie ten typ, chyba że kod musi czekać na kilka zdarzeń lub synchronizacja dotyczy kilku procesów.

Zauważ, że w zdarzeniach resetujących zaimplementowany jest interfejs `IDisposable`. Dlatego gdy zdarzenie nie jest już potrzebne, reprezentujący je obiekt należy usnąć. Na listingu 22.10 używana jest do tego instrukcja `using`. Obiekt typu `CancellationTokenSource` zawiera zdarzenie `ManualResetEvent`, dlatego też obejmuje implementację interfejsu `IDisposable`.

Choć metody `Wait()` i `Pulse()` z klasy `System.Threading.Monitor` nie działają dokładnie tak samo, w niektórych sytuacjach zapewniają mechanizmy podobne do opisanych w tym miejscu.

ZAGADNIENIE DLA ZAAWANSOWANYCH

Przedkładaj typ `ManualResetEvent` i semafory nad typ `AutoResetEvent`

Istnieje też trzeci typ zdarzeń resetujących, `System.Threading.AutoResetEvent`. Podobnie jak typ `ManualResetEvent`, umożliwia on jednemu wątkowi przesłanie (za pomocą wywołania `Set()`) do innego wątku sygnału z informacją, że pierwszy wątek dotarł do określonej lokalizacji w kodzie. Różnica polega na tym, że typ `AutoResetEvent` powoduje odblokowanie wywołania `Wait()` w tylko jednym wątku. Gdy pierwszy wątek zostanie automatycznie zresetowany, blokada jest ponownie ustawiana. Gdy używany jest typ `AutoResetEvent`, istnieje duże ryzyko błędnego napisania wątku producenta z liczbą iteracji większą niż w wątku konsumenta. Dlatego zwykle lepiej stosować wzorzec z metodami `Wait()` i `Pulse()` klasy `Monitor` lub posługiwać się semaforami (jeśli dany blok może być wykonywany przez mniej niż n wątków).

Typ `ManualResetEvent` (w odróżnieniu od typu `AutoResetEvent`) wraca do stanu „nie otrzymano sygnału” dopiero po jawnym wywołaniu metody `Reset()`.

4.0

Typy `Semaphore`, `SemaphoreSlim` i `CountdownEvent`

Różnice w wydajności między typami `Semaphore` i `SemaphoreSlim` są podobne jak między typami `ManualResetEvent` i `ManualResetEventSlim`. W odróżnieniu od typów `ManualResetEvent` i `ManualResetEventSlim`, udostępniających blokadę, która może być tylko otwarta lub zamknięta (jak most), semafory pozwalają na jednoczesne wykonywanie sekcji krytycznej N wywołaniom. Semafor przechowuje licznik puli zasobów. Gdy wartość licznika spada do zera, semafor blokuje dostęp do puli do momentu zwrócenia jednego z zasobów. Wtedy zasób jest udostępniany następnemu zablokowanemu żądaniu z kolejki.

Typ `CountdownEvent` działa podobnie jak semafor, ale zapewnia synchronizację w odwrotnym warunku — zapewnia dostęp do zasobu tylko wtedy, gdy wartość licznika spada do zera (a nie tylko wtedy, gdy jest większa niż zero). Wyobraź sobie na przykład równoległą operację pobierania cen akcji różnych firm. Algorytm wyszukiwania należy uruchamiać dopiero po pobraniu cen akcji wszystkich spółek. Typ `CountdownEvent` można wykorzystać do synchronizowania pracy algorytmu wyszukiwania, dekrementacji licznika po pobraniu cen akcji poszczególnych firm i uruchamiania wyszukiwania w momencie, gdy licznik przyjmie wartość zero.

Zauważ, że typy `SemaphoreSlim` i `CountdownEvent` wprowadzono w platformie .NET 4. W platformie .NET 4.5 pierwszy z tych typów udostępnia metodę `SemaphoreSlim.WaitAsync()`, dzięki czemu w procesie oczekiwania na uzyskanie semafora można wykorzystać wzorzec TAP.

Klasy kolekcji przetwarzanych równolegle

Inny zestaw klas wprowadzonych w platformie Microsoft .NET Framework 4 to klasy kolekcji przetwarzanych równolegle. Te klasy zaprojektowano po to, by wbudować w nie kod obsługujący synchronizację. Dlatego wiele wątków może jednocześnie korzystać z obiektów tych klas i nie grozi to wystąpieniem sytuacji wyścigu. Listę klas kolekcji przetwarzanych równolegle zawiera tabela 22.4.

Tabela 22.4. Klasy kolekcji przetwarzanych równolegle

Klasa kolekcji	Opis
BlockingCollection<T>	Tworzy kolekcję z blokadami, która umożliwia producentom zapis danych w kolekcji i wczytywanie tych danych przez konsumentów. Ta klasa to generyczny typ kolekcji z synchronizacją operacji dodawania i usuwania danych bez uwzględniania sposobu przechowywania informacji na zapleczu (za pomocą kolejki, stosu, listy itd.). Typ BlockingCollection<T> zapewnia obsługę blokad i ograniczeń kolekcjom z implementacją interfejsu IProducerConsumerCollection<T>.
ConcurrentBag<T>*	Bezpieczna ze względu na wątki nieuporządkowana kolekcja obiektów typu T.
ConcurrentDictionary<TKey, TValue>	Bezpieczny ze względu na wątki słownik (kolekcja kluczy i wartości).
ConcurrentQueue<T>*	Bezpieczna ze względu na wątki kolejka przechowująca obiekty typu T w modelu FIFO (ang. <i>first in, first out</i> , czyli „pierwszy na wejściu, pierwszy na wyjściu”).
ConcurrentStack<T>*	Bezpieczny ze względu na wątki stos przechowujący obiekty typu T w modelu FILO (ang. <i>first in, last out</i> , czyli „pierwszy na wejściu, ostatni na wyjściu”).

* Klasy kolekcji z implementacją interfejsu IProducerConsumerCollection<T>.

Często stosowany wzorzec, możliwy dzięki wprowadzeniu kolekcji przetwarzanych równolegle, polega na zapewnieniu producentom i konsumentom bezpiecznego ze względu na wątki dostępu do danych. Klasy z implementacją interfejsu IProducerConsumerCollection<T> (wyróżnione w tabeli 22.4 za pomocą gwiazdki — *) są specjalnie zaprojektowane z myślą o tym mechanizmie. Umożliwia on jednej klasie (lub kilku klasom) przesyłanie danych do kolekcji w czasie, gdy inny zestaw klas wczytuje te dane, usuwając je z kolekcji. Kolejność dodawania i usuwania danych jest zależna od konkretnych klas kolekcji z implementacją interfejsu IProducerConsumerCollection<T>.

Istnieje też dodatkowa biblioteka (System.Collections.Immutable) zawierająca niemodyfikowalne kolekcje, przy czym nie jest ona wbudowana w standardową platformę .NET i Dotnet Core — jest ona dostępna jako pakiet NuGet. Zaletą niemodyfikowalnych kolekcji jest to, że można je swobodnie przekazywać między wątkami bez obaw o wystąpienie zakleszczenia lub wprowadzenie tymczasowych zmian. Ponieważ kolekcji niemodyfikowalnych nie można zmieniać, tymczasowe modyfikacje nigdy nie występują. Dlatego takie kolekcje są automatycznie bezpieczne ze względu na wątki (nie trzeba więc blokować dostępu do nich).

Pamięć lokalna wątków

W niektórych sytuacjach stosowanie blokad synchronizacyjnych może prowadzić do nieakceptownego spadku wydajności i ograniczenia skalowalności. Ponadto czasem zapewnienie synchronizacji dostępu do konkretnych danych może się okazać zbyt skomplikowane — zwłaszcza gdy synchronizacja jest dodawana już po napisaniu podstawowego kodu.

Zamiast synchronizacji można się posłużyć izolacją danych. Jedną z technik jej zapewniania jest **pamięć lokalna wątków**. Dzięki pamięci lokalnej wątków każdy wątek ma własnąinstancję zmiennej. Dlatego synchronizacja nie jest konieczna, ponieważ synchronizowanie danych występujących tylko w jednym wątku nie ma sensu. Dwa przykładowe typy z implementacją pamięci lokalnej wątków to `ThreadLocal<T>` i `ThreadStaticAttribute`.

ThreadLocal<T>

Aby w platformie Microsoft .NET Framework 4 (lub nowszej) utworzyć pamięć lokalną wątku, należy zadeklarować pole (lub zmienną w przypadku tworzenia domknięć przez kompilator) typu `ThreadLocal<T>`. W efekcie w każdym wątku utworzona zostanie inna instancja tego pola, co pokazano na listingu 22.11 i w danych wyjściowych 22.7. Zauważ, że różne instancje pola są tworzone nawet wtedy, gdy pole jest statyczne.

Listing 22.11. Używanie typu `ThreadLocal<T>` do tworzenia pamięci lokalnej wątku

```
using System;
using System.Threading;

public class Program
{
    static ThreadLocal<double> _Count =
        new ThreadLocal<double>(() => 0.01134);

    public static double Count
    {
        get { return _Count.Value; }
        set { _Count.Value = value; }
    }

    public static void Main()
    {
        Thread thread = new Thread(Decrement);
        thread.Start();

        // Inkrementacja.
        for (double i = 0; i < short.MaxValue; i++)
        {
            Count++;
        }

        thread.Join();
        Console.WriteLine("Count w metodzie Main = {0}", Count);
    }

    static void Decrement()
    {
        Count = -Count;
    }
}
```

```

for (double i = 0; i < short.MaxValue; i++)
{
    Count--;
}
Console.WriteLine(
    "Count w metodzie Decrement = {0}", Count);
}
}

```

DANE WYJŚCIOWE 22.7.

```

Count w metodzie Decrement = -32767.01134
Count w metodzie Main = 32767.01134

```

W danych wyjściowych 22.7 pokazano, że wartość właściwości Count w wątku wykonującym metodę Main() nigdy nie jest pomniejszana przez wątek wykonujący metodę Decrement(). W wątku metody Main() wartość początkowa to 0.01134, a wartość końcowa to 32767.01134. W metodzie Decrement() wartości są podobne, ale ujemne. Ponieważ właściwość Count jest oparta na polu statycznym typu ThreadLocal<T>, wątki wykonujące metody Main() i Decrement() przechowują we właściwości _Count.Value niezależne wartości.

Koniec
4.0

Pamięć lokalna wątku tworzona za pomocą atrybutu ThreadStaticAttribute

Opatrzenie pola statycznego atrybutem ThreadStaticAttribute, tak jak na listingu 22.12, to drugi sposób na sprawienie, by dla każdego wątku tworzona była odrębna instancja zmiennej statycznej. Ta technika ma kilka ograniczeń (w porównaniu z korzystaniem z typu ThreadLocal<T>), ale ma tę zaletę, że jest dostępna także w wersjach platformy .NET starszych niż 4. Ponadto ponieważ typ ThreadLocal<T> jest oparty na atrybucie ThreadStaticAttribute, używanie samego atrybutu wymaga mniejszej ilości pamięci i pozwala uzyskać niewielką poprawę wydajności, jeśli często wykonywane są niewielkie porcje kodu.

Listing 22.12. Tworzenie lokalnej pamięci wątku za pomocą atrybutu ThreadStaticAttribute

```

using System;
using System.Threading;

public class Program
{
    [ThreadStatic]
    static double _Count = 0.01134;
    public static double Count
    {
        get { return Program._Count; }
        set { Program._Count = value; }
    }

    public static void Main()
    {
        Thread thread = new Thread(Decrement);
        thread.Start();

        // Inkrementacja.
        for (int i = 0; i < short.MaxValue; i++)

```

```
{  
    Count++;  
}  
  
thread.Join();  
Console.WriteLine("Count w metodzie Main = {0}", Count);  
}  
  
static void Decrement()  
{  
    for (int i = 0; i < short.MaxValue; i++)  
    {  
        Count--;  
    }  
    Console.WriteLine("Count w metodzie Decrement = {0}", Count);  
}
```

Wynik działania kodu z listingu 22.12 pokazano w danych wyjściowych 22.8.

DANE WYJŚCIOWE 22.8.

```
Count w metodzie Decrement = -32767  
Count w metodzie Main = 32767.01134
```

Podobnie jak na listingu 22.11 wartość właściwości Count w wątku wykonującym metodę `Main()` nigdy nie jest zmniejszana przez wątek wykonujący metodę `Decrement()`. Tak więc gdy stosowany jest atrybut `ThreadStaticAttribute`, wartość właściwości Count w każdym wątku jest specyficzna dla tego wątku — nie jest dostępna w innych wątkach.

Zauważ, że (inaczej niż na listingu 22.11) wartość właściwości Count w metodzie `Decrement` nie ma żadnych cyfr po przecinku. To oznacza, że wartość nie została zainicjowana liczbą 0.01134. Choć wartość pola `_Count` jest ustawiana w deklaracji (za pomocą instrukcji `private double _Count = 0.01134`), inicjowanie dotyczy tylko instancji statycznego pola z wątku, w którym działa konstruktor statyczny. Na listingu 22.12 zmienna z pamięcią lokalną jest inicjowana wartością 0.01134 tylko w wątku wykonującym metodę `Main()`. Wartość pola `_Count` zmniejszana przez metodę `Decrement()` jest inicjowana liczbą 0 (to wartość wyrażenia `default(double)`, używana, ponieważ pole `_Count` jest typu `double`). Podobnie jeśli to konstruktor inicjuje pole z pamięcią lokalną wątku, wartość zostanie ustawiona tylko w wątku wywołującym ten konstruktor. Dlatego dobrym zwyczajem jest inicjowanie pola z pamięcią lokalną wątku w metodzie wywoływanej na początku pracy wszystkich wątków. Nie zawsze jest to jednak sensownym rozwiązaniem, zwłaszcza gdy używane jest słowo kluczowe `async`, ponieważ wtedy różne fragmenty obliczeń mogą być wykonywane w różnych wątkach. W takim scenariuszu w każdym fragmencie obliczeń wartość pamięci lokalnej wątku może się nieoczekiwanie okazać różna.

Decyzja o tym, czy zastosować pamięć lokalną wątków, wymaga analizy kosztów i zysków. Warto na przykład rozważyć wykorzystanie pamięci lokalnej wątków dla połączenia z bazą danych. Obsługa połączeń z niektórymi systemami zarządzania bazami danych bywa kosztowna, dlatego tworzenie połączenia dla każdego wątku może wymagać wielu zasobów. Jednak blokowanie połączenia w taki sposób, by wszystkie wywołania kierowane do bazy były

synchronizowane, znacznie obniża możliwości w zakresie skalowania rozwiązania. Każdy wzorzec zwiążany jest z kosztami i zyskami, a to, które podejście okaże się najlepsze, zależy zwykle od konkretnej sytuacji.

Następnym powodem stosowania pamięci lokalnej wątków jest udostępnianie potrzebnych informacji o kontekście innym metodom bez konieczności jawnego przekazywania tych danych za pomocą parametrów. Na przykład jeśli wiele metod ze stosu wywołań wymaga danych uwierzytelniających użytkownika, można przekazywać te dane za pomocą pól pamięci lokalnej wątków, a nie przy użyciu parametrów. Dzięki temu interfejs API jest bardziej przejrzysty, a informacje można udostępniać metodom w sposób bezpieczny ze względu na wątki. To podejście wymaga zapewnienia, że lokalne dane wątku zawsze są ustawiane. Ten krok jest ważny zwłaszcza wtedy, gdy korzystasz z obiektów typu Task lub wątków z puli, ponieważ wątki są wówczas wykorzystywane wielokrotnie.

Początek
5.0

Zegary

W niektórych sytuacjach konieczne jest opóźnienie wykonywania kodu o pewien czas lub zarejestrowanie części otrzymania powiadomienia po upływie danego czasu. Na przykład gdy dane często się zmieniają, ekran można odświeżać co określony czas, a nie natychmiast po wprowadzeniu modyfikacji. Jedną z technik obsługi zegarów jest wykorzystanie wzorca z modyfikatorami `async` i `await` z wersji C# 5.0 oraz metody `Task.Delay()` dodanej w platformie .NET 4.5. W rozdziale 19. wyjaśniono, że ważną cechą wzorca TAP jest to, iż kod wykonywany po wywoaniu asynchronicznym kontynuuje pracę w kontekście odpowiedniego wątku, co pozwala uniknąć problemów z wpływem innych wątków na działanie interfejsu użytkownika. Na listingu 22.13 pokazano przykład ilustrujący, jak korzystać z metody `Task.Delay()`.

Listing 22.13. Stosowanie zegara opartego na metodzie `Task.Delay()`

```
using System;
using System.Threading.Tasks;

public class Pomodoro
{
    // ...

    private static async Task TickAsync(
        System.Threading.CancellationToken token)
    {
        for (int minute = 0; minute < 25; minute++)
        {
            DisplayMinuteTicker(minute);
            for (int second = 0; second < 60; second++)
            {
                await Task.Delay(1000);
                if (token.IsCancellationRequested) break;
                DisplaySecondTicker();
            }
            if (token.IsCancellationRequested) break;
        }
    }
}
```

Wywołanie `Task.Delay(1000)` powoduje utworzenie zegara ze wstecznym odliczaniem, który po sekundzie uruchomi dalszy kod.

Na szczeble w wersji C# 5.0 kontekst synchronizacji we wzorcu TAP pozwala wykonywać kod zвязany z interfejsem użytkownika wyłącznie w wątku zarządzającym tym interfejsem. W starszych wersjach języka konieczne było stosowanie specjalnych klas do obsługi zegarów. Te klasy były bezpieczne ze względu na wątki lub mogły być skonfigurowane w ten sposób. Zegary takie jak `System.Windows.Forms.Timer`, `System.Windows.Threading.DispatcherTimer` i `System.Timers.Timer` są (po odpowiednim skonfigurowaniu) dostosowane do obsługi wątków interfejsu użytkownika. Inne, na przykład `System.Threading.Timer`, są zoptymalizowane pod kątem wydajności.

Koniec
5.0

ZAGADNIENIE DLA POCZĄTKUJĄCYCH I ZAAWANSOWANYCH

Kontrolowanie modelu wątkowego w technologii COM za pomocą atrybutu `STAThreadAttribute`

W technologii COM stosowane są cztery różne modele pracy wątków w przedziałach (ang. *apartments*). Te modele określają reguły pracy wątków związanego z wywołaniami zgłaszanymi przez obiekty COM. Na szczeble te reguły (i związane z nimi komplikacje) nie występują w platformie .NET, o ile program nie wywołuje komponentów COM. Ogólny sposób obsługi współdziałania z komponentami COM polega na umieszczeniu wszystkich komponentów platformy .NET w głównym przedziale jednowątkowym. Wymaga to opatrzenia metody `Main` w procesie atrybutem `System.STAThreadAttribute`. Nie trzeba wtedy wychodzić poza granice przedziału, by móc wywołać większość komponentów COM. Ponadto przedział jest inicjowany dopiero po wywołaniu kodu współdziałającego z komponentami COM. Wadą tego podejścia jest to, że wszystkie inne wątki (w tym te związane z obiektami typu `Task`) domyślnie działają w modelu **MTA** (ang. *Multi-threaded Apartment*, czyli z przedziałami wielowątkowymi). Wymaga to zachowania ostrożności przy wywoływaniu komponentów COM w wątkach innych niż główny.

Współdziałanie z komponentami COM nie musi być wynikiem jawnych działań programisty. Microsoft zaimplementował liczne komponenty platformy .NET za pomocą nakładek **RCW** (ang. *runtime callable wrapper*), zamiast przekształcać wszystkie mechanizmy technologii COM na kod zarządzany. Dlatego programiści często nieświadomie kierują wywołania do komponentów COM. Aby mieć pewność, że te wywołania zawsze są zgłaszane w przedziale jednowątkowym, warto opatrzyć metodę `Main` we wszystkich programach typu Windows Forms atrybutem `System.STAThreadAttribute`.

Podsumowanie

W tym rozdziale przedstawiono różne mechanizmy synchronizacji i pokazano, że dostępnych jest wiele klas pozwalających uniknąć sytuacji wyścigu. Omówiono między innymi słowo kluczowe `lock`, powodujące zastosowanie na zapleczu klasy `System.Threading.Monitor`. Inne klasy związane z synchronizacją to `System.Threading.Interlocked`, `System.Threading.Mutex` i `System.Threading.WaitHandle`, a także klasy zdarzeń resetujących, semaforów i klas równolegle przetwarzanych kolekcji.

Mimo postępów, jakie dokonały się w programowaniu wielowątkowym od czasów pierwszych wersji platformy .NET do dziś, synchronizowanie kodu nadal pozostaje skomplikowane i wiąże się z licznymi pułapkami. Aby ich uniknąć, warto uwzględnić zestaw najlepszych praktyk. Obejmują one spójne zajmowanie docelowych obiektów synchronizacji w tej samej kolejności i opakowywanie składowych statycznych w kod zapewniający synchronizację.

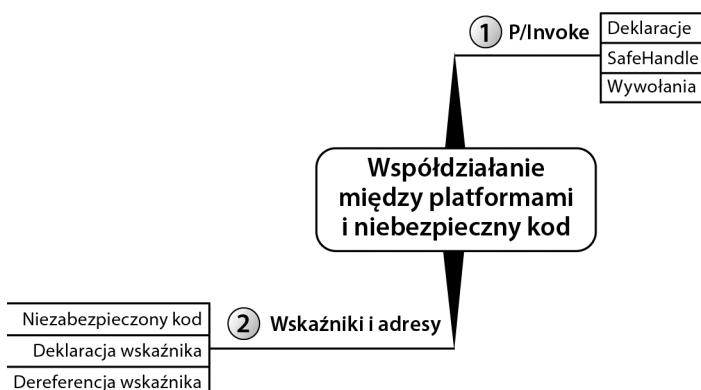
W końcowej części rozdziału opisano metodę `Task.Delay()` oraz wprowadzony w platformie .NET 4.5 interfejs API służący do tworzenia zegarów z użyciem wzorca TAP.

Następny rozdział dotyczy kolejnej skomplikowanej technologii z platformy .NET — szeregowania wywołań poza platformę .NET do kodu niezarządzanego za pomocą mechanizmu P/Invoke. Ponadto opisano tam niezabezpieczony kod, w którym język C# bezpośrednio stosuje wskaźniki do pamięci, podobnie jak dzieje się to w kodzie niezarządzanym (na przykład w języku C++).

■ 23 ■

Współdziałanie między platformami i niebezpieczny kod

JĘZYK C# OFERUJE BARDZO BOGATE MOŻLIWOŚCI, ponieważ cała platforma jest zarządzana. Jednak czasem konieczna jest rezygnacja z bezpieczeństwa zapewnianego przez język C# i wkroczenie w świat adresów pamięci oraz wskaźników. W C# taki efekt można uzyskać za pomocą dwóch podstawowych technik. Pierwsza możliwość polega na użyciu mechanizmu P/Invoke (ang. *Platform Invoke*) i kierowaniu wywołań do interfejsów API udostępnianych przez niezarządzane biblioteki DLL. Druga technika to pisanie **niebezpiecznego kodu**, który umożliwia dostęp do wskaźników i adresów pamięci.



Większość rozdziału poświęcona jest współdziałaniu z niezarządzanym kodem i posługiwaniu się nim. Omówienie tych tematów kończy się prezentacją krótkiego programu, który sprawdza identyfikator procesora komputera. W kodzie programu trzeba:

1. Skierować wywołanie do biblioteki DLL systemu operacyjnego i zażądać alokacji pamięci na potrzeby wykonywania instrukcji.
2. Zapisać instrukcje asemblerowe w zaalokowanym obszarze.

3. Wstrzyknąć adres pamięci do zapisanych instrukcji asemblerowych.
4. Wykonać kod asemblerowy.

W kodzie tego programu przedstawiono nie tylko mechanizm P/Invoke oraz konstrukcje stosowane w niezabezpieczonym kodzie, ale też pokazano pełnię możliwości języka C# i to, że można w nim (a także w kodzie zarządzanym) korzystać z kodu niezarządzanego.

Mechanizm P/Invoke

Założymy, że programista chce skierować wywołanie do biblioteki z istniejącym niezarządzanym kodem, uzyskać w systemie operacyjnym dostęp do kodu niezarządzanego, który nie jest dostępny za pomocą zarządzanego interfejsu API, lub zmaksymalizować wydajność konkretnego algorytmu, unikając występujących w środowisku uruchomieniowym kosztów sprawdzania typów i odzyskiwania pamięci. W takich sytuacjach niezbędne jest wywołanie skierowane do niezarządzanego kodu. Interfejs CLI umożliwia zgłaszanie takich wywołań za pomocą mechanizmu P/Invoke. Przy jego użyciu możesz kierować wywołania do interfejsów API z funkcjami eksportowanymi przez niezarządzane biblioteki DLL.

Wszystkie interfejsy API wywoływanie w tym podrozdziale pochodzą z systemu Windows. Choć te interfejsy nie występują w innych systemach operacyjnych, programiści mogą korzystać z mechanizmu P/Invoke do wywoływanego interfejsów API z innych systemów, a także do wywoływanego własnych bibliotek DLL. Wskazówki i składnia są w każdej sytuacji takie same.

Deklarowanie funkcji zewnętrznych

Po ustaleniu docelowej funkcji należy za pomocą mechanizmu P/Invoke zadeklarować ją w kodzie zarządzanym. Podobnie jak w przypadku wszystkich zwykłych metod należących do klas, trzeba zadeklarować docelowy interfejs API w kontekście klasy, dodając jednak modyfikator extern. Na listingu 23.1 pokazano, jak to zrobić.

Listing 23.1. Deklarowanie metody zewnętrznej

```
using System;
using System.Runtime.InteropServices;
class VirtualMemoryManager
{
    [DllImport("kernel32.dll", EntryPoint="GetCurrentProcess")]
    internal static extern IntPtr GetCurrentProcessHandle();
}
```

Tu klasa nosi nazwę `VirtualMemoryManager` (czyli menedżer pamięci wirtualnej), ponieważ ma zawierać funkcje powiązane z zarządzaniem pamięcią. Używana tu funkcja jest dostępna bezpośrednio w klasie `System.Diagnostics.Processor`, dlatego nie trzeba deklarować jej w kodzie. Zauważ, że zadeklarowana metoda zwraca wartość typu `IntPtr`, opisanego w następnym podrozdziale.

Metody zewnętrzne (z modyfikatorem `extern`) nigdy nie mają ciała i (prawie) zawsze są statyczne. Zamiast tworzyć ciało metody, należy dodać do jej deklaracji atrybut `DllImport` i wskazać w nim implementację metody. W tym atrybutie trzeba podać przynajmniej nazwę biblioteki DLL z definicją danej funkcji. Środowisko uruchomieniowe ustala nazwę funkcji na podstawie nazwy zadeklarowanej metody, choć można zmienić to domyślne rozwiązanie, podając nazwę funkcji w parametrze `EntryPoint`. Platforma .NET automatycznie próbuje wywołać wersję interfejsu API dla kodowania Unicode [...]W lub ASCII [...A].

Tu funkcja zewnętrzna `GetCurrentProcess()` pobiera pseudouchwyt bieżącego procesu, potrzebny w wywołaniu alokującym pamięć wirtualną. Oto deklaracja tej funkcji z kodu niezarządzanego:

```
HANDLE GetCurrentProcess();
```

Typy danych parametrów

Jeśli programista ustalił już docelową bibliotekę DLL i eksportowaną funkcję, najtrudniejszym krokiem jest określenie lub utworzenie zarządzanych typów danych odpowiadających niezarządzanym typom z funkcji zewnętrznej¹. Na listingu 23.2 pokazano interfejs API bardziej skomplikowany niż ten przedstawiony wcześniej.

Listing 23.2. Interfejs API funkcji `VirtualAllocEx()`

```
LPVOID VirtualAllocEx(
    HANDLE hProcess, // Uchwyt procesu. Funkcja alokuje
                    // pamięć w wirtualnej przestrzeni
                    // adresów danego procesu.
    LPVOID lpAddress, // Wskaźnik do żądanego początkowego
                     // adresu obszaru stron, który programista
                     // chce zaalokować. Jeśli lpAddress to NULL,
                     // funkcja sama ustala, gdzie ma
                     // zaalokować dany obszar stron.
    SIZE_T dwSize, // Wielkość alokowanego obszaru pamięci
                  // w bajtach. Jeśli lpAddress
                  // to NULL, funkcja zaokrąglą wartość dwSize
                  // w górę do granicy następnej strony.
    DWORD  fAllocationType, // Sposób alokacji pamięci.
    DWORD  fProtect); // Poziom ochrony alokowanego obszaru pamięci.
```

Metoda `VirtualAllocEx()` alokuje pamięć wirtualną, którą system operacyjny przeznacza specjalnie na wykonywanie kodu lub na dane. Aby wywołać tę metodę, w kodzie zarządzanym potrzebne są odpowiednie definicje wszystkich typów danych. Choć typy `HANDLE`, `LPVOID`, `SIZE_T` i `DWORD` są często używane w programowaniu z wykorzystaniem interfejsu API Win32, nie występują w kodzie zarządzanym w interfejsie CLI. Deklarację funkcji `VirtualAllocEx()` w języku C# przedstawiono na listingu 23.3.

¹ Bardzo przydatnym źródłem informacji o deklarowaniu interfejsów API Win32 jest strona <http://www.pinvoke.net>. Stanowi ona świetny punkt wyjścia do zapoznania się z wieloma interfejsami API. Pomoże Ci to uniknąć niektórych subtelnego problemów, które mogą wystąpić, gdy samodzielnie piszesz wywołania zewnętrznych interfejsów API.

Listing 23.3. Deklaracja funkcji VirtualAllocEx() w języku C#

```
using System;
using System.Runtime.InteropServices;
class VirtualMemoryManager
{
    [DllImport("kernel32.dll")]
    internal static extern IntPtr GetCurrentProcess();

    [DllImport("kernel32.dll", SetLastError = true)]
    private static extern IntPtr VirtualAllocEx(
        IntPtr hProcess,
        IntPtr lpAddress,
        IntPtr dwSize,
        AllocationType fAllocationType,
        uint fProtect);
}
```

Cechą charakterystyczną kodu zarządzanego jest to, że proste typy danych (na przykład `int`) nie zmieniają swojej wielkości w zależności od używanego procesora. Niezależnie od tego, czy procesor jest 16-, 32-, czy 64-bitowy, typ `int` zawsze zajmuje 32 bity. W kodzie niezarządzanym wielkość wskaźników pamięci zależy od procesora. Dlatego zamiast przekształcać typy takie jak `HANDLE` lub `LPVOID` na typ `int`, należy zastosować typ `System.IntPtr`, którego wielkość zależy od układu pamięci używanego w procesorze. W przykładowym kodzie zastosowano też typ wyliczeniowy `AllocationType`, opisany w podrozdziale „Upraszczanie wywołań interfejsu API za pomocą nakładek” w dalszej części rozdziału.

Na listingu 23.3 warto zwrócić uwagę na to, że typ `IntPtr` jest przydatny nie tylko dla wskaźników. Można go stosować także dla innych wartości. `IntPtr` to nie tyle „wskaźnik zapisany jako liczba całkowita”, ile „liczba całkowita o wielkości wskaźnika”. W obiekcie typu `IntPtr` nie trzeba więc zapisywać wskaźnika, ale przechowywane dane muszą mieć wielkość taką jak wskaźnik. Wiele danych ma wielkość wskaźnika, choć nie są przy tym wskaźnikiem.

Stosowanie parametrów ref zamiast wskaźników

W niezarządzanym kodzie wskaźniki często używane są do przekazywania parametrów przez referencję. W takich sytuacjach mechanizm P/Invoke nie wymaga przekształcania danego typu na wskaźnik w kodzie zarządzanym. Zamiast tego odpowiednie parametry można opatrzyć modyfikatorem `ref` (lub `out` w zależności od tego, czy parametr jest wejściowy i wyjściowy, czy tylko wyjściowy). Na listingu 23.4 parametr `lpfOldProtect` typu `PDWORD` zwraca „wskaźnik do zmiennej, w której zapisany jest poprzedni poziom ochrony dostępu pierwsiowej strony z określonego obszaru stron”².

Listing 23.4. Stosowanie parametrów `ref` i `out` zamiast wskaźników

```
class VirtualMemoryManager
{
    // ...
}
```

² Według dokumentacji MSDN.

```
[DllImport("kernel32.dll", SetLastError = true)]
static extern bool VirtualProtectEx(
    IntPtr hProcess, IntPtr lpAddress,
    IntPtr dwSize, uint flNewProtect,
    [ref uint] lpflOldProtect);
}
```

Choć parametr `lpf1OldProtect` jest w dokumentacji opisany jako `[out]` (mimo że sygnatura tego nie wymusza), z jego opisu wynika, że parametr musi prowadzić do poprawnej zmiennej (nie może być równy `NULL`). Ta niespójność jest myląca, ale często spotykana. Zgodnie z wytycznymi do parametrów używanych przez mechanizm P/Invoke należy stosować modyfikator `ref` zamiast `out`, ponieważ jednostka wywołująca zawsze może pominąć dane przekazane za pomocą parametru `ref`, natomiast parametr `out` nie pozwala przekazać informacji.

Inne parametry są takie same jak w metodzie `VirtualAllocEx()`, przy czym `lpAddress` to adres zwracany przez wywołanie `VirtualAllocEx()`. Ponadto parametr `flNewProtect` określa sposób ochrony pamięci — strony z możliwością wykonywania, strony tylko do odczytu itd.

Używanie atrybutu `StructLayoutAttribute` do zapewniania układu sekwencyjnego

Niekotere interfejsy API obejmują typy, które nie mają swoich odpowiedników w kodzie zarządzanym. Aby zastosować takie typy, trzeba je ponownie zadeklarować w kodzie zarządzanym. Na listingu 23.5 pokazano kod zarządzany z deklaracją struktury `COLORREF`.

Listing 23.5. Deklarowanie typów na podstawie niezarządzanych struktur

```
[StructLayout(LayoutKind.Sequential)]
struct ColorRef
{
    public byte Red;
    public byte Green;
    public byte Blue;
    // Wyłączenie ostrzeżenia informującego, że zmienna Unused nie jest używana.
    #pragma warning disable 414
    private byte Unused;
    #pragma warning restore 414

    public ColorRef(byte red, byte green, byte blue)
    {
        Blue = blue;
        Green = green;
        Red = red;
        Unused = 0;
    }
}
```

W systemie Microsoft Windows różne interfejsy API związane z kolorem używają typu `COLORREF` do reprezentowania kolorów w formacie RGB (obejmującego natężenie składowych czerwonej, zielonej i niebieskiej).

Najważniejszy w deklaracji z listingu 23.5 jest atrybut StructLayoutAttribute. Kod zarządzany domyślnie może optymalizować układ składowych typu w pamięci, dlatego układ pól nie zawsze jest sekwencyjny. Aby wymusić układ sekwencyjny, co pozwala bezpośrednio odwzorować typ i kopiować obiekty bit po bicie z kodu zarządzanego do niezarządzanego (oraz w drugą stronę), należy dodać atrybut StructLayoutAttribute z parametrem w postaci wartości wyliczeniowej LayoutKind.Sequential. Ta technika jest przydatna także wtedy, gdy dane są zapisywane do strumieni plików i z nich pobierane, gdyż wówczas układ sekwencyjny często jest oczekiwany.

Ponieważ definicja struktur z kodu niezarządzanego (w języku C++) nie pasuje do definicji z języka C#, nie istnieje bezpośrednie odwzorowanie między strukturami niezarządzanymi i zarządzanymi. Dlatego programiści powinni przestrzegać standardowych wytycznych z języka C#, określających, czy dany typ powinien być bezpośredni, czy referencyjny. Należy przy tym uwzględnić wielkość typu (to, czy zawiera mniej niż 16 bajtów).

Obsługa błędów

Niewygodnym aspektem programowania z wykorzystaniem interfejsu API Win32 jest to, że interfejsy API często zgłaszają błędy w niespójny sposób. Na przykład niektóre z nich zwracają wartość (0, 1, false itd.), aby poinformować o błędzie, natomiast inne ustawiają w określony sposób parametr out. Ponadto uzyskanie szczegółowych informacji o problemie wymaga dodatkowego wywołania metody GetLastError(), a w celu pobrania komunikatu o błędzie należy wywołać metodę FormatMessage(). Zgłaszanie błędów z interfejsu API Win32 w niezarządzanym kodzie rzadko odbywa się z użyciem wyjątków.

Na szczęście projektanci technologii P/Invoke udostępnili mechanizm obsługi błędów. Jak go włączyć? Jeśli nazwany parametr SetLastError atrybutu DllImport ma wartość true, można utworzyć wyjątek System.ComponentModel.Win32Exception, który natychmiast po wywołaniu instrukcji mechanizmu P/Invoke jest automatycznie inicjowany danymi o błędzie z Win32 (zobacz listing 23.6).

Listing 23.6. Obsługa błędów w interfejsie API Win32

```
class VirtualMemoryManager
{
    [DllImport("kernel32.dll", SetLastError = true)]
    private static extern IntPtr VirtualAllocEx(
        IntPtr hProcess,
        IntPtr lpAddress,
        IntPtr dwSize,
        AllocationType fAllocationType,
        uint fProtect);

    // ...
    [DllImport("kernel32.dll", SetLastError = true)]
    static extern bool VirtualProtectEx(
        IntPtr hProcess, IntPtr lpAddress,
        IntPtr dwSize, uint fNewProtect,
        ref uint lpfOldProtect);
```

```
[Flags]
private enum AllocationType : uint
{
    // ...
}

[Flags]
private enum ProtectionOptions
{
    // ...
}

[Flags]
private enum MemoryFreeType
{
    // ...
}

public static IntPtr AllocExecutionBlock(
    int size, IntPtr hProcess)
{
    IntPtr codeBytesPtr;
    codeBytesPtr = VirtualAllocEx(
        hProcess, IntPtr.Zero,
        (IntPtr)size,
        AllocationType.Reserve | AllocationType.Commit,
        (uint)ProtectionOptions.PageExecuteReadWrite);

    if (codeBytesPtr == IntPtr.Zero)
    {
        throw new System.ComponentModel.Win32Exception();
    }

    uint lpfiOldProtect = 0;
    if (!VirtualProtectEx(
        hProcess, codeBytesPtr,
        (IntPtr)size,
        (uint)ProtectionOptions.PageExecuteReadWrite,
        ref lpfiOldProtect))
    {
        throw new System.ComponentModel.Win32Exception();
    }
    return codeBytesPtr;
}

public static IntPtr AllocExecutionBlock(int size)
{
    return AllocExecutionBlock(
        size, GetCurrentProcessHandle());
}
```

To rozwiązanie umożliwia programistom dodanie do wszystkich interfejsów API niestandardowej obsługi błędów, przy czym błędy dodatkowo wciąż są zgłaszane w standardowy sposób.

Na listingach 23.1 i 23.3 metody wykorzystujące mechanizm P/Invoke są zadeklarowane jako wewnętrzne lub prywatne. Jednak większość interfejsów API (oprócz tych najprostszego) warto opakować w publiczne nakładki. Pozwala to uprościć wywołania interfejsów API opartych na mechanizmie P/Invoke, zwiększyć ich użyteczność i uzyskać bardziej obiektowy kod. Deklaracja metody AllocExecutionBlock() na listingu 23.6 to dobry przykład zastosowania tego podejścia.

Wskazówka

TWÓRZ publiczne zarządzane nakładki dla niezarządzanych metod. Stosuj w nich konwencje typowe dla kodu zarządzanego (na przykład ustrukturyzowaną obsługę wyjątków).

Początek
2.0

Używanie typu SafeHandle

Mechanizm P/Invoke często korzysta z zasobu, na przykład z uchwytu okna, który po użyciu trzeba w kodzie usunąć. Zamiast zmuszać programistów do pamiętania o tym kroku i każdorazowego ręcznego dodawania kodu, warto udostępnić klasę z implementacją interfejsu IDisposable oraz finalizatora. Na listingu 23.7 adres zwracany przez metody VirtualAllocEx() i VirtualProtectEx() wymaga późniejszego wywołania metody VirtualFreeEx(). Aby zapewnić wbudowaną obsługę tego mechanizmu, można zdefiniować klasę VirtualMemoryPtr pochodną od System.Runtime.InteropServices.SafeHandle.

Listing 23.7. Tworzenie zarządzanych zasobów za pomocą typu SafeHandle

```
public class VirtualMemoryPtr :  
    System.Runtime.InteropServices.SafeHandle  
{  
    public VirtualMemoryPtr(int memorySize) :  
        base(IntPtr.Zero, true)  
    {  
        _ProcessHandle =  
            VirtualMemoryManager.GetCurrentProcessHandle();  
        _MemorySize = (IntPtr)memorySize;  
        _AllocatedPointer =  
            VirtualMemoryManager.AllocExecutionBlock(  
                memorySize, ProcessHandle);  
        _Disposed = false;  
    }  
  
    public readonly IntPtr _AllocatedPointer;  
    readonly IntPtr _ProcessHandle;  
    readonly IntPtr _MemorySize;  
    bool _Disposed;  
  
    public static implicit operator IntPtr(  
        VirtualMemoryPtr virtualMemoryPointer)  
    {  
        return virtualMemoryPointer._AllocatedPointer;  
    }
```

```
// Abstrakcyjna składowa z typu SafeHandle.
public override bool IsInvalid
{
    get
    {
        return _Disposed;
    }
}

// Abstrakcyjna składowa z typu SafeHandle.
protected override bool ReleaseHandle()
{
    if (!_Disposed)
    {
        _Disposed = true;
        GC.SuppressFinalize(this);
        VirtualMemoryManager.VirtualFreeEx(_ProcessHandle,
            _AllocatedPointer, _MemorySize);
    }
    return true;
}
}
```

Klasa `System.Runtime.InteropServices.SafeHandle` obejmuje abstrakcyjne składowe `IsInvalid` i `ReleaseHandle()`. Kod zwalniający zasoby należy umieścić w drugiej z tych składowych. Pierwsza informuje, czy ów kod został już wykonany.

Za pomocą typu `VirtualMemoryPtr` można alokować pamięć, tworząc obiekt tego typu i określając wymagania pamięciowe.

Koniec
2.0

Wywoływanie funkcji zewnętrznych

Po użyciu mechanizmu P/Invoke do zadeklarowania funkcji można wywoływać je w ten sam sposób jak inne składowe klasy. Ważne jest to, że importowana biblioteka DLL (podobnie jak katalog z plikiem wykonywalnym) musi być dostępna w ścieżce klas, by można ją było poprawnie wczytać. Na listingach 23.6 i 23.7 przedstawiono to podejście z wykorzystaniem określonych stałych.

Ponieważ `f1AllocationType` i `f1Protect` to flagi, dobrą praktyką jest udostępnianie reprezentujących je stałych lub wartości wyliczeniowych. Zamiast oczekiwania, że zostaną one zdefiniowane w jednostce wywołującej, zgodnie z zasadami hermetyzacji warto udostępnić je w deklaracji interfejsu API, tak jak na listingu 23.8.

Listing 23.8. Łączenie interfejsów API w ramach hermetyzacji

```
class VirtualMemoryManager
{
    // ...
    /// <summary>
    /// Typ alokacji pamięci. Ten parametr musi
    /// mieć jedną z określonych dalej wartości.
    /// </summary>
    [Flags]
```

```
private enum AllocationType : uint
{
    /// <summary>
    /// Alokuje fizycznie obszar na określone zarezerwowane
    /// strony w pamięci lub w pliku stronicowania na dysku.
    /// Inicjuje tę pamięć wartością zero.
    /// </summary>
    Commit = 0x1000,
    /// <summary>
    /// Rezerwuje zakres przestrzeni adresów wirtualnych
    /// procesu bez alokowania fizycznego obszaru w
    /// pamięci lub w pliku stronicowania na dysku.
    /// </summary>
    Reserve = 0x2000,
    /// <summary>
    /// Informuje, że dane z obszaru pamięci określonego przez
    /// parametry lpAddress i dwSize nie są już istotne. Tych
    /// stron nie należy wczytywać ani zapisywać w pliku
    /// stronicowania. Jednak dany blok pamięci może być
    /// jeszcze używany, dlatego nie należy go zwalniać. Tej
    /// wartości nie można łączyć z innymi.
    /// </summary>
    Reset = 0x80000,
    /// <summary>
    /// Alokuje fizyczną pamięć z dostępem do odczytu i do zapisu.
    /// Ta wartość jest dostępna tylko dla pamięci
    /// AWE (ang. Address Windowing Extensions).
    /// </summary>
    Physical = 0x400000,
    /// <summary>
    /// Alokuje pamięć o najwyższym możliwym adresie.
    /// </summary>
    TopDown = 0x100000,
}

/// <summary>
/// Określa poziom ochrony alokowanego obszaru pamięci ze stronami.
/// </summary>
[Flags]
private enum ProtectionOptions : uint
{
    /// <summary>
    /// Umożliwia wykonywanie zajętego obszaru ze stronami.
    /// Próba odczytu lub zapisu w tym obszarze
    /// spowoduje błąd naruszenia poziomu dostępu.
    /// </summary>
    Execute = 0x10,
    /// <summary>
    /// Zapewnia możliwość wykonywania i odczytu zajętego
    /// obszaru ze stronami. Próba zapisu w tym
    /// obszarze spowoduje błąd naruszenia poziomu dostępu.
    /// </summary>
    PageExecuteRead = 0x20,
    /// <summary>
    /// Umożliwia wykonywanie, odczyt i zapis w
    /// zajętym obszarze ze stronami.
    /// </summary>
    PageExecuteReadWrite = 0x40,
    // ...
}
```

```

}

/// <summary>
/// Typ operacji zwalniającej pamięć.
/// </summary>
[Flags]
private enum MemoryFreeType : uint
{
    /// <summary>
    /// Zwalnia określony zajęty obszar stron.
    /// Po tej operacji strony są uznawane za zarezerwowane (stan Reserved).
    /// </summary>
    Decommit = 0x4000,
    /// <summary>
    /// Zwalnia określony obszar ze stronami. Po wykonaniu
    /// tej operacji strony są uznawane za zwolnione (stan Free).
    /// </summary>
    Release = 0x8000
}
// ...
}

```

Zaletą wartości wyliczeniowych jest to, że można łączyć różne wartości. Ponadto za pomocą wyliczeń można ograniczyć zakres stosowanych wartości tylko do tych określonych.

Upraszczanie wywołań interfejsów API za pomocą nakładek

Niezależnie od tego, czy dodawana jest obsługa błędów, struktur, czy stałych, jednym z celów autorów dobrych interfejsów API jest udostępnienie uproszczonego zarządzanego interfejsu API, opakowującego używany interfejs API Win32. Na przykład na listingu 23.9 tworzona jest przejęta wersja metody `VirtualFreeEx()` w postaci metody publicznej, która upraszcza wywołania.

Listing 23.9. Opakowywanie używanego interfejsu API

```

class VirtualMemoryManager
{
    // ...

[DllImport("kernel32.dll", SetLastError = true)]
static extern bool VirtualFreeEx(
    IntPtr hProcess, IntPtr lpAddress,
    IntPtr dwSize, IntPtr dwFreeType);
public static bool VirtualFreeEx(
    IntPtr hProcess, IntPtr lpAddress,
    IntPtr dwSize)
{
    bool result = VirtualFreeEx(
        hProcess, lpAddress, dwSize,
        (IntPtr)MemoryFreeType.Decommit);
    if (!result)
    {
        throw new System.ComponentModel.Win32Exception();
    }
    return result;
}

```

```
public static bool VirtualFreeEx(
    IntPtr lpAddress, IntPtr dwSize)
{
    return VirtualFreeEx(
        GetCurrentProcessHandle(), lpAddress, dwSize);
}

[DllImport("kernel32", SetLastError = true)]
static extern IntPtr VirtualAllocEx(
    IntPtr hProcess,
    IntPtr lpAddress,
    IntPtr dwSize,
    AllocationType fAllocationType,
    uint fProtect);

// ...
}
```

Odwzorowywanie wskaźników do funkcji na delegaty

Ostatnią kwestią związaną z mechanizmem P/Invoke jest to, że wskaźniki do funkcji z kodu niezarządzanego odpowiadają w kodzie zarządzanym delegatom. Na przykład aby utworzyć zegar z systemu Windows, należy udostępnić wskaźnik do funkcji, którą zegar po upływie określonego czasu może zwrotnie wywołać. Wymaga to przekazania użycia delegata o sygnaturze pasującej do wywołania zwrotnego.

Wskazówki

Z powodu osobliwości związanych z mechanizmem P/Invoke warto stosować się do zestawu wskazówek, które ułatwiają pisanie kodu z wykorzystaniem tego mechanizmu.

Wskazówki

NIE replikuj bez potrzeby istniejących zarządzanych klas, które wykonują funkcje określonego niezarządzanego interfejsu API.

DEKLARUJ metody z modyfikatorem `extern` jako prywatne lub wewnętrzne.

UDOSTĘPNIAJ publiczne metody nakładkowe, w których stosowane są konwencje z kodu zarządzanego (ustrukturyzowana obsługa wyjątków, specjalne wartości pochodzące z typu `wyliczeniowego` itd.).

UPRASZCZAJ metody nakładkowe, stosując wartości domyślne dla parametrów, które nie są wymagane.

STOSUJ atrybut `SetLastErrorAttribute`, aby przekształcać interfejsy API używające kodów błędów ustawianych w wywołaniu `SetLastError` w metody zgłaszające wyjątek `Win32Exception`.

ROZSZERZAJ typ `SafeHandle` lub implementuj interfejs `IDisposable` i twórz finalizator, by zapewnić efektywne zwalnianie niezarządzanych zasobów.

STOSUJ typy delegatów o sygnaturze pasującej do potrzebnej metody, jeśli niezarządzany interfejs API wymaga wskaźnika do funkcji.

STOSUJ parametry `ref` zamiast typów wskaźnikowych (jeśli jest to możliwe).

Wskaźniki i adresy

Czasem programista chce uzyskać dostęp do pamięci i móc pracować bezpośrednio z nią oraz wskaźnikami do niej. Jest to potrzebne na przykład w niektórych interakcjach z systemem operacyjnym, a także w pewnych algorytmach krytycznych czasowo. Aby umożliwić pracę z pamięcią, w C# udostępniono kod niezabezpieczony.

Kod niezabezpieczony

Jedną z bardzo wartościowych cech języka C# jest to, że zapewnia ścisłą kontrolę typów i obsługuje sprawdzanie typów w trakcie wykonywania kodu w środowisku uruchomieniowym. Tym, co dodatkowo zwiększa wartość tego rozwiązania, jest możliwość pominięcia go i bezpośredniego manipulowania pamięcią oraz adresami. Tę technikę można stosować na przykład w trakcie korzystania z urządzeń odwzorowywanych w pamięci lub w implementacjach algorytmów krytycznych czasowo. Ważne jest, aby oznaczyć fragment kodu jako niezabezpieczony.

Kod niezabezpieczony to jawnie oznaczony blok kodu i opcja komplikacji, co pokazano na listingu 23.10. Modyfikator unsafe nie ma wpływu na wygenerowany kod CIL, a jest jedynie dyrektywą dla kompilatora, oznaczającą, że kompilator ma zezwalać na manipulowanie w danym bloku kodu wskaźnikami i adresami. To, że kod jest niezabezpieczony, nie oznacza, że jest *nieszarządzany*.

Listing 23.10. Określanie, że metoda może zawierać niezabezpieczony kod

```
class Program
{
    unsafe static int Main(string[] args)
    {
        // ...
    }
}
```

Modyfikator unsafe można stosować do typu lub do wybranych składowych.

W języku C# modyfikator unsafe można też zastosować jako instrukcję, która informuje, że w danym bloku dozwolony jest niezabezpieczony kod (zobacz listing 23.11).

Listing 23.11. Określanie, że dany blok może zawierać niezabezpieczony kod

```
class Program
{
    static int Main(string[] args)
    {
        unsafe
        {
            // ...
        }
    }
}
```

W bloku opatrzonym modyfikatorem `unsafe` można stosować niezabezpieczone mechanizmy, na przykład wskaźniki.

Uwaga

Trzeba jawnie poinformować kompilator, że w danym miejscu dozwolony jest niezabezpieczony kod.

Gdy piszesz niezabezpieczony kod, jest on podatny na przepełnienie bufora i podobne problemy, które mogą skutkować lukami bezpieczeństwa. Dlatego konieczne jest bezpośrednio powiadomianie kompilatora o niezabezpieczonym kodzie. Aby to zrobić, należy ustawić właściwość `AllowUnsafeBlocks` na wartość `true` w pliku `.csproj`. Ilustruje to listing 23.12.

Listing 23.12. Powiadomianie kompilatora o niezabezpieczonym kodzie

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp1.0</TargetFramework>
    <ProductName>Chapter20</ProductName>
    <WarningLevel>2</WarningLevel>
    <AllowUnsafeBlocks>True</AllowUnsafeBlocks>
  </PropertyGroup>
  <Import Project="..\Versioning.targets" />
  <ItemGroup>
    <ProjectReference Include="..\SharedCode\SharedCode.csproj" />
  </ItemGroup>
</Project>
```

Inna możliwość to przekazanie potrzebnej właściwości w wierszu poleceń w instrukcji `dotnet build` (zobacz dane wyjściowe 23.1).

DANE WYJŚCIOWE 23.1

```
dotnet build /property:AllowUnsafeBlocks=True
```

Gdy bezpośrednio wywołujesz kompilator języka C#, musisz zastosować opcję `/unsafe` (zobacz dane wyjściowe 23.2).

DANE WYJŚCIOWE 23.2

```
csc.exe /unsafe Program.cs
```

W środowisku Visual Studio można aktywować ten mechanizm, zaznaczając pole wyboru *Zezwalać na niebezpieczny kod* w zakładce *Kompilacja* w oknie właściwości projektu.

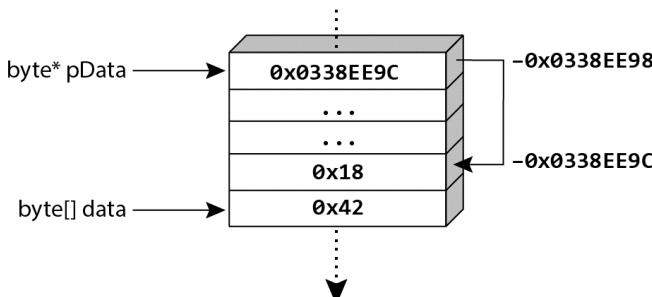
Opcja `/unsafe` umożliwia bezpośrednie manipulowanie pamięcią i wykonywanie niezarządzanych instrukcji. Wymóg stosowania opcji `/unsafe` powoduje, że programista jawnie deklaruje świadomość potencjalnych problemów. Duże możliwości wiążą się z dużą odpowiedzialnością.

Deklarowanie wskaźników

Skoro wiesz już, jak oznaczyć kod jako niezabezpieczony, pora zobaczyć, jak pisać taki kod. W niezabezpieczonym kodzie można przede wszystkim deklarować wskaźniki. Przyjrzyj się poniższemu przykładowi:

```
byte* pData;
```

Jeśli zmienna pData ma wartość różną od null, jej wartość prowadzi do lokalizacji zawierającej jeden bajt lub więcej przyległych bajtów. Wartość zmiennej pData reprezentuje adres tych bajtów w pamięci. Typ podany przed gwiazdką (*) to **typ docelowy**, czyli typ wartości znajdującej się w miejscu, do którego prowadzi wskaźnik. W tym przykładzie pData to wskaźnik, a byte to typ docelowy, co pokazano na rysunku 23.1.



Rysunek 23.1. Wskaźniki zawierają adresy danych

Ponieważ wskaźniki to liczby całkowite, które określają adres pamięci, nie są usuwane przez mechanizm odzyskiwania pamięci. C# nie pozwala na stosowanie typów docelowych innych niż **typy niezarządzane** (niedozwolone są więc typy referencyjne, generyczne lub zawierające typy referencyjne). Dlatego poniższe polecenie jest błędne:

```
string* pMessage;
```

Także następne polecenie jest nieprawidłowe:

```
ServiceStatus* pStatus;
```

Typ ServiceStatus jest zdefiniowany na listingu 23.13. Problem polega tu na tym, że typ ServiceStatus obejmuje pole typu string.

Listing 23.13. Przykładowy kod z błędny typem docelowym

```
struct ServiceStatus
{
    int State;
    string Description; // Pole Description jest typu referencyjnego.
}
```

Porównanie języków — deklaracje wskaźników w językach C i C++

W językach C i C++ kilka wskaźników w jednej deklaracji można zapisać w następujący sposób:

```
int *p1, *p2;
```

Zwróć uwagę na gwiazdkę (*) przy wskaźniku p2. Sprawia ona, że p2 jest typu `int*`, a nie `int`. Natomiast w języku C# gwiazdka zawsze umieszczana jest przy typie danych:

```
int* p1, p2;
```

Ten kod tworzy dwie zmienne typu `int*`. Ta składnia działa podobnie jak zadeklarowanie kilku tablic w jednej instrukcji:

```
int[] array1, array2;
```

Wskaźniki to typ zupełnie innego rodzaju od pozostałych. Wskaźniki (w odróżnieniu od struktur, wyliczeń i klas) nie dziedziczą po typie `System.Object`. Nie można ich nawet przekształcić na ten typ. Można je jednak (jawnie) przekształcić na typ `System.IntPtr`, który z kolei umożliwia konwersję na typ `System.Object`.

Typami docelowymi oprócz standardowych struktur, które zawierają tylko pola typów niezarządzanych, mogą być typy wyliczeniowe, wbudowane typy bezpośrednie (sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal i bool) i typy wskaźnikowe (na przykład `byte**`). Dozwolone są też wskaźniki typu `void*`; są to wskaźniki do wartości nieznanego typu.

Przypisywanie wartości do wskaźników

Po zdefiniowaniu wskaźnika, a przed uzyskaniem dostępu do niego trzeba przypisać do niego wartość. Wskaźniki (podobnie jak typy referencyjne) mogą mieć wartość `null`. Jest to ich wartość domyślna. Wartość zapisana we wskaźniku to adres. Dlatego aby przypisać wartość do wskaźnika, najpierw trzeba pobrać adres danych.

Dozwolone jest jawnie rzutowanie liczb typu `int` lub `long` na wskaźnik, jednak tę technikę stosuje się rzadko, jeśli nie są dostępne mechanizmy określania adresu konkretnej wartości w czasie wykonywania programu. Zamiast tego należy zastosować operator adresu (&), by pobrać adres wartości typu bezpośredniego:

```
byte* pData = &bytes[0]; // Błąd kompilacji.
```

Problem polega na tym, że w środowisku zarządzanym dane mogą zmieniać lokalizację, co powoduje unieważnienie poprzedniego adresu. Komunikat wyświetlany po uruchomieniu pokazanego błędnego kodu informuje, że „adres wyrażenia o niestałej lokalizacji można pobrać tylko w instrukcji `fixed`”. W tym kodzie docelowy bajt znajduje się w tablicy, a tablica jest typu referencyjnego (czyli może zmieniać lokalizację). Wartości typów referencyjnych są przechowywane na stercie i podlegają odzyskiwaniu pamięci oraz mogą być przenoszone. Podobny problem występuje, gdy używane jest pole typu bezpośredniego w typie umożliwiającym przenoszenie danych:

```
int* a = &"message".Length;
```

Aby przypisać adres danych, spełnione muszą być następujące warunki:

- Dane muszą mieć postać zmiennej.
- Dane muszą być typu niezarządzanego.
- Zmienna musi mieć trwałą lokalizację (nie może być przenoszona).

Jeśli dane to zmienna typu niezarządzanego, która jednak może być przenoszona, zastosuj instrukcję `fixed`, aby zapewnić stałą lokalizację zmiennej.

Zapewnianie stałej lokalizacji danych

Aby pobrać adres danych umożliwiających przenoszenie, trzeba najpierw zapewnić im stałą lokalizację. Na listingu 23.14 pokazano, jak to zrobić.

Listing 23.14. Instrukcja `fixed`

```
byte[] bytes = new byte[24];
fixed (byte* pData = &bytes[0]) // Dozwolony jest też zapis pData = bytes.
{
    // ...
}
```

Dane przypisywane w bloku kodu podanym po instrukcji `fixed` nie zmieniają lokalizacji. Ta tablica `bytes` zachowuje ten sam adres (przynajmniej do końca bloku po instrukcji `fixed`).

Instrukcja `fixed` wymaga zadeklarowania w jej zasięgu zmiennej przechowującej wskaźnik. Pozwala to uniknąć dostępu do zmiennej poza zasięgiem tej instrukcji, gdzie dane nie muszą już mieć stałej lokalizacji. Programista odpowiada za to, by nie przypisać wskaźnika (na przykład w wyniku wywołania jakiegoś interfejsu API) do innej zmiennej, używanej także poza zasięgiem instrukcji `fixed`. Nazwa „niezabezpieczony kod” nie powstała bez przyczyny. Programista piszący taki kod musi sam zadbać o bezpieczne używanie wskaźników i nie może polegać na tym, że zrobi to za niego środowisko uruchomieniowe. W niezabezpieczonym kodzie także stosowanie parametrów `ref` i `out` jest problematyczne, jeśli dane nie są zachowywane poza wywołaniem metody.

Ponieważ `string` nie może być typem docelowym, definiowanie wskaźników do łańcuchów znaków na pozór jest niemożliwe. Jednak, podobnie jak w języku C++, typ `string` wewnętrznie przechowuje wskaźnik do pierwszego znaku z ich tablicy. Dlatego można zadeklarować wskaźnik do znaków za pomocą typu `char*`. Tak więc język C# umożliwia zadeklarowanie wskaźnika typu `char*` i powiązanie go z łańcuchem znaków w instrukcji `fixed`. Instrukcja `fixed` chroni przed przeniesieniem łańcucha znaków w czasie używania takiego wskaźnika. W podobny sposób można tworzyć w instrukcji `fixed` wskaźniki do innych typów pozwalających na przenoszenie danych, pod warunkiem jednak, że możliwa jest niejawną konwersja na wskaźnik dozwolonego typu.

Dłuższe przypisanie `&bytes[0]` można zastąpić skróconą formą `bytes`, co pokazano na listingu 23.15.

Listing 23.15. Instrukcja **fixed** bez operatora adresu i indeksu tablicy

```
byte[] bytes = new byte[24];
fixed (byte* pData = bytes)
{
    // ...
}
```

W zależności od częstotliwości wywołań i czasu ich wykonywania instrukcje **fixed** mogą powodować fragmentację na stercie, ponieważ mechanizm odzyskiwania pamięci nie potrafi kompaktować obiektów o stałej lokalizacji. Aby ograniczyć ten problem, warto utworzyć stałą lokalizację bloków na początku wykonywania kodu i stosować omawianą technikę do niewielkiej liczby dużych bloków, a nie do licznych małych bloków. Niestety, jest to niezgodne z zaleceniem zapewniania stałej lokalizacji jak najmniejszych bloków danych na jak najkrótszy czas, co minimalizuje ryzyko wystąpienia cyklu odzyskiwania pamięci w czasie, gdy lokalizacja danych nie może się zmieniać. W platformie .NET 2.0 w pewnym stopniu ograniczono ten problem dzięki dodaniu kodu uwzględniającego możliwość fragmentacji.

Możliwe, że chcesz zapewnić stałą lokalizację obiektu w ciele jednej metody i nie zmieniać tej lokalizacji do czasu wywołania innej metody. Instrukcja **fixed** tego nie umożliwia. Jeśli przytrafi Ci się ta niewygodna sytuacja, możesz wykorzystać metody z typu **GCHandle**, aby lokalizacja obiektu nigdy się nie zmieniała. Tę technikę należy jednak stosować tylko w ostatczności. Wymuszanie stałej lokalizacji obiektu na długim czasie sprawia, że mechanizm odzyskiwania pamięci prawdopodobnie nie będzie mógł wydajnie jej skompaktować.

Alokowanie danych na stosie

Można zastosować instrukcję **fixed** do tablicy, by zapobiec przenoszeniu danych przez mechanizm odzyskiwania pamięci. Inną możliwość to zaalokowanie tablicy na stosie wywołań. Dane zaalokowane na stosie nie są uwzględniane przez mechanizm odzyskiwania pamięci ani przez wzorce oparte na finalizatorze. Tablica alokowana na stosie (za pomocą modyfikatora **stackalloc**) musi zawierać elementy typów niezarządzanych; jest to podobne ograniczenie jak w przypadku typów docelowych. Zamiast alokować tablicę bajtów na stercie, można więc zapisać ją na stosie wywołań, co pokazano na listingu 23.16.

Listing 23.16. Alokowanie danych na stosie wywołań

```
byte* bytes = stackalloc byte[42];
```

Ponieważ typ danych to tablica elementów typu niezarządzanego, środowisko uruchomieniowe może zaalokować na potrzeby tablicy bufor o stałej pojemności, a następnie odzyskać pamięć po wyjściu wskaźnika z zasięgu programu. Alokowany jest blok o wielkości **sizeof(T) * E**, gdzie E to liczba elementów, a T to docelowy typ. Ponieważ modyfikator **stackalloc** można stosować tylko do tablic elementów typów niezarządzanych, środowisko uruchomieniowe przywraca pamięć systemowi, rozwijając stos. Pozwala to wyeliminować skomplikowane iterowanie po kolejce finalizacji (zobacz podrozdział „Odzyskiwanie pamięci” w rozdziale 10. i omówienie finalizacji) i kompaktowanie potrzebnych jeszcze danych. Nie ma sposobu na jawne zwolnienie danych zaalokowanych za pomocą modyfikatora **stackalloc**.

Stos to cenny zasób. Choć ma małą pojemność, jej wyczerpanie ma poważne skutki — program przestaje działać. Dlatego należy dołożyć wszelkich starań, by zapobiegać wyczerpaniu się pamięci na stosie. Jeśli programowi zabraknie miejsca na stosie, najlepszym rozwiązaniem jest natychmiastowe zamknięcie programu. Programy zwykle mają stos o pojemności megabajta (a często znacznie mniejszej). Dlatego zachowaj dużą ostrożność i unikaj alokowania na stosie buforów o nieokreślonym rozmiarze.

Dereferencja wskaźników

Dostęp do danych zapisanych w zmiennej typu docelowego wymaga dereferencji wskaźnika. W tym celu należy umieścić przed wyrażeniem operator dereferencji. Na przykład instrukcja `byte data = *pData;` powoduje dereferencję lokalizacji typu `byte` powiązanej ze wskaźnikiem `pData` i zwraca zmienną typu `byte`. Otrzymana zmienna umożliwia dostęp w trybie odczytu i zapisu do jednej wartości typu `byte` z danej lokalizacji.

Zastosowanie tej techniki w niezabezpieczonym kodzie umożliwia wykonywanie nie-standardowych operacji, na przykład modyfikację „niemodyfikowalnych” łańcuchów znaków, co pokazano na listingu 23.17. Takie rozwiązanie nie jest oczywiście zalecane, choć umożliwia niskopoziomowe zarządzanie pamięcią.

Listing 23.17. Modyfikowanie „niemodyfikowalnych” łańcuchów znaków

```
string text = "S5280ft";
Console.WriteLine("{0} = ", text);
unsafe // Wymaga opcji /unsafe.
{
    fixed (char* pText = text)
    {
        char* p = pText;
        *++p = 'm';
        *++p = 'i';
        *++p = 'l';
        *++p = 'e';
        *++p = ' ';
        *++p = ' ';
    }
}
Console.WriteLine(text);
```

Wynik działania kodu z listingu 23.17 pokazano w danych wyjściowych 21.3.

DANE WYJŚCIOWE 23.3.

```
S5280ft = Smile
```

Przedstawiony kod przyjmuje pierwotny adres i zwiększa go o rozmiar typu docelowego (`sizeof(char)`). Używany jest do tego operator preinkrementacji. Dalej następuje dereferencja adresu (za pomocą operatora dereferencji) i przypisanie nowego znaku do uzyskanej lokalizacji. Zastosowanie operatorów `+ i -` pozwala zmienić adres o `sizeof(T)`, gdzie `T` to typ docelowy.

Do porównywania wskaźników można zastosować operatory porównywania (`==, !=, <, >, <=, >=`), które w tym kontekście porównują wartości adresów.

Operator dereferencji nie umożliwia dereferencji wartości typu `void*`. Typ `void*` reprezentuje wskaźnik do wartości nieznanego typu. Ponieważ typ danych nie jest znany, nie można przeprowadzić dereferencji, aby uzyskać zmienną. Aby uzyskać dostęp do danych wskazywanych za pomocą wskaźnika typu `void*`, trzeba najpierw przekształcić go na wskaźnik innego typu, a potem przeprowadzić dereferencję z użyciem tego typu.

Efekt pokazany na listingu 23.17 można uzyskać nie tylko przy użyciu operatora dereferencji, ale też za pomocą operatora indeksu (zobacz listing 23.18).

Listing 23.18. Modyfikowanie „niemodyfikowanego” łańcucha znaków w niezabezpieczonym kodzie za pomocą operatora indeksu

```
string text;
text = "S5280ft";
Console.WriteLine("{0} = ", text);

unsafe // Wymaga użycia opcji /unsafe.
{
    fixed (char* pText = text)
    {
        pText[1] = 'm';
        pText[2] = 'i';
        pText[3] = 'l';
        pText[4] = 'e';
        pText[5] = ' ';
        pText[6] = ' ';
    }
}
Console.WriteLine(text);
```

Wynik działania kodu z listingu 23.18 pokazano w danych wyjściowych 23.4.

DANE WYJŚCIOWE 23.4.

```
S5280ft = Smile
```

Modyfikacje takie jak przeprowadzone na listingach 23.17 i 23.18 mogą prowadzić do nieoczekiwanej działania kodu. Jeśli po wykonaniu instrukcji `Console.WriteLine()` ponownie przypiszesz do zmiennej `text` wartość `"S5280ft"` i jeszcze raz wyświetlisz zawartość tej zmiennej, i tak zobaczyś napis `Smile`, ponieważ w wyniku optymalizacji dwa takie same litery tekstowe są przekształcane w jeden literal wskazywany przez obie zmienne. Założmy, że na listingu 23.17 po niezabezpieczonym kodzie znajduje się następujące przypisanie:

```
text = "S5280ft";
```

Jednak wewnętrznie zmieniona `text` jest już powiązana z adresem lokalizacji ze zmodyfikowaną wersją napisu `"S5280ft"`, dlatego zmieniona nigdy nie jest ustawiana na oczekiwana wartość.

Dostęp do składowych typu docelowego

Dereferencja wskaźnika prowadzi do uzyskania zmiennej typu docelowego użytego w danym wskaźniku. Dostęp do składowych tego typu można uzyskać w standardowy sposób, za pomocą operatora kropki. Jednak priorytety operatorów sprawiają, że instrukcja `*x.y` oznacza `*(x.y)`,

co zwykle nie jest zgodne z oczekiwaniami programisty. Dlatego jeśli `x` to wskaźnik, po prawny zapis wygląda tak: `(*x).y`. Ta składnia nie wygląda atrakcyjnie. Aby ułatwić dostęp do składowych dostępnych po dereferencji wskaźnika, w języku C# dodano specjalny operator dostępu do składowych — `x->y`. Jest to skrótowy zapis wyrażenia `(*x).y`, co pokazano na listingu 23.19.

Listing 23.19. Bezpośredni dostęp do składowych typu docelowego

```
unsafe
{
    Angle angle = new Angle(30, 18, 0);
    Angle* pAngle = &angle;
    System.Console.WriteLine("{0}° {1}' {2}\"",
        pAngle->Hours, pAngle->Minutes, pAngle->Seconds);
}
```

Wynik działania kodu z listingu 23.19 pokazano w danych wyjściowych 23.5.

DANE WYJŚCIOWE 23.5.

```
30° 18' 0
```

Wykonywanie niezabezpieczonego kodu za pomocą delegata

Na początku rozdziału wspomniano, że w końcowej części znajdzie się kompletny, działający przykład z ilustracją najbardziej „niebezpiecznej” rzeczy, jaką można zrobić w języku C#. Przykładowy kod pobiera wskaźnik do bloku pamięci, zapełnia ten blok bajtami kodu maszynowego, tworzy delegat powiązany z tym nowym kodem, a następnie go uruchamia. Pokazany tu fragment kodu asemblerowego określa identyfikator procesora. Jeśli używany jest system Windows, kod wyświetla ten identyfikator. Kod przedstawiono na listingu 23.20.

Listing 23.20. Oznaczanie bloku z niezabezpieczonym kodem

```
using System;
using System.Runtime.InteropServices;
using System.Text;

class Program
{
    public unsafe delegate void MethodInvoker(byte* buffer);

    public unsafe static int ChapterMain()
    {
        if (RuntimeInformation.IsOSPlatform(OSPlatform.Windows))
        {
            unsafe
            {
                byte[] codeBytes = new byte[] {
                    0x49, 0x89, 0xd8, // mov %rbx,%r8
```

```

        0x49, 0x89, 0xc9,    // mov %rcx,%r9
        0x48, 0x31, 0xc0,    // xor   %rax,%rax
        0x0f, 0xa2,          // cpuid
        0x4c, 0x89, 0xc8,    // mov   %r9,%rax
        0x89, 0x18,          // mov   %ebx,0x0(%rax)
        0x89, 0x50, 0x04,    // mov   %edx,0x4(%rax)
        0x89, 0x48, 0x08,    // mov   %ecx,0x8(%rax)
        0x4c, 0x89, 0xc3,    // mov   %r8,%rbx
        0xc3                 // retq
    };
byte[] buffer = new byte[12];

using (VirtualMemoryPtr codeBytesPtr =
    new VirtualMemoryPtr(codeBytes.Length))
{
    Marshal.Copy(
        codeBytes, 0,
        codeBytesPtr, codeBytes.Length);

    MethodInvoker method =
        Marshal.GetDelegateForFunctionPointer<MethodInvoker>(codeBytesPtr);
    fixed (byte* newBuffer = &buffer[0])
    {
        method(newBuffer);
    }
    Console.Write("Identyfikator procesora: ");
    Console.WriteLine(ASCIIEncoding.ASCII.GetChars(buffer));
} // Niezabezpieczony kod.
}
else
{
    Console.WriteLine("Ten przykład jest poprawny tylko w systemie Windows");
}
return 0;
}
}

```

Wynik działania kodu z listingu 23.20 pokazano w danych wyjściowych 23.6.

DANE WYJŚCIOWE 23.6.

Identyfikator procesora: GenuineIntel

Podsumowanie

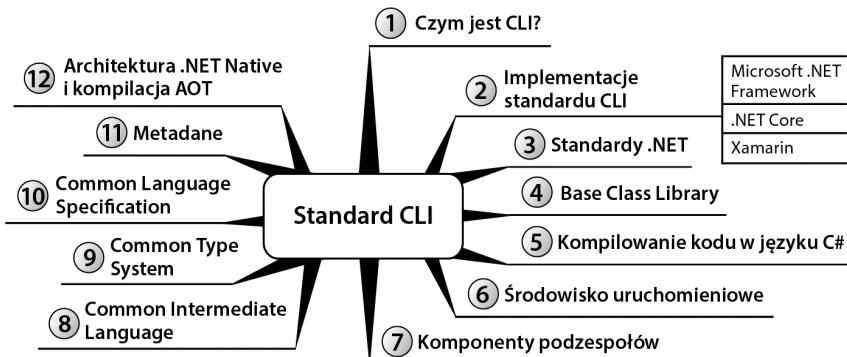
W całej książce pokazano, że język C# zapewnia bardzo duże możliwości, elastyczność, spójność i fantastyczną strukturę. W tym rozdziale zwrócono uwagę na możliwość wykonywania w języku C# niskopoziomowych operacji na kodzie maszynowym.

Przed zakończeniem książki w rozdziale 24. pokrótko opisano platformę wykonywania kodu. Koncentrujemy się tam w mniejszym stopniu na języku C#, a w większym na ogólnej platformie, w której działają pisane w nim programy.

24

Standard CLI

Jedną z pierwszych rzeczy (obok składni), z jaką stykają się programiści używający języka C#, jest kontekst wykonywania programów. W tym rozdziale wyjaśniono alokację i zwalnianie pamięci, sprawdzanie typów, współdziałanie z innymi językami, wykonywanie kodu w różnych systemach i obsługę metadanych. Ten rozdział opisuje więc standard **CLI** (ang. *Common Language Infrastructure*), z którego język C# korzysta zarówno na etapie komplikacji, jak i w czasie wykonywania kodu. W rozdziale omówiono silnik wykonawczy, który zarządza programami w języku C# w czasie ich wykonywania. Wyjaśniono też miejsce języka C# w większym zbiorze języków zarządzanych przez ten sam silnik wykonawczy. Z powodu bliskiego związku języka C# z ogólną infrastrukturą większość oferowanych przez nią funkcji jest dostępna także w języku C#.



Definiowanie standardu CLI

Zamiast generować instrukcje, które procesor może bezpośrednio interpretować, kompilator języka C# generuje polecenia w języku pośrednim CIL (ang. *Common Intermediate Language*). Drugi etap komplikacji, mający miejsce zwykle w czasie wykonywania programu, polega na konwersji kodu CIL na zrozumiałą dla procesora kod maszynowy. Jednak nawet konwersja na kod maszynowy nie wystarcza do wykonania programu. Program w języku C#

musi być wykonywany w kontekście agenta. Agentem odpowiedzialnym za zarządzanie wykonywaniem programu w języku C# jest system VES (ang. *Virtual Execution System*), zwykle nazywany **środowiskiem uruchomieniowym**. Środowisko uruchomieniowe odpowiada za wczytywanie i uruchamianie programów, a także udostępnia wykonywanym programom dodatkowe usługi (zabezpieczenia, odzyskiwanie pamięci itd.).

Specyfikacje języka CIL i środowiska uruchomieniowego znajdują się w międzynarodowej standardzie **CLI** (ang. *Common Language Infrastructure*)¹. CLI to najważniejsza specyfikacja pomagająca zrozumieć kontekst działania programów w języku C# oraz to, jak mogą one płynnie współpracować z innymi programami i bibliotekami (nawet tymi napisanymi w innych językach). Zauważ, że CLI nie określa implementacji standardu. Opisuje tylko wymogi dotyczące działania platformy zgodnej z tym standardem. Dzięki temu autorzy implementacji standardu CLI mają swobodę wprowadzania innowacji (gdy jest to potrzebne), a jednocześnie zachowana zostaje struktura sprawiająca, że programy utworzone z wykorzystaniem jednej wersji platformy mogą działać także w innej implementacji standardu CLI, a nawet w innym systemie operacyjnym.

■ Uwaga

Zwróć uwagę na podobieństwa między akronimami CIL i CLI oraz reprezentowanymi przez nie nazwami. Jeśli teraz zrozumiesz, czego dotyczą te nazwy, łatwiej będzie Ci uniknąć pomyłek w przyszłości.

W standardzie CLI opisane są specyfikacje następujących elementów:

- Virtual Execution System,
- Common Intermediate Language,
- Common Type System,
- Common Language Specification,
- metadanych,
- platformy.

Ten rozdział ma poszerzyć wiedzę Czytelników na temat języka C# o standard CLI, który jest bardzo ważny ze względu na działanie programów napisanych w tym języku i komunikowanie się ich z innymi aplikacjami oraz systemem operacyjnym.

Implementacje standardu CLI

Głównymi implementacjami standardu CLI są obecnie: .NET Core, działająca w systemie Windows, a także w systemach UNIX, Linux i macOS; .NET Framework dla systemu Windows; Xamarin, która ma być uniwersalną platformą przeznaczoną między innymi dla systemów

¹ W tym rozdziale akronim CLI oznacza architekturę wspólnego języka (ang. *Common Language Infrastructure*), a nie interfejs uruchamiany w wierszu poleceń taki jak Dotnet CLI.

iOS, macOS i Android. Każda implementacja standardu CLI obejmuje kompilator języka C# i zestaw bibliotek klas. W poszczególnych implementacjach obsługiwane są różne wersje języka C# i różne zestawy klas w bibliotekach. Wiele implementacji ma obecnie znaczenie czysto historyczne. Wybrane implementacje opisano w tabeli 24.1.

Tabela 24.1. Implementacje standardu CLI

Kompilator	Opis
Microsoft .NET Framework	Ta tradycyjna (i pierwsza) wersja środowiska CLR służy do tworzenia aplikacji działających w systemie Windows. Obejmuje obsługę technologii Windows Presentation Foundation, Windows Forms i ASP.NET. Używa biblioteki Base Class Library z platformy .NET.
.NET Core i CoreCLR	Projekt .NET Core, jak wskazuje nazwa, obejmuje podstawowe mechanizmy wspólnie wszystkim nowym implementacjom platformy .NET. .NET Core jest działającą w różnych systemach otwartą implementacją platformy .NET, zaprojektowaną na potrzeby tworzenia wysoce wydajnych aplikacji. CoreCLR to implementacja środowiska CLR dla omawianego projektu. W czasie powstawania tej książki udostępniono wersję .NET Core 3.1 dla systemów Windows, macOS, Linux, FreeBSD i NetBSD. Jednak niektóre API, na przykład WPF, działają tylko w systemie Windows. Więcej informacji znajdziesz na stronie https://github.com/dotnet/coreclr .
Xamarin	Jest to zestaw narzędzi programistycznych i działających w różnych systemach bibliotek platformy .NET, a także implementacja środowiska CLR. Xamarin pomaga programistom tworzyć aplikacje działające w systemach Microsoft Windows, iOS, macOS i Android przy bardzo wysokim poziomie ponownego wykorzystania kodu. W Xamarinie używana jest biblioteka Mono BCL.
Microsoft Silverlight	Ta dawna implementacja standardu CLI działała w różnych systemach i służyła do tworzenia aplikacji klienckich pracujących w przeglądarce. W 2013 roku Microsoft zaprzestał rozwijania technologii Silverlight.
.NET Compact Framework	Jest to uproszczona implementacja platformy .NET zaprojektowana z myślą o palmtopach, telefonach i konsoli Xbox 360. Biblioteka XNA i narzędzia służące do budowania aplikacji na konsolę Xbox 360 są oparte na implementacji Compact Framework 2.0. W 2013 roku Microsoft zaprzestał rozwijania biblioteki XNA.
.NET Micro Framework	Micro Framework to rozwijana przez Microsoft otwarta implementacja standardu CLI przeznaczona dla urządzeń na tyle niewydajnych, że nie radzą sobie z implementacją Compact Framework.
Mono	Mono to otwarta implementacja standardu CLI działająca w wielu systemach. Jest przeznaczona dla licznych systemów uniksowych, dla systemów operacyjnych z urządzeń przenośnych (na przykład dla systemu Android), a także dla konsoli do gier (takich jak PlayStation i Xbox).
DotGNU Portable.NET	Ten projekt miał prowadzić do utworzenia implementacji standardu CLI działającej w różnych systemach, został jednak zawieszony w 2012 roku.
Shared Source CLI (Rotor)	W latach od 2001 do 2006 Microsoft udostępniał na licencji shared source referencyjne implementacje standardu CLI przeznaczone do użytku niekomercyjnego.

Choć ta lista jest długa, jeśli wziąć pod uwagę ilość pracy potrzebnej do zaimplementowania środowiska CLI, najważniejsze są trzy platformy.

Microsoft .NET Framework

Platforma **Microsoft .NET Framework** była pierwszą implementacją standardu CLI; udostępniono ją w lutym 2000 r. Dlatego jest to najdojrzalsza platforma z największym zestawem API. Obsługuje budowanie aplikacji sieciowych, konsolowych i klienckich dla systemu Microsoft Windows. Największym ograniczeniem platformy .NET Framework jest to, że działa tylko w systemie Microsoft Windows — co więcej, jest wbudowana w ten system. Platforma Microsoft .NET Framework ma wiele komponentów. Oto najważniejsze z nich:

- **.NET Framework Base Class Library (BCL).** Zapewnia typy reprezentujące wbudowane typy standardu CLI. Dostępne typy obsługują plikowe operacje wejścia – wyjścia, podstawowe kolekcje, niestandardowe atrybuty, przetwarzaniełańcuchów znaków itd. Biblioteka BCL udostępnia definicje wszystkich typów natywnych z języka C#, takich jak `int` i `string`.
- **ASP.NET.** Służył do tworzenia witryn i internetowych API. Ten komponent od czasu udostępnienia go (w 2002 r.) stanowi podstawę witryn opartych na technologiach Microsoftu. Powoli jest jednak wypierany przez jego następcę, ASP.NET Core, który działa w różnych systemach, zapewnia znacznie wyższą wydajność i obejmuje zaktualizowane API ze spójnymi wzorcami programowania.
- **Windows Presentation Foundation (WPF).** Jest to platforma z graficznym interfejsem użytkownika służąca do budowania aplikacji z rozbudowanym interfejsem działającym w systemie Microsoft Windows. WPF udostępnia nie tylko zestaw komponentów interfejsu użytkownika, ale też język deklaratywny **XAML** (ang. *eXtended Application Markup Language*), który umożliwia hierarchiczne definiowanie interfejsu użytkownika aplikacji.

Zamiast nazwy Microsoft .NET Framework często stosowane jest prostsze określenie **.NET Framework**.

.NET Core

.NET Core to stale rozwijana i działająca w różnych systemach implementacja standardu CLI. Jest to też otwarta wersja platformy .NET Framework zaprojektowana z myślą o wysokości wydajności i pracy w różnych systemach.

.NET Core obejmuje środowisko uruchomieniowe .NET Core Runtime (Core CLR), biblioteki platformy .NET Core, a także używane w wierszu polecen narzędzi Dotnet, które pozwalają tworzyć i kompilować rozmaite aplikacje. Te komponenty znajdują się w pakiecie .NET Core SDK. Jeśli wykonywałeś przykłady z tej książki, używałeś już narzędzi .NET Core i Dotnet.

API platformy .NET Core jest zgodne z istniejącymi implementacjami .NET Framework, Xamarin i Mono dzięki specyfikacji .NET Standard, opisanej szczegółowo dalej w rozdziale.

Obecnie twórcy platformy .NET Core kładą nacisk na budowanie wydajnych i przenośnych aplikacji konsolowych, a także aplikacji dla ASP.NET Core i Universal Windows Platform (UWP) z systemu Windows 10. Wraz z dodawaniem obsługi kolejnych systemów operacyjnych na podstawie .NET Core powstają coraz to nowe platformy.

Xamarin

To działające w różnych systemach narzędzie programistyczne umożliwia tworzenie interfejsów użytkownika aplikacji dla systemów Android, macOS i iOS. Od czasu pojawienia się specyfikacji .NET Standard 2.0 pozwala też tworzyć aplikacje **Universal Windows Application**, zgodne z technologiami Windows 10, Xbox One i HoloLens. Platforma Xamarin jest tak wartościowa przede wszystkim dlatego, że ten sam kod bazowy można wykorzystać do tworzenia natywnie wyglądających interfejsów użytkownika dla wszystkich obsługiwanych systemów operacyjnych.

Specyfikacja .NET Standard

W przeszłości dość trudno było pisać w języku C# biblioteki kodu, z których można by było korzystać w różnych systemach operacyjnych, a nawet w różnych platformach .NET z tego samego systemu. Problem wynikał stąd, że API poszczególnych platform udostępniały inne klasy (i metody w tych klasach). Specyfikacja .NET Standard rozwiązuje problem, ponieważ definiuje wspólny zestaw API, jaki każda platforma musi udostępniać, aby być zgodna z daną wersją specyfikacji. Ta jednorodność gwarantuje, że programiści mają dostęp do spójnego zestawu API we wszystkich platformach .NET zgodnych z docelową wersją specyfikacji .NET Standard. Jeśli chcesz napisać podstawowy kod aplikacji raz i mieć pewność, że można go użyć we wszystkich nowych implementacjach platformy .NET, najłatwiej jest utworzyć projekt biblioteki .NET Standard (dostępny jako typ projektu w środowisku Visual Studio 2017 lub jako szablon biblioteki klas w Dotnet CLI). Kompilator .NET Core gwarantuje, że kod takiej biblioteki będzie używał wyłącznie klas i metod dostępnych w docelowej wersji specyfikacji .NET Standard.

Twórcy biblioteki klas powinni starannie przemyśleć wybór wersji specyfikacji. Im nowsza wersja, tym mniej trzeba się zajmować pisaniem własnych implementacji API, które mogą być niedostępne w starszych wersjach specyfikacji .NET Standard. Wadą nowszych wersji jest to, że są obsługiwane przez mniejszą liczbę implementacji platformy .NET. Na przykład jeśli chcesz, aby biblioteka była zgodna z platformą .NET Core 1.0, musisz wybrać specyfikację .NET Standard 1.6, dlatego nie będziesz mieć dostępu do wszystkich API dla mechanizmu refleksji dostępnych w platformie Microsoft .NET Framework. Oto krótki opis nieuniknionego kompromisu: wybierz nowsze wersje specyfikacji .NET Standard, jeśli jesteś leniwy, a starsze, jeżeli obsługa różnych platform jest dla Ciebie ważniejsza niż uproszczenie swojej pracy.

Więcej informacji, w tym na temat powiązania między implementacjami platformy .NET i ich wersjami a wersjami specyfikacji .NET Standard, znajdziesz na stronie <http://itl.tc/NETStandard>.

Biblioteka BCL

Oprócz udostępniania środowiska uruchomieniowego, w którym można wykonywać kod w języku CIL, standard CLI definiuje zestaw podstawowych bibliotek klas, jakie można stosować w programach. Ten zestaw to **Base Class Library** (BCL). Biblioteki zawarte w BCL zapewniają podstawowe typy i API oraz umożliwiają programom jednolite interakcje ze

środowiskiem uruchomieniowym i systemem operacyjnym. BCL zapewnia obsługę kolekcji, prosty dostęp do plików, zabezpieczenia, podstawowe typy danych (m.in. string), strumienie itd.

Podobnie oferowana przez Microsoft biblioteka **Framework Class Library (FCL)** pozwala tworzyć rozbudowane klienckie interfejsy użytkownika i interfejsy aplikacji sieciowych, zapewnia dostęp do baz danych, obsługuje komunikację w środowisku rozproszonym itd.

Kompilacja kodu w języku C# na kod maszynowy

Program HelloWorld z listingu z rozdziału 1. jest napisany w języku C#. W celu wykonania kodu należy go skompilować za pomocą kompilatora tego języka. Jednak procesor nie potrafi bezpośrednio zinterpretować skompilowanego kodu w języku C#. Potrzebny jest dodatkowy krok, by przekształcić efekt komplikacji kodu w języku C# na kod maszynowy. Ponadto w trakcie wykonywania kodu potrzebny jest agent, który udostępnia usługi programowi w języku C#. Chodzi tu o usługi, których programista nie musiał jawnie programować.

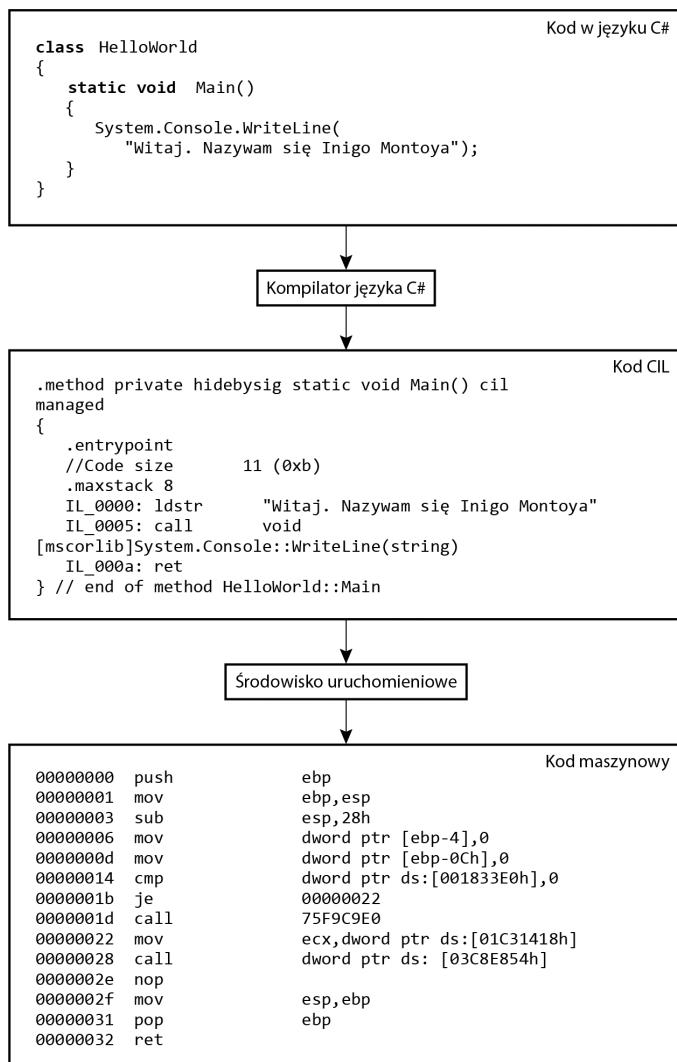
Wszystkie języki komputerowe określają składnię i semantykę na potrzeby programowania. Ponieważ języki takie jak C i C++ są kompilowane do kodu maszynowego, platformą ich uruchamiania jest używany system operacyjny (na przykład Microsoft Windows, Linux, macOS itd.) i zestaw dostępnych w nim instrukcji maszynowych. Natomiast dla języków takich jak C# platformą uruchamiania jest środowisko uruchomieniowe (VES).

Kompilator języka C# w wyniku komplikacji generuje kod CIL. **CIL** to akronim od *Common Intermediate Language*, czyli wspólny język pośredni. Nazwa pochodzi od tego, że potrzebny jest dodatkowy pośredni krok w celu przekształcenia kodu CIL na postać zrozumiałą dla procesorów. Proces przekształcania przedstawiono na rysunku 24.1.

Kompilacja kodu w języku C# wymaga więc dwóch kroków:

1. Konwersji z kodu C# na kod CIL wykonywanej przez kompilator języka C#.
2. Konwersji z kodu CIL na instrukcje, które mogą zostać wykonane przez procesor.

Środowisko uruchomieniowe potrafi zrozumieć instrukcje w kodzie CIL i skompilować je do postaci kodu maszynowego. Za komplikację kodu CIL do kodu maszynowego odpowiada **komponent** środowiska uruchomieniowego. Tym komponentem jest **kompilator JIT** (ang. *just-in-time*). Kompilacja JIT ma miejsce, gdy program jest instalowany lub wykonywany. W większości implementacji standardu CLI preferowana jest komplikacja kodu CIL w czasie wykonywania programu, jednak standard CLI nie określa, kiedy komplikacja ma mieć miejsce. Standard CLI umożliwia nawet interpretowanie kodu CIL (zamiast kompilowania go); podobnie działa wiele języków skryptowych. Ponadto platforma .NET udostępnia narzędzie NGEN, umożliwiające komplikację programu do postaci kodu maszynowego przed uruchomieniem go. Taka komplikacja musi być wykonywana na komputerze, na którym program będzie uruchamiany, ponieważ uwzględniane są przy tym cechy maszyny (procesora, pamięci itd.), co pozwala wygenerować wydajniejszy kod. Zaletą stosowania narzędzia NGEN w trakcie instalowania programu (lub w dowolnym momencie przed jego uruchomieniem) jest to, że nie trzeba uruchamiać kompilatora JIT w momencie włączania aplikacji, co przyspiesza ten proces.



Rysunek 24.1. Kompilacja kodu w języku C# do kodu maszynowego

W środowisku Visual Studio 2015 kompilator języka C# obsługuje też komplikację w trybie .NET Native. Polega to na komplikacji kodu w języku C# do natywnego kodu maszynowego w trakcie tworzenia instalowanej wersji aplikacji (proces ten przypomina korzystanie z narzędzia NGEN). Ten mechanizm jest wykorzystywany w aplikacjach w technologii Windows Universal.

Środowisko uruchomieniowe

Nawet gdy środowisko uruchomieniowe przekształci już kod CIL na kod maszynowy i zarazem go wykonywać, nadal kontroluje pracę programu. Kod wykonywany w kontekście agenta (takiego jak środowisko uruchomieniowe) to **kod zarządzany**. Proces wykonywania programu pod kontrolą środowiska uruchomieniowego to **wykonywanie zarządzane**.

Kontrola nad wykonaniem programu dotyczy także danych. Nazywa się je **danymi zarządzanymi**, ponieważ pamięć, w której się znajdują, jest automatycznie alokowana i zwalniana przez środowisko uruchomieniowe.

O określenie *Common Language Runtime (CLR)*, czyli środowisko uruchomieniowe wspólnego języka, jest nieco mylące, ponieważ technicznie nie jest to uniwersalna nazwa komponentu ze standardu CLI. CLR to specyficzna dla Microsoftu implementacja środowiska uruchomieniowego dla platformy .NET. Jednak nazwa CLR często jest stosowana jako zastępnik określenia *środowisko uruchomieniowe*, a technicznie poprawny termin, czyli *Virtual Execution System*, rzadko pojawia się poza specyfikacją CLI.

Ponieważ to agent kontroluje wykonywanie programu, można wstrzymać do kodu dodatkowe usługi, które nie zostały jawnie zaprogramowane przez programistę. Kod zarządzany obejmuje informacje umożliwiające dodanie takich usług. Dostępne są na przykład: lokalizacja z metadanymi o składowych danego typu, obsługa wyjątków, dostęp do informacji o bezpieczeństwie i możliwość poruszania się po stosie. W dalszej części podrozdziału znajdziesz opis dodatkowych usług oferowanych przez środowisko uruchomieniowe i dostępnych w trakcie wykonywania programu w środowisku zarządzanym. Nie wszystkie te usługi są wymagane w standardzie CLI, jednak w popularnych implementacjach dostępna jest każda z tych usług.

Odzyskiwanie pamięci

Odzyskiwanie pamięci to proces automatycznego zwalniania pamięci na podstawie potrzeb programu. Zwalnianie pamięci stanowi poważny problem programistyczny w językach, które nie zapewniają mechanizmu do automatycznego wykonywania tej operacji. Jeśli mechanizm odzyskiwania pamięci nie jest dostępny, programiści muszą pamiętać, aby zawsze zwalniać zaalokowaną pamięć. Jeśli o tym zapomną (lub przypadkowo ponownie zaalokują ten sam obszar pamięci), spowoduje to wyciekanie pamięci lub uszkodzenie programu, co jest odczuwalne zwłaszcza w programach działających przez długi czas — na przykład w serwerach WWW. Dzięki obsłudze odzyskiwania pamięci wbudowanej w środowisko uruchomieniowe programiści, którzy z niego korzystają, mogą się skoncentrować na dodawaniu funkcji programu, zamiast mierzyć się z „hydrauliką” związaną z zarządzaniem pamięcią.

Porównanie języków — deterministyczne usuwanie obiektów w języku C++

Standard CLI nie zawiera dokładnego opisu działania mechanizmu odzyskiwania pamięci. Dlatego w każdej implementacji tego standardu stosowane może być odmienne podejście. Ponadto sam mechanizm odzyskiwania pamięci też nie jest bezpośrednio wymagany przez standard CLI. Ważną kwestią, o której powinni pamiętać programiści używający języka C++, jest to, że obiekty zwalniane przez mechanizm odzyskiwania pamięci nie zawsze są usuwane **deterministycznie** (czyli w jasno zdefiniowanych i znanych na etapie komplikacji miejscach). Pamięć zajmowana przez obiekty może zostać zwolniona w dowolnym momencie między ostatnim użyciem danego obiektu a zamknięciem programu. Dlatego odzyskiwanie pamięci obiektu może mieć miejsce jeszcze przed wyjściem obiektu z zasięgu, a także długo po ostatnim użyciu danego obiektu w kodzie.

Mechanizm odzyskiwania pamięci odpowiada tylko za zarządzanie pamięcią — nie zapewnia automatycznego systemu zarządzania innymi zasobami. Dlatego jeśli konieczne jest jawne zwolnienie zasobów innych niż pamięć, programista korzystający z tych zasobów powinien wykorzystać specjalne, zgodne ze standardem CLI wzorce programowania, które pomogą zwolnić dane zasoby (zobacz rozdział 10.).

ZAGADNIENIE DLA POCZĄTKUJĄCYCH

Odzyskiwanie pamięci w platformie .NET

W większości implementacji standardu CLI do odzyskiwania pamięci używany jest uwzględniający generacje algorytm „oznacz i usuń” (ang. *mark and sweep*), który kompaktuje zajmowaną pamięć. Uwzględnianie generacji polega na tym, że obiekty używane tylko przez krótki czas są usuwane szybciej niż obiekty, które przetrwały wcześniejsze cykle odzyskiwania pamięci (ponieważ były wtedy używane). Jest to zgodne z ogólnym wzorcem alokowania pamięci. Zgodnie z nim obiekty, które były używane dłużej, powinny „przeżyć” obiekty utworzone niedawno.

Mechanizm odzyskiwania pamięci w platformie .NET posługuje się algorytmem „oznacz i usuń”. W każdym cyklu odzyskiwania pamięci algorytm oznacza obiekty przeznaczone do zwolnienia, a następnie kompaktuje pozostałe obiekty, tak by między nimi nie znajdowała się wolna pamięć. Zastosowanie kompresji w celu zapełnienia miejsca pozostałoego przez zwolnione obiekty często pomaga w szybszym (w porównaniu z kodem niezarządzanym) tworzeniu nowych obiektów, ponieważ nie trzeba wtedy wyszukiwać miejsca na nie w pamięci. Kompaktowanie zmniejsza też ryzyko przełączania stron, ponieważ więcej obiektów mieści się wtedy na jednej stronie. Jest to następny czynnik zwiększający wydajność.

Mechanizm odzyskiwania pamięci uwzględnia zasoby dostępne maszynie i zapotrzebowanie na te zasoby w trakcie wykonywania programu. Na przykład jeśli komputer ma dużą ilość wolnej pamięci, mechanizm odzyskiwania pamięci jest uruchamiany rzadziej, dlatego poświęca mniej czasu na porządkowanie tego zasobu. Ta optymalizacja rzadko jest stosowana w platformach i językach, które nie zapewniają mechanizmu odzyskiwania pamięci.

ZAGADNIENIE DLA POCZĄTKUJĄCYCH

Bezpieczeństwo ze względu na typ

Jedną z najważniejszych zalet środowiska uruchomieniowego jest sprawdzanie konwersji między typami, czyli **kontrola typów**. Dzięki kontroli typów środowisko uruchomieniowe chroni programistów przed przypadkowym błędnym rzutowaniem, które może prowadzić do luk wynikających z przepełnienia bufora. Takie luki to jedna z najczęstszych przyczyn włamań do systemów komputerowych, a automatyczne zapobieganie temu przez środowisko uruchomieniowe jest dużą zaletą². Kontrola typów zapewniana przez środowisko uruchomieniowe gwarantuje, że:

² Chyba że stoisz po drugiej stronie barykady i zależy Ci na tym, by takie luki się pojawiały.

- Obie zmienne i powiązane z nimi dane mają określony typ, a typ zmiennych jest zgodny z typem danych.
- Można przeprowadzić lokalną analizę typu (bez analizowania całego kodu, w którym ten typ jest używany), by ustalić, jakie uprawnienia będą potrzebne do wykonywania składowych z danego typu.
- Każdy typ ma zdefiniowany na etapie komplikacji zestaw metod i danych, jakie może zawierać. Środowisko uruchomieniowe wymusza przestrzeganie reguł określających, które klasy mają dostęp do tych metod i danych. Na przykład metody oznaczone jako prywatne są dostępne tylko dla zawierającego je typu.

■ ■ ■ ZAGADNIENIE DLA ZAAWANSOWANYCH

Omijanie hermetyzacji i modyfikatorów dostępu

Odpowiednie uprawnienia pozwalają ominąć hermetyzację i modyfikatory dostępu za pomocą mechanizmu **refleksji**. Refleksja umożliwia późne wiązanie, ponieważ pozwala sprawdzać składowe typu, wyszukiwać nazwy określonych elementów w metadanych obiektu i wywoływać składowe.

Przenośność między platformami

Programy w języku C# są **przenośne między platformami i umożliwiają wykonywanie kodu w różnych systemach operacyjnych**. Taki kod może działać w wielu systemach operacyjnych i w różnych implementacjach standardu CLI. Przenośność w tym kontekście nie ogranicza się do kodu źródłowego (co wymagałoby ponownego kompilowania go). Moduł zgodny ze standardem CLI skompilowany w jednej platformie powinien bez konieczności ponownej komplikacji działać w dowolnej innej platformie tego rodzaju. Ta przenośność wynika z tego, że za dostosowanie kodu do systemu odpowiada implementacja środowiska uruchomieniowego (dzięki specyfikacji .NET Standard), a nie programista aplikacji. Ograniczeniem jest tu oczywiście to, że nie można stosować żadnych interfejsów API specyficznych dla danej platformy. W trakcie pisania aplikacji przeznaczonej na różne platformy programiści mogą spakować wspólny kod w bibliotekach zgodnych z tymi platformami, a następnie wywoływać ten kod we fragmentach specyficznych dla konkretnych platform. Zmniejsza to łączną ilość kodu potrzebnego w aplikacjach działających w różnych platformach.

Wydajność

Wielu programistów przyzwyczajonych do pisania niezarządzanego kodu słusznie zauważa, że środowiska zarządzane powodują dodatkowe koszty w pracy nawet najprostszych aplikacji. Jednak kosztem wydajności zarządzany kod zapewnia wyższą produktywność programistów i mniejszą liczbę błędów. Podobna sytuacja miała miejsce, gdy twórcy oprogramowania rezygnowali z asemblera na rzecz języków wyższego poziomu (takich jak C) i przechodzili od programowania strukturalnego do programowania obiektowego. W większości sytuacji produktywność programistów okazuje się ważniejsza — zwłaszcza teraz, gdy wydajność

i niższe ceny sprzętu sprawiają, że łatwo można spełnić wymagania aplikacji. Czas poświęcony na projektowanie architektury daje dużo większe szanse (niż manipulowanie skomplikowanym niskopoziomowym kodem) na istotną poprawę wydajności. Czynnikiem dodatkowo zwiększającym atrakcyjność zarządzanego wykonywania programów są zagrożenia związane z lukami bezpieczeństwa wynikającymi z przepelenienia bufora.

W niektórych scenariuszach (na przykład związanych z pracą sterowników urządzeń) zarządzane wykonywanie kodu nie jest odpowiednie. Jednak ponieważ możliwości i zaawansowanie środowisk zarządzanych stale rosną, wiele problemów z wydajnością prawdopodobnie zostanie rozwiążanych. Wykonywanie niezarządzane będzie wtedy potrzebne tylko wtedy, gdy konieczne są precyzyjna kontrola nad kodem lub ominięcie środowiska uruchomieniowego³.

Środowisko uruchomieniowe udostępnia też kilka rozwiązań, które pozwalają poprawić wydajność aplikacji w porównaniu z programami kompilowanymi do kodu natywnego. Na przykład dzięki temu, że środowisko uruchomieniowe przekształca program na kod maszynowy na docelowej maszynie, wynikowy skompilowany kod jest dostosowany do procesora i układu pamięci z tej maszyny. Pozwala to poprawić wydajność w sposób zwykle niemożliwy w językach, dla których nie stosuje się kompilatora JIT. Środowisko uruchomieniowe potrafi też reagować na warunki wykonywania kodu, które rzadko są uwzględniane w programach bezpośrednio kompilowanych do postaci kodu maszynowego. Jeśli komputer ma więcej pamięci, niż jest to wymagane, kod w językach niezarządzanych i tak będzie zwalniał pamięć w deterministycznie określonych na etapie kompilacji miejscach kodu. Języki z kompilatorem JIT mogą zwalniać pamięć tylko wtedy, gdy zaczyna jej brakować, lub w momencie zamykania programu. Choć kompilacja JIT wymaga dodatkowego kroku w procesie wykonywania kodu, kompilator JIT może na tyle poprawić wydajność kodu, że będzie ona porównywalna z programami kompilowanymi bezpośrednio do postaci kodu maszynowego. Tak więc programy oparte na środowisku zgodnym ze standardem CLI niekoniecznie są szybsze od zwykłych aplikacji, ale mają porównywalną wydajność.

Podzespoły, manifesty i moduły

W standardzie CLI opisana jest specyfikacja danych wyjściowych w postaci kodu CIL generowanego przez kompilator źródłowego języka. Te dane wyjściowe to zwykle podzespoły (ang. *assembly*). Podzespoły oprócz instrukcji w języku CIL zawierają też **manifest**, obejmujący następujące komponenty:

- typy definiowane i importowane w podzespołe,
- informacje o wersji podzespołu,
- dodatkowe pliki potrzebne w podzespołe,
- uprawnienia podzespołu związane z bezpieczeństwem.

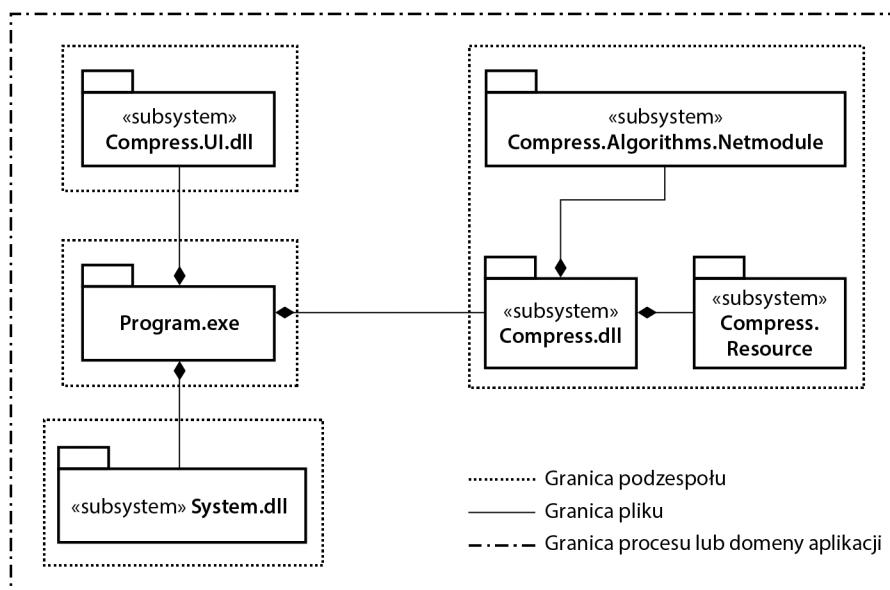
Manifest pełni funkcję nagłówka podzespołu i udostępnia wszystkie informacje na temat tego, z czego podzespoły się składają, oraz pozwala go zidentyfikować.

³ Microsoft sugeruje, że w przyszłości aplikacje dla systemu Windows będą pisane głównie jako programy zarządzane. Ma to dotyczyć nawet aplikacji zintegrowanych z systemem operacyjnym.

Podzespół może być biblioteką klas lub plikiem wykonywalnym. W jednym podzespolu mogą się znajdować referencje do innych podzespołów (a w tych — do jeszcze innych). W ten sposób powstaje aplikacja składająca się z wielu komponentów, a nie jeden duży, monolityczny program. Jest to ważny aspekt, który w nowych platformach programowania jest przyjmowany za standardowe rozwiązanie, ponieważ znacznie ułatwia konserwację kodu i umożliwia wykorzystywanie jednego komponentu w wielu programach.

Oprócz manifestu podzespół zawiera kod CIL w jednym lub kilku modułach. Zwykle podzespoły i manifest są połączone w jednym pliku (tak jak w programie *HelloWorld.exe* z rozdziału 1.). Można jednak umieścić moduły w odrębnych plikach, a następnie zastosować konsolidator podzespołów (program *al.exe*), aby utworzyć plik podzespołu zawierający manifest z referencjami do wszystkich modułów⁴. To podejście nie tylko stanowi dodatkową technikę podziału programu na komponenty, ale też umożliwia zbudowanie jednego podzespołu z wykorzystaniem wielu języków źródłowych.

Pojęcia *moduł* i *podzespół* często stosowane są wymiennie. Jednak nazwa *podzespół* najczęściej pojawia się w kontekście omawiania programów i bibliotek zgodnych ze standardem CLI. Na rysunku 24.2 przedstawiono różne pojęcia z tego obszaru.



Rysunek 24.2. Podzespoły z używanymi w nich modułami i plikami

⁴ Po części wynika to z tego, że Visual Studio .NET (jedno z podstawowych środowisk IDE powiązanych ze standardem CLI) nie umożliwia pracy z podzespołami składającymi się z wielu modułów. Dostępne obecnie wersje środowiska Visual Studio .NET nie udostępniają zintegrowanych narzędzi do budowania podzespołów o wielu modułach, a gdy w środowisku rozwijany jest taki podzespół, mechanizm IntelliSense nie działa w pełni poprawnie.

Zauważ, że w podzespołach i modułach można korzystać z plików takich jak pliki zasobów z tłumaczeniem programu na określony język. Choć zdarza się to rzadko, dwa różne podzespoły mogą korzystać z tego samego modułu lub pliku.

Mimo że podzespoł może obejmować wiele modułów i plików, cała grupa plików ma tylko jeden numer wersji, umieszczany w manifeście podzespołu. Dlatego w aplikacji najmniejszym komponentem umożliwiającym określenie wersji jest właśnie podzespoł (także jeśli składa się z wielu plików). Modyfikacja któregoś z używanych plików (nawet w celu dodania poprawki) bez aktualizacji manifestu podzespołu to naruszenie integralności manifestu i całego podzespołu. Podzespoły stanowią więc logiczny komponent lub logiczną jednostkę instalacji.

Uwaga

To podzespoły (a nie używane w nich poszczególne moduły) są najmniejszą jednostką, dla której można określić wersję i którą można zainstalować.

Choć podzespoł (jako jednostka logiczna) może się składać z wielu modułów, większość podzespołów obejmuje tylko jeden moduł. Microsoft udostępnia narzędzie *ILMerge.exe*, pozwalające połączyć wiele modułów i ich manifestów w jeden plik podzespołu.

Ponieważ manifest zawiera referencje do wszystkich plików potrzebnych w podzespołach, za pomocą manifestu można określić zależności podzespołu. Dlatego w czasie wykonywania programu środowisko uruchomieniowe musi sprawdzić tylko manifest, aby ustalić, które pliki będą potrzebne. Jedyne autorzy narzędzi udostępniający biblioteki współużytkowane przez wiele aplikacji (na przykład Microsoft) muszą rejestrować pliki na etapie ich instalacji. Dzięki temu instalowanie podzespołów jest proste. Często instalowanie aplikacji opartych na standardzie CLI nazywane jest **instalacją xcopy**. Nazwa ta pochodzi od polecenia xcopy z systemu Windows, które kopiuje pliki we wskazane miejsce.

Porównanie języków — rejestrowanie plików DLL w technologii COM

Podzespoły zgodne ze standardem CLI (w odróżnieniu od stosowanych dawniej plików z technologią COM Microsoftu) rzadko wymagają rejestracji. Aplikacje można zainstalować, kopując wszystkie pliki programu do określonego katalogu. Potem można już wykonywać program.

Język Common Intermediate Language

Inną ważną funkcją języka CIL i standardu CLI (obok możliwości przenoszenia kodu źródłowego między wieloma systemami operacyjnymi) jest obsługa interakcji między różnymi językami w ramach tej samej aplikacji, co jest zgodne z nazwą Common Language Infrastructure (czyli architektura wspólnego języka). CIL to język pośredni używany nie tylko dla C#, ale też dla wielu innych języków, takich jak Visual Basic .NET, przypominający Javę J#, niektóre wersje Smalltalka, C++ itd. W czasie, gdy powstawała ta książka, obsługiwanych było ponad

20 języków, w tym wersje COBOLA i FORTRANA. Języki kompilowane do postaci kodu CIL to **języki źródłowe**. Dla każdego z nich używany jest odrębny kompilator, który przekształca język źródłowy na kod CIL. Po komplikacji programu do kodu CIL język źródłowy przestaje mieć znaczenie. Ta ważna cecha umożliwia rozwijanie bibliotek przez różne grupy programistów z różnych organizacji bez martwienia się o to, z jakiego języka korzystają inne grupy. Tak więc CIL umożliwia współdziałanie różnych języków programowania, a także przenośność kodu między systemami operacyjnymi.

Uwaga

Ważną cechą standardu CLI jest obsługa wielu języków. Dzięki temu można rozwijać programy za pomocą wielu języków, a biblioteki napisane w jednym języku są dostępne w kodzie rozwijanym w innych językach.

Common Type System

Niezależnie od używanego języka programowania wynikowy program posługuje się wewnętrznie typami danych. Dlatego w standardzie CLI opisany jest wspólny system typów **CTS** (ang. *Common Type System*). CTS określa strukturę i układ typów w pamięci, a także ich działanie oraz związane z nimi mechanizmy. W standardzie CTS określone są dyrektywy dotyczące manipulowania typami oraz informacje o danych przechowywanych w każdym typie. Standard CTS ma umożliwiać współdziałanie kodu w różnych językach, dlatego określa, jak typy wyglądają i funkcjonują w ramach komunikacji między językami. To środowisko uruchomieniowe w czasie wykonywania programu wymusza przestrzeganie kontraktów opisanych w standardzie CTS.

W standardzie CTS typy są podzielone na dwie kategorie. Oto one:

- **Wartości** (typów bezpośrednich) to wzorce bitów używane do reprezentowania typów podstawowych, takich jak liczby całkowite i znaki, a także bardziej skomplikowanych danych w postaci struktur. Każdy typ bezpośredni odpowiada odrębному oznaczeniu typu, które nie jest przechowywane w samych bitach. Odrębne oznaczenie typu jest powiązane z definicją typu, w której opisane jest znaczenie każdego bitu w wartości, a także operacje obsługiwane dla tej wartości.
- **Obiekty** (typów referencyjnych) zawierają w sobie oznaczenie typu. Obiekty mają tożsamość, która sprawia, że każdy obiekt jest unikatowy. Ponadto w obiektach znajdują się miejsca na wartości innych typów (bezpośrednich lub referencyjnych). Zmiana zawartości takiego miejsca nie wpływa na tożsamość obiektu (w przypadku typów bezpośrednich jest inaczej).

Te dwie kategorie typów są bezpośrednio powiązane ze składnią języka C#, która umożliwia deklarowanie poszczególnych typów.

Common Language Specification

Ponieważ zapewniane przez CTS zalety wynikające z integracji języków zwykle przeważają nad kosztami implementacji CTS-u, większość języków źródłowych jest zgodna z CTS-em. Fragmentem specyfikacji CTS jest specyfikacja **CLS** (ang. *Common Language Specification*), w której nacisk położony jest na implementowanie bibliotek. Specyfikacja CLS jest przeznaczona dla autorów bibliotek i opisuje standardy pisania bibliotek dostępnych w większości języków źródłowych. Nie ma przy tym znaczenia, czy języki źródłowe korzystające z tych bibliotek są zgodne ze specyfikacją CTS. Nazwa *Common Language Specification* (czyli specyfikacja wspólnego języka) wzięła się z tego, że specyfikacja CLS ma umożliwiać tworzenie w językach zgodnych z CLI bibliotek współpracujących z innymi językami.

Na przykład choć w pełni zrozumiałe jest udostępnianie w języku obsługi liczb całkowitych bez znaku, w specyfikacji CLS taki typ nie jest opisany. Dlatego programiści implementujący bibliotekę klas nie powinni zewnętrznie udostępniać liczb całkowitych bez znaku, ponieważ utrudni to dostęp do biblioteki z poziomu języków źródłowych zgodnych z omawianą specyfikacją, które nie obsługują takich liczb. Dlatego każda biblioteka, która ma być dostępna w wielu językach, powinna być zgodna ze specyfikacją CLS. Zauważ, że w kontekście specyfikacji CLS nie są istotne wewnętrzne typy, nieudostępniane zewnętrznie podzespołom.

Zwróć też uwagę na to, że kompilator może zgłaszać ostrzeżenia w reakcji na utworzenie interfejsu API, który nie jest zgodny ze specyfikacją CLS. Aby otrzymywać takie ostrzeżenia, należy zastosować używany do podzespołów atrybut `System.CLSCompliant` i ustawić wartość jego parametru na `true`.

Metadane

Oprócz instrukcji związanych z wykonywaniem programu kod CIL obejmuje **metadane** dotyczące typów i plików z programu. Metadane obejmują następujące informacje:

- Opis każdego typu z programu lub biblioteki klas.
- Informacje z manifestu zawierające dane na temat samego programu oraz bibliotek, które są mu potrzebne.
- Umsiączone w kodzie niestandardowe atrybuty, zapewniające dodatkowe informacje na temat powiązanych z nimi elementów.

Metadane nie są tylko mało istotnym, opcjonalnym dodatkiem do kodu CIL. Są one jednym z podstawowych elementów implementacji standardu CLI. Zawierają informacje o wyglądzie i działaniu typu oraz określają, który podzespoł zawiera definicję danego typu. Pełnią istotną rolę w zapisywaniu danych z kompilatora i zapewniają dostęp do nich debuggerom oraz środowisku uruchomieniowemu w trakcie wykonywania programu. Te dane są dostępne nie tylko w kodzie CIL, ale też w trakcie wykonywania kodu maszynowego, dzięki czemu środowisko uruchomieniowe może przeprowadzać potrzebną kontrolę typów.

Metadane są mechanizmem umożliwiającym środowisku uruchomieniowemu wykonywanie kodu natywnego i zarządzanego. Ponadto metadane zwiększą stabilność kodu i procesu wykonywania go, ponieważ ułatwiają przechodzenie z jednej wersji biblioteki na drugą (dzięki zastąpieniu wiązania z etapu komplikacji wybieraniem implementacji na etapie ładowania kodu).

Wszystkie informacje nagłówkowe na temat biblioteki i wymaganych przez nią komponentów znajdują się we fragmencie metadanych nazywanym manifestem. Dlatego manifest umożliwia programistom ustalenie zależności modułu (w tym informacji o konkretnych wersjach wymaganych komponentów) i podpisów określających autora modułu. W czasie wykonywania kodu środowisko uruchomieniowe wykorzystuje manifest do określenia, które wymagane biblioteki należy wczytać, czy ktoś manipulował przy bibliotekach lub głównym programie, a także czy nie brakuje jakichś podzespołów.

Metadane obejmują też **niestandardowe atrybuty**, które mogą być powiązane z kodem. Atrybuty zapewniają dodatkowe metadane na temat instrukcji z kodu CIL i są dostępne w programie w czasie jego wykonywania.

Metadane są dostępne w trakcie wykonywania kodu za pomocą mechanizmu **refleksji**. Dzięki refleksji można znaleźć typ lub jego składową w czasie wykonywania programu, a następnie wywołać daną składową lub ustalić, czy dany element jest opatrzony określonym atrybutem. Te mechanizmy umożliwiają **późne wiązanie**, polegające na tym, że system dopiero w czasie wykonywania programu (a nie na etapie komplikacji) ustala, który kod należy uruchomić. Refleksję można zastosować także do generowania dokumentacji w wyniku iterowania po metadanych i kopiowania ich do systemu pomocy (zobacz rozdział 18.).

Architektura .NET Native i kompilacja AOT

Architektura .NET Native (obsługiwana przez .NET Core i nowe implementacje .NET Framework) pozwala tworzyć pliki wykonywalne dostosowane do platformy. Służy do tego proces kompilacji AOT (ang. *Ahead of Time*).

Architektura .NET Native umożliwia programistom pisanie kodu w C# i uzyskanie przy tym wydajności typowej dla kodu natywnego oraz szybkie uruchamianie aplikacji dzięki wyeliminowaniu kompilacji JIT. W trakcie kompilowania aplikacji w architekturze .NET Native biblioteka .NET FCL jest statycznie dołączana do aplikacji. Dołączane są też komponenty środowiska uruchomieniowego .NET Framework zoptymalizowane pod kątem statycznej wstępnej kompilacji. Te specjalnie zaprojektowane komponenty są zoptymalizowane na potrzeby architektury .NET Native i zapewniają wydajność wyższą niż standardowe środowisko uruchomieniowe platformy .NET. Etap kompilacji nie wpływa na aplikację. Możesz swobodnie korzystać z wszystkich mechanizmów i API platformy .NET, a także polegać na zarządzanej pamięci i porządkowaniu pamięci, ponieważ architektura .NET Native dodaje wszystkie potrzebne komponenty platformy .NET Framework do pliku wykonywalnego.

Podsumowanie

W tym rozdziale pojawiło się wiele nowych pojęć i akronimów, które są ważne, jeśli chcesz zrozumieć kontekst działania programów w języku C#. Mnogość trzyliterowych akronimów może przytłaczać. W tabeli 24.2 znajdziesz krótką listę nazw i akronimów związanych ze standardem CLI.

Tabela 24.2. Często używane akronimy związane z językiem C#

Akronim	Rozwinięcie	Opis
.NET	Brak	Opracowana przez Microsoft implementacja wszystkich komponentów standardu CLI. Obejmuje środowisko CLR, język CIL i różne języki (wszystkie one są zgodne ze specyfikacją CLS).
BCL	Base Class Library	Część specyfikacji CLI definiująca klasy bazowe do obsługi kolekcji, wątków i konsoli, a także inne klasy przydatne przy rozwijaniu prawie wszystkich programów.
C#	Brak	Język programowania. Niezależnie od standardu CLI istnieje specyfikacja C# Language Specification, także uznawana przez jednostki standaryzacyjne ECMA i ISO.
CIL (IL)	Common Intermediate Language	Język ze specyfikacji CLI definiujący instrukcje używane w kodzie wykonywanym w implementacjach tej specyfikacji. Inne nazwy to IL oraz Microsoft IL (MSIL); ta nazwa odróżnia omawiany język od innych języków pośrednich). Nazwa CIL wskazuje na to, że ten język jest używany nie tylko w rozwiązaniach Microsoftu, dlatego zaleca się stosowanie jej zamiast określenia MSIL, a nawet zamiast nazwy IL.
CLI	Common Language Infrastructure	Jest to specyfikacja definiująca język pośredni, klasy bazowe i operacje, umożliwiająca tworzenie środowisk uruchomieniowych i kompilatorów. Zgodnie ze specyfikacją języki źródłowe mają współdziałać ze sobą we wspólnym środowisku wykonawczym.
CLR	Common Language Runtime	Opracowana przez Microsoft implementacja środowiska uruchomieniowego zgodna ze specyfikacją CLI.
CLS	Common Language Specification	Fragment specyfikacji CLI definiujący podstawowy podzbiór funkcji, które języki źródłowe muszą obsługiwać, aby napisany w nich kod działał w środowiskach uruchomieniowych zaimplementowanych zgodnie ze specyfikacją CLI.
CTS	Common Type System	Standard powszechnie implementowany w językach zgodnych ze specyfikacją CLI. Definiuje strukturę i działanie typów udostępnianych przez język poza modułem. Określa też, jak typy można łączyć w nowe typy.
FCL	.NET Framework Class Library	Biblioteka klas platformy Microsoft .NET Framework. Obejmuje opracowaną przez Microsoft implementację biblioteki BCL, a także bogatą bibliotekę klas związanych z tworzeniem aplikacji sieciowych, komunikacją w środowisku rozproszonym, dostępem do baz danych, rozwijaniem aplikacji z bogatym interfejsem użytkownika itd.
VES (środowisko uruchomieniowe)	Virtual Execution System	Agent zarządzający wykonywaniem programów skompilowanych zgodnie ze standardem CLI.

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 



KOMPLEKSOWO SZKOLIMY NOWOCZESNY BIZNES



IT



BIZNES



PROJEKTY



PROCESY

NASZE SZKOLENIA SĄ PROWADZONE
ZGODNIE Z METODĄ

BLENDDED LEARNING

modelem kształcenia, który łączy tradycyjne szkolenie z dostępem do nowoczesnych narzędzi - videokursów, e-booków i audiobooków

T: 609 850 372 E: SZKOLENIA@HELION.PL

WWW.HELIONSZKOLENIA.PL