Poznaj najskrytsze zakamarki C#!

Jon Skeet

C# OD PODSZEWKI

WYDANIE 2.

Tytuł oryginału: C# in Depth, Second Edition

Tłumaczenie: Janusz Grabis
Projekt okładki: Studio Gravite / Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki

ISBN: 978-83-246-3921-2

Original edition copyright © 2011 by Manning Publications Co.
All rights reserved.

Polish edition copyright © 2012 by HELION SA.
All rights reserved.

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 the Publisher.

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

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

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

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: http://helion.pl (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/cshop2
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

• Kup książkę
• Poleć książkę
• Oceń książkę

• Księgarnia internetowa
• Lubię to! » Nasza społeczność

Spis treści

Przedmowa 17

Wstęp 19

Podziękowania 21

O książce 23

Część I Przygotowanie do wyprawy 31

1. Nieustająca metamorfoza C# 33

1.1. Na początek prosty typ danych 34
1.1.1. Typ Product w C# 1 35
1.1.2. Kolekcje o mocnym typie w C# 2 36
1.1.3. Właściwości implementowane automatycznie w C# 3 37
1.1.4. Argumenty nazwane w C# 4 38
1.2. Sortowanie i filtrowanie 39
1.2.1. Sortowanie produktów po nazwie 40
1.2.2. Wyszukiwanie elementów w kolekcjach 43
1.3. Obsługa braku danych 44
1.3.1. Reprezentacja nieznanej ceny 45
1.3.2. Parametry opcjonalne i wartości domyślne 46
1.4. Wprowadzenie do LINQ 46
1.4.1. Wyrażenia kwerendowe i zapytania wewnątrzprocesowe 47
1.4.2. Wykonywanie zapytań na XML-u 48
1.4.3. LINQ dla SQL-a 49

1.5. COM i typy dynamiczne ... **50**
1.5.1. Uproszczenie współpracy z COM ... 50
1.5.2. Współpraca z językiem dynamicznym ... 51
1.6. Analiza zawartości środowiska .NET ... **52**
1.6.1. Język C# ... 53
1.6.2. Środowisko wykonania ... 53
1.6.3. Biblioteki środowiska ... 54
1.7. Jak zmienić Twój kod w kod doskonały? ... **54**
1.7.1. Prezentacja pełnych programów w formie fragmentów kodu ... 55
1.7.2. Kod dydaktyczny nie jest kodem produkcyjnym ... 56
1.7.3. Twój nowy najlepszy przyjaciel — specyfikacja języka ... 56
1.8. Podsumowanie ... **57**

2. Rdzeń systemu — C# 1 59

2.1. Delegaty ... **60**
2.1.1. Przepis na prosty delegat ... 61
2.1.2. Łączenie i usuwanie delegatów ... 66
2.1.3. Dygresja na temat zdarzeń ... 67
2.1.4. Podsumowanie delegatów ... 68
2.2. Charakterystyka systemu typów ... **69**
2.2.1. Miejsce C# w świecie systemów typów ... 69
2.2.2. Kiedy system typów C# 1 jest niedostatecznie bogaty? ... 73
2.2.3. Podsumowanie charakterystyki systemu typów ... 75
2.3. Typy wartościowe i referencyjne ... **76**
2.3.1. Wartości i referencje w rzeczywistym świecie ... 76
2.3.2. Podstawy typów referencyjnych i wartościowych ... 77
2.3.3. Obalanie mitów ... 79
2.3.4. Opakowywanie i odpakowywanie ... 81
2.3.5. Podsumowanie typów wartościowych i referencyjnych ... 82
2.4. Więcej niż C# 1 — nowe cechy na solidnym fundamencie ... **83**
2.4.1. Cechy związane z delegatami ... 83
2.4.2. Cechy związane z systemem typów ... 85
2.4.3. Cechy związane z typami wartościowymi ... 87
2.5. Podsumowanie ... **88**

Część II C# 2 — rozwiązanie problemów C# 1 89

3. Parametryzowanie typów i metod 91

3.1. Czemu potrzebne są typy generyczne? ... **92**
3.2. Proste typy generyczne do codziennego użycia ... **94**
3.2.1. Nauka przez przykład — słownik typu generycznego ... 94
3.2.2. Typy generyczne i parametry typów ... 96
3.2.3. Metody generyczne i czytanie deklaracji generycznych ... 100
3.3. Wkraczamy głębiej ... **103**
3.3.1. Ograniczenia typów ... 104
3.3.2. Interfejs argumentów typu dla metod generycznych ... 109
3.3.3. Implementowanie typów generycznych ... 110
3.4. Zaawansowane elementy typów generycznych ... **116**
3.4.1. Pola i konstruktory statyczne ... 117
3.4.2. Jak kompilator JIT traktuje typy generyczne ... 119

3.4.3. Iteracja przy użyciu typów generycznych ... 121
3.4.4. Refleksja i typy generyczne ... 123
3.5. Ograniczenia typów generycznych C# i innych języków ... 128
3.5.1. Brak wariancji typów generycznych ... 128
3.5.2. Brak ograniczeń operatorów lub ograniczeń „numerycznych” ... 133
3.5.3. Brak generycznych właściwości, indekserów i innych elementów typu ... 135
3.5.4. Porównanie z C++ ... 136
3.5.5. Porównanie z typami generycznymi Javy ... 137
3.6. Podsumowanie ... 139

4. Wyrażanie „niczego” przy użyciu typów nullowalnych 141

4.1. Co robisz, kiedy zwyczajnie nie masz wartości? ... 142
4.1.1. Czemu zmienne typu wartościowego nie mogą zawierać null? ... 142
4.1.2. Metody reprezentacji wartości null w C# 1 ... 143
4.2. System.Nullable<T> i System.Nullable ... 145
4.2.1. Wprowadzenie Nullable<T> ... 146
4.2.2. Opakowywanie i odpakowywanie Nullable<T> ... 149
4.2.3. Równość instancji Nullable<T> ... 150
4.2.4. Wsparcie ze strony niegenerycznej klasy Nullable ... 151
4.3. Składnia C# 2 dla typów nullowalnych ... 152
4.3.1. Modyfikator ? ... 152
4.3.2. Przypisania i porównania z null ... 154
4.3.3. Konwersje i operatory nullowalne ... 156
4.3.4. Logika typów nullowalnych ... 159
4.3.5. Stosowanie operatora as z typami nullowalnymi ... 161
4.3.6. Nullowy operator koalescencyjny ... 161
4.4. Nowatorskie zastosowania typów nullowalnych ... 164
4.4.1. Testowanie operacji bez parametrów zwrotnych ... 165
4.4.2. Bezbolesne porównania przy użyciu nullowalnego operatora koalescencyjnego ... 167
4.5. Podsumowanie ... 169

5. Przyspieszone delegaty 171

5.1. Pożegnanie z dziwaczną składnią delegatów ... 173
5.2. Konwersja grupy metod ... 174
5.3. Kowariancja i kontrawariancja ... 176
5.3.1. Kontrawariancja parametrów delegatów ... 176
5.3.2. Kowariancja typu zwracanego delegata ... 178
5.3.3. Małe ryzyko niekompatybilności ... 179
5.4. Akcje delegatów tworzone w miejscu przy użyciu metod anonimowych ... 180
5.4.1. Rozpoczynamy prosto — operując na parametrach ... 181
5.4.2. Zwracanie wartości z metod anonimowych ... 183
5.4.3. Ignorowanie parametrów typu ... 185
5.5. Przechwytywanie zmiennych w metodach anonimowych ... 186
5.5.1. Definicja domknięcia i różnych typów zmiennych ... 187
5.5.2. Analiza zachowania zmiennych przechwyconych ... 188
5.5.3. Jaki cel mają zmienne przechwycone? ... 189
5.5.4. Przedłużone życie zmiennych przechwyconych ... 190
5.5.5. Instancje zmiennej lokalnej ... 192
5.5.6. Mieszanka zmiennych współdzielonych i odrębnych ... 194
5.5.7. Wskazówki odnośnie do zmiennych przechwyconych i podsumowanie ... 196
5.6. Podsumowanie ... 197

6. Implementowanie iteratorów w prosty sposób 199

6.1. C# 1 — udręka ręcznego pisania iteratorów 201
6.2. C# 2 — proste iteratory z wyrażeniami yield 204
6.2.1. Wprowadzenie do uproszczeń iteratorów i wyrażenia yield return 204
6.2.2. Wizualizacja toku wykonania iteratora 206
6.2.3. Zaawansowany tok wykonania iteratora 208
6.2.4. Dziwactwa w implementacji 211
6.3. Przykłady z życia 213
6.3.1. Iterowanie po datach w harmonogramie 213
6.3.2. Iterowanie po wierszach pliku 214
6.3.3. Leniwe filtrowanie elementów z użyciem uproszczenia iteratora i predykatu 217
6.4. Pseudosynchroniczny kod z użyciem biblioteki CCR 219
6.5. Podsumowanie 222

7. Pozostałe cechy C# 2 225

7.1. Typy częściowe 227
7.1.1. Tworzenie typu składającego się z kilku plików 227
7.1.2. Użycie typów częściowych 229
7.1.3. Metody częściowe — tylko w C# 3 231
7.2. Klasy statyczne 233
7.3. Niezależny poziom dostępu do getterów i setterów właściwości 235
7.4. Aliasy przestrzeni nazw 237
7.4.1. Kwalifikowanie aliasów przestrzeni nazw 238
7.4.2. Aliasy globalnej przestrzeni nazw 239
7.4.3. Aliasy zewnętrzne 240
7.5. Dyrektywy pragma 241
7.5.1. Pragmy ostrzeżeń 242
7.5.2. Pragmy sum kontrolnych 243
7.6. Bufory o stałym rozmiarze w kodzie niezarządzanym 243
7.7. Udostępnianie wybranych elementów innym modułom 245
7.7.1. Zaprzyjaźnione moduły w prostym przypadku 246
7.7.2. Do czego warto używać InternalsVisibleTo? 247
7.7.3. InternalsVisibleTo i moduły podpisane 247
7.8. Podsumowanie 248

Część III C# 3 — rewolucja w metodzie programowania 251

8. Redukcja nadmiarowości przez zmyślny kompilator 253

8.1. Właściwości implementowane automatycznie 255
8.2. Zmienne lokalne o typie niejawnym 257
8.2.1. Zastosowanie var do deklarowania zmiennych lokalnych 257
8.2.2. Ograniczenia w typach niejawnych 259
8.2.3. Zalety i wady typów niejawnych 260
8.2.4. Zalecenia 261
8.3. Uproszczona inicjalizacja 262
8.3.1. Definicja prostych typów 262
8.3.2. Ustawianie prostych właściwości 263
8.3.3. Ustawianie właściwości obiektów zagnieżdżonych 265
8.3.4. Inicjalizatory kolekcji 266
8.3.5. Zastosowanie inicjalizatorów 269

8.4. Tablice o typie niejawnym ... **270**
8.5. Typy anonimowe ... **272**
8.5.1. Pierwsze spotkania z gatunkiem anonimowym ... 272
8.5.2. Części składowe typów anonimowych ... 274
8.5.3. Inicjalizatory odwzorowujące ... 275
8.5.4. Jaki to ma sens? ... 276
8.6. Podsumowanie ... **277**

9. Wyrażenia lambda i drzewa wyrażeń 279

9.1. Wyrażenia lambda i delegaty ... **281**
9.1.1. Przygotowanie — wprowadzenie delegatów typu Func<...> ... 281
9.1.2. Pierwsze przekształcenie na wyrażenie lambda ... 282
9.1.3. Używanie pojedynczego wyrażenia jako ciała ... 283
9.1.4. Lista parametrów o typie niejawnym ... 284
9.1.5. Skrót dla pojedynczego parametru ... 284
9.2. Proste przykłady użycia List<T> i zdarzeń ... **286**
9.2.1. Filtrowanie, sortowanie i operacje na listach ... 286
9.2.2. Logowanie w metodach obsługi zdarzeń ... 288
9.3. Drzewa wyrażeń ... **289**
9.3.1. Budowanie drzew wyrażeń w sposób programistyczny ... 289
9.3.2. Kompilowanie drzew wyrażeń do postaci delegatów ... 291
9.3.3. Konwersja wyrażeń lambda C# na drzewa wyrażeń ... 292
9.3.4. Drzewa wyrażeń w sercu LINQ ... 296
9.3.5. Drzewa wyrażeń poza LINQ ... 297
9.4. Zmiany we wnioskowaniu typów i rozwiązywaniu przeciążeń ... **299**
9.4.1. Powód do zmiany — usprawnienie wywołań metod generycznych ... 300
9.4.2. Wnioskowanie typu zwracanego funkcji anonimowych ... 301
9.4.3. Dwufazowe wnioskowanie typu ... 302
9.4.4. Wybieranie odpowiedniego przeciążenia metody ... 306
9.4.5. Podsumowanie wnioskowania typów i rozwiązywania przeciążeń ... 308
9.5. Podsumowanie ... **308**

10. Metody rozszerzające 311

10.1. Życie przed metodami rozszerzającymi ... **312**
10.2. Składnia metod rozszerzających ... **315**
10.2.1. Deklarowanie metod rozszerzających ... 315
10.2.2. Wywoływanie metod rozszerzających ... 316
10.2.3. Wykrywanie metod rozszerzających ... 318
10.2.4. Wywołanie metody na pustej referencji ... 319
10.3. Metody rozszerzające w .NET 3.5 ... **321**
10.3.1. Pierwsze kroki z Enumerable ... 321
10.3.2. Filtrowanie z użyciem Where i spinania wywołań metod ... 323
10.3.3. Antrakt: czy metody Where nie widzieliśmy już wcześniej? ... 325
10.3.4. Projekcja przy użyciu metody Select i typów anonimowych ... 326
10.3.5. Sortowanie przy użyciu OrderBy ... 327
10.3.6. Przykłady logiki biznesowej z użyciem łańcuchowania ... 328
10.4. Wskazówki i uwagi odnośnie do użycia ... **330**
10.4.1. „Rozszerzanie świata” i wzbogacanie interfejsów ... 330
10.4.2. Płynne interfejsy ... 331
10.4.3. Rozważne użycie metod rozszerzających ... 332
10.5. Podsumowanie ... **334**

11. Wyrażenia kwerendowe i LINQ dla Obiektów 335

11.1. Wprowadzenie do LINQ ... 336
11.1.1. Podstawowe koncepcje w LINQ ... 336
11.1.2. Definiowanie przykładowego modelu danych ... 341
11.2. Prosty początek — wybieranie elementów ... 343
11.2.1. Rozpoczynanie od źródła i kończenie na selekcji ... 343
11.2.2. Translacja kompilatora jako podstawa wyrażeń kwerendowych ... 344
11.2.3. Zmienne zakresu i projekcje nietrywialne ... 347
11.2.4. Cast, OfType i zmienne zakresu o typie jawnym ... 349
11.3. Filtrowanie i porządkowanie sekwencji ... 351
11.3.1. Filtrowanie przy użyciu klauzuli where ... 351
11.3.2. Zdegenerowane wyrażenia kwerendowe ... 352
11.3.3. Porządkowanie przy użyciu klauzuli orderby ... 353
11.4. Klauzule let i identyfikatory przezroczyste ... 356
11.4.1. Wprowadzenie do wykonania pośredniego z użyciem let ... 356
11.4.2. Identyfikatory przezroczyste ... 357
11.5. Złączenia ... 359
11.5.1. Złączenia wewnętrzne korzystające z klauzul join ... 359
11.5.2. Złączenia grupowe z użyciem klauzul join … into ... 363
11.5.3. Złączenia krzyżowe i spłaszczanie sekwencji przy użyciu wielokrotnych klauzul from ... 366
11.6. Grupowania i kontynuacje ... 369
11.6.1. Grupowanie przy użyciu klauzuli group ... by ... 369
11.6.2. Kontynuacje zapytań ... 373
11.7. Wybór pomiędzy wyrażeniami kwerendowymi a notacją kropkową ... 375
11.7.1. Operacje wymagające notacji kropkowej ... 376
11.7.2. Wyrażenia kwerendowe, dla których prostszym rozwiązaniem może się okazać notacja kropkowa ... 377
11.7.3. Gdzie wyrażenia kwerendowe lśnią? ... 377
11.8. Podsumowanie ... 379

12. LINQ — nie tylko kolekcje 381

12.1. Odpytywanie bazy danych przez LINQ dla SQL-a ... 382
12.1.1. Zaczynamy od bazy danych i modelu ... 383
12.1.2. Zapytania wstępne ... 385
12.1.3. Zapytania wymagające złączeń ... 388
12.2. Translacje przy użyciu IQueryable i IQueryProvider ... 390
12.2.1. Wprowadzenie do IQueryable<T> i związanych z nim interfejsów ... 391
12.2.2. Prototyp — implementacja interfejsów wykonująca wpisy w dzienniku ... 393
12.2.3. Spajanie wszystkiego razem — metody rozszerzające typu Queryable ... 395
12.2.4. Udawany dostawca w działaniu ... 397
12.2.5. Podsumowanie IQueryable ... 398
12.3. Interfejsy zaprzyjaźnione z LINQ i LINQ dla XML-a ... 399
12.3.1. Rdzenne typy LINQ dla XML-a ... 399
12.3.2. Konstrukcja deklaratywna ... 401
12.3.3. Zapytania na pojedynczych węzłach ... 404
12.3.4. Operatory zapytań spłaszczonych ... 406
12.3.5. Praca w harmonii z LINQ ... 407

12.4. Zastąpienie LINQ dla Obiektów Równoległym LINQ ... 408
12.4.1. Kreślenie zbioru Mandelbrota przez pojedynczy wątek ... 409
12.4.2. Wprowadzenie ParallelEnumerable, ParallelQuery i AsParallel ... 410
12.4.3. Podkręcanie zapytań równoległych ... 411
12.5. Odwrócenie modelu zapytania przy użyciu LINQ dla Rx ... 413
12.5.1. IObservable<T> i IObserver<T> ... 414
12.5.2. Zaczynamy (ponownie) łagodnie ... 416
12.5.3. Odpytywanie obiektów obserwowalnych ... 417
12.5.4. Jaki to wszystko ma sens? ... 419
12.6. Rozszerzanie LINQ dla Obiektów ... 420
12.6.1. Wytyczne odnośnie do projektu i implementacji ... 421
12.6.2. Proste rozszerzenie — wybieranie losowego elementu ... 422
12.7. Podsumowanie ... 424

Część IV C# 4 — dobra współpraca z innymi interfejsami 427

13. Małe zmiany dla uproszczenia kodu 429

13.1. Parametry opcjonalne i argumenty nazwane ... 430
13.1.1. Parametry opcjonalne ... 430
13.1.2. Argumenty nazwane ... 437
13.1.3. Złożenie dwóch cech w całość ... 441
13.2. Usprawnienia we współpracy z COM ... 446
13.2.1. Horror automatyzacji Worda przed C# 4 ... 446
13.2.2. Zemsta parametrów opcjonalnych i argumentów nazwanych ... 447
13.2.3. Kiedy parametr ref nie jest parametrem ref? ... 448
13.2.4. Wywoływanie indekserów nazwanych ... 449
13.2.5. Łączenie głównych bibliotek COM-owych ... 451
13.3. Wariancja typów generycznych dla interfejsów i delegatów ... 453
13.3.1. Typy wariancji: kowariancja i kontrawariancja ... 454
13.3.2. Użycie wariancji w interfejsach ... 455
13.3.3. Zastosowanie wariancji w delegatach ... 458
13.3.4. Złożone sytuacje ... 459
13.3.5. Restrykcje i uwagi ... 461
13.4. Mikroskopijne zmiany w blokadach i zdarzeniach w formie pól ... 465
13.4.1. Solidne blokowanie ... 465
13.4.2. Zmiany w zdarzeniach w formie pól ... 467
13.5. Podsumowanie ... 467

14. Dynamiczne wiązanie w języku statycznym 469

14.1. Co? Kiedy? Dlaczego? Jak? ... 471
14.1.1. Czym są typy dynamiczne? ... 471
14.1.2. Kiedy typy dynamiczne są użyteczne i dlaczego? ... 472
14.1.3. W jaki sposób C# zapewnia typy dynamiczne? ... 474
14.2. Pięciominutowy przewodnik po typie dynamic ... 474
14.3. Przykłady użycia typów dynamicznych ... 477
14.3.1. COM w ogólności i Microsoft Office w szczególności ... 477
14.3.2. Języki dynamiczne, takie jak IronPython ... 479
14.3.3. Typy dynamiczne w kodzie całkowicie zarządzanym ... 484
14.4. Zaglądamy pod maskę ... 490
14.4.1. Wprowadzenie do DLR ... 491
14.4.2. Fundamentalne koncepcje DLR ... 493

14.4.3. Jak kompilator C# obsługuje słowo dynamic? ... 496
14.4.4. Kompilator C# staje się jeszcze sprytniejszy ... 500
14.4.5. Ograniczenia kodu dynamicznego ... 503
14.5. Implementacja zachowania dynamicznego ... 506
14.5.1. Użycie klasy ExpandoObject ... 506
14.5.2. Użycie klasy DynamicObject ... 511
14.5.3. Implementacja IDynamicMetaObjectProvider ... 518
14.6. Podsumowanie ... 522

15. Jaśniejsze wyrażanie kodu przy użyciu kontraktów kodu 523

15.1. Życie przed kontraktami kodu ... 525
15.2. Wprowadzenie do kontraktów kodu ... 527
15.2.1. Warunki wstępne ... 529
15.2.2. Warunki końcowe ... 530
15.2.3. Inwarianty ... 531
15.2.4. Asercje i założenia ... 533
15.2.5. Kontrakty legacyjne ... 534
15.3. Przepisywanie kodu binarnego przez ccrewrite i ccrefgen ... 536
15.3.1. Proste przepisywanie ... 536
15.3.2. Dziedziczenie kontraktów ... 538
15.3.3. Referencyjne moduły kontraktów ... 541
15.3.4. Zachowanie w przypadku porażki ... 543
15.4. Sprawdzanie statyczne ... 545
15.4.1. Wprowadzenie do analizy statycznej ... 545
15.4.2. Zobowiązania niejawne ... 548
15.4.3. Selektywne włączanie opcji ... 551
15.5. Dokumentowanie kodu przy użyciu ccdocgen ... 554
15.6. Kontrakty w praktyce ... 556
15.6.1. Filozofia — czym jest kontrakt? ... 557
15.6.2. Jak mam zacząć? ... 558
15.6.3. Opcje, wszędzie opcje ... 559
15.7. Podsumowanie ... 562

16. Dokąd teraz? 565

16.1. C# — połączenie tradycji z nowoczesnością ... 566
16.2. Spotkanie .NET z informatyką ... 567
16.3. Świat informatyki ... 568
16.4. Pożegnanie ... 569

Dodatki 571

A Standardowe operatory kwerendowe LINQ 573

A.1. Agregacja ... 574
A.2. Konkatenacja ... 575
A.3. Konwersja ... 575
A.4. Operatory jednoelementowe ... 577
A.5. Równość ... 578
A.6. Generacja ... 579
A.7. Grupowanie ... 580
A.8. Złączenia ... 580
A.9. Partycjonowanie ... 582

A.10. Projekcja 582
A.11. Kwantyfikatory 583
A.12. Filtrowanie 584
A.13. Operatory bazujące na zbiorach 584
A.14. Sortowanie 585

B Kolekcje generyczne w .NET 587

B.1. Interfejsy 588
B.2. Listy 590
B.2.1. List<T> 590
B.2.2. Tablice 591
B.2.3. LinkedList<T> 592
B.2.4. Collection<T>, BindingList<T>, ObservableCollection<T> i KeyedCollection<TKey, TItem> 593
B.2.5. ReadOnlyCollection<T> i ReadOnlyObservableCollection<T> 594
B.3. Słowniki 594
B.3.1. Dictionary<TKey, TValue> 594
B.3.2. SortedList<TKey, TValue> i SortedDictionary<TKey, TValue> 595
B.4. Zbiory 596
B.4.1. HashSet<T> 597
B.4.2. SortedSet<T> (.NET 4) 597
B.5. Queue<T> i Stack<T> 598
B.5.1. Queue<T> 598
B.5.2. Stack<T> 598
B.6. Kolekcje konkurencyjne (.NET 4) 598
B.6.1. IProducerConsumerCollection<T> i BlockingCollection<T> 599
B.6.2. ConcurrentBag<T>, ConcurrentQueue<T>, ConcurrentStack<T> 600
B.6.3. ConcurrentDictionary<TKey, TValue> 600
B.7. Podsumowanie 600

C Podsumowanie wersji środowisk .NET 603

C.1. Główne wersje dystrybucyjne środowiska typu desktop 604
C.2. Cechy języka C# 605
C.2.1. C# 2.0 605
C.2.2. C# 3 605
C.2.3. C# 4.0 606
C.3. Biblioteki środowiska 606
C.3.1. .NET 2.0 606
C.3.2. .NET 3.0 606
C.3.3. .NET 3.5 607
C.3.4. .NET 4 607
C.4. Cechy środowiska uruchomieniowego (CLR) 608
C.4.1. CLR 2.0 608
C.4.2. CLR 4.0 609
C.5. Inne rodzaje środowiska uruchomieniowego 609
C.5.1. Compact Framework 609
C.5.2. Silverlight 610
C.5.3. Micro Framework 611
C.6. Podsumowanie 611

Skorowidz 613

3 Parametryzowanie typów i metod

Ten rozdział omawia:

- typy i metody generyczne,
- interfejs typów dla metod generycznych,
- ograniczenia typów,
- refleksję i typy generyczne,
- zachowanie środowiska CLR,
- ograniczenia typów generycznych,
- porównanie z innymi językami.

Oto prawdziwa historia[1]: pewnego dnia ja i moja żona robiliśmy cotygodniowe zakupy. Tuż przed naszym wyjazdem usłyszałem pytanie, czy mam listę. Potwierdziłem, że *mam* listę, i pojechaliśmy. Dopiero kiedy byliśmy już w sklepie, na jaw wyszła nasza pomyłka. Moja żona pytała o listę *zakupów*, podczas gdy ja wziąłem ze sobą listę zmyślnych cech C# 2. Sprzedawca był trochę zdziwiony, kiedy zapytaliśmy go, czy nie ma przypadkiem do sprzedania trochę metod anonimowych.

Gdybyśmy tylko mogli wyrazić się trochę jaśniej! Gdyby tylko moja żona mogła w jakiś sposób przekazać mi, że chce, abym wziął ze sobą listę rzeczy, które chcemy kupić! Gdybyśmy tylko mieli typy generyczne…

Dla większości programistów typy generyczne stanowią najważniejszą nową cechę C# 2. Poprawiają wydajność, sprawiają, że kod jest bardziej ekspresyjny, oraz przenoszą sporą dawkę zabezpieczeń z czasu wykonania do czasu kompilacji. W największym skrócie typy generyczne pozwalają na *parametryzowanie* typów i metod. Tak jak zwyczajne wywołania metod posiadają parametry wyrażające *wartości* przekazywane do środka, tak typy generyczne posiadają parametry określające, jakich *typów* użyć do ich konstrukcji. Brzmi to trochę mgliście — jeśli do tej pory nie miałeś do czynienia z typami generycznymi, być może będziesz się musiał dłużej zatrzymać przy tym temacie. Mogę się jednak założyć, że kiedy zrozumiesz podstawową ideę, pokochasz typy generyczne.

W tym rozdziale zajmiemy się użyciem typów i metod generycznych oferowanych przez środowisko lub biblioteki innych dostawców, a także omówimy, jak napisać własne. Po drodze dowiemy się, jak typy generyczne współdziałają z wywołaniami refleksyjnymi interfejsu programistycznego, oraz poznamy kilka detali odnośnie do sposobu obsługi typów generycznych przez środowisko wykonawcze. Na koniec rozdziału wspomnę o kilku najczęściej spotykanych ograniczeniach typów generycznych z możliwymi obejściami oraz porównam typy generyczne C# z podobnymi mechanizmami w innych językach programowania.

Na początku musimy jednak zrozumieć problemy, jakie stały się podstawą do opracowania typów generycznych.

3.1. Czemu potrzebne są typy generyczne?

Jeżeli posiadasz w swoich zasobach kod napisany przy użyciu C# 1, przyjrzyj mu się i policz rzutowania — szczególnie w kodzie, który intensywnie korzysta z kolekcji. Nie zapominaj, że pod niemal każdą pętlą `foreach` kryje się niejawne rzutowanie. Kiedy używasz typów zaprojektowanych do pracy z wieloma różnymi typami danych, nieuniknioną konsekwencją jest duża liczba rzutowań — cichego sposobu informowania kompilatora, aby się nie przejmował, założył, że wszystko jest w najlepszym porządku, i potraktował napotkane wyrażenie tak, jakby miało *ten* szczególny typ danych. Wykorzystanie dowolnego interfejsu programistycznego, który używa typu `object` do przekazywania parametrów wejściowych lub zwracania wyniku, prędzej czy później będzie

---

[1] Prawdę mówiąc, jest to „historia na potrzeby wprowadzenia do rozdziału” — niekoniecznie całkowicie prawdziwa.

wymagać rzutowania. Pewnym ułatwieniem jest wprowadzenie hierarchii klas bazującej na typie `object`. Podstawowy problem pozostanie ten sam — typ `object` jest zupełnie prosty, a więc żeby móc wykonać jakąkolwiek użyteczną pracę, trzeba w pierwszej kolejności dokonać jego rzutowania.

Czy rzutowanie nie jest „be”? Nie, nie należy rozumieć, że nigdy nie powinno się tego robić (rzutowanie przydaje się do pracy ze strukturami mutowalnymi i publicznymi polami klas), ale trzeba je traktować jako zło konieczne. Rzutowanie wskazuje, że jesteś zmuszony do przekazania kompilatorowi dodatkowej wiedzy i wskazania, aby ten zaufał Ci w czasie kompilacji i przeniósł sprawdzenie poprawności założeń do czasu wykonania.

Jeśli musisz dać do zrozumienia, że masz więcej wiedzy niż kompilator i dlatego chcesz dokonać rzutowania, takiej samej wiedzy będzie potrzebować osoba *czytająca* Twój kod. Możesz pozostawić komentarz w miejscu rzutowania, ale nie będzie to szczególnie użyteczne. Znacznie lepszym miejscem na taką informację jest deklaracja metody lub zmiennej. Jest to szczególnie ważne, jeśli dostarczasz typ lub metodę, z której inni użytkownicy będą korzystać *bez dostępu do Twojego kodu*. Typy generyczne umożliwiają dostawcom bibliotek zablokowanie wywołań z wnętrza bibliotek z użyciem nieprawidłowych parametrów. W C# 1 musieliśmy polegać na ręcznie napisanej dokumentacji, która (jak każda zduplikowana informacja) szybko staje się niekompletna i nieaktualna. Kiedy możemy taką informację zawrzeć bezpośrednio w kodzie jako część deklaracji metody lub typu, praca wszystkich stron staje się bardziej produktywna. Kompilator może dokonać więcej sprawdzeń, środowisko IDE może zaoferować dodatkowe informacje poprzez mechanizm IntelliSense (na przykład podczas przeglądania elementów listy łańcuchów może pokazać pola i metody typu `string`), użytkownicy metod będą pewniejsi co do poprawności swojego kodu pod względem przekazanych argumentów i zwróconych wartości, a czytelnicy Twojego kodu będą mogli lepiej zrozumieć, co miałeś na myśli, kiedy pisałeś dany fragment.

CZY TYPY GENERYCZNE ZREDUKUJĄ LICZBĘ TWOICH BŁĘDÓW?

Wszystkie opisy typów generycznych, jakie czytałem (włączając w to moje własne), podkreślają znaczenie przeniesienia sprawdzenia poprawności typów z czasu wykonania na czas kompilacji. Powierzę Ci mały sekret: nie przypominam sobie, abym kiedykolwiek musiał poprawiać błąd w wypuszczonym kodzie spowodowany brakiem sprawdzenia typu. Inaczej mówiąc, rzutowania, które umieszczaliśmy w C# 1, zawsze działały. Umieszczenie rzutowania w kodzie działa trochę jak znak ostrzegawczy, dzięki czemu zwracamy większą uwagę na poprawność typów. Typy generyczne być może nie zmniejszą radykalnie błędów związanych z bezpieczeństwem typów, ale wprowadzona przez nie czytelność kodu wpłynie pozytywnie na redukcję błędów związanych z użyciem kodu przez jego odbiorców. Prosty kod ma większą szansę być dobrze zrozumiany. O wiele łatwiej jest napisać dobry kod, który musi być odporny na nieprzemyślane zachowania programistów, kiedy odpowiednie gwarancje co do typu argumentów zapewnia sam system typów.

To, co do tej pory powiedzieliśmy o typach generycznych, przemawia wystarczająco na ich korzyść, ale są też jeszcze korzyści pod względem wydajnościowym. Po pierwsze, skoro kompilator może wykonać więcej sprawdzeń na etapie kompilacji, pozostaje

mniej rzeczy do weryfikacji w czasie wykonania. Po drugie, kompilator JIT może traktować typy wartościowe w sprytniejszy sposób, dzięki czemu w wielu sytuacjach udaje się uniknąć opakowywania i odpakowywania wartości. W pewnych przypadkach może to zdecydowanie wpłynąć zarówno na wydajność, jak i zużycie pamięci.

Wiele z korzyści, jakie dają typy generyczne, może do złudzenia przypominać te płynące z przewagi statycznego systemu typów nad systemem dynamicznym. Mowa tu o lepszej weryfikacji typów na poziomie kompilacji, większej liczbie informacji zawartych bezpośrednio w kodzie, lepszym wsparciu ze strony środowiska IDE i lepszej wydajności. Powód jest prosty: kiedy korzystasz z ogólnego interfejsu programistycznego (na przykład z klasy `ArrayList`), który nie jest w stanie wykryć różnic pomiędzy różnymi typami, znajdujesz się — jeśli chodzi o dostęp do tego interfejsu — w sytuacji dynamicznego systemu typów. Warto przy okazji wspomnieć, że sytuacja odwrotna — przewaga dynamicznego systemu typów nad systemem statycznym — ma miejsce w nielicznych przypadkach (korzyści płynące z zastosowania dynamicznych języków rzadko są bowiem związane z koniecznością dokonania wyboru pomiędzy interfejsem generycznym i niegenerycznym). Kiedy *możesz* skorzystać z typów generycznych, decyzja o ich użyciu jest oczywista.

To wszystko czeka na nas w C# 2 — teraz przyszła pora na rzeczywiste wykorzystanie typów generycznych.

3.2. Proste typy generyczne do codziennego użycia

Jeżeli chcesz wiedzieć *wszystko* o typach generycznych, musisz mieć świadomość, że mają one wiele skomplikowanych elementów. Specyfikacja języka C# przedstawia cały szereg szczegółowych informacji, aby opisać zachowanie w niemal każdej możliwej sytuacji. My nie musimy jednak znać wszystkich przypadków szczególnych, aby wykorzystać typy generyczne w sposób produktywny. (W rzeczywistości ta sama zasada obowiązuje dla dowolnego obszaru języka. Dla przykładu nie musisz znać wszystkich zasad rządzących przypisaniami, wystarczy, że będziesz umiał naprawić kod w przypadku błędu kompilacji).

W tym podrozdziale opiszemy wszystko, co powinieneś wiedzieć o typach generycznych, aby móc zastosować je w codziennej pracy — zarówno w sensie użytkownika interfejsu stworzonego przez innych programistów, jak i producenta własnych interfejsów. Jeżeli utkniesz na tym etapie, proponuję, abyś się skoncentrował na wiedzy potrzebnej do korzystania z typów i metod generycznych zawartych w środowisku i innych bibliotekach. Ta wiedza jest przydatna o wiele częściej niż umiejętność samodzielnego tworzenia typów i metod generycznych.

Zaczniemy od przyjrzenia się jednej z klas kolekcji wprowadzonych w .NET 2.0 — `Dictionary<TKey, TValue>`.

3.2.1. Nauka przez przykład — słownik typu generycznego

Użycie typów generycznych jest proste, o ile nie napotkasz przypadkiem jakiegoś ograniczenia i nie zaczniesz się zastanawiać, czemu Twoje rozwiązanie nie chce działać. Bez jakiejkolwiek wiedzy teoretycznej możesz z łatwością przewidzieć zachowanie

kodu, po prostu go analizując. Korzystając z metody prób i błędów, będziesz w stanie napisać własny działający kod. (Jednym z udogodnień typów generycznych jest to, że spora część weryfikacji kodu odbywa się w czasie kompilacji, jest zatem spora szansa, że Twój kod będzie działał, kiedy doprowadzisz do pomyślnej kompilacji — takie zachowanie jeszcze bardziej upraszcza eksperymentowanie z kodem). Oczywiście, celem tego rozdziału jest wyposażenie Cię w wiedzę, dzięki której *nie będziesz* musiał zgadywać — będziesz dokładnie wiedział, co się dzieje na każdym kroku.

Teraz przyjrzyjmy się fragmentowi kodu, który wygląda w miarę prosto, nawet jeśli składnia jest nieznajoma. Listing 3.1 używa typu `Dictionary<TKey, TValue>` (w przybliżeniu generycznego odpowiednika niegenerycznej klasy `Hashtable`) do zliczenia częstotliwości występowania słów w zadanym tekście.

Listing 3.1. Użycie Dictionary<TKey, TValue> do zliczenia słów w tekście

```
static Dictionary<string, int> CountWords(string text)
{
  Dictionary<string, int> frequencies;
  frequencies = new Dictionary<string, int>();

  string[] words = Regex.Split(text, @"\W+");

  foreach (string word in words)
  {
    if (frequencies.ContainsKey(word))
    {
      frequencies[word]++;
    }
    else
    {
      frequencies[word] = 1;
    }
  }
  return frequencies;
}
...
string text = @"Chodzi lisek koło drogi,
                Cichuteńko stawia nogi,
                Cichuteńko się zakrada,
                Nic nikomu nie powiada.";

Dictionary<string, int> frequencies = CountWords(text);
foreach (KeyValuePair<string, int> entry in frequencies)
{
  string word = entry.Key;
  int frequency = entry.Value;
  Console.WriteLine("{0}: {1}", word, frequency);
}
```

❶ Utworzenie nowego słownika „słowo – liczba wystąpień”

❷ Utworzenie nowego słownika „słowo – liczba wystąpień”

❸ Dodanie do słownika lub jego uaktualnienie

❹ Utworzenie komórki widoku tabeli

Metoda `CountWords` tworzy pusty słownik dla par łańcuch (`string`) – liczba (`int`) (❶). Jej zadaniem będzie zliczanie wystąpień każdego słowa w zadanym tekście. Następnie używamy wyrażenia regularnego (❷) do podzielenia tekstu na słowa. Wyrażenie nie jest szczególnie wyszukane — ze względu na kropkę na końcu zdania wśród słów pojawia się pusty łańcuch, a te same słowa pisane wielką i małą literą są traktowane jako różne.

Problemy te można łatwo rozwiązać — nie zrobiłem tego tylko dlatego, aby uzyskać jak najprostszy kod dla tego przykładu. Sprawdzamy, czy dane słowo jest już w naszym słowniku. Jeśli jest, zwiększamy licznik, w przeciwnym razie tworzymy wartość początkową licznika (❸). Zwróć uwagę, że kod dokonujący inkrementacji nie musi wykonywać rzutowania na typ `int`. Wiemy, że otrzymana wartość jest typu `int` już na etapie kompilacji. Krok wykonujący inkrementację tak naprawdę pobiera indekser słownika, zwiększa wartość, a następnie ustawia indekser z powrotem. Niektórzy programiści wolą wykonać ten krok w sposób jawny, używając wyrażenia `frequencies[word] = frequen` ➥`cies[word] + 1;`.

Ostatnia część listingu wygląda znajomo: enumeracja przez instancję typu `Hashtable` daje podobną (niegeneryczną) parę `DictionaryEntry` z właściwościami `Key` i `Value` dla każdego elementu kolekcji (❹). Różnica polega na tym, że w C# 1 klucz i wartość zostałyby zwrócone jako typ `object`, co zmusiłoby nas do rzutowania zarówno słowa, jak i częstotliwości. Dodatkowo częstotliwość (typ wartościowy) zostałaby opakowana. Trzeba przyznać, że nie *musimy* przypisywać słowa i częstotliwości do zmiennych lokalnych — moglibyśmy użyć pojedynczego wywołania `Console.WriteLine` i przekazać mu `entry.Key` i `entry.Value`. Sporne zmienne zostały przeze mnie użyte wyłącznie po to, by podkreślić brak konieczności rzutowania.

Teraz, kiedy widzieliśmy już przykład, przyjrzyjmy się, czym właściwie jest `Dictio` ➥`nary<TKey, TValue>`, do czego odnoszą się symbole `TKey` i `TValue` oraz jakie znaczenie mają ostre nawiasy wokół nich.

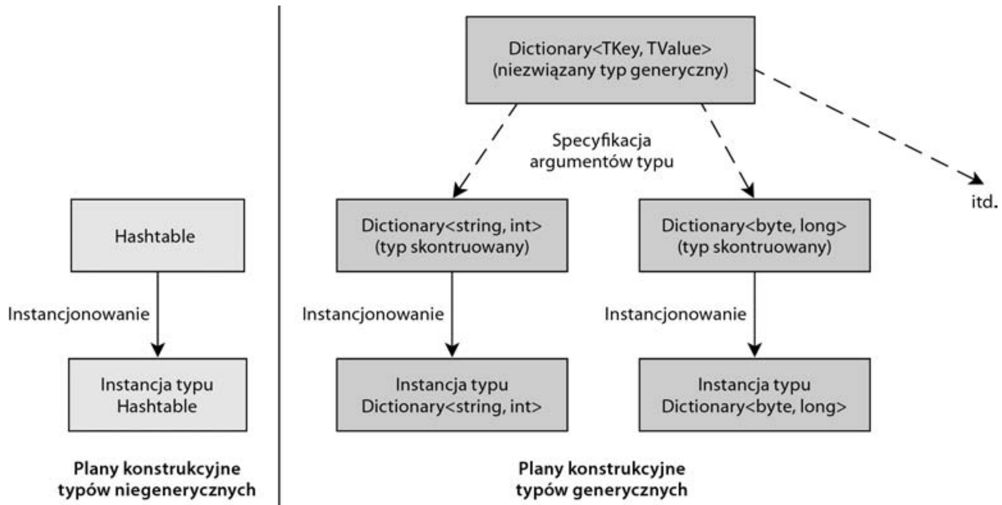
3.2.2. Typy generyczne i parametry typów

W C# występują dwie formy generycznych elementów języka: *typy generyczne* (włączając w to klasy, interfejsy, delegaty i struktury — nie ma enumeracji generycznych) oraz *metody generyczne*. Obie formy służą do wyrażania interfejsu programistycznego (w postaci pojedynczej metody generycznej lub całego typu generycznego) w taki sposób, iż w niektórych miejscach, gdzie spodziewasz się zobaczyć normalny typ, widzisz w zamian *parametr typu*.

Parametr typu jest jedynie rezerwacją miejsca na typ rzeczywisty. Parametry pojawiają się w ostrych nawiasach wewnątrz deklaracji typu generycznego i są oddzielone od siebie przecinkami. Zatem w deklaracji `Dictionary<TKey, TValue>` parametrami typu są `TKey` i `TValue`. Chcąc użyć typu lub metody generycznej, należy określić *rzeczywiste* typy — nazywane *argumentami typu* — jakimi chcemy się posługiwać. Dla przykładu na listingu 3.1 argumentami typu są `string` (dla `TKey`) i `int` (dla `TValue`).

ALERT ŻARGONOWY! Typy generyczne niosą ze sobą sporą dawkę szczegółowej terminologii. Umieściłem ją w książce w celach informacyjnych i dlatego, że czasem ułatwia ona rozmowę, w której poruszane są detale implementacyjne. Dla Ciebie terminologia ta może się okazać przydatna, jeśli będziesz miał zamiar zajrzeć do specyfikacji języka. Jest mało prawdopodobne, abyś musiał się nią posługiwać w codziennej pracy z typami generycznymi. Proszę zatem, abyś zniósł dzielnie tę dawkę terminologii. Wiele z używanych tutaj terminów znajdziesz w podrozdziale 4.4 specyfikacji C# 4 — proponuję, abyś zajrzał tam, jeśli chcesz znaleźć więcej informacji na ten temat.

Forma, w której żadnemu z parametrów typu nie przekazano argumentu typu, nazywana jest *niezwiązanym typem generycznym* (ang. *unbound generic type*). Kiedy argumenty typu są określone, mówimy o *typie skonstruowanym* (ang. *constructed type*). Niezwiązane typy generyczne można określić mianem planów konstrukcyjnych dla typów skonstruowanych, podobnie jak typy (generyczne lub nie) można uważać za plany konstrukcyjne obiektów. Jest to pewnego rodzaju dodatkowa warstwa abstrakcji. Rysunek 3.1 pokazuje ją w formie schematu graficznego.



Rysunek 3.1. Niezwiązane typy generyczne stanowią plany konstrukcyjne dla typów skonstruowanych, które z kolei służą jako plany konstrukcyjne dla rzeczywistych obiektów (na takiej samej zasadzie jak typy niegeneryczne)

Typy mogą być otwarte lub zamknięte. *Typ otwarty* nadal wymaga parametru typu (na przykład w postaci argumentu typu lub tablicy elementów typu), podczas gdy *typ zamknięty* jest przeciwieństwem typu otwartego — każdy element typu jest szczegółowo określony. Cały kod *wykonuje się* w kontekście zamkniętych typów skonstruowanych. Jedyny przypadek, kiedy masz szansę zobaczyć niezwiązany typ generyczny w kodzie C# (wyłączając deklarację), ma miejsce podczas użycia operatora `typeof`, z którym spotkamy się w sekcji 3.4.4.

Idea parametru typu „odbierającego" informację i argumentu typu „dostarczającego" informację — patrz przerywane linie na rysunku 3.1 — jest dokładnie taka sama jak w przypadku parametrów i argumentów metod, chociaż przy argumentach typu mamy do czynienia z typami, a nie arbitralnymi wartościami. Argument typu musi być znany w czasie kompilacji, ale w odpowiednim kontekście może nim być również parametr typu.

O typie zamkniętym możesz myśleć jak o interfejsie programowania typu otwartego, ale z parametrami typu zastąpionymi przez odpowiadające im argumenty typu[2].

[2] Taki model *nie zawsze* jest gwarantowany — istnieją przypadki szczególne, które nie będą działać po zastosowaniu tej prostej reguły — ale umożliwia proste pojmowanie typów generycznych, sprawdzające się w przeważającej większości przypadków.

Tabela 3.1 pokazuje deklaracje niektórych metod i właściwości publicznych otwartego typu `Dictionary<TKey, TValue>` oraz jego odpowiedniki w typie zamkniętym `Dictionary` ↳`<string, int>`.

Tabela 3.1. Przykłady sygnatur metod typu generycznego z symbolami rezerwującymi miejsce na typy i te same metody po podstawieniu argumentów typu

Sygnatura metody w typie generycznym	Sygnatura metody po podstawieniu argumentów typu
`void Add(TKey key, TValue value)`	`void Add(string key, int value)`
`TValue this[TKey key] {get; set; }`	`int this[string key] {get; set; }`
`bool ContainsValue(TValue value)`	`bool ContainsValue(int value)`
`bool ContainsKey(TKey key)`	`bool ContainsKey(string key)`

Zwracam uwagę, że żadna z metod w tabeli 3.1 nie jest w rzeczywistości metodą generyczną. Są to „zwyczajne” metody wewnątrz typu generycznego, które przypadkiem używają parametrów będących częścią deklaracji typu. Metodom generycznym przyjrzymy się w następnym podrozdziale.

Teraz, kiedy wiesz już, co znaczą symbole `TKey` oraz `TValue` i do czego służą ostre nawiasy, możemy sprawdzić, jak deklaracje z tabeli 3.1 wyglądałyby w deklaracji klasy. Oto, jak mógłby wyglądać kod dla `Dictionary<TKey, TValue>` (w poniższym przykładzie brakuje faktycznych implementacji metod i w rzeczywistości jest ich więcej):

```
namespace System.Collections.Generic
{
  public class Dictionary<TKey, TValue>             ← Deklaracja klasy generycznej
    : IEnumerable<KeyValuePair<TKey, TValue>>       ← Implementacja interfejsu generycznego
  {
    public Dictionary() { ... }                     ←

    public void Add(TKey key, TValue value) { ... } ← Deklaracja metody z użyciem parametrów typu

    public TValue this[TKey key]
    {
      get { ... }
      set { ... }
    }

    public bool ContainsValue(TValue value) { ... }

    public bool ContainsKey(TKey key) { ... }

    [...pozostałe metody...]
  }
}
```

Zauważ, jak `Dictionary<TKey, TValue>` implementuje interfejs generyczny `IEnumerable` ↳`<KeyValuePair<TKey, TValue>>` (i wiele innych interfejsów w rzeczywistym wykonaniu). Jakiekolwiek argumenty typu wyspecyfikujesz dla klasy, takie same zostaną zastosowane do interfejsu w miejscach, gdzie zostały użyte jednakowe parametry typu. A zatem w naszym przykładzie `Dictionary<string, int>` zaimplementuje `IEnumerable`

↪<KeyValuePair<string, int>>. Jest to swego rodzaju podwójny interfejs generyczny — interfejs IEnumerable<T> z argumentem typu w postaci struktury KeyValuePair<string, int>. Właśnie dzięki implementacji tego interfejsu przykład z listingu 3.1 był w stanie przejść przez wszystkie pary wartość-klucz w taki, a nie inny sposób.

Warto zauważyć, że konstruktor nie zawiera listy parametrów typu w nawiasach ostrych. Parametry typu należą raczej do *typu* niż do danego konstruktora i właśnie tam są deklarowane. Członkowie typu deklarują wyłącznie nowo wprowadzane parametry typu, mogą to jednak robić jedynie metody.

WYMOWA TYPÓW GENERYCZNYCH. Jeśli kiedykolwiek będziesz potrzebował opisać typ generyczny swojemu koledze lub koleżance, wymieniaj parametry lub argumenty typu w taki sposób, jak zrobiłbyś to dla zwykłej deklaracji. Na przykład dla List<T> możesz powiedzieć „lista typu T". W Visual Basicu taka wymowa jest sugerowana przez składnię języka: ten sam typ zostałby zapisany jako List(Of T). W przypadku kilku parametrów typu moim zdaniem dobrze jest oddzielić je słowem lub zdaniem sugerującym całościowe znaczenie typu. Zatem Dictionary<string, int> opisałbym jako „słownik łańcuchów *w* liczby" — podkreślając w ten sposób mapujący charakter typu.

Typy generyczne mogą być przeciążane w odniesieniu do liczby parametrów typu. Mógłbyś zatem zdefiniować MyType, MyType<T>, MyType<T,U>, MyType<T,U,V>, itd. — wszystkie w ramach tej samej przestrzeni nazw. Nazwy typów parametrów są w takiej sytuacji nieistotne, ważna jest jedynie ich liczba. Typy te, poza nazwą, nie są ze sobą związane — oznacza to między innymi, że nie istnieje między nimi domyślna konwersja. Ta sama zasada obowiązuje dla metod generycznych: dwie metody mogą mieć dokładnie taką samą sygnaturę z wyjątkiem liczby parametrów typu. Chociaż brzmi to trochę jak przepis na katastrofę, takie rozwiązanie może być przydatne, jeśli chcesz skorzystać z *wnioskowania typów generycznych*, w którym kompilator jest w stanie wywnioskować niektóre z argumentów typu za Ciebie. Wrócimy do tego zagadnienia w sekcji 3.3.2.

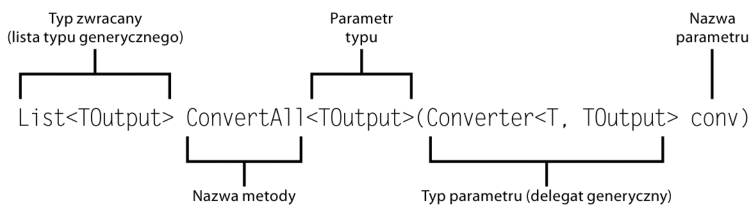
NAZEWNICTWO STOSOWANE DLA PARAMETRÓW TYPU. *Możesz* stworzyć typ, posługując się takimi parametrami jak T, U i V. Takie nazwy będą jednak mówić bardzo mało o swoim przeznaczeniu lub sposobie użytkowania. Dla porównania weźmy typ Dictionary<TKey, TValue>, w którym jasno widać, że TKey reprezentuje typ kluczy, a TValue — typ wartości. W przypadku pojedynczego parametru typu, którego przeznaczenie jest w miarę oczywiste, zgodnie z konwencją stosowana jest nazwa T (dobrym przykładem jest List<T>). Parametry typu występujące w liczbie mnogiej są zwykle nazywane zgodnie ze swoim przeznaczeniem i poprzedzane przedrostkiem T, aby wskazać, że chodzi o parametr typu. Mimo tych reguł czasem natrafisz na typ z kilkoma parametrami w postaci pojedynczych liter (przykładem jest SynchronizedKeyed↪Collection<T, K>). Staraj się unikać tworzenia podobnych konstrukcji.

Wiemy już, jakie zadania spełniają typy generyczne. Zajmijmy się teraz metodami generycznymi.

3.2.3. Metody generyczne i czytanie deklaracji generycznych

Zdążyłem już kilkakrotnie wspomnieć o metodach generycznych, ale do tej pory nie spotkaliśmy jeszcze żadnej z nich. W porównaniu do typów generycznych idea metod generycznych może się dla Ciebie okazać bardziej zawiła — wydają się one mniej naturalne dla naszego mózgu, chociaż podstawowa zasada działania pozostaje bez zmian. Jesteśmy przyzwyczajeni, że parametry wejściowe oraz wartości zwracane przez metody mają zawsze ściśle określony typ. Widzieliśmy również, jak typ generyczny używa parametrów typu do deklarowania swoich metod. Metody generyczne posuwają się jeszcze dalej — w ramach skonstruowanego typu mogą posiadać własne parametry typu. Nie przejmuj się, jeśli nie rozumiesz w tej chwili, o co dokładnie chodzi — myślę, że doznasz nagłego olśnienia po przeanalizowaniu kilku przykładów.

Dictionary<TKey, TValue> nie posiada metod generycznych, ale jego bliski krewny — List<T> — tak. Jak wiadomo, List<T> jest listą elementów określonego typu. Pamiętając, że T jest parametrem typu dla całej *klasy*, dokonajmy szczegółowej analizy deklaracji metody generycznej. Rysunek 3.2 pokazuje znaczenie poszczególnych części deklaracji metody ConvertAll[3].



Rysunek 3.2. Anatomia deklaracji metody generycznej

Patrząc na deklarację typu generycznego — bez znaczenia, czy jest to typ, czy też metoda — możesz mieć problem ze zrozumieniem jej znaczenia, szczególnie jeśli napotkasz typy generyczne wewnątrz typów generycznych, jak widzieliśmy to w przykładzie interfejsu implementowanego przez słownik. Najważniejsze to nie panikować, tylko przyjąć całość spokojnie, a następnie przeanalizować na przykładzie. Weź inny typ dla każdego parametru typu i odpowiednio je zastosuj.

W tym przypadku zacznijmy od zastąpienia parametru typu będącego właścicielem metody (część <T> typu List<T>). Będziemy się trzymać przykładu listy łańcuchów i we wszystkich miejscach deklaracji wstawimy string zamiast T:

```
List<TOutput> ConvertAll<TOutput>(Converter<string, TOutput> conv)
```

Jest trochę lepiej, ale dalej musimy się jeszcze uporać z TOutput. Widać, że jest to parametr typu metody (z góry przepraszam za mylącą terminologię), ponieważ występuje w ostrych nawiasach zaraz za jej nazwą. Spróbujmy użyć innego znanego nam typu — Guid — jako argumentu typu w miejsce TOutput. Ponownie zastępujemy wszystkie miejsca wystąpienia TOutput. Usuwamy część informującą o parametrze typu i od tego momentu możemy traktować metodę jako niegeneryczną:

```
List<Guid> ConvertAll(Converter<string, Guid> conv)
```

[3] Parametr converter został skrócony do conv, dzięki czemu deklaracja mieści się w jednej linii. Pozostała część ściśle odpowiada dokumentacji.

Teraz, kiedy wszystko jest wyrażone przy użyciu typów konkretnych, łatwiej będzie nam myśleć. Chociaż metoda jest faktycznie generyczna, przez moment będziemy ją traktować tak, jakby nią nie była. W ten sposób lepiej zrozumiemy jej znaczenie. Idąc przez elementy deklaracji od lewej strony, widzimy, że metoda:

- zwraca wynik typu `List<Guid>`,
- nazywa się `ConvertAll`,
- ma pojedynczy parametr `Converter<string, Guid>` o nazwie `conv`.

Teraz musimy się tylko dowiedzieć, czym jest `Converter<string, Guid>`, i będziemy mieć wszystko rozpracowane. Nie będzie chyba zaskoczeniem, jeśli dowiesz się, że `Converter<string, Guid>` jest skonstruowanym *delegatem typu generycznego* (jego wersją niezwiązaną jest `Converter<TInput, TOutput>`), który konwertuje łańcuch na GUID.

Mamy zatem metodę, która operuje na liście łańcuchów i używa konwertera do wyprodukowania listy GUID-ów. Skoro rozumiemy już sygnaturę metody, łatwiej będzie nam zrozumieć dokumentację, która potwierdza, że metoda tworzy nową listę typu `List<Guid>`, konwertuje każdy element z oryginalnej listy do typu docelowego i dodaje go do listy, którą zwraca na samym końcu. Myślenie o sygnaturze w konkretnych kategoriach daje nam przejrzysty model mentalny i ułatwia odgadnięcie faktycznego przeznaczenia metody. Chociaż technika ta może się wydawać trochę prymitywna, ja nadal uważam ją za bardzo użyteczną podczas analizy skomplikowanych metod. Prawdziwymi „potworami" pod tym względem są niektóre czteroparametrowe sygnatury metod należące do LINQ. Zastosowanie przedstawionej analizy pozwala poradzić sobie również z nimi.

Aby udowodnić, że cały ten wywód nie był tylko zwodzeniem, przyjrzyjmy się naszej metodzie w działaniu. Listing 3.2 przedstawia konwersję listy liczb całkowitych na listę liczb zmiennoprzecinkowych w taki sposób, że każdy element listy wynikowej jest pierwiastkiem kwadratowym odpowiadającej mu wartości z listy pierwotnej. Po wykonaniu konwersji wyświetlamy zawartość całej listy.

Listing 3.2. Metoda List<T>.ConvertAll<TOutput> w działaniu

```
static double TakeSquareRoot(int x)
{
  return Math.Sqrt(x);
}
...
List<int> integers = new List<int>();
integers.Add(1);
integers.Add(2);
integers.Add(3);
integers.Add(4);
Converter<int, double> converter = TakeSquareRoot;
List<double> doubles;
doubles = integers.ConvertAll<double>(converter);
foreach (double d in doubles)
{
  Console.WriteLine(d);
}
```

❶ **Utworzenie i wypełnienie listy liczb całkowitych**

❷ **Utworzenie instancji delegata**

❸ **Konwersja listy przy użyciu metody ConvertAll**

Utworzenie i wypełnienie listy jest banalnie proste (❶) — korzystamy z listy liczb całkowitych o mocnym typie. W punkcie (❷) używamy mechanizmu delegatowego (konwersji grupy metod) — wprowadzonego w C# 2 — który będziemy szczegółowo omawiać w podrozdziale 5.2. Chociaż nie lubię się posługiwać elementami języka przed ich pełnym opisem, tym razem byłem zmuszony. Ten wiersz kodu nie zmieściłby się tutaj, gdybyśmy posłużyli się składnią C# 1. W punkcie (❸) wywołujemy metodę generyczną, określając jej argument typu, podobnie jak robimy to dla typów generycznych. W tym miejscu moglibyśmy skorzystać z wnioskowania typu, ale nie chcę wprowadzać zbyt wielu cech jednocześnie. Wypisanie zawartości listy na końcu jest już zupełnie proste. Po uruchomieniu programu zgodnie z oczekiwaniami zobaczysz wartości `1`, `1.414`…, `1.732`… i `2`.

Ktoś mógłby zapytać, jaki to ma sens. Czy nie moglibyśmy zwyczajnie użyć pętli `foreach`, żeby przejść po wszystkich wartościach całkowitych i bezpośrednio wypisać wynik pierwiastka? Oczywiście, że tak. Przykład demonstruje jednak jeszcze inną rzecz — konwersję listy jednego typu na listę innego typu z użyciem pewnej logiki. Kod wykonujący taką operację „ręcznie” jest równie prosty, ale wersja wykonująca tę operację w pojedynczym wierszu jest czytelniejsza. Jest to typowa cecha metod generycznych — wykonują one w prostszy sposób operacje, które wcześniej mogłeś z powodzeniem wykonać dłuższą metodą. Przed wprowadzeniem metod generycznych operację podobną do `ConvertAll` można było wykonać na typie `ArrayList`, dokonując konwersji z typu `object` na `object`, ale efekt był o wiele mniej satysfakcjonujący. Dodatkowe usprawnienie niosą metody anonimowe (zobacz podrozdział 5.4) — przy ich użyciu moglibyśmy określić logikę konwersji w miejscu, unikając konieczności tworzenia w tym celu dodatkowej metody. Duże uproszczenie pod tym względem wprowadzają LINQ i wyrażenia lambda, o czym przekonamy się w części trzeciej.

Warto dodać, że metody generyczne mogą występować również w typach niegenerycznych. Listing 3.3 pokazuje metodę generyczną zadeklarowaną i użytą wewnątrz niegenerycznej klasy.

Listing 3.3. Implementacja metody generycznej wewnątrz niegenerycznego typu

```
static List<T> MakeList<T>(T first, T second)
{
  List<T> list = new List<T>();
  list.Add(first);
  list.Add(second);
  return list;
}
...
List<string> list = MakeList<string>("Wiersz 1", "Wiersz 2");
foreach (string x in list)
{
  Console.WriteLine(x);
}
```

Metoda generyczna `MakeList<T>` potrzebuje tylko jednego parametru typu (`T`). Jej działanie jest zupełnie proste i polega na utworzeniu listy z wartości przekazanych jako argumenty. Warto jednak zauważyć, że w jej wnętrzu możemy użyć `T` jako argumentu

typu do stworzenia obiektu typu `List<T>`. Implementację możesz traktować mniej więcej jako zastąpienie wszystkich wystąpień `T` przez `string`. Wywołując metodę, używamy tej samej składni, jaką widzieliśmy wcześniej podczas specyfikowania argumentów typu.

Do tej pory wszystko jasne? Na tym etapie powinieneś już móc tworzyć samodzielnie proste typy i metody generyczne. Przed nami jeszcze jeden stopień skomplikowania do pokonania, ale jeśli rozumiesz podstawową ideę typów generycznych, najtrudniejszy etap masz już za sobą. Nie przejmuj się zbytnio, jeśli nie wszystko jest dla Ciebie zupełnie klarowne, zwłaszcza jeśli chodzi o terminy „otwarty", „zamknięty", „niezwiązany" i „skonstruowany". Zanim przystąpisz do lektury dalszej części materiału, możesz wykonać kilka eksperymentów z typami generycznymi, aby zobaczyć ich działanie w praktyce. Jeżeli do tej pory nie używałeś kolekcji generycznych, polecam zajrzeć do dodatku B, który opisuje dostępne elementy języka. Kolekcje stanowią dobry punkt startowy do zabawy z typami generycznymi, a ponadto są powszechnie stosowane w niemal każdym nietrywialnym programie .NET.

Podczas eksperymentów przekonasz się, że trudno będzie zatrzymać implementację pomysłu w połowie. Kiedy zamienisz jakiś interfejs na generyczny, prawdopodobnie będziesz musiał poprawić pozostały kod, zamieniając go również na generyczny, lub dodać odpowiednie rzutowania, wymagane przez nowe — mocniejsze pod względem typu — wywołania. Alternatywnym rozwiązaniem jest użycie typów generycznych o silnym typie „pod maską", ale pozostawienie interfejsu o słabym typie na zewnątrz. Tak jak zawsze, wyczucie, kiedy należy użyć typów generycznych, przychodzi z czasem.

3.3. Wkraczamy głębiej

Proste sposoby wykorzystania typów generycznych, jakie widzieliśmy do tej pory, pozwolą na pracę z nimi w dłuższej perspektywie czasu. Są jednak jeszcze inne cechy, które mogą usprawnić naszą pracę. Zaczniemy od przeanalizowania *ograniczeń typów*, które dają możliwość większej kontroli nad argumentami typów (zwłaszcza kiedy tworzysz własne typy i metody generyczne). Zrozumienie ograniczeń typów ma również znaczenie z punktu widzenia znajomości opcji, jakie oferuje środowisko.

W następnym kroku zajmiemy się *wnioskowaniem typów* — poręcznym trikiem kompilatora, który pozwala na pominięcie jawnych argumentów typu podczas pracy z metodami generycznymi. Można się obyć bez tego mechanizmu, chociaż jego obecność przy odpowiednim zastosowaniu wpływa na uproszczenie kodu i sprawia, że jest on bardziej czytelny. W trzeciej części książki przekonamy się, że kompilator może teraz częściej wnioskować pewne informacje w oparciu o Twój kod przy jednoczesnym zachowaniu bezpieczeństwa i statyczności języka[4].

Ostatnia część tego podrozdziału będzie poświęcona pozyskiwaniu wartości domyślnej dla parametru typu, a także porównaniom dostępnym podczas pisania kodu generycznego. Rozważania zakończymy przykładem — w postaci użytecznej klasy — demonstrującym większość z omówionych cech.

---

[4] Z wyłączeniem C# 4, w którym pozwolono na *jawne* użycie typów dynamicznych.

Chociaż wchodzimy głębiej w szczegóły typów generycznych, nie ma przed nami nic *naprawdę* trudnego. Jest sporo rzeczy do zapamiętania, ale wszystkie cechy mają swój cel, o czym przekonasz się, kiedy dojrzejesz do ich wykorzystania. Zaczynajmy.

3.3.1. Ograniczenia typów

Do tej pory wszystkie parametry typów, z jakimi się zetknęliśmy, akceptowały dowolne typy bez wyjątku. Mogliśmy tworzyć różne typy, takie jak `List<int>` czy `Dictionary ↪<object, FileMode>` — cokolwiek przyszło nam do głowy. Nie ma w tym nic niebezpiecznego, dopóki mamy do czynienia z kolekcjami, które nie muszą współdziałać z przechowywaną zawartością. Jednak nie wszystkie kolekcje mają tyle szczęścia. Zdarzają się sytuacje, w których chcesz wywoływać metody na instancjach typu wskazanego przez parametr typu, tworzyć takie instancje lub ograniczyć zawartość listy wyłącznie do typów referencyjnych (lub tylko typów wartościowych). Innymi słowy, chcesz nakreślić reguły decydujące o poprawności argumentu typu dla swojego typu lub metody generycznej. W C# 2 do tego celu służą *ograniczenia*.

Dostępne są cztery rodzaje ograniczeń. Ich podstawowa składnia jest jednakowa. Ograniczenia pojawiają się na końcu deklaracji metody lub typu generycznego. Do ich wprowadzenia służy kontekstowe słowo kluczowe `where`. Później przekonamy się również, że ograniczenia mogą być łączone. Zaczniemy od przedstawienia kolejno każdego rodzaju ograniczenia.

Ograniczenie typu referencyjnego

Pierwszy rodzaj ograniczenia, wyrażony jako `T: class` i występujący zawsze jako pierwszy dla tego parametru typu, zapewnia, że użyty argument typu jest typem referencyjnym. Może to być klasa, interfejs, tablica, delegat lub inny parametr typu, o którym już wiadomo, że jest typem referencyjnym. Weźmy na przykład następującą deklarację:

```
struct RefSample<T> where T : class
```

Poprawne typy zamknięte dla tej deklaracji to między innymi:

- `RefSample<IDisposable>`,
- `RefSample<string>`,
- `RefSample<int[]>`.

Niepoprawne typy zamknięte to na przykład:

- `RefSample<Guid>`,
- `RefSample<int>`.

Celowo zadeklarowałem `RefSample` jako strukturę (i tym samym typ wartościowy), aby pokazać różnicę pomiędzy parametrem typu, który podlega ograniczeniu, a typem deklarowanym. `RefSample<string>` jest nadal typem wartościowym i podlega semantyce wartościowej, tyle że wszystkie wystąpienia parametru `T` zostały zastąpione typem `string`.

Kiedy parametr typu jest ograniczony w taki sposób, możesz porównywać referencje (łącznie z `null`), używając operatorów `==` i `!=`. Bądź jednak świadomy, że jeżeli nie ma innych ograniczeń, porównywane będą jedynie referencje, nawet jeśli dany typ prze-

ciąża te operatory (ma to miejsce w przypadku typu `string`). Do pojawienia się „gwarantowanych przez kompilator" przeciążeń operatorów `==` i `!=` możesz doprowadzić przez zastosowanie opisanego dalej ograniczenia konwersji typu, jednak sytuacja, w której będzie to potrzebne, zdarza się bardzo rzadko.

Ograniczenie typu wartościowego

To ograniczenie, wyrażone jako `T: struct`, wymusza użycie typu wartościowego jako argumentu typu, włączając w to enumeracje. Niedozwolone są również typy nullowalne, o których będziemy mówić w rozdziale czwartym. Przyjrzyjmy się przykładowej deklaracji:

```
class ValSample<T> where T : struct
```

Poprawnymi typami zamkniętymi dla tej deklaracji są:

- `ValSample<int>`,
- `ValSample<FileMode>`.

Niepoprawne typy zamknięte to między innymi:

- `ValSample<object>`,
- `ValSample<StringBuilder>`.

Tym razem `ValSample` jest typem referencyjnym, w którym `T` musi być typem wartościowym. Zauważ, że zarówno `System.Enum`, jak i `System.ValueType` są typami referencyjnymi i jako takie nie mogą być poprawnymi argumentami typu dla `ValSample`. Kiedy parametr typu jest ograniczony wyłącznie do typów wartościowych, zabronione są porównania przy użyciu operatorów `==` i `!=`.

Osobiście rzadko znajduję zastosowanie dla ograniczeń typów wartościowych i referencyjnych. W następnym rozdziale zobaczymy jednak, że mają one istotne znaczenie dla typów nullowalnych. Dwa pozostałe ograniczenia mogą się okazać bardziej użyteczne dla Ciebie, kiedy zaczniesz tworzyć własne typy generyczne.

Ograniczenie typu konstruktora

Trzeci typ ograniczenia wyrażany jest jako `T: new()` i musi występować zawsze jako *ostatnie* ograniczenie dla danego parametru typu. Jego zadaniem jest sprawdzenie, czy argument dla danego parametru typu posiada konstruktor bezparametrowy, który może zostać użyty do stworzenia instancji typu. Dotyczy to dowolnego typu wartościowego, dowolnej niestatycznej i nieabstrakcyjnej klasy bez jawnie zadeklarowanych konstruktorów, a także dowolnej nieabstrakcyjnej klasy z jawnym publicznym konstruktorem bezparametrowym.

C# KONTRA STANDARDY CLI. Istnieje pewna rozbieżność pomiędzy C# i standardem CLI pod względem typów wartościowych i konstruktorów. Specyfikacja C# mówi, że wszystkie typy wartościowe posiadają domyślny konstruktor bezparametrowy i że wywołania konstruktorów jawnych i bezparametrowych korzystają z tej samej składni — za skonstruowanie prawidłowego wywołania jest odpowiedzialny konstruktor. Specyfikacja CLI nie stawia takiego wymogu, za to dostarcza specjalną instrukcję do stworzenia wartości domyślnej

bez specyfikowania jakichkolwiek parametrów. Rozbieżność tę możesz zobaczyć w działaniu, kiedy użyjesz refleksji do znalezienia konstruktora typu wartościowego (nie znajdziesz konstruktora bezparametrowego).

Przyjrzyjmy się prostemu przykładowi, tym razem w formie metody. Aby zaprezentować użyteczność tego ograniczenia, dodam również ciało metody:

```
public T CreateInstance<T> where T : new()
{
  return new T();
}
```

Ta metoda zwraca nową instancję dowolnego typu, pod warunkiem że typ ten posiada konstruktor bezparametrowy. Dozwolone są zatem wywołania `CreateInstance<int>()` i `CreateInstance<object>`, ale nie `CreateInstance<string>()`, ponieważ typ `string` nie posiada konstruktora bezparametrowego.

Nie istnieje metoda pozwalająca na zmuszenie parametru typu do posiadania konstruktora o określonej sygnaturze — dla przykładu nie można nałożyć ograniczenia, iż konstruktor powinien posiadać jeden parametr typu `string`. Jest to trochę frustrujące, ale nie możemy nic na to poradzić. Problemowi temu przyjrzymy się bliżej, kiedy będziemy omawiać rozmaite ograniczenia typów generycznych .NET w podrozdziale 3.5.

Ograniczenie typu konstruktora może się okazać przydatne, kiedy zachodzi potrzeba skorzystania z wzorca fabryki klas, w której pewien obiekt wytwarza inne obiekty na żądanie. Często zachodzi wymóg, aby fabryka produkowała obiekty o określonym interfejsie, i to właśnie w tym miejscu do gry wchodzi nasz ostatni typ ograniczenia.

Ograniczenie typu konwersji

Ostatni (najbardziej skomplikowany) typ ograniczenia pozwala na wskazanie innego typu, do którego argument typu powinien dać się zrzutować poprzez identyczność, referencję lub operację opakowania. Możesz wskazać, aby określony argument typu był konwertowany na inny argument typu — jest to tak zwane *ograniczenie parametru typu*. Jego obecność utrudnia zrozumienie deklaracji, ale ograniczenie to może być bardzo przydatne w wielu sytuacjach. Tabela 3.2 pokazuje kilka deklaracji typu generycznego z ograniczeniami typu konwersji i towarzyszące im poprawne oraz niepoprawne przykłady typów skonstruowanych.

Trzeci wiersz, zawierający `T: IComparable<T>`, jest jednym z przykładów użycia typu generycznego jako ograniczenia. Dozwolone są również inne warianty, takie jak `T : List<U>` (gdzie `U` jest kolejnym parametrem typu) i `T: IList<string>`. Możesz wyspecyfikować wiele interfejsów, ale tylko jedną klasę. Poniższy przykład jest poprawny (i jednocześnie bardzo restrykcyjny):

```
class Sample<T> where T : Stream,
                          IEnumerable<string>,
                          IComparable<int>
```

Ten przykład jest niepoprawny:

```
class Sample<T> where T : Stream,
                          ArrayList,
                          IComparable<int>
```

Tabela 3.2. Przykłady ograniczeń typów konwersji

Deklaracja	Przykłady skonstruowanych typów
class Sample<T> where T : Stream	*Poprawny:* Sample<Stream> (rzutowanie przez identyczność) *Niepoprawny:* Sample<string>
struct Sample<T> where T : IDisposable	*Poprawny:* Sample<SqlConnection> (rzutowanie przez referencję) *Niepoprawny:* Sample<StringBuilder>
class Sample<T> where T : IComparable<T>	*Poprawny:* Sample<int> (opakowanie) *Niepoprawny:* Sample<FileInfo>
class Sample<T, U> where T : U	*Poprawny:* Sample<Stream, IDisposable> (rzutowanie przez referencję) *Niepoprawny:* Sample<string, IDisposable>

Żaden typ nie może dziedziczyć bezpośrednio po więcej niż jednej klasie, zatem takie ograniczenie byłoby niemożliwe do zrealizowania lub jego część byłaby nadmiarowa (na przykład wymuszenie, aby typ był potomkiem zarówno typu Stream, jak i MemoryStream). Istnieje jeszcze jedno obostrzenie: wskazany typ nie może być typem wartościowym, klasą zamkniętą (jaką jest na przykład string) lub jednym z następujących typów „specjalnych":

- System.Object,
- System.Enum,
- System.ValueType,
- System.Delegate.

SPOSÓB NA BRAK OGRANICZEŃ DOTYCZĄCYCH ENUMERACJI I DELEGATÓW. Wydaje się, że brak możliwości wskazania wymienionych wyżej typów w ograniczeniu typu wynika z jakiegoś ograniczenia w samym środowisku wykonawczym, ale tak nie jest. Być może dzieje się tak z powodów czysto historycznych (restrykcje zostały wprowadzone, kiedy dopiero pracowano nad typami generycznymi). Jeżeli jednak skonstruujesz odpowiedni kod bezpośrednio w języku IL, będzie on działał. Specyfikacja CLI wymienia nawet te typy jako przykłady i wyjaśnia, które deklaracje byłyby prawidłowe, a które nie. Jest to troszeczkę denerwujące, ponieważ z łatwością można sobie wyobrazić wiele metod generycznych, które dobrze byłoby ograniczyć wyłącznie do typów delegatowych lub enumeracji. Prowadzę projekt open source o nazwie Unconstrained Melody („Nieograniczona melodia", *http://mng.bz/s9Ca*), który — przy użyciu pewnych trików — buduje bibliotekę klas *posiadającą* ograniczenia na różnorodnych metodach użytkowych. Chociaż kompilator nie pozwoli Ci na zadeklarowanie tego typu ograniczeń, nie będzie zgłaszał żadnych zastrzeżeń, kiedy wywołasz odpowiednią metodę z biblioteki. Być może w przyszłych wersjach C# zakaz stosowania wymienionych typów zostanie zniesiony.

Ograniczenia typu konwersji są chyba najbardziej użyteczne, gdyż pozwalają wymusić użycie wyłącznie konkretnych typów jako instancji parametrów typu. Jednym ze szczególnie poręcznych przykładów tego ograniczenia jest T : IComparable<T>. Jego

zastosowanie daje pewność, że możesz w bezpośredni i znaczący sposób porównać dwie instancje typu T. Przykład takiego zachowania, a także innych form porównań znajduje się w sekcji 3.3.3.

Łączenie ograniczeń

Wspomniałem już, że istnieje możliwość istnienia wspólnie kilku ograniczeń. Przykład widzieliśmy podczas omawiania ograniczeń typu konwersji. To, czego jeszcze nie widzieliśmy, to łączenie razem różnych typów ograniczeń. Jasne jest, że żaden typ nie może być jednocześnie typem referencyjnym i wartościowym, zatem to połączenie odpada. *Każdy* typ wartościowy posiada konstruktor bezparametrowy, w związku z czym nie można użyć ograniczenia typu konstruktora, kiedy zostało użyte ograniczenie wymuszające typ wartościowy (nadal jednak możliwe jest stosowanie `new T()` wewnątrz metod, jeśli T zostało ograniczone do typów wartościowych). Jeśli posiadasz wiele ograniczeń typów konwersji i jednym z nich jest klasa, musi się ona pojawić przed interfejsami — każdy interfejs może wystąpić wyłącznie jeden raz. Każdy parametr typu posiada własne, niezależne ograniczenia, wprowadzane z użyciem słowa kluczowego `where`.

Spójrzmy na kilka poprawnych i niepoprawnych przykładów:

Poprawne:

```
class Sample<T> where T : class, IDisposable, new()
class Sample<T> where T : struct, IDisposable
class Sample<T, U> where T: class where U : struct, T
class Sample<T, U> where T : Stream where U : IDisposable
```

Niepoprawne:

```
class Sample<T> where T : class, struct
class Sample<T> where T : Stream, class
class Sample<T> where T : new(), Stream
class Sample<T> where T : IDisposable, Stream
class Sample<T> where T : XmlReader, IComparable, IComparable
class Sample<T,U> where T : struct where U : class, T
class Sample<T,U> where T : Stream, U : IDisposable
```

Ostatnie przykłady na obu listach demonstrują, jak łatwo można z wersji poprawnej stworzyć niepoprawną wersję ograniczenia. W takiej sytuacji błąd zgłoszony przez kompilator zupełnie nie pomaga. Warto w takim momencie przypomnieć sobie, że każda lista ograniczeń parametru wymaga własnego słowa wprowadzającego — `where`. Interesujący jest trzeci poprawny przykład — jeśli U jest typem wartościowym, jakim cudem może dziedziczyć po T, który jest typem referencyjnym? Odpowiedź: T mógłby być typem `object` lub interfejsem implementowanym przez U. Trzeba przyznać, że jest to dość paskudny przypadek.

TERMINOLOGIA UŻYWANA W SPECYFIKACJI. Specyfikacja dzieli ograniczenia na kategorie w trochę inny sposób. Wyróżnia ograniczenia *podstawowe*, *drugorzędne* i ograniczenia konstruktora. Pierwszy rodzaj dotyczy ograniczeń typu referencyjnego, wartościowego oraz ograniczeń konwersji przy użyciu klas. Drugi rodzaj wiąże się z ograniczeniami typu z użyciem interfejsów lub innego parametru typu. Nie uważam powyższej klasyfikacji za szczególnie użyteczną, chociaż ułatwia ona zdefiniowanie gramatyki ograniczeń: ograniczenie podstawowe

jest opcjonalne, ale może istnieć tylko jedno, ograniczeń drugorzędnych może być dowolnie wiele, a ograniczenie konstruktora jest opcjonalne (o ile nie występuje ograniczenie typu wartościowego, w przypadku którego jest ono zabronione).

Skoro posiadasz już całą wiedzę potrzebną do czytania deklaracji typów generycznych, przyjrzyjmy się wspomnianemu wcześniej interfejsowi argumentów typu. Na listingu 3.2 wskazaliśmy jawnie argumenty typu dla `List<T>.ConvertAll`. Podobnie zrobiliśmy na listingu 3.3 dla naszej własnej metody `MakeList`. Spróbujmy teraz poprosić kompilator o wypracowanie wartości tych argumentów, a tym samym — uproszczenie wywołań metod generycznych.

3.3.2. Interfejs argumentów typu dla metod generycznych

Określanie argumentów typu podczas wywoływania metod generycznych wydaje się często zupełnie nadmiarowe. W większości przypadków oczywiste jest, że argumenty typu powinny odpowiadać typom argumentów metody. Dla ułatwienia życia, poczynając od wersji drugiej C#, pozwolono kompilatorowi na samodzielność w ściśle określonych przypadkach, dzięki czemu możesz wywoływać metody bez jawnego wskazywania argumentów typu.

Zanim przejdziemy dalej, chcę zaznaczyć, że dotyczy to wyłącznie *metod* generycznych. Mechanizm nie działa dla *typów* generycznych. Skoro wyjaśniliśmy to sobie, przyjrzyjmy się odpowiednim wierszom na listingu 3.3 i zobaczmy, jak można uprościć nasz kod. Oto wiersze, które deklarują metodę, a następnie ją wywołują:

```
static List<T> MakeList<T>(T first, T second)
...
List<string> list = MakeList<string>("Wiersz 1", "Wiersz 2");
```

Przyjrzyj się argumentom; w obu przypadkach są to łańcuchy. Każdy z zadeklarowanych parametrów tej metody jest typu `T`. Nawet gdybyśmy nie mieli części `<string>` pomiędzy nazwą metody a listą jej argumentów, byłoby w miarę oczywiste, że zamierzamy wywołać ją z typem `string` jako argumentem dla parametru typu `T`. Kompilator pozwala na ominięcie tej części:

```
List<string> list = MakeList ("Wiersz 1", "Wiersz 2");
```

Czy tak nie wygląda lepiej? Na pewno jest krócej. Oczywiście, nie znaczy to wcale, że kod w takiej formie będzie zawsze czytelniejszy — w pewnych sytuacjach czytelnikowi kodu może być trudno dojść do tego, jakie argumenty typów miałeś na myśli, nawet jeśli kompilator jest w stanie zrobić to z łatwością. Proponuję, abyś każdy przypadek traktował indywidualnie. Ja pozwalam kompilatorowi na wywnioskowanie typu argumentów *w większości* możliwych przypadków.

Zauważ, że kompilator z całą pewnością wie, iż używamy typu `string`, ponieważ przypisanie do listy jest akceptowane, a mimo to argument typu dla listy pozostaje na miejscu (i musi pozostać). To przypisanie nie ma wpływu na proces wnioskowania parametru typu. Jeżeli kompilator wywnioskuje argument dla parametru typu w sposób błędny, najprawdopodobniej zostanie zgłoszony błąd kompilacji.

Jak to możliwe, że kompilator jest w stanie się pomylić? Załóżmy, że jako argumentu chcielibyśmy użyć typu `object`. Parametry naszej metody są nadal poprawne,

ale ponieważ oba parametry są łańcuchami, kompilator myśli, że chcemy użyć typu `string`. Wymuszenie typu na jednym z parametrów przez jawne rzutowanie sprawi, że wnioskowanie typu nie zadziała — jeden z argumentów metody będzie sugerował, że `T` to `object`, a drugi, że `string`. W takiej sytuacji kompilator *mógłby* uznać, że wybranie typu `object` jest satysfakcjonujące, a wybranie typu `string` nie. Niestety specyfikacja dopuszcza tylko ograniczoną liczbę kroków algorytmu wyboru. Ten mechanizm jest już całkiem złożony w C# 2, a C# 3 komplikuje sprawy jeszcze bardziej. Nie będę wnikał w szczegóły działania algorytmu w C# 2, ograniczę się jedynie do przedstawienia podstawowych kroków:

1. Dla każdego argumentu metody (mówimy o zwykłych argumentach, w nawiasach okrągłych) spróbuj, używając nieskomplikowanych technik, wywnioskować niektóre z argumentów typu metody generycznej.
2. Sprawdź, czy wszystkie wyniki z pierwszego kroku są spójne — inaczej mówiąc, jeśli jeden z argumentów zasugerował dany argument typu dla pewnego parametru typu, a inny argument zasugerował odmienny argument typu dla tego samego parametru typu, wnioskowanie zawodzi dla tego wywołania metody.
3. Sprawdź, czy wszystkie parametry typów potrzebne dla wywołania metody generycznej zostały wywnioskowane. Nie można określić samodzielnie części parametrów i pozostawić resztę do „odgadnięcia" kompilatorowi. Obowiązuje zasada: „wszystko albo nic".

Jest jedna rzecz, którą można zrobić, aby uniknąć nauki wszystkich reguł (nie polecam tego, o ile nie jesteś szczególnie zainteresowany detalami tego mechanizmu): spróbuj i zobacz, co się stanie. Jeśli myślisz, że kompilator *jest w stanie* wywnioskować *wszystkie* argumenty typu, wywołaj metodę, nie wskazując żadnego z nich. Kiedy się nie uda, wstaw jawnie wszystkie typy. Nie tracisz niczego poza chwilą na dodatkową kompilację i nie musisz mieć w głowie całego algorytmu postępowania.

W celu łatwiejszego użycia typów generycznych wnioskowanie typu może zostać połączone z ideą przeciążania nazw typów w oparciu o liczbę parametrów typu. Przykład takiego działania zobaczymy wkrótce, kiedy poskładamy wszystko w całość.

3.3.3. Implementowanie typów generycznych

Chociaż jest bardziej prawdopodobne, że spędzisz więcej czasu, używając metod i typów generycznych, niż pisząc je samodzielnie, jest kilka rzeczy, o których powinieneś wiedzieć, na wypadek gdyby przyszło Ci kiedyś stworzyć własną implementację. W większości przypadków możesz przyjąć, że `T` (lub dowolna inna nazwa, jaką upatrzyłeś dla swojego parametru) jest nazwą typu, i zacząć pisać kod w taki sposób, jakbyś nie miał do czynienia z typem generycznym. Powinieneś jednak mieć świadomość istnienia kilku dodatkowych czynników.

Wyrażenia wartości domyślnych

Pracując ze znanym sobie typem, znasz jego wartość domyślną — jest to na przykład wartość, jaką posiadałoby niezainicjalizowane pole tego typu. Kiedy nie wiesz, z jakim typem masz do czynienia, nie ma możliwości bezpośredniego wskazania wartości domyśl-

nej. Nie możesz użyć `null`, ponieważ nie ma gwarancji, że będzie to typ referencyjny, ani też zera, ponieważ może to być typ nienumeryczny. Chociaż potrzeba posiadania wartości domyślnej nie należy do częstych sytuacji, czasem jej obecność może się przydać. Dobrym przykładem jest `Dictionary<TKey, TValue>` — typ ten ma metodę `TryGetValue`, która zachowuje się nieco podobnie do obecnych w typach numerycznych metod `TryParse`, to znaczy używa parametru wyjściowego do zwrócenia wyniku działania i zwraca wartość boolowską, informującą, czy jej działanie zakończyło się sukcesem. Oznacza to, że metoda ta *musi* mieć pewną wartość typu `TValue`, którą wypełni parametr wyjściowy (pamiętasz zapewne, że parametrowi wyjściowemu trzeba przypisać wartość przed powrotem z metody).

> **WZORZEC TRYXXX.** Kilka wzorców projektowych środowiska .NET można łatwo zidentyfikować poprzez zaangażowane w nie metody. Przykładowo `BeginXXX` i `EndXXX` sugerują operację asynchroniczną. Wzorzec `TryXXX` jest jednym z kilku, których użycie zostało rozszerzone pomiędzy .NET 1.1 i 2.0. Został zaprojektowany dla sytuacji, które w zwykłych warunkach można byłoby traktować jako błędne (w sensie braku możliwości wykonania przez metodę jej podstawowego celu), ale w których porażka mogłaby nastąpić bez wskazywania wyraźnego błędu lub traktowania go jako wyjątku. Przykładowo użytkownicy często popełniają błędy podczas próby wpisania wartości numerycznej, zatem użyteczna byłaby *możliwość* przetłumaczenia pewnego fragmentu tekstu bez konieczności łapania i przetwarzania wyjątków. Takie rozwiązanie nie tylko poprawia wydajność w przypadku porażki, ale również oszczędza wyjątki dla poważniejszych błędów, kiedy coś jest nie tak z samym *systemem* (niezależnie od tego, jak szeroko chcesz traktować to pojęcie). Jest to wzorzec, który warto posiadać w swoim „arsenale" projektanta biblioteki.

C# 2 dostarcza *wyrażenie wartości domyślnej*, które zaspokaja tę potrzebę. Chociaż w specyfikacji wyrażenie to nie jest opisywane jako operator, możesz o nim myśleć podobnie jak o operatorze `typeof`, tyle że zwracającym inną wartość. Zostało to zobrazowane na przykładzie metody generycznej na kolejnym listingu (3.4). Znajdziemy w nim również użycie wnioskowania typu i ograniczenia typu konwersji.

Listing 3.4. Porównywanie wartości z wartością domyślną w sposób generyczny

```
static int CompareToDefault<T>(T value)
  where T : IComparable<T>
{
  return value.CompareTo(default(T));
}
...
Console.WriteLine(CompareToDefault("x"));
Console.WriteLine(CompareToDefault(10));
Console.WriteLine(CompareToDefault(0));
Console.WriteLine(CompareToDefault(-10));
Console.WriteLine(CompareToDefault(DateTime.MinValue));
```

Listing 3.4 pokazuje metodę generyczną użytą z trzema różnymi typami: `string`, `int` i `DateTime`. `CompareToDefault` narzuca konieczność użycia jej wyłącznie z typami implementującymi interfejs `IComparable<T>`, co pozwala nam wywołać metodę `CompareTo(T)` na wartości przekazanej do środka. Drugim elementem porównania jest wartość domyślna

typu. Dla referencyjnego typu string wartością domyślną jest null, natomiast dokumentacja metody CompareTo mówi, że dla typów referencyjnych „cokolwiek" jest większe niż null. Stąd wartością pierwszego wyrażenia jest 1. Kolejne trzy wiersze pokazują porównanie z domyślną wartością int, wynoszącą 0. Wynikiem ostatniej linii jest zero, co wskazuje, że DateTime.MinValue jest wartością domyślną dla typu DateTime.

Oczywiście metoda z listingu 3.4 nie zadziała, jeśli jako argument przekażesz null — wiersz wywołujący CompareTo rzuci wyjątek NullReferenceException. Nie przejmuj się tym — jak za chwilę pokażemy, istnieje alternatywa w postaci interfejsu IComparer<T>.

Porównania bezpośrednie

Chociaż listing 3.4 pokazał, w jaki sposób można dokonać porównania, nie zawsze jesteśmy skłonni do wymuszenia na naszych typach implementacji IComparable<T> lub siostrzanego interfejsu IEquatable<T>, który dostarcza metodę o mocnym typie — Equals(T) — uzupełniającą istniejącą w każdym typie metodę Equals(object). Bez dodatkowej informacji, do jakiej dostęp dają nam te interfejsy, nie jesteśmy w stanie zrobić wiele więcej ponad wywołanie Equals(object), co w przypadku typów wartościowych spowoduje opakowanie wartości, z którą chcemy porównać wartość w bieżącym kontekście. (W praktyce istnieje kilka typów, które mogą nam pomóc w pewnych sytuacjach — dojdziemy do nich za moment).

Kiedy parametr typu jest nieograniczony (nie zostały użyte żadne ograniczenia wobec niego), możesz stosować operatory == i !=, ale *wyłącznie* do porównywania wartości tego typu z null. Nie możesz porównać dwóch wartości typu T ze sobą nawzajem. Kiedy argumentem typu jest typ referencyjny, zostanie zastosowane zwykłe porównanie referencji. W przypadku kiedy argumentem podstawionym pod T jest nienullowalny typ wartościowy, wynikiem porównania z null będzie zawsze nierówność (w związku z czym porównanie może być usunięte przez kompilator JIT). Kiedy argumentem jest nullowalny typ wartościowy, porównanie będzie zachowywać się w tradycyjny sposób — nastąpi porównanie z wartością nullową danego typu[5]. (Nie przejmuj się, jeśli nie widzisz w tym jeszcze sensu — wszystko wyklaruje się, kiedy przeczytasz następny rozdział. Niestety niektóre cechy przeplatają się ze sobą tak mocno, że nie jestem w stanie opisać którejkolwiek z nich w sposób kompletny bez odwoływania się do innej).

Kiedy parametr typu jest ograniczony do typów wartościowych, porównania == i != są zabronione. W przypadku ograniczenia do typów referencyjnych rodzaj wykonywanej operacji porównania zależy *dokładnie* od tego, do czego ograniczony został parametr typu. Jeżeli jest to *tylko* typ referencyjny, wykonywane są proste porównania referencyjne. Jeżeli występuje dodatkowe ograniczenie w postaci konieczności wydziedziczenia z typu przeciążającego operatory == i !=, w porównaniach są używane te przeciążone operatory. Uważaj jednak, bo dodatkowe przeciążenia, które przypadkiem stały się dostępne poprzez argument typu wyspecyfikowany w kodzie wywołującym, *nie* są stosowane. Dokumentuje to kolejny listing (3.5) z prostym typem referencyjnym i argumentem w formie typu string.

---

[5] W chwili pisania tej książki kod generowany przez kompilator JIT dla porównań nieograniczonych wartości parametrów typu z null jest niesamowicie wolny dla wartościowych typów nullowalnych. Jeśli ograniczysz parametr T do typów nienullowalnych, a następnie porównasz wartość typu T? z null, czas wykonania tej operacji będzie znacznie krótszy. Jest to pole do dalszej optymalizacji JIT.

Listing 3.5. Porównania referencyjne z wykorzystaniem operatorów == i !=

```
static bool AreReferencesEqual<T>(T first, T second)
  where T : class
{
  return first == second;        ❶ Porównanie referencji
}
...
string name =  "Jan";
string intro1 = "Mam na imię " + name;
string intro2 = "Mam na imię " + name;
Console.WriteLine(intro1 == intro2);        ❷ Porównanie przy użyciu przeciążonego operatora porównania łańcuchów
Console.WriteLine(AreReferencesEqual(intro1, intro2));
```

Chociaż `string` przeciąża operator == (co demonstruje (❷), wyświetlając `True`), to przeciążony operator nie jest używany w porównaniu (❶). Mówiąc prościej, kiedy kompilowany jest typ `AreReferencesEqual<T>`, kompilator nie wie, że dostępne będą przeciążenia — zachowuje się trochę tak, jakby przekazane parametry były typu `object`.

To zachowanie nie jest wyłączną domeną operatorów — kiedy kompilator napotka w trakcie kompilacji typ generyczny, wyszukuje wszystkie przeciążenia metod dla typu niezwiązanego. Nie ma mowy o rozważaniu możliwych wywołań metod dla specyficznych przeciążeń w trakcie wykonania. Na przykład wyrażenie `Console.WriteLine(default(T));` zostanie zawsze rozwinięte do wywołania `Console.WriteLine(object value)` — nie będzie wywołania `WriteLine(string value)`, kiedy `T` przypadkiem będzie typu `string`. Jest to podejście zbliżone do zwykłego przeciążania, które jest rozwiązywane w trakcie kompilacji, a nie w trakcie wykonania, chociaż Czytelnicy mający pewne doświadczenie we wzorcach C++ mogą dać się jeszcze zaskoczyć[6].

W przypadku porównywania wartości dwiema *niezwykle* użytecznymi klasami są `EqualityComparer<T>` i `Comparer<T>` — obie w przestrzeni nazw `System.Collections.` ↪`Generic`. Implementują one, odpowiednio, `IEqualityComparer<T>` i `IComparer<T>`. Właściwość `Default` zwraca implementację, która na ogół wykonuje to, co trzeba, dla odpowiedniego typu.

> **GENERYCZNE INTERFEJSY PORÓWNUJĄCE.** Istnieją cztery podstawowe interfejsy do implementacji porównań. Dwa z nich — `IComparer<T>` i `ICompara` ↪`ble<T>` — służą do *porządkowania* (sprawdzają, czy pierwsza wartość jest mniejsza, równa, czy większa od drugiej), a pozostałe dwa — `IEqualityComparer` ↪`<T>` i `IEquatable<T>` — porównują, stosując pewne kryteria, elementy pod względem *równości* oraz wykonują ich skróty (w sposób zgodny z tą samą ideą równości obiektów).
>
> Szeregując je w inny sposób, `IComparer<T>` i `IEqualityComparer<T>` są implementowane przez typy zdolne do porównania dwóch różnych wartości, podczas gdy instancje `IComparable<T>` i `IEquatable<T>` są zdolne do porównania *samych siebie* z inną wartością.

[6] W rozdziale 14. zobaczymy, że typy dynamiczne dają możliwość rozwiązywania przeciążeń w czasie wykonania programu.

Dowiedz się więcej na temat tych interfejsów, czytając dokumentację, i rozważ użycie ich (i innych podobnych typów — jak `StringComparer`) podczas wykonywania operacji porównywania. W naszym następnym przykładzie użyjemy interfejsu `IEqualityCompa` ➥`rer<T>`.

Pełny przykład z porównywaniem — reprezentacja pary wartości

Na koniec naszych rozważań na temat implementacji typów i metod generycznych (można z całą śmiałością powiedzieć, że był to poziom dla średnio zaawansowanych) prezentujemy kompletny przykład. Implementuje on użyteczny typ generyczny — `Pair<T1, T2>`, który przechowuje dwie wartości, podobnie jak para klucz-wartość, ale bez żadnych oczekiwań co do związku pomiędzy nimi.

> **.NET 4 I KROTKI.** Wiele z funkcjonalności naszego przykładu można znaleźć w gotowych rozwiązaniach oferowanych przez .NET 4, w tym również struktury obsługujące różne ilości parametrów typu. Szukaj klas `Tuple<T1>`, `Tuple<T1, T2>` itd. w przestrzeni nazw `System`.

Oprócz implementacji właściwości dających dostęp do samych wartości nadpiszemy również metody `Equals` i `GetHashCode`, aby pozwolić instancjom naszego typu na prawidłowe zachowanie w sytuacji, kiedy zostaną one użyte jako klucze słownika. Kompletny przykład znajduje się na poniższym listingu (3.6).

Listing 3.6. Klasa generyczna reprezentująca parę wartości

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

public sealed class Pair<T1, T2> : IEquatable<Pair<T1, T2>>
{
  private static readonly IEqualityComparer<T1> FirstComparer =
     EqualityComparer<T1>.Default;
  private static readonly IEqualityComparer<T2> SecondComparer =
     EqualityComparer<T2>.Default;

  private readonly T1 first;
  private readonly T2 second;

  public Pair(T1 first, T2 second)
  {
    this.first = first;
    this.second = second;
  }

  public T1 First { get { return first; } }

  public T2 Second { get { return second; } }

  public bool Equals(Pair<T1, T2> other)
  {
    return other != null &&
       FirstComparer.Equals(this.First, other.First) &&
       SecondComparer.Equals(this.Second, other.Second);
  }
```

```
    public override bool Equals(object o)
    {
      return Equals(o as Pair<T1, T2>);
    }

    public override int GetHashCode()
    {
      return FirstComparer.GetHashCode(first) * 37 +
         SecondComparer.GetHashCode(second);
    }
  }
```

Listing 3.6 jest prosty. W polach odpowiedniego typu są przechowywane składniki klasy. Dostęp do nich dają proste właściwości tylko do odczytu. Implementujemy `IEquatable<Pair<T1, T2>>`, dzięki czemu udostępniamy na zewnątrz interfejs programistyczny o mocnym typie i unikamy niepotrzebnych sprawdzeń w czasie wykonywania programu. Kod sprawdzający równość i generujący skrót korzysta z domyślnych instancji porównujących dla naszych parametrów typu, dzięki czemu mamy automatycznie obsłużone wartości `null`, co upraszcza implementację[7].

Instancje interfejsów porównujących `T1` i `T2` zostały umieszczone w zmiennych statycznych głównie ze względu na ograniczenia w formatowaniu kodu narzucane przez rozmiar drukowanej strony. Będą one jednak dobrym punktem odniesienia do następnej sekcji.

Gdybyśmy chcieli wyposażyć naszą klasę w funkcję sortowania, moglibyśmy zaimplementować interfejs `IComparer<Pair<T1, T2>>` i założyć w implementacji, że pierwszy element występuje przed drugim. Tego rodzaju typ jest dobrym przykładem pokazującym, jakiej funkcjonalności *moglibyśmy* potrzebować, bez konieczności implementowania jej do momentu, kiedy faktycznie będzie potrzebna.

Mamy już naszą klasę `Pair`. A jak skonstruujemy jej instancję? Chwilowo możemy skorzystać z czegoś takiego:

```
Pair<int, string> pair = new Pair<int, string>(10, "wartość");
```

Nie wygląda to szczególnie ładnie. Dobrze byłoby skorzystać z wnioskowania typów, ale ten mechanizm działa tylko dla metod, a my żadnej nie posiadamy. Jeżeli umieścimy metodę generyczną wewnątrz typu generycznego, nadal będziemy zmuszeni zacząć od wskazania argumentów typu. Rozwiązaniem jest użycie niegenerycznej klasy pomocniczej z metodą generyczną w środku, jak pokazuje to poniższy listing (3.7).

Listing 3.7. Użycie typu niegenerycznego w połączeniu z metodą generyczną w celu skorzystania z wnioskowania typu

```
public static class Pair
{
  public static Pair<T1, T2> Of<T1, T2>(T1 first, T2 second)
  {
```

[7] Formuła użyta do obliczenia skrótu, oparta na dwóch wynikach częściowych, pochodzi z książki *Java. Efektywne Programowanie. Wydanie II* Joshuy Blocha (Helion, Gliwice 2009). Nie gwarantuje ona dobrego rozkładu skrótów, ale moim zdaniem jest znacznie lepsza niż użycie XOR-a bit po bicie. Więcej szczegółów na ten temat, a także inne użyteczne ciekawostki znajdziesz we wspomnianej książce.

```
        return new Pair<T1, T2>(first, second);
    }
}
```

Jeżeli czytasz tę książkę po raz pierwszy, zignoruj fakt, iż klasa została zadeklarowana jako statyczna. Dlaczego tak się stało, dowiemy się w rozdziale 7. Ważniejsza rzeczą jest to, że mamy niegeneryczną klasę z metodą generyczną. Dzięki temu możemy zamienić nasz poprzedni przykład na znacznie przyjemniejszy dla oka:

```
Pair<int, string> pair = Pair.Of(10, "wartość");
```

W C# 3 moglibyśmy nawet obyć się bez jawnego wskazywania typów zmiennej `pair`, ale nie uprzedzajmy faktów. Użycie tego typu niegenerycznych klas (lub klas *częściowo* generycznych, jeśli masz co najmniej dwa parametry typu i chcesz wywnioskować niektóre z nich, a pozostałe wskazać jawnie) jest użytecznym trikiem.

W tym momencie zakończyliśmy analizę cech „pośrednich". Zdaję sobie sprawę, że wszystko to wydaje się trochę skomplikowane na pierwszy rzut oka, ale zachęcam do wytrwałości. Ten dodatkowy stopień skomplikowania jest niczym w porównaniu do korzyściami, jakie osiągniemy. Z czasem użycie tych cech stanie się Twoją drugą naturą. Teraz nadszedł dobry moment, abyś przyjrzał się swojej własnej bibliotece klas i sprawdził, czy nie ma tam przypadkiem wzorców, które wciąż od nowa implementujesz wyłącznie ze względu na konieczność używania innych typów.

Każdy obszerny temat sprawia, że nieustannie pojawiają się nowe elementy do nauczenia. Kolejny podrozdział przeprowadzi Cię przez jedno z najważniejszych zagadnień zaawansowanej wiedzy o typach generycznych. Jeśli uważasz, że jest to nieco ponad Twoje siły, możesz pominąć tę część i przejść bezpośrednio do podrozdziału 3.5, gdzie omawiamy niektóre z ograniczeń[8] typów i metod generycznych. Materiał zawarty w następnym podrozdziale i tak warto zrozumieć. Jeśli jednak wszystko, co do tej pory czytałeś, jest dla Ciebie nowe, nie zaszkodzi go w tej chwili pominąć.

3.4. Zaawansowane elementy typów generycznych

Być może spodziewasz się, że w pozostałej części tego rozdziału zajmiemy się każdym możliwym aspektem typów i metod generycznych, o którym nie wspomnieliśmy do tej pory. Istnieje *tak wiele* zaułków i ciemnych korytarzy związanych z typami generycznymi, że jest to zwyczajnie niemożliwe. Ja w każdym razie nie chciałbym czytać o nich wszystkich, nie wspominając już o ich opisywaniu. Na nasze szczęście dobrzy ludzie z Microsoftu i ECMA zapisali wszystkie detale w specyfikacji języka, więc jeśli kiedykolwiek będziesz potrzebował zweryfikować jakąś niejasność, powinieneś zacząć od tych opracowań. Niestety, nie jestem w stanie wskazać jednej konkretnej części specyfikacji, która mówiłaby o typach generycznych. Ich obecność objawia się niemal wszędzie. Z drugiej strony warto się zastanowić, czy doprowadzenie swojego kodu do takiego stanu, że jego zrozumienie będzie wymagać analizy przypadków szczególnych w specyfikacji języka, ma jakikolwiek sens. Wskazane jest raczej dokonanie refaktory-

[8] W tym miejscu chodzi o ograniczenia typów i metod generycznych jako całości, a nie o opisywaną wcześniej składnię języka pozwalającą wpływać na postać konkretnych parametrów typu — *przyp. tłum.*

zacji kodu do prostszej postaci, w przeciwnym wypadku każdy kolejny inżynier przypisany do utrzymywania tego projektu będzie musiał rozpoczynać swoją pracę od lektury najmniej przyjemnych części specyfikacji.

Moim celem w tym podrozdziale jest zapoznanie Cię z wszystkimi detalami, które *prawdopodobnie* chciałbyś znać. Będę mówił więcej na temat środowiska wykonania niż na temat samej składni języka C# 2, chociaż wszystko to będzie miało oczywiście związek z programowaniem w C#. Zaczniemy od rozważenia statycznych elementów typów generycznych, włączając w to inicjalizację typu. Dalej zajmiemy się analizą tego, co właściwie dzieje się „pod maską”, chociaż będę się starał traktować w miarę lekko wszelkie szczegóły dotyczące istotnych efektów podjętych decyzji projektowych. Zobaczymy, co się dzieje, kiedy enumerujesz kolekcję generyczną, używając `foreach` w C# 2. Na koniec przyjrzymy się, jaki wpływ mają typy generyczne na mechanizm refleksji w .NET.

3.4.1. Pola i konstruktory statyczne

Pola instancji typu należą do instancji, a pola statyczne do typu, w którym zostały zadeklarowane. Jeśli zadeklarujesz statyczne pole `x` w klasie `SomeClass`, niezależnie od tego, ile instancji tej klasy stworzysz lub ile klas potomnych utworzysz w oparciu o `SomeClass`, będzie istnieć dokładnie jedno pole `SomeClass.x`[9]. Ten scenariusz obowiązuje w C# 1, a jak ma się sprawa w przypadku typów generycznych?

Odpowiedź brzmi: każdy z *zamkniętych* typów ma swój własny zestaw pól statycznych. Widzieliśmy to już na listingu 3.6, kiedy w polach statycznych umieściliśmy domyślne instancje interfejsów porównujących dla `T1` i `T2`, ale ponownie przyjrzyjmy się temu bliżej, używając innego przykładu. Listing 3.8 implementuje typ generyczny ze statycznym polem. Ustawiamy wartość tego pola dla różnych typów zamkniętych, a następnie wyświetlamy ich zawartość, aby pokazać, że faktycznie są od siebie odseparowane.

Listing 3.8. Dowód na to, że różne typy zamknięte posiadają niezależne pola statyczne

```
class TypeWithField<T>
{
  public static string field;

  public static void PrintField()
  {
    Console.WriteLine(field + ": " + typeof(T).Name);
  }
}
...
TypeWithField<int>.field = "Pierwszy";
TypeWithField<string>.field = "Drugi";
TypeWithField<DateTime>.field = "Trzeci";
```

[9] Ściśle rzecz biorąc, będzie istnieć jedna instancja na aplikację. Na potrzeby tej sekcji założymy, że mamy do czynienia z pojedynczą aplikacją. W przypadku wielu aplikacji wykonywanych jednocześnie takie same zasady obowiązują typy generyczne i niegeneryczne. Spod obowiązujących reguł w obu przypadkach są wyłączone zmienne deklarowane z użyciem `[ThreadStatic]`.

```
TypeWithField<int>.PrintField();
TypeWithField<string>.PrintField();
TypeWithField<DateTime>.PrintField();
```

Każdemu polu przypisujemy inną wartość, a następnie wyświetlamy ją wraz z nazwą argumentu użytego do stworzenia tego konkretnego typu zamkniętego. Oto wynik działania programu z listingu 3.8:

```
Pierwszy: Int32
Drugi: String
Trzeci: DateTime
```

Zatem podstawową zasadą jest: „jedno pole statyczne na każdy typ zamknięty". To samo dotyczy statycznych inicjalizatorów i konstruktorów. Istnieje jednak możliwość posiadania jednego typu generycznego wewnątrz innego, a także typów z wieloma parametrami typów. Wydaje się, że jest to zdecydowanie bardziej *skomplikowane*, ale tak naprawdę proces ten przebiega właśnie tak, jak prawdopodobnie sobie to wyobrażasz. Przykład takiego zachowania pokazuje poniższy listing (3.9).

Listing 3.9. Konstruktory statyczne z zagnieżdżonymi typami generycznymi

```
public class Outer<T>
{
  public class Inner<U, V>
  {
    static Inner()
    {
      Console.WriteLine("Outer<{0}>.Inner<{1},{2}>",
                typeof(T).Name,
                typeof(U).Name,
                typeof(V).Name);
    }

    public static void DummyMethod()
    {
    }
  }
}

...
Outer<int>.Inner<string, DateTime>.DummyMethod();
Outer<string>.Inner<int, int>.DummyMethod();
Outer<object>.Inner<string, object>.DummyMethod();
Outer<string>.Inner<string, object>.DummyMethod();
Outer<object>.Inner<object, string>.DummyMethod();
Outer<string>.Inner<int, int>.DummyMethod();
```

Pierwsze wywołanie `DummyMethod()` dla dowolnego typu spowoduje zainicjalizowanie typu. W tym momencie konstruktor statyczny wypisuje pewne informacje diagnostyczne. Każda unikalna lista argumentów typu liczy się jako inny typ zamknięty, zatem wynik działania kodu z listingu 3.9 wygląda następująco:

```
Outer<Int32>.Inner<String,DateTime>
Outer<String>.Inner<Int32,Int32>
Outer<Object>.Inner<String,Object>
```

```
Outer<String>.Inner<String,Object>
Outer<Object>.Inner<Object,String>
```

Tak jak w przypadku typów niegenerycznych, konstruktor statyczny dowolnego typu zamkniętego jest wywoływany tylko raz. Właśnie z tego powodu ostatni wiersz z listingu 3.9 nie tworzy szóstego wiersza wyniku — statyczny konstruktor dla `Outer<string>.Inner<int, int>` został wykonany wcześniej i wyprodukował drugi wiersz wyniku. Dla ścisłości: gdybyśmy mieli niegeneryczną klasę `PlainInner` w klasie `Outer`, nadal istniałby jeden możliwy typ `Outer<T>.PlainInner` na każdy zamknięty typ `Outer`, zatem `Outer<int>.PlainInner` różniłby się od `Outer<long>.PlainInner` i każdy z nich posiadałby swój zestaw pól statycznych.

Wiemy już, co wpływa na ukonstytuowanie się oddzielnego zestawu pól statycznych. Zastanówmy się, jakie mogą być efekty takiego działania z punktu widzenia wygenerowanego kodu natywnego. Dodam, że nie jest tak źle, jak może Ci się wydawać...

3.4.2. Jak kompilator JIT traktuje typy generyczne

Zadaniem kompilatora JIT jest przekonwertowanie kodu IL typu generycznego na kod natywny, który może być uruchomiony na danej platformie. Do pewnego stopnia nie powinno nas interesować, jak dokładnie kompilator wykonuje swoje zadanie — nie stanowiłoby dla nas wielkiej różnicy, pomijając pamięć i czas procesora, gdyby JIT przyjął najprostszą z możliwych strategii i wygenerował oddzielny kod natywny dla każdego możliwego typu, tak jakby każdy z nich nie miał nic wspólnego z dowolnym innym typem. Warto jednak sprawdzić, co się dokładnie dzieje, choćby po to, aby się przekonać, jak pomysłowi byli autorzy kompilatora JIT.

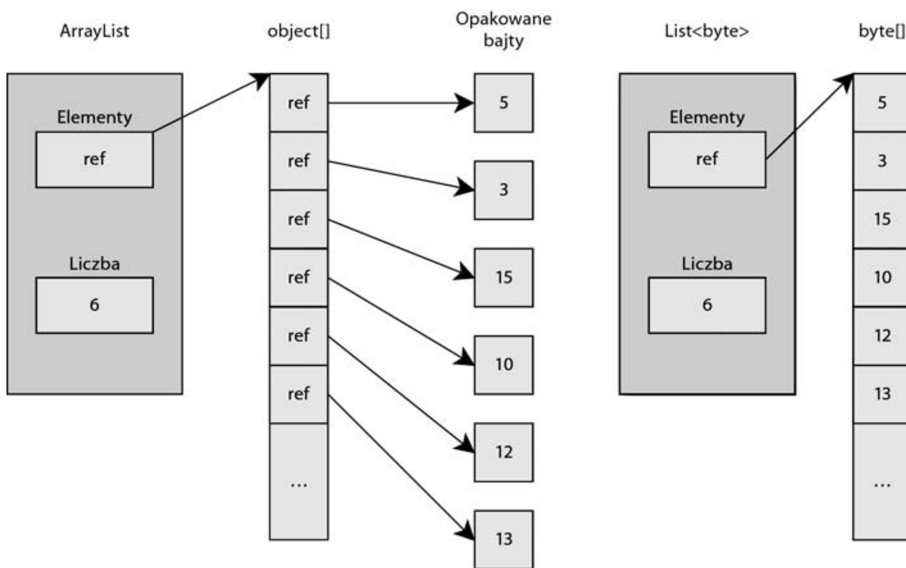
Zacznijmy od prostej sytuacji z pojedynczym parametrem typu — dla wygody niech będzie to `List<T>`. Kompilator JIT tworzy oddzielny kod natywny dla każdego zamkniętego typu, którego argumentem typu jest typ wartościowy — `int`, `long`, `Guid` itp. Kod natywny jest za to współdzielony dla wszystkich typów zamkniętych, które używają typów referencyjnych, takich jak `string`, `Stream` i `StringBuilder`. Może tak zrobić, ponieważ wszystkie referencje mają taki sam rozmiar (jest on zależny od platformy — CLR-32 lub CLR-64, ale w ramach tej samej platformy niezmienny). Lista referencji będzie miała zawsze taki sam rozmiar, niezależnie od tego, na co wskazują te referencje. Ilość pamięci potrzebna na stosie do zapamiętania referencji również będzie taka sama. Można zastosować takie same mechanizmy optymalizacji rejestrów, niezależnie od użytego typu. Tę listę moglibyśmy ciągnąć jeszcze bardzo długo.

Tak jak powiedzieliśmy wcześniej (w sekcji 3.4.1), każdy typ nadal posiada własne pola statyczne, ale korzysta ze wspólnego kodu. Oczywiście JIT wykazuje się pewnym „lenistwem" — kod dla `List<int>` jest generowany dopiero wtedy, kiedy jest faktycznie potrzebny, a po wygenerowaniu jest zapamiętywany dla przypadków użycia `List<int>` w przyszłości. Teoretycznie możliwe jest wykorzystanie tego samego kodu dla *niektórych* typów wartościowych. Kompilator JIT musiałby być w takich sytuacjach niezwykle ostrożny, nie tylko ze względu na rozmiar instancji w pamięci, ale również z powodu mechanizmu odśmiecania — musiałaby istnieć możliwość szybkiego identyfikowania obszarów struktury będących aktywnymi referencjami. Uwzględniając te warunki, typy wartościowe o tym samym rozmiarze i takim samym obrazie w pamięci (z punktu widzenia odśmiecania) *mogłyby* współdzielić kod. Na etapie powstawania niniejszej książki

możliwość ta miała tak niski priorytet, że nie została zaimplementowana, i prawdopodobnie sytuacja ta nieprędko się zmieni.

Tak duży poziom szczegółowości jest domeną akademicką, ale ma on również mały wpływ na wydajność w związku z większą ilością kodu, jaką musi kompilować JIT. Z drugiej strony *korzyści* wydajnościowe oferowane przez typy generyczne mogą być niewyobrażalne, a wszystko to właśnie dzięki możliwościom tworzenia kodu dla różnych typów przez ten kompilator. Weźmy dla przykładu `List<byte>`. W .NET 1.1 dodawanie pojedynczych bajtów do kolekcji typu `ArrayList` wymagało opakowania każdego z nich, a następnie zapamiętania referencji do opakowanej wartości. Użycie `List<byte>` nie nakłada takich wymogów — `List<T>` ma pole typu `T[]`, będące odpowiednikiem `object[]` w `ArrayList`. Dzięki temu tablica ma odpowiedni typ i zajmuje właściwą ilość miejsca w pamięci. Zatem `List<byte>` przechowuje swoje elementy w tablicy `byte[]` (co pod wieloma względami upodabnia ją do typu `MemoryStream`).

Rysunek 3.3 pokazuje instancje dwóch typów: `ArrayList` i `List<byte>`, z których każda zawiera sześć jednakowych wartości. Obie tablice pozwalają na dokładanie elementów, obie mają pewien bufor i w miarę możliwości mogą go powiększyć.



Rysunek 3.3. Przedstawienie w sposób graficzny, dlaczego List<T> zajmuje o wiele mniej miejsca niż ArrayList

W tym konkretnym przypadku różnica w wydajności jest niesamowita. Przyjrzyjmy się najpierw tablicy `ArrayList`, zakładając, że jesteśmy na platformie CLR-32[10]. Każdy z bajtów będzie wymagać 8-bajtowego narzutu związanego z przechowującym go obiektem plus 4 bajty (1 bajt zaokrąglony do granicy słowa) na dane. Do tego dochodzą wszystkie referencje, z których każda zajmuje 4 bajty. Za przechowanie pojedynczego bajta „płacimy" szesnastoma bajtami. Pozostaje jeszcze miejsce zajmowane przez referencje w buforze.

[10]Ta sama sytuacja na platformie CLR-64 zwiększa narzut pamięciowy.

Porównaj ten stan rzeczy z typem List<byte>. Każdy bajt w liście zajmuje pojedynczy bajt w tablicy elementów. Pozostaje pamięć zarezerwowana na potrzeby nowych elementów, ale zużywamy tylko po jednym bajcie na każdy taki element.

Zyskujemy nie tylko miejsce, ale również szybkość wykonania. Nie potrzebujemy czasu na przydzielenie miejsca „pudełku”, sprawdzanie typu związanego z operacjami opakowywania i odpakowywania czy zbieranie śmieci pozostałych po odpakowaniu wartości.

Nie musimy schodzić do poziomu środowiska wykonania, aby znaleźć rzeczy, które dzieją się w sposób niezauważalny dla nas. C# od zawsze ułatwiał życie przez skróty syntaktyczne. W kolejnej sekcji przyjrzymy się znanemu już przykładowi — iteracji przy użyciu foreach — ale tym razem z domieszką typów generycznych.

3.4.3. Iteracja przy użyciu typów generycznych

Jedną z najczęściej wykonywanych operacji na kolekcjach jest iterowanie po wszystkich jej elementach. Najprostszym sposobem wykonania tej operacji jest użycie pętli foreach. W pierwszej wersji C# wymagało to implementowania przez kolekcję interfejsu System.Collections.IEnumerable lub posiadania podobnej metody — GetEnumerator()— która zwracała typ z metodą MoveNext() i właściwością Current. Właściwość Current nie musiała być typu object i właśnie z tego powodu obowiązywały wszystkie powyższe reguły, które na pierwszy rzut oka wydają się dziwne. Jak widać, posiadając odpowiedni typ danych, nawet w C# 1 można było uniknąć operacji opakowywania i odpakowywania podczas iteracji.

W C# 2 zadanie zostało uproszczone przez wprowadzenie dodatkowych reguł dla foreach. Zezwolono na użycie interfejsu System.Collections.Generic.IEnumerable<T> razem z jego partnerem IEnumerator<T>. Są to odpowiedniki starszych interfejsów iteratora, a ich użycie jest preferowane w stosunku do niegenerycznych wersji. Oznacza to, że jeśli iterujesz po generycznej kolekcji typu wartościowego — niech będzie to List<int> — operacja opakowywania nie jest wykonywana. W przypadku starego interfejsu, nawet gdybyśmy uniknęli kosztu opakowywania podczas *zapisywania* elementów w liście, nadal musielibyśmy opakowywać przy dostępie do nich w pętli foreach!

Wszystkie te operacje są wykonywane „pod maską” — od Ciebie wymaga się jedynie użycia pętli foreach i odpowiedniego typu dla zmiennej iteracyjnej. To jednak nie koniec historii. W rzadkich przypadkach, kiedy będziesz musiał zaimplementować iterację po swoim własnym typie, przekonasz się, że IEnumerable<T> rozszerza interfejs IEnumerable, co oznacza, że będziesz musiał zaimplementować dwie różne metody:

```
IEnumerator<T> GetEnumerator();
IEnumerator GetEnumerator();
```

Czy widzisz problem? Metody różnią się jedynie typem zwracanym, a zasady przeciążania metod w C# nie pozwalają w normalnych warunkach na napisanie dwóch takich metod. Podobną sytuację spotkaliśmy w sekcji 2.2.2 i teraz możemy użyć podobnego obejścia. Jeżeli zaimplementujesz IEnumerable poprzez jawną implementację interfejsu, nie będziesz mógł w „normalny” sposób zaimplementować IEnumerable<T>. Na szczęście, ponieważ IEnumerator<T> rozszerza IEnumerator, możesz użyć tej samej wartości w obu metodach i zaimplementować wersję niegeneryczną przez wywołanie wersji

generycznej. Teraz jednak musisz zaimplementować `IEnumerator<T>` i szybko trafiasz na problem, tym razem z właściwością `Current`.

Poniższy listing (3.10) przedstawia kompletny przykład implementacji klasy enumerowalnej, która zawsze iteruje po wartościach typu całkowitego od 0 do 9.

Listing 3.10. Iterator generyczny dla liczb od 0 do 9

```
class CountingEnumerable : IEnumerable<int>
{
  public IEnumerator<int> GetEnumerator()      ❶ Niejawna implementacja IEnumerable<T>
  {
    return new CountingEnumerator();
  }

  IEnumerator IEnumerable.GetEnumerator()      ❷ Jawna implementacja IEnumerable
  {
    return GetEnumerator();
  }
}

class CountingEnumerator : IEnumerator<int>
{
  int current = -1;

  public bool MoveNext()
  {
    current++;
    return current < 10;
  }

  public int Current
  {
    get { return current; }
  }                                            ❸ Niejawna implementacja IEnumerator<T>.Current

  object IEnumerator.Current
  {
    get { return Current; }
  }                                            ❹ Jawna implementacja IEnumerator.Current

  public void Reset()
  {
    throw new NotSupportedException();
  }

  public void Dispose()
  {
  }
}

...
CountingEnumerable counter = new CountingEnumerable();   ❺ Dowód na działanie typu
foreach (int x in counter)
{
  Console.WriteLine(x);
}
```

Oczywiście użyteczność tego przykładu, z punktu widzenia zwracanego wyniku, jest znikoma, ale pokazuje on pewne „wyboje", przez które trzeba przejść, aby poprawnie zaimplementować iterację w sposób generyczny — przynajmniej wtedy, kiedy robisz to długą metodą i bez rzucania wyjątków przy próbach odwołania się do `Current` w nieodpowiednim momencie. Jeśli uważasz, że listing 3.10 jest trochę duży jak na wypisywanie liczb od 0 do 9, nie mogę zrobić nic więcej ponad przyznanie Ci racji — gdybyśmy chcieli iterować po bardziej użytecznych wartościach, byłoby jeszcze więcej kodu. Na szczęście w rozdziale 6. zobaczymy, że w pewnych sytuacjach C# 2 potrafi odsunąć od nas sporą część pracy związaną z iteratorami. Pokazałem tutaj pełny przykład po to, abyś mógł zobaczyć malutkie niedociągnięcia, celowo wprowadzone przez projektantów przy rozszerzaniu `IEnumerable` przez `IEnumerable<T>`. Nie sugeruję bynajmniej, że była to zła decyzja — pozwala ona na przekazanie dowolnej wartości `IEnume↪rable<T>` do metody napisanej w C# 1 poprzez parametr `IEnumerable`. Dzisiaj nie ma to już takiego wielkiego znaczenia jak jeszcze w 2005 roku, ale nadal stanowi użyteczną ścieżkę przekazywania danych z nowszej wersji kodu do starszej.

Potrzebujemy jedynie dwukrotnie skorzystać z triku jawnej implementacji interfejsu — pierwszy raz dla `IEnumerable.GetEnumerator` (❷) i drugi dla `IEnumerator.↪Current` (❹). Oba przypadki wywołują swoje generyczne odpowiedniki (odpowiednio (❶) i (❸)). `IEnumerator<T>` rozszerza `IDisposable`, zatem musimy dostarczyć implementację metody `Dispose`. Wyrażenie `foreach` w C# 1 wywołuje `Dispose` na iteratorze, jeżeli ten implementuje `IDisposable`. W C# 2, jeśli kompilator wykryje, że zaimplementowałeś `IEnumerable<T>`, stworzy bezwarunkowe wywołanie `Dispose` na końcu pętli (w bloku `finally`). Wiele iteratorów nie musi w praktyce niczego zwalniać, ale warto wiedzieć, że kiedy zwalnianie jest potrzebne, najczęściej stosowana metoda przechodzenia po kolekcji — `foreach` (❺) — automatycznie dokłada odpowiednie wywołanie. Ten mechanizm jest wykorzystywany najczęściej do zwalniania zasobów po zakończonej iteracji — przykładem może być iterator chodzący po wierszach pliku, który na końcu zwalnia uchwyt do pliku.

Przejdziemy teraz od wydajności czasu kompilacji do elastyczności w czasie wykonywania. Naszym ostatnim tematem jest refleksja. Mechanizm ten potrafi być podchwytliwy nawet w .NET 1.0/1.1, a po wprowadzeniu typów generycznych sytuacja staje się jeszcze ciekawsza. Chociaż środowisko dostarcza wszystkiego, czego potrzebujemy (z odrobiną pomocnej składni języka C# 2), zrozumienie tego tematu może być trudne. Proponuję zabrać się do niego bez zbędnego pośpiechu.

3.4.4. Refleksja i typy generyczne

Refleksja jest wykorzystywana przez programistów do różnych celów. W czasie wykonania możesz jej użyć do introspekcji obiektów i przeprowadzenia prostej formy bindowania danych. Możesz dokonać inspekcji katalogu pełnego modułów w celu znalezienia implementacji interfejsu plug-inu. Możesz napisać plik dla środowiska „odwracania kontrolki" (zobacz *http://mng.bz/xc3J*) w celu załadowania i dynamicznej konfiguracji komponentów Twojej aplikacji. Ze względu na tak dużą różnorodność sposobów wykorzystania refleksji nie będę się skupiał na żadnym konkretnym zastosowaniu, ale postaram się dostarczyć ogólne wytyczne odnośnie do tych zadań. Zaczniemy od rozszerzeń operatora `typeof`.

Używanie typeof z typami generycznymi

Refleksja sprowadza się do analizowania obiektów i sprawdzania ich typów. Zatem pierwszym krokiem, jaki trzeba wykonać, jest uzyskanie referencji do obiektu typu `System.Type`, który daje dostęp do wszystkich informacji na temat typu. Do uzyskania takiej informacji dla typów znanych w czasie kompilacji można użyć operatora `typeof`, który swoim zasięgiem obejmuje również typy generyczne.

Istnieją dwa sposoby użycia `typeof` w połączeniu z typami generycznymi — pierwszy wydobywa *definicję typu generycznego* (inaczej mówiąc, niezwiązany typ generyczny), a drugi konkretny typ skonstruowany. Aby uzyskać definicję typu generycznego — typu bez wskazywania jakichkolwiek argumentów dla parametrów typu — weź deklarację typu i usuń z niej wszystkie parametry typu, pozostawiając jedynie przecinki. Dla typu skonstruowanego musisz wskazać argumenty typu w taki sam sposób, jakbyś deklarował zmienną typu generycznego. Oba przypadki zostały zademonstrowane na listingu 3.11. Przykład korzysta z metody generycznej, dzięki czemu widzimy użycie `typeof` w połączeniu z parametrem typu (podobną rzecz widzieliśmy na listingu 3.8).

Listing 3.11. Użycie operatora typeof w połączeniu z parametrami typu

```
static internal void DemonstrateTypeof<X>()
{
  Console.WriteLine(typeof(X));            // Wyświetlenie parametru typu metody

  Console.WriteLine(typeof(List<>));       // 
  Console.WriteLine(typeof(Dictionary<,>));
                                           // ❶ Wyświetlenie typów zamkniętych (pomimo użycia parametru typu)
  Console.WriteLine(typeof(List<X>));
  Console.WriteLine(typeof(Dictionary<string, X>));

  Console.WriteLine(typeof(List<long>));   // Wyświetlenie typów zamkniętych
  Console.WriteLine(typeof(Dictionary<long, Guid>));
}

...
DemonstrateTypeof<int>();
```

Większość kodu z listingu 3.11 zachowuje się zgodnie z Twoimi oczekiwaniami, ale warto zwrócić uwagę na dwie rzeczy. Zobacz, w jaki sposób jest pobierana definicja typu generycznego dla `Dictionary<TKey, TValue>`. Przecinek w ostrych nawiasach jest potrzebny, aby powiedzieć kompilatorowi, żeby szukał typu z dwoma parametrami typu (pamiętaj, że może istnieć wiele typów o tej samej nazwie, pod warunkiem że każdy z nich ma inną liczbę tych parametrów). Według tej samej zasady definicję typu generycznego dla `MyClass<T1, T2, T3, T4>` otrzymasz, używając `typeof(MyClass<,,,>)`. Liczba parametrów jest określona w kodzie IL (oraz w pełnych nazwach typów środowiska) przez umieszczenie znaku lewego apostrofu po pierwszej części nazwy typu, a następnie liczby parametrów. Parametry typu są następnie wymienione w nawiasach kwadratowych, w przeciwieństwie do ostrych nawiasów, jakich używamy w kodzie. Dla przykładu drugi wiersz kończy się wyrażeniem ``List`1[T]``, co świadczy o tym, że jest jeden parametr typu, a wiersz trzeci wyrażeniem ``Dictionary`2[TKey,TValue]``.

Poza tym za każdym razem, kiedy w kodzie zostaje użyty parametr typu (`X`), w czasie wykonania w jego miejsce jest podstawiany argument typu. Dlatego wiersz (❶) wyświetla `System.Int32`, a nie, czego mogłeś oczekiwać, `List`1[X]`[11]. Innymi słowy, typ otwarty w czasie kompilacji może zostać zamknięty w czasie wykonania. *Takie zachowanie jest bardzo mylące. Powinieneś o tym pamiętać, kiedy otrzymywane wyniki różnią się od tego, czego oczekujesz.* Wydobycie otwartego, skonstruowanego typu w czasie wykonania wymaga trochę więcej wysiłku. Przykład znajdziesz w MSDN-owym opisie właściwości `Type.IsGenericType` (*http://mng.bz/9W6O*).

Oto wynik działania programu z listingu 3.11:

```
System.Int32
System.Collections.Generic.List`1[T]
System.Collections.Generic.Dictionary`2[TKey,TValue]
System.Collections.Generic.List`1[System.Int32]
System.Collections.Generic.Dictionary`2[System.String,System.Int32]
System.Collections.Generic.List`1[System.Int64]
System.Collections.Generic.Dictionary`2[System.Int64,System.Guid]
```

Mając dostęp do obiektu reprezentującego typ generyczny, otwiera się przed nami wiele możliwych dróg dalszego działania. Nadal są dostępne operacje, które wykonywaliśmy wcześniej (wyszukiwanie elementów składowych typu, tworzenie instancji itd.) — chociaż niektóre z nich nie mają zastosowania dla definicji typów generycznych — ale pojawiają się również nowe, które pozwalają odkryć generyczną naturę typu.

Metody i właściwości typu System.Type

`System.Type` ma zbyt wiele właściwości i metod, abyśmy mogli przyjrzeć się im wszystkim szczegółowo, jednak dwie z nich są szczególnie istotne: `GetGenericTypeDefinition` i `MakeGenericType`. Ich działanie jest zupełnie przeciwne — pierwsza operuje na skonstruowanym typie, pobierając z niego definicję typu generycznego, druga na definicji typu generycznego i zwraca skonstruowany typ. Być może lepiej by było, gdyby druga z metod została nazwana `ConstructType`, `MakeConstructedType` lub jeszcze inaczej, ale ze słowem *construct* lub *constructed* w nazwie. Niestety utknęliśmy z taką nazwą, jaką mamy.

Tak jak w przypadku zwykłych typów, istnieje tylko jeden obiekt klasy `Type` dla danego typu, zatem dwukrotne wywołanie `MakeGenericType` z tymi samymi argumentami spowoduje dwukrotne zwrócenie tej samej referencji. Na tej samej zasadzie wywołanie `GetGenericTypeDefinition` na obiektach stworzonych w oparciu o tę samą definicję typu generycznego da ten sam wynik, nawet jeśli skonstruowane typy są różne (np. `List<int>` i `List<string>`).

Kolejna metoda, istniejąca już w środowisku .NET 1.1, którą warto poznać, to `Type.` ↪`GetType(string)`. Jest ona związana z metodą `Assembly.GetType(string)`. Obie stanowią dynamiczny odpowiednik operatora `typeof`. Mógłbyś się spodziewać, że wynik działania każdego wiersza z listingu 3.11 można wstawić do metody `GetType` na odpowiednim module (ang. *assembly*), niestety życie nie jest takie proste. Nie ma problemu

[11] Z pełną świadomością odstąpiłem w tym miejscu od konwencji nazywania parametrów typu literą `T`, abyś mógł zobaczyć różnicę pomiędzy `T` w deklaracji `List<T>` i `X` w deklaracji naszej metody.

w przypadku zamkniętych typów skonstruowanych — wtedy wystarczy umieścić argument w nawiasach kwadratowych. Dla definicji typów generycznych trzeba usunąć nawiasy kwadratowe, w przeciwnym wypadku `GetType` będzie sądzić, że masz na myśli typ tablicowy. Wszystkie te operacje pokazuje w działaniu listing 3.12.

Listing 3.12. Różne metody pobierania typu obiektów na podstawie typów generycznych i skonstruowanych

```
string listTypeName = "System.Collections.Generic.List`1";

Type defByName = Type.GetType(listTypeName);

Type closedByName = Type.GetType(listTypeName + "[System.String]");
Type closedByMethod = defByName.MakeGenericType(typeof(string));
Type closedByTypeof = typeof(List<string>);

Console.WriteLine(closedByMethod == closedByName);
Console.WriteLine(closedByName == closedByTypeof);

Type defByTypeof = typeof(List<>);
Type defByMethod = closedByName.GetGenericTypeDefinition();

Console.WriteLine(defByMethod == defByName);
Console.WriteLine(defByName == defByTypeof);
```

Wynikiem działania listingu 3.12 jest czterokrotne wyświetlenie prawdy (`True`), co stanowi dowód na to, że niezależnie od tego, jak uzyskamy referencję do danego typu obiektu, będzie to zawsze jeden i ten sam obiekt.

Tak jak wspomniałem wcześniej, `Type` oferuje wiele metod i właściwości, takich jak `GetGenericArguments`, `IsGenericTypeDefinition` czy `IsGenericType`. Najlepszym sposobem na dalsze zgłębianie tego tematu jest przyjrzenie się dokumentacji właściwości `IsGenericType`.

Refleksja metod generycznych

Metody generyczne mają również podobny, chociaż mniejszy zestaw właściwości i metod. Demonstruje to listing 3.13, w którym metoda generyczna jest wywoływana przez refleksję.

Listing 3.13. Pobieranie i wywoływanie metody generycznej przez refleksję

```
public static void PrintTypeParameter<T>()
{
    Console.WriteLine (typeof(T));
}

...
Type type = typeof(Snippet);
MethodInfo definition = type.GetMethod("PrintTypeParameter");
MethodInfo constructed;
constructed = definition.MakeGenericMethod(typeof(string));
constructed.Invoke(null, null);
```

W pierwszej kolejności wydobywamy definicję metody generycznej, a następnie tworzymy ją, używając metody `MakeGenericMethod`. Podobnie jak w przypadku typów, moglibyśmy użyć innego sposobu, ale w przeciwieństwie do `Type.GetType`, w wywołaniu metody `Type.GetMethods` nie ma możliwości wyspecyfikowania skonstruowanej metody. Poza tym środowisko ma problem, jeśli istnieją metody przeciążone wyłącznie przez liczbę parametrów typu. Żeby obejść ten problem, musiałbyś wywołać `Type.GetMethods`, a następnie znaleźć tę metodę, której szukasz, przeszukując *wszystkie* z nich.

Po otrzymaniu skonstruowanej metody uruchamiamy ją. Ponieważ wywołujemy metodę statyczną, która nie ma żadnych normalnych argumentów wywołania, w naszym przykładzie są nimi dwie wartości `null`. Zgodnie z naszymi oczekiwaniami wynikiem działania jest `System.String`. Zwróć uwagę, że metody wydobyte z definicji typów generycznych nie mogą być wywołane bezpośrednio — musisz pobrać je z typu skonstruowanego. Odnosi się to zarówno do metod generycznych, jak i niegenerycznych.

> **RATUNEK ZE STRONY C# 4.** Jeżeli to zachowanie wydaje Ci się niechlujne, zgadzam się z Tobą. Na szczęście w wielu przypadkach pomoc niosą typy dynamiczne C#, redukując narzut pracy związany z refleksją typów generycznych. Pomoc nie jest dostępna we wszystkich przypadkach, dlatego warto mieć świadomość funkcjonowania przedstawionego powyżej kodu, jednak tam, *gdzie* jest dostępna, daje doskonałe wyniki. Typom dynamicznym przyjrzymy się z bliska w rozdziale czternastym.

`MethodInfo` oferuje całe mnóstwo metod i właściwości — ich lekturę proponuję rozpocząć od właściwości `IsGenericMethod` w dokumentacji MSDN. Zakładam, że informacje zawarte w tej sekcji są wystarczające, abyś mógł samodzielnie podjąć zgłębianie tego zagadnienia, oraz wskazują dodatkowy stopień skomplikowania, którego prawdopodobnie nie spodziewałeś się, rozpoczynając zabawę z dostępem do typów i metod generycznych poprzez refleksję.

Na tym kończymy część poświęconą cechom zaawansowanym. Przypominam, że przedstawiony materiał w żadnym przypadku nie wyczerpuje tematu, jednak większość programistów najprawdopodobniej nie potrzebuje wnikać głębiej w tajniki refleksji. Biorąc pod uwagę fakt, że wraz z wchodzeniem w coraz większe szczegóły czytanie dokumentacji staje się trudniejsze, mam nadzieję, że dla swojego własnego dobra zaliczasz się do tego obozu. Pamiętaj, że o ile nie programujesz samodzielnie i wyłącznie dla siebie, nie tylko Ty będziesz pracował nad swoim kodem. Jeśli będziesz potrzebować rozwiązań bardziej złożonych niż prezentowane tutaj, możesz z dużą pewnością założyć, że ktokolwiek będzie czytał Twój kod, nie zrozumie go bez Twojej pomocy. Z drugiej strony, jeśli zauważysz, że Twoi koledzy nie znają niektórych z przedstawionych do tej pory tematów, wyślij ich, proszę, do najbliższej księgarni...

Ostatnia część tego rozdziału omawia ograniczenia typów generycznych w C#, a także podobne cechy dostępne w innych językach programowania.

3.5. Ograniczenia typów generycznych C# i innych języków

Bez wątpienia typy generyczne składają się w dużej mierze na łatwość wyrażania, bezpieczeństwo typów oraz wydajność języka C#. Cecha ta została zaprojektowana, aby radzić sobie z większością zadań, do których programiści na ogół stosowali wzorce, ale bez towarzyszących im skutków ubocznych. Nie oznacza to wcale, że typy generyczne nie mają żadnych ograniczeń. Istnieją pewne problemy, które wzorce C++ rozwiązują z łatwością, a których nie da się rozwiązać przy użyciu typów generycznych C#. Podobnie w Javie, której typy generyczne są ogólnie mniej funkcjonalne od typów generycznych C#, istnieją pewne koncepcje, które można wyrazić w Javie, a nie da się tego zrobić w C#. W tej sekcji omówimy najczęściej spotykane słabości, a ja postaram się dokonać skrótowego porównania implementacji typów generycznych C#/.NET z wzorcami C++ i typami generycznymi Javy.

Warto zwrócić uwagę, że wymienione słabości języka w żadnym wypadku *nie wskazują*, iż można było ich uniknąć. Moją intencją nie jest stwierdzenie, że można było zrobić coś lepiej! Projektanci języka i platformy musieli wyważyć funkcjonalność języka ze stopniem jego skomplikowania (uwzględniając po drodze ograniczenie czasowe na zaprojektowanie i implementację całości). Pracując z typami generycznymi, z bardzo dużym prawdopodobieństwem nie napotkasz problemów, a jeśli tak się zdarzy, będziesz w stanie pokonać trudności dzięki wskazówkom z tego podrozdziału.

Zaczniemy od odpowiedzi na pytanie, które wcześniej czy później stawia każdy programista: dlaczego nie mogę przekonwertować `List<string>` na postać `List<object>`?

3.5.1. Brak wariancji typów generycznych

W sekcji 2.2.2 oglądaliśmy *kontrawariancję* tablic — możliwość przeglądania tablicy typu referencyjnego jako tablicy swojego typu bazowego lub tablicy implementowanych interfejsów. Ta idea ma dwie możliwe postacie, określane jako *kowariancja* i *kontrawariancja* lub wspólnie *wariancja*. Typy generyczne nie dają takiej możliwości — są *inwariancyjne*. Ma to na celu zachowanie bezpieczeństwa typu, ale czasem potrafi być irytujące.

Na początku pragnę wyjaśnić jedną rzecz — C# 4 poprawia do pewnego stopnia wariancję typów generycznych. W mocy *pozostaje* jednak nadal wiele z ograniczeń, o których będziemy mówić za chwilę. Tę sekcję można traktować jako wprowadzenie do idei wariancji. Na czym polega pomoc C# 4, dowiemy się w rozdziale 13., ale wiele z najbardziej wyrazistych przykładów wariancji typów generycznych bazuje na nowych cechach C# 3, włączając w to LINQ. Wariancja sama w sobie jest całkiem złożonym zagadnieniem, warto zatem poczekać z jej analizą do momentu, kiedy będziesz się czuł komfortowo z pozostałą częścią C# 2 i 3. Dla zachowania czytelności nie będę w tej sekcji wskazywał każdego miejsca, które różni się jedynie odrobinę od wersji C# 4... Wszystko stanie się jasne w rozdziale 13.

Dlaczego typy generyczne nie obsługują kowariancji?

Załóżmy, że mamy dwie klasy, `Turtle` i `Cat`, obie będące potomkami abstrakcyjnej klasy `Animal`. W poniższym przykładzie kod tablicy (po lewej stronie) jest poprawny w C# 2, a kod generyczny (po prawej) — nie:

Kod poprawny (w czasie kompilacji)	*Kod niepoprawny*
`Animal[] animals = new Cat[5];`	`List<Animal> animals = new List<Cat>();`
`animals[0] = new Turtle();`	`animals.Add(new Turtle());`

Kompilator nie ma problemu z drugim wierszem w obu przypadkach, ale pierwszy z nich po prawej stronie spowoduje następujący błąd kompilatora:

```
error CS0029: Cannot implicitly convert type
      'System.Collections.Generic.List<Cat>' to
      'System.Collections.Generic.List<Animal>'
```

Takie zachowanie zostało celowo wprowadzone przez projektantów środowiska i języka. Na usta ciśnie się pytanie, *dlaczego* zostało to zabronione? Odpowiedź jest w drugim wierszu. Nie ma w nim nic, co powinno wzbudzić nasze podejrzenia. Przecież `List` ↪`<Animal>` posiada metodę o sygnaturze `void Add(Animal value)` — powinna zatem istnieć możliwość dodania referencji klasy `Turtle` do dowolnej listy zwierząt. *W rzeczywistości* zmienna `animals` odnosi się jednak do `Cat[]` (w kodzie po lewej stronie) lub `List<Cat>` (po prawej) i w obu przypadkach do środka można wstawiać wyłącznie referencje klasy `Cat` lub klasy potomnej. Chociaż wersja z tablicą skompiluje się, nie będzie działać po uruchomieniu. Takie zachowanie zostało uznane przez projektantów typów generycznych za zachowanie gorsze niż błąd podczas kompilacji, co ma sens — głównym celem typów statycznych jest znajdowanie błędów, zanim kod zostanie uruchomiony.

> **DLACZEGO TABLICE SĄ KOWARIANCYJNE?** Skoro odpowiedzieliśmy, dlaczego typy generyczne są inwariancyjne, następne oczywiste pytanie brzmi: „Dlaczego tablice są kowariancyjne?”. Zgodnie z *Common Language Infrastructure Annotated Standard* (J.S. Miller i S. Ragsdale, Addison-Wesley Professional, Indianapolis 2003) w pierwszej edycji projektanci chcieli sięgnąć do jak najszerszej rzeszy użytkowników, co oznaczało umożliwienie kompilacji kodu pochodzącego z języka Java. Innymi słowy, .NET ma tablice kowariancyjne, ponieważ Java ma tablice kowariancyjne — mimo że ten element jest uznawany za wadę Javy.

Wiesz już, dlaczego jest tak, jak jest. Dlaczego powinieneś się tym przejmować i jak możesz ominąć to ograniczenie?

Gdzie mogłaby się przydać kowariancja?

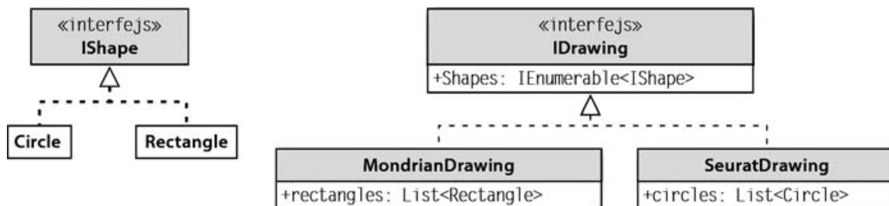
Podany przykład z listą jest najwyraźniej problematyczny. Możemy dodawać elementy do listy, ale w tym momencie tracimy bezpieczeństwo typu. Operacja dodania jest przykładem użycia wartości jako *danych wejściowych* do interfejsu programistycznego (kod wywołujący dostarcza wartości). Co by się stało, gdybyśmy ograniczyli się wyłącznie do pobierania wartości? Oczywistym przykładem tego jest `IEnumerator<T>` i (przez

skojarzenie) IEnumerable<T>. W rzeczy samej, są to niemal *kanoniczne* przykłady kowariancji typów generycznych. Razem opisują sekwencję wartości — wszystko, co o nich wiemy, to to, że każda z nich będzie kompatybilna z T w taki sposób, iż będziesz mógł napisać:

```
T currentValue = iterator.Current;
```

Jest to wykorzystanie zwykłej idei kompatybilności — byłoby zupełnie zrozumiałe, gdyby na przykład IEnumerator<Animal> zwracał referencje instancji typu Cat lub Turtle. Ponieważ nie istnieje sposób na „wepchnięcie" wartości o nieprawidłowym typie do konkretnej kolekcji, chcielibyśmy traktować IEnumerator<Cat> jako IEnumerator<Animal>. Oto przykład, gdzie takie zachowanie może się okazać użyteczne.

Załóżmy, że istnieje stworzony przez nas typ reprezentujący kształt w formie interfejsu (IShape). Przyjmijmy również, że istnieje inny interfejs, IDrawing, reprezentujący rysunek stworzony z kształtów. Chcemy stworzyć dwa konkretne typy rysunków — MondrianDrawing (zbudowany z prostokątów) i SeuratDrawing (zbudowany z okręgów)[12]. Hierarchie klas używane w przykładzie zostały przedstawione na rysunku 3.4.



Rysunek 3.4. Schemat hierarchii klas z naszego przykładu

Oba interfejsy muszą implementować interfejs IDrawing, zatem muszą udostępnić właściwość o sygnaturze:

```
IEnumerable<IShape> Shapes { get; }
```

Jednocześnie byłoby łatwiej, gdyby każdy typ rysunku mógł utrzymywać wewnętrznie listę o mocnym typie. Dla przykładu SeuratDrawing mógłby posiadać pole typu List ↪<Circle>. Taki typ będzie dla niego wygodniejszy niż List<IShape>, ponieważ pozwoli na manipulację okręgami bez konieczności rzutowania referencji na właściwy typ. Gdybyśmy mieli kolekcję List<IShape>, moglibyśmy zwrócić ją bezpośrednio lub przynajmniej opakować przez ReadOnlyCollection<IShape>, aby uniemożliwić osobie wywołującej nasz kod popsucie jej przez rzutowanie. Tak czy inaczej, implementacja samej właściwości wydaje się prosta i tania. Niestety, nie możemy tego zrobić, kiedy nasze typy nie pasują do siebie... nie możemy przekonwertować IEnumerable<Circle> na IEnumerable<Shape>. Co zatem *możemy* zrobić?

[12] Jeżeli nazwy klas nic Ci nie mówią, sprawdź ich znaczenie w Wikipedii (*http://pl.wikipedia.org/wiki/Piet_Mondrian* i *http://pl.wikipedia.org/wiki/Georges_Seurat*). Mają one dla mnie znaczenie z różnych powodów. Mondrian jest nazwą używanego w Google narzędzia do recenzji kodu, natomiast od imienia Seurata pochodzi imię głównego bohatera wspaniałego musicalu Stephena Sondheima, *Sunday in the Park with George*.

Jest kilka możliwości. Możemy:

- zmienić typ pola na `List<IShape>` i przyzwyczaić się do rzutowania, choć nie jest to rozwiązanie zadowalające, gdyż pod wieloma względami zaprzecza idei używania typów generycznych;
- użyć nowych cech oferowanych przez C# 2, które zobaczymy w rozdziale szóstym, do implementacji iteratorów; jest to rozwiązanie racjonalne *wyłącznie* dla tego przypadku (kiedy mamy do czynienia z `IEnumerable<T>`);
- zmusić każdą implementację właściwości `Shapes` do stworzenia nowej kopii listy, najlepiej — dla prostoty — z użyciem `List<T>.ConvertAll`; stworzenie niezależnej kopii kolekcji jest często wskazanym działaniem, ale wymaga wiele kopiowania, co czasem może się okazać mało wydajne;
- zmienić interfejs `IDrawing` na generyczny, wskazując typ kształtów używanych przez rysunek; w ten sposób `MondrianDrawing` mógłby implementować `IDrawing<Rectangle>`, a `SeuratDrawing` — `IDrawing<Circle>`, jednak jest to opłacalne tylko wtedy, kiedy jesteś właścicielem interfejsu;
- stworzyć klasę pomocniczą do adaptacji jednego typu interfejsu `IEnumerable<T>` na inny:

```
class EnumerableWrapper<TOriginal, TWrapper> : IEnumerable<TWrapper>
  where TOriginal : TWrapper
```

  Ponieważ ta konkretna sytuacja (`IEnumerable<T>`) jest wyjątkowa, moglibyśmy poradzić sobie, nawet jeśli zastosowalibyśmy metodę pomocniczą. .NET 3.5 dostarcza dwie użyteczne metody, które mogą się nam przydać: `Enumerable.` ↪`Cast<T>` i `Enumerable.OfType<T>`. Są one częścią LINQ i będziemy je analizować w rozdziale 11. Chociaż mówimy o przypadku szczególnym, jest to prawdopodobnie najbardziej powszechna forma kowariancji generycznej, z jaką będziesz miał do czynienia.

Kiedy natkniesz się na problemy związane z kowariancją, być może będziesz musiał rozważyć wymienione wyżej możliwości oraz inne, które przyjdą Ci do głowy. Wszystko zależy od konkretnej sytuacji, w jakiej się znajdziesz. Niestety kowariancja nie jest jedynym problemem, z którym musisz sobie poradzić. Istnieje jeszcze odwrotność kowariancji — *kontrawariancja*.

Gdzie mogłaby się przydać kontrawariancja?

Kontrawariancja wydaje się trochę mniej intuicyjna niż kowariancja, ale ma swój sens. Poprzez kowariancję chcieliśmy przekonwertować `SomeType<Circle>` na `SomeType<IShape>` (używając w naszym przykładzie `IEnumerable<T>`). Kontrawariancja dotyczy konwersji w drugą stronę — z `SomeType<IShape>` na `SomeType<Circle>`. Jak takie przejście może być bezpieczne? Kowariancja jest bezpieczna, kiedy `SomeType` opisuje wyłącznie operacje *zwracające* parametr typu, natomiast kontrawariancja jest bezpieczna, kiedy `SomeType` opisuje wyłącznie operacje *akceptujące* parametr typu[13].

---

[13] W rozdziale 13. przekonamy się, że jest jeszcze coś, co możemy dodać do tej ogólnej zasady.

Najprostszym przykładem typu, który używa swojego parametru typu wyłącznie na wejściu danych, jest `IComparer<T>`, stosowany powszechnie do sortowania kolekcji. Rozwińmy nasz pusty do tej pory interfejs `IShape`, dokładając do niego właściwość `Area`. W ten sposób można teraz łatwo napisać implementację `IComparer<IShape>`, która porównuje dwa kształty po ich polu powierzchni. *Chcielibyśmy* następnie móc napisać następujący kod:

Kod nieprawidłowy

```
IComparer<IShape> areaComparer = new AreaComparer();
List<Circle> circles = new List<Circle>();
circles.Add(new Circle(Point.Empty, 20));
circles.Add(new Circle(Point.Empty, 10));
circles.Sort(areaComparer);
```

To jednak nie zadziała, ponieważ metoda `Sort` na typie `List<Circle>` wymaga w praktyce typu `IComparer<Circle>`. Fakt, że nasz `AreaComparer` potrafi porównywać *dowolny* kształt, a nie tylko okręgi, nie robi żadnego wrażenia na kompilatorze. Z jego punktu widzenia `IComparer<Circle>` i `IComparer<IShape>` to dwa zupełnie różne typy. Czyż nie jest to szalone? Byłoby lepiej, gdyby metoda `Sort` miała następującą sygnaturę:

```
void Sort<S>(IComparer<S> comparer) where T : S
```

Na nasze nieszczęście to nie tylko nie jest sygnatura metody `Sort`, ale również *nie może* nią być — ograniczenie jest błędne, ponieważ dotyczy `S`, a nie `T`. Chcemy ograniczenia typu konwersji, ale w przeciwnym kierunku, oczekując, że `S` będzie gdzieś *wyżej* w drzewie dziedziczenia w stosunku do `T`, a nie *niżej*.

Wiedząc, że jest to niemożliwe, co *możemy* zrobić? Tym razem mamy mniej możliwości. Po pierwsze, moglibyśmy rozważyć ponownie ideę stworzenia pomocniczej klasy generycznej, jak pokazuje listing 3.14.

Listing 3.14. Obejście braku kontrawariancji przy użyciu klasy pomocniczej

```
class ComparisionHelper<TBase, TDerived> : IComparer<TDerived>
  where TDerived : TBase
{
  private readonly IComparer<TBase> comparer;

  public ComparisionHelper(IComparer<TBase> comparer)
  {
    this.comparer = comparer;
  }

  public int Compare(TDerived x, TDerived y)
  {
    return comparer.Compare(x, y);
  }
}
```

❶ **Poprawne ograniczenie parametru typu**
❷ **Zapamiętanie oryginalnego komparatora**
❸ **Użycie niejawnej konwersji w celu wywołania komparatora**

Ponownie jest to praktyczne wykorzystanie wzorca adaptera, tym razem jednak ze zmiennym typem elementów do porównania. Zapamiętujemy oryginalny komparator, dostarczający faktyczną logikę do porównywania elementów typu bazowego (❷), a następnie wywołujemy go, kiedy zostaniemy poproszeni o porównanie elementów typu potomnego (❸). Fakt, iż po drodze nie stosujemy żadnych rzutowań (nawet ukrytych),

daje nam pewność, że ta klasa zapewnia bezpieczeństwo typu. Dzięki dostępności niejawnej konwersji z typu `TDerived` na `TBase`, którą wymusiliśmy ograniczeniem typu (❶), jesteśmy w stanie wywołać komparator instancji klasy bazowej.

Drugą możliwością jest uczynienie klasy porównującej powierzchnie klasą generyczną z ograniczeniem typu w taki sposób, aby mogła porównać dwie dowolne wartości tego samego typu, o ile tylko typ ten implementuje interfejs `IShape`. Dla zachowania prostoty w sytuacji, kiedy tak naprawdę nie potrzebujesz tej funkcjonalności, mógłbyś zastosować klasę niegeneryczną, zmuszając ją jedynie do dziedziczenia po klasie generycznej:

```
class AreaComparer<T> : IComparer<IShape> where T : IShape
class AreaComparer : AreaComparer<IShape>
```

Oczywiście, możesz tak zrobić tylko w sytuacji, kiedy jesteś w stanie zmodyfikować klasę porównującą. Rozwiązanie może być funkcjonalne, ale nadal wydaje się nienaturalne. Dlaczego jesteś zmuszony do konstruowania komparatora w różny sposób dla różnych typów, w sytuacji kiedy ich zachowanie nie będzie różne? Dlaczego jesteś zmuszony stosować dziedziczenie po klasie dla uproszczenia kodu, kiedy Twoim celem nie jest jego specjalizacja?

Zauważ, że różne warianty dla kowariancji i kontrawariancji używają dużej ilości typów generycznych oraz ograniczeń w celu wyrażenia interfejsu w sposób bardziej ogólny lub dostarczenia generycznej klasy pomocniczej. Wiem, że dodanie ograniczenia sprawia wrażenie *zawężenia* ogólności, ale ogólność ta jest dodana w pierwszej kolejności przez uczynienie typu generycznym. Kiedy napotkasz podobny problem, przekształcenie typu w generyczny i dodanie pewnych ograniczeń powinno być pierwszym krokiem, jaki rozważysz. W takiej sytuacji — ze względu na wnioskowanie typu, sprawiające, że wariancja jest niewidoczna gołym okiem — często bardziej pomocne w porównaniu do typów są *metody* generyczne. Jest to szczególnie prawdziwe w przypadku C# 3, który posiada większe możliwości wnioskowania typu w porównaniu do C# 2.

To ograniczenie jest źródłem *bardzo* częstych pytań na forach dyskusyjnych C#. Pozostałe problemy mają raczej charakter akademicki lub dotyczą niewielkiej części społeczności programistów. Następna sytuacja dotyczy głównie osób, które w swojej codziennej pracy wykonują duże ilości obliczeń (naukowych lub związanych z zagadnieniami finansowymi).

3.5.2. Brak ograniczeń operatorów lub ograniczeń „numerycznych”

C# nie jest pozbawiony minusów, jeśli chodzi o kod mocno obciążony obliczeniami matematycznymi. Środowiska naukowe jako główne bariery przy adaptowaniu C# dla swoich celów podają niemożność wykonania jakichkolwiek operacji matematycznych poza zupełnie podstawową arytmetyką bez jawnego użycia klasy `Math`, a także brak możliwości łatwego definiowania i modyfikowania danych na przestrzeni całego kodu przy użyciu mechanizmu podobnego do synonimów typów uzyskiwanych na przykład w języku C przez słowo kluczowe `typedef`. Prawdopodobnie typy generyczne nie zostały

wymyślone po to, aby w pełni rozwiązać te problemy. Jest jednak pewien niuans uniemożliwiający im niesienie pomocy choćby w zadowalającym stopniu. Przyjrzyj się poniższej (generycznej) metodzie:

```
public T FindMean<T>(IEnumerable<T> data)
{
  T sum = default(T);
  int count = 0;
  foreach (T element in data)
  {
    sum += element;
    count++;
  }
  return sum / count;
}
```

Łatwo można zauważyć, że ten kod nie ma szans działać dla *wszystkich* typów danych. Co miałoby dla przykładu oznaczać dodanie jednego obiektu typu `Exception` do drugiego? Ten przypadek prosi się o wprowadzenie jakiegoś ograniczenia. Czegoś, co pozwoliłoby nam wyrazić nasz zamiar: dodanie dwóch instancji `T`, a następnie podzielenie instancji `T` przez liczbę całkowitą. Gdyby istniała taka możliwość, nawet przy ograniczeniu wyłącznie do typów wbudowanych, moglibyśmy pisać działające algorytmy generyczne, niezależnie od tego, czy typem danych jest `int`, `long`, `double`, `decimal`, czy jakikolwiek inny. Ograniczenie wyłącznie do typów wbudowanych byłoby rozczarowujące, ale lepsze to niż nic. Idealne rozwiązanie musiałoby pozwalać typom zdefiniowanym przez użytkownika zachowywać się tak jak wyrażenia matematyczne — mógłbyś wtedy na przykład zdefiniować typ `Complex` do obsługi liczb zespolonych[14]. Taka reprezentacja liczby zespolonej mogłaby następnie przechowywać każdy ze swoich składników w sposób generyczny. W ten sposób byłoby możliwe stworzenie liczb typu `Complex<float>`, `Complex<double>` itd.

Można wyobrazić sobie dwa powiązane ze sobą rozwiązania. Jednym byłoby umożliwienie stosowania ograniczeń na operatorach, dzięki którym mógłbyś zapisać rzecz następującą:

```
where T : T operator+ (T, T), T operator/ (T, int)
```

Taka definicja wymagałaby od `T` posiadania operacji, których użyliśmy wcześniej. Drugim rozwiązaniem byłoby zdefiniowanie kilku operatorów i być może konwersji, które musiałyby być obsługiwane przez typ, aby mógł on sprostać narzuconym ograniczeniom — takie ograniczenie moglibyśmy nazwać „ograniczeniem numerycznym" i zapisać jako `where T: numeric`.

Problem z jednym i drugim rozwiązaniem jest taki, że nie mogą być one wyrażone jako zwykłe interfejsy, ponieważ przeciążanie operatorów jest wykonywane na elementach *statycznych*, których nie można użyć do zaimplementowania interfejsów. W takiej sytuacji atrakcyjna wydaje się idea *interfejsów statycznych*, czyli deklarujących jedynie statyczne elementy składowe (metody, operatory i konstruktory). Tego typu statyczne interfejsy byłyby użyteczne jedynie w ograniczeniach typów, ale oferowałyby

[14] Oczywiście, pod warunkiem że nie używasz środowiska .NET 4, które posiada już taki typ o nazwie `System.Numerics.ComplexNumber`.

możliwość dostępu do statycznych elementów z zachowaniem bezpieczeństwa typu. Oczywiście jest to tylko bujanie w obłokach (*http://mng.bz/3Rk3*) — nic mi nie wiadomo o jakichkolwiek planach udostępnienia takiej możliwości w przyszłych wersjach języka C#.

Dwa najsprytniejsze obejścia tego problemu wymagają najnowszej wersji środowiska .NET. Pierwsze z nich zostało opracowane przez Marca Gravella (*http://mng.bz/9m8i*) i używa drzew wyrażeń (z którymi spotkamy się w rozdziale 9.) w celu zbudowania metod dynamicznych. Drugie to zastosowanie dynamicznych cech języka C# 4. Przykład zobaczymy w rozdziale 14. Już z samego opisu możesz jednak wywnioskować, że oba rozwiązania korzystają z mechanizmów dynamicznych — o ich pomyślnej współpracy z danym typem przekonasz się dopiero w czasie wykonywania programu. Istnieje jeszcze kilka obejść korzystających nadal z typów statycznych, ale mają one inne niedoskonałości (i, co może być zaskakujące, potrafią być wolniejsze w porównaniu do kodu dynamicznego).

Dwa ograniczenia typów generycznych, którym się do tej pory przyglądaliśmy, mają charakter całkiem praktyczny — masz szansę spotkać się z nimi w trakcie swojej pracy deweloperskiej. Jeśli jesteś osobą ciekawą świata, tak jak ja, możesz się zastanawiać, czy istnieją inne ograniczenia, które nie wpływają na tempo pracy, ale stanowią swego rodzaju intelektualne ciekawostki. Na przykład dlaczego „generyczność" ogranicza się jedynie do metod i typów?

3.5.3. Brak generycznych właściwości, indekserów i innych elementów typu

Widzieliśmy typy generyczne (klasy, struktury, delegaty i interfejsy), a także metody generyczne. Istnieje jeszcze całe mnóstwo innych elementów, które *mogłyby* być sparametryzowane. Nie ma jednak generycznych właściwości, indekserów, operatorów, konstruktorów, metod finalizujących ani zdarzeń. Zacznijmy od wyjaśnienia, co mamy tutaj na myśli. Indekser może posiadać typ zwracany będący parametrem typu — bezspornym przykładem jest `List<T>`. `KeyValuePair<TKey, TValue>` jest podobnym przykładem dla właściwości. To, czego *nie można* mieć, to indekser lub właściwość, a także każdy inny element z powyższej listy, które by posiadały *dodatkowe* parametry typu. Odkładając chwilowo na bok prawdopodobną deklarację, zobaczmy, jak takie elementy musiałyby być używane w kodzie:

```
SomeClass<string> instance = new SomeClass<string><Guid>("x");
int x = instance.SomeProperty<int>;
byte y = instance.SomeIndexer<byte>["key"];
instance.Click<byte> += ByteHandler;
instance = instance +<int> instance;
```

Zgodzisz się chyba ze mną, że te przykłady wyglądają trochę niepoważnie. Metod finalizujących nie da się wywołać jawnie z poziomu kodu C#, stąd nie ma ich w powyższym przykładzie. Według mojej oceny fakt, iż nie możemy wykonać żadnej z przedstawionych operacji, nie spowoduje większych problemów w kodzie. Jest to czysto akademicki problem, o którym warto wiedzieć.

Jedynym wyjątkiem *mógłby być* tutaj konstruktor. W tym przypadku podobny efekt można jednak osiągnąć, stosując statyczną metodę generyczną wewnątrz klasy, chociaż składnia z dwoma listami argumentów typu wygląda straszliwie.

Nie są to w żadnym wypadku *jedyne* ograniczenia typów generycznych C#, ale moim zdaniem te masz największą szansę spotkać na swojej drodze — w trakcie codziennej pracy, prowadząc dyskusję z innymi programistami lub w wolnej chwili analizując całościowo jakąś cechę języka. W dwóch kolejnych sekcjach przekonamy się, jak dwa inne języki, najczęściej porównywane z typami generycznymi C#, radzą sobie z podobnym problemem. Są to C++ (z wzorcami) i Java (z typami generycznymi w wersji Java 5). Zaczniemy od C++.

3.5.4. Porównanie z C++

Wzorce C++ można traktować trochę jak makra posunięte do granic możliwości. Są niezwykle funkcjonalne, ale wprowadzają pewien koszt zarówno pod względem skomplikowania kodu, jak i łatwości jego zrozumienia.

Kiedy w C++ zostanie użyty wzorzec, kod jest kompilowany dla określonego zbioru argumentów, tak jakby argumenty wzorca znajdowały się w kodzie źródłowym. Oznacza to, że nie ma większej potrzeby wprowadzania ograniczeń, ponieważ kompilator i tak sprawdzi, czy wolno Ci zrobić z danym typem to, co chcesz, podczas kompilowania kodu źródłowego dla określonych argumentów wzorca. Mimo to komitet standaryzacji C++ uznał, że ograniczenia mogą być nadal przydatne. Miały one trafić do wersji C++0x (kolejnej wersji C++), ale tak się nie stało. Być może pewnego dnia ujrzą jeszcze światło dzienne pod nazwą *konceptów*.

W C++ kompilator jest wystarczająco mądry, aby skompilować kod wyłącznie raz dla dowolnego zestawu argumentów wzorca, ale nie jest w stanie współdzielić kodu w sposób, jaki robi to CLR dla typów referencyjnych. Ten brak współdzielenia ma swoje zalety — pozwala na optymalizację pod względem typów, na przykład przez tworzenie wywołań inline dla pewnych parametrów typu, ale nie dla wszystkich pochodzących z tego samego wzorca. Oznacza to również, że rozstrzyganie przeciążeń w C++ może być wykonane niezależnie dla każdego zestawu typu parametrów w porównaniu do wyłącznie jednego wspólnego przebiegu dla całego kodu w C#, wynikającego z niepełnej wiedzy kompilatora C# spowodowanej obecnością ograniczeń typów.

Nie zapominaj, że w przypadku C++ ma miejsce tylko jeden rodzaj kompilacji, podczas gdy w modelu .NET mamy do czynienia z „kompilacją do postaci IL", a następnie „kompilacją JIT do postaci natywnej". Program C++ używający standardowego wzorca na dziesięć różnych sposobów będzie zawierał w programie 10 kopii kodu wzorca. Podobny program w C#, używający typu generycznego ze środowiska na dziesięć różnych sposobów, nie będzie w ogóle zawierał kodu typu generycznego — będzie się do niego odwoływał, a kompilator JIT będzie kompilował tyle różnych wersji, ile będzie potrzebnych (mówiliśmy o tym w sekcji 3.4.2) w czasie wykonania.

Jedyną znaczącą przewagą, jaką C++ posiada nad typami generycznymi C#, jest to, że argumenty wzorca nie muszą być nazwami typu. Równie dobrze mogą to być nazwy zmiennych, funkcji, a także wyrażenia stałe. Znanym przykładem jest typ bufora, którego rozmiar jest jednym z argumentów wzorca — `buffer<int, 20>` będzie zawsze buforem 20 zmiennych całkowitych, a `buffer<double, 35>` będzie zawsze buforem 35

liczb typu `double`. Ta zdolność ma kluczowe znaczenie przy metaprogramowaniu z użyciem wzorców (zobacz *http://mng.bz/c1G0*) — zaawansowanej techniki C++, która dla mnie jest przerażająca już w samej idei, ale w rękach eksperta może się okazać bardzo produktywna.

Wzorce C++ są również bardziej elastyczne pod innymi względami. Nie „cierpią" z powodów opisanych w sekcji 3.5.2 i cechuje je mniejsza liczba ograniczeń: możesz stworzyć klasę potomną w oparciu o parametr typu i wyspecjalizować wzorzec dla określonego zestawu argumentów. Druga cecha pozwala autorowi wzorca na stworzenie ogólnego kodu, w sytuacji kiedy brakuje mu szczegółowej wiedzy na temat danego zagadnienia, ale również kodu wyspecjalizowanego (często zoptymalizowanego) dla określonych typów.

We wzorcach C++ istnieją te same problemy z wariancją, z jakimi borykają się generyki .NET — przykład podany przez Bjarne'a Stroustrupa[15] mówi, że nie ma niejawnej konwersji pomiędzy `vector<shape*>` i `vector<circle*>`. Powód braku takiej możliwości jest podobny — w tym przykładzie dozwolone byłoby włożenie kwadratu w miejsce kółka.

Chętnym do głębszego poznania wzorców C++ polecam książkę autorstwa Stroustrupa *Język C++* (WNT, Warszawa 2002). Nie jest to książka łatwa w lekturze, ale sam rozdział poświęcony wzorcom jest w miarę przystępny (pod warunkiem wcześniejszego przyswojenia sobie terminologii i składni C++). Dalsze szczegóły na temat porównania generyków .NET znajdziesz we wpisie zamieszczonym na stronie zespołu Visual C++ (*http://mng.bz/En13*).

Drugim oczywistym językiem do porównania z C# pod względem typów generycznych jest Java, która wprowadziła tę cechę do głównego nurtu języka w wersji 1.5[16] kilka lat po tym, jak inne projekty doprowadziły do powstania języków podobnych do Javy i obsługujących ten mechanizm.

3.5.5. Porównanie z typami generycznymi Javy

Tam, gdzie C++ — w porównaniu do C# — dołączał *więcej* wzorców w wygenerowanym kodzie, Java dołącza *mniej*. Prawdę mówiąc, środowisko wykonania Javy nie ma pojęcia o istnieniu typów generycznych. Kod bajtowy Javy (mniej więcej odpowiednik kodu IL pod względem terminologii) zawiera dodatkowe metadane, aby wskazać, iż chodzi o kod generyczny, jednak po kompilacji nie pozostaje niemal żaden ślad wskazujący na generyczność kodu. Instancja typu generycznego z całą pewnością zna tylko swoją niegeneryczną naturę. Dla przykładu instancja `HashSet<T>` nie wie, czy została utworzona jako `HashSet<String>`, czy `HashSet<Object>`. Kompilator zajmuje się rzutowaniem w miejscach, gdzie jest to wymagane, a także wykonuje więcej sprawdzeń odnośnie do poprawności kodu. Oto przykład — najpierw generyczny kod Javy:

```
ArrayList<String> strings = new ArrayList<String>();
strings.add("cześć");
String entry = strings.get(0);
strings.add(new Object());
```

[15] Autora C++.

[16] Lub 5.0, w zależności od tego, którego systemu numerowania używasz.

a teraz odpowiednik powyższego kodu w formie niegenerycznej:

```
ArrayList strings = new ArrayList();
strings.add("cześć");
String entry = (String) strings.get(0);
strings.add(new Object());
```

Oba fragmenty doprowadziłyby do powstania tego samego kodu bajtowego z wyjątkiem ostatniej linii, która jest poprawna w kodzie niegenerycznym, ale spowodowałaby błąd kompilatora w wersji generycznej. Typu generycznego możesz użyć jako „surowego" typu, będącego odpowiednikiem `java.lang.Object` dla każdego z argumentów typu. Takie przypisanie i utratę informacji nazywa się *wykreślaniem typu* (ang. *type erasure*). Java nie posiada definiowanych przez użytkownika typów wartościowych, ale nie ma to znaczenia, ponieważ jako argumentów typu nie możesz użyć nawet typów wbudowanych. Zamiast tego jesteś zmuszony do korzystania z wersji opakowanej, na przykład `ArrayList<Integer>` w przypadku listy liczb całkowitych.

Możesz bez żalu przyznać, że taka funkcjonalność jest odrobinę rozczarowująca w porównaniu do typów generycznych oferowanych przez C#. Należy jednak dodać, że Java ma też jaśniejsze strony generyków:

- Maszyna wirtualna nie ma pojęcia o typach generycznych, zatem możesz używać skompilowanego kodu opartego na typach generycznych w starszych wersjach środowiska (pod warunkiem że nie stosujesz klas i metod nieistniejących w tych wersjach). Wersjonowanie środowiska .NET jest o wiele bardziej restrykcyjne — musisz wskazać, czy wersja każdego modułu, do którego się odnosisz, ma dokładnie pasować. Ponadto kod zbudowany do uruchomienia na wersji 2.0 środowiska CLR nie będzie działał w środowisku .NET 1.1.
- Nie musisz poznawać nowego zestawu klas, aby móc skorzystać z typów generycznych w Javie. Tam, gdzie „niegeneryczny" programista użyłby `ArrayList`, programista „generyczny" zastosuje `ArrayList<T>`. Istniejące klasy mogą być w sposób rozsądny wykorzystane w wersji generycznej.
- Poprzednia cecha została całkiem efektywnie wykorzystana w systemie refleksji — klasa `java.lang.Class` (odpowiednik `System.Type`) jest generyczna, co pozwala na rozciągnięcie systemu bezpieczeństwa typu czasu kompilacji na wiele sytuacji, w których zachodzi refleksja. Jednak w pozostałych przypadkach może już nie być tak różowo.
- Java obsługuje wariancję typów generycznych poprzez wieloznaczniki. Na przykład `ArrayList<? extends Base>` można odczytać jako: „`ArrayList` pewnego typu, który wywodzi się z typu `Base`, chociaż nie wiemy, jaki dokładnie jest to typ". Wrócimy do tego tematu, kiedy w rozdziale 13. będziemy omawiać wsparcie dla wariancji typów generycznych w C# 4.

Moim zdaniem typy generyczne .NET górują pod niemal każdym względem, chociaż kiedy utknę na problemie związanym z wariancją/kontrawariancją, marzy mi się posiadanie wieloznaczników. Ograniczona wariancja generyków w C# 4 poprawia nieco sytuację, ale niekiedy nadal się zdarza, że model wariancyjny Javy okazuje się lepszy. Java z typami generycznymi jest lepsza od Javy bez nich, ale nie wiąże się z tym szczególny zysk wydajnościowy, a bezpieczeństwo typu jest zapewnione jedynie na etapie kompilacji.

3.6. Podsumowanie

No i dotarliśmy! Dobrą rzeczą jest to, że typy generyczne są znacznie prostsze w użyciu niż w ich opisywaniu. Chociaż *mogą* być skomplikowane, są powszechnie uważane za jeden z najważniejszych i najbardziej funkcjonalnych dodatków, jaki pojawił się w C# 2. Najgorszą rzeczą, która może się przydarzyć podczas pisania przy użyciu typów i metod generycznych, jest to, że jeśli kiedykolwiek będziesz musiał powrócić do C# 1, będziesz ogromnie tęsknił za tym mechanizmem.

Moją intencją w tym rozdziale nie było opisanie w najdrobniejszych szczegółach tego, co jest możliwe i niemożliwe przy użyciu typów generycznych — to zadanie należy do specyfikacji języka, która podaje wszystko w formie suchych faktów. Starałem się raczej podejść do tematu praktycznie, podając informacje, które będą Ci potrzebne w codziennej pracy, i dorzucając jedynie okruszki teorii dla zaspokojenia potrzeb akademickich.

Widzieliśmy trzy główne korzyści użycia typów generycznych: zapewnienie bezpieczeństwa typu w czasie kompilacji, wydajność i przejrzyste wyrażanie myśli w kodzie. Umożliwienie środowisku IDE oraz kompilatorowi wczesnej weryfikacji Twojego kodu jest zdecydowanie dobrym podejściem. Kwestią sporną pozostaje to, czy więcej można oczekiwać od inteligentnych narzędzi sprawdzających, czy też od samego aspektu bezpieczeństwa typu.

Wydajność została poprawiona najbardziej w odniesieniu do typów wartościowych, które nie muszą być dłużej opakowywane i odpakowywane w kontekście interfejsów o mocnym typie, ze szczególnym wskazaniem na kolekcje generyczne udostępnione przez .NET 2.0. W przypadku typów referencyjnych wzrost wydajności jest niewielki.

Dzięki typom generycznym Twój kod jest w stanie jaśniej wyrazić Twoje intencje. Komentarze lub długie nazwy zmiennych opisujące faktycznie użyte typy zostają zastąpione przez deklaracje typów, które spełniają to samo zadanie. Komentarze i nazwy zmiennych mogą z biegiem czasu stracić na aktualności, szczególnie jeśli zostaną pominięte przy refaktoryzacji kodu — definicja typu pozostanie jednak zawsze aktualna.

Typy generyczne nie są w stanie wykonać *wszystkich* zadań, jakie chcielibyśmy przed nimi postawić. Niektóre z ograniczeń przestudiowaliśmy w tym rozdziale. Jeśli jednak faktycznie weźmiesz na poważnie C# 2 i typy generyczne w środowisku .NET 2.0, przekonasz się o ich skuteczności i będziesz je stosował w swoim kodzie niezwykle często.

Ponieważ inne cechy języka opierają się na typach generycznych, ten temat będzie się przewijał w kolejnych rozdziałach. Nie musimy szukać daleko, temat omawiany w następnej kolejności — typy nullowalne, implementowane przez `Nullable<T>` — byłby bardzo trudny do zrealizowania bez wsparcia typów generycznych.

Skorowidz

A

- agregacja, 328, 574
- akcja
  - Action<IList<double>>, 181
  - Action<int>, 181
  - Action<string>, 181
  - instancji delegata, 62
- algorytm
  - MD5, 243
  - SHA-1, 243
- algorytmy kryptograficzne, 607
- aliasy
  - przestrzeni nazw, 226, 237–239
  - zewnętrzne, 240
- analiza
  - arytmetyczna, 550
  - kodu, 567
  - nazwy i argumentów, 473
  - poprawności kodu, 545
  - selektywna, 553
  - statyczna, 533, 545–547, 553
  - wyrażenia, 331
  - zakresu, 550
- aplikacja e-commerce, 142
- argument, 430
  - null, 441
  - pozycyjny, 441
- argumenty
  - nazwane, 38, 427, 437, 443
    - składnia, 437
  - pozycyjne, 439
  - typu, 96
- asercje, 524, 533, 534
- asocjacja prawostronna, 163
- ASP.NET, 54
- atrybut
  - [Pure], 535
  - InternalsVisibleToAttribute, 246
  - System, 318
- automatyczne dokumentowanie kontraktów, 554
- automatyzowanie Worda, 447

B

baza danych, 383
BCL, Base Class Library, 524, 561
bezpieczeństwo, 247, 470
  typów, 69, 72, 128
bezpieczna konwersja, 161
biblioteka, 606
  CardSpace, 607
  CCR, 219
  GTK#, 54
  klas BCL, 524, 561
  MiscUtil, 332
  mscorlib, 537
  Parallel Extensions, 408, 607
  Reactive Extensions, 341, 608
  typów, 451
  WCF, 607
  WF, 607
  WPF, 606
biblioteki
  LINQ, 413
  .NET, 54
binder, 493, 515
blok
  finally, 209, 216
  try, 466
blokada, 466
  na referencji this, 467
blokowanie, 465
błąd zmodyfikowany, 343
błędy
  asercji, 562
  czasu kompilacji, 502
  i subskrypcje, 361, 363
  kompilacji, 179, 258, 301, 432, 502
  metody int.TryParse, 531
  odtwarzacza mediów, 360
  w kodzie, 567
  w wywołaniach dynamicznych, 502
buforowanie, 323
bufory o stałym rozmiarze, 226, 243, 245

C

C# 2.0, 605
C# 3.0, 605
C# 4.0, 606
C++, 136
callback, 539
CCR, Concurrency and Coordination Runtime, 219
cechy C#, 226, 605
cel akcji, 63
CHESS, 567
ciało wyrażenia lambda, 306
CLI, Common Language Infrastructure, 53, 105
CLR, Common Language Runtime, 53, 430, 462, 491, 608
CLS, Common Language Specification, 526
CodeDOM, 292
COM, 50, 427, 446, 479
Compact Framework, 609
CoreCLR, 610
cykl życia wyrażenia dynamicznego, 492
czas życia
  instancji, 215
  zmiennych przechwyconych, 190

D

debugger, 243
definicja
  domknięcia, 187
  typu generycznego, 124
deklaracja
  funkcji statycznej, 65
  klasy wewnętrznej, 541
  metody generycznej, 100
  pętli, 197
  typów, 505
  typu częściowego, 228
  typu delegatowego, 61, 65
deklarowanie
  metod rozszerzających, 315
  parametrów opcjonalnych, 431
delegat
  Action<Action<T>>, 461
  Converter<string, Guid>, 101
  Converter<TInput, TOutput>, 459
  Comparision<T>, 535
  Func<TextReader>, 217
  MethodInvoker, 175
  Predicate<T>, 535
delegaty, 60, 64, 68, 89
  Action<...>, 282
  Func<...>, 281
  generyczne, 84
  typu generycznego, 101
  typu multicast, 464
dereferencja, 72
diagram
  interakcji, 206
  klas, 342, 392, 400, 411
  sekwencji, 415

DLR, Dynamic Language Runtime, 50, 87, 297, 428, 491–496, 521, 607
dodawanie
  bajtów do kolekcji, 120
  elementów statycznych, 512
  liczb, 475
  warunków wstępnych, 542
  własnych operatorów, 332
dokumentacja
  kontraktów kodu, 535, 554, 556
  projektu, 421
  XML, 525, 554, 555
DOM, document object model, 510
domknięcia, 186
dostawcy LINQ, 397
dostęp do
  gettera i settera, 226, 236
  instancji klasy, 215
  silnika DLR, 494
  statycznych elementów, 135
  tablicy, 550
drzewa wyrażeń, 280, 289–297, 424
  reprezentacja graficzna, 290, 294
drzewo XML DOM, 507
DSL, domain-specific language, 331
dwustopniowe pobranie, 405
dynamiczne
  dodawanie, 475
  konwersje wyniku, 499
  sumowanie elementów, 486, 487
  środowisko wykonania, 297
  typy, 52
  wnioskowanie typu, 485
  wyliczanie wyrażeń, 497
  wywoływanie metod, 485, 518
  zachowanie, 470
dyrektywa
  EXPENSIVE_CONTRACTS, 561
  using, 237
  pragma, 226, 241
dziedziczenie, 133, 444
  kontraktów, 538

E

element
  domyślny, 449
  exception, 555
  requires, 555
encje, 383, 385
Entity Framework, 383, 607
enumeracje, 77, 200, 383
ewaluacja argumentów, 439
Excel, 50, 478

F

FIFO, first in, first out, 598
filtrowanie, 276, 323–351, 417, 584
  kolekcji, 47, 222
  listy, 183
  sekwencji, 361
flaga typu boolowskiego, 45, 145
formularz z przyciskiem, 173
forum kontraktów kodu, 545
fragmenty kodu, 56
Func, 84
funkcja
  GetConsoleScreenBufferEx, 244
funkcje
  anonimowe, 259, 281, 308
  wyższego rzędu, 286, 468

G

garbage collector, 145
generacja, 579
generator
  kodu, 229
  liczb losowych, 534
generowanie
  kodu SQL, 388, 389
  skrótu obiektu, 274
generyczne typy delegatów, 85
generyki, 245
  C#, 465
  Javy, 464
getter publiczny, 256
gettery, 36, 235
globalna przestrzeń nazw, 239
głęboka optymalizacja, 494
Google Protocol Buffer, 230
gorące obiekty obserwowalne, hot observable, 417
Groovy, 332
grupowanie, 329, 417, 580
  błędów, 370, 372
  elementów według klucza, 369
GUID, 243

H

Hashtable, 165
hierarchia klas, 130

I

identyfikator SupplierID, 47
identyfikatory przezroczyste, 356–358
ignorowanie ostrzeżenia, 242
IL, Intermediate Language, 53
iloczyn logiczny, 159
impas, deadlock, 466
implementacja, 212
  Enumerable, 395
  GetDynamicMemberNames, 517
  IDynamicMetaObjectProvider, 506, 518
  IEnumerable, 201
  IEnumerable<T>, 391
  IQueryable, 393
  IQueryable<T>, 391
  IQueryProvider, 394
  Queryable, 395
  XML DOM, 508
implementowanie
  dynamicznej właściwości, 515
  iteratora, 200
  LINQ dla Obiektów, 218, 360
  metody LINQ, 218
  metody generycznej, 102
  typów generycznych, 110
  wzorca TryXXX, 166
indekser, 135
  SynonymInfo, 449
indeksery nazwane, 449
informacja o typie, 500
inicjalizacja, 262
inicjalizator
  kolekcji, 266–269
  obiektów, 262–266, 442
  obiektów i kolekcji, 254
  obiektu anonimowego, 275
  odwzorowujący, 275
  projekcji, 329
  tablic, 266
instancja
  bez wartości, 152
  budowniczego, 270
  delegata, 62, 189
  Nullable<T>, 149
  typu generycznego, 137
  zmiennej, 197
  zmiennej lokalnej, 192
integracja LINQ, 404
IntelliSense, 180, 317
interfejs
  ICloneable, 74
  ICollection<T>, 267, 588, 590
  IComparer, 40, 115
  IDictionary<TKey, TValue>, 589
  IDisposable, 123, 216
  IDrawing, 131
  IDynamicMetaObjectProvider, 518
  IEnumerable, 98, 121, 130, 200, 588
  IEnumerable<ITask>, 221
  IEnumerable<string>, 338
  IEnumerable<T>, 121, 588
  IEnumerator, 200
  IEnumerator<T>, 122, 588
  IList<T>, 313, 588
  IObservable<T>, 414, 608
  IObserver<T>, 416, 608
  IProducerConsumerCollection<T>, 599
  IQueryable, 390, 398
  IQueryable<T>, 319, 391, 392
  IQueryProvider, 390, 391
  IQueryProvider.Execute, 398
  ISet<T>, 589, 596
interfejsy, 77, 588
  argumentów typu, 109
  inwariancyjne, 588
  kontrawariancja, 458
  kowariancja, 457
  wariancja, 455
  reprezentujące zapytanie, 391
  słownika, 507
  statyczne, 134
  z klasą kontraktu, 540
  związane z IQueryable<T>, 392
interpreter, 472, 473
inwariancja, 455
inwariant obiektu, 532
inwarianty, 531, 539
iteracja, 121, 200
  po IQueryable, 391
  po wierszach, 214, 215
iterator, 90, 200
  generyczny, 122
  kolekcji, 202

J

jawna implementacja interfejsów, 75
jawne rzutowanie wartości, 110, 271
język
  Axum, 567
  C#, 53, 566
  C , 527
  Eiffel, 525
  F#, 546, 567
  IL, 53, 264
  IronPython, 479, 480
  Spec#, 546
  XAML, 230
  XPath, 404
języki
  dynamiczne, 479
  dziedzinowe, 331

K

kacze typizowanie, 472, 487, 488
Kanał 9, 414
kapsułkowanie, 237
klasa
  ArrayList, 36
  BindingList<T>, 593
  BlockingCollection<T>, 599
  Collection<T>, 593
  ConcurrentDictionary, 594
  Contract, 530, 535
  CurrentCultureUpperCaseFormatter, 541
  DataReader, 323
  DynamicObject, 511
    obsługa metod, 512
    wirtualne metody TryXXX, 515
  DynamicXElement, 511, 517
  Enemy, 247
  Enumerable, 321, 324, 334, 410
  ExpandoObject, 506, 512
  Expression, 289
  Expression<TDelegate>, 291
  HashSet<T>, 597
  ICaseConverterContracts, 540
  Interlocked, 256
  InternalsVisibleToAttribute, 226
  IterationSample, 203
  Lazy<T>, 608
  LinkedList<T>, 592
  List<T>, 590
  Message, 442
  MetaRumpelstiltskin, 520
  Microsoft.CSharp.RuntimeBinder.Binder, 493
  Pair<T1, T2>, 608
  ParallelEnumerable, 410
  ParallelQuery<TSource>, 410
  Person, 263
  PlainInner, 119
  Queryable, 321, 324, 395
  Queue<T>, 598
  Random, 534
  ReadOnlyCollection<T>, 594
  ReadOnlyObservableCollection<T>, 594
  RouteValueDictionary, 594
  Rumpelstiltskin, 518
  ScriptScope, 481
  Snippet, 499
  SortedDictionary<TKey, TValue>, 595
  SortedList<TKey, TValue>, 595
  SortedSet<T>, 597
  Stream, 313, 315, 319
  StreamUtil, 314–316
  System.Diagnostics.Contracts.Contract, 527
  System.Dynamic.ExpandoObject, 506
  System.Math, 233
  System.Nullable, 145, 151
  Table<T>, 391
  TextReader, 216
  Tuple, 608
  User, 383
klasy
  częściowe, 230
  encji, 230
  generyczne, 114, 291
  kontraktu, 539
  narzędziowe, 233, 314
  niegeneryczne, 116, 133
  pomocnicze, 168
  statyczne, 226, 233, 312
  zagnieżdżone, 212
  zamknięte, 107
  zapieczętowane, sealed, 235
klauzula
  from, 366
  group … by, 343, 369
  join … into, 359, 363, 374, 419
  let, 356, 357, 387
  orderby, 353, 378
  select, 343, 352
  where, 47, 351, 352
klauzule grupujące, 372

klucz
  grupowania, 370
  publiczny modułu, 247
kod
  bajtowy Javy, 137
  dynamiczny, 298
  klasy Snippet, 499
  natywny, 76, 119, 243
  Pythona, 51
  sprawdzający, 530
  tworzący wątek, 175
  właściwej implementacji, 530
  wyliczający wiek, 154
  wywołujący, 531
kodowanie UTF-8, 435
koklasa, CoClass, 452
kolejka Queue<T>, 598
kolejki, 598
kolejność ewaluacji argumentów, 440
kolekcja, 268
  BindingList<T>, 593
  BlockingCollection<T>, 599
  Collection<T>, 593
  ConcurrentBag<T>, 600
  ConcurrentDictionary<TKey, TValue>, 600
  ConcurrentQueue<T>, 600
  ConcurrentStack<T>, 600
  ICollection<T>, 489
  IProducerConsumerCollection<T>, 599
  KeyCollection<TKey, TItem>, 593
  Lookup, 576
  ObservableCollection<T>, 593
kolekcje
  generyczne, 588
  konkurencyjne, 598
  o mocnym typie, 73
  o słabym typie, 73
  stałe, 269
  wariancyjne, 588
  zagnieżdżone, 268
kolizja skrótów, 595
kombinacja typów generycznych, 170
komparator, 132
  StringComparer.OrdinalIgnoreCase, 376
kompatybilna sygnatura typu, 179
kompilacja, 125
  do postaci IL, 136
  do postaci natywnej, 136
  drzewa wyrażeń, 291
  kodu, 136
  typu JIT, 53
kompilator
  C#, 53, 500
  JIT, 94, 119
kompletny typ implementujący, 215
komunikacja, 563
komunikat błędu, 549
konfiguracja
  Debug, 561
  Release, 561
koniec bloku kontraktu, 535
konkatenacja, 575
konstruktor bezparametrowy, 38, 105
  this(), 256
konstruktory, 505
  statyczne, 117, 118
  wątku, 186
kontrakty, 524–567
  kodu, 527, 530, 541, 567
  legacyjne, 534, 559
  niespełnione, 559
  testów jednostkowych, 559
kontrawariancja, 75, 128, 179, 454
  interfejsów, 458
  parametrów delegatów, 176
  tablic, 128
  typów generycznych, 176
  typu parametru, 75
kontynuacje
  grupowania, 373, 374
  wyrażenia kwerendowego, 375
  zapytań, 369, 372
kontynuacyjno-przekazujący styl, 221
konwersja, 575
  delegatów, 504
  grupy metod, 43
  jawna, 156
  klasy narzędziowej, 234
  łańcuchów, 101
  metod i kontrawariancji, 177
  niejawna, 156
  pomiędzy typami CLR i dynamic, 497
  referencji, 176, 462
  referencyjna, 455
  typów, 149
  typu dynamicznego, 52
  wyrażenień, 390
  wyrażeń lambda, 292, 345
koszt opakowywania, 121
kowariancja, 73, 128, 179, 454
  IEnumerable<T>, 461
  interfejsów, 457

typów generycznych, 130
typu zwracanego, 75, 178
kształty, 457
kwantyfikator, 583
All, 583
aliasu przestrzeni nazw, 238
Any, 583
Contains, 583
kwerendowe drzewo wyrażeń, 392

L

leniwe filtrowanie, 217
lewostronne złączenie zewnętrzne, 365, 389, 390
liczba ostrzeżeń, 552
liczba synonimów, 450
licznik białych znaków, 529
LIFO, last in, first out, 598
LINQ, Language Integrated Query, 46
251, 336, 351
LINQ dla Obiektów, 280, 296, 336, 408, 574
LINQ dla Rx, 413–420
LINQ dla SQL-a, 49, 340, 379–387
LINQ dla XML-a, 48, 399–407, 507
LINQPad, 56, 321
lista, 590
dwukierunkowa LinkedList<T>, 592
kierunkowa, 592
kształtów, 457
List<T>, 590
łańcuchów, 266
nazw, 516
parametrów metody, 433
parametrów o typie niejawnym, 284
sortowanie, 184, 287
logika boolowska, 159
logowanie zdarzeń, 288

Ł

łańcuchowanie, 328
łapanie wyjątku klasy Exception, 543
łączenie
biblioteki typów, 452
delegatów, 66
metod statycznych, 324
ograniczeń, 108
operacji w łańcuch, 328
PIA, 451
wyrażeń lambda, 296

M

mapowanie obiektowo-relacyjne, 229
MARS, multiple active result sets, 606
maszyna stanowa, 198, 205
maszyna wirtualna, 138
mechanizm
multimetod, 489
refleksji, 117
wnioskowania typów, 308
MEF, Managed Extensibility Framework, 608
metadane, 383
metaobiekt, 518, 520
metaprogramowanie z użyciem wzorców, 137
metoda
Add, 267, 401
AddProperty, 298
Array.CreateInstance, 592
Array.Sort, 328
AsOrdered, 411
AsParallel, 410
Assembly.GetType, 125
Assert, 533
Assume, 533
AsUnordered, 412
base.VirtualMethod, 539
BindInvokeMember, 521
CcrCheck, 221
Clone, 74
Compare, 40, 151, 168
Compile, 291
ContractInvariantMethod, 531
ConvertAll, 100
CopyTo, 313
CountImpl, 489
CountWords, 95
CreateDelegateInstance, 191
CreateEnumerable, 207
CreateQuery, 391, 392, 395
Debug.Assert, 533
Delegate.Combine, 66, 464
Delegate.Remove, 66
Dequeue, 599
Dispose, 123, 207
DontPassInNull, 542
Dump, 440
EndContractBlock, 535
engine.Execute, 481
Enumerable.AsEnumerable, 577
Enumerable.Reverse, 340
Enumerable.Sum, 535

metoda
- Equals, 112, 150–151, 511
- EventHandler, 173
- Exists, 533
- Expression.Call, 521
- Expression.Convert, 521
- FindAll, 190
- ForAll, 533
- GenerateSampleData, 179
- generyczna, 116
- GetDynamicMemberNames, 511, 516
- GetEnumerator, 121, 200, 391, 397
- GetHashCode, 147, 511
- GetInstanceRestriction, 521
- GetKeyFromItem, 593
- GetMetaObject, 494, 511
- GetSampleProducts, 36
- GetValueOrDefault, 147
- GetVariable, 482
- GetViewBetween, 597
- GroupBy, 329
- GroupJoin, 365, 419
- HandleFailure, 544
- int.TryParse, 531
- Interlocked.CompareExchange<T>, 467
- Invariant, 532
- Invoke, 63
- List, 101
- List<T>.ConvertAll, 275
- List<T>.ForEach, 182
- List<T>.Sort, 328
- LogEntity, 233
- Main, 55
- MakeList, 102
- MightReturnNull, 546
- Monitor.Enter, 466
- MoveNext, 121, 208, 216
- MyMethod, 174
- obsługi zdarzenia, 68
- OrderBy, 42, 327
- ParallelEnumerable.Range, 413
- Pop, 598
- Push, 598
- Random.Next, 534
- Range, 322, 416
- ReadFully, 313, 316
- ReferenceCompare, 168
- Report, 539
- Requires, 529, 560
- Reverse, 322, 597
- SaveAs, 51, 447
- Select, 326
- SelectMany, 369, 418
- SetHandled, 544
- SetVariable, 482
- Sort, 41
- StartsWith, 295
- StreamUtil.CopyTo, 319
- StringBuilder.Replace, 324
- Substring, 499, 502
- System.Nullable, 151
- ThenBy, 331
- ToDictionary, 576
- ToLookup, 576
- ToString, 508, 511
- TryGetMember, 515
- Type.GetMethods, 127
- Type.GetType, 125
- Where, 44, 218, 323, 325
- WithCancellation, 412
- WithDegreeOfParallelism, 412
- WithExecutionMode, 412
- WithMergeOptions, 413
- Workbooks.Add, 478

metody
- anonimowe, 43, 83, 180–198
- częściowe, 227, 231
- dynamiczne, 503
- generyczne, 96, 100, 126, 217
- instancji, 500
- inwariantowe, 532
- o tej samej sygnaturze, 431
- obsługi zdarzeń, 174, 288
- osiowe, 404, 407
- pobierania typu obiektów, 126
- porównujące, 151
- porównywania produktów, 169
- przeciążone, 267
- publiczne, 485
- rozszerzające, 42, 86, 312–332, 341, 410, 422
  - klasy Queryable, 395, 398
  - pusta referencja, 319
  - składnia, 315
  - udostępnianie, 319
  - w .NET 3.5, 321
  - wykorzystywanie, 332
  - wykrywanie, 318
  - wywoływanie, 316
  - z argumentami dynamicznymi, 503
- statyczne, 63, 505
  - klasy Array, 592
- TryXXX, 511, 515

wirtualne, 70
wyższego rzędu, 460
z kontraktem, 537
z parametrami opcjonalnymi, 432
ze sprawdzaniem argumentów, 525
zwracające typy referencyjne, 165
Micro Framework, 611
miejsce wywołania, 493, 494
mocny typ zmiennej, 73, 276
model
błędów, 342
DOM, 510
wpychający, 415
moduł, assembly, 86, 125, 240
PIA, 451, 452
produkcyjny, 247
System.Interactive, 414, 419
zaprzyjaźniony, 246
źródłowy, 246
referencyjny kontraktów, 527
modyfikator
?, 152
in, 456
Key, 275
out, 438, 455, 456
private, 236
ref, 438
static, 70, 234
monady, 567
MondrianDrawing, 130
MoreLINQ, 421
MSDN, 588, 606
multimetody, 489

N

nadpisywanie
GetDynamicMemberNames, 516
metod TryXXX, 514
metod, 511
NaN, Not a Number, 144
narzędzia
dla kontraktów kodu, 527
do wizualizacji drzew wyrażeń, 295
narzędzie
analizy automatycznej, 545
analizy statycznej, 533, 547, 552
binary rewriter, 527, 530
cccheck, 527, 545
ccdocgen, 527, 554
ccrefgen, 527, 541
ccrewrite, 527
ILDasm, 155, 182, 498
projektowe, 229, 383
przepisujące, 536–538
Reflector, 155, 182, 211
Snippy, 55, 498
Solution Explorer, 241
static checker, 527
tlbimp, 451
wyliczające wyrażenia, 472
narzut pamięciowy, 120
natywna implementacja, 389
nawiasy
klamrowe, 183
kwadratowe, 124
nadmiarowe, 332
ostre, 96, 211
zwykłe, 183
nazwa
klasy, 526
metody, 521
obiektu, 521
T, 99
niejawne
konwersje, 302, 306, 479
typy tablic, 272
zobowiązania
arytmetyczne, 550
rozmiaru tablicy, 549
wartości niepustych, 548
niekompatybilność C# 1 i C# 2, 179
nienullowalny
typ referencyjny, 563
typ wartościowy, 112
nieokreślona zmienna typu, 302
niezmienność, 147, 443
delegatów, 66
obiektów, 442
właściwości, 275
niszczenie iteratora, 216
notacja kropkowa, 375–378
null, 142
wartość rzeczywista, 436
wartość domyślna, 435
nullowalny typ wartościowy, 112
nullowy operator koalescencyjny, 161, 163, 436
numer ostrzeżenia, 242

O

obiekt
  DynamicXElement, 512, 514
  ExpandoObject, 507
  Rumpelstiltskin, 518
  Table<T>, 391
obiekty
  expando, 507
  obserwowalne, 416
  osadzone, embedded objects, 265
  zagnieżdżone, 266
obliczenia numeryczne, 608
obserwator, 414
obsługa
  błędu, 544
  kodu natywnego, 492
  kowariancji, 129
  liczb zespolonych, 134
  typów dynamicznych, 292
  wyrażeń lambda, 287
  zdarzeń, 177, 288
  zdarzeń w .NET, 177
odbiorca wywołania, 493
odpakowywanie, 81, 147
odpowiedź usługi sieciowej, 314
odpytywanie bazy danych, 386
odwołanie
  do właściwości, 507
  do zmiennej, 188
odwracanie łańcuchów, 56
ograniczenie, 108
  kodu dynamicznego, 503
  konwersji typów, 497
  konwertowania, 292
  numeryczne, 134
  parametrów opcjonalnych, 433
  parametru typu, 106
  typów, 104
    generycznych, 127, 135
    niejawnych, 259
  typu konstruktora, 105
  typu konwersji, 106, 132
  typu referencyjnego, 104
  typu wartościowego, 105
określona zmienna typu, 302
opakowanie wyrażeniem using, 211
opakowywanie, 81, 147
  instancji, 149
  typem referencyjnym, 144
opcja, 551, 559
  Assert on Contract Failure, 543
  Call-site Requires Checking, 542
  Emit Contracts into XML doc file, 554
  Implicit Arithmetic Obligations, 551
  Implicit Array Bounds Obligations, 549
  Implicit Non-null Obligations, 548
  Only Public Surface Contracts, 561
operacje
  akceptujące parametr typu, 131
  dwuargumentowe, 289
  dynamiczne, 499
  jednoargumentowe, 289
  zwracające parametr typu, 131
operator
  !=, 112
  &, 157
  &&, 534
  |, 157
  ==, 112, 157, 578
  as, 161
  AsEnumerable, 576
  AsQueryable, 576
  Cast, 349
  Distinct, 585
  Except, 585
  false, 157
  Intersect, 585
  OfType, 349
  Union, 585
  SelectMany, 406
  Sum, 486
  true, 157
  typeof, 123, 124
operatory
  agregacji, 398, 574
  bazujące na zbiorach, 584
  dwuargumentowe, 157
  filtrujące, 584
  generujące, 579
  generyczne, 486
  grupujące, 580
  jednoargumentowe, 157
  jednoelementowe, 577
  koalescencyjne, 161
  konkatenacji, 575
  konwersji, 575
  kwantyfikatorów, 583
  kwerendowe, 340, 349, 376, 420
    Cast, 349
    OfType, 349

LINQ, 420
logiczne, 160
partycjonujące, 582
projekcji, 582
relacyjne, 157
sortujące, 585
warunkowe, 152, 163, 168
wzniesione, lifted operators, 157
zapytań spłaszczonych, 406
złączeń, 580
optymalizacja, 421
optymalizator zapytań, 388
ORM, Object-Relational Mapping, 229, 382
osadzanie języka w C#, 480
ostrzeżenia kompilatora, 242
ostrzeżenie, 549, 551

P

pamięć podręczna, 494, 495
Parallel Extensions, 220, 222, 408, 567, 607
parametr, 430
Action<T>, 182
decimal?, 45
out, 462
ref, 448
ref typu T, 462
this, 500
TOutput, 276
typu, 96
typu EventArgs, 177
typu T, 146
parametry
dodatkowe typu, 135
formalne, 430
opcjonalne, 46, 427–445
ograniczenia, 433
typów, 104
partycjonowanie, 582
Pex, 567
pętla foreach, 43, 123, 193
PIA, Primary Interop Assembly, 51, 451
plany złączeń, 419
platforma programistyczna, 54
plik
AssemblyInfo.cs, 554
books.xml, 509
HelloWorld.py, 481
PDB, 537
XML, 48
XmlSampleData.cs, 404
pliki
konfiguracyjne, 482
aktywne, 483
pasywne, 483
XAML, 229
PLINQ, 408, 411
płynne interfejsy, fluent interfaces, 331, 332
pobieranie wartości, 507
podpisywanie modułów, 247
podpowiedź IntelliSense, 180, 317
pojedynczy dispatcher, single dispatch, 488
pole, 264
instancji typu, 117
SomeClass.x, 117
statyczne, 117
pomocnicza klasa generyczna, 132
porównania
niestandardowe, 376
referencyjne, 113
strukturalne, 608
z null, 158
porównanie
Comparer<T>.Default, 422
EqualityComparer<T>.Default, 422
porównywanie, 169
kluczy, 576, 580, 595
wartości, 112
poszukiwanie metody rozszerzającej, 318
potok danych, 200, 283
potrójne cudzysłowy, 481
prawdopodobne zakończenie metody, 165
predykaty, 193, 293, 323, 351
prefiks X, 399
prekompilacja XAML, 230
preprocesor, 346
produkt z nieznaną ceną, 45
programowanie
sterowane testami, 556
w trybie REPL, 472
projekcje, 326, 348, 417, 580
projekt
Mono, 472
MoreLINQ, 341
Parallel Extensions, 408
Reactive Extensions dla .NET, 413
projekt Unconstrained Melody, 107
projekty stertowane testami, 558
przechowywanie danych, 277
przechwytywanie
instancji zmiennej, 192
parametrów, 217
zmiennej, 187, 193

przeciążenie metody, 46, 306, 496
  sygnatura, 362
  Create, 431
przeciążenie złożone, 422
przekazywanie
  argumentów przez wartość, 448
  parametrów przez referencję, 80, 448
przepisywanie, 536
przestrzeń nazw
  Microsoft.Office.Interop.Excel, 478
  System.Collections.Concurrent, 598
  System.Collections.Generic, 588
  System.Collections.ObjectModel, 593
  System.Collections.Specialized, 73
  System.Interactive, 414, 419
  System.Linq, 321, 325, 410
  System.Linq.Expressions, 289
  System.Numeric, 608
  System.Xml.Linq, 399
przetwarzanie
  listy, 286
  po stronie serwera, 389
  rekursywne, 403
  równoległe, 568
  sekwencji, 407
przypadki użycia, 230, 270
przypisanie, 78
przyrostek List, 508
pula wątków, thread pool, 190
pusta referencja, 160, 319, 460
puste wartości, 320
pusty typ zwracany, 232

Q

Queue<T>, 598

R

reagowanie dynamiczne, 506
refaktoryzacja, 298
referencja, 76, 80
  do obiektu, 460
  do Scheduler.ThreadPool, 419
  do typu object, 402
  IDisposable, 415
  nullowa, 142
  this, 63, 182
referencyjny moduł kontraktów, 541
refleksja, 123, 138, 299
  metod generycznych, 126
  typów generycznych, 127
reguły, 494
  dotyczące dynamic, 474
  walidujące konwersję, 176
  wnioskowania, 300
rekursywne
  przetwarzanie sekwencji, 403
  wywołanie metody, 509
Relative Extensions, 567
REPL, read-evaluate-print loop, 472
reprezentacja klucza, 359
rozszerzanie
  LINQ dla Obiektów, 420
  zapytań, 408
rozwiązywanie przeciążeń, 299, 443, 501
równoległe LINQ, 408–411
równość, 578
  instancji Nullable<T>, 150
  obiektów, 151
rzucanie wyjątku, 67
rzutowanie, 40, 93, 351, 444

S

schemat
  hierarchii klas, 130
  powiązań .NET 4, 491
sekwencja, 337, 358, 363
  łańcuchów, 348
  wykonania wyrażenia kwerendowego, 339
serializowanie właściwości, 298
setter prywatny, 256
settery, 36, 235
SeuratDrawing, 130
silnik add-inów, 607
silnik DLR, 493
Silverlight, 610
składnia delegatów, 173
skrót, 274, 595
skrót dla parametru, 284
skrypty użytkowników, 484
słaby typ, 73
słownik, 96, 594
  Dictionary<string, int>, 99
  Dictionary<TKey, TValue>, 594
  IDictionary<string, object>, 507
  SortedDictionary<TKey, TValue>, 595
  SortedList<TKey, TValue>, 595
słowo
  ascending, 355
  Attribute, 246
  delegat, 61
  descending, 355

dynamic, 496
dynamiczny, 471
inner, 359
operator, 341
outer, 359
statyczny, 70, 471
User, 383
właściwość, 264
słowo kluczowe
class, 77
const, 563
delegate, 181
dynamic, 52, 474
equals, 360
interface, 77
into, 374
new, 266, 272
return, 184, 283
struct, 77
this, 315
typedef, 133
var, 47, 85, 257, 476
Snippy, 55, 498
Solution Explorer, 241
sortowanie, 40, 167, 327, 354, 412, 458, 585
listy, 184, 287
plików, 184
specyfikacja
C# 4, 57
DLR, 522
specyfikowanie kontraktów, 540
spinanie, 324
spłaszczanie, 418
spłaszczona sekwencja, 406
spłaszczone wyniki, 406
sprawdzanie
argumentów, 421
istnienia delegata, 185
Stack<T>, 598
stan uszkodzenia obiektu, 539
standard
CLI, 105
ECMA, 57
standardowe operatory kwerendowe, 574
statyczny typ dynamic, 86
sterta, 79
stos, 62, 79, 598
Stack<T>, 598
struktura, 79
Nullable<T>, 45
pliku XML, 510
System.Nullable<T>, 145, 151
TIME_ZONE_INFORMATION, 244
strumieniowanie, 323
strumieniowy model obsługi danych, 338
strumień
dodawanie funkcjonalności, 314
kopiowanie, 316
odpowiedzi, 315
sieciowy, 216
subscribe, 415
subskrybowanie zdarzeń, 185
subskrypcja zdarzeń przycisku, 173
suma kontrolna pliku, 243
suma logiczna, 159
sygnatury metod, 98
sygnatura przeciążenia metody, 362
symbol preprocesora, 554
system typów, 69, 75, 97
systemy ORM, 425
szkielet klasy DynamicXElement, 511

Ś

śledzenie błędów, 329
środowisko
.NET, 52, 487, 524, 567, 600, 606
CLR, 254, 274, 462
desktopowe, 604
IDE, 93, 139, 227, 516
IronPythona, 480
Parallel Extensions, 220
REPL, 516
serializujące, 230
uruchomieniowe, 608, 609
wykonawcze .NET, 53

T

tabela
Defect, 383
DefectUser, 383
tablica, 73, 77, 591
ArrayList, 120
List, 120
tablice
jednowymiarowe, 591
kowariancyjne, 73, 129
kwadratowe, 592
o typie niejawnym, 254, 270
parametrów, 433
prawdy, 159, 160
typu Stream[], 591
typów referencyjnych, 591

technologia
ASP.NET, 607
ASP.NET MVC, 594
COM, 50, 427, 446, 479
LINQ, 251
Parallel Extensions, 222
Remoting, 607
test
jednostkowy, 421, 557
wydajnościowy, 161
testowanie, 559
klasy DynamicXElement, 516
niejawnych zobowiązań, 548
tłumaczenie wyrażenia kwerendowego, 348
transformacja, 338, 340
Select, 338
Where, 338
translacja, 358
kompilatora, 344, 345
trywialna właściwość, 255
T-SQL, 383
tworzenie
aliasu, 238
binderów, 498
delegatów, 43, 278
dokumentu Worda, 446
drzewa modelu DOM, 507
drzewa wyrażeń z lambdy, 294
elementów, 401–404
instancji delegata, 62, 83, 189, 282, 285
instancji typu ScriptEngine, 480
klas encji, 384
klasy narzędziowej, 330
kolekcji, 266
metody przepisującej, 544
miejsc wywołań, 498
nietrywialnego obiektu, 268
obiektów, 262
obiektu budowniczego, 270
rysunku, 130
schematu bazy danych, 383
tablic, 266
testów jednostkowych, 269
typu częściowego, 227
wątku, 186
wielokrotne miejsc wywołań, 499
zagnieżdżonej subskrypcji, 417
zapytania, 351
typ
_ContractsRuntime, 537
ArrayList, 102
bool, 519
bool?, 159, 169
Complex, 134
Configuration, 483
Dictionary, 95, 111
dynamic, 51, 87, 475, 479, 505
DynamicMethod, 491
Expression, 491
Func, 325
generyczny sparametryzowany, 273
Guid, 100
Hashtable, 96
IComparer<Circle>, 132
IComparer<IShape>, 132
IDisposable[], 271
IEnumerable<string>, 216
IEnumerable<T>, 222, 404
InstanceMethods, 63
IObservable<out T>, 414
IObserver<in T>, 414
Lazy<T>, 608
List<Circle>, 132
List<T>, 42, 197, 266
MemoryStream, 179
Nullable<T>, 147
object, 519
Pair, 114
Product, 35
ProductNameComparer, 41
ReadOnlyCollection<T>, 269
Rumpelstiltskin, 519
StaticMethods, 63
Stream, 313
string[], 259
StringProcessor, 61, 63
System.Delegate, 61, 175
System.Type, 125
timeSpan, 487
Tuple, 608
void, 61, 183, 519
XAttribute, 400
XContainer, 400
XDocument, 401
XElement, 399, 401
XName, 399
XNamespace, 399
XNode, 400
XObject, 400
XText, 400
typy
aliasów, 238
anonimowe, 85, 254, 272–277, 326, 357
elementy, 274

częściowe, 226–229
danych, 400
delegatowe, 61, 77, 325, 459
delegatów generycznych, 84
dynamiczne, 50, 71, 427, 471–477, 484, 504
dynamiczne w C# 4, 87
enumeracyjne, 332
generyczne, 36, 85–92, 121–139
.NET, 138
Javy, 137
niezwiązane, 97
jawne, 71, 261
kolekcji, 73
kontraktów, 560
mocne, 74
niejawne, 71, 259–261, 271
niezmienne, 146
nullowalne, 88, 142-165, 170, 254
o tej samej nazwie, 240
otwarte, 97, 125
potomne, 78
projekcji, 370
referencyjne, 76, 77, 82
rozszerzane, 315
skonstruowane, 97, 124
słabe, 73
statyczne, 70, 478
tablicowe, 488
wariantowe, 453
wartościowe, 76, 82, 87
węzłów, 290
zamknięte, 97, 105, 117
zmiennej, 261
zwracane, 178
zwracane funkcji anonimowej, 302

U

udostępnianie elementów, 245
ukrywanie kontraktu, 543
unikanie przeciążeń, 444
uporządkowanie, 355
uproszczenia iteratorów, 204, 215, 217, 221
uproszczona inicjalizacja, 37
usługa P/Invoke, 243
ustawianie właściwości, 263–265
usunięcie nawiasów, 284
usuwanie delegatów, 66

V

VB.NET, 54
Visual Basic, 50, 275, 430, 446
Visual Studio, 56, 229, 241, 273, 384, 527
Visual Studio 2010 Premium, 545
Visual Studio 2010 Ultimate, 545
void, 232

W

walidacja, 56, 526, 562
argumentów, 332, 524
metody, 533
wariancja, 456, 460, 462–465
interfejsów, 455
parametrów typu, 461
typów generycznych, 176, 453, 468
w delegatach, 458
warstwa prezentacyjna Silverlight, 610
warstwy pośredniczące, 173
wartości
domyślne, 44, 46, 111, 430
domyślne null, 112
dynamiczne, 496
magiczne, 144, 435
nullowe, 143
puste, 152, 155
typu Nullable<int>, 146
typu referencyjnego, 81, 142
typu wartościowego, 81, 142
wartościowanie
leniwe, lazy evaluation, 323
zachłanne, eager evaluation, 323
wartościowy typ nullowalny, 152
wartość
DateTime.MinValue, 144
DBNull.Value, 144
int.MinValue, 551
int?, 153
ListControl.DisplayMember, 484
warunek inwariancji, 539
warunki
końcowe, 530, 531, 536
wstępne, 529, 536, 542
WCF, Windows Communication Foundation, 607
wektor, 591
wersja
debug, 557
release, 557

wersje bazowe, 552
wersje środowiska .NET, 604
wersjonowanie, 434, 453
weryfikowanie kontraktów, 546
wewnętrzna sekwencja, 359
wewnętrzne równozłączenie, 388
WF, Windows Workflow Foundation, 607
wiązanie, binding, 471
widok dynamiczny, dynamic view, 517
widok właściwości dynamicznych, 517
wielokropek, 55
wielokrotne
  iterowanie, 422
  uporządkowania, 355
  wyniki wyszukiwania, 405
wieloznaczniki, 138
Windows CardSpace, 607
wizualizacja drzewa wyrażeń, 295
wizualizator, 295
własne metody przepisujące, 544
własny operator, 421
właściwości
  automatyczne, 256
  domyślne, 449
  encji, 384
  implementowane automatycznie, 37, 254, 255
  kontraktów kodu, 528
  obiektu osadzonego, 265
  rozszerzające, 332
  sparametryzowane, 449
  tylko do odczytu, 38
właściwość
  ActiveSheet, 51, 478
  Comparer<T>.Default, 167
  Count, 488
  Current, 212
  Embed Interop Types, 477
  Friends, 268
  HasValue, 45
  Key, 370
  NodeType, 290
  Subscriptions, 364
  Type, 290
  Value, 478, 521
wnioskowanie typów, 103, 110, 300, 308, 349
  algorytm, 303
  dwufazowe, 302
  wielostopniowe, 305
wnioskowanie typu
  generycznego, 502
  w czasie wykonania, 485
Word, 446
WPF, Windows Presentation Foundation, 593, 606
wpychanie danych, 413
wskaźnik funkcji, 60
współprogramy, 223
wyciek pamięci, 63
wydajność, 324
wyjątek
  ArgumentNullException, 526, 556
  ArrayTypeMismatchException, 73
  ContractException, 529, 543
  InvalidCastException, 482
  InvalidOperationException, 146, 598
  NotSupportedException, 591
  NullReferenceException, 149, 319
wykonanie
  natychmiastowe, 340
  odroczone, 322
  opóźnione, 338
wykreślanie typu, type erasure, 138
wyłączenie ostrzeżenia, 242
wyniki analizy statycznej, 547
wypełnianie tablicy, 273
wyrażenia
  blokujące, 466
  filtrujące, 351
  grupowanie, 370
  kwerendowe, 47, 336–352, 377–379, 577
    zdegenerowane, 353
    group by, 417
  lambda, 84, 280, 292, 308, 504
  projekcja, 370
  wartości domyślnej, 111
  wyboru klucza, 359
wyrażenie
  #pragma, 241
  Func<object>, 464
  join, 50
  lambda, 41, 282–286
  using, 53, 216, 237
  yield, 204
  yield break, 209
  yield return, 205, 208
wyszukiwanie
  elementów, 43
  przeciążeń, 308
wyświetlanie łańcucha, 291
wywołania równoległe, 220
wywołanie
  CreateInstance<int>(), 106
  Dispose, 211, 237

GetGenericTypeDefinition, 125
instancji delegata, 63
konstruktora bezparametrowego, 232
metody rozszerzającej, 317
operatora Cast, 350
OrderBy, 355
OrderByDescending, 355
Range, 478
string.Substring, 502
Subscribe, 415
ThenBy, 355
funkcji, 482
wzorce C++, 136, 137
wzorzec
czynnościowy, 200
inicjalizacyjny, 442
iteratora, 200, 202, 222
projektowy budowniczego, 270
TryXXX, 111, 166
wyrażeń kwerendowych, 345

X

XAML, Extensible Application Markup Language, 229

Z

zakładka Code Contracts, 528
zakres, 550
zakres zmiennych, 195
zamortyzowana złożoność, 590, 595
zaprzyjaźnione moduły, 246
zapytania, 48
na pojedynczych węzłach, 404
zapytanie jednowątkowe, 410
zapytanie równoległe, 410
zasada podstawiania Liskov, 538
zasięg, 481
zasięg globalny, 481
zasięg zmiennych, 195
zbiór, 595
HashSet<T>, 597
kolekcji alternatywnych, 601
Mandelbrota, 408, 411
SortedSet<T>, 597
zdarzenia, 67
.NET, 420
w formie pól, 68, 467
wewnątrz klasy deklarującej, 467
zdarzenie
Click, 174
Contract.ContractFailed, 543
ContractFailed, 544
MouseClick, 174
zdegenerowane wyrażenia kwerendowe, 352
zestaw przeciążeń, 431
zewnętrzna sekwencja, 359
zimny obiekt obserwowalny, cold observable, 416
złączenia, 355, 580
jawne, 388
niejawne, 390
złączenie, 50, 355, 580
grupowe, 363, 365
jawne, 388
krzyżowe, 366
niejawne, 390
wewnętrzne, inner join, 359–362, 389
składnia, 360
zewnętrzne, outer join, 359
złożoność, 595, 596
zmiana nazwy parametru, 445
zmienna
args, 259
captured, 189
counter, 191
zmienne
dynamiczne, 52
inicjalizowane przez konstruktor, 261
lokalne, 192
deklarowane niejawnie, 85
instancjonowane, 192
o typie niejawnym, 47, 254, 257
metod anonimowych, 187
przechwycone, 186–192, 196
publiczne, 255
typu, 302
typu delegatowego, 43
zakresu, 343, 347–351
zewnętrzne, 187, 188
znajdowanie błędów, 129
znaki UTF-16, 56
znaki
gwiazdka (*), 358
podwójny dwukropek (::), 238
zapytania (??), 161
zniknięcie wywołania Select, 353
zobowiązania niejawne, 548
zrównoleglanie zapytań, 409
zwalnianie blokady, 466
zwalnianie iteratorów, 422

zwracanie
  pustej referencji, 547
  sekwencji, 407
  wartości, 183, 547

Ź

źle sformułowany kontrakt, malformed contract, 530

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION

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

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

http://program-partnerski.helion.pl

C# OD PODSZEWKI

WYDANIE 2.

Obecnie szczególnie cenione są te języki programowania, które pozwalają na błyskawiczne osiągnięcie oczekiwanych efektów. Dodatkowo absolutnie niezbędne jest zachowanie możliwości uruchamiania raz napisanej aplikacji na różnych platformach bez konieczności jej przepisywania. Nikt nie ma na to czasu! C# to nowoczesny język, który zdobył uznanie programistów na całym świecie, ponieważ spełnia nawet najbardziej wyśrubowane wymagania!

Ten bezcenny podręcznik zaprowadzi Cię w najskrytsze zakątki języka C#. Autorzy założyli, że znasz jego podstawy — to pozwoliło im skupić uwagę na niuansach, ciekawostkach oraz subtelnych szczegółach. W trakcie lektury zrozumiesz, co się dzieje w czeluściach C#, a dzięki temu unikniesz ukrytych pułapek. Książka ta jest obowiązkową pozycją każdego programisty C#. Bez niej najciekawsze funkcje C# wciąż będą Ci obce!

Dowiedz się:

- do czego przydają się typy generyczne
- jak zaimplementować iteratory
- jak zoptymalizować skompilowany kod
- w czym może pomóc język LINQ
- jak testować kod

Fascynująca podróż w głąb C#!

sięgnij po WIĘCEJ

KOD KORZYŚCI

helion.pl
księgarnia internetowa

Nr katalogowy: 8672

Księgarnia internetowa:
http://helion.pl

Zamówienia telefoniczne:
0 801 339900
0 601 339900

Helion

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

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

ISBN 978-83-246-3921-2

9 788324 639212

Cena: 99,00 zł

Informatyka w najlepszym wydaniu