O'REILLY

C# 9.0 w pigułce

Joseph Albahari

Tytuł oryginału: C# 9.0 in a Nutshell: The Definitive Reference

Tłumaczenie: Łukasz Piwko, Robert Górczyński, Jakub Hubisz

ISBN: 978-83-283-8198-8

© 2021 Helion S.A.
Authorized Polish translation of the English edition of C# 9.0 in a Nutshell
ISBN 9781098100964 © 2021 Joseph Albahari

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

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

Helion S.A.
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/c9wpig
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

- Kup książkę
- Poleć książkę
- Oceń książkę

- Księgarnia internetowa
- Lubię to! » Nasza społeczność

Spis treści

Wstęp **11**

1. Wprowadzenie do C# i .NET **17**
Obiektowość 17
Bezpieczeństwo typów 18
Zarządzanie pamięcią 19
Platformy 19
Historia C# w pigułce 24

2. Podstawy języka C# **39**
Pierwszy program w języku C# 39
Składnia 42
Podstawy typów 44
Typy liczbowe 54
Typ logiczny i operatory 63
Łańcuchy znaków i pojedyncze znaki 65
Tablice 67
Zmienne i parametry 72
Wyrażenia i operatory 83
Operatory null 87
Instrukcje 88
Przestrzenie nazw 99

3. Tworzenie typów w języku C# **105**
Klasy 105
Dziedziczenie 124
Typ object 134
Struktury 138

Kup ksi k

Pole ksi k

Modyfikatory dostępu 140
Interfejsy 142
Wyliczenia 148
Typy zagnieżdżone 152
Typy generyczne 153

4. Zaawansowane elementy języka C# ... 166
Delegaty 166
Zdarzenia 174
Wyrażenia lambda 180
Metody anonimowe 185
Instrukcje try i wyjątki 186
Wyliczenia i iteratory 194
Typy wartościowe dopuszczające wartość null 199
Typy referencyjne dopuszczające wartość null 205
Metody rozszerzające 207
Typy anonimowe 210
Krotki 211
Rekordy (C# 9) 214
Wzorce 224
Atrybuty 229
Atrybuty informacji wywołującego 231
Wiązanie dynamiczne 232
Przeciążanie operatorów 239
Niebezpieczny kod i wskaźniki 243
Dyrektywy preprocesora 248
Dokumentacja XML 250

5. Ogólny zarys platformy ... 255
.NET Standard 255
Wersje środowiska i C# 258
Zestawy referencyjne 258
CLR i BCL 259
Warstwy aplikacji 263

6. Podstawowe wiadomości o platformie .NET ... 267
Obsługa łańcuchów i tekstu 267
Data i godzina 280
Daty i strefy czasowe 288

Kup ksi k Pole ksi k

Formatowanie i parsowanie obiektów DateTime 292
Standardowe łańcuchy formatu i flagi parsowania 298
Inne mechanizmy konwersji 304
Globalizacja 308
Praca z liczbami 309
Wyliczenia 314
Struktura Guid 317
Porównywanie 318
Określanie kolejności 328
Klasy pomocnicze 331

7. Kolekcje ... 336
Przeliczalność 336
Interfejsy ICollection i IList 343
Klasa Array 346
Listy, kolejki, stosy i zbiory 354
Słowniki 363
Kolekcje i pośredniki z możliwością dostosowywania 369
Niezmienne kolekcje 374
Dołączanie protokołów równości i porządkowania 378

8. Zapytania LINQ ... 385
Podstawy 385
Składnia płynna 387
Wyrażenia zapytań 393
Wykonywanie opóźnione 397
Podzapytania 403
Tworzenie zapytań złożonych 406
Strategie projekcji 410
Zapytania interpretowane 412
EF Core 418
Budowanie wyrażeń zapytań 429

9. Operatory LINQ ... 434
Informacje ogólne 435
Filtrowanie 438
Projekcja 442
Łączenie 453
Porządkowanie 461

Kup ksi k Pole ksi k

Grupowanie 464
Operatory zbiorów 467
Metody konwersji 468
Operatory elementów 471
Metody agregacyjne 473
Kwantyfikatory 477
Metody generujące 478

10. LINQ to XML ... 480
Przegląd architektury 480
Informacje ogólne o X-DOM 481
Tworzenie drzewa X-DOM 484
Nawigowanie i wysyłanie zapytań 487
Modyfikowanie drzewa X-DOM 492
Praca z wartościami 495
Dokumenty i deklaracje 497
Nazwy i przestrzenie nazw 501
Adnotacje 506
Projekcja do X-DOM 507

11. Inne technologie XML i JSON ... 511
Klasa XmlReader 511
Klasa XmlWriter 519
Typowe zastosowania klas XmlReader i XmlWriter 521
Praca z formatem JSON 526

12. Zwalnianie zasobów i mechanizm usuwania nieużytków ... 533
IDisposable, Dispose i Close 533
Automatyczne usuwanie nieużytków 539
Finalizatory 541
Jak działa mechanizm usuwania nieużytków? 546
Wycieki pamięci zarządzanej 553
Słabe odwołania 556

13. Diagnostyka ... 561
Kompilacja warunkowa 561
Debugowanie i klasy monitorowania 565
Integracja z debuggerem 568
Procesy i wątki procesów 569
Klasy StackTrace i StackFrame 570

Kup ksi k Pole ksi k

Dziennik zdarzeń Windows 572
Liczniki wydajności 574
Klasa Stopwatch 578
Międzyplatformowe narzędzia diagnostyczne 579

14. Współbieżność i asynchroniczność ... 583
Wprowadzenie 583
Wątki 584
Zadania 600
Reguły asynchroniczności 609
Funkcje asynchroniczne w języku C# 614
Wzorce asynchroniczności 633
Przestarzałe wzorce 641

15. Strumienie i wejście-wyjście ... 645
Architektura strumienia 645
Użycie strumieni 647
Adapter strumienia 661
Kompresja strumienia 669
Praca z plikami w postaci archiwum ZIP 672
Operacje na plikach i katalogach 673
Plikowe operacje wejścia-wyjścia w UWP 683
Bezpieczeństwo systemu operacyjnego 687
Mapowanie plików w pamięci 690

16. Sieć ... 694
Architektura sieci 694
Adresy i porty 697
Adresy URI 698
Klasy po stronie klienta 700
Praca z HTTP 713
Tworzenie serwera HTTP 717
Użycie FTP 719
Użycie DNS 721
Wysyłanie poczty elektronicznej za pomocą SmtpClient 722
Użycie TCP 723
Otrzymywanie poczty elektronicznej POP3 za pomocą TCP 726
TCP w UWP 728

Kup ksi k Pole ksi k

17. Zestawy ... 730
Co znajduje się w zestawie? 730
Silne nazwy i podpisywanie zestawu 734
Nazwy zestawów 735
Technologia Authenticode 738
Zasoby i zestawy satelickie 740
Ładowanie, znajdowanie i izolowanie zestawów 748

18. Refleksja i metadane ... 768
Refleksja i aktywacja typów 769
Refleksja i wywoływanie składowych 775
Refleksja dla zestawów 787
Praca z atrybutami 788
Generowanie dynamicznego kodu 793
Emitowanie zestawów i typów 800
Emitowanie składowych typów 802
Emitowanie generycznych typów i klas 808
Kłopotliwe cele emisji 810
Parsowanie IL 813

19. Programowanie dynamiczne ... 818
Dynamiczny system wykonawczy języka 818
Unifikacja typów liczbowych 820
Dynamiczne wybieranie przeciążonych składowych 821
Implementowanie obiektów dynamicznych 827
Współpraca z językami dynamicznymi 830

20. Kryptografia ... 832
Informacje ogólne 832
Windows Data Protection 832
Obliczanie skrótów 834
Szyfrowanie symetryczne 835
Szyfrowanie kluczem publicznym i podpisywanie 840

21. Zaawansowane techniki wielowątkowości ... 845
Przegląd technik synchronizacji 846
Blokowanie wykluczające 846
Blokady i bezpieczeństwo ze względu na wątki 854
Blokowanie bez wykluczania 859
Sygnalizacja przy użyciu uchwytów zdarzeń oczekiwania 866
Klasa Barrier 873

Kup ksi k Pole ksi k

Leniwa inicjalizacja 874
Pamięć lokalna wątku 877
Zegary 880

22. Programowanie równoległe 884
Dlaczego PFX? 885
PLINQ 888
Klasa Parallel 900
Równoległe wykonywanie zadań 906
Klasa AggregateException 916
Kolekcje współbieżne 918
Klasa BlockingCollection<T> 921

23. Struktury Span<T> i Memory<T> 925
Struktura Span i plasterkowanie 926
Struktura Memory<T> 929
Enumeratory działające tylko do przodu 930
Praca z pamięcią alokowaną na stosie i niezarządzaną 932

24. Współdziałanie macierzyste i poprzez COM 934
Odwołania do natywnych bibliotek DLL 934
Szeregowanie typów i parametrów 935
Wywołania zwrotne z kodu niezarządzanego 939
Symulowanie unii C 942
Pamięć współdzielona 943
Mapowanie struktury na pamięć niezarządzaną 945
Współpraca COM 949
Wywołanie komponentu COM z C# 951
Osadzanie typów współpracujących 954
Udostępnianie obiektów C# COM 955

25. Wyrażenia regularne 957
Podstawy wyrażeń regularnych 957
Kwantyfikatory 962
Asercje o zerowej wielkości 963
Grupy 966
Zastępowanie i dzielenie tekstu 967
Receptury wyrażeń regularnych 969
Leksykon języka wyrażeń regularnych 972

Skorowidz 977

Kup ksi k   Pole ksi k

4

Zaawansowane elementy języka C#

W tym rozdziale opisujemy zaawansowane zagadnienia dotyczące języka C#, stanowiące rozszerzenie pojęć opisanych w rozdziałach 2. i 3. Cztery pierwsze sekcje należy przeczytać po kolei, natomiast dalsze można czytać w dowolnej kolejności.

Delegaty

Delegat to obiekt „wiedzący", jak wywołać metodę.

Typ delegacyjny definiuje rodzaj metody, jaki mogą wywoływać *egzemplarze delegatu*. Mówiąc dokładniej: określa *typ zwrotny* metody i *typy jej parametrów*. Poniżej znajduje się definicja typu delegacyjnego o nazwie `Transformer`:

```
delegate int Transformer (int x);
```

Typ `Transformer` jest zgodny z każdą metodą o typie zwrotnym `int` przyjmującą jeden parametr typu `int`, np. tą:

```
int Square (int x) { return x * x; }
```

Można to też zapisać zwięźlej:

```
int Square (int x) => x * x;
```

Przypisanie metody do zmiennej typu delegacyjnego powoduje utworzenie *egzemplarza* delegatu:

```
Transformer t = Square;
```

Egzemplarz ten można wywoływać tak jak zwykłą metodę:

```
int answer = t(3); // wynik to 9
```

Poniżej przedstawiamy kompletny przykład:

```
Transformer t = Square;        // utworzenie egzemplarza delegatu
int result = t(3);             // wywołanie delegatu
Console.WriteLine (result);    // 9

int Square (int x) => x * x;
delegate int Transformer (int x); // deklaracja typu delegacyjnego
```


Kup ksi k Pole ksi k

Egzemplarz delegatu dosłownie pełni funkcję przedstawiciela wywołującego, który wywołuje delegat, aby za jego pośrednictwem została wywołana metoda docelowa. Taki pośrednik eliminuje zależność wywołującego od metody docelowej.

Instrukcja:

```
Transformer t = Square;
```

jest skróconą formą zapisu instrukcji:

```
Transformer t = new Transformer (Square);
```

Tak naprawdę odwołanie do metody `Square` bez nawiasu i argumentów oznacza, że określamy *grupę metod*. Jeśli metoda jest przeciążona, C# wybierze odpowiednią wersję na podstawie sygnatury delegatu, do którego jest przypisana.

Wyrażenie:

```
t(3)
```

jest skróconą formą zapisu:

```
t.Invoke(3)
```

Delegat jest podobny do **wywołania zwrotnego**. Wywołanie zwrotne to ogólne pojęcie obejmujące m.in. takie konstrukcje jak wskaźniki do funkcji w języku C.

Pisanie metod wtyczek przy użyciu delegatów

Metody do zmiennych delegacyjnych są przypisywane w czasie działania programu. Fakt ten można wykorzystać do pisania metod wtyczek (ang. *plug-in methods*). W przykładzie, który przedstawiamy poniżej, mamy metodę pomocniczą o nazwie `Transform` przekształcającą wszystkie elementy w tablicy liczb całkowitych. Metoda ta ma jeden parametr delegacyjny służący do określania wtyczki przekształcającej:

```
int[] values = { 1, 2, 3 };
Transform (values, Square); // podłączenie metody Square

foreach (int i in values)
Console.Write (i + " "); // 1 4 9

void Transform (int[] values, Transformer t)
{
    for (int i = 0; i < values.Length; i++)
    values[i] = t (values[i]);
}

int Square (int x) => x * x;
int Cube (int x) => x * x * x;

delegate int Transformer (int x);
```


Kup ksi k Pole ksi k

Możemy zmienić transformację przez zmianę `Square` na `Cube` w drugim wierszu kodu.

Nasza metoda `Transform` to funkcja *wyższego rzędu*, ponieważ jako argument pobiera inną funkcję. (Metoda *zwracająca* delegat również byłaby funkcją wyższego rzędu).

Docelowe metody egzemplarzowe i statyczne

Docelowa metoda delegatu może być lokalna, statyczna lub egzemplarzowa. Poniżej znajduje się przykład statycznej metody docelowej:

```
Transformer t = Test.Square;
Console.WriteLine (t(10)); // 100

class Test { public static int Square (int x) => x * x; }

delegate int Transformer (int x);
```

Poniżej znajduje się przykładowa metoda docelowa egzemplarza:

```
Test test = new Test();
Transformer t = test.Square;
Console.WriteLine (t(10)); // 100

class Test { public int Square (int x) => x * x; }

delegate int Transformer (int x);
```

Kiedy metoda egzemplarza zostaje przypisana do obiektu delegatu, obiekt ten przechowuje nie tylko referencję do tej metody, ale również do *egzemplarza*, do którego ta metoda należy. Własność `Target` klasy `System.Delegate` reprezentuje ten egzemplarz (i będzie miała wartość `null` w przypadku delegatu odwołującego się do metody statycznej). Oto przykład:

```
MyReporter r = new MyReporter();
r.Prefix = "%Complete: ";
ProgressReporter p = r.ReportProgress;
p(99); // %Complete: 99
Console.WriteLine (p.Target == r); // prawda
Console.WriteLine (p.Method); // Void ReportProgress(Int32)
r.Prefix = "";
p(99); // 99

public delegate void ProgressReporter (int percentComplete);

class MyReporter
{
    public string Prefix = "";
    public void ReportProgress (int percentComplete)
    => Console.WriteLine (Prefix + percentComplete);
}
```

Ponieważ egzemplarz jest przechowywany we własności `Target` delegatu, będzie on istniał przynajmniej tak długo jak ten delegat.

Kup ksi k Pole ksi k

Delegaty multiemisji

Wszystkie egzemplarze delegatów mają zdolność **multiemisji** (ang. *multicast*). Oznacza to, że egzemplarz delegatu może się odnosić nie tylko do jednej metody, ale do całej listy metod. Do łączenia egzemplarzy delegatów służą operatory + i +=. Na przykład:

```
JakiśDelegat d = JakaśMetoda1;
d += JakaśMetoda2;
```

Drugi wiersz tego kodu jest równoznaczny z poniższym:

```
d = d + JakaśMetoda2;
```

Wywołanie d spowoduje wywołanie metod JakaśMetoda1 i JakaśMetoda2. Delegaty są wywoływane w kolejności dodania.

Operatory - i -= usuwają delegat podany jako prawy argument z delegatu podanego jako lewy argument. Na przykład:

```
d -= JakaśMetoda1;
```

Teraz wywołanie d spowoduje wywołanie tylko metody JakaśMetoda2.

Operatorów + i += można też używać ze zmiennymi delegacyjnymi o wartości null. Taka operacja jest równoznaczna z przypisaniem zmiennej nowej wartości:

```
JakiśDelegat d = null;
d += JakaśMetoda1; // równoznaczne (gdy d ma wartość null) z d = JakaśMetoda1;
```

Analogicznie zastosowanie operatora -= do zmiennej z jedną metodą docelową jest równoważne z przypisaniem do tej zmiennej null.

Delegaty są **niezmienne**, więc użycie operatorów += i -= w rzeczywistości oznacza utworzenie *nowego* egzemplarza delegatu i przypisanie go do istniejącej zmiennej.

Jeśli delegat multiemisji ma inny typ zwrotny niż void, wywołujący otrzymuje wartość zwrotną od ostatniej wywołanej metody. Wywołane zostają także wszystkie poprzednie metody, ale ich wartości zwrotne są ignorowane. W większości przypadków użycia delegatów multiemisji metody te mają typ zwrotny void, więc jest to mało znaczący drobiazg.

Wszystkie typy delegacyjne niejawnie pochodzą od typu System.MulticastDelegate, który dziedziczy po System.Delegate. Kompilator kompiluje wszystkie operacje +, +=, - i -= wykonywane na delegatach do postaci statycznych metod Combine i Remove klasy System.Delegate.

Przykład użycia delegatu multiemisji

Powiedzmy, że napisaliśmy metodę, której wykonywanie zajmuje dużo czasu. Metoda ta mogłaby regularnie informować wywołującego o postępie pracy za pomocą wywoływania delegatu. W poniższym przykładzie metoda HardWork przyjmuje parametr delegacyjny ProgressReporter, który wywołuje w celu pokazania, ile pracy już wykonała:

Kup ksi k Pole ksi k

```
public delegate void ProgressReporter (int percentComplete);

public class Util
{
  public static void HardWork (ProgressReporter p)
  {
    for (int i = 0; i < 10; i++)
    {
      p (i * 10);                           // wywołanie delegatu
      System.Threading.Thread.Sleep (100);   // symulacja ciężkiej pracy
    }
  }
}
```

W celu monitorowania postępu pracy metoda `Main` tworzy egzemplarz delegatu multiemisji `p`, tak że postęp jest monitorowany przez dwie niezależne od siebie metody:

```
ProgressReporter p = WriteProgressToConsole;
p += WriteProgressToFile;
Util.HardWork (p);


void WriteProgressToConsole (int percentComplete)
  => Console.WriteLine (percentComplete);

void WriteProgressToFile (int percentComplete)
  => System.IO.File.WriteAllText ("progress.txt",
                                  percentComplete.ToString());
```

Generyczne typy delegacyjne

Typ delegacyjny może zawierać generyczne parametry typów. Na przykład:

```
public delegate T Transformer<T> (T arg);
```

Przy użyciu tej definicji możemy napisać uogólnioną metodę pomocniczą `Transform` działającą z każdym typem:

```
int[] values = { 1, 2, 3 };
Util.Transform (values, Square); // podpięcie do Square
foreach (int i in values)
  Console.Write (i + " ");         // 1 4 9

int Square (int x) => x * x;

public class Util
{
    public static void Transform<T> (T[] values, Transformer<T> t)
    {
        for (int i = 0; i < values.Length; i++)
        values[i] = t (values[i]);
    }
}
```


Kup książkę Poleć książkę

Delegaty Func i Action

Przy użyciu delegatów generycznych można pisać typy delegacyjne o tak dużym stopniu ogólności, że działają z metodami o każdym typie zwrotnym i dowolnej (sensownej) liczbie argumentów. Są to delegaty `Func` i `Action` zdefiniowane w przestrzeni nazw `System` (adnotacje `in` i `out` określają *wariancję*, o której jest mowa nieco dalej):

```
delegate TResult Func <out TResult>             ();
delegate TResult Func <in T, out TResult>       (T arg);
delegate TResult Func <in T1, in T2, out TResult> (T1 arg1, T2 arg2);
... itd. do T16

delegate void Action                 ();
delegate void Action <in T>          (T arg);
delegate void Action <in T1, in T2> (T1 arg1, T2 arg2);
... itd. do T16
```

Delegaty te są bardzo ogólne. Delegat `Transformer` z naszego poprzedniego przykładu można zastąpić delegatem `Func` przyjmującym jeden argument typu `T` i zwracającym wartość tego samego typu:

```
public static void Transform<T> (T[] values, Func<T,T> transformer)
{
  for (int i = 0; i < values.Length; i++)
    values[i] = transformer (values[i]);
}
```

Jedynymi praktycznymi zastosowaniami niepokrytymi przez te delegaty są `ref`/`out` i parametry wskaźnikowe.

Delegaty `Func` i `Action` nie istnieją w języku C# od początku (wcześniej nie było typów generycznych). To właśnie z tego powodu w .NET tak powszechne są dostosowywane do konkretnych potrzeb typy delegacyjne zamiast delegatów `Func` i `Action`.

Delegaty a interfejsy

Każdy problem dający się rozwiązać za pomocą delegatu można też rozwiązać przy użyciu interfejsu. Możemy m.in. przepisać nasz przykład z użyciem interfejsu o nazwie `ITransformer` zamiast delegatu:

```
int[] values = { 1, 2, 3 };
Util.TransformAll (values, new Squarer());
foreach (int i in values)
  Console.WriteLine (i);

public interface ITransformer
{
  int Transform (int x);
}

public class Util
{
  public static void TransformAll (int[] values, ITransformer t)
  {
```


Kup ksi k   Pole ksi k

```
      for (int i = 0; i < values.Length; i++)
        values[i] = t.Transform (values[i]);
    }
  }

  class Squarer : ITransformer
  {
    public int Transform (int x) => x * x;
  }
  ...
```

Delegat może być lepszym rozwiązaniem od interfejsu, jeśli spełniony jest przynajmniej jeden z poniższych warunków:

- interfejs zawiera definicję tylko jednej metody;
- potrzebna jest możliwość skorzystania z multiemisji;
- subskrybent musi zaimplementować interfejs wiele razy.

W przykładzie z interfejsem `ITransformer` nie musimy korzystać z multiemisji, ale interfejs definiuje tylko jedną metodę. Ponadto nasz subskrybent może być zmuszony do wielokrotnego zaimplementowania interfejsu `ITransformer` w celu obsługi różnych rodzajów przekształceń, np. podnoszenia do kwadratu albo sześcianu. Interfejs zmusza nas do napisania osobnego typu dla każdej transformacji, ponieważ `Test` może implementować ten interfejs tylko raz. To dość kłopotliwe:

```
int[] values = { 1, 2, 3 };
Util.TransformAll (values, new Cuber());
foreach (int i in values)
  Console.WriteLine (i);

class Squarer : ITransformer
{
  public int Transform (int x) => x * x;
}

class Cuber : ITransformer
{
  public int Transform (int x) => x * x * x;
}
```

Zgodność delegatów

Zgodność typów

Typy delegacyjne są niezgodne ze sobą nawzajem, nawet jeśli mają takie same sygnatury:

```
D1 d1 = Method1;
D2 d2 = d1;          // błąd kompilacji

void Method1() { }
delegate void D1();
delegate void D2();
```

Ale poniższy kod jest poprawny:

```
D2 d2 = new D2 (d1);
```


Kup książkę   Poleć książkę

Egzemplarze delegatów są sobie równe, jeżeli mają te same metody docelowe:

```
D d1 = Method1;
D d2 = Method1;
Console.WriteLine (d1 == d2);   // prawda

void Method1() { }
delegate void D();
```

Delegaty multiemisji są sobie równe, jeśli odwołują się do tych samych metod *w takiej samej kolejności*.

Zgodność parametrów

W wywołaniu metody można podać argumenty o bardziej specyficznym typie niż wskazują jej parametry. Jest to normalny polimorfizm. Dokładnie z tego samego powodu delegat może mieć bardziej specyficzne typy parametrów niż jego metoda docelowa. Nazywa się to **kontrawariancją**. Na przykład:

```
StringAction sa = new StringAction (ActOnObject);
sa ("cześć");
void ActOnObject (object o) => Console.WriteLine (o); // cześć

delegate void StringAction (string s);
```

(Podobnie jak w przypadku wariancji parametrów typu, delegaty są wariantne tylko dla **konwersji referencji**).

Delegat jedynie wywołuje metodę w czyimś imieniu. W tym przypadku wywoływana jest metoda `StringAction` z argumentem typu `string`. Podczas przekazywania argumentu do metody docelowej następuje jego niejawne rzutowanie w górę do typu `object`.

Standardowy wzorzec zdarzeń pomaga w wykorzystywaniu kontrawariancji przez użycie wspólnej klasy bazowej `EventArgs`. Można np. wywołać jedną metodę poprzez dwa różne delegaty, z których jeden przekazuje obiekt typu `MouseEventArgs`, a drugi `KeyEventArgs`.

Zgodność typów zwrotnych

Wywołując metodę, można otrzymać bardziej specyficzny typ niż się chce. Jest to normalny polimorfizm. Z dokładnie tego samego powodu metoda docelowa delegatu może zwracać bardziej specyficzny typ niż typ wskazany przez delegat. Nazywa się to **kowariancją**:

```
ObjectRetriever o = new ObjectRetriever (RetrieveString);
object result = o();
Console.WriteLine (result); // cześć

static string RetrieveString() => "cześć";

delegate object ObjectRetriever();
```

`ObjectRetriever` oczekuje typu `object`, ale może też być obiekt *podklasy* typu `object`: typy zwrotne delegatów są **kowariantne**.

Kup ksi k   Pole ksi k

Wariancja parametrów typów delegatów generycznych

W rozdziale 3. pokazaliśmy, jak interfejsy generyczne wspierają kowariantne i kontrawariantne parametry typów. Taka sama funkcjonalność istnieje także w odniesieniu do delegatów.

Przy definiowaniu generycznego typu delegacyjnego do dobrych praktyk należy:

- oznaczenie parametru typu używanego tylko w odniesieniu do wartości zwrotnej jako kowariantnego (`out`);
- oznaczenie wszystkich typów parametrów używanych tylko w odniesieniu do parametrów jako kontrawariantnych (`in`).

W ten sposób umożliwia się naturalne działanie konwersji poprzez zapewnienie respektowania relacji dziedziczenia między typami.

Poniższy delegat (zdefiniowany w przestrzeni nazw `System`) ma kowariant `TResult`:

```
delegate TResult Func<out TResult>();
```

Pozwala to na:

```
Func<string> x = ...;
Func<object> y = x;
```

Poniższy delegat (zdefiniowany w przestrzeni nazw `System`) ma kontrawariant `T`:

```
delegate void Action<in T> (T arg);
```

Pozwala to na:

```
Action<object> x = ...;
Action<string> y = x;
```

Zdarzenia

Podczas pracy z delegatami pojawiają się dwie typowe role: **nadawcy** i **subskrybenta**.

Nadawca to typ zawierający pole delegacyjne. Decyduje on o tym, kiedy rozpocząć nadawanie poprzez wywołanie delegatu.

Subskrybenci to odbiorcy metody docelowej. Subskrybent decyduje, kiedy rozpocząć i zakończyć nasłuchiwanie, przez wywołanie operatorów `+=` i `-=` na delegacie nadawcy. Subskrybent nie zna innych subskrybentów i nie wchodzi z nimi w żadne reakcje.

Zdarzenia to funkcja języka stanowiąca formalne ujęcie tego wzorca. Zdarzenie (`event`) to konstrukcja udostępniająca tylko część funkcjonalności delegatów potrzebną do realizacji modelu nadawca – subskrybent. Głównym zastosowaniem zdarzeń jest *zapobieżenie interakcji między subskrybentami*.

Najłatwiejszym sposobem na zadeklarowanie zdarzenia jest wstawienie słowa kluczowego `event` przed składową delegacyjną:

```
// definicja delegatu
public delegate void PriceChangedHandler (decimal oldPrice,
                                          decimal newPrice);

public class Broadcaster
{
  // deklaracja zdarzenia
  public event PriceChangedHandler PriceChanged;
}
```


Kup ksi k Pole ksi k

Kod znajdujący się w typie `Broadcaster` ma pełny dostęp do zdarzenia `PriceChanged` i może je traktować jako delegat. Natomiast kod znajdujący się poza typem `Broadcaster` może tylko wykonywać operacje `+=` i `-=` na tym zdarzeniu.

Wewnętrzny mechanizm działania zdarzeń

Gdy programista deklaruje zdarzenie w sposób pokazany poniżej, dzieją się trzy rzeczy.

```
public class Broadcaster
{
  public event PriceChangedHandler PriceChanged;
}
```

Po pierwsze, kompilator tłumaczy deklarację zdarzenia na postać podobną do poniższej:

```
PriceChangedHandler priceChanged; // delegat prywatny
public event PriceChangedHandler PriceChanged
{
  add { priceChanged += value; }
  remove { priceChanged -= value; }
}
```

Słowa kluczowe `add` i `remove` to jawne **metody dostępowe zdarzenia** — działają podobnie do metod dostępowych własności. Sposoby ich pisania przedstawiamy nieco dalej.

Po drugie, kompilator szuka *w* klasie `Broadcaster` referencji do `PriceChanged` wykonujących inne operacje niż `+=` i `-=` oraz przekierowuje je do pola delegacyjnego `priceChanged`.

Po trzecie, kompilator zamienia operacje `+=` i `-=` dotyczące zdarzenia na wywołania metod `add` i `remove` tego zdarzenia. Co ciekawe, w wyniku tego operatory `+=` i `-=` w przypadku zdarzeń zyskują odmienne znaczenie. W innych przypadkach są one tylko skróconą formą zapisu operatorów `+` i `-` z przypisaniem.

Spójrz na poniższy przykład. Klasa `Stock` uruchamia swoje zdarzenie `PriceChanged` za każdym razem, gdy zmienia się cena towaru:

```
public delegate void PriceChangedHandler (decimal oldPrice,
                                          decimal newPrice);
public class Stock
{
  string symbol;
  decimal price;

  public Stock (string symbol) { this.symbol = symbol; }

  public event PriceChangedHandler PriceChanged;

  public decimal Price
  {
    get { return price; }
    set
    {
      if (price == value) return; // koniec pracy, jeśli nic się nie zmieniło
      decimal oldPrice = price;
      price = value;
      if (PriceChanged != null) // jeśli lista wywołań nie jest pusta,
```


Kup ksi k   Pole ksi k

```
            PriceChanged (oldPrice, price); // następuje uruchomienie zdarzenia
        }
      }
    }
```

Jeśli usuniemy z tego przykładu słowo kluczowe `event`, tak że `PriceChanged` stanie się zwykłym polem delegacyjnym, otrzymamy taki sam wynik. Ale wówczas klasa `Stock` będzie mniej niezawodna, ponieważ subskrybenci będą mogli wchodzić w interakcje, robiąc takie rzeczy:

- zamiana innych subskrybentów przez ponowne przypisanie do `PriceChanged` (zamiast przy użyciu operatora `+=`);
- skasowanie wszystkich subskrybentów (przez ustawienie `PriceChanged` na `null`);
- nadawanie do innych subskrybentów przez wywołanie delegatu.

Standardowy wzorzec zdarzeń

Prawie wszystkie zdarzenia w bibliotekach .NET są zdefiniowane wg standardowego wzorca. Ma on zapewnić spójność kodu użytkownika i samej platformy. Rdzeniem standardowego wzorca zdarzeń jest niemająca składowych (oprócz statycznej własności `Empty`) predefiniowana klasa .NET `System.EventArgs`. Jest to klasa bazowa do przekazywania informacji dla zdarzenia. W naszym przykładzie z klasą `Stock` moglibyśmy utworzyć podklasę klasy `EventArgs`, aby przekazywać stare i nowe ceny w odpowiedzi na uruchomienie zdarzenia `PriceChanged`:

```
public class PriceChangedEventArgs : System.EventArgs
{
  public readonly decimal LastPrice;
  public readonly decimal NewPrice;

  public PriceChangedEventArgs (decimal lastPrice, decimal newPrice)
  {
    LastPrice = lastPrice;
    NewPrice = newPrice;
  }
}
```

Ze względu na możliwość wielokrotnego wykorzystania podklasie klasy `EventArgs` nadaliśmy nazwę odpowiadającą informacjom, jakie zawiera, a nie zdarzeniu, dla jakiego będzie używana. Dane z reguły są udostępniane w postaci własności albo pól tylko do odczytu.

Następnym krokiem po zdefiniowaniu podklasy klasy `EventArgs` jest wybór lub zdefiniowanie delegatu dla zdarzenia. Obowiązują trzy zasady:

- Delegat musi mieć typ zwrotny `void`.
- Delegat musi przyjmować dwa argumenty: pierwszy typu `object`, a drugi podklasy klasy `EventArgs`. Pierwszy argument określa nadawcę zdarzeń, a drugi zawiera dodatkowe informacje do przekazania.
- Nazwa delegatu musi się kończyć słowami **`EventHandler`**.

Kup ksi k Pole ksi k

Platforma .NET definiuje generyczny delegat o nazwie `System.EventHandler<>`, który spełnia te warunki:

```
public delegate void EventHandler<TEventArgs>
  (object source, TEventArgs e) where TEventArgs : EventArgs;
```

Zanim do języka C# wprowadzono typy generyczne (co nastąpiło w wersji 2.0), konieczne było pisanie delegatów dostosowanych do konkretnych potrzeb w następujący sposób:

```
public delegate void PriceChangedHandler
  (object sender, PriceChangedEventArgs e);
```

Z powodów historycznych większość zdarzeń na platformie .NET używa tak właśnie zdefiniowanych delegatów.

Następnym krokiem jest zdefiniowanie zdarzenia wybranego typu delegacyjnego. W poniższym przykładzie używamy generycznego delegatu `EventHandler`:

```
public class Stock
{
  ...
  public event EventHandler<PriceChangedEventArgs> PriceChanged;
}
```

I w końcu wzorzec zakłada napisanie chronionej metody wirtualnej uruchamiającej zdarzenie. Jej nazwa musi być taka sama jak nazwa zdarzenia, poprzedzona przedrostkiem **`On`**. Metoda ta powinna przyjmować jeden argument `EventArgs`:

```
public class Stock
{
  ...

  public event EventHandler<PriceChangedEventArgs> PriceChanged;

  protected virtual void OnPriceChanged (PriceChangedEventArgs e)
  {
    if (PriceChanged != null) PriceChanged (this, e);
  }
}
```

Jest to centralny punkt, od którego podklasy mogą wywoływać lub przesłaniać zdarzenia (przy założeniu, że klasa nie jest zapieczętowana).

W programach wielowątkowych (rozdział 14.) należy przypisać delegat do zmiennej tymczasowej, zanim się go przetestuje i wywoła. W ten sposób można uniknąć błędu związanego z bezpieczeństwem wątków:

```
var temp = PriceChanged;
if (temp != null) temp (this, e);
```

Taką samą funkcjonalność można uzyskać bez użycia zmiennej `temp`, za pomocą operatora warunkowego `null`:

```
PriceChanged?.Invoke (this, e);
```

Ten bezpieczny dla wątków i zwięzły kod jest aktualnie najlepszym ogólnym sposobem wywoływania zdarzeń.

Kup ksi k Pole ksi k

Oto kompletny przykład:

```
using System;

Stock stock = new Stock ("THPW");
stock.Price = 27.10M;
// rejestracja w zdarzeniu PriceChanged
stock.PriceChanged += stock_PriceChanged;
stock.Price = 31.59M;

void stock_PriceChanged (object sender, PriceChangedEventArgs e)
{
    if ((e.NewPrice - e.LastPrice) / e.LastPrice > 0.1M)
        Console.WriteLine ("Uwaga: wzrost cen akcji o 10%!");
}

public class PriceChangedEventArgs : EventArgs
{
  public readonly decimal LastPrice;
  public readonly decimal NewPrice;

  public PriceChangedEventArgs (decimal lastPrice, decimal newPrice)
  {
    LastPrice = lastPrice; NewPrice = newPrice;
  }
}

public class Stock
{
  string symbol;
  decimal price;

  public Stock (string symbol) => this.symbol = symbol;

  public event EventHandler<PriceChangedEventArgs> PriceChanged;

  protected virtual void OnPriceChanged (PriceChangedEventArgs e)
  {
    PriceChanged?.Invoke (this, e);
  }

  public decimal Price
  {
    get => price;
    set
    {
      if (price == value) return;
      decimal oldPrice = price;
      price = value;
      OnPriceChanged (new PriceChangedEventArgs (oldPrice, price));
    }
  }
}
```

Predefiniowanego niegenerycznego delegatu `EventHandler` można używać, gdy zdarzenie nie przenosi dodatkowych informacji. W poniższym przykładzie przedstawiamy zmienioną wersję klasy `Stock`, w której zdarzenie `PriceChanged` jest uruchamiane po zmianie ceny i nie są potrzebne żadne dodatkowe informacje o zdarzeniu poza tym, że ono nastąpiło. Wykorzystujemy też właściwość `EventArgs.Empty`, aby uniknąć niepotrzebnego tworzenia egzemplarza `EventArgs`.

Kup ksi k Pole ksi k

```
public class Stock
{
  string symbol;
  decimal price;

  public Stock (string symbol) { this.symbol = symbol; }

  public event EventHandler PriceChanged;

  protected virtual void OnPriceChanged (EventArgs e)
  {
    PriceChanged?.Invoke (this, e);
  }

  public decimal Price
  {
    get { return price; }
    set
    {
      if (price == value) return;
      price = value;
      OnPriceChanged (EventArgs.Empty);
    }
  }
}
```

Metody dostępowe zdarzeń

Metody dostępowe zdarzeń (ang. *accessors*) są implementacjami funkcji += i -=. Domyślnie są one implementowane niejawnie przez kompilator. Spójrz na poniższą deklarację zdarzenia:

```
public event EventHandler PriceChanged;
```

Kompilator przekonwertuje je na następującą postać:

- prywatne pole delegacyjne;
- publiczna para funkcji dostępowych zdarzenia (`add_PriceChanged` i `remove_PriceChanged`), których implementacje przekazują operacje += i -= do prywatnego pola delegacyjnego.

Proces ten można przejąć przez zdefiniowanie **jawnych** metod dostępowych zdarzenia. Poniżej znajduje się napisana ręcznie implementacja zdarzenia `PriceChanged` z poprzedniego przykładu:

```
private EventHandler priceChanged; // deklaracja prywatnego delegatu

public event EventHandler PriceChanged
{
  add    { priceChanged += value; }
  remove { priceChanged -= value; }
}
```

Pod względem funkcjonalnym ten kod jest identyczny z domyślną implementacją C# (z tym wyjątkiem, że C# dodatkowo zapewnia bezpieczeństwo wątkowe w odniesieniu do aktualizowania delegatu poprzez bezblokadowy algorytm porównywania i zamieniania — zob. *http://albahari.com/threading*). Definiując metody dostępowe samodzielnie, sygnalizujemy C#, aby nie generował domyślnego pola i logiki dostępowej.

Kup ksi k   Pole ksi k

Mając jawnie zdefiniowane metody dostępowe zdarzenia, można wykorzystywać bardziej wyszukane techniki zapisywania i używania delegatu. Jest to przydatne w trzech sytuacjach:

- Gdy metody dostępowe zdarzenia są jedynie przekaźnikami dla innej klasy, która nadaje zdarzenie.
- Gdy klasa udostępnia wiele zdarzeń i przez większość czasu istnieje bardzo niewielu subskrybentów, np. kontrolka Windows. W takich przypadkach lepszym rozwiązaniem jest zapisanie egzemplarzy subskrybenta delegatu w słowniku, ponieważ słownik zajmie mniej nadmiarowej pamięci niż dziesiątki pustych referencji do pól delegacyjnych.
- Przy jawnej implementacji interfejsu deklarującego zdarzenie.

Poniżej znajduje się przykład ilustrujący treść ostatniego punktu:

```
public interface IFoo { event EventHandler Ev; }

class Foo : IFoo
{
  private EventHandler ev;

  event EventHandler IFoo.Ev
  {
    add    { ev += value; }
    remove { ev -= value; }
  }
}
```

Składniki `add` i `remove` zdarzenia są kompilowane do postaci metod `add_XXX` i `remove_XXX`.

Modyfikatory zdarzeń

Zdarzenia, podobnie jak metody, mogą być wirtualne, abstrakcyjne, przesłaniane i pieczętowane, a także statyczne:

```
public class Foo
{
  public static event EventHandler<EventArgs> StaticEvent;
  public virtual event EventHandler<EventArgs> VirtualEvent;
}
```

Wyrażenia lambda

Wyrażenie lambda to metoda bez nazwy wpisana w miejsce egzemplarza delegatu. Kompilator konwertuje wyrażenie lambda na:

- egzemplarz delegatu;
- **drzewo wyrażeń** (typu `Expression<TDelegate>`) reprezentujące kod wyrażenia lambda w postaci nadającego się do przeglądania modelu obiektów. Umożliwia to interpretację wyrażenia w czasie wykonywania programu (zob. „Budowanie wyrażeń zapytań” w rozdziale 8.).

Kup ksi k Pole ksi k

W poniższym przykładzie `x => x * x` jest wyrażeniem lambda:

```
Transformer sqr = x => x * x;
Console.WriteLine (sqr(3)); // 9

delegate int Transformer (int i);
```

Kompilator rozpoznaje tego typu wyrażenia lambda przez napisanie metody prywatnej i przeniesienie do tej metody kodu wyrażenia.

Ogólna postać wyrażenia lambda jest taka:

```
(parametry) => wyrażenie-lub-blok-instrukcji
```

Dla wygody nawias można opuścić, ale tylko, jeżeli jest dokładnie jeden parametr typu, który da się wydedukować.

W naszym przykładzie jest tylko jeden parametr, `x`, a wyrażenie to `x * x`:

```
x => x * x;
```

Każdy parametr wyrażenia lambda odpowiada parametrowi delegatu, a typ wyrażenia (który może być też `void`) odpowiada typowi zwrotnemu delegatu.

W naszym przykładzie `x` odpowiada parametrowi `i`, a wyrażenie `x * x` odpowiada typowi zwrotnemu `int`, więc wyrażenie jest zgodne z delegatem `Transformer`:

```
delegate int Transformer (int i);
```

Kod wyrażenia lambda może stanowić też **blok instrukcji**, nie tylko pojedyncze wyrażenie. Nasz przykład możemy przepisać tak:

```
x => { return x * x; };
```

Wyrażeń lambda najczęściej używa się z delegatami `Func` i `Action`, więc poprzednie wyrażenie w większości przypadków miałoby taką postać:

```
Func<int,int> sqr = x => x * x;
```

Poniżej znajduje się przykład wyrażenia przyjmującego dwa parametry:

```
Func<string,string,int> totalLength = (s1, s2) => s1.Length + s2.Length;
int total = totalLength ("witaj", "świecie"); // zmienna total ma wartość 12
```

W C# 9, jeśli nawias jest niepotrzebny, można to zaznaczyć za pomocą znaków podkreślenia:

```
Func<string,string,int> totalLength = (_,_) => ...
```

Jawne określanie typów parametrów lambdy

Kompilator zazwyczaj jest w stanie *wydedukować* typy parametrów lambdy. Ale jeśli jest to niemożliwe, programista musi jawnie podać typ każdego parametru. Spójrz na poniższe dwie metody:

```
void Foo<T> (T x)         {}
void Bar<T> (Action<T> a) {}
```


Kup ksi k   Pole ksi k

Poniższy kod nie przejdzie kompilacji, ponieważ kompilator nie będzie w stanie wydedukować typu x:

```
Bar (x => Foo (x)); // Jakiego typu jest x?
```

Rozwiązaniem tego problemu jest jawne określenie typu x:

```
Bar ((int x) => Foo (x));
```

Ten konkretny przykład jest tak prosty, że można go poprawić jeszcze na dwa inne sposoby:

```
Bar<int> (x => Foo (x));      // podanie parametru typu dla Bar
Bar<int> (Foo);               // jak powyżej, tylko z grupą metod
```

Przechwytywanie zmiennych zewnętrznych

Wyrażenie lambda może odwoływać się do każdej zmiennej dostępnej w zakresie, w którym jest zdefiniowane to wyrażenie. Są to tzw. **zmienne zewnętrzne** i zaliczają się do nich zmienne lokalne, parametry oraz pola:

```
int factor = 2;
Func<int, int> multiplier = n => n * factor;
Console.WriteLine (multiplier (3)); // 6
```

Zmienne zewnętrzne używane przez wyrażenie lambda nazywa się **zmiennymi przechwyconymi** (ang. *captured variables*). Wyrażenie lambda przechwytujące zmienne nazywa się **domknięciem** (ang. *closure*).

Zmienne mogą być przechwytywane także przez metody anonimowe i lokalne. Reguły ich dotyczące są w tych przypadkach takie same.

Wartości przechwyconych zmiennych są obliczane w chwili *wywoływania* delegatu, a nie w czasie *przechwytywania*:

```
int factor = 2;
Func<int, int> multiplier = n => n * factor;
factor = 10;
Console.WriteLine (multiplier (3)); // 30
```

Wyrażenia lambda mogą zmieniać przechwycone zmienne:

```
int seed = 0;
Func<int> natural = () => seed++;
Console.WriteLine (natural());    // 0
Console.WriteLine (natural());    // 1
Console.WriteLine (seed);         // 2
```

Zakres istnienia przechwyconych zmiennych rozszerza się na zakres istnienia delegatu. W poniższym przykładzie zmienna lokalna seed normalnie zniknęłaby z zakresu dostępności po zakończeniu działania Natural. Ale ponieważ została *przechwycona*, jej zakres istnienia został rozszerzony do zakresu istnienia delegatu natural:

```
static Func<int> Natural()
{
  int seed = 0;
  return () => seed++; // zwraca domknięcie
```


Kup ksi k Pole ksi k

```
}

static void Main()
{
  Func<int> natural = Natural();
  Console.WriteLine (natural()); // 0
  Console.WriteLine (natural()); // 1
}
```

Zmienna lokalna, której egzemplarz został *utworzony* w wyrażeniu lambda, jest unikatowa dla każdego wywołania egzemplarza delegatu. Jeśli zmienimy nasz przykład tak, aby egzemplarz zmiennej `seed` był tworzony *w* wyrażeniu lambda, otrzymamy inny wynik (w tym przypadku niepożądany):

```
static Func<int> Natural()
{
  return() => { int seed = 0; return seed++; };
}

static void Main()
{
  Func<int> natural = Natural();
  Console.WriteLine (natural()); // 0
  Console.WriteLine (natural()); // 0
}
```

Wewnętrzna implementacja przechwytywania zmiennych polega na „wciągnięciu” przechwyconych zmiennych do pól prywatnej klasy. Gdy zostaje wywołana metoda, następuje utworzenie egzemplarza klasy i powstanie powiązania z egzemplarzem delegatu.

Statyczne lambdy (C# 9)

Kiedy przechwytujemy zmienne lokalne, parametry, pola egzemplarza lub referencję `this`, kompilator może być zmuszony utworzyć prywatną klasę i jej obiekt, aby móc przechowywać referencję do przechwyconych danych. To pociąga za sobą drobny koszt wydajnościowy, ponieważ wymaga alokacji pamięci (którą potem trzeba odzyskać). W przypadku ostrych wymogów w zakresie wydajności można zdecydować się na mikrooptymalizację polegającą na minimalizacji obciążenia systemu usuwania nieużytków przez wyeliminowanie alokacji w newralgicznych miejscach.

Od C# 9 do definicji wyrażenia lambda, funkcji lokalnej i metody anonimowej można dodać słowo kluczowe `static`, aby wyłączyć przechowywanie stanu. Taka mikrooptymalizacja pozwala wyeliminować niepotrzebne alokacje pamięci. Poniżej znajduje się przykład zastosowania modyfikatora `static` do wyrażenia lambda:

```
Func<int, int> multiplier = static n => n * 2;
```

Jeśli później spróbujemy zmodyfikować to wyrażenie tak, aby przechwytywało zmienną lokalną, kompilator wygeneruje błąd:

```
int factor = 2;
Func<int, int> multiplier = static n => n * factor; // nie przejdzie kompilacji
```


Kup książkę Poleć książkę

Sama lambda staje się egzemplarzem delegatu, który wymaga alokacji pamięci. Jeśli jednak lambda ta nie przechwytuje zmiennych, kompilator będzie wykorzystywać jeden zapisany w buforze egzemplarz przez cały okres działania aplikacji, więc w rzeczywistości nie powstanie żaden koszt.

Można to wykorzystać także w odniesieniu do metod lokalnych. W poniższym przykładzie metoda `Multiply` nie ma dostępu do zmiennej `factor`:

```
void Foo()
{
    int factor = 123;
    static int Multiply (int x) => x * 2; // lokalna metoda statyczna
}
```

Oczywiście metoda `Multiply` nadal może jawnie alokować pamięć za pomocą operatora `new`. Opisana technika chroni nas tylko przed *potajemną* alokacją. Zastosowanie słowa kluczowego `static` w tym przypadku może mieć znaczenie dokumentacyjne, ponieważ wskazuje na ograniczony poziom powiązań.

Statyczne lambdy mają dostęp do statycznych zmiennych i stałych (gdyż te nie wymagają domknięcia).

Słowo kluczowe `static` odgrywa jedynie rolę *sprawdzającą*. Nie ma wpływu na wygenerowany przez kompilator kod IL. Bez słowa kluczowego `static` kompilator nie generuje domknięcia, jeśli nie musi tego robić (a nawet wtedy stosuje sztuczki pozwalające ograniczyć koszt).

Przechwytywanie zmiennych iteracyjnych

Gdy przechwycimy zmienną iteracyjną pętli `for`, C# traktuje ją tak, jakby została zadeklarowana *poza* pętlą. Oznacza to, że w każdej iteracji przechwytywana jest *ta sama* zmienna. Poniższy program wydrukuje `333` zamiast `012`:

```
Action[] actions = new Action[3];
for (int i = 0; i < 3; i++)
  actions [i] = () => Console.Write (i);
foreach (Action a in actions) a();     // 333
```

Każde domknięcie (oznaczone tłustym drukiem) przechwytuje tę samą zmienną `i`. (Ma to sens, jeśli weźmie się pod uwagę fakt, że wartość zmiennej `i` jest przechowywana między iteracjami pętli; w razie potrzeby można nawet jawnie zmienić `i` w kodzie pętli). W konsekwencji, jeśli później zostaną wywołane delegaty, każdy z nich otrzyma wartość `i` z czasu *wywołania*, która wynosi 3. Lepiej to zrozumieć, gdy rozwinie się pętlę `for`:

```
Action[] actions = new Action[3];
int i = 0;
actions[0] = () => Console.Write (i);
i = 1;
actions[1] = () => Console.Write (i);
i = 2;
actions[2] = () => Console.Write (i);
i = 3;
foreach (Action a in actions) a();   // 333
```


Kup ksi k   Pole ksi k

Jeśli oczekiwanym wynikiem jest `012`, to rozwiązaniem jest przypisanie zmiennej iteracyjnej do zmiennej lokalnej o zakresie dostępności ograniczonym do *wnętrza* pętli:

```
Action[] actions = new Action[3];
for (int i = 0; i < 3; i++)
{
  int loopScopedi = i;
  actions [i] = () => Console.Write (loopScopedi);
}
foreach (Action a in actions) a(); // 012
```

Dzięki temu, że zmienna `loopScopedi` jest tworzona na nowo w każdej iteracji, każde domknięcie przechwytuje *inną* zmienną.

Przed C# 5.0 w taki sam sposób działały pętle `foreach`. Było to przyczyną wielu nieporozumień: zmienna iteracyjna pętli `foreach`, w odróżnieniu od pętli `for`, jest niezmienna, więc programista może pomyśleć, że będzie traktowana jako lokalna w pętli. Dobra wiadomość jest taka, że już to poprawiono i można bezpiecznie przechwytywać zmienną iteracyjną pętli `foreach` bez niespodzianek.

Wyrażenia lambda a metody lokalne

Funkcjonalność metod lokalnych (zobacz „Metody lokalne" w rozdziale 1.) częściowo pokrywa się z funkcjonalnością wyrażeń lambda. Metody lokalne mają trzy następujące zalety:

- Mogą być rekurencyjne (wywoływać same siebie) bez stosowania brzydkich sztuczek.
- Pozwalają uniknąć niezręcznego określania typu delegatu.
- Wiążą się z nieco mniejszym narzutem.

Metody lokalne są efektywniejsze, ponieważ unikają pośredniości charakterystycznej dla delagatów (która kosztuje cykle procesora i miejsce w pamięci). Ponadto mogą używać lokalnych zmiennych metody nadrzędnej bez potrzeby przenoszenia ich przez kompilator do ukrytej klasy.

Jednak w wielu przypadkach delegat jest po prostu *niezbędny*, najczęściej przy wywoływaniu funkcji wyższego rzędu, czyli metody z parametrem typu delegacyjnego:

```
public void Foo (Func<int,bool> predicate) { ... }
```

(Więcej przykładów pokazujemy w rozdziale 8.). W takich przypadkach delegat i tak jest potrzebny i to właśnie w tych przypadkach wyrażenia lambda są z reguły zwięźlejsze i klarowniejsze.

Metody anonimowe

Metody anonimowe zostały wprowadzone w C# 2.0 i są w zasadzie zastąpione przez wyrażenia lambda wprowadzone w C# 3.0. Metoda anonimowa jest podobna do wyrażenia lambda, tylko brakuje jej:

- niejawnie typowanych parametrów;
- składni wyrażeniowej (metoda anonimowa musi być blokiem instrukcji);
- możliwości kompilacji do postaci drzewa wyrażeń przez przypisanie do `Expression<T>`.

Kup ksi k Pole ksi k

Aby napisać metodę anonimową, należy użyć słowa kluczowego `delegate` z opcjonalną deklaracją parametru, po której dodaje się właściwy kod metody. Weźmy np. poniższy delegat:

```
Transformer sqr = delegate (int x) {return x * x;};
Console.WriteLine (sqr(3)); // 9

delegate int Transformer (int i);
```

Pierwszy wiersz tego kodu jest semantycznie równoważny poniższemu wyrażeniu lambda:

```
Transformer sqr = (int x) => {return x * x;};
```

Albo prościej:

```
Transformer sqr = x => x * x;
```

Metody anonimowe przechwytują zmienne zewnętrzne w taki sam sposób, jak robią to wyrażenia lambda, i mogą mieć w definicji słowo kluczowe `static`, które sprawia, że zachowują się jak statyczne lambdy.

Cechą wyróżniającą metody anonimowe jest możliwość całkowitego opuszczenia deklaracji parametru, nawet jeśli jest on wymagany przez delegat. Może się to przydać w deklarowaniu zdarzeń z domyślną pustą procedurą obsługową:

```
public event EventHandler Clicked = delegate { };
```

W ten sposób eliminujemy konieczność sprawdzania wartości `null` przed uruchomieniem zdarzenia. Poniższy kod też jest poprawny:

```
// zwróć uwagę na pominięcie parametrów
Clicked += delegate { Console.WriteLine ("clicked"); };
```

Instrukcje try i wyjątki

Instrukcja `try` oznacza blok kodu podlegającego procedurom obsługi błędów lub procedurom porządkującym. Każdemu *blokowi* `try` musi towarzyszyć przynajmniej jeden *blok* `catch`, *blok* `finally` lub oba. Blok `catch` jest wykonywany, gdy w bloku `try` wystąpi błąd. Blok `finally` jest wykonywany po wyjściu z bloku `try` (lub ewentualnie `catch`, jeśli ten istnieje) w celu przeprowadzenia czynności porządkowych niezależnie od tego, czy wystąpił błąd, czy nie.

Blok `catch` ma dostęp do obiektu `Exception` zawierającego informacje o błędzie. Blok ten można wykorzystać w celu rozwiązania problemu albo *ponownego zgłoszenia* wyjątku. Wyjątek zgłasza się ponownie, gdy chce się tylko zarejestrować problem lub trzeba zgłosić nowy typ wyjątku wyższego poziomu.

Blok `finally` dodaje do programu odrobinę przewidywalności: CLR zawsze próbuje go wykonać. Można go wykorzystać do wykonywania czynności porządkowych, np. zamykania połączeń sieciowych.

Instrukcja `try` wygląda tak:

```
try
{
  ... // w czasie wykonywania tego bloku może zostać zgłoszony wyjątek
}
catch (ExceptionA ex)
```


Kup ksi k Pole ksi k

```
{
  ... // obsługa wyjątku typu ExceptionA
}
catch (ExceptionB ex)
{
  ... // obsługa wyjątku typu ExceptionB
}
finally
{
  ... // kod porządkowy
}
```

Spójrz na poniższy program:

```
int y = Calc (0);
Console.WriteLine (y);

int Calc (int x) => 10 / x
```

Jako że wartość x wynosi 0, system wykonawczy zgłosi wyjątek DivideByZeroException i program zakończy działanie. Możemy temu zapobiec przez przechwycenie wyjątku w następujący sposób:

```
try
{
    int y = Calc (0);
    Console.WriteLine (y);
}
catch (DivideByZeroException ex)
{
  Console.WriteLine ("x nie może mieć wartości zero.");
}
Console.WriteLine ("Program zakończył działanie z powodzeniem.");

int Calc (int x) => 10 / x;
```

Wynik:

```
x nie może mieć wartości zero.
Program zakończył działanie z powodzeniem.
```

Jest to prosty przykład ilustrujący technikę obsługi wyjątków. W praktyce można by było to lepiej rozwiązać przez sprawdzenie wprost, czy dzielnik nie ma wartości zero, przed wywołaniem Calc.

Błędy, którym można zapobiec, lepiej jest wykrywać zawczasu niż pozostawiać do wykrycia w blokach try-catch, ponieważ obsługa wyjątków pochłania przynajmniej setki cykli zegara.

Gdy dochodzi do zgłoszenia wyjątku w bloku try, CLR wykonuje test:

Czy istnieje odpowiednia klauzula catch dla tej instrukcji try?

- Jeśli tak, sterowanie zostaje przekazane do odpowiedniego bloku catch, potem do bloku finally (jeśli jest) i w końcu program wraca do normalnego wykonywania.
- Jeżeli nie, sterowanie jest przekazywane prosto do bloku finally (jeśli jest), następnie CLR szuka na stosie innych bloków try. Jeśli jakiś znajdzie, powtarza test…

Jeśli żadna funkcja na stosie wywołań nie obsługuje wyjątku, następuje zamknięcie programu.

Kup ksi k Pole ksi k

Klauzula catch

Klauzula `catch` określa, jakiego rodzaju wyjątki mają być przechwytywane. Wszystkie one muszą być typu `System.Exception` lub jego podklasy.

Przechwytywanie wyjątków typu `System.Exception` jest tożsame z przechwytywaniem wszystkich możliwych błędów. Jest to przydatne, gdy:

- program może odzyskać sprawność niezależnie od konkretnego typu wyjątku;
- programista planuje zgłosić wyjątek ponownie (uprzednio rejestrując go np. w dzienniku);
- procedura obsługi błędów jest ostatnią deską ratunku przed zamknięciem programu.

Częściej jednak przechwytuje się **specyficzne typy wyjątków**, aby uniknąć konieczności rozwiązywania problemów, do których dana procedura nie została przewidziana (np. wyjątków `OutOfMemoryException`).

Wiele typów wyjątków można obsługiwać za pomocą wielu klauzul `catch` (w poniższym przykładzie również lepiej byłoby bezpośrednio sprawdzać argumenty, zamiast posługiwać się wyjątkami):

```
class Test
{
  static void Main (string[] args)
  {
    try
    {
      byte b = byte.Parse (args[0]);
      Console.WriteLine (b);
    }
    catch (IndexOutOfRangeException)
    {
      Console.WriteLine ("Podaj przynajmniej jeden argument.");
    }
    catch (FormatException)
    {
      Console.WriteLine ("To nie jest liczba!");
    }
    catch (OverflowException)
    {
      Console.WriteLine ("Przekazano więcej niż jeden bajt!");
    }
  }
}
```

Dla danego typu wyjątku wykonywana jest tylko jedna klauzula `catch`. Jeśli potrzebne jest zabezpieczenie przechwytujące bardziej ogólne typy wyjątków (np. `System.Exception`), to klauzule dotyczące bardziej specyficznych typów powinny znaleźć się *przed* nim.

Jeśli nie jest potrzebny dostęp do własności, wyjątek można przechwycić bez określania zmiennej:

```
catch (OverflowException) // brak zmiennej
{
  ...
}
```


Kup ksi k Pole ksi k

Ponadto można opuścić zarówno zmienną, jak i typ (co oznacza, że przechwytywane mają być wszystkie wyjątki):

```
catch { ... }
```

Filtry wyjątków

W klauzuli `catch` można używać **filtrów wyjątków**, które definiuje się za pomocą klauzuli `when`:

```
catch (WebException ex) when (ex.Status == WebExceptionStatus.Timeout)
{
  ...
}
```

Gdyby w tym przykładzie został zgłoszony wyjątek typu `WebException`, to nastąpiłoby obliczenie wartości wyrażenia logicznego znajdującego się za słowem kluczowym `when`. W przypadku fałszywego wyniku blok `catch` zostałby zignorowany i nastąpiłoby sprawdzenie następnych klauzul `catch`. Użycie filtrów wyjątków sprawia, że ma sens przechwytywanie tego samego wyjątku kilka razy:

```
catch (WebException ex) when (ex.Status == WebExceptionStatus.Timeout)
{ ... }
catch (WebException ex) when (ex.Status == WebExceptionStatus.SendFailure)
{ ... }
```

Wyrażenie logiczne w klauzuli `when` może mieć skutki uboczne, jak np. metoda rejestrująca wyjątek w dzienniku w celach diagnostycznych.

Blok finally

Blok `finally` jest zawsze wykonywany, niezależnie od tego, czy wyjątek zostanie zgłoszony i czy blok `try` zostanie wykonany do końca. W blokach `finally` najczęściej wpisuje się kod porządkujący.

Blok `finally` zostaje wykonany:

- po zakończeniu wykonywania bloku `catch` (lub zgłoszenia w nim wyjątku);
- po zakończeniu wykonywania bloku `try` (lub po zgłoszeniu przez niego wyjątku, dla którego nie ma bloku `catch`);
- po wyjściu sterowania z bloku `try` z powodu wykonania instrukcji skoku (np. `return` albo `goto`);

Wykonanie bloku `finally` może zostać uniemożliwione tylko przez pętlę nieskończoną i przez nagłe zakończenie procesu.

Blok `finally` pomaga w zapewnieniu przewidywalności działania programu. W poniższym przykładzie otwarty plik *zawsze* zostanie zamknięty, bez względu na to, czy:

- blok `try` zostanie wykonany normalnie;
- wykonywanie zakończy się przedwcześnie z powodu tego, że plik jest pusty (`EndOfStream`);
- podczas odczytu pliku zostanie zgłoszony wyjątek `IOException`.

Kup ksi k Pole ksi k

```
void ReadFile()
{
  StreamReader reader = null; // w przestrzeni nazw System.IO
  try
  {
    reader = File.OpenText ("file.txt");
    if (reader.EndOfStream) return;
    Console.WriteLine (reader.ReadToEnd());
  }
  finally
  {
    if (reader != null) reader.Dispose();
  }
}
```

W tym przykładzie zamknęliśmy plik przez wywołanie metody `Dispose` obiektu klasy `StreamReader`. Wywoływanie tej metody na obiekcie w bloku `finally` jest standardową techniką, która jest bezpośrednio obsługiwana w C# poprzez instrukcję `using`.

Instrukcja using

Wiele klas zawiera niezarządzane zasoby, takie jak: uchwyty do plików, uchwyty graficzne czy połączenia z bazami danych. Klasy te implementują interfejs `System.IDisposable`, który definiuje jedną bezparametrową metodę o nazwie `Dispose` służącą do porządkowania zasobów. Instrukcja `using` zapewnia elegancką składnię do wywoływania metody `Dispose` na obiektach implementujących interfejs `IDisposable` w blokach `finally`:

Poniższy kod:

```
using (StreamReader reader = File.OpenText ("file.txt"))
{
  ...
}
```

jest równoważny z tym:

```
{
  StreamReader reader = File.OpenText ("file.txt");
  try
  {
    ...
  }
  finally
  {
    if (reader != null)
      ((IDisposable)reader).Dispose();
  }
}
```

Deklaracje using

Jeśli opuścisz nawias i blok instrukcji za instrukcją `using`, staje się ona *deklaracją* `using` (C# 8). Wówczas dany zasób zostaje zlikwidowany, gdy sterowanie wyjdzie z *otaczającego* bloku instrukcji:

Kup ksi k Pole ksi k

```
if (File.Exists ("file.txt"))
{
    using var reader = File.OpenText ("file.txt");
    Console.WriteLine (reader.ReadLine());
    ...
}
```

W tym przypadku zmienna `reader` zostanie usunięta po zakończeniu wykonywania instrukcji `if`.

Zgłaszanie wyjątków

Wyjątki mogą być zgłaszane przez system wykonawczy lub przez instrukcje znajdujące się w kodzie użytkownika. W poniższym przykładzie metoda `Display` zgłasza wyjątek `System.Argument` ↳`NullException`:

```
try { Display (null); }
catch (ArgumentNullException ex)
{
    Console.WriteLine ("Caught the exception");
}

void Display (string name)
{
    if (name == null)
    throw new ArgumentNullException (nameof (name));
    Console.WriteLine (name);
}
```

Wyrażenia throw

`throw` może występować w roli wyrażenia w funkcjach będących wyrażeniami:

```
public string Foo() => throw new NotImplementedException();
```

Wyrażenie `throw` może występować także w trójargumentowych wyrażeniach warunkowych:

```
string ProperCase (string value) =>
value == null ? throw new ArgumentException ("value") :
value == "" ? "" :
char.ToUpper (value[0]) + value.Substring (1);
```

Ponawianie zgłoszenia wyjątku

Wyjątek można przechwycić i zgłosić jeszcze raz w następujący sposób:

```
try { ... }
catch (Exception ex)
{
  // zapisanie błędu w dzienniku
  ...
  throw; // ponowienie zgłoszenia wyjątku
}
```

Gdybyśmy instrukcję `throw` zamienili na `throw ex`, to przykład i tak by działał, tylko własność `StackTrace` nowego wyjątku nie opisywałaby już oryginalnego błędu.

Kup książkę Poleć książkę

Ponowienie w ten sposób zgłoszenia wyjątku umożliwia jego zarejestrowanie bez **pochłonięcia**. Przy okazji zyskujemy możliwość wycofania się z obsługi wyjątku, gdyby sytuacja okazała się inna, niż się spodziewaliśmy:

```
using System.Net; // (zob. rozdział 16.)
...

string s = null;
using (WebClient wc = new WebClient())
  try { s = wc.DownloadString ("http://www.albahari.com/nutshell/"); }
  catch (WebException ex)
  {
    if (ex.Status == WebExceptionStatus.Timeout)
      Console.WriteLine ("Timeout");
    else
      throw; // brak możliwości obsługi innych rodzajów wyjątków WebException, więc ponawiamy zgłoszenie
}
```

Można też stosować zwięźlejszą formę zapisu z użyciem filtra wyjątków:

```
catch (WebException ex) when (ex.Status == WebExceptionStatus.Timeout)
{
  Console.WriteLine ("Przekroczono limit czasu.");
}
```

Innym typowym działaniem jest ponowienie zgłoszenia z bardziej specyficznym typem wyjątku. Na przykład:

```
try
{
  ... // pobranie danych DateTime z XML
}
catch (FormatException ex)
{
  throw new XmlException ("Nieprawidłowe dane DateTime.", ex);
}
```

Zwróć uwagę, że przy tworzeniu wyjątku `XmlException` w drugim argumencie przekazaliśmy oryginalny wyjątek ex. Argument ten przekazuje wartość dla własności `InnerException` nowego wątku i jest pomocny w diagnostyce. Prawie wszystkie typy wyjątków udostępniają podobny konstruktor.

Mniej specyficzny typ wyjątku można zgłosić w sytuacji ograniczonego zaufania, aby nie ujawnić technicznych szczegółów potencjalnym hakerom.

Najważniejsze własności klasy System.Exception

Najważniejsze własności klasy `System.Exception` to:

`StackTrace`

Łańcuch reprezentujący wszystkie metody wywoływane od miejsca pochodzenia wyjątku do bloku `catch`.

`Message`

Łańcuch reprezentujący opis błędu.

Kup ksi k Pole ksi k

`InnerException`

Wewnętrzny wyjątek (jeśli jest), który spowodował wyjątek zewnętrzny. Może mieć kolejną własność `InnerException`.

W C# wszystkie wyjątki są wyjątkami wykonawczymi, tzn. w języku tym nie ma odpowiednika kontrolowanych wyjątków kompilacji znanych z Javy.

Najczęściej używane typy wyjątków

Poniższe typy wyjątków są powszechnie wykorzystywane w CLR i platformie .NET. Można je zgłaszać lub używać ich jako klas bazowych do definiowania własnych typów wyjątków.

`System.ArgumentException`

Rodzaj wyjątku zgłaszany, gdy funkcja zostanie wywołana z bezsensownym argumentem. Najczęściej sygnalizuje błąd programu.

`System.ArgumentNullException`

Podklasa klasy `ArgumentNullException` używana, gdy argument funkcji (nieoczekiwanie) ma wartość `null`.

`System.ArgumentOutOfRangeException`

Podklasa klasy `ArgumentException` używana, gdy argument (zazwyczaj liczbowy) jest za duży lub za mały. Wyjątek tego typu może np. zostać zgłoszony, gdy ktoś przekaże ujemną liczbę do funkcji przyjmującej tylko dodatnie wartości.

`System.InvalidOperationException`

Typ wyjątków zgłaszanych, gdy stan obiektu uniemożliwia metodzie prawidłowe działanie, bez względu na konkretne wartości argumentów. Przykładem może być próba odczytu nieotwartego pliku albo pobieranie następnego elementu z wyliczenia, którego lista została zmieniona w czasie trwania iteracji.

`System.NotSupportedException`

Typ wyjątków zgłaszany w celu zasygnalizowania, że określony element funkcjonalności nie jest obsługiwany. Dobrym przykładem jest wywołanie metody `Add` na kolekcji, dla której `IsReadOnly` zwraca wartość `true`.

`System.NotImplementedException`

Typ wyjątków oznaczający, że funkcja jeszcze nie została zaimplementowana.

`System.ObjectDisposedException`

Typ wyjątków zgłaszanych, gdy obiekt, na którym wywołano funkcję, został usunięty.

Innym często używanym typem wyjątków jest `NullReferenceException`. System CLR zgłasza go, gdy programista chce użyć składowej obiektu o wartości `null` (co wskazuje na błąd w kodzie programu). Wyjątek typu `NullReferenceException` można zgłosić bezpośrednio (w celach testowych) w następujący sposób:

```
throw null;
```


Kup ksi k Pole ksi k

Wzorzec metod TryXXX

Programista piszący metodę może zdecydować, aby w razie problemów metoda ta zwracała jakiś kod sygnalizujący niepowodzenie lub zgłaszała wyjątek. Generalnie wyjątki zgłasza się w przypadkach, gdy błąd jest poza normalnym przepływem sterowania albo gdy można się spodziewać, że bezpośredni podmiot wywołujący nie będzie w stanie sobie z nim poradzić. Czasami jednak najlepszym rozwiązaniem jest przekazanie konsumentowi obu możliwości. Przykładem takiego działania jest typ `int`, w którym zdefiniowano dwie wersje metody `Parse`:

```
public int Parse (string input);
public bool TryParse (string input, out int returnValue);
```

Jeśli coś się nie uda, metoda `Parse` zgłasza wyjątek, a `TryParse` zwraca wartość `false`.

Wzorzec ten można zaimplementować, stosując wywołanie przez metodę *`XXX`* metody `Try`*`XXX`*:

```
public typ-zwrotny XXX (typ-wejściowy input)
{
  typ-zwrotny returnValue;
  if (!TryXXX (input, out returnValue))
    throw new YYYException (...)
  return returnValue;
}
```

Alternatywy dla wyjątków

Tak jak jest w metodzie `int.TryParse`, funkcja może informować o niepowodzeniu przez wysłanie kodu sygnalizującego błąd do funkcji wywołującej za pomocą typu zwrotnego lub parametru. Choć ta technika sprawdza się w odniesieniu do prostych i przewidywalnych awarii, zastosowanie jej do wszystkich rodzajów błędów jest kłopotliwe, wymusza bowiem zaśmiecenie sygnatur metod oraz wprowadza niepotrzebne komplikacje. Ponadto nie da się jej rozszerzyć na funkcje niebędące metodami, takie jak operatory (np. dzielenia) czy własności. Alternatywą jest umieszczenie błędu w miejscu dostępnym dla wszystkich funkcji znajdujących się na stosie wywołań (np. w metodzie statycznej przechowującej bieżący błąd dla każdego wątku). Wówczas jednak każda funkcja musi uczestniczyć w propagacji błędów, co jest kłopotliwe i, jak na ironię, łatwo przy realizacji tego pomysłu popełnić błąd.

Wyliczenia i iteratory

Wyliczenia

Enumerator to umożliwiający tylko odczyt i poruszający się tylko do przodu kursor do przeglądania *sekwencji wartości*. C# traktuje jako enumerator każdy typ spełniający następujące warunki:

- Ma publiczną bezparametrową metodę o nazwie `MoveNext` i własność o nazwie `Current`.
- Implementuje interfejs `System.Collections.IEnumerator`,
- Implementuje interfejs `System.Collections.Generic.IEnumerator<T>`,

Kup ksi k		Pole ksi k

Instrukcja `foreach` iteruje przez obiekt **przeliczalny** (ang. *enumerable*). Obiekt przeliczalny to logiczna reprezentacja sekwencji elementów. Nie jest kursorem, tylko obiektem zapewniającym kursor dla samego siebie. Obiekt przeliczalny:

- Ma publiczną bezparametrową metodę o nazwie `GetEnumerator` zwracającą **enumerator**.
- Implementuje interfejs `System.Collections.IEnumerable`.
- Implementuje interfejs `System.Collections.Generic.IEnumerable<T>`.
- Od C# 9 może wiązać się z metodą rozszerzającą o nazwie `GetEnumerator`, która zwraca wyliczenie (zobacz „Metody rozszerzające”).

Typowy sposób korzystania z enumeratorów przedstawia się następująco:

```
class Enumerator // standardowo implementuje interfejs IEnumerator lub IEnumerator<T>
{
  public TypZmiennejIteratora Current { get {...} }
  public bool MoveNext() {...}
}

class Enumerable // standardowo implementuje interfejs IEnumerable lub IEnumerable<T>
{
  public Enumerator GetEnumerator() {...}
}
```

Oto wysokopoziomowy przykład iteracji przez znaki słowa *piwo* za pomocą instrukcji `foreach`:

```
foreach (char c in "piwo")
  Console.WriteLine (c);
```

A poniżej znajduje się niskopoziomowy przykład iteracji przez znaki słowa *piwo*, bez użycia instrukcji `foreach`:

```
using (var enumerator = "piwo".GetEnumerator())
  while (enumerator.MoveNext())
  {
    var element = enumerator.Current;
    Console.WriteLine (element);
  }
```

Jeżeli enumerator implementuje interfejs `IDisposable`, instrukcja `foreach` działa też jako instrukcja `using`, niejawnie usuwając obiekt enumeratora.

Bardziej szczegółowy opis interfejsów wyliczeniowych znajduje się w rozdziale 7.

Inicjalizatory kolekcji

Obiekt przeliczalny można utworzyć i zapełnić elementami w jednej instrukcji. Na przykład:

```
using System.Collections.Generic;
...
List<int> list = new List<int> {1, 2, 3};
```

Kompilator przetłumaczy ten kod na następującą postać:

```
using System.Collections.Generic;
...
List<int> list = new List<int>();
list.Add (1);
list.Add (2);
list.Add (3);
```


Kup ksi k Pole ksi k

Obiekt przeliczalny musi zatem implementować interfejs `System.Collections.IEnumerable` oraz mieć metodę `Add` przyjmującą odpowiednią liczbę argumentów wywołania. W podobny sposób można inicjalizować słowniki (zob. „Słowniki” w rozdziale 7.):

```
var dict = new Dictionary<int, string>()
{
  { 5, "five" },
  { 10, "ten" }
};
```

Lub zwięźlej:

```
var dict = new Dictionary<int, string>()
{
  [3] = "three",
  [10] = "ten"
};
```

Drugie rozwiązanie można stosować nie tylko do słowników, lecz do wszystkich typów, dla których istnieje indeksator.

Iteratory

Podczas gdy instrukcja `foreach` jest **konsumentem** enumeratora, iterator jest jego **producentem**. Poniżej przedstawiamy przykład, w którym za pomocą iteratora zwracamy ciąg liczb Fibonacciego (każda liczba jest sumą dwóch poprzednich):

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

foreach (int fib in Fibs(6))
    Console.Write (fib + " ");

  IEnumerable<int> Fibs (int fibCount)
  {
    for (int i = 0, prevFib = 1, curFib = 1; i < fibCount; i++)
    {
      yield return prevFib;
      int newFib = prevFib+curFib;
      prevFib = curFib;
      curFib = newFib;
    }
  }

Wynik: 1 1 2 3 5 8
```

Podczas gdy instrukcja `return` oznacza: „Oto wartość, o której zwrot prosiłeś tę metodę”, instrukcja `yield return` oznacza: „Oto następny element, którego żądałeś od tego enumeratora”. Instrukcja `yield` przekazuje sterowanie do podmiotu wywołującego, ale stan podmiotu wywołanego pozostaje zachowany, dzięki czemu metoda może kontynuować wykonywanie, gdy tylko wywołujący zażąda następnego elementu. Czas istnienia tego stanu jest powiązany z enumeratorem w taki sposób, że stan ten może zostać zwolniony, gdy wywołujący zakończy wyliczanie.

Kup ksi k Pole ksi k

Kompilator konwertuje metody iterujące na prywatne klasy implementujące interfejs `IEnumerable<T>` i/lub `IEnumerator<T>`. Logika znajdująca się w bloku iteratora zostaje „odwrócona” i wklejona do metody `MoveNext` i własności `Current` klasy wygenerowanej przez kompilator. Jeśli więc programista wywoła metodę iteracyjną, to tak naprawdę utworzy egzemplarz klasy utworzonej przez kompilator i jego kod nie zostanie wykonany! Kod programisty jest wykonywany dopiero po rozpoczęciu enumeracji przez sekwencję elementów, najczęściej za pomocą pętli `foreach`.

Iteratory mogą być metodami lokalnymi (zobacz „Metody lokalne” w rozdziale 3.).

Semantyka iteratorów

Iterator jest metodą, własnością lub indeksatorem zawierającym przynajmniej jedną instrukcję `yield`. Iterator musi zwracać jeden z czterech następujących interfejsów (w przeciwnym razie kompilator zgłosi błąd):

```
// interfejsy IEnumerable
System.Collections.IEnumerable
System.Collections.Generic.IEnumerable<T>

// interfejsy Enumerator
System.Collections.IEnumerator
System.Collections.Generic.IEnumerator<T>
```

Semantyka iteratora różni się w zależności od tego, czy zwraca interfejs typu `IEnumerable`, czy `IEnumerator`. Szerzej na ten temat piszemy w rozdziale 7.

Można użyć *wielu instrukcji* `yield`. Na przykład:

```
foreach (string s in Foo())
    Console.WriteLine(s); // drukuje: "Jeden","Dwa","Trzy"

  IEnumerable<string> Foo()
  {
    yield return "Jeden";
    yield return "Dwa";
    yield return "Trzy";
  }
```

Instrukcja yield break

W bloku iteratora nie można używać instrukcji `return`, tylko `yield break`, która sygnalizuje, że blok ten ma zakończyć działanie przedwcześnie bez zwracania żadnych dalszych elementów. Jej działanie można zaobserwować w poniższej zmodyfikowanej wersji metody `Foo`:

```
IEnumerable<string> Foo (bool breakEarly)
{
  yield return "Jeden";
  yield return "Dwa";

  if (breakEarly)
    yield break;

  yield return "Trzy";
}
```


Kup książkę Poleć książkę

Iteratory i bloki try-catch-finally

Instrukcji `yield return` nie można używać w blokach `try` z dołączoną klauzulą `catch`:

```
IEnumerable<string> Foo()
{
  try { yield return "Jeden"; } // niepoprawne
  catch { ... }
}
```

Ponadto instrukcja `yield return` nie może występować w blokach `catch` ani `finally`. Ograniczenia te wiążą się z tym, że kompilator tłumaczy iteratory na zwykłe klasy ze składowymi `MoveNext`, `Current` i `Dispose` i konwersja bloków obsługi wyjątków byłaby wyjątkowo skomplikowana.

Instrukcji `yield return` można natomiast używać w blokach `try` z dołączonym (tylko) blokiem `finally`:

```
IEnumerable<string> Foo()
{
  try { yield return "Jeden"; } // OK
  finally { ... }
}
```

Kod znajdujący się w bloku `finally` zostanie wykonany, gdy enumerator dojdzie do końca sekwencji lub zostanie usunięty. Instrukcja `foreach` niejawnie usuwa enumerator, gdy programista przedwcześnie przerwie operację, dzięki czemu konsumpcja enumeratorów w ten sposób jest bezpieczna. Podczas bezpośredniej pracy z enumeratorami można wpaść w pułapkę polegającą na przedwczesnym zakończeniu enumeracji bez usunięcia enumeratora, pomijając blok `finally`. Ryzyko to można wyeliminować przez opakowanie jawnej operacji użycia enumeratorów w instrukcję `using`:

```
string firstElement = null;
var sequence = Foo();
using (var enumerator = sequence.GetEnumerator())
  if (enumerator.MoveNext())
    firstElement = enumerator.Current;
```

Komponowanie sekwencji

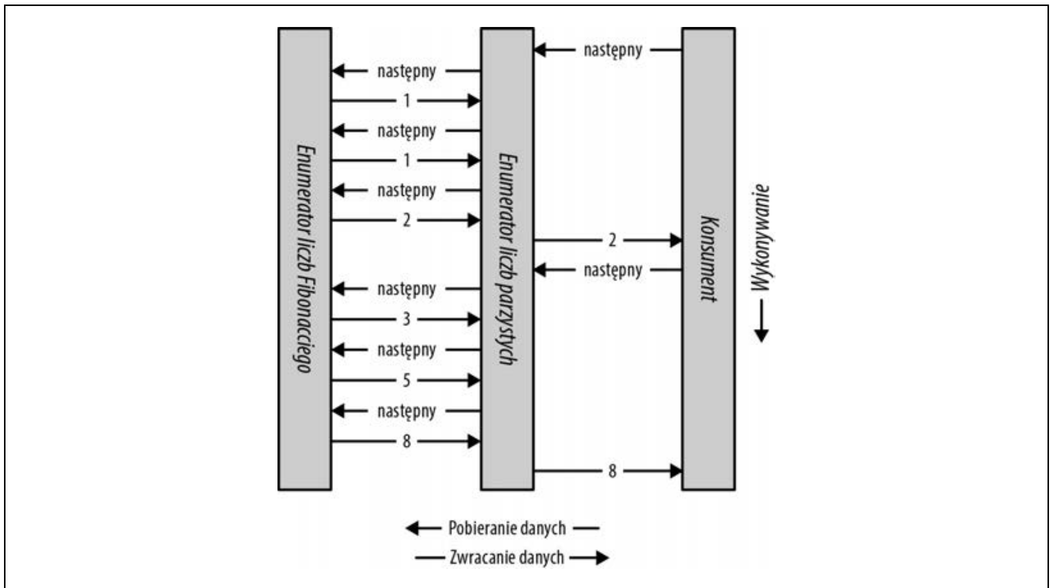
Iteratory można z łatwością komponować. Możemy zmodyfikować nasz przykład tak, aby zwracał tylko parzyste liczby z ciągu Fibonacciego:

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

foreach (int fib in EvenNumbersOnly (Fibs(6)))
    Console.WriteLine (fib);


  IEnumerable<int> Fibs (int fibCount)
  {
    for (int i = 0, prevFib = 1, curFib = 1; i < fibCount; i++)
    {
      yield return prevFib;
      int newFib = prevFib+curFib;
      prevFib = curFib;
      curFib = newFib;
```


Kup ksi k Pole ksi k

```
    }
  }

  IEnumerable<int> EvenNumbersOnly (IEnumerable<int> sequence)
  {
    foreach (int x in sequence)
      if ((x % 2) == 0)
        yield return x;
  }
```

Obliczenie każdego elementu zostaje odłożone do ostatniej chwili — gdy nadejdzie żądanie w postaci wywołania `MoveNext()`. Na rysunku 4.1 pokazano przebieg żądań danych i operacji ich zwracania w czasie.



Rysunek 4.1. Komponowanie sekwencji

Możliwość komponowania iteratorów jest bardzo przydatna w LINQ. Szerzej tym tematem zajmujemy się w rozdziale 8.

Typy wartościowe dopuszczające wartość null

Typy referencyjne mogą reprezentować nieistniejące wartości za pomocą pustej referencji. Natomiast typy wartościowe normalnie nie mają takiej możliwości. Na przykład:

```
string s = null;    // OK, typ referencyjny
int i = null;       // błąd kompilacji, typ wartościowy nie może być null
```

Do reprezentacji wartości `null` w typie wartościowym służy specjalna konstrukcja zwana **typem dopuszczającym wartość `null`** (ang. *nullable type*). Taki typ oznacza się za pomocą znaku `?` za nazwą typu wartościowego:

```
int? i = null;                  // OK, typ dopuszczający wartość null
Console.WriteLine (i == null);  // prawda
```


Kup ksi k   Pole ksi k

Struktura Nullable<T>

Parametr `T?` zamienia się w `System.Nullable<T>`. Jest to lekka niezmienna struktura zawierająca tylko dwa pola: `Value` i `HasValue`. Struktura ta zasadniczo jest bardzo prosta:

```
public struct Nullable<T> where T : struct
{
  public T Value {get;}
  public bool HasValue {get;}
  public T GetValueOrDefault();
  public T GetValueOrDefault (T defaultValue);
  ...
}
```

Kod:

```
int? i = null;
Console.WriteLine (i == null);        // prawda
```

zostanie zamieniony na:

```
Nullable<int> i = new Nullable<int>();
Console.WriteLine (! i.HasValue);     // prawda
```

Próba pobrania wartości `Value`, gdy `HasValue` ma wartość `false`, skutkuje wyjątkiem `InvalidOpe` ↪`rationException`. Metoda `GetValueOrDefault()` zwraca `Value`, jeśli `HasValue` ma wartość `true`. W przeciwnym przypadku zwraca `new T()` lub określoną wartość domyślną.

Wartość domyślna `T?` to `null`.

Jawne i niejawne konwersje typów dopuszczających wartość null

Konwersja z `T` na `T?` odbywa się niejawnie, a `T?` na `T` jest jawna. Na przykład:

```
int? x = 5;       // niejawne
int y = (int)x;   // jawne
```

Rzutowanie jawne jest wprost równoważne z wywołaniem własności `Value` obiektu dopuszczającego wartość `null`. Dlatego jeśli `HasValue` ma wartość `false`, następuje zgłoszenie wyjątku `InvalidOperationException`.

Pakowanie i rozpakowywanie wartości typów dopuszczających wartość null

Gdy wartość typu `T?` zostanie opakowana, to opakowana wartość na stercie zawiera `T`, nie `T?`. Taka optymalizacja jest możliwa dlatego, że wartość opakowana jest typu referencyjnego, który może reprezentować wartość `null`.

Ponadto w języku C# istnieje możliwość rozpakowywania typów dopuszczających wartość `null` za pomocą operatora `as`. W razie niepowodzenia operacji rzutowania wynik będzie `null`:

```
object o = "łańcuch";
int? x = o as int?;
Console.WriteLine (x.HasValue); // fałsz
```


Kup ksi k Pole ksi k

Pożyczanie operatorów

Struktura `Nullable<T>` nie definiuje takich operatorów jak `<`, `>` ani nawet `==`. Mimo to poniższy kod przejdzie kompilację i zostanie wykonany prawidłowo:

```
int? x = 5;
int? y = 10;
bool b = x < y; // true
```

Jest to możliwe dzięki temu, że kompilator pożycza (ang. *lift*) operator mniejszości od podstawowego typu wartościowego. Pod względem semantycznym powyższe wyrażenie porównywania zostanie przekształcone na taką postać:

```
bool b = (x.HasValue && y.HasValue) ? (x.Value < y.Value) : false;
```

Innymi słowy: jeśli zarówno `x`, jak i `y` mają wartości, do porównywania użyty zostaje operator mniejszości typu `int`. W przeciwnym razie zostaje zwrócona wartość `false`.

Pożyczanie operatorów to technika, dzięki której można niejawnie używać operatorów typu `T` na wartościach typu `T?`. W razie potrzeby można zdefiniować specjalne operatory dla `T?`, ale w ogromnej większości przypadków najlepiej jest polegać na systemowej logice zapewnianej automatycznie przez kompilator. Oto kilka przykładów:

```
int? x = 5;
int? y = null;

// przykłady użycia operatora równości
Console.WriteLine (x == y);    // fałsz
Console.WriteLine (x == null); // fałsz
Console.WriteLine (x == 5);    // prawda
Console.WriteLine (y == null); // prawda
Console.WriteLine (y == 5);    // fałsz
Console.WriteLine (y != 5);    // prawda

// przykłady użycia operatorów relacyjnych
Console.WriteLine (x < 6);     // prawda
Console.WriteLine (y < 6);     // fałsz
Console.WriteLine (y > 6);     // fałsz

// przykłady użycia innych operatorów
Console.WriteLine (x + 5);     // 10
Console.WriteLine (x + y);     // null (drukuje pusty wiersz)
```

Kompilator stosuje różną logikę w odniesieniu do wartości `null` w zależności od rodzaju operatora. W poniższych sekcjach zamieściliśmy opis tych zasad.

Operatory równości (== i !=)

Pożyczone operatory równości traktują wartości `null` tak, jak robią to typy referencyjne. Oznacza to, że dwie wartości `null` są sobie równe:

```
Console.WriteLine ( null == null);               // prawda
Console.WriteLine ((bool?)null == (bool?)null);  // prawda
```


Kup ksi k   Pole ksi k

Ponadto:

- Jeśli dokładnie jeden argument jest `null`, argumenty są różne.
- Jeśli oba argumenty są różne od `null`, porównywane są ich wartości `Value`.

Operatory relacyjne (<, <=, >=, >)

Operatory relacyjne działają wg zasady, że porównywanie argumentów `null` nie ma sensu. W związku z tym wynikiem porównania wartości `null` z wartością `null` lub inną jest `false`:

```
bool b = x < y; // tłumaczenie:

bool b = (x.HasValue && y.HasValue)
          ? (x.Value < y.Value)
          : false;

// b ma wartość false (przy założeniu, że x wynosi 5, a y ma wartość null)
```

Pozostałe operatory (+, –, *, /, %, &, |, ^, <<, >>, +, ++, --, !, ~)

Te operatory zwracają wartość `null`, gdy którykolwiek z argumentów ma wartość `null`. Wzorzec ten powinien wyglądać znajomo dla użytkowników języka SQL:

```
int? c = x + y; // tłumaczenie:

int? c = (x.HasValue && y.HasValue)
          ? (int?) (x.Value + y.Value)
          : null;
// c ma wartość null (przy założeniu, że x wynosi 5, a y ma wartość null)
```

Wyjątkiem jest sytuacja, gdy operatory `&` i `|` zostaną zastosowane do wartości typu `bool?`, ale do tego wrócimy za chwilę.

Mieszanie operatorów dopuszczających wartość null ze zwykłymi

Typy dopuszczające wartość `null` można dowolnie mieszać ze zwykłymi typami (jest to możliwe dzięki niejawnej konwersji `T` na `T?`):

```
int? a = null;
int b = 2;
int? c = a + b; // c jest null — równoznaczne z a + (int?)b
```

Operatory & i | z typem bool?

Jeśli operatorom `&` i `|` przekaże się argumenty typu `bool?`, to traktują one `null` jako **wartość nieznaną**. Zatem wynikiem operacji `null | true` jest `true`, ponieważ:

- Jeśli nieznana wartość jest fałszywa, wynikiem jest `true`.
- Jeśli nieznana wartość jest prawdziwa, wynikiem jest `true`.

Kup ksi k Pole ksi k

Analogicznie `null & false` ma wartość `false`. Zasady te są z pewnością znane użytkownikom języka SQL. Poniżej znajduje się lista innych kombinacji:

```
bool? n = null;
bool? f = false;
bool? t = true;
Console.WriteLine (n | n); // (null)
Console.WriteLine (n | f); // (null)
Console.WriteLine (n | t); // prawda
Console.WriteLine (n & n); // (null)
Console.WriteLine (n & f); // fałsz
Console.WriteLine (n & t); // (null)
```

Typy wartościowe dopuszczające wartość null i operatory null

Typy wartościowe dopuszczające wartość `null` szczególnie dobrze współpracują z operatorem `??` (zob. podrozdział „Operatory null" w rozdziale 2.). Na przykład:

```
int? x = null;
int y = x ?? 5; // wartość y wynosi 5

int? a = null, b = 1, c = 2;
Console.WriteLine (a ?? b ?? c); // 1 (pierwsza wartość różna od null)
```

Użycie operatora `??` w odniesieniu do typu wartościowego dopuszczającego wartość `null` jest tożsame z wywołaniem metody `GetValueOrDefault` z jawną domyślną wartością, z tym wyjątkiem, że wyrażenie wartości domyślnej nie jest obliczane, jeśli zmienna nie jest `null`.

Typy wartościowe dopuszczające wartość `null` dobrze współpracują też z operatorem warunkowym `null` (zob. sekcję „Operator warunkowy null" w rozdziale 2.). W poniższym przykładzie zmienna `length` będzie miała wartość `null`:

```
System.Text.StringBuilder sb = null;
int? length = sb?.ToString().Length;
```

Możemy dołączyć operator sprawdzania `null`, aby zamiast `null` otrzymywać wartość zero:

```
int length = sb?.ToString().Length ?? 0; // wynikiem jest 0, jeśli sb ma wartość null
```

Zastosowania typów wartościowych dopuszczających wartość null

Jednym z najpowszechniejszych zastosowań typów wartościowych dopuszczających wartość `null` jest reprezentowanie nieznanych wartości. Często robi się to przy pracy z bazami danych, gdy trzeba zmapować klasę na tabelę z kolumnami mogącymi zawierać wartości `null`. Jeżeli wartości w tych kolumnach są łańcuchami (np. kolumna `EmailAddress` w tabeli `Customer`), to nie ma problemu, ponieważ `string` w CLR jest typem referencyjnym, a więc może reprezentować `null`. Ale większość typów kolumn SQL odpowiada typom strukturalnym CLR, w związku z czym typy wartościowe dopuszczające wartość `null` są szczególnie przydatne przy mapowaniu SQL na CLR. Na przykład:

```
// mapowanie na tabelę Customer w bazie danych
public class Customer
{
  ...
  public decimal? AccountBalance;
}
```


Kup ksi k   Pole ksi k

Za pomocą typu dopuszczającego wartość `null` można także reprezentować pole zapasowe tzw. **własności otoczenia** (ang. *ambient property*). Własność taka, jeśli ma wartość `null`, zwraca wartość swojego rodzica. Na przykład:

```
public class Row
{
  ...
  Grid parent;
  Color? color;

  public Color Color
  {
    get { return color ?? parent.Color; }
    set { color = value == parent.Color ? (Color?)null : value; }
  }
}
```

Alternatywa dla typów wartościowych dopuszczających wartość null

Zanim wprowadzono typy wartościowe dopuszczające wartość `null` do języka C# (tj. przed wersją C# 2.0), radzono sobie z problemem reprezentowania tej wartości za pomocą typów wartościowych na różne sposoby, których przykłady z powodów historycznych wciąż można znaleźć w kodzie platformy .NET. Jedną z tych strategii jest wyznaczenie wartości innej niż `null` do reprezentowania wartości `null` (przykłady można znaleźć w klasach łańcuchów i tablic). Operator `String.IndexOf` zwraca magiczną wartość `-1`, gdy nie znajdzie określonego znaku:

```
int i = "Różowy".IndexOf ('b');
Console.WriteLine (i); // −1
```

Natomiast operator `Array.IndexOf` zwraca `-1` tylko wtedy, gdy indeksowanie zaczyna się od zera. Bardziej ogólnie: operator `IndexOf` zwraca wartość o jeden mniejszą niż najmniejszy możliwy indeks tablicy. W poniższym przykładzie opisywany operator zwróci `0`, jeśli nie znajdzie szukanego elementu:

```
// utworzenie tablicy, której najniższy indeks to 1, a nie 0:

Array a = Array.CreateInstance (typeof (string),
                                new int[] {2}, new int[] {1});
a.SetValue ("a", 1);
a.SetValue ("b", 2);
Console.WriteLine (Array.IndexOf (a, "c")); // 0
```

Wyznaczenie „magicznej wartości” sprawia problemy z kilku powodów:

- Każdy typ wartościowy ma inną reprezentację wartości `null`. Natomiast wszystkie typy wartościowe dopuszczające wartość `null` obsługują jeden wspólny wzorzec odpowiedni dla wszystkich.
- Może nie być sensownej wartości do wyznaczenia. W poprzednim przykładzie wartość `-1` nie zawsze może być użyta. To samo dotyczy wcześniejszego przykładu dotyczącego reprezentacji nieznanego salda konta.

Kup ksi k   Pole ksi k

- Jeśli programista zapomni o teście wartości magicznej, może powstać niepoprawna wartość, która może pozostać niezauważona przez dłuższy czas, aż przyczyni się do wykonania jakiejś niezamierzonej magicznej sztuczki. Natomiast brak testu wartości `HasValue` na wartości `null` powoduje zgłoszenie wyjątku `InvalidOperationException`.
- Możliwość reprezentowania przez typ wartości `null` nie jest cechą samego *typu*. Typy komunikują intencje programu, pozwalają kompilatorowi sprawdzić poprawność kodu oraz określają spójny zestaw reguł, których egzekwowaniem zajmuje się kompilator.

Typy referencyjne dopuszczające wartość null

Podczas gdy **typy wartościowe dopuszczające wartość `null`** (C# 8) umożliwiają stosowanie tej wartości typom wartościowym, **typy referencyjne dopuszczające wartość `null`** działają odwrotnie, tzn. umożliwiają **ograniczenie stosowania wartości `null`** w celu uniknięcia wyjątków `NullReferenceException`.

Typy referencyjne dopuszczające wartość `null` wprowadzają dodatkowy stopień bezpieczeństwa, egzekwowany wyłącznie przez kompilator, który ostrzega programistę, jeśli odkryje w kodzie ryzyko wystąpienia wyjątku `NullReferenceException`.

Aby móc korzystać z typów referencyjnych dopuszczających wartość `null`, należy dodać element `Nullable` do pliku *.csproj* projektu (jeśli możliwość ta ma obejmować cały projekt):

```
<PropertyGroup>
    <Nullable>enable</Nullable>
</PropertyGroup>
```

i/lub stosować następujące dyrektywy w swoim kodzie w miejscach, w których efekt ten ma być dostępny:

```
#nullable enable   // włącza typy referencyjne dopuszczające wartość null od tego miejsca
#nullable disable  // włącza typy referencyjne dopuszczające wartość null od tego miejsca
#nullable restore  // przywraca ustawienie projektu dotyczące typów referencyjnych dopuszczających wartość null
```

Po włączeniu tej funkcjonalności kompilator domyślnie przyjmuje, że wartość `null` jest niedozwolona: jeśli programista chce, aby typ referencyjny ją akceptował, i kompilator nie generował ostrzeżenia, musi dodać przyrostek ?. W poniższym przykładzie `s1` nie dopuszcza wartości `null`, a `s2` — tak:

```
#nullable enable    // Włącza typy referencyjne dopuszczające wartość null.
string s1 = null;   // Kompilator zgłosi ostrzeżenie!
string? s2 = null;  // OK: s2 jest typem referencyjnym dopuszczającym wartość null.
```

Jako że typy referencyjne dopuszczające wartość `null` są konstrukcjami analizowanymi w czasie kompilacji, w trakcie wykonywania między `string` a `string?` nie ma żadnej różnicy. Natomiast typy wartościowe dopuszczające wartość `null` wprowadzają do systemu typów konkretną konstrukcję — strukturę `Nullable<T>`.

Poniższy kod spowoduje wygenerowanie ostrzeżenia, ponieważ zmienna `x` nie jest zainicjalizowana:

```
class Foo { string x; }
```

Ostrzeżenie to zniknie, kiedy zainicjalizujemy `x` za pomocą inicjalizatora pól lub kodu w konstruktorze.

Kup ksi k   Pole ksi k

Operator ignorowania null

Kompilator ostrzeże programistę także w przypadku, gdy uzna podczas dereferencji typu referencyjnego dopuszczającego wartość `null`, że może wystąpić wyjątek `NullReferenceException`. W poniższym przykładzie użycie własności `Length` łańcucha spowoduje wygenerowanie ostrzeżenia:

```
void Foo (string? s) => Console.Write (s.Length);
```

Można się go pozbyć za pomocą *operatora ignorowania* `null` (`!`):

```
void Foo (string? s) => Console.Write (s!.Length);
```

Użycie operatora ignorowania null w powyższym przykładzie jest niebezpieczne ze względu na to, że może dojść do zgłoszenia wyjątku `NullReferenceException`, którego próbujemy uniknąć. Możemy to naprawić tak:

```
void Foo (string? s)
{
    if (s != null) Console.Write (s.Length);
}
```

Zwróć uwagę, że nie potrzebujemy operatora ignorowania `null`, ponieważ kompilator przeprowadza *statyczną analizę przepływu* sterowania i potrafi — przynajmniej w prostych przypadkach — stwierdzić, kiedy dereferencja nie stwarza ryzyka powstania wyjątku `NullReferenceException`.

Możliwości kompilatora w zakresie wykrywania i ostrzegania są ograniczone. Nie można np. liczyć, że kompilator będzie wiedział, że elementy tablicy zostały zapełnione, w związku z czym poniższy kod nie spowoduje wygenerowania ostrzeżenia:

```
var strings = new string[10];
Console.WriteLine (strings[0].Length);
```

Rozdzielanie kontekstów adnotacji i ostrzeżeń

Włączenie typów referencyjnych dopuszczających wartość `null` za pomocą dyrektywy `#nullable enable` (lub ustawienia projektu `<Nullable>enable</Nullable>`) wywołuje dwa skutki:

- Włącza **kontekst adnotacji w zakresie `null`**, w którym kompilator traktuje wszystkie deklaracje zmiennych typu referencyjnego jako niedopuszczające wartości `null`, chyba że mają przyrostek ?.
- Włącza **kontekst ostrzegania w zakresie `null`**, w którym kompilator generuje ostrzeżenia, gdy napotka kod grożący ryzykiem zgłoszenia wyjątku `NullReferenceException`.

Czasami dobrze jest rozdzielić te dwa konteksty, tzn. włączyć tylko kontekst adnotacji (mniej praktyczne) lub tylko kontekst ostrzeżeń:

```
#nullable enable annotations // Włącza kontekst adnotacji.
// LUB:
#nullable enable warnings // Włącza kontekst ostrzeżeń.
```

(To samo można zrobić z dyrektywami `#nullable disable` i `#nullable restore`).

Kup ksi k Pole ksi k

W pliku projektu należy zastosować następujące ustawienia:

```
<Nullable>annotations</Nullable>
<!-- LUB -->
<Nullable>warnings</Nullable>
```

Włączenie samego kontekstu adnotacji dla konkretnej klasy lub konkretnego zestawu może być dobrym pierwszym krokiem do wprowadzenia typów referencyjnych dopuszczających wartość `null` do starego kodu. Za pomocą odpowiednich adnotacji składowych publicznych można sprawić, że klasa lub zestaw będą poprawnie współpracować z innymi klasami lub zestawami — w pełni korzystając z dobrodziejstw typów referencyjnych dopuszczających wartość `null` — bez zgłaszania ostrzeżeń w naszych własnych klasach lub zestawach.

Traktowanie ostrzeżeń dotyczących null jako błędów

W nowych projektach dobrym pomysłem jest włączenie pełnego kontekstu `null` od samego początku. Można nawet przyjąć regułę traktowania wszystkich ostrzeżeń dotyczących wartości `null` jako błędów, która uniemożliwi kompilację projektu, dopóki wszystkie te ostrzeżenia nie zostaną zlikwidowane:

```
<PropertyGroup>
    <Nullable>enable</Nullable>
    <WarningsAsErrors>CS8600;CS8602;CS8603</WarningsAsErrors>
</PropertyGroup>
```

Metody rozszerzające

Metody rozszerzające umożliwiają rozszerzanie istniejących typów o nowe metody bez zmieniania pierwotnej definicji tych typów. Metoda rozszerzająca jest statyczną metodą statycznej klasy, gdzie modyfikator `this` jest stosowany do pierwszego parametru. Typ pierwszego parametru będzie tym typem, który zostanie rozszerzony. Na przykład:

```
public static class StringHelper
{
  public static bool IsCapitalized (this string s)
  {
    if (string.IsNullOrEmpty(s)) return false;
    return char.IsUpper (s[0]);
  }
}
```

Metoda rozszerzająca `IsCapitalized` może być wywoływana na łańcuchach tak, jakby była metodą egzemplarzową, np.:

```
Console.WriteLine ("Perth".IsCapitalized());
```

W procesie kompilacji wywołanie metody rozszerzającej jest zamieniane z powrotem na zwykłe wywołanie metody statycznej:

```
Console.WriteLine (StringHelper.IsCapitalized ("Perth"));
```


Kup ksi k Pole ksi k

Translacja ta odbywa się następująco:

```
arg0.Method (arg1, arg2, ...);              // wywołanie metody rozszerzającej
StaticClass.Method (arg0, arg1, arg2, ...); // wywołanie metody statycznej
```

Interfejsy także można rozszerzać:

```
public static T First<T> (this IEnumerable<T> sequence)
{
  foreach (T element in sequence)
    return element;

  throw new InvalidOperationException ("Brak elementów!");
}
...
Console.WriteLine ("Seattle".First()); // S
```

Łańcuchowe wywoływanie metod rozszerzających

Metody rozszerzające, podobnie jak metody egzemplarzy, można wygodnie wywoływać łańcuchowo. Spójrz na dwie poniższe funkcje:

```
public static class StringHelper
{
  public static string Pluralize (this string s) {...}
  public static string Capitalize (this string s) {...}
}
```

Zmienne `x` i `y` są równoważne i wartość każdej z nich to `"sausages"`, ale wartość `x` została obliczona przy użyciu metod rozszerzających, a `y` — metod statycznych:

```
string x = "sausage".Pluralize().Capitalize();
string y = StringHelper.Capitalize (StringHelper.Pluralize ("sausage"));
```

Postępowanie w niejednoznacznych przypadkach

Przestrzenie nazw

Z metody rozszerzającej można korzystać tylko wtedy, gdy klasa, do której metoda należy, jest w zasięgu dostępności, najczęściej dzięki zaimportowaniu odpowiedniej przestrzeni nazw. Spójrz na metodę rozszerzającą `IsCapitalized` w poniższym przykładzie:

```
using System;

namespace Utils
{
  public static class StringHelper
  {
    public static bool IsCapitalized (this string s)
    {
      if (string.IsNullOrEmpty(s)) return false;
      return char.IsUpper (s[0]);
    }
  }
}
```


Kup ksi k Pole ksi k

Aby można było użyć metody `IsCapitalized` w poniższej aplikacji, należy zaimportować przestrzeń nazw `Utils`. Jeśli się tego nie zrobi, wystąpi błąd kompilacji:

```
namespace MyApp
{
  using Utils;

  class Test
  {
    static void Main() => Console.WriteLine ("Perth".IsCapitalized());
  }
}
```

Metody rozszerzające a metody egzemplarzowe

Jeśli istnieje odpowiednia metoda egzemplarzowa, to zawsze będzie miała ona pierwszeństwo przed metodą rozszerzającą. W poniższym przykładzie zawsze będzie używana metoda `Foo` klasy `Test`, nawet jeśli argumentem wywołania będzie `x` typu `int`:

```
class Test
{
  public void Foo (object x) { } // ta metoda będzie zawsze wybierana
}

static class Extensions
{
  public static void Foo (this Test t, int x) { }
}
```

W tym przypadku jedynym sposobem na wywołanie metody rozszerzającej jest użycie normalnej składni statycznej, tzn. `Extensions.Foo(...)`.

Metody rozszerzające kontra metody rozszerzające

Jeśli dwie metody rozszerzające mają taką samą sygnaturę, to jedna z nich musi być wywoływana jako zwykła metoda statyczna, aby było jasne, o którą chodzi. Jeżeli jednak jedna metoda ma bardziej specyficzne argumenty, to wybrana zostanie właśnie ona.

W ramach przykładu spójrz na dwie poniższe klasy:

```
static class StringHelper
{
  public static bool IsCapitalized (this string s) {...}
}
static class ObjectHelper
{
public static bool IsCapitalized (this object s) {...}
}
```

Poniższa instrukcja wywołuje metodę `IsCapitalized` klasy `StringHelper`:

```
bool test1 = "Perth".IsCapitalized();
```

Klasy i struktury są uważane za bardziej specyficzne niż interfejsy.

Kup ksi k Pole ksi k

Typy anonimowe

Typ anonimowy to prosta klasa tworzona przez kompilator w locie w celu przechowywania zestawu wartości. Typy anonimowe tworzy się za pomocą słowa kluczowego `new`, a po nim wpisuje się inicjalizator obiektu zawierający własności i wartości, które dany typ anonimowy ma zawierać. Na przykład:

```
var dude = new { Name = "Bartek", Age = 23 };
```

Kompilator przetworzy taki kod na mniej więcej taką postać:

```
internal class AnonymousGeneratedTypeName
{
  private string name; // rzeczywista nazwa pola nie ma znaczenia
  private int age;     // rzeczywista nazwa pola nie ma znaczenia

  public AnonymousGeneratedTypeName (string name, int age)
  {
    this.name = name; this.age = age;
  }

  public string Name { get { return name; } }
  public int    Age  { get { return age;  } }

  // metody Equals i GetHashCode są przesłonięte (zob. rozdział 6.)
  // metoda ToString też jest przesłonięta
}
...
var dude = new AnonymousGeneratedTypeName ("Bartek", 23);
```

Do typu anonimowego należy odnosić się przy użyciu słowa kluczowego `var`, ponieważ ten typ nie ma nazwy.

Nazwy własności typu anonimowego mogą zostać wydedukowane z wyrażenia, które samo w sobie jest identyfikatorem (lub się nim kończy). Na przykład:

```
int Age = 23;
var dude = new { Name = "Bartek", Age, Age.ToString().Length };
```

Ten kod jest równoważny z tym:

```
var dude = new { Name = " Bartek", Age = Age, Length = Age.ToString().Length };
```

Dwa egzemplarze typu anonimowego zadeklarowane w jednym zestawie mają ten sam typ podstawowy, jeśli zawierają elementy o takich samych nazwach i typach:

```
var a1 = new { X = 2, Y = 4 };
var a2 = new { X = 2, Y = 4 };
Console.WriteLine (a1.GetType() == a2.GetType()); // prawda
```

Ponadto metoda `Equals` zostaje przesłonięta, aby można było wykonywać operacje *porównywania strukturalnego* (danych):

```
Console.WriteLine (a1.Equals (a2)); // prawda
```

Operator równości (==) wykonuje zaś porównywanie referencyjne:

```
Console.WriteLine (a1 == a2); // fałsz
```


Kup ksi k Pole ksi k

Tablice typów anonimowych można tworzyć tak:

```
var dudes = new[]
{
  new { Name = "Bartek", Age = 30 },
  new { Name = "Tomek", Age = 40 }
};
```

Metoda nie może (tak, aby to było przydatne) zwrócić obiektu o anonimowym typie, ponieważ nie można napisać metody o typie zwrotnym `var`:

```
var Foo() => new { Name = "Bartek", Age = 30 }; // niedozwolone!
```

Zamiast tego należy użyć `object` lub `dynamic` i wówczas ten, kto wywoła `Foo` będzie musiał polegać na wiązaniu dynamicznym, licząc się z utratą statycznego bezpieczeństwa typów (i funkcji IntelliSense w Visual Studio):

```
dynamic Foo() => new { Name = "Bartek", Age = 30 }; // brak statycznego bezpieczeństwa typów
```

Typów anonimowych najczęściej używa się w zapytaniach LINQ (zob. rozdział 8.).

Krotki

Podobnie jak typy anonimowe, krotki służą do łatwego przechowywania zbiorów wartości. Ich podstawowym przeznaczeniem jest bezpieczne zwracanie licznych wartości z metod bez użycia parametrów wyjściowych (tego nie da się zrobić przy użyciu typów anonimowych).

Krotki robią prawie wszystko to, co typy anonimowe, i mają jeszcze kilka dodatkowych funkcji. Jedną z ich wad — jak wkrótce się przekonasz — jest wymazywanie typów w czasie działania programu z elementami nazwanymi.

Najprostszym sposobem na utworzenie **literału krotki** jest wypisanie wszystkich potrzebnych wartości w nawiasie. W ten sposób tworzy się krotki z elementami bez nazwy, do których można odwoływać się wg wzoru `Item1`, `Item2` itd.:

```
var bob = ("Bartek", 23);          // kompilator wydedukuje typy elementów
Console.WriteLine (bob.Item1);     // Bartek
Console.WriteLine (bob.Item2);     // 23
```

Krotki to typy wartościowe zawierające modyfikowalne elementy (z możliwością odczytu i zapisu):

```
var joe = bob;              // joe jest *kopią* boba
joe.Item1 = "Jacek";        // zmiana elementu Item1 krotki joe z Bartek na Jacek
Console.WriteLine (bob);    // (Bartek, 23)
Console.WriteLine (joe);    // (Jacek, 23)
```

W odróżnieniu od typów anonimowych typ krotki można wprost określić. W tym celu wystarczy podać listę typów wszystkich elementów w nawiasie:

```
(string,int) bob = ("Bartek", 23); // można też użyć var
```

Oznacza to, że krotka zwrócona przez metodę może być przydatna:

```
(string,int) person = GetPerson(); // można tu użyć var
Console.WriteLine (person.Item1);  // Bartek
Console.WriteLine (person.Item2);  // 23

(string,int) GetPerson() => ("Bartek", 23);
```


Kup ksi k   Pole ksi k

Krotki bardzo dobrze współpracują z typami generycznymi, więc poniższe typy są prawidłowe:

```
Task<(string,int)>
Dictionary<(string,int),Uri>
IEnumerable<(int ID, string Name)> // sposoby nadawania nazw elementom opisaliśmy poniżej
```

Nadawanie nazw elementom krotek

Podczas tworzenia literału krotki jego elementom można nadać nazwy:

```
var tuple = (Name:"Bartek", Age:23);
Console.WriteLine (tuple.name);    // Bartek
Console.WriteLine (tuple.age);     // 23
```

To samo można zrobić przy określaniu typów krotki:

```
var person = GetPerson();
Console.WriteLine (person.name); // Bartek
Console.WriteLine (person.age);  // 23

(string name, int age) GetPerson() => ("Bartek", 23);
```

Do elementów mających zdefiniowane nazwy nadal można odwoływać się za pomocą standardowych nazw `Item1`, `Item2` itd. (chociaż Visual Studio usuwa te pola z IntelliSense).

Nazwy elementów są automatycznie dedukowane na podstawie nazw własności lub pól:

```
var now = DateTime.Now;
var tuple = (now.Day, now.Month, now.Year);
Console.WriteLine (tuple.Day); // OK
```

Krotki pasują do siebie typami, jeśli ich elementy do siebie pasują (w kolejności występowania). Nazwy elementów mogą być różne:

```
(string name, int age, char sex) bob1 = ("Bartek", 23, 'M');
(string age, int sex, char name) bob2 = bob1;            // to nie błąd!
```

Ten przykład ujawnia potencjalne źródło nieporozumień:

```
Console.WriteLine (bob2.name); // M
Console.WriteLine (bob2.age);  // Bartek
Console.WriteLine (bob2.sex);  // 23
```

Wymazywanie typów

Wcześniej napisaliśmy, że kompilator C# obsługuje typy anonimowe poprzez tworzenie dla każdego z elementów własnych klas zawierających nazwane własności. Z krotkami obchodzi się inaczej, wykorzystując gotową rodzinę generycznych struktur:

```
public struct ValueTuple<T1>
public struct ValueTuple<T1,T2>
public struct ValueTuple<T1,T2,T3>
...
```

Każda struktura `ValueTuple<>` zawiera pola o nazwach `Item1`, `Item2` itd.

Dlatego `(string,int)` jest aliasem dla `ValueTuple<string,int>`, co z kolei oznacza, że w podstawowych typach nazwanym elementom krotek nie odpowiadają żadne nazwy własności. Nazwy te istnieją tylko w kodzie źródłowym i „wyobraźni” kompilatora. W czasie działania programu

Kup ksi k Pole ksi k

większość nazw znika, więc po dekompilacji w konstrukcjach odwołujących się do krotek najczęściej można znaleźć odwołania do elementów `Item1`, `Item2` itd. Ponadto, jeśli w debuggerze zbadamy zmienną krotki, która została przypisana do obiektu (lub wykonamy jej zrzut w LINQPad), nie znajdziemy nazw elementów. Na dodatek w większości przypadków za pomocą refleksji (rozdział 18.) nie uda się nam odkryć nazw elementów krotki w czasie działania programu.

Napisaliśmy, że znika *większość* nazw, ponieważ jest jeden wyjątek. W przypadku metod i własności zwracających nazwane typy krotek kompilator emituje nazwy elementów, stosując specjalny atrybut o nazwie `TupleElementNamesAttribute` (zobacz podrozdział „Atrybuty") do typu zwrotnego składowej. To umożliwia działanie nazwanych elementów także w przypadku wywoływania metod w innym zestawie (którego kod źródłowy dla kompilatora jest niedostępny).

ValueTuple.Create

Krotki można też tworzyć za pomocą metody fabrycznej należącej do niegenerycznego typu `ValueTuple`:

```
ValueTuple<string,int> bob1 = ValueTuple.Create ("Bartek", 23);
(string,int)           bob2 = ValueTuple.Create ("Bartek", 23);
(string name, int age) bob3 = ValueTuple.Create ("Bartek", 23);
```

Dekonstruowanie krotek

Krotki obsługują wzorzec dekonstrukcji (zobacz „Dekonstruktory" w rozdziale 3.), dzięki czemu każdą krotkę z łatwością można rozebrać na indywidualne zmienne. Spójrz na poniższy przykład:

```
var bob = ("Bartek", 23);

string name = bob.Item1;
int age = bob.Item2;
```

Dzięki dekonstruktorowi krotki ten kod można uprościć następująco:

```
var bob = ("Bartek", 23);
(string name, int age) = bob; // dekonstrukcja krotki bob na
                              // indywidualne zmienne (name i age)
Console.WriteLine (name);
Console.WriteLine (age);
```

Składnia dekonstrukcji jest niebezpiecznie podobna do składni deklaracji krotek z nazwanymi elementami! Ilustruje to poniższy przykład:

```
(string name, int age) = bob;      // dekonstrukcja krotki
(string name, int age) bob2 = bob; // deklaracja nowej krotki
```

Oto kolejny przykład, tym razem z wywołaniem metody i inferencją typów (`var`):

```
var (name, age, sex) = GetBob();
Console.WriteLine (name);      // Bartek
Console.WriteLine (age);       // 23
Console.WriteLine (sex);       // M

string, int, char) GetBob() => ( "Bartek", 23, 'M');
```


Kup ksi k Pole ksi k

Dekonstrukcję można wykonać także do pól i własności, co jest wygodnym skrótem do wypełniania wielu pól lub własności w konstruktorze:

```
class Point
{
    public readonly int X, Y;
    public Point (int x, int y) => (X, Y) = (x, y);
}
```

Porównywanie

Podobnie jak typy anonimowe, typy `ValueTuple<>` przesłaniają metodę `Equals`, aby możliwe było dokonywanie porównań. To znaczy, że porównuje ona dane, a nie **referencję**:

```
var t1 = ("jeden", 1);
var t2 = ("jeden", 1);
Console.WriteLine (t1.Equals (t2)); // prawda
```

Ponadto `ValueTuple<>` przeciąża operatory `==` i `!=`:

```
Console.WriteLine (t1 == t2); // True (od C# 7.3)
```

Krotki przeciążają też metodę `GetHashCode`, dzięki czemu znajdują także zastosowanie jako klucze w słownikach. Szerzej na temat porównywania piszemy w rozdziale 6., a o słownikach — w rozdziale 7.

Ponadto typy `ValueTuple<>` implementują interfejs `IComparable` (zobacz „Porównywanie” w rozdziale 6.), dzięki czemu krotek można używać jako kluczy sortowania.

Klasy System.Tuple

W przestrzeni nazw System znajduje się jeszcze jedna rodzina typów generycznych, o nazwie `Tuple` (nie `ValueTuple`). Typy te wprowadzono w 2010 r. i są to klasy (podczas gdy typy `ValueTuple` to struktury). Kiedyś definiowanie krotek jako klas uważano za błąd: w typowych przypadkach użycia krotek struktury mają niewielką przewagę wydajnościową (ponieważ pozwalają uniknąć niepotrzebnych alokacji pamięci), a jednocześnie praktycznie żadnych wad. Dlatego kiedy firma Microsoft zdecydowała się wprowadzić obsługę krotek w C# 7, to zignorowała istniejące typy `Tuple` na rzecz nowego typu `ValueTuple`. Niemniej jednak w kodzie powstałym przed pojawieniem się C# 7 wciąż można spotkać stare typy. Nie mają one żadnego wsparcia ze strony języka, a używa się ich w następujący sposób:

```
Tuple<string,int> t = Tuple.Create ("Bartek", 23); // metoda fabryczna
Console.WriteLine (t.Item1); // Bartek
Console.WriteLine (t.Item2); // 23
```

Rekordy (C# 9)

Rekord to specjalny rodzaj klasy przeznaczony do pracy z danymi niezmiennymi (tylko do odczytu). Jego najbardziej przydatną cechą jest **mutacja niedestrukcyjna**. Poza tym rekordy są przydatne do tworzenia typów, które łączą lub przechowują dane. W prostych przypadkach pozwalają wyeliminować szablonowy kod przy zachowaniu semantyki równości najbardziej odpowiedniej dla typów niezmiennych.

Kup ksi k Pole ksi k

Rekord to czysto kompilacyjna konstrukcja języka C#. W środowisku wykonawczym dla CLR są zwykłymi klasami (z kilkoma dodatkowymi składowymi dodanymi przez kompilator).

Informacje ogólne

Pisanie niezmiennych klas (których pól nie można modyfikować po inicjalizacji) to popularna strategia upraszczania programu i zmniejszania liczby błędów. Stanowi też podstawowy aspekt programowania funkcyjnego, w którym unika się możliwości modyfikowania stanów, a funkcje traktuje jak dane. Technologia LINQ opiera się na tej zasadzie.

Aby „zmodyfikować" niezmienny obiekt, należy utworzyć nowy obiekt i skopiować dane, wprowadzając swoje zmiany (nazywa się to **mutacją niedestrukcyjną**). Nie jest to wcale tak nieefektywne, jak się może wydawać, ponieważ zawsze wystarcza utworzenie **płytkiej kopii** (**kopia głęboka**, obejmująca także podobiekty i kolekcje, jest zbędna w przypadku niezmiennych danych). Natomiast z punktu widzenia kodowania niedestrukcyjna mutacja może być bardzo nieefektywna, zwłaszcza gdy własności jest dużo. Rekordy rozwiązują ten problem za pomocą wzorca obsługiwanego przez język.

Innym problemem jest to, że programiści — szczególnie używający języków *funkcyjnych* — czasami korzystają z niezmiennych klas tylko po to, aby połączyć dane (bez dodawania zachowań). Definiowanie takich klas jest zbyt pracochłonne, ponieważ wymaga, aby konstruktor przypisywał każdy parametr do każdej własności (dekonstruktor też może być przydatny). Dzięki rekordom kompilator może to zrobić za nas.

Ponadto jedną z konsekwencji uczynienia obiektu niezmiennym jest to, że jego tożsamość nie może się zmieniać, co oznacza, że w takich typach lepszym rozwiązaniem jest zaimplementowanie *równości strukturalnej*, a nie *referencyjnej*. Dwa obiekty są równe strukturalnie, kiedy ich dane są takie same (podobnie jak w przypadku krotek). Rekordy zapewniają równość strukturalną domyślnie — bez żadnego szablonowego kodu.

Definiowanie rekordu

Definicja rekordu wygląda jak definicja klasy i może zawierać takie same rodzaje składowych, a więc pola, własności, metody itd. Rekordy mogą implementować interfejsy i dziedziczyć jak klasy po innych rekordach (ale nie po klasach).

Prosty rekord może zawierać jedynie kilka własności tylko do inicjalizacji i ewentualnie konstruktor:

```
record Point
{
    public Point (double x, double y) => (X, Y) = (x, y);
    public double X { get; init; }
    public double Y { get; init; }
}
```

W konstruktorze wykorzystaliśmy skrót, który został opisany w poprzednim punkcie:

```
(X, Y) = (x, y);
```

Ten zapis jest równoważny (w tym przypadku) z tym:

```
{ this.X = x; this.Y = y; }
```


Kup ksi k Pole ksi k

W trakcie kompilacji C# zamieni definicję rekordu w klasę i wykona następujące dodatkowe czynności:

- Napisze chroniony *konstruktor kopiujący* (i ukrytą metodę `Clone`) do obsługi mutacji niedestrukcyjnej.
- Przesłoni/przeciąży funkcje związane z określaniem równości, aby zaimplementować równość strukturalną.
- Przesłoni metodę `ToString()` (aby rozszerzyć własności publiczne rekordu, tak jak w przypadku typów anonimowych).

Powyższa deklaracja rekordu zostanie rozwinięta do następującej postaci:

```
class Point
{
    public Point (double x, double y) => (X, Y) = (x, y);

    public double X { get; init; }
    public double Y { get; init; }

    protected Point (Point original) // „konstruktor kopiujący”
    {
        this.X = original.X; this.Y = original.Y
    }

    // Ta metoda ma dziwną nazwę wygenerowaną przez kompilator:
    public virtual Point <Clone>$() => new Point (this); // metoda Clone

    // Dodatkowy kod przesłonięcia Equals, ==, !=, GetHashCode, ToString()
    // ...
}
```

Choć nic nie stoi na przeszkodzie, aby umieścić w konstruktorze parametry opcjonalne, dobrym zwyczajem (przynajmniej w bibliotekach publicznych) jest umieszczenie ich poza konstruktorem oraz udostępnienie ich jako własności tylko do inicjalizacji:

```
new Foo (123, 234) { Optional2 = 345 };
record Foo
{
    public Foo (int required1, int required2) { ... }

    public int Required1 { get; init; }
    public int Required2 { get; init; }

    public int Optional1 { get; init; }
    public int Optional2 { get; init; }
}
```

Zaletą tego podejścia jest to, że pozwala na dodanie własności tylko do inicjalizacji w późniejszym czasie bez obawy, że zostanie złamana zgodność binarna z konsumentami skompilowanymi z użyciem starszych wersji naszego zestawu.

Kup ksi k Pole ksi k

Listy parametrów

Definicja rekordu może także zawierać listę parametrów:

```
record Point (double X, double Y)
{
    // Tutaj mogą się znajdować opcjonalne definicje dodatkowych składowych klasy...
}
```

Parametry mogą mieć modyfikatory `in` i `params`, ale nie `out` i `ref`. Jeśli jest podana lista parametrów, kompilator wykonuje następujące dodatkowe czynności:

- Dla każdego parametru tworzy własność tylko do inicjalizacji.
- Tworzy **konstruktor główny** do zapisania danych w tych własnościach.
- Tworzy dekonstruktor.

To znaczy, że jeśli zadeklarujemy nasz rekord `Point` tak:

```
record Point (double X, double Y);
```

kompilator wygeneruje prawie dokładnie taki kod jak pokazany poprzednio. Jedyna różnica będzie polegała na tym, że parametry w konstruktorze głównym będą miały nazwy `X` i `Y`, a nie `x` i `y`:

```
public Point (double X, double Y) // „konstruktor główny”
{
    this.X = X; this.Y = Y;
}
```

Ponadto parametry `X` i `Y` z konstruktora głównego staną się magicznie dostępne dla wszystkich inicjalizatorów pól i własności w rekordzie. Te zawiłości zostały dokładniej opisane w punkcie „Konstruktory główne”.

Inną różnicą, gdy zostanie zdefiniowana lista parametrów, jest to, że kompilator dodatkowo wygeneruje dekonstruktor:

```
public void Deconstruct (out double X, out double Y) // dekonstruktor
{
    X = this.X; Y = this.Y;
}
```

Rekordy z listami parametrów mogą mieć „podrekordy”, które tworzy się przy użyciu następującej składni:

```
record Point3D (double X, double Y, double Z) : Point (X, Y);
```

W takim przypadku kompilator utworzy następujący konstruktor główny:

```
class Point3D : Point
{
    public double Z { get; init; }

    public Point3D (double X, double Y, double Z) : base (X, Y)
        => this.Z = Z;
}
```


Kup książkę Poleć książkę

Listy parametrów oferują przydatny skrót, kiedy potrzebna jest tylko klasa gromadząca pewien zbiór wartości (**iloczyn typów** w programowaniu funkcyjnym), oraz może być przydatna do prototypowania. Natomiast na niewiele się zdaje przy dodawaniu logiki do akcesorów `init`, o czym przekonasz się później.

Mutacja niedestrukcyjna

Najważniejszą czynnością wykonywaną przez kompilator w odniesieniu do wszystkich rekordów jest utworzenie *konstruktora kopiującego* (i ukrytej metody `Clone`), ponieważ umożliwia to wykonywanie mutacji niedestrukcyjnej za pomocą słowa kluczowego `with` języka C# 9:

```
Point p1 = new Point (3, 3);
Point p2 = p1 with { Y = 4 };
Console.WriteLine (p2); // Point { X = 3, Y = 4 }

record Point (double X, double Y);
```

W tym przykładzie `p2` jest kopią `p1`, ale z własnością `Y` ustawioną na `4`. Korzyści łatwiej dostrzec, gdy własności jest więcej:

```
Test t1 = new Test (1, 2, 3, 4, 5, 6, 7, 8);
Test t2 = t1 with { A = 10, C = 30 };
Console.WriteLine (t2);

record Test (int A, int B, int C, int D, int E, int F, int G, int H);
```

Wynik:

```
Test { A = 10, B = 2, C = 30, D = 4, E = 5, F = 6, G = 7, H = 8 }
```

Mutacja niedestrukcyjna przebiega dwuetapowo:

1. Najpierw *konstruktor kopiujący* klonuje rekord. Domyślnie kopiuje wszystkie jego pola i tworzy wierną kopię, jednocześnie unikając logiki (i narzutu) w akcesorach `init`. Uwzględnione zostają wszystkie pola (publiczne i prywatne oraz ukryte, na których opierają się własności automatyczne).
2. Następnie zostają zaktualizowane wszystkie własności znajdujące się na *liście inicjalizacji składowych* (tym razem przy użyciu akcesorów `init`).

Ten kod:

```
Test t2 = t1 with { A = 10, C = 30 };
```

kompilator zamieni na funkcjonalny odpowiednik tego:

```
Test t2 = new Test(t1); // sklonowanie t1 pole po polu za pomocą konstruktora kopiującego
t2.A = 10; // aktualizacja własności A
t2.C = 30; // aktualizacja własności C
```

(Gdyby ten kod został napisany bezpośrednio przez programistę, to nie przeszedłby kompilacji, ponieważ `A` i `C` są własnościami tylko do inicjalizacji. Ponadto konstruktor kopiujący jest chroniony. C# obchodzi to przez wywołanie go przez publiczną ukrytą metodę, którą zapisuje w rekordzie o nazwie `<Clone>$`).

Kup ksi k Pole ksi k

W razie potrzeby programista może zdefiniować własny *konstruktor kopiujący* i wtedy C# nie utworzy własnego, tylko użyje tego:

```
protected Point (Point original)
{
    this.X = original.X; this.Y = original.Y;
}
```

Własny konstruktor kopiujący możemy napisać, gdy rekord zawiera zmienne podobiekty lub kolekcje, które chcemy sklonować, albo gdy są obliczane pola, które chcemy wyczyścić. Niestety domyślną implementację można tylko *wymienić*, nie można jej *rozszerzyć*.

Przy tworzeniu podrekordu konstruktor kopiujący odpowiada tylko za skopiowanie własnych pól. Aby skopiować pola rekordu bazowego, należy oddelegować zadanie do bazy:

```
protected Point (Point original) : base (original)
{
    ...
}
```

Walidacja własności

Logikę sprawdzania jawnie zdefiniowanych własności można zawrzeć w akcesorach `init`. W poniższym przykładzie nie chcemy, aby własność `X` kiedykolwiek przybrała wartość `NaN` (nie liczba):

```
record Point
{
    // Zwróć uwagę, że przypisujemy x do własności X (a nie do pola _x):
    public Point (double x, double y) => (X, Y) = (x, y);
    double _x;
    public double X
    {
        get => _x;
        init
        {
            if (double.IsNaN (value))
            throw new ArgumentException ("X nie może być NaN");
            _x = value;
        }
    }
    public double Y { get; init; }
}
```

W ten sposób gwarantujemy sobie, że poprawność jest sprawdzana zarówno w trakcie tworzenia obiektu, jak i po utworzeniu jego kopii:

```
Point p1 = new Point (2, 3);
Point p2 = p1 with { X = double.NaN }; // zgłasza wyjątek
```

Przypomnę, że automatycznie generowany konstruktor kopiujący kopiuje wszystkie pola i własności automatyczne. To znaczy, że teraz wygenerowany konstruktor kopiujący będzie wyglądał tak:

Kup ksi k   Pole ksi k

```
protected Point (Point original)
{
    _x = original._x; Y = original.Y;
}
```

Zauważ, że operacja kopiowania pola `_x` omija akcesor własności `X`. To jednak nie szkodzi w żaden sposób, ponieważ sytuacja dotyczy wiernego skopiowania obiektu, który zostanie wcześniej wypełniony danymi przez akcesor `init` własności `X`.

Pola obliczane i leniwa ewaluacja

W programowaniu funkcyjnym jest stosowany popularny wzorzec, który doskonale sprawdza się w pracy z typami niezmiennymi. Jest to tzw. **leniwa ewaluacja**, która zakłada obliczenie wartości dopiero wtedy, gdy jest potrzebna, a następnie zapisanie jej w buforze do wykorzystania w przyszłości. Wyobraź sobie na przykład, że w rekordzie `Point` chcesz zdefiniować własność zwracającą odległość od początku układu (0, 0):

```
record Point (double X, double Y)
{
    public double DistanceFromOrigin => Math.Sqrt (X*X + Y*Y);
}
```

Teraz zmienimy nasz kod tak, aby uniknąć ponownego obliczania wartości `DistanceFromOrigin` za każdym razem, gdy jest potrzebna. Zaczniemy od usunięcia listy własności oraz zdefiniowania `X`, `Y` i `DistanceFromOrigin` jako własności tylko do odczytu. Następnie obliczamy wartość tej ostatniej w konstruktorze:

```
record Point
{
    public double X { get; }
    public double Y { get; }
    public double DistanceFromOrigin { get; }
    public Point (double x, double y) =>
    (X, Y, DistanceFromOrigin) = (x, y, Math.Sqrt (x*x + y*y));
}
```

Jest dobrze, ale teraz nie ma możliwości wykonywania mutacji niedestrukcyjnej (zmiana `X` i `Y` na własności tylko do inicjalizacji byłaby błędem, wówczas bowiem własność `DistanceFromOrigin` stałaby się przestarzała po wykonaniu akcesorów `init`). Poza tym nie byłoby to optymalne rozwiązanie, ponieważ obliczenia byłyby wykonywane zawsze bez względu na to, czy własność `DistanceFromOrigin` byłaby kiedykolwiek odczytana. Najlepszym wyjściem jest leniwe zapisanie jej wartości (przy pierwszym użyciu) w polu:

```
record Point
{
    ...

    double? _distance;
    public double DistanceFromOrigin
    {
        get
        {
            if (_distance == null)
              _distance = Math.Sqrt (X*X + Y*Y);
```


Kup ksi k Pole ksi k

```
                return _distance.Value;
            }
        }
    }
```

W tym kodzie zasadniczo *wykonujemy kopię* własności `_distance`. Nadal jednak możemy powiedzieć, że `Point` jest typem niezmiennym. Zmiana pola tylko po to, aby leniwie wstawić wartość, nie łamie zasady ani nie niweczy zalet niezmienności i można ją nawet zamaskować za pomocą typu `Lazy<T>`, którego opis znajduje się w rozdziale 21.

Za pomocą operatora C# `??=` możemy skrócić deklarację własności do jednego wiersza kodu:

```
public double DistanceFromOrigin => _distance ??= Math.Sqrt (X*X + Y*Y);
```

(W ten sposób mówimy: zwróć `_distance`, jeśli ma wartość różną od `null`, w przeciwnym razie zwróć `Math.Sqrt (X*X + Y*Y)` i przypisz ją do `_distance`).

Aby to działało z własnościami tylko do inicjalizacji, musimy zrobić coś jeszcze — skasować zbuforowane pole `_distance` w momencie aktualizacji `X` lub `Y` przez akcesor `init`. Oto kompletny kod:

```
record Point
{
    public Point (double x, double y) => (X, Y) = (x, y);
    double _x, _y;
    public double X { get => _x; init { _x = value; _distance = null; } }
    public double Y { get => _y; init { _y = value; _distance = null; } }
    double? _distance;
    public double DistanceFromOrigin => _distance ??= Math.Sqrt (X*X + Y*Y);
}
```

Teraz obiekty typu `Point` można kopiować niedestrukcyjnie:

```
Point p1 = new Point (2, 3);
Console.WriteLine (p1.DistanceFromOrigin); // 3,605551275463989
Point p2 = p1 with { Y = 4 };
Console.WriteLine (p2.DistanceFromOrigin); // 4,47213595499958
```

Przyjemnym dodatkiem jest to, że automatycznie wygenerowany konstruktor kopiujący kopiuje zbuforowane pole `_distance`. To znaczy, że gdyby rekord zawierał inne własności niebiorące udziału w obliczeniach, mutacja niedestrukcyjna tych własności nie spowodowałaby niepotrzebnej straty zapisanej wartości. Jeśli Ci na tym nie zależy, to nie musisz kasować zapisanej wartości w akcesorach `init`, tylko możesz napisać własny konstruktor kopiujący, który ignoruje to pole. Takie rozwiązanie jest zwięźlejsze, ponieważ działa z listami parametrów, a własny konstruktor kopiujący może wykorzystywać destruktor:

```
record Point (double X, double Y)
{
    double? _distance;
    public double DistanceFromOrigin => _distance ??= Math.Sqrt (X*X + Y*Y);
    protected Point (Point other) => (X, Y) = other;
}
```

W obu przypadkach dodatek leniwie obliczanych pól łamie domyślną operację porównywania strukturalnego (ponieważ takie pola mogą, ale nie muszą, być wypełniane), ale łatwo to można naprawić, o czym wkrótce się przekonasz.

Kup książkę   Poleć książkę

Konstruktory główne

Kiedy programista definiuje rekord z listą parametrów, kompilator automatycznie generuje deklaracje własności i **konstruktor główny** (oraz dekonstruktor). Jak widzieliśmy, w prostych przypadkach sprawdza się to bardzo dobrze, a w bardziej złożonych sytuacjach listę parametrów można pominąć oraz samodzielnie napisać deklaracje własności i konstruktor.

Ponadto w języku C# jest dostępna jeszcze opcja pośrednia — jeśli ktoś ma ochotę bawić się z konstruktorami głównymi, które mają ciekawą semantykę, polegająca na zdefiniowaniu listy parametrów oraz samodzielnym napisaniu niektórych lub wszystkich deklaracji własności:

```
record Student (string ID, string LastName, string GivenName)
{
    public string ID { get; } = ID;
}
```

W tym przypadku „przejęliśmy” definicję własności `ID` i określiliśmy ją jako tylko do odczytu (zamiast tylko do inicjalizacji), dzięki czemu nie będzie brała udziału w mutacji niedestrukcyjnej. Jeśli dana własność ma nigdy nie być powielana niedestrukcyjnie, uczynienie jej tylko do odczytu pozwala na zapisanie obliczonych danych w rekordzie bez konieczności tworzenia mechanizmu odświeżania.

Zwróć uwagę, że musimy dodać **inicjalizator własności** (pogrubiony):

```
public string ID { get; } = ID;
```

Kiedy „przejmujemy” deklarację własności, stajemy się odpowiedzialni za inicjalizację jej wartości, ponieważ przestaje to być obowiązkiem konstruktora głównego. Podkreślę, że `ID` w pogrubieniu dotyczy parametru *konstruktora głównego*, a nie własności `ID`.

Konstruktory główne mają wyjątkową cechę powodującą, że ich parametry (w tym przypadku `ID`, `LastName` i `GivenName`) są w magiczny sposób widoczne dla wszystkich inicjalizatorów pól i własności. Ilustruje to poniższy przykład:

```
record Student (string ID, string LastName, string FirstName)
{
    public string ID { get; } = ID;
    readonly int _enrolmentYear = int.Parse (ID.Substring (0, 4));
}
```

W tym przypadku także pogrubiony `ID` to parametr konstruktora głównego, a nie własność. (Kod jest mimo to jednoznaczny, ponieważ inicjalizatory nie mają dostępu do własności).

W tym przykładzie obliczyliśmy `_enrolmentYear` z czterech pierwszych cyfr `ID`. Choć można bezpiecznie przechowywać tę wartość w polu tylko do odczytu (ponieważ własność `ID` jest tylko do odczytu, a więc nie może być powielona niedestrukcyjnie), w prawdziwym programie ten kod nie działałby dobrze. Wynika to z faktu, że bez jawnego konstruktora nie ma centralnego miejsca do sprawdzania `ID` i zgłaszania odpowiedniego wyjątku w razie problemu (to typowy wymóg).

Kup książkę Poleć książkę

Walidacja jest także dobrym powodem do napisania jawnych akcesorów tylko do inicjalizacji (o czym była mowa w punkcie „Walidacja własności"). Niestety konstruktory główne słabo spisują się w takiej sytuacji. Spójrz np. na poniższy rekord, w którym akcesor `init` sprawdza obecność wartości `null`:

```
record Person (string Name)
{
    string _name = Name;
    public string Name
    {
        get => _name;
        init => _name = value ?? throw new ArgumentNullException ("Name");
    }
}
```

Własność `Name` nie jest automatyczna, więc nie może definiować inicjalizatora. Najlepsze, co możemy zrobić, to umieszczenie inicjalizatora na polu zapasowym (pogrubienie). To niestety powoduje obejście testu `null`:

```
var p = new Person (null); // Powodzenie! (obejście testu null)
```

Trudność polega na tym, że nie ma sposobu na przypisanie parametru konstruktora głównego do własności bez własnoręcznego napisania tego konstruktora. Choć istnieją obejścia (np. przeniesienie logiki walidacji `init` do osobnej metody statycznej, którą wywołujemy dwa razy), najprościej jest unikać listy parametrów i ręcznie napisać zwykły konstruktor (a w razie potrzeby także dekonstruktor):

```
record Person
{
    public Person (string name) => Name = name; // przypisanie do *WŁASNOŚCI*
    string _name;
    public string Name { get => _name; init => ... }
}
```

Rekordy i porównywanie

Rekordy, tak jak struktury, typy anonimowe i krotki, standardowo zapewniają równość strukturalną, co znaczy, że dwa rekordy są równe, jeśli ich pola (i własności automatyczne) są równe:

```
var p1 = new Point (1, 2);
var p2 = new Point (1, 2);
Console.WriteLine (p1.Equals (p2)); // prawda

record Point (double X, double Y);
```

Operator równości także działa z rekordami (tak jak i krotkami):

```
Console.WriteLine (p1 == p2); // prawda
```

Domyślna implementacja równości w rekordach jest siłą rzeczy krucha. Wystarczy obecność leniwych wartości, wartości przejściowych, tablic lub typów kolekcyjnych (które wymagają specjalnego traktowania, jeśli chodzi o porównywanie), aby ją złamać. Na szczęście względnie łatwo to naprawić (jeśli potrzebujemy operacji równości) i jest to mniej pracochłonne niż definiowanie pełnej operacji porównywania w klasie lub strukturze.

Kup książkę Poleć książkę

W odróżnieniu od klas i struktur w tym przypadku nie przesłania się metody `object.Equals`. Zamiast tego należy zdefiniować publiczną metodę `Equals` o następującej sygnaturze:

```
record Point (double X, double Y)
{
    double _someOtherField;
    public virtual bool Equals (Point other) =>
    other != null && X == other.X && Y == other.Y;
}
```

Ta metoda `Equals` musi być wirtualna (nie być przesłonięciem) i musi być *silnie typizowana*, aby przyjmowała rzeczywisty typ rekordu (w tym przypadku `Point`, nie obiekt). Kiedy sygnatura będzie poprawna, kompilator automatycznie skonstruuje metodę.

W naszym przypadku zmieniliśmy logikę porównywania tak, aby pod uwagę były brane tylko pola `X` i `Y` (`_someOtherField` ma być ignorowane).

Gdybyśmy utworzyli podrekord innego rekordu, moglibyśmy wywołać metodę `base.Equals`:

```
public virtual bool Equals (Point other) => base.Equals (other) && ...
```

Tak jak w przypadku każdego typu, jeśli przejmiemy operację porównywania, to dodatkowo powinniśmy nadpisać także metodę `GetHashCode()`. Miłą cechą rekordów jest to, że nie trzeba przeciążać operatorów `!=` ani `==` oraz nie trzeba implementować interfejsu `IEquatable<T>`: wszystko to jest robione automatycznie. Szczegółowy opis zagadnienia porównywania znajduje się w podrozdziale „Porównywanie” w rozdziale 6.

Wzorce

W rozdziale 3. pokazaliśmy, jak sprawdzać za pomocą operatora `is`, czy operacja konwersji referencji ma szansę się udać:

```
if (obj is string)
Console.WriteLine (((string)obj).Length);
```

Albo zwięźlej:

```
if (obj is string s)
    Console.WriteLine (s.Length);
```

Wykorzystujemy tu jeden rodzaj wzorca, zwany **wzorcem typu**. Operator `is` obsługuje także inne wzorce wprowadzone w C# 7 i C# 8, np. **wzorzec własności**:

```
if (obj is string { Length:4 })
    Console.WriteLine ("Łańcuch zawierający cztery znaki.");
```

Wzorce są obsługiwane w następujących kontekstach:

- po operatorze `is` (**zmienna** jest **wzorcem**);
- w instrukcjach `switch`;
- w wyrażeniach `switch`.

Kup ksi k Pole ksi k

Wzorzec typu opisaliśmy już w rozdziale 2., w punkcie „Instrukcja switch z typami”, i w rozdziale 3., w punkcie „Operator is”. Zwięźle przedstawiliśmy też już wzorzec krotki. W tym podrozdziale opisujemy bardziej zaawansowane wzorce, które zostały wprowadzone do języka C# niedawno.

Niektóre bardziej specjalistyczne wzorce są przeznaczone do użycia w instrukcjach i wyrażeniach `switch` — zmniejszają w nich zapotrzebowanie na klauzule `when` i umożliwiają używanie konstrukcji `switch` tam, gdzie wcześniej było to niemożliwe.

Wzorce opisane w tym podrozdziale są umiarkowanie przydatne w pewnych sytuacjach. Pamiętaj, że wyrażenia `switch` z wykorzystaniem wzorców zawsze możesz zastąpić prostymi instrukcjami `if` — a w niektórych przypadkach trzyargumentowym operatorem warunkowym — często bez konieczności dodawania zbyt dużej ilości kodu.

Wzorzec var

Wzorzec `var` stanowi wersję **wzorca typu**, w której nazwę typu zastąpiono słowem kluczowym `var`. Konwersja zawsze się udaje, więc celem jest jedynie umożliwienie ponownego użycia zmiennej:

```
bool IsJanetOrJohn (string name) =>
    name.ToUpper() is var upper && (upper == "JANET" || upper == "JOHN");
```

Ten kod jest równoważny z tym:

```
bool IsJanetOrJohn (string name)
{
    string upper = name.ToUpper();
    return upper == "JANET" || upper == "JOHN";
}
```

Możliwość wprowadzenia i ponownego wykorzystania zmiennej pośredniej (w tym przypadku `product`) w metodzie o postaci wyrażenia jest bardzo wygodna. Niestety, można z niej skorzystać tylko, gdy metoda ma logiczny typ zwrotny.

Wzorzec stałej

Wzorzec stałej umożliwia bezpośrednie porównywanie ze stałymi i jest przydatny podczas pracy z typem `object`:

```
void Foo (object obj)
{
    if (obj is 3) ...
}
```

Pogrubione wyrażenie jest równoważne z poniższym:

```
obj is int && (int)obj == 3
```

(Operator == jest statyczny i C# nie pozwoli go użyć do porównania obiektu bezpośrednio ze stałą, ponieważ kompilator musi znać typy z góry).

Kup ksi k Pole ksi k

Samodzielnie ten wzorzec ma niewielkie zastosowanie, istnieje bowiem dobra alternatywna opcja:

```
if (3.Equals (obj)) ...
```

Jak zobaczysz wkrótce, wzorzec stałej jest bardziej przydatny w połączeniu z **kombinatorami wzorców**.

Wzorce relacyjne (C# 9)

Od C# 9 we wzorcach można używać operatorów <, >, <= oraz >=:

```
if (x is > 100) Console.WriteLine ("Wartość x jest większa od 100.");
```

To nabiera praktycznego znaczenia w instrukcji `switch`:

```
string GetWeightCategory (decimal bmi) => bmi switch
{
    < 18.5m => "niedowaga",
    < 25m => "norma",
    < 30m => "nadwaga",
    _ => "otyłość"
};
```

Wzorce relacyjne są jeszcze bardziej przydatne w połączeniu z **kombinatorami wzorców**.

Wzorzec relacyjny działa także wtedy, gdy zmienna ma typ obiektu czasu kompilacji, ale trzeba bardzo uważać podczas używania stałych numerycznych. Ostatni wiersz poniższego kodu drukuje fałsz, ponieważ próbujemy porównać wartość dziesiętną z literałem całkowitoliczbowym:

```
object obj = 2m; // obj jest typu dziesiętnego
Console.WriteLine (obj is < 3m); // prawda
Console.WriteLine (obj is < 3); // fałsz
```

Kombinatory wzorców (C# 9)

Od C# 9 wzorce można łączyć za pomocą słów kluczowych `and`, `or` i `not`:

```
bool IsJanetOrJohn (string name) => name.ToUpper() is "JANET" or "JOHN";
bool IsVowel (char c) => c is 'a' or 'e' or 'i' or 'o' or 'u';
bool Between1And9 (int n) => n is >= 1 and <= 9;
bool IsLetter (char c) => c is >= 'a' and <= 'z'
                            or >= 'A' and <= 'Z';
```

Tak jak między operatorami `&&` i `||`, słowo kluczowe `and` ma pierwszeństwo przed `or`. Można to zmienić za pomocą nawiasów.

Ciekawą sztuczką jest połączenie kombinatora `not` ze wzorcem typu w celu sprawdzenia, czy obiekt jest (lub nie jest) danego typu:

```
if (obj is not string) ...
```

To wygląda lepiej niż to:

```
if (!(obj is string))
```


Kup ksi k Pole ksi k

Wzorce krotek i pozycyjne

Wzorzec krotki (wprowadzony w C# 8) zapewnia mechanizm porównywania krotek:

```
var p = (2, 3);
Console.WriteLine (p is (2, 3)); // prawda
```

Można to wykorzystać do wyboru jednej spośród wielu wartości:

```
int AverageCelsiusTemperature (Season season, bool daytime) =>
(season, daytime) switch
{
    (Season.Spring, true) => 20,
     (Season.Spring, false) => 16,
     (Season.Summer, true) => 27,
     (Season.Summer, false) => 22,
     (Season.Fall, true) => 18,
     (Season.Fall, false) => 12,
     (Season.Winter, true) => 10,
     (Season.Winter, false) => -2,
    _ => throw new Exception ("Niespodziewana kombinacja.")
};
enum Season { Spring, Summer, Fall, Winter };
```

Wzorzec krotki można traktować jak specjalny typ wzorca pozycyjnego (C# 8+), który wybiera dowolny typ udostępniający metodę `Deconstruct` („Dekonstruktory” w rozdziale 3.). W poniższym przykładzie wykorzystujemy wygenerowany przez kompilator dekonstruktor rekordu `Point`:

```
var p = new Point (2, 2);
Console.WriteLine (p is (2, 2)); // prawda
record Point (int X, int Y); // ma dekonstruktor wygenerowany przez kompilator
```

Dekonstrukcję po dopasowaniu można wykonać przy użyciu następującej składni:

```
Console.WriteLine (p is (var x, var y) && x == y); // prawda
```

Poniżej znajduje się wyrażenie `switch` łączące wzorzec typu ze wzorcem pozycyjnym:

```
string Print (object obj) => obj switch
{
    Point (0, 0) => "Pusty punkt",
    Point (var x, var y) when x == y => "Przekątna"
    ...
};
```

Wzorce własności

Wzorzec własności (C# 8) odnosi się do wartości jednej lub większej liczby własności obiektu. We wcześniejszym opisie operatora is podaliśmy już prosty przykład:

```
if (obj is string { Length:4 }) ...
```

To jednak daje niewielką korzyść w porównaniu z tym:

```
if (obj is string s && s.Length == 4) ...
```


Kup ksi k Pole ksi k

Większe korzyści dają wzorce własności w połączeniu z instrukcjami i wyrażeniami `switch`. Weźmy np. klasę `System.Uri`, reprezentującą URI. Zawiera ona m.in. własności `Scheme`, `Host`, `Port` i `IsLoopback`. Pisząc zaporę sieciową, do podejmowania decyzji, czy blokować dane URI, możemy używać wyrażenia `switch` wykorzystującego wzorce:

```
bool ShouldAllow (Uri uri) => uri switch
{
    { Scheme: "http", Port: 80   } => true,
    { Scheme: "https", Port: 443 } => true,
    { Scheme: "ftp", Port: 21    } => true,
    { IsLoopback: true           } => true,
    _ => false
};
```

Własności można zagnieżdżać, dzięki czemu poniższa klauzula staje się poprawna:

```
{ Scheme: { Length: 4 }, Port: 80 } => true,
```

We wzorcach własności można używać także innych wzorców, w tym wzorca relacyjnego:

```
{ Host: { Length: < 1000 }, Port: > 0 } => true,
```

Bardziej rozbudowane warunki można wyrazić przy użyciu klauzuli `when`:

```
{ Scheme: "http" } when string.IsNullOrWhiteSpace (uri.Query) => true,
```

Wzorzec typu można połączyć także z wzorcem własności:

```
bool ShouldAllow (object uri) => uri switch
{
    Uri { Scheme: "http", Port: 80 } => true,
    Uri { Scheme: "https", Port: 443 } => true,
    ...
```

Jak się pewnie spodziewasz, w przypadku wzorców typu na końcu klauzuli można wprowadzić zmienną, aby potem jej użyć:

```
Uri { Scheme: "http", Port: 80 } httpUri => httpUri.Host.Length < 1000,
```

Zmiennej tej można też użyć w klauzuli `when`:

```
Uri { Scheme: "http", Port: 80 } httpUri
                                 when httpUri.Host.Length < 1000 => true,
```

Nieco dziwną cechą wzorców własności jest możliwość wprowadzania zmiennych na poziomie *własności*:

```
{ Scheme: "http", Port: 80, Host: string host } => host.Length < 1000,
```

Dozwolone jest także niejawne określanie typów, a więc zamiast `string` można napisać `var`. Oto kompletny przykład:

```
bool ShouldAllow (Uri uri) => uri switch
{
    { Scheme: "http",  Port: 80, Host: var host } => host.Length < 1000,
    { Scheme: "https", Port: 443                } => true,
    { Scheme: "ftp",   Port: 21                 } => true,
    { IsLoopback: true } => true,
    _ => false
};
```


Kup ksi k   Pole ksi k

Trudno wymyślić przykład, w którym technika ta pozwalałaby zyskać więcej niż pozbycie się paru znaków. W naszym przypadku alternatywne rozwiązanie jest nawet krótsze:

```
{ Scheme: "http", Port: 80 } => uri.Host.Length < 1000 => ...
```

lub

```
{ Scheme: "http", Port: 80, Host: { Length: < 1000 } } => ...
```

Atrybuty

Znane jest nam już pojęcie oznaczania atrybutami elementów kodu za pomocą modyfikatorów, np. `virtual` lub `ref`. Są to konstrukcje wbudowane w język programowania. **Atrybuty** natomiast są rozszerzalnym mechanizmem do dodawania informacji do różnych konstrukcji programistycznych (zestawów, typów, składowych, wartości zwrotnych, parametrów i parametrów typów generycznych). Rozszerzalność ta jest potrzebna do działania usług głęboko integrujących się z systemem typów, bez konieczności stosowania specjalnych słów kluczowych czy konstrukcji języka C#.

Dobrym przykładem zastosowania atrybutów jest serializacja, czyli proces polegający na konwersji obiektów na określony format do przechowywania i przesyłania i odwrotnie. W tym procesie atrybut pola może określać sposób przełożenia między reprezentacją pola w języku C# a reprezentacją tego pola w danym formacie.

Klasy atrybutów

Atrybut definiuje się jako klasę dziedziczącą (pośrednio lub bezpośrednio) po abstrakcyjnej klasie `System.Attribute`. Aby powiązać atrybut z elementem kodu źródłowego, należy przed tym elementem wpisać nazwę typu atrybutu w nawiasie kwadratowym. W poniższym kodzie np. wiążemy atrybut `ObsoleteAttribute` z klasą `Foo`:

```
[ObsoleteAttribute]
public class Foo {...}
```

Atrybut ten zostanie rozpoznany przez kompilator i będzie powodował wyświetlanie ostrzeżeń, jeśli użyty typ lub składowa zostały oznaczone jako przestarzałe. Zwyczajowo nazwy wszystkich typów atrybutów kończą się słowem **`Attribute`**. Kompilator C# zna tę konwencję i pozwala na opuszczenie tego przedrostka przy podawaniu nazwy atrybutu:

```
[Obsolete]
public class Foo {...}
```

`ObsoleteAttribute` to typ zadeklarowany w przestrzeni nazw `System` w następujący sposób (deklaracja skrócona dla uproszczenia):

```
public sealed class ObsoleteAttribute : Attribute {...}
```

Biblioteki .NET zawierają wiele gotowych atrybutów. Sposoby tworzenia własnych atrybutów opisaliśmy w rozdziale 18.

Kup ksi k   Pole ksi k

Atrybuty nazwane i pozycyjne

Atrybuty mogą mieć parametry. W poniższym przykładzie przypisaliśmy klasie atrybut XmlTypeAttribute, który informuje serializator XML (z przestrzeni nazw System.Xml.Serialization), jaka jest reprezentacja obiektu w XML, oraz przyjmuje kilka **parametrów atrybutu**. Poniższy atrybut mapuje klasę CustomerEntity na element XML o nazwie Customer należący do przestrzeni nazw http://oreilly.com:

```
[XmlType ("Customer", Namespace="http://oreilly.com")]
public class CustomerEntity { ... }
```

Każdy parametr atrybutu należy do jednej z dwóch kategorii: atrybutów *pozycyjnych* lub atrybutów *nazwanych*. W powyższym przykładzie pierwszy argument jest **parametrem pozycyjnym**, a drugi **parametrem nazwanym**. Parametry pozycyjne odpowiadają parametrom konstruktorów publicznych typu atrybutu. Parametry nazwane odpowiadają polom lub własnościom publicznym typu atrybutu.

Przy określaniu atrybutu należy podać parametry pozycyjne odpowiadające jednemu z konstruktorów atrybutu. Parametry nazwane są opcjonalne.

W rozdziale 18. opisujemy dozwolone typy parametrów i zasady obliczania ich wartości.

Stosowanie atrybutów do zestawów i pól pomocniczych

Celem atrybutu domyślnie jest element kodu, który znajduje się bezpośrednio za atrybutem, a którym najczęściej jest typ lub składowa typu. Ale atrybuty można też przypisywać zestawom, choć w tym przypadku konieczne jest określenie celu wprost. Poniżej znajduje się przykład użycia atrybutu AssemblyFileVersion w celu powiązania wersji z zestawem:

```
[assembly: AssemblyFileVersion ("1.2.3.4")]
```

Od C# 7.3 można używać konstrukcji field: przedrostek w celu zastosowania atrybutu do pól pomocniczych automatycznej własności. Pomaga to w kontrolowaniu serializacji:

```
[field:NonSerialized]
public int MyProperty { get; set; }
```

Przypisywanie wielu atrybutów

Jednemu elementowi kodu można przypisać kilka atrybutów. Wszystkie atrybuty można wypisać na liście w jednym nawiasie kwadratowym (rozdzielając je przecinkami) lub w osobnych nawiasach kwadratowych (można też zastosować rozwiązanie mieszane). Poniższe trzy przykłady pod względem semantycznym niczym się nie różnią:

```
[Serializable, Obsolete, CLSCompliant(false)]
public class Bar {...}

[Serializable] [Obsolete] [CLSCompliant(false)]
public class Bar {...}

[Serializable, Obsolete]
[CLSCompliant(false)]
public class Bar {...}
```


Kup ksi k Pole ksi k

Atrybuty informacji wywołującego

Istnieje możliwość znakowania atrybutów opcjonalnych jednym z trzech **atrybutów informacji wywołującego** (ang. *caller info attributes*), które nakazują kompilatorowi użycie informacji uzyskanych z kodu źródłowego podmiotu wywołującego jako wartości domyślnej parametrów:

- Atrybut `[CallerMemberName]` przekazuje nazwę składowej wywołującego.
- Atrybut `[CallerFilePath]` przekazuje ścieżkę do pliku z kodem źródłowym wywołującego.
- Atrybut `[CallerLineNumber]` przekazuje numer wiersza w pliku z kodem źródłowym wywołującego.

Metoda `Foo` zdefiniowana w poniższym programie ilustruje sposób użycia wszystkich trzech atrybutów:

```
using System;
using System.Runtime.CompilerServices;

class Program
{
  static void Main() => Foo();

  static void Foo (
    [CallerMemberName] string memberName = null,
    [CallerFilePath] string filePath = null,
    [CallerLineNumber] int lineNumber = 0)
  {
    Console.WriteLine (memberName);
    Console.WriteLine (filePath);
    Console.WriteLine (lineNumber);
  }
}
```

Jeśli nasz program znajduje się w pliku *c:\source\test\Program.cs*, wynik jego działania będzie następujący:

```
Main
c:\source\test\Program.cs
6
```

Tak jak w przypadku zwykłych parametrów opcjonalnych, zamiana odbywa się w *miejscu wywołania*. W związku z tym nasza metoda `Main` jest tylko udogodnieniem syntaktycznym dla tego:

```
static void Main() => Foo ("Main", @"c:\source\test\Program.cs", 6);
```

Atrybuty informacji wywołującego można wykorzystać w dziennikach, a także do implementacji pewnych wzorców, jak np. uruchamianie pojedynczego zdarzenia powiadamiającego o zmianie, gdy zmieni się którakolwiek z własności obiektu. Istnieje służący do tego standardowy interfejs o nazwie `INotifyPropertyChange`, który znajduje się w przestrzeni nazw `System.ComponentModel`:

```
public interface INotifyPropertyChanged
{
  event PropertyChangedEventHandler PropertyChanged;
}

public delegate void PropertyChangedEventHandler
  (object sender, PropertyChangedEventArgs e);
```


Kup ksi k Pole ksi k

```
public class PropertyChangedEventArgs : EventArgs
{
  public PropertyChangedEventArgs (string propertyName);
  public virtual string PropertyName { get; }
}
```

Zwróć uwagę, że konstruktor `PropertyChangedEventArgs` wymaga podania nazwy własności, która się zmieniła. Ale dzięki użyciu atrybutu `[CallerMemberName]` możemy zaimplementować ten interfejs i wywołać zdarzenie bez podawania jakichkolwiek nazw własności:

```
public class Foo : INotifyPropertyChanged
{
  public event PropertyChangedEventHandler PropertyChanged = delegate { };

  void RaisePropertyChanged ([CallerMemberName] string propertyName = null)
    => PropertyChanged (this, new PropertyChangedEventArgs (propertyName));

  string customerName;
  public string CustomerName
  {
    get => customerName;
    set
    {
      if (value == customerName) return;
      customerName = value;
      RaisePropertyChanged();
      // kompilator przekonwertuje powyższy wiersz na:
      // RaisePropertyChanged ("CustomerName");
    }
  }
}
```

Wiązanie dynamiczne

Wiązanie dynamiczne polega na odłożeniu **wiązania** — procesu rozpoznawania typów, składowych i operacji — z czasu kompilacji do czasu wykonywania programu. Jest to przydatne, gdy na etapie kompilacji **programista** wie, że dana funkcja, składowa lub operacja istnieje, ale nie wie tego **kompilator**. Taka sytuacja często zachodzi podczas pracy z dynamicznymi językami programowania (np. IronPython) i technologią COM oraz podczas korzystania z technik refleksji.

Typy dynamiczne deklaruje się za pomocą kontekstowego słowa kluczowego `dynamic`:

```
dynamic d = GetSomeObject();
d.Quack();
```

Obecność typu dynamicznego jest dla kompilatora sygnałem do rozluźnienia. Spodziewamy się, że w czasie działania programu typ zmiennej `d` będzie dysponował metodą `Quack`, choć nie możemy tego udowodnić w sposób statyczny. Jako że typ zmiennej `d` jest dynamiczny, kompilator odłoży wiązanie `Quack` z `d` do czasu wykonywania programu. Aby zrozumieć, co to znaczy, trzeba znać różnicę między **wiązaniem statycznym** i **wiązaniem dynamicznym**.

Wiązanie statyczne a wiązanie dynamiczne

Klasycznym przykładem wiązania jest skojarzenie nazwy z konkretną funkcją podczas kompilacji wyrażenia. Aby skompilować poniższe wyrażenie, kompilator musi znaleźć implementację metody `Quack`:

```
d.Quack();
```


Kup ksi k Pole ksi k

Powiedzmy, że statycznym typem zmiennej d jest Duck:

```
Duck d = ...
d.Quack();
```

W najprostszym przypadku wiązania kompilator poszuka bezparametrowej metody o nazwie Quack w typie Duck. Jeśli jej nie znajdzie, rozszerzy zakres poszukiwań na metody przyjmujące parametry opcjonalne, metody znajdujące się w klasach nadrzędnych klasy Duck oraz metody rozszerzające przyjmujące Duck w pierwszym parametrze. Jeśli i to nie przyniesie efektu, zostanie zgłoszony błąd kompilacji. Bez względu na to, jaka metoda zostanie związana, należy podkreślić, że wiązania dokonuje kompilator i powodzenie tej operacji w całości zależy od statycznej znajomości typów argumentów (w tym przypadku d). To właśnie nazywa się **wiązaniem statycznym** (ang. *static binding*).

Teraz zmienimy statyczny typ d na object:

```
object d = ...
d.Quack();
```

Wywołanie metody Quack spowoduje błąd kompilacji, ponieważ choć wartość zapisana w d może zawierać metodę o nazwie Quack, kompilator o tym nie wie, gdyż dysponuje tylko informacją o typie zmiennej, którym w tym przypadku jest object. Zmienimy więc statyczny typ d na dynamiczny:

```
dynamic d = ...
d.Quack();
```

Typ dynamic jest jak object — podobnie jak on nic nie mówi o rzeczywistym typie zmiennej. Różnica między nimi polega na tym, że typ dynamic umożliwia posługiwanie się zmienną w sposób nieokreślony podczas kompilacji. Gdy kompilator napotyka dynamicznie wiązane wyrażenie (jakim jest każde wyrażenie zawierające wartość typu dynamic), to tylko je pakuje, aby wiązania można było dokonać w czasie działania programu.

W czasie działania programu, jeśli obiekt implementuje interfejs IDynamicMetaObjectProvider, to wiązanie jest wykonywane przy użyciu tego interfejsu. W przeciwnym przypadku wiązanie jest wykonywane prawie dokładnie tak samo, jak gdyby kompilator znał typ wykonawczy obiektu dynamicznego. Te dwa rozwiązania nazywają się odpowiednio wiązaniem **niestandardowym** (ang. *custom binding*) i wiązaniem **językowym** (ang. *language binding*).

Wiązanie niestandardowe

Wiązanie niestandardowe zachodzi, gdy obiekt dynamiczny implementuje interfejs IdynamicMe↪taObjectProvider (IDMOP). Choć ten interfejs można implementować w typach tworzonych w języku C# i niekiedy jest to uzasadnione, częściej jest tak, że otrzymujemy obiekt IDMOP z języka dynamicznego zaimplementowanego w .NET na bazie DLR, np. IronPython lub IronRuby. Obiekty pochodzące z tych języków niejawnie implementują interfejs IDMOP jako środek umożliwiający bezpośrednią kontrolę znaczenia wykonywanych na nich operacji.

Wiązaniami niestandardowymi bardziej szczegółowo zajmujemy się w rozdziale 19., a na razie przedstawiamy tylko prosty przykład demonstracyjny:

```
using System;
using System.Dynamic;

dynamic d = new Duck();
```


Kup ksi k Pole ksi k

```
d.Quack();              // Wywołano metodę Quack
d.Waddle();             // Wywołano metodę Waddle

public class Duck : DynamicObject
{
  public override bool TryInvokeMember (
    InvokeMemberBinder binder, object[] args, out object result)
  {
    Console.WriteLine ("Wywołano metodę " + binder.Name);
    result = null;
    return true;
  }
}
```

Klasa Duck tak naprawdę nie zawiera metody Quack. Zamiast tego wykorzystuje wiązanie niestandardowe w celu przechwycenia i interpretacji wszystkich wywołań metod.

Wiązanie językowe

Wiązanie językowe zachodzi wtedy, gdy obiekt dynamiczny nie implementuje interfejsu IDynamicMetaObjectProvider. Technika ta jest przydatna, gdy trzeba obejść problem źle zaprojektowanego typu lub ograniczenia systemu typów .NET (szerzej tym tematem zajmujemy się w rozdziale 19.). Charakterystycznym problemem dotyczącym typów liczbowych jest to, że nie mają żadnego wspólnego interfejsu. Wiemy już, że metody można wiązać dynamicznie. To samo dotyczy operatorów:

```
int x = 3, y = 4;
Console.WriteLine (Mean (x, y));

dynamic Mean (dynamic x, dynamic y) => (x + y) / 2;
```

Korzyści z tego są oczywiste — nie trzeba powielać kodu dla każdego typu liczbowego. Niestety, jednocześnie tracimy statyczne bezpieczeństwo typowe i ryzykujemy, że w czasie działania programu zamiast błędów kompilacji otrzymamy wyjątki.

Wiązanie dynamiczne stanowi obejście statycznej kontroli typów, ale nie kontroli dynamicznej. W odróżnieniu od refleksji (rozdział 18.), za pomocą wiązania dynamicznego nie da się obejść reguł dostępności składowych.

Językowe wiązanie dynamiczne z zasady działa w sposób maksymalnie zbliżony do wiązania statycznego, gdyby typy wykonawcze obiektów dynamicznych były znane w czasie kompilacji. Gdybyśmy w poprzednim przykładzie na stałe wpisali, że metoda Mean operuje na typie int, program działałby dokładnie tak samo. Największa różnica między wiązaniem statycznym i dynamicznym jest widoczna w przypadku metod rozszerzających, o czym więcej piszemy w sekcji „Funkcje, których nie da się wywołać”.

Warto też pamiętać, że wiązanie dynamiczne pogarsza wydajność, choć algorytmy buforowania DLR sprawiają, że kolejne wywołania tego samego wyrażenia dynamicznego są zoptymalizowane, dzięki czemu wyrażenia takie można efektywnie wykonywać w pętli. Optymalizacja ta sprawia, że typowy narzut dla jednego wyrażenia dynamicznego w nowoczesnym sprzęcie wynosi mniej niż 100 ns.

Kup ksi k   Pole ksi k

RuntimeBinderException

Jeśli wiązanie składowej się nie powiedzie, następuje zgłoszenie wyjątku `RuntimeBinderException`. Można go traktować jak błąd kompilacji występujący w czasie działania programu:

```
dynamic d = 5;
d.Hello(); // spowoduje wyjątek RuntimeBinderException
```

Przyczyną wystąpienia wyjątku w tym przykładzie jest brak metody `Hello` w typie `int`.

Reprezentacja typu dynamic w czasie działania programu

Typy `dynamic` i `object` są pod wieloma względami równoznaczne. Na przykład dla systemu wykonawczego prawdziwe jest poniższe wyrażenie:

```
typeof (dynamic) == typeof (object)
```

Zasada ta dotyczy także typów konstruowanych i tablicowych:

```
typeof (List<dynamic>) == typeof (List<object>)
typeof (dynamic[]) == typeof (object[])
```

Referencja dynamiczna, podobnie jak obiektowa, może wskazywać obiekt dowolnego typu (z wyjątkiem typów wskaźnikowych):

```
dynamic x = "cześć";
Console.WriteLine (x.GetType().Name); // String
x = 123; // nie ma błędu (mimo tej samej zmiennej)
Console.WriteLine (x.GetType().Name); // Int32
```

Pod względem strukturalnym referencja obiektowa niczym się nie różni od referencji dynamicznej. Referencja dynamiczna po prostu umożliwia dynamiczne wykonywanie operacji na wskazywanych przez siebie obiektach. Można dokonać konwersji typu `object` na `dynamic`, aby móc wykonywać operacje dynamiczne:

```
object o = new System.Text.StringBuilder();
dynamic d = o;
d.Append ("cześć");
Console.WriteLine (o); // cześć
```

Refleksja w odniesieniu do typu udostępniającego (publiczne) składowe dynamiczne ujawnia, że składowe te są reprezentowane jako obiekty typu `object` z adnotacją:

```
public class Test
{
  public dynamic Foo;
}
```

Ten kod jest równoważny z tym:

```
public class Test
{
  [System.Runtime.CompilerServices.DynamicAttribute]
  public object Foo;
}
```

Dzięki temu konsumenci tego typu wiedzą, że `Foo` należy traktować jako obiekt dynamiczny, a jednocześnie języki nieobsługujące wiązania dynamicznego mogą awaryjnie skorzystać z typu `object`.

Kup ksi k Pole ksi k

Konwersje dynamiczne

Typ `dynamic` obsługuje niejawne konwersje z wszystkimi innymi typami:

```
int i = 7;
dynamic d = i;
long j = d; // nie potrzeba rzutowania (konwersja niejawna)
```

Aby konwersja się powiodła, typ wykonawczy obiektu dynamicznego musi dać się niejawnie przekonwertować na docelowy typ statyczny. Powyższy przykład działa, ponieważ typ `int` można niejawnie przekonwertować na `long`.

W poniższym przykładzie zostaje zgłoszony wyjątek `RuntimeBinderException`, ponieważ typu `int` nie można niejawnie przekonwertować na `short`:

```
int i = 7;
dynamic d = i;
short j = d; // wyjątek RuntimeBinderException
```

Typy var i dynamic

Typy `var` i `dynamic` łączy powierzchowne podobieństwo, ale dzielą je głębokie różnice:

- Typ `var` oznacza: „Niech *kompilator* określi typ".
- Typ `dynamic` oznacza: „Niech *system wykonawczy* określi typ".

Na przykład:

```
dynamic x = "cześć"; // typem statycznym jest dynamic, a typem wykonawczym — string
var y = "cześć";     // typem statycznym i wykonawczym jest string
int i = x;           // błąd wykonawczy (nie można przekonwertować typu string na int)
int j = y;           // błąd kompilacji (nie można przekonwertować typu string na int)
```

Statycznym typem zmiennej zadeklarowanej przy użyciu `var` może być `dynamic`:

```
dynamic x = "cześć";
var y = x;           // typem statycznym y jest dynamic
int z = y;           // błąd wykonawczy (nie można przekonwertować typu string na int)
```

Wyrażenia dynamiczne

Dynamicznie można wywoływać: pola, własności, metody, zdarzenia, konstruktory, indeksatory, operatory i konwersje.

Konsumpcja wyniku wyrażenia dynamicznego przy użyciu typu zwrotnego `void` jest zabroniona, podobnie jak w przypadku wyrażeń typowanych statycznie. Różnica polega na tym, że błąd zostaje zgłoszony w czasie działania programu:

```
dynamic list = new List<int>();
var result = list.Add (5); // wyjątek RuntimeBinderException
```


Kup ksi k Pole ksi k

Wyrażenia zawierające dynamiczne argumenty z reguły same są dynamiczne, ponieważ skutki braku informacji o typie przechodzą kaskadowo:

```
dynamic x = 2;
var y = x * 3; // statycznym typem y jest dynamic
```

Są dwa oczywiste wyjątki od tej reguły. Po pierwsze, rzutowanie wyrażenia dynamicznego na typ statyczny daje wyrażenie statyczne:

```
dynamic x = 2;
var y = (int)x; // typem statycznym y jest int
```

Po drugie, wywołania konstruktora zawsze dają wyrażenia statyczne — nawet w przypadku wywołań z dynamicznymi argumentami. W tym przykładzie zmienna `x` otrzyma statyczny typ `StringBuilder`:

```
dynamic capacity = 10;
var x = new System.Text.StringBuilder (capacity);
```

Ponadto istnieją liczne przypadki brzegowe, w których wyrażenie zawierające dynamiczny argument jest statyczne. Zaliczają się do nich operacje przekazywania indeksu do tablicy i wyrażenia tworzące delegaty.

Wywołania dynamiczne bez dynamicznych odbiorców

Klasyczny przykład użycia typu `dynamic` zakłada istnienie dynamicznego **odbiorcy**. Oznacza to, że dynamiczny obiekt jest odbiorcą dynamicznego wywołania funkcji:

```
dynamic x = ...;
x.Foo(); // x jest odbiorcą
```

Istnieje też możliwość wywoływania funkcji znanych statycznie z dynamicznymi argumentami. Wywołania takie podlegają dynamicznym zasadom rozpoznawania przeciążeń i mogą dotyczyć:

- metod statycznych,
- konstruktorów egzemplarzy,
- metod egzemplarzy na odbiorcach ze statycznie znanym typem.

W poniższym przykładzie konkretna metoda `Foo`, która zostaje dynamicznie związana, jest zależna od typu wykonawczego dynamicznego argumentu:

```
class Program
{
  static void Foo (int x) => Console.WriteLine ("int");
  static void Foo (string x) { Console.WriteLine ("2"); }

  static void Main()
  {
    dynamic x = 5;
    dynamic y = "arbuz";

    Foo (x); // int
    Foo (y); // string
  }
}
```

Jako że w grę nie wchodzi dynamiczny odbiorca, kompilator może statycznie przeprowadzić podstawowy test, aby dowiedzieć się, czy wywołanie dynamiczne się powiedzie. Sprawdza, czy istnieje funkcja o odpowiedniej nazwie i liczbie parametrów. Jeśli takiej nie znajdzie, zgłasza błąd kompilacji. Na przykład:

```
class Program
{
  {static void Foo (int x) => Console.WriteLine ("int");
  static void Foo (string x) => Console.WriteLine ("string");

  static void Main()
  {
    dynamic x = 5;
    Foo (x, x); // błąd kompilatora — nieodpowiednia liczba parametrów
    Fook (x); // błąd kompilatora — nie ma metody o takiej nazwie
  }
}
```

Typy statyczne w wyrażeniach dynamicznych

Jest oczywiste, że typy dynamiczne są używane w wiązaniu dynamicznym. Nie jest natomiast takie oczywiste, że typy statyczne są używane także — jeśli to tylko możliwe — w wiązaniu dynamicznym. Spójrz na poniższy przykład:

```
class Program
{
  static void Foo (object x, object y) { Console.WriteLine ("oo"); }
  static void Foo (object x, string y) { Console.WriteLine ("os"); }
  static void Foo (string x, object y) { Console.WriteLine ("so"); }
  static void Foo (string x, string y) { Console.WriteLine ("ss"); }

  static void Main()
  {
    object o = "cześć";
    dynamic d = "żegnaj";
    Foo (o, d); // os
  }
}
```

Wywołanie `Foo(o,d)` podlega wiązaniu dynamicznemu, ponieważ jeden z jego argumentów (`d`) jest dynamiczny. Jednak, jako że typ `o` jest znany statycznie, zostanie wykorzystany w wiązaniu, mimo że jest ono przeprowadzane dynamicznie. W tym przypadku, zgodnie z zasadami rozpoznawania przeciążenia, zostanie wybrana druga implementacja metody `Foo` ze względu na statyczny typ `o` i wykonawczy typ `d`. Innymi słowy: kompilator „jest tak statyczny, jak to możliwe".

Funkcje, których nie da się wywołać

Niektórych funkcji nie można wywoływać dynamicznie. Należą do nich:

- metody rozszerzające (utworzone za pomocą składni metod rozszerzających);
- składowe interfejsu, jeśli trzeba dokonać rzutowania na ten interfejs;
- składowe klasy bazowej schowane przez podklasę.

Kup ksi k   Pole ksi k

Wiedza, dlaczego tak jest, pomaga też w zrozumieniu wiązania dynamicznego.

Wiązanie dynamiczne wymaga podania dwóch informacji: nazwy funkcji do wywołania i obiektu, na którym ta funkcja ma zostać wywołana. Ale we wszystkich trzech przypadkach wymienionych na liście w grę wchodzi jeszcze **dodatkowy typ**, który jest znany tylko w czasie kompilacji. Obecnie nie ma możliwości dynamicznego określenia tych dodatkowych typów.

Przy wywoływaniu metod rozszerzających dodatkowy typ jest niejawny. Jest to statyczna klasa, dla której została zdefiniowana metoda. Kompilator szuka jej na podstawie tego, co dołączono do kodu źródłowego za pomocą dyrektyw `using`. Jako że dyrektywy `using` są usuwane w procesie kompilacji (po tym, jak zostaną wykorzystane do powiązania prostych nazw z pełnymi nazwami zawierającymi przestrzenie nazw), metody rozszerzające nie mogą być wykorzystywane dynamicznie.

Przy wywoływaniu składowych przez interfejs określamy dodatkowy typ poprzez jawne lub niejawne rzutowanie. Może to być konieczne w dwóch sytuacjach: przy wywoływaniu składowych jawnie zaimplementowanego interfejsu i przy wywoływaniu składowych interfejsu zaimplementowanego w typie należącym do innego zestawu. Pierwszy przypadek ilustrują dwa poniższe typy:

```
interface IFoo { void Test(); }
class Foo : IFoo { void IFoo.Test() {} }
```

Aby wywołać metodę `Test`, należy dokonać rzutowania na interfejs `IFoo`. Jest to łatwe w przypadku typowania statycznego:

```
IFoo f = new Foo(); // niejawne rzutowanie na interfejs
f.Test();
```

A teraz spójrz, co się zmieni, gdy zastosujemy typowanie dynamiczne:

```
IFoo f = new Foo();
dynamic d = f;
d.Test(); // wyjątek
```

Oznaczone pogrubieniem niejawne rzutowanie nakazuje *kompilatorowi* wiązanie dalszych wywołań składowych `f` z `IFoo`, a nie `Foo` — innymi słowy: chcemy, aby kompilator „patrzył" na obiekt przez pryzmat interfejsu `IFoo`. Problem w tym, że pryzmat ten znika w czasie wykonywania programu, więc DLR nie może wykonać wiązania. Ta strata jest pokazana poniżej:

```
Console.WriteLine (f.GetType().Name); // Foo
```

Podobna sytuacja zachodzi przy wywoływaniu schowanej składowej typu bazowego: należy określić dodatkowy typ przez rzutowanie lub za pomocą słowa kluczowego `base` i typ ten zostanie utracony w czasie wykonywania programu.

Przeciążanie operatorów

Operatory można przeciążać w celu zapewnienia bardziej naturalnej składni dla typów własnych programisty. Technikę przeciążania operatorów najlepiej jest stosować przy implementacji własnych struktur reprezentujących dość proste typy danych. Doskonałym kandydatem do przeciążenia operatorów jest własny typ liczbowy programisty.

Kup ksi k Pole ksi k

Przeciążać można poniższe operatory symboliczne:

```
+ (jednoargumentowy)  - (jednoargumentowy)  !   ~  ++
--                    +                     -   *  /
%                     &                     |   ^  <<
>>                    ==                    !=  >  <
>=                    <=
```

Ponadto można przeciążać następujące operatory:

- jawnej i niejawnej konwersji (przy użyciu słów kluczowych `implicit` i `explicit`);
- *operatory* (nie *literały*) `true` i `false`.

Poniższe operatory są przeciążane pośrednio:

- Złożone operatory przypisania (np. +=, /=) są przeciążane niejawnie w efekcie przeciążenia operatorów prostych (np. +, /).
- Operatory warunkowe `&&` i `||` są przeciążane niejawnie w efekcie nadpisania bitowych operatorów `&` i `|`.

Funkcje operatorowe

Przeciążanie operatora polega na zadeklarowaniu **funkcji operatorowej**, która musi spełniać następujące warunki:

- Nazwę w postaci symbolu operatora należy podać po słowie kluczowym `operator`.
- Funkcja operatorowa musi być statyczna i publiczna.
- Parametry funkcji operatorowej reprezentują argumenty.
- Typ zwrotny funkcji operatorowej reprezentuje wynik wyrażenia.
- Przynajmniej jeden argument musi być tego samego typu, w którym zadeklarowano funkcję operatorową.

Poniżej znajduje się przykład definicji struktury o nazwie `Note` reprezentującej nutę, w której to strukturze przeciążyliśmy operator +:

```
public struct Note
{
  int value;
  public Note (int semitonesFromA) { value = semitonesFromA; }
  public static Note operator + (Note x, int semitones)
  {
    return new Note (x.value + semitones);
  }
}
```

Dzięki temu możemy dodawać wartości typu `int` do `Note`:

```
Note B = new Note (2);
Note CSharp = B + 2;
```

Przeciążenie operatora powoduje automatyczne przeciążenie odpowiadającego mu złożonego operatora przypisania. W naszym przykładzie przeciążyliśmy operator +, więc możemy używać także +=:

```
CSharp += 2;
```


Kup ksi k   Pole ksi k

W C# funkcje operatorowe składające się z tylko jednego wyrażenia, podobnie jak metody i własności, można zapisywać zwięźlej przy użyciu składni wyrażeniowej:

```
public static Note operator + (Note x, int semitones)
                              => new Note (x.value + semitones);
```

Przeciążanie operatorów równości i porównywania

Operatory równości i porównywania czasami przesłania się w strukturach i rzadziej w klasach. Z przeciążaniem tych rodzajów operatorów wiążą się pewne specjalne zasady i obowiązki, które opisujemy w rozdziale 6. Poniżej przedstawiamy tylko ich zestawienie:

Parzystość

Kompilator C# wymaga definicji obu operatorów stanowiących logiczne pary: `==` i `!=`, `<` i `>` oraz `<=` i `=>`.

Metody `Equals` **i** `GetHasCode`

W większości przypadków przeciążenie operatorów `==` i `!=` pociąga za sobą konieczność dodatkowego przeciążenia metod `Equals` i `GetHashCode` zdefiniowanych w klasie `object`. Jeśli się tego nie zrobi, kompilator zgłosi ostrzeżenie (zob. podrozdział „Porównywanie łańcuchów” w rozdziale 6.).

Interfejsy `IComparable` **i** `IComparable<T>`

Jeśli przeciąży się operatory `<` i `>` oraz `<=` i `=>`, to należy zaimplementować interfejsy `IComparable` i `IComparable<T>`.

Niestandardowe konwersje jawne i niejawne

Konwersje jawne i niejawne to także operatory, które można przeciążać. Konwersje te zazwyczaj przeciąża się po to, by umożliwić zwięzłe i naturalne konwertowanie ściśle powiązanych ze sobą typów (np. liczbowych).

Jeśli trzeba dokonać konwersji między słabo spokrewnionymi typami, bardziej odpowiednie są następujące techniki:

- napisanie konstruktora z parametrem typu, na który ma być dokonywana konwersja;
- napisanie metod `To`*`XXX`* i (statycznej) `From`*`XXX`* do konwersji między typami.

Jak wyjaśniliśmy w części poświęconej typom, konwersje niejawne wykonuje się wtedy, gdy wiadomo, że zawsze się udadzą i nie przyczynią się do utraty informacji. Natomiast konwersje jawne powinno się przeprowadzać, gdy o powodzeniu operacji decydują warunki panujące w czasie działania programu lub gdy może dojść do utraty informacji.

W poniższym przykładzie definiujemy konwersje między typem `Note` (reprezentującym nuty) a typem `double` (reprezentującym częstotliwość w hercach danej nuty):

```
...
// konwersja na herce
public static implicit operator double (Note x)
=> 440 * Math.Pow (2, (double) x.value / 12 );
```


Kup książkę   Poleć książkę

```
// konwersja z herców (z dokładnością do najbliższego półtonu)
public static explicit operator Note (double x)
=> new Note ((int) (0.5 + 12 * (Math.Log (x/440) / Math.Log(2) ) ));
...

Note n = (Note)554.37; // konwersja jawna
double x = n;          // konwersja niejawna
```

Zgodnie z naszymi własnymi wskazówkami w tym przykładzie lepiej byłoby zaimplementować metodę `ToFrequency` (i statyczną metodę `FromFrequency`) zamiast operatorów jawnej i niejawnej konwersji.

Konwersje niestandardowe są ignorowane przez operatory `as` i `is`:

```
Console.WriteLine (554.37 is Note); // fałsz
Note n = 554.37 as Note;            // błąd
```

Przeciążanie true i false

Operatory `true` i `false` przeciąża się w bardzo rzadkich przypadkach, gdy typy będące „z natury" logiczne nie mają konwersji na typ `bool`. Przykładem takiej sytuacji jest typ implementujący logikę trójstanową: dzięki przeciążeniu `true` i `false` taki typ może płynnie współpracować z instrukcjami warunkowymi i operatorami `if`, `do`, `while`, `for`, `&&`, `||` oraz `?:`. Taką funkcjonalność zapewnia struktura `System.Data.SqlTypes.SqlBoolean`. Na przykład:

```
SqlBoolean a = SqlBoolean.Null;
if (a)
  Console.WriteLine ("True");
else if (!a)
  Console.WriteLine ("False");
else
  Console.WriteLine ("Null");

Wynik:
Null
```

Poniżej znajduje się nowa implementacja części struktury `SqlBoolean` potrzebnych do zademonstrowania operatorów `true` i `false`:

```
public struct SqlBoolean
{
  public static bool operator true (SqlBoolean x)
   => x.m_value == True.m_value;

  public static bool operator false (SqlBoolean x)
   => x.m_value == False.m_value;

  public static SqlBoolean operator ! (SqlBoolean x)
  {
    if (x.m_value == Null.m_value) return Null;
    if (x.m_value == False.m_value) return True;
    return False;
  }

  public static readonly SqlBoolean Null  = new SqlBoolean(0);
  public static readonly SqlBoolean False = new SqlBoolean(1);
```


Kup ksi k   Pole ksi k

```
    public static readonly SqlBoolean True  = new SqlBoolean(2);

    private SqlBoolean (byte value) { m_value = value; }
    private byte m_value;
  }
```

Niebezpieczny kod i wskaźniki

W języku C# istnieje możliwość bezpośredniej pracy z pamięcią za pomocą wskaźników, których można używać w blokach kodu oznaczonych jako niebezpieczne i które kompiluje się przy użyciu opcji `/unsafe` kompilatora. Głównym zastosowaniem typów wskaźnikowych jest umożliwienie współpracy z interfejsami API języka C, choć można ich też używać do uzyskiwania dostępu do pamięci spoza sterty zarządzanej i w celu optymalizacji szybkości działania niektórych części programu.

Podstawowe wiadomości o wskaźnikach

Dla każdego typu wartościowego i referencyjnego **V** istnieje odpowiadający mu typ **V***. Wskaźnik zawiera adres zmiennej. Typy wskaźnikowe można (niebezpiecznie) rzutować na wszystkie inne typy wskaźnikowe. Najważniejsze operatory do pracy ze wskaźnikami to:

Operator	Opis
`&`	Operator **adresowania** zwracający wskaźnik do adresu zmiennej
`*`	Operator **dereferencji** zwracający zmienną zapisaną pod adresem przechowywanym we wskaźniku
`->`	Operator **wskaźnika do składowej** to syntaktyczny skrót taki, że `x -> y` jest równoważne z `(*x).y`

Kod niebezpieczny

Oznaczając typ, składową typu lub blok instrukcji słowem kluczowym `unsafe`, programista zapewnia sobie możliwość używania typów wskaźnikowych i wykonywania na pamięci operacji w stylu języka C++ w danym zakresie. Oto przykład szybkiego przetwarzania mapy bitowej za pomocą wskaźników:

```
unsafe void BlueFilter (int[,] bitmap)
{
  int length = bitmap.Length;
  fixed (int* b = bitmap)
  {
    int* p = b;
    for (int i = 0; i < length; i++)
    *p++ &= 0xFF;
  }
}
```

Niebezpieczny kod może działać szybciej od odpowiedniej bezpiecznej implementacji. W tym przypadku konieczne byłoby napisanie zagnieżdżonej pętli z indeksowaniem tablicy i sprawdzaniem granic. Niebezpieczna metoda C# może też być szybsza niż wywołanie zewnętrznej funkcji C, ponieważ nie jest obciążona narzutem związanym z opuszczaniem zarządzanego środowiska wykonawczego.

Kup ksi k   Pole ksi k

Instrukcja fixed

Instrukcja fixed służy do przypinania obiektów zarządzanych, jak to zrobiono z mapą bitową w poprzednim przykładzie. Gdy program działa, na stercie alokowanych i dezalokowanych jest wiele obiektów. Aby uniknąć marnowania i fragmentacji pamięci, system usuwania nieużytków przesuwa obiekty. Zapisywanie wskaźnika do obiektu, którego adres może się zmienić, byłoby pozbawione sensu, dlatego należy używać instrukcji fixed, która nakazuje systemowi usuwania nieużytków „przypiąć" obiekt w miejscu i nigdzie go nie przenosić. Może to mieć wpływ na efektywność systemu wykonawczego, więc zaleca się, aby bloków przypiętych używać tylko przez krótki czas oraz by unikać w nich alokacji na stercie.

W instrukcji fixed można utworzyć wskaźnik do obiektu dowolnego typu wartościowego, tablicy wartości typów wartościowych lub łańcucha. W przypadku tablic i łańcuchów wskaźnik wskazuje pierwszy element, który jest typu wartościowego.

Typy wartościowe zadeklarowane wewnątrz typów referencyjnych wymagają, aby typ referencyjny był przypięty, jak pokazano poniżej:

```
class Test
{
  int x;
  static void Main()
  {
    Test test = new Test();
    unsafe
    {
      fixed (int* p = &test.x) // przypina test
      {
        *p = 9;
      }
      System.Console.WriteLine (test.x);
    }
  }
}
```

Bardziej szczegółowy opis instrukcji fixed zamieściliśmy w podrozdziale „Mapowanie struktury na pamięć niezarządzaną" w rozdziale 24.

Operator wskaźnika do składowej

Oprócz operatorów & i *, w języku C# dostępny jest też znany z C++ operator ->, którego można używać w odniesieniu do struktur:

```
struct Test
{
  int x;
  unsafe static void Main()
  {
    Test test = new Test();
    Test* p = &test;
    p->x = 9;
    System.Console.WriteLine (test.x);
  }
}
```


Kup ksi k Pole ksi k

Słowo kluczowe stackalloc

Za pomocą słowa kluczowego `stackalloc` można alokować pamięć w blokach bezpośrednio na stosie. Alokacja na stosie oznacza, że obiekt istnieje tylko przez czas działania metody, tak jak w przypadku innych zmiennych lokalnych (których czas istnienia nie został przedłużony przez wyrażenie lambda, blok iteratora ani funkcję asynchroniczną). W takim bloku można używać operatora [] do indeksowania pamięci:

```
int* a = stackalloc int [10];
for (int i = 0; i < 10; ++i)
  Console.WriteLine (a[i]); // drukuje nieprzetworzoną zawartość pamięci
```

W rozdziale 23. opisujemy sposób użycia `Span<T>` do zarządzania pamięcią alokowaną na stosie bez używania słowa kluczowego `unsafe`:

```
Span<int> a = stackalloc int [10];
for (int i = 0; i < 10; ++i)
    Console.WriteLine (a[i]);
```

Bufory o stałym rozmiarze

Słowo kluczowe `fixed` ma też inne zastosowanie. Przy jego użyciu można tworzyć w strukturach bufory o stałym rozmiarze (taka możliwość bywa przydatna przy wywoływaniu funkcji niezarządzanych — zobacz rozdział 24.):

```
unsafe struct UnsafeUnicodeString
{
  public short Length;
  public fixed byte Buffer[30]; // alokuje blok 30 bajtów
}

unsafe class UnsafeClass
{
  UnsafeUnicodeString uus;

  public UnsafeClass (string s)
  {
    uus.Length = (short)s.Length;
    fixed (byte* p = uus.Buffer)
      for (int i = 0; i < s.Length; i++)
        p[i] = (byte) s[i];
  }
}
class Test
{
  static void Main() { new UnsafeClass ("Christian Troy"); }
}
```

Bufory o stałym rozmiarze nie są tablicami: gdyby konstrukcja `Buffer` była tablicą, zawierałaby referencję do obiektu przechowywanego na (niezarządzanej) stercie, a nie 30 bajtów danych w samej strukturze.

W tym przykładzie słowo kluczowe `fixed` zostało też użyte do przypięcia obiektu na stercie zawierającej bufor (którym będzie egzemplarz klasy `UnsafeClass`). Zatem słowo kluczowe `fixed` ma dwa znaczenia: stały *rozmiar* lub niezmienne *miejsce*. Często używa się go w obu tych zastosowaniach jednocześnie, tzn. do tworzenia buforów o niezmiennym rozmiarze, które muszą być przypięte do jednego miejsca.

Kup ksi k Pole ksi k

Wskaźnik pusty

Wskaźnik pusty (`void*`) jest pozbawiony wszelkich założeń co do typu danych, więc może być wykorzystywany w funkcjach pracujących na surowej pamięci. Każdy typ wskaźnikowy może być niejawnie przekonwertowany na typ `void*`. Zmiennej tego typu nie można poddać dereferencji i nie można przy jej użyciu wykonywać operacji arytmetycznych. Na przykład:

```
class Test
{
  unsafe static void Main()
  {
    short[ ] a = {1,1,2,3,5,8,13,21,34,55};
    fixed (short* p = a)
    {
      // sizeof zwraca rozmiar typu wartościowego w bajtach
      Zap (p, a.Length * sizeof (short));
    }
    foreach (short x in a)
      System.Console.WriteLine (x); // drukuje same zera
  }

  unsafe static void Zap (void* memory, int byteCount)
  {
    byte* b = (byte*) memory;
      for (int i = 0; i < byteCount; i++)
        *b++ = 0;
  }
}
```

Wskaźniki do funkcji (C# 9)

Wskaźnik do funkcji jest jak delegat, ale bez charakterystycznej dla jego egzemplarza pośredniości, ponieważ wskazuje bezpośrednio metodę. Wskaźnik do funkcji może wskazywać tylko metody statyczne, nie obsługuje multiemisji oraz wymaga niebezpiecznego kontekstu (pomija bowiem bezpieczeństwo typów w środowisku wykonawczym). Jego głównym zastosowaniem jest uproszczenie i optymalizacja współpracy z niezarządzanymi API (podrozdział „Wywołania zwrotne z kodu niezarządzanego" w rozdziale 24.).

Deklaracja typu wskaźnika do funkcji wygląda tak (typ zwrotny jest podany na końcu):

```
delegate*<int, char, string, void> // (void oznacza typ zwrotny)
```

Ten wskaźnik pasuje do funkcji o następującej sygnaturze:

```
void SomeFunction (int x, char y, string z)
```

Operator `&` tworzy wskaźnik do funkcji z grupy metod. Oto kompletny przykład:

```
unsafe
{
    delegate*<string, int> functionPointer = &GetLength;
    int length = functionPointer ("Witaj, świecie");
    static int GetLength (string s) => s.Length;
}
```


Kup ksi k Pole ksi k

W tym przykładzie `functionPointer` nie jest *obiektem*, na którym można wywołać metodę, taką jak `Invoke` (lub z referencją do obiektu `Target`). Jest to zmienna wskazująca bezpośrednio adres metody docelowej w pamięci:

```
Console.WriteLine ((IntPtr)functionPointer);
```

Jak każdy inny wskaźnik nie podlega kontroli typów w środowisku wykonawczym. Poniżej wartość zwrotna naszej funkcji jest traktowana jako wartość dziesiętna (która jest dłuższa od typu `int`, więc powoduje wprowadzenie losowej porcji pamięci do wyniku):

```
var pointer2 = (delegate*<string, decimal>) (IntPtr) functionPointer;
Console.WriteLine (pointer2 ("Witaj, niebezpieczny świecie."));
```

[SkipLocalsInit] (C# 9)

Kiedy C# kompiluje metodę, emituje flagę nakazującą środowisku wykonawczemu inicjalizację lokalnych zmiennych tej metody ich wartościami domyślnymi (przez wyzerowanie pamięci). Od C# 9 programista może nakazać kompilatorowi zaniechanie emisji tej flagi. Służy do tego atrybut `[SkipLocalsInit]` (z przestrzeni nazw `System.Runtime.CompilerServices`), który należy przypisać metodzie:

```
[SkipLocalsInit]
void Foo() ...
```

Atrybut ten można także przypisać typowi — co jest równoznaczne z przypisaniem go wszystkim metodom tego typu — a nawet całemu modułowi (kontenerowi zestawu):

```
[module: System.Runtime.CompilerServices.SkipLocalsInit]
```

W normalnym bezpiecznym kontekście atrybut `[SkipLocalsInit]` ma niewielki wpływ na funkcjonalność lub wydajność, ponieważ obowiązująca w C# zasada sprecyzowanego przypisania wymaga jawnego przypisania wartości zmiennym lokalnym przed ich odczytem. To znaczy, że optymalizator JIT może wyemitować taki sam kod maszynowy niezależnie od tego, czy ten atrybut został zastosowany, czy nie.

Natomiast w niebezpiecznym kontekście użycie atrybutu `[SkipLocalsInit]` może uchronić CLR przed inicjalizacją typizowanych wartościowo zmiennych lokalnych, co daje drobną korzyść pod względem wydajności w przypadku metod intensywnie wykorzystujących stos (przez `stackalloc`). Poniższy kod drukuje niezainicjalizowaną pamięć, gdy zostanie zastosowany atrybut `[SkipLocalsInit]` (zamiast samych zer):

```
[SkipLocalsInit]
unsafe void Foo()
{
    int local;
    int* ptr = &local;
    Console.WriteLine (*ptr);
    int* a = stackalloc int [100];
    for (int i = 0; i < 100; ++i) Console.WriteLine (a [i]);
}
```


Kup ksi k		Pole ksi k

Co ciekawe, ten sam efekt można uzyskać w „bezpiecznym” kontekście dzięki użyciu `Span<T>`:

```
[SkipLocalsInit]
void Foo()
{
    Span<int> a = stackalloc int [100];
    for (int i = 0; i < 100; ++i) Console.WriteLine (a [i]);
}
```

W konsekwencji użycie atrybutu `[SkipLocalsInit]` wymaga kompilacji zestawu z opcją `unsafe` — nawet jeśli żadna z metod nie jest oznaczona jako *niebezpieczna*.

Dyrektywy preprocesora

Dyrektywy preprocesora dostarczają kompilatorowi dodatkowych informacji o pewnych obszarach kodu. Najczęściej używane są dyrektywy warunkowe umożliwiające dodawanie do lub wykluczanie z kompilacji fragmentów kodu. Na przykład:

```
#define DEBUG
class MyClass
{
  int x;
  void Foo()
  {
    #if DEBUG
    Console.WriteLine ("Testowanie: x = {0}", x);
    #endif
  }
  ...
}
```

W tej klasie kompilacja instrukcji w metodzie `Foo` jest uzależniona od obecności symbolu `DEBUG`. Jeśli go usuniemy, ta instrukcja nie zostanie skompilowana. Symbole preprocesora można definiować w plikach źródłowych (jak w tym przypadku) lub na poziomie projektu w pliku *.csproj*:

```
<PropertyGroup>
    <DefineConstants>DEBUG;ANOTHERSYMBOL</DefineConstants>
</PropertyGroup>
```

W dyrektywach `#if` i `#elif` można używać operatorów `||`, `&&` i `!` reprezentujących operacje logiczne *alternatywy*, *koniunkcji* i *negacji*. Poniższa dyrektywa nakazuje kompilatorowi dodanie znajdującego się pod nią kodu, jeśli symbol `TESTMODE` jest zdefiniowany, a symbol `DEBUG` nie jest:

```
#if TESTMODE && !DEBUG
  ...
```

Należy jednak pamiętać, że nie jest to zwykłe wyrażenie C#, a używane symbole nie mają nic wspólnego ze *zmiennymi* — statycznymi ani żadnymi innymi.

Symbole `#error` i `#warning` zapobiegają przypadkowemu niewłaściwemu użyciu dyrektyw warunkowych przez zmuszenie kompilatora do wygenerowania błędu lub ostrzeżenia w przypadku podania nieodpowiedniego zestawu symbolu kompilacji. W tabeli 4.1 znajduje się lista dostępnych dyrektyw preprocesora.

Kup ksi k Pole ksi k

Tabela 4.1. Dyrektywy preprocesora

<table>
<tr><th>Dyrektywa preprocesora</th><th>Opis</th></tr>
<tr><td><code>#define</code> symbol</td><td>Definiuje symbol</td></tr>
<tr><td><code>#undef</code> symbol</td><td>Usuwa definicję symbolu</td></tr>
<tr><td><code>#if</code> symbol [operator symbol2]...</td><td>symbol do przetestowania
Dostępne operatory: <code>==</code>, <code>!=</code>, <code>&&</code> oraz <code>||</code>; po nich może się znajdować symbol <code>#else</code>, <code>#elif</code> oraz <code>#endif</code></td></tr>
<tr><td><code>#else</code></td><td>Wykonuje kod do najbliższego <code>#endif</code></td></tr>
<tr><td><code>#elif</code> symbol [operator symbol2]</td><td>Łączy gałąź <code>#else</code> i test <code>#if</code></td></tr>
<tr><td><code>#endif</code></td><td>Kończy dyrektywy warunkowe</td></tr>
<tr><td><code>#warning</code> tekst</td><td>Tekst ostrzeżenia mający się pojawić w danych zwróconych przez kompilator</td></tr>
<tr><td><code>#error</code> tekst</td><td>Tekst błędu mający się pojawić w danych zwróconych przez kompilator</td></tr>
<tr><td><code>#error version</code></td><td>Informuje kompilator o wersji i kończy działanie</td></tr>
<tr><td><code>#pragma warning [disable | restore]</code></td><td>Wyłącza lub przywraca ostrzeżenia kompilatora</td></tr>
<tr><td><code>#line [</code> liczba <code>["</code>plik<code>"] | hidden]</code></td><td>Liczba określa numer wiersza w kodzie źródłowym. Parametr plik określa nazwę pliku, jaka ma się pojawić w danych zwróconych. Parametr <code>hidden</code> nakazuje debuggerom pominięcie kodu od tego miejsca do następnej dyrektywy <code>#line</code></td></tr>
<tr><td><code>#region</code> nazwa</td><td>Oznacza początek obszaru</td></tr>
<tr><td><code>#endregion</code></td><td>Oznacza koniec obszaru</td></tr>
<tr><td><code>#nullable</code> opcja</td><td>Zobacz podrozdział „Typy referencyjne dopuszczające wartość null”</td></tr>
</table>

Atrybuty warunkowe

Atrybut oznaczony atrybutem `Conditional` zostanie skompilowany tylko wtedy, gdy obecny będzie określony symbol preprocesora. Na przykład:

```
// file1.cs
#define DEBUG
using System;
using System.Diagnostics;
[Conditional("DEBUG")]
public class TestAttribute : Attribute {}

// file2.cs
#define DEBUG
[Test]
class Foo
{
  [Test]
  string s;
}
```

Kompilator doda do programu atrybuty `[Test]` tylko wtedy, gdy symbol `DEBUG` będzie w zakresie dostępności pliku *file2.cs*.

Kup ksi k Pole ksi k

Ostrzeżenia pragma

Kompilator zgłasza ostrzeżenia, gdy znajdzie w kodzie źródłowym coś podejrzanego. W odróżnieniu od błędów, ostrzeżenia normalnie nie uniemożliwiają kompilacji całego programu.

Ostrzeżenia kompilatora bardzo pomagają w wykrywaniu usterek w kodzie. Jeśli jednak jest dużo *fałszywych* alarmów, ich przydatność znacznie spada. W dużych aplikacjach, jeśli ostrzeżenia mają prawidłowo pełnić swoją funkcję, musi być zachowany odpowiedni stosunek „sygnału do szumu".

Dlatego stworzono dyrektywy kompilatora `#pragma` umożliwiające tłumienie wybranych ostrzeżeń. W poniższym przykładzie informujemy kompilator, że nie życzymy sobie ostrzeżeń o tym, że pole `Message` jest nieużywane:

```
public class Foo
{
  static void Main() { }
  #pragma warning disable 414
  static string Message = "Cześć";
  #pragma warning restore 414
}
```

Jeśli w dyrektywie `#pragma warning` nie ma żadnej liczby, to znaczy, że mają zostać wyłączone lub przywrócone wszystkie kody ostrzeżeń.

Jeśli stosuje się wiele takich dyrektyw, to można użyć przełącznika kompilacji `/warnaserror`, który nakazuje kompilatorowi traktowanie wszystkich pozostałych ostrzeżeń jako błędy.

Dokumentacja XML

Komentarz dokumentacyjny to osadzone w kodzie źródłowym programu dane w formacie XML stanowiące dokumentację typu lub składowej. Komentarze takie umieszcza się bezpośrednio przed deklaracją typu lub składowej, a ich początek wyznaczają trzy ukośniki:

```
/// <summary>Anuluje działające zapytanie.</summary>
public void Cancel() { ... }
```

Komentarze wielowierszowe można tworzyć tak:

```
/// <summary>
/// Anuluje działające zapytanie.
/// </summary>
public void Cancel() { ... }
```

Albo tak (zwróć uwagę na dodatkową gwiazdkę na początku):

```
/**
  <summary>Anuluje działające zapytanie.</summary>
*/
public void Cancel() { ... }
```


Kup ksi k Pole ksi k

Jeśli do pliku *.csproj* dodasz poniższą opcję:

```
<PropertyGroup>
    <DocumentationFile>SoneFile.xnl</DocumentationFile>
</PropertyGroup>
```

kompilator pobierze i połączy komentarze dokumentacyjne w określonym pliku XML, który można wykorzystać na dwa sposoby:

- Jeśli plik XML zostanie umieszczony w tym samym folderze co skompilowany zestaw, Visual Studio (i LINQPad) automatycznie go odczyta i wykorzysta zawarte w nim informacje w funkcji IntelliSense do wyświetlenia listy składowych.
- Narzędzia innych producentów (np. Sandcastle i NDoc) mogą przekonwertować dokument XML na plik pomocy HTML.

Standardowe znaczniki dokumentacyjne XML

Oto lista standardowych znaczników XML rozpoznawanych przez Visual Studio i generatory dokumentacji:

`<summary>`

```
<summary>...</summary>
```

Zawiera treść chmurki, którą funkcja IntelliSense powinna wyświetlić dla typu lub składowej. Najczęściej jest to pojedyncze wyrażenie lub zdanie.

`<remarks>`

```
<remarks>...</remarks>
```

Dodatkowy opis typu lub składowej. Generatory dokumentacji dołączają go do ogólnego opisu typu lub składowej.

`<param>`

```
<param name="nazwa">...</param>
```

Objaśnia zastosowanie parametru metody.

`<returns>`

```
<returns>...</returns>
```

Objaśnia wartość zwrotną metody.

`<exception>`

```
<exception [cref="typ"]>...</exception>
```

Przedstawia wyjątek, jaki może zgłosić metoda (`cref` oznacza typ wyjątku).

`<example>`

```
<example>...</example>
```

Zawiera przykład (używany przez generatory dokumentacji). Przykład ten zazwyczaj zawiera zarówno opis tekstowy, jak i kod źródłowy (kod najczęściej znajduje się w znaczniku `<c>` lub `<code>`).

Kup ksi k

Pole ksi k

<c>

```
<c>...</c>
```

Oznacza fragment kodu w tekście. Znacznik ten jest najczęściej używany w blokach <example>.

<code>

```
<code>...</code>
```

Oznacza wielowierszowy przykład kodu. Znacznik ten jest najczęściej używany w blokach <example>.

<see>

```
<see cref="składowa">...</see>
```

Oznacza odwołanie do innego typu lub składowej. Generatory dokumentacji HTML najczęściej zamieniają je na hiperłącza. Kompilator generuje ostrzeżenie, jeśli typ lub nazwa składowej są niepoprawne. Odwołania do typów generycznych umieszcza się w klamrze, np. `cref="Foo{T,U}"`.

<seealso>

```
<seealso cref="składowa">...</seealso>
```

Oznacza odwołanie do innego typu lub składowej. Generatory dokumentacji najczęściej przenoszą je do osobnej sekcji *Zobacz również* umieszczonej na dole strony.

<paramref>

```
<paramref name="nazwa"/>
```

Odniesienie do parametru z wnętrza znacznika <summary> lub <remarks>.

```
<list>
<list type=[ bullet | number | table ]>
  <listheader>
    <term>...</term>
    <description>...</description>
  </listheader>
  <item>
    <term>...</term>
    <description>...</description>
  </item>
</list>
```

Nakazuje generatorom dokumentacji wygenerowanie listy punktowanej, numerowanej lub tabelarycznej.

<para>

```
<para>...</para>
```

Nakazuje generatorom dokumentacji zapisanie treści w osobnym akapicie.

<include>

```
<include file='nazwapliku' path='ścieżkadoznacznika[@name="id"]'>...</include>
```

Dołącza zewnętrzny plik XML zawierający dokumentację. Atrybut `path` oznacza zapytanie XPath do konkretnego elementu w tym pliku.

Kup ksi k Pole ksi k

Znaczniki niestandardowe

Standardowe znaczniki XML rozpoznawane przez kompilator C# nie są pod żadnym względem wyjątkowe i programista w razie potrzeby może też zdefiniować własne znaczniki. Specjalnie traktowane są tylko znacznik `<param>` — kompilator sprawdza nazwę parametru oraz weryfikuje, czy wszystkie parametry metody mają dokumentację — i atrybut `cref` — kompilator sprawdza, czy ten atrybut odnosi się do istniejącego typu lub składowej, oraz rozwija ich nazwę do pełnej postaci. Atrybutu `cref` można też używać we własnych znacznikach — jest on wówczas weryfikowany oraz rozwijany tak samo jak w standardowych znacznikach `<exception>`, `<permission>`, `<see>` i `<seealso>`.

Odwołania do typów i składowych

Nazwy typów i odwołania do typów oraz składowych są zamieniane na identyfikatory jednoznacznie definiujące typ lub składową. Nazwy te składają się z przedrostka określającego, co reprezentuje dany identyfikator, oraz sygnatury typu lub składowej. W poniższej tabeli znajduje się wykaz możliwych przedrostków.

Przedrostek typu XML	Reprezentowana konstrukcja
N	Przestrzeń nazw
T	Typ (klasa, struktura, wyliczenie, interfejs, delegat)
F	Pole
P	Własność (dotyczy także indeksatorów)
M	Metoda (dotyczy także metod specjalnych)
E	Zdarzenie
!	Błąd

Reguły generowania sygnatur zostały dokładnie opisane, ale są dość skomplikowane.

Oto przykład typu z wygenerowanymi dla niego identyfikatorami:

```
// przestrzenie nazw nie mają własnych sygnatur
namespace NS
{
  /// T:NS.MyClass
  class MyClass
  {
    /// F:NS.MyClass.aField
    string aField;

    /// P:NS.MyClass.aProperty
    short aProperty {get {...} set {...}}

    /// T:NS.MyClass.NestedType
    class NestedType {...};

    /// M:NS.MyClass.X()
    void X() {...}
```


Kup ksi k   Pole ksi k

```
    /// M:NS.MyClass.Y(System.Int32,System.Double@,System.Decimal@)
    void Y(int p1, ref double p2, out decimal p3) {...}

    /// M:NS.MyClass.Z(System.Char[ ],System.Single[0:,0:])
    void Z(char[ ] 1, float[,] p2) {...}

    /// M:NS.MyClass.op_Addition(NS.MyClass,NS.MyClass)
    public static MyClass operator+(MyClass c1, MyClass c2) {...}

    /// M:NS.MyClass.op_Implicit(NS.MyClass)~System.Int32
    public static implicit operator int(MyClass c) {...}

    /// M:NS.MyClass.#ctor
    MyClass() {...}

    /// M:NS.MyClass.Finalize
    ~MyClass() {...}

    /// M:NS.MyClass.#cctor
    static MyClass() {...}
  }
}
```


Kup ksi k Pole ksi k

Skorowidz

.NET 5, 22, 255
.NET Framework, 23
.NET Standard, 255– 257

A

abstrakcyjne
  klasy, 130
  składowe, 130
ACL, 689
adaptery strumienia, 661
  binarne, 666
    BinaryReader, 666
    BinaryWriter, 666
  tekstowe, 661
    StringReader, 666
    StringWriter, 666
  zamykanie, 667
adnotacje, 206, 506
adres
  IPv4, 697
  IPv6, 697
  URI, 698
    prefiksy, 703
    typu pack, 743
agregacje, 475
akcesor init, 219
akcesory widoku, 692
ALC, assembly load context, 749
algorytm
  Brotli, 669
  DeflateStream, 669
  GZipStream, 671
  obliczania skrótów, 834
    MD5, 834
    SHA1, 834
    SHA, 835
  RSA, 842
aliasy typów, 102
analiza
  procesów, 569
  wątków, 570
anonimowe wywoływanie
  składowych, 824
aplikacje
  konsolowe, 764
  UWP, 683– 687, 728
architektura sieci, 694
archiwum ZIP, 672
argument NumberStyles, 299
argumenty nazwane, 79, 952
ASCII, 664
asercja
  granicy słowa, 965
  o zerowej wielkości, 963
ASP.NET Core, 263
asynchroniczne
  strumienie, 31
  wyrażenia lambda, 624
asynchroniczność, 260
atomowość, 850
atrapa procedury obsługi, 706
atrybut, 229, 261, 788
  [Conditional], 563
  [DebuggerHidden], 569
  [DebuggerStepThrough], 569
  [DllImport], 759
  [SkipLocalsInit], 247
  [ThreadStatic], 877
  [UnmanagedCallersOnly], 940
  AttributeUsage, 790
  DllImport, 934, 935
  Flags, 149
  MarshalAs, 935
  StructLayout, 937
atrybuty
  bitmapowe, 789
  informacji wywołującego, 231
  kompresji, 674
  nazwane, 230
  niestandardowe, 789, 790
  pobieranie, 792
  pozycyjne, 230
  przypisywanie, 230, 807
  pseudoniestandardowe, 790
  stosowanie, 230
  szyfrowania, 674
  warunkowe, 249
  własne, 791
  zestawu, 732
awaitery, 601, 605

B

bariera
  pamięci, 850
  wątku wykonawczego, 873
BCL, Base Class Library, 19,
  259, 367

Kup ksi k Pole ksi k

bezpieczeństwo
plików, 676
procesów, 948
systemu operacyjnego, 687
typów, 18
wątków, 651, 854–857, 893
zarządzanie kluczami, 840
biblioteka
.NET BCL, 19, 259, 367
Plugin.Common, 763
biblioteki
DLL, 934
niezarządzane, 759
blok
catch, 186
finally, 186, 189
instrukcji, 40
blokady, 849, 850, 854
aktualizacji, 865
asynchroniczne, 861
odczytu i zapisu, 861
rekurencyjne, 866
z możliwością uaktualnienia, 864
z podwójnym zatwierdzeniem, 875
zagnieżdżone, 851
blokowanie
bez wykluczania, 846, 859
obiektów bezpiecznych wątkowo, 855
wykluczające, 846
błędy, 186
parsowania, 303
zaokrąglania, 62
BMP, Basic Multilingual Plane, 280
bufor zdarzeń, 682
buforowanie, 557
bufory o stałym rozmiarze, 245

C

CCW, COM-Callable Wrapper, 955
certyfikat, 739
cyfrowy witryny, 841
ciąg tekstowy zapytania, 713
CLR, Common Language Runtime, 19, 20, 259
implementacja
indeksatorów, 120
własności, 118
COM, Component Object Model, 37, 262, 934, 949
indeksatory, 953
interfejsy, 950
manifest, 956
system typów, 950
cookie, 715
czas
bieżący, 285
letni, 292

D

data i godzina, 280, 286, 288, 302
deasembler, 813
debugger, 568
debugowanie, 565
deklaracja
using, 28, 190
XDeclaration, 497
deklaracje XML, 499
dekonstruktory, 34, 111
dekoratory, 401
delegat, 166, 429, 783
Action, 171
Comparison, 353
EventHandler, 177
Func, 171, 391
MatchEvaluator, 968
delegaty, 166, 429, 783
dla zdarzeń, 176
metody docelowe, 168
multiemisji, 169
typy delegacyjne, 170
zgodność
parametrów, 173
typów, 172
typów zwrotnych, 173
diagnostyka, 579
diagram UML, 13
DLR, dynamic language runtime, 818
DNS, Domain Name Service, 696, 721
dodawanie zależności, 766
dokumentacja XML, 250
DOM, Document Object Model, 37, 431, 480, 481
domknięcie, 182
dostawcy formatu, 294, 295
dostęp
do katalogów i plików, 685
do składowych niepublicznych, 784
do urządzeń wymiennych, 686
drzewa
wyrażeń, 37, 180, 412, 429– 432
kompilowanie, 430
struktura DOM, 431
wywołań, 610, 611
asynchronicznych, 622
X-DOM, 482, 484
dynamiczne
generowanie kodu, 793
metody, 793
wiązanie, 953
wybieranie przeciążonych składowych, 821, 825
wywoływanie, 781, 783
dynamiczny
odbiorca, 237
system wykonawczy języka, 818
dyrektywa
#define, 562
#elif, 562
#else, 562
#if, 562
#nullable enable, 206
#undef, 562
using, 40, 100, 534
static, 36, 100
zagnieżdżanie, 102
dyrektywy preprocesora, 248
dziedziczenie, 124, 132

Kup ksi k Pole ksi k

dzielenie
  całkowitoliczbowe, 58
  zakresowe, 906
dziennik zdarzeń
  monitorowanie, 573
  odczyt danych, 573
  zapis danych, 572

E

EF Core, 418
  dodawanie jednostek, 426
  klasy jednostek, 418
  konfiguracja
    modelu, 420
    połączenia, 419
  ładowanie
    leniwe, 427
    własności nawigacyjnych, 426
  metoda
    GroupBy, 466
    SelectMany, 451
  operator
    Contains, 440
    LIKE, 440
  podzapytania, 444
  śledzenie
    obiektów, 424
    zmian, 424
  tworzenie bazy danych, 421
  usuwanie jednostek, 426
  własności nawigacyjne, 425
  wykonywanie opóźnione, 428
  złączenia, 444
egzemplarze, 46
element główny, 541
elementy
  LINQ, 385
  tablic, 72
emitowanie
  generycznych klas, 808
  generycznych typów, 808
  konstruktorów, 806
  metod, 803
  metod generycznych, 808
  pól, 805
  składowych typów, 802
  typów, 800
  właściwości, 805
  zestawów, 800
enumeratory, 194, 930

F

fabryka zadań, 915
FIFO, first-in, first-out, 359
filtry wyjątków, 36, 189
finalizatory, 121, 541
  C#, 779
  wywołanie metody Dispose, 543
flaga
  AssemblyBuilderAccess.RunAndCollect, 793
  DeclaredOnly, 785
  Faulted, 913
  RanToCompletion, 913
  UseShellExecute, 334
flagi parsowania, 301
folder aplikacji, 685
format
  base64, 306
  CDFS, 675
  FAT, 675
  JSON, 526
  NTFS, 675
formatowanie
  daty i godziny, 302
  wyliczeń, 304
  złożone, 296
formularze, 714
  przekazanie danych, 714
framework, 21
FTP, File Transfer Protocol, 696, 719
funkcja, 18
  CreateFileMapping, 943
  getcwd, 936
  GetSystemTime, 937
  MessageBox, 934
  Registry-free, 956
  Type.GetTypeFromProgID, 953
funkcje
  asynchroniczne, 31, 262, 614–616, 620
  operatorowe, 240
  wyrażeniowe, 35

G

garbage collection, 533, 793
generator liczb losowych, 313
generowanie
  dynamicznego kodu, 793
  liczb, 313
  metod instancji, 804
  metod w locie, 793
  obiektów DbSet<>, 419
  zmiennych lokalnych, 796
generyczne typy delegacyjne, 170
globalizacja, 308
globalny bufor zestawów, 762

H

HTTP, Hypertext Transfer Protocol, 696, 713
  tworzenie serwera, 717

I

identyfikator GUID, 955
identyfikatory, 42
  definiujące typ, 253
IIS, Internet Information Services, 696
IL, Intermediate Language, 20, 730, 768
  obsługa wyjątków, 799
  parsowanie, 813
  rozgałęzianie, 797
  stos ewaluacji, 794
  tablica bajtów, 813
  tworzenie instancji obiektów, 798
  wywoływanie
    konstruktorów bazowych, 807
    metod instancji, 798

Kup ksi k   Pole ksi k

- implementacja
  - indeksatora, 119
  - obiektów dynamicznych, 827
  - własności, 118
- indeks, 27
- indeksatory, 119, 953
  - wirtualne, 128
- indekser C#, 779
- indeksy, 69, 120
- inferencja typów, 55
- informacje o woluminie, 681
- inicjalizacja elementów, 68
- inicjalizatory
  - indeksów, 35
  - kolekcji, 195
  - obiektów, 37, 113, 114, 410
  - własności, 35, 116, 222
- instrukcja
  - break, 97
  - continue, 97
  - fixed, 244
  - foreach, 195, 196
  - goto, 97
  - if, 90
  - lock, 846, 847
  - return, 98
  - switch, 33, 91
  - switch z typami, 93
  - throw, 98
  - try, 186
  - using, 190, 198
  - yield return, 198, 341, 401
- instrukcje
  - deklaracji, 88
  - iteracyjne, 95
  - najwyższego poziomu, 24, 41, 49
  - skoku, 97
  - wyboru, 90
  - wyrażeniowe, 89
- interfejs, 17, 142, 171, 772
  - API, 420
    - Microsoft Dataflow, 884
    - Reactive Extensions, 884
  - COM, 950
  - ICollection, 343, 344
  - ICollection<T>, 344
  - IComparable, 328, 330
  - IComparer, 381
  - ICustomFormatter, 297
  - IDictionary, 365
  - IDictionary<TKey,TValue>, 364
  - IDispatch, 951
  - IDisposable, 195, 339, 423, 533
  - IDynamicMetaObjectProvider, 827
  - IEnumerable, 337
  - IEnumerable<T>, 338, 339, 354, 776
  - IEnumerator, 143, 337
  - IEnumerator<T>, 338
  - IEqualityComparer, 378
  - IEquatable<T>, 322, 327
  - IFormatProvider, 297
  - IFormattable, 294
  - IGrouping<,>, 787
  - IList, 343, 345, 786
  - IList<T>, 345
  - IOrderedEnumerable, 463
  - IOrderedQueryable, 463
  - IProducerConsumerCollection<T>, 919
  - IProgress<T>, 636
  - IReadOnlyCollection<T>, 346
  - IReadOnlyList<T>, 346
  - IStructuralComparable, 383
  - IStructuralEquatable, 383
  - IUnknown, 951
  - IXmlSerializable, 522
  - System.Collections.IEnumerable, 196
  - System.IDisposable, 190
  - WPF, 264
- interfejsy, 142, 171, 772
  - domyślne składowe, 147
  - implementacja
    - jawna, 144
    - wirtualna, 144
  - niegeneryczne, 339
  - przeliczeniowe, 340
  - reimplementacja, 145
  - rozszerzanie, 143
- interoperacyjność macierzysta, 262
- interpolacja łańcuchów, 36, 67
- IP, Internet Protocol, 696
- iteratory, 31, 196, 340, 625
- izolowanie
  - zależności, 766
  - zestawów, 748

J

- jednolity system typów, 17
- język
  - C#, 39
  - IL, 20, 730, 768
  - IronPython, 830
  - LINQ, 385, 434, 480
  - Perl 5, 957
  - wyrażeń regularnych, 957
  - XML, 511
- języki dynamiczne, 830
- JIT, just-in-time, 20
- JSON, JavaScript Object Notation JSON, 260, 511, 526
  - modyfikacje danych, 531
  - odczyt
    - obiektów, 530
    - tablic, 530
    - wartości, 530

K

- katalog bazowy aplikacji, 653
- katalogi specjalne, 680
- klasa, 17, 105
  - Aes, 835
  - AggregateException, 916, 917
  - AppContext, 335
  - Array, 346
  - ArrayList, 355
  - Assembly, 733
    - elementy składowe, 734
  - AssemblyBuilder, 800
  - AssemblyDependencyResolver, 760
  - AssemblyLoadContext, 749, 761

Kup ksi k Pole ksi k

AssemblyName, 737
AsyncLocal<T>, 879
AuthenticationManager, 710
AutoResetEvent, 866
BackgroundWorker, 643
Barrier, 873
BinaryReader, 666
BinaryWriter, 666
BitArray, 63, 360
BitConverter, 308
BlockingCollection<T>, 921
BufferedStream, 660
Collection<T>, 369
CollectionBase, 371
ConcurrentBag<T>, 920
Console, 331
ConsoleTraceListener, 566
Convert, 305
CountdownEvent, 870
CredentialCache, 710
CryptoStream, 837–839
CustomAttributeBuilder, 807
DateTime, 283
 formatowanie, 292
 parsowanie, 292
DateTimeOffset, 284
DbContext, 418, 422, 423
Debug, 565, 568
Debugger, 568
Dictionary, 365
DictionaryBase, 373
Directory, 677, 678
Disposable, 861
Dns, 694
DynamicMethod, 793–798
DynamicObject, 827
Encoding, 278, 279, 664
Enumerable, 385, 460
Environment, 332
EqualityComparer, 379
EventLog, 574
EventWaitHandle, 866, 867
Exception, 186, 572
ExpandoObject, 829
File, 673, 678
 metody skrótów, 653
 metody statyczne, 663
FileOpenPicker, 687
FileSecurity, 676
FileStream, 652
 funkcje zaawansowane, 654
 nazwa pliku, 652
 tryb pliku, 653
 tworzenie egzemplarza, 652
FileSystemWatcher, 682
FolderPicker, 687
Foo, 229
GenericTypeParameterBuilder, 809
HashAlgorithm, 834
HashSet<T>, 361
Hashtable, 365
HttpClient, 694, 703, 711
HttpContent, 706
HttpListener, 694, 717, 719
HttpMessageHandler, 706
HybridDictionary, 367
ILGenerator, 794–807
IPAddress, 697
JsonDocument, 529, 531
JsonElement, 530
JsonReaderOptions, 527
JsonSerializer, 526
KeyedCollection<,>, 372
KnownFolders, 685, 686
Lazy<T>, 875
LazyInitializer, 876
LinkedList<T>, 357
List<>, 786
List<T>, 355
ListDictionary, 367
ManualResetEvent, 869
Math, 309
MemberInfo, 776, 778
MemoryStream, 536, 655, 837
MethodBuilder, 801
MethodInfo, 801
ModuleBuilder, 800
Monitor, 848
Mutex, 846, 853
NumberFormatInfo, 295
object, 137
OpCodes, 814
OperationCanceledException, 910
OrderedDictionary, 367
Package, 685
Parallel, 884, 886, 900
ParallelEnumerable, 611
ParallelLoopState, 904
Path, 679
 elementy składowe, 679
PipeStream, 656
Process, 333, 569
Progress<T>, 636
ProtectedData, 833
QueryOptions, 684
Queue<T>, 359
Random, 313
ReaderWriterLockSlim, 862, 863, 865
ReadOnlyCollection<T>, 374
ResourceManager, 744, 745
Rijndael, 835
RSA, 842
RSACryptoServiceProvider, 844
Semaphore, 860
SemaphoreSlim, 860
SmtpClient, 694, 722
Socket, 694
SortedDictionary<,>, 367
SortedList<,>, 368
SortedSet<T>, 361
SpinLock, 846
Stack<T>, 360
StackFrame, 570, 571
StackTrace, 570, 571
Stopwatch, 578
StorageFile, 684
StorageFolder, 683
Stream, 647
 elementy składowe, 647
StreamReader, 663
StreamSocket, 729
StreamWriter, 663
StringBuilder, 259, 276, 936
StringComparer, 382
StringInfo, 280

Kup książkę   Poleć książkę

klasa
  StringReader, 537, 661, 666
  StringWriter, 661, 666
  SurnameComparer, 382
  SynchronizationContext, 597
  System.Attribute, 229
  System.Delegate, 168
  System.Exception, 192
  System.Globalization.
  ↪CultureInfo, 748
  System.Tuple, 214
  System.Type, 769
  Task, 601, 602
  Task<TResult>, 602, 621, 911
  TaskCompletionSource, 606, 607
  TaskFactory, 915
  TcpClient, 694, 723, 725
  TcpListener, 694, 723, 725
  TextReader, 661
    elementy składowe, 662
  TextWriter, 661, 662
    elementy składowe, 663
  ThreadLocal<T>, 877, 892
  Timer, 881
  TimeZone, 289
  TimeZoneInfo, 289
  Trace, 565, 568
  TraceListener, 566, 567
  Type, 772
  TypeBuilder, 805, 806
  TypeInfo, 770
  UdpClient, 694
  Uri, 698
  WeakReference, 556
  WebClient, 537, 694, 700, 714, 719
  WebRequest, 694, 700, 702
  WebResponse, 694, 700, 702
  XAttribute, 495
  XContainer, 487, 490
  XDocument, 491, 497
  XElement, 495, 524, 525
  XmlConvert, 281, 293, 306
  XmlReader, 511–521, 524
  XmlReaderSettings, 512
  XmlWriter, 500, 519–525
  XNode, 487, 491
  XObject, 506
  XStreamingElement, 510
  ZipArchive, 672
  ZipFile, 672
klasy, 105
  abstrakcyjne, 130
  generyczne, 159
  konstrukcyjne, 376
  konstruktor egzemplarzy, 110
  metody, 107
  pochodne, 125
  pola, 105
  rekordy, 214
  słownikowe, 363
  stałe, 106
  statyczne, 121
  ukrywanie składowych, 130
klauzula
  case, 93
  catch, 186, 188
  else, 90
  EXISTS, 440
  from, 394
klienty, 584
  bogate, 263
  ubogie, 263
klucz
  prywatny, 841
  publiczny, 840
kodowanie
  tekstu, 277
  znaków, 664
kolejki, 359
  typu producent-konsument, 922
kolejność wykonywania działań, 84
kolekcje, 259, 336
  blokujące ograniczone, 921
  niezmienne, 374
    tworzenie, 375
    wydajność, 377
  współbieżne, 887, 918
kombinatory wzorców, 26, 226
komentarze, 39, 43
  dokumentacyjne, 250
komparatory, 381
  IEqualityComparer, 328
  równości strukturalnej, 348
kompilacja, 41
  na żądanie, 20
  warunkowa, 561
kompilator Roslyn, 263
komponowanie iteratorów, 199
kompresja
  strumienia, 669
  w pamięci, 670
  w Uniksie, 671
komunikacja międzyprocesowa, 656
komunikaty żądania, 705
konkatenacja łańcuchów, 66
konstruktor główny, 222
konstruktory, 46, 110, 132
  bezparametrowe, 132
  emitowanie, 806
  klasy bazowej, 807
  niejawne bez parametrów, 111
  niejawne wywoływanie, 132
  niepubliczne, 111
  przeciążanie, 110
  statyczne, 120, 121
konteksty
  adnotacji, 206
  ALC, 749, 757, 759
    bieżące, 756
    domyślne, 754
  asynchroniczności, 628
  ładowania zestawów, 749, 750
  ostrzeżeń, 206
  synchronizacji, 597, 628
kontenery .resources, 741
kontrawariancja, 163, 173
kontrola
  konta użytkownika, 688
  typów
    dynamiczna, 136
    statyczna, 18, 136
    ścisła, 19

Kup ksi k   Pole ksi k

kontynuacje, 611, 871, 910, 911, 912
warunkowe, 912, 913
z wieloma przodkami, 914
konwencje wywoływania, 939
konwersje, 160, 304, 309
dynamiczne, 236, 306
jawne, 241
liczbowe, 56, 305
łańcuchowe, 316
na format base64, 306
niejawne, 241
referencji, 125
typów, 48, 200, 298
wyliczeń, 149, 314
znaków, 65
konwertery typów, 293, 307
kopiowanie tablicy, 348
kotwice, 964
kowariancja, 161, 163, 173
krotki, 34, 95, 211
dekonstrukcja, 213
nadawanie nazw elementom, 212
porównywanie, 214
wymazywanie typów, 212
kryptografia, 262, 832
kultury, 747
kwantyfikator, 393, 477
leniwy, 962
zachłanny, 962

L

lambda, 180
LAN, Local Area Network, 696
leniwa
ewaluacja, 220
inicjalizacja, 874
leniwe
ładowanie, 427
wykonywanie, 398
liczniki wydajności, 574
odczyt danych, 576
sprawdzenie dostępności, 575
tworzenie, 577
zapis danych, 577
LIFO, last in, first out, 134, 360
LINQ, Language Integrated Query, 37, 260, 385
agregacje bez ziarna, 475
filtrowanie, 436, 438
filtrowanie z indeksowaniem, 440
grupowanie, 464
grupowanie według wielu kluczy, 467
hierarchie obiektów, 443
klasa JsonDocument, 531
komparatory, 463
kwantyfikatory, 438, 477
łączenie, 450, 436, 453
dekoratorów, 401
operatorów zapytań, 387
według wielu kluczy, 456
metody
agregacyjne, 437, 473
generujące, 438, 478
konwersji, 437, 468
opakowywanie zapytań, 409
operatory
elementów, 437, 471
zbiorów, 437, 467
płaskie złączenia zewnętrzne, 458
podzapytania, 403, 444
porównania łańcuchów, 440
porządkowanie, 437, 461
projekcja, 410, 436, 442
do typów konkretnych, 446
do X-DOM, 507
z indeksowaniem, 443
przechwytywanie zmiennych, 399
sekwencje, 436, 437
składnia
mieszana, 397
płynna, 387, 396
zapytaniowa, 396
sortowanie, 463
wykonywanie opóźnione, 397, 400
wyrażenia lambda, 390, 393
zapytania
interpretowane, 412
złożone, 406
złączenia
w składni płynnej, 456
z widokami wyszukiwania, 459
zewnętrzne, 452
zmienne zakresowe, 395
LINQ to XML, 480
lista kontroli dostępu, 689
listy, 354
parametrów, 217
literały, 43
liczbowe, 55
lokalizacja, 308

Ł

ładowanie zestawu, 748, 750
łańcuch null, 270
łańcuchy, 269
dzielenie, 272
formatu, 294
niestandardowe, 300
numeryczne, 298
standardowe, 298
złożone, 272, 286
formatowania
daty i godziny, 302
wyliczeń, 304
interpolacja, 67
konkatenacja, 66, 272
dekoratorów, 401
operatorów zapytań, 387
modyfikowanie, 271
pobieranie znaków, 270
połączenia, 419
porównywanie, 67, 273, 440
procedur obsługi, 707
przeszukiwanie, 270
strumieni szyfrowania, 838, 839
tworzenie, 269
zapisywanie deklaracji, 500

Kup ksi k
Pole ksi k

łączenie
dekoratorów w łańcuchy, 401
operatorów zapytań, 387
procedur obsługi, 707
łączność operatorów
lewostronna, 84
prawostronna, 84

M

magazyn
danych, 645
prywatny, 685
manifest
aplikacji, 732
zestawu, 731
mapowanie
plików, 690
struktur, 945, 948
mechanizm
szeregowania, 935
usuwania nieużytków, 533, 546
dostrajanie, 552
kolekcja pokoleniowa, 547
powiadomienia, 550
pule tablic, 552
sterta ogromnych obiektów, 548
tryb stacji roboczej i serwera, 549
usuwanie w tle, 550
wymuszenie działania, 551
metadane, 20
składowych, 778
typu, 769
metaznaki, 960
metoda, 18, 39,107
Activator.CreateInstance, 773
Add, 286, 345, 376
AddAfterSelf, 493
AddAnnotation, 506
AddBeforeSelf, 493
Aggregate, 475, 898
All, 478
Ancestors, 491
Any, 477
Array.ConvertAll, 354
Array.CreateInstance, 349, 350
Array.Sort, 353
AsEnumerable, 418, 471
AsParallel, 888, 889
AsQueryable, 471
AsReadOnly, 354
Assembly.Load, 756, 761
AsSequential, 896
Assert, 565
AsTask, 637, 683
Attribute.GetCustomAttribute, 792
Average, 474
await, 626
BeginInvoke, 596
BinaryReader, 725
BinarySearch, 351
BinaryWriter, 725
cancelSource, 910
Cast, 469
Clone, 354
Close, 535, 568, 868
Combine, 680
Compare, 274
CompareOrdinal, 274
CompareTo, 274, 275, 440
Compile, 430
ComputeHash, 834
Concat, 467
ConfigureAwait, 632
Connect, 723
ConnectAsync, 723
ConstrainedCopy, 354
Contains, 270, 361, 440, 477
ContinueWith, 605, 910, 914
Convert.ToDecimal, 315
ConvertTime, 290
ConvertTimeFromUtc, 290
ConvertTimeToUtc, 290
CopyTo, 927
CopyToAsync, 705
Count, 398, 473
Create, 703
CreateDelegate, 774, 796
CreateFileQueryWithOptions, 684
CreateFromDirectory, 672
CryptoStreamMode.Read, 837
CryptoStreamMode.Write, 837
DefaultIfEmpty, 459, 472
DefineConstructor, 806
DefineField, 805
DefineGenericParameters, 809
DefineType, 801
Delay, 608, 611
Delete, 674
DescendantNodes, 490
Descendants, 490, 494
Disassemble, 814
DisplayPrimeCounts, 611
Dispose, 190, 534–537, 543
Distinct, 442
Document.Root, 491
DynamicVisit, 822, 823
EF.Functions.Like, 440
Element, 489
ElementAt, 472
ElementAtOrDefault, 471
Elements, 488
EmitWriteLine, 794
Empty, 478
Encoding.GetEncoding, 278
EndsWith, 270
Enqueue, 923
EnsureInitialized, 876
EnterAsync, 861
EnterContextualReflection, 758
Enum.Format, 316
Enum.GetNames, 316
Enum.GetValues, 316
Enum.Parse, 316
Enum.ToObject, 315
Enumerable.Join, 460
EnumerateArray, 530
EnumerateObject, 530
EqualityComparer<T>.Default, 380
Equals, 214, 273, 320, 323, 329
Escape, 960
Except, 468

Kup ksi k Pole ksi k

ExceptWith, 362
Exists, 351, 352
ExportedTypes, 788
ExtractToDirectory, 672
Fail, 565, 566
File.Encrypt, 833
FileShare.ReadWrite, 655
Finalize, 121
Find, 351
FindAll, 351, 352
FindIndex, 351
FindLast, 351
FindLastIndex, 351
First, 471
FirstNode, 488, 491
FirstOrDefault, 471
Flatten, 917
Flush, 568, 655, 839
Foo, 75
ForAll, 895
Format, 272
FormatOperand, 815
GC.ReRegisterForFinalize, 545
get, 115, 117
GetAmbiguousTimeOffsets, 290
GetAnswerToLife, 621
GetArrayLength, 530
GetAsync, 705
GetAwaiter, 605
GetBuffer, 655
GetByteArrayAsync, 705
GetCustomAttributes, 792
GetData, 878
GetDrives, 681
GetElementType, 770
GetEncodings, 278
GetEnumerator, 338, 340, 341
GetFullPath, 680
GetGenericTypeDefinition, 775
GetHashCode, 325
GetHostAddresses, 721
GetHostEntry, 721
GetILAsByteArray, 813
GetInt32, 530
GetInterfaces, 772
GetKeyForItem, 372
GetLength, 351
GetLongLength, 351
GetLowerBound, 351
GetMembers, 775, 776, 777
GetMethod, 779
GetNestedTypes, 770
GetPrimesCount, 611
GetPrimesCountAsync, 612, 616
GetResult, 605
GetStreamAsync, 705
GetString, 530
GetStringAsync, 705
GetType, 136
GetTypes, 788
GetUpperBound, 351
GetUtcOffset, 289
GetValue, 349, 781
GetWebPageAsync, 630
GroupBy, 464
GroupJoin, 454, 457
Handle, 917
IEnumerator<int>.Current, 343
Include, 426
IndexOf, 345, 351
Intersect, 468
Invoke, 596, 773
IsAmbiguousTime, 290
IsAssignableFrom, 773
IsControl, 269
IsDaylightSavingTime, 289, 292
IsDigit, 269
IsInstanceOfType, 772
IsInvalidTime, 290
IsLetter, 269
IsLetterOrDigit, 269
IsLower, 269
IsMatch, 958
IsNumber, 269
IsPunctuation, 269
IsSeparator, 269
IsSubclassOf, 773
IsSymbol, 269
IsUpper, 269
IsWhiteSpace, 269
Join, 454, 586
Last, 471
LastIndex, 351
LastIndexOf, 271, 352
LastIndexOfAny, 271
LastNode, 488, 491
LastOrDefault, 471
Load(byte[]), 762
Load, 483
LoadAsync, 728
LoadFile, 762
LoadFrom, 762
LoadFromAssemblyName, 751
LoadUnmanagedDll, 759
LoadUnmanagedDllFromPath, 759
Lock, 590, 655
LongCount, 473
LookupNamespace, 519
Main, 47, 104
MakeByRefType, 803
MakeGenericType, 774
Matches, 958
Max, 473
Min, 473
Monitor.Enter, 847
Monitor.Exit, 847
Move, 674
MoveNext, 337
MoveToAttribute, 517, 518
MoveToElement, 517
NextBytes, 313
NextDouble, 313
NextNode, 491
Nodes, 488
Object.Equals, 320–322
OfType, 469
OnCompleted, 605
OnConfiguring, 419
OnModelCreating, 420
OperationCompleted, 629
OperationStarted, 629
PadLeft, 271
PadRight, 271
Parallel.For, 901, 905

Kup ksi k Pole ksi k

metoda
- Parallel.ForEach, 899–902, 905
- Parallel.Invoke, 900
- ParallelEnumerable.Range, 897
- Parse, 281, 287, 299, 483, 529
- ParseExact, 287
- Peek, 662
- PreviousNode, 491
- PrintAnswerToLife, 620, 621, 622
- QueryInterface, 951
- Range, 479
- RangeAsync, 627
- Read, 649, 693
- ReadBlock, 662
- ReadByte, 649
- ReadElementContentAsString, 514
- ReadEndElement, 514
- ReadInnerXml, 516
- ReadLine, 662
- ReadLineAsync, 839
- ReadStartElement, 514
- ReadSubtree, 516
- ReadXml, 521, 523
- Regex.Match, 958
- Regex.Replace, 967
- Regex.Split, 968
- RegisterWaitForSingleObject, 872
- Release, 860
- Remove, 376, 493
- RemoveAll, 493
- RemoveAttributes, 493
- RemoveNodes, 493
- RemoveXXX, 492
- Repeat, 479
- Replace, 271, 407, 968
- ReplaceWith, 493
- ReRegisterForFinalize, 546
- Resize, 354
- ResumeEvents, 538
- Reverse, 353
- RunAsync, 596
- Save, 484
- Select, 442
- SelectMany, 447, 450–452
- Send, 722
- SendAsync, 705, 707
- SequenceEqual, 478
- set, 115, 117
- SetAttributes, 675
- SetAttributeValue, 493
- SetCustomAttribute, 807
- SetData, 878
- SetElementValue, 492, 493
- SetGenericParameterAttributes, 809
- SetValue, 349, 495, 781
- SignalAndWait, 872, 873
- SignHash, 844
- Single, 471
- SingleOrDefault, 471
- Skip, 441
- SkipWhile, 441
- Sort, 352
- Split, 272
- StartsWith, 270
- Stop, 535, 579
- StorageFile.GetFileFromPathAsync, 684
- StreamReader.ReadToEnd, 725
- string.Equals, 274
- string.Format, 273, 296
- string.IsNullOrEmpty, 270
- StringComparer.CurrentCulture, 382
- StringComparer.Ordinal, 382
- Substring, 271
- Sum, 474
- SuspendEvents, 538
- SymmetricExceptWith, 362
- Take, 441
- TakeWhile, 441
- Task.Delay, 608
- Task.Factory.StartNew, 907
- Task.Run, 601
- Task.WaitAll, 909
- Task.WhenAll, 639
- Task.WhenAny, 638
- Thread.FreeNamedDataSlot, 879
- ThrowException, 800
- TimeZoneInfo, 289
- ToArray, 470
- ToBoolean, 307
- ToDictionary, 470
- ToHashSet, 470
- ToList, 470
- ToLocalTime, 287, 288
- ToLongDateString, 287
- ToLookup, 470
- ToLower, 272
- ToShortDateString, 287
- ToString, 137, 272, 293, 484, 928
- ToUniversalTime, 287, 288, 292
- ToUpper, 272
- Trace, 566
- TraceError, 565, 567
- TraceInformation, 565
- TraceWarning, 565, 567
- TrimEnd, 271
- TrimStart, 271
- TrueForAll, 351, 352
- TryAdd, 919
- TryCopyTo, 927
- TryEnter, 848
- TryGetSwitch, 335
- TryParse, 281, 287, 293
- TryParseExact, 287
- TryTake, 919, 921
- Unescape, 960
- UploadValues, 701, 714
- UseSqlServer, 419
- ValueTuple.Create, 213
- Wait, 602
- WaitAll, 872
- WaitAny, 872
- Where, 439
- WithDegreeOfParallelism, 894
- Write, 331, 565, 650, 693
- WriteAllText, 279
- WriteByte, 650
- WriteEvent, 581
- WriteLine, 39, 331, 662
- WriteLineAsync, 839

Kup ksi k Pole ksi k

WriteTo, 484, 531
WriteValue, 520
WriteXml, 521
XNode.ReadFrom, 524
metody, 18, 39,107
agregacyjne, 473
anonimowe, 185
asynchroniczne w WinRT, 627
częściowe, 38, 122
częściowe rozszerzone, 123
docelowe delegatu, 168
dostępowe, 115, 117
dostępowe zdarzenia, 175, 179
dynamiczne, 783, 793
parsowanie argumentów, 795
egzemplarzowe, 209
emitowanie, 803
generowane w locie, 793
generujące, 478
generyczne, 155, 156, 785
emitowanie, 808
pobieranie, 782
wywoływanie, 782
klasy
DynamicObject, 827
Math, 310
ReaderWriterLockSlim, 862
konwersji, 468
lokalne, 108, 185
lokalne statyczne, 109
ładujące, 761
parametry, 781
płynnego API, 420
przeciążanie, 109, 133, 822, 825
przysłanianie, 804
rozszerzające, 207, 209
wywoływanie łańcuchowe, 208
rozszerzeń, 37
sygnatura, 108
ukrywanie, 804
Union, 467
wirtualne, 128, 370, 419
wtyczek, 167
wyrażeniowe, 108
z rodziny
ReadXXX, 516
TryXXX, 194
zwracające składowe, 778
miejsca wywołania, 819
migracja EF Core, 422
moduł, 800
modyfikator
async, 614
in, 78
out, 76, 162
override, 128
params, 78
readonly, 28, 106
ref, 76
static, 29, 121, 183
unmanaged, 941
modyfikatory
dostępu, 140
zdarzeń, 180
multiemisja, 169
mutacja niedestrukcyjna, 218
muteks, 860

N

nadawca, 174
nagłówki, 711, 713
narzędzia diagnostyczne, 579
narzędzie
dotnet, 41
dotnet-counters, 579
dotnet-dump, 582
dotnet-trace, 580
gunzip, 671
gzip, 671
ildasm, 741
ILSpy, 794
signtool.exe, 740
nazwy
kwalifikowane zestawu, 769
typów
generycznych, 771
osadzonych, 771
parametrów ref i out, 772
tablic, 772
wskaźników, 772
niejawna konwersja referencji, 161
niezmienność, 858

O

obiektowość, 17
obiekty, 73
niezmienne, 858
synchronizacji, 848
typu IAsyncEnumerable<T>, 626
obliczanie skrótów, 833–835
obsługa
komunikatów, 707
wyjątków,186, 593, 711, 799
blok finally, 189
instrukcja try, 186
klauzula catch, 188
oczekiwanie
na interfejs użytkownika, 616
na zadanie, 615
odrzucenia, 32, 77
odwołania lokalne ref, 80
ograniczanie dostępności, 142
ograniczenia typów generycznych, 157
operacje
asynchroniczne, 609
limit czasu, 651
na plikach i katalogach, 673
opróżnienie, 650
synchroniczne, 609
wejścia-wyjścia, 261
losowe plikowe, 690
w UWP, 683
zamknięcie, 650
operator, 43
?., 87
??, 87
??=, 87

Kup książkę Poleć książkę

operator
- ==, 323, 329
- adresowania, 243
- Aggregate, 898
- as, 126
- AsEnumerable, 417
- AsQueryable, 430
- checked, 58
- Contains, 440
- dekrementacji, 58
- dereferencji, 243
- FirstOrDefault, 489
- ignorowania null, 206
- inkrementacji, 58
- is, 127, 224, 772
- Join, 455
- nameof, 36, 124
- new, 82
- null, 87
- porównywania, 63
- równości, 63, 201
- rzutowania, 160
- trójargumentowy, 64
- typeof, 136, 156
- warunkowy null, 35
- wskaźnika do składowej, 243, 244
- Zip, 461

operatory
- < i >, 330
- == i !=, 319
- agregacji, 393
- arytmetyczne, 57
- bitowe, 59
- C#, 85, 779
- elementów, 471
- LINQ, 434–79
- łączność
  - lewostronna, 84
  - prawostronna, 84
- mieszanie, 202
- pożyczanie, 201
- przeciążanie, 239
- relacyjne, 202
- warunkowe, 64
- wyliczeń, 150
- zapytań, 385
  - Concat, 393
  - ElementAt, 392
  - First, 392
  - Last, 392
  - łączenie, 387
  - OrderBy, 388, 395, 461
  - OrderByDescending, 461
  - Reverse, 392
  - Select, 388, 391, 395
  - Skip, 392
  - SQL IN, 440
  - SQL LIKE, 440
  - Take, 392
  - ThenBy, 461
  - ThenByDescending, 461
  - Where, 388, 391, 395
  - wykonywanie opóźnione, 397, 400
  - wyrażenia lambda, 390
  - wywoływanie, 389
- zbiorów, 467
- zmieniające kształt, 436

optymalizacja
- PLINQ, 895
- z wartościami lokalnymi, 905

osadzanie typów współpracujących, 954

ostrzeżenia pragma, 250

ostrzeżenie, 206

P

pakiet
- NuGet, 830
- NuGet System.Linq.Async, 626

pakowanie, 134, 146

pamięć
- alokowana na stosie, 932
- diagnozowanie wycieku, 556
- lokalna wątku, 877
- mapowanie plików, 690
- migawka stanu, 582
- monitorowanie, 540
- niezarządzana, 932, 945
- współdzielona, 691, 692, 943, 947
- wyciek pamięci, 553–556

paralelizm strukturalny, 885

parametr, 40, 72, 75
- lockTaken, 848
- out, 782
- ref, 782
- T?, 200

parametry, 40, 72, 75
- metod, 781
- nazwane, 230
- niejawne ref, 952
- opcjonalne, 36, 79, 114, 952
- pozycyjne, 230
- typów, 156

parsowanie, 292, 296
- argumentów, 795
- błędy, 303
- IL, 813
- liczb, 306

pętla
- do-while, 95
- for, 95
- foreach, 96, 196
- while, 95

pętle
- wewnętrzne, 902
- zewnętrzne, 902

PFX, Parallel Framework, 884, 885
- komponenty bibliotek, 886
- używanie biblioteki, 887

pieczętowanie
- funkcji, 131
- klas, 131

planowanie zadań, 914

plasterkowanie, 925, 926

platforma, 19
- .NET 5, 255

plik
- ClientApp.dll, 760
- Microsoft.Data.SqlClient.dll, 760
- Terrain.dll, 749
- UIEngine.dll., 749

Kup ksi k

Pole ksi k

pliki
.resources, 741, 743
.resx, 744
pliki i katalogi
bezpieczeństwo, 676
kompresja, 674
metody dostępu, 683, 685
szyfrowanie, 674
PLINQ, Parallel LINQ, 884, 886, 888
anulowanie zapytania, 894
bezpieczeństwo wątkowe, 893
dzielenie
na części, 896
przy użyciu skrótów, 896
zakresowe, 896
kolejność elementów, 890
ograniczenia, 891
optymalizacja
na wejściu, 896
na wyjściu, 895
własnych agregacji, 898
stopień zrównoleglenia, 894
używanie, 893
płynny API, 420
pobieranie
atrybutów, 792
metadanych składowych, 778
typów osadzonych, 770
typów tablicowych, 770
podpisy cyfrowe, 843
podpisywanie kodu, 738, 739
podzapytania, 403
pola, 72, 105
deklarowanie, 106
emitowanie, 805
inicjalizacja, 106
kolejność inicjalizacji, 111, 121, 133
modyfikatory, 106
pole
HideBySig, 804
OpCode, 814
polimorfizm, 125
wielokierunkowy, 824
pomiar czasu, 578
ponowne obliczanie, 398
POP, Post Office Protocol, 696
POP3, 726
porównywanie, 318
strukturalne, 383
potoki
anonimowe, 656, 658
nazwane, 656
procesy, 569
programowanie
asynchroniczne, 609, 611
dynamiczne, 261, 818
funkcyjne, 18
równoległe, 262, 884
projekcje, 410
protokół
FTP, 696, 720
HTTP, 696
POP3, 726
porządkowania, 378
refleksji składowych, 776
równości, 319, 378
SMTP, 722
TCP, 723
UDP, 723
uwierzytelniania, 711
przechwytywanie zmiennych, 399
przeciążanie
konstruktorów, 110
metod, 109, 133
operatorów, 239
== i !=, 326
porównywania, 241
równości, 241
true i false, 242
przekazywanie
argumentów
przez referencję, 76, 77
przez wartość, 75
wyjątku, 628
zmiennych do Pythona, 831
przekierowywanie strumieni, 333
przełączanie kontekstu, 587
przepełnienie całkowitoliczbowe, 58
przesłanianie metody
Equals, 326
GetHashCode, 325
przestrzeń nazw, 47, 99, 208
globalna, 99
http://oreilly.com, 520
LocalName, 519
NamespaceURI, 519
OReilly.Nutshell.CSharp, 501
Package.Current.
↪InstalledLocation, 742
Reflection.Emit, 261
System, 39, 45
System.Collections, 63, 143, 259, 336
System.Collections.
↪Concurrent, 259, 336, 918
System.Collections.Generic, 259, 336
System.Collections.
↪Immutable, 375
System.Collections.
↪ObjectModel, 259, 336, 369
System.Collections.
↪Specialized, 259, 336
System.ComponentModel, 231, 307, 597, 643
System.Data, 536
System.Diagnostics, 332
System.Dynamic, 261, 818
System.Globalization, 267, 276, 730
System.IO, 190, 261, 536, 673
System.IO.Compression, 669, 672
System.IO.Memory, 690
System.Linq, 260, 352, 354, 385
System.Linq.Expressions, 260
System.Net, 261, 697
System.Net.Mail, 722
System.Numerics, 310
System.Object, 320
System.Reflection, 261, 368, 730, 733

Kup ksi k   Pole ksi k

przestrzeń nazw
System.Reflection.Emit, 768, 793, 801
System.Resources, 730
System.Runtime.↪CompilerServices, 247, 818
System.Runtime.↪InteropServices, 262, 934
System.Security.↪Cryptography, 99, 311, 313
System.Security.↪Cryptography.X509Certificates, 844
System.Text, 259, 267, 664
System.Text.Regular↪Expressions, 259, 957
System.Threading, 260, 262, 309
System.Threading.Tasks, 260, 601, 906
System.Timers, 881
System.Windows, 264
System.Windows.Forms, 264
System.Xml, 306
System.Xml.Linq, 260, 494
System.Xml.Serialization, 230
Utils, 209
Windows.ApplicationModel, 685
Windows.Storage, 261, 652
Windows.UI, 266
Windows.UI.Xaml, 266
przestrzenie nazw, 47, 99, 208
aliasy, 102
kwalifikatory aliasów, 103
powtarzanie, 101
ukrywanie nazw, 101
w X-DOM, 503
w XML, 501
właściwości, 103
zakres, 100
przetwarzanie tekstu, 259
przewidywanie
negatywne, 964
pozytywne, 963
wsteczne, 963
negatywne, 964
pozytywne, 964
przypisanie, 74, 83
przyrostki literałów liczbowych, 56
przysłanianie metod, 804
punkt wejściowy, 41

R

refaktoryzacja, 40
referencja, 125
dynamiczna, 235
this, 113
refleksja, 261, 769, 772, 775
dla zestawów, 787
rekordy, 25, 214
definiowanie, 215
konstruktory główne, 222
leniwa ewaluacja, 220
listy parametrów, 217
mutacja niedestrukcyjna, 218
pola obliczane, 220
porównywanie, 223
walidacja własności, 219
REST, REpresentational State Transfer, 696
rozgałęzianie, 797
rozpakowywanie, 134
równoległe wykonywanie zadań, 906
równość, 323, 324
referencyjna, 318
wartościowa, 318
równoważenie obciążenia, 897
równoważność typów, 955
rzutowanie, 48
w dół, 126
w górę, 126

S

SDI, Single Document Interface, 597
sekwencje, 385, 436
dekoratora, 400
specjalne, 65
semafory, 859
separatory cyfr, 32
serializacja, 263, 612
serwer
aplikacji, 857
bezpieczeństwo wątkowe, 857
FTP, 719
HTTP, 717
POP3, 726
proxy, 708
settery tylko do inicjalizacji, 25
sieć, 261
silne nazwy, 734
składnia
mieszana, 397
płynna, 396
SQL, 396
zapytaniowa, 393–396
składowe
abstrakcyjne, 130
C#, 779
CLR, 779
egzemplarza, 46
interfejsu, 142
domyślne, 29
generycznego, 785
klasy object, 137
niepubliczne, 784
statyczne, 46, 856
typu, 801
emitowanie, 802
generycznego, 780
ukrywanie, 130
skróty, 834
haseł, 835
słabe odwołania, 556–558
słowniki, 363
sortowane, 367

Kup ksi k Pole ksi k

słowo kluczowe, 42
  Action, 171
  add, 175
  and, 226
  async, 613
  await, 613, 614, 620, 622, 683
  base, 131, 132
  class, 105
  default, 74, 138, 157
  delegate, 186
  dynamic, 232, 768, 786, 818, 830
  event, 174
  extern, 103
  fixed, 245, 949
  from, 395
  Func, 171
  init, 117
  into, 406, 408, 411
  let, 411
  namespace, 99
  new, 26, 131, 210, 798
  override, 131
  public, 47
  remove, 175
  sealed, 131
  stackalloc, 245
  static, 183, 186
  switch, 29
  unmanaged, 940
  unsafe, 105, 243
  var, 37, 81, 210, 411
  virtual, 128
  void, 40
  when, 93
słowa kluczowe
  kontekstowe, 43
  zarezerwowane, 42
SMTP, Simple Mail Transfer Protocol, 696, 722
sortowanie, 352
SSL, Secure Sockets Layer, 703
stałe, 106
sterta, 73
stos, 72, 360
  ewaluacji, 794
strefy czasowe, 288
struktura, 138
  BigInteger, 310
  Complex, 312
  DateTime, 282, 288, 292
  DateTimeOffset, 282, 288
  DOM, 431
  Guid, 317
  Half, 311
  Memory<T>, 925, 929
  Nullable<T>, 200
  SortedDictionary<,>, 368
  Span<T>, 925
  TimeSpan, 280
  Utf8JsonReader, 526
  Utf8JsonWriter, 528
  ValueTask<T>, 631, 632
struktury, 138
  mapowanie, 945, 948
  referencyjne, 139, 929
  tylko do odczytu, 139
strumienie, 261, 645
  adaptery
    binarne, 661, 666
    tekstowe, 661
    XML, 661
    zamykanie, 667
  architektura, 646
  asynchroniczne, 31, 625
  bezpieczeństwo wątków, 651
  błędów, 333
  dekoracyjne, 646, 660
  kompresja, 669
  magazynu danych, 646, 651
  operacja
    odczytu, 649
    wyszukiwania, 650
    zapisu, 649
  wyjścia, 333
strumień
  BrotliStream, 669
  BufferedStream, 660
  FileStream, 652
  MemoryStream, 655
  PipeStream, 656
subkultury, 747
subskrybenci, 174
surogaty, 280
sygnalizacja, 846, 866, 867
  dwustronna, 868, 869
synchronizacja, 846, 848
system operacyjny
  bezpieczeństwo, 687
system wtyczek, 763
szablony w C++, 164
szeregowanie
  In i Out, 938
  klas, 937
  struktur, 937
  typów wspólnych, 935
szyfrowanie, 833
  kluczem publicznym, 833, 840
  symetryczne, 833, 835
  w pamięci, 837

Ś

ścieżki sondowania, 755
śledzenie
  obiektów, 424
  zmian, 424
środowiska wykonawcze, 21, 24

T

tablice, 67, 346
  bajtów, 279
  długość, 351
  dynamiczne, 349
  elementy, 67
  indeksy, 69, 349
  inicjalizacja elementów, 68
  konwertowanie, 354
  kopiowanie, 348, 353
  liczba wymiarów, 351
  nieregularne, 70
  odwracanie kolejności elementów, 353
  prostokątne, 70
  przeszukiwanie, 350, 351
  skrótów, 325

Kup ksi k Pole ksi k

tablice
    sortowanie, 352
    sprawdzanie granic, 72
    tworzenie, 349
    wyrażenie inicjalizacji, 68, 71
    zakresy, 69
    zmienianie rozmiarów, 354
TCP, Transmission and Control Protocol, 696, 723
    w UWP, 728
    współbieżność, 725
TDD, test-driven development, 556
technologia Authenticode, 738
testowanie, 309
tęczowe tabele, 835
tokeny
    JSON, 526
    metadanych, 788
TPL, Task Parallel Library, 884
tryb pliku, 654
tworzenie
    deasemblera, 813
    egzemplarzy, 46
    fabryk zadań, 915
    funkcji asynchronicznych, 620
    instancji typów, 773
    łańcuchów, 269
    łańcuchów strumieni szyfrowania, 838
    serwera HTTP, 717
    struktur, 138
    wątków, 584
    wtyczek, 763, 767
    wyrażeń lambda, 390
    zadań, 606, 907
    zestawu satelickiego, 746
typ, 17, 44
    char, 65, 267
    CultureInfo, 295
    DateTimeFormatInfo, 295
    DateTimeOffset, 298
    decimal, 62
    double, 61, 62
    dynamic, 235, 236, 237, 238
    enum, 314
    float, 61
    logiczny, 63
    Memory<T>, 262
    Nullable, 287
    NumberFormatInfo, 295
    object, 134, 235
    ref, 81
    Span<T>, 262
    string, 66, 268
    TKey, 391
    TResult, 391
    TSource, 391
    Type, 769
    ValueTuple, 213
    var, 236
    wyliczeniowy
        RegexOptions, 959
        UnmanagedType, 935
typizowanie
    wyrażenia new, 82
    elementów, 391
typy
    anonimowe, 210, 410
    bazowe, 772
    całkowitoliczbowe, 59
        8- i 16-bitowe, 59
        o rozmiarze natywnym, 60
    częściowe, 122
    delegacyjne, 166, 170
        generyczne, 170, 174
    dopuszczające wartość null, 199
    dynamiczne, 232
    emitowanie, 800
    generyczne, 153–156, 774, 824
        definiowanie, 809
        niepowiązane, 156
        niestworzone zamknięte, 810
        odwołania do samego siebie, 159
        ograniczenia, 157
        parametry typów, 156
        składowe, 780
        szablony C++, 164
        tworzenie podklas, 159
    kowariantne, 162
    liczbowe, 54
        unifikacja, 820
    nazwy, 771
    osadzone, 770
    parametrów lambdy, 181
    predefiniowane, 44
    referencyjne, 30, 50–53, 68, 318
        wartość null, 205
    składowych, 777
    statyczne, 238
    systemowe, 259
    tablicowe, 162, 770
    tworzenie, 105, 773
    wartości domyślne, 74
    wartościowe, 50, 53, 68, 318
        wartość null, 199, 203, 205
    węzłów, 520
    własne, 45
    wskaźnikowe, 243
    wyjątków, 193
    X-DOM, 482
    zagnieżdżone, 152
    zależności cykliczne, 811
    zwrotne, 40
        kowariantne, 129
        ref, 81

U

uchwyty zdarzeń oczekiwania, 866, 871
UDP, Universal Datagram Protocol, 696, 723
ukończenie synchroniczne, 629
ukrywanie metod, 804
UNC, Universal Naming Convention, 696
unia, 942
Unicode, 277
unifikacja, 134
uprawnienia administratora, 689
URI, Uniform Resource Identifier, 696, 698
URL, Uniform Resource Locator, 696

Kup ksi k   Pole ksi k

usługi P/Invoke, 934
usuwanie
anonimowe, 538
automatyczne nieużytków, 539
kontekstów ALC, 761
nieużytków, 533, 546
obiektów szyfrowania, 839
uchwytów oczekiwania, 868
UTF-16, 280
UTF-8, 665
utrata zgodności binarnej, 118
uwierzytelnienie, 709
UWP, Universal Windows Platform, 23, 265
folder
aplikacji, 685
pobranych plików, 686
wybierany przez użytkownika, 687
operacje wejścia-wyjścia, 683
TCP, 728

V

Visual Studio
identyfikator GUID, 955
kompilacja wtyczek, 763
narzędzia, 747
osadzanie zasobów, 742
pliki .resx, 744
podzespół współpracujący COM, 951
zasoby pack URI, 745

W

walidacja własności, 219
wariancja, 161
parametrów, 174
typów, 37
warstwy aplikacji, 263
wartości specjalne typów liczbowych, 61
wartość
+∞, 61
−0, 61
−∞, 61
generyczna, 157
NaN, 61, 219
null, 28, 52, 87, 199–205, 287, 509
skrótu, 325
zwrotna ref, 81
wątki, 262, 584
aktywne, 594
bariera, 873
bezpieczeństwo, 590, 651, 854–857
blokady, 590
blokowanie, 586, 587
bez wykluczania, 846
wykluczające, 846
dołączanie, 586
działające w tle, 594
interfejsu użytkownika, 597
pamięć lokalna, 877
priorytet, 595
procedura obsługi, 593, 594
procesów, 569
przechwytywane zmienne, 592
przekazywanie danych, 591
pula, 598
robocze, 596
spinning, 587
stan lokalny, 588
sygnalizacja, 595, 846, 866
dwustronna, 868, 869
tworzenie, 584
usypianie, 586
współdzielenie danych, 589
wyrażenia lambda, 592
wywłaszczenie, 585
zagnieżdżanie blokad, 851
zakleszczenia, 851
wersje środowiska, 258
węzły
atrybutów, 517
XML, 512
wiązanie
dynamiczne, 36, 232, 824, 953
językowe, 234
niestandardowe, 233
statyczne, 232
wielowątkowość, 845
Windows
Data Protection, 832
Desktop, 264
Forms, 264
WinRT, 627, 729
metody asynchroniczne, 627
WinUI 3, 266
wirtualizacja, 689
wirtualne składowe funkcyjne, 128
własności, 18, 115
automatyczne, 38, 116
klasy System.Exception, 192
tylko do inicjalizacji, 117, 780
tylko do odczytu, 115
wirtualne, 128
wyrażeniowe, 116
własność, 18, 115
AssemblyLoadContext.Default, 754
Listeners, 566
LowestBreakIteration, 904
Root, 499
ShouldExitCurrentIteration, 905
Value, 495
właściwości, 779
emitowanie, 805
właściwość
BaseAddress, 701
BaseType, 772
ByteOrder, 729
ContentLength, 720
CredentialCache.DefaultNet ↪workCredentials, 711
Credentials, 709
CultureInfo.CurrentUI ↪Culture, 747
DeclaredMembers, 776
DeclaringType, 777
ElapsedTicks, 579
IsFaulted, 604
Language, 747
LastModified, 720
MemberType, 776
MethodHandle, 777

Kup ksi k
Pole ksi k

właściwość
Name, 777
ReflectedType, 777
StackTrace, 572
UseDefaultCredentials, 710
UseProxy, 709
WPF, 264
wskaźnik, 27, 243, 932
do funkcji, 26, 246, 939
pusty, 246
wskrzeszenie, 544
współbieżność, 260, 583, 624
drobnoziarnista, 610
gruboziarnista, 610, 619
w TCP, 725
wtyczki, 749
tworzenie, 763, 767
wyciek pamięci, 553– 556
wydajność, 783, 853, 870
wyjątek, 186, 711
AggregateException, 911
ArgumentException, 316
CryptographicException, 836
DivideByZeroException, 187, 916
FormatException, 287, 495
IndexOutOfRangeException, 72, 399
NullReference, 320
NullReferenceException, 30, 87, 205, 206, 489, 496
OperationCanceledException, 603, 895, 910
OverflowException, 58
RuntimeBinderException, 235
XmlException, 192, 514
wyjątki, 186, 711
filtry, 189
obsługiwanie, 799
ponawianie zgłoszenia, 191
typy, 193
zgłaszanie, 191
wykonywanie
opóźnione, 398, 400, 428
zapytania lokalnego, 403
wyliczenia, 148, 194, 314, 316, 779
bezpieczeństwo typów, 151
konwersje, 149
operatory, 150
wyliczenie
BindingFlags, 784
DateTimeStyles, 303
MethodAttributes, 804
TaskCreationOptions, 908
TypeAttributes, 800
wyrażenia
dynamiczne, 236
lambda, 18, 37, 180, 390, 565
asynchroniczne, 624
metody lokalne, 185
przechwytywanie zmiennych, 182, 184
statyczne, 183
typy parametrów, 181
podstawowe, 83
przypisania, 83
puste, 83
regularne, 957
asercje o zerowej wielkości, 963
dzielenie tekstu, 967
grupy, 966
grupy nazwane, 967
kompilowane, 959
kotwice, 964
kwantyfikatory, 962
leksykon, 972
opcje, 960
receptury, 969
zastępowanie tekstu, 967
zestawy znaków, 961
zapytań, 37, 393, 407, 429
wyrażenie
switch, 29, 94
throw, 35, 191
wywołania
dynamiczne, 237, 238, 781, 783
zwrotne, 939
niezarządzane, 941
z delegatami, 941
ze wskaźnikami do funkcji, 939
wywoływanie
komponentów COM, 951–954
konstruktorów bazowych, 807
metod generycznych, 782
składowych, 775
dynamiczne, 781
interfejsu generycznego, 785
wzorce, 224
asynchroniczności, 633
blokowanie asynchroniczne, 641
informacje o postępie, 635
łączniki zadań, 638, 640
model programowania, 642
oparte na zadaniu, 637
opartej na zdarzeniach, 643
przerwanie operacji, 633
kombinatory, 226
krotek, 30, 227
metod TryXXX, 194
pozycyjne, 30, 227
relacyjne, 25, 226
typów, 32, 224
własności, 30, 224, 227
zdarzeń, 176
wzorzec
BackgroundWorker, 643
likwidacji obiektów, 868
stałej, 225
usuwania anonimowego, 539
var, 225
Wizytator, 821

X

Xamarin, 23, 266
Xamarin Forms, 266
X-DOM, 481
adnotacje, 506
aktualizowanie danych, 493
definiowanie treści, 486
deklaracje, 497
dokumenty, 497
domyślne przestrzenie nazw, 504
drzewo, 482

Kup ksi k Pole ksi k

eliminowanie pustych elementów, 508
klonowanie
automatyczne, 486
głębokie, 486
konkatenacja węzłów, 497
konstrukcja funkcyjna, 485
ładowanie, 483
modyfikowanie
atrybutów, 492
wartości, 492
węzłów potomnych, 492
nawigacja, 487
do rodzica, 490
na tym samym poziomie, 491
po atrybutach, 491
parsowanie, 483
pobieranie
elementów, 488
elementów potomnych, 490
jednego elementu, 489
wartości, 495
przedrostki, 505
przestrzenie nazw, 503
serializacja, 484
strumieniowanie projekcji, 509
tworzenie drzewa, 484
typy, 482
ustawianie wartości, 495
węzły z treścią mieszaną, 496
wysyłanie zapytań, 487
zapisywanie, 484
zapisywanie deklaracji, 500
XML, 250, 260, 511
dane hierarchiczne, 521
elementy puste, 515
metody wczytujące, 516
odczytywanie elementów, 513
przestrzenie nazw, 501
typy węzłów, 520
wczytywanie
atrybutów, 517
węzłów, 512
węzły atrybutów, 517
wpisywanie atrybutów, 520

Z

zadania, 600
anulowanie, 909
autonomiczne, 604
długo wykonywane, 602
kontynuacje, 604, 910, 911, 912
warunkowe, 912, 913
z wieloma przodkami, 914
łączniki, 638, 640
oczekiwanie, 614, 909
planowanie, 914
potomne, 908, 912
programowanie asynchroniczne, 611
tworzenie, 606, 907
uruchamianie, 501, 907
wartość zwrotna, 602
własna fabryka, 915
zgłaszające wyjątek, 603
zakleszczenia, 851
zakresy, 69, 120
zapytania, 260, *Patrz także* operatory zapytań
budowane progresywnie, 409
do obiektów IAsyncEnumerable<T>, 626
EF Core, 418, 428
interpretowane, 412–416, 429
LINQ, 385–418
lokalne, 429
opakowanie, 409
progresywne, 407
składnia, 394
wykonywanie, 402
zewnętrzne, 403
złożone, 406
zarządzanie pamięcią, 19
zasada pewnego przypisania, 74
zasoby, 741, 742
zdarzenia, 18, 174, 558, 779
definiowanie delegatu, 176
mechanizm działania, 175
modyfikatory, 180
niestandardowe, 581
standardowy wzorzec, 176
systemu plików, 682
wirtualne, 128
zdarzenie
Error, 682
IncludeSubdirectories, 682
Resolving, 752, 761
zegary, 554, 880
jednowątkowe, 882
wielowątkowe, 881, 882
zestaw znaków, 277
zestawy, 20, 261, 730
atrybuty, 732
emitowanie, 800
instalacja certyfikatu, 739
konteksty ładowania, 750
ładowanie, 748, 750, 757
manifest, 731
moduły, 733
nazwy kwalifikowane, 736, 769
osadzanie zasobów, 742
podpisywanie, 734
referencyjne, 258
refleksja, 787
satelickie, 741, 745
testowanie, 747
tworzenie, 746
silne nazwy, 734
technologia Authenticode, 738
wersja
informacyjna, 738
pliku, 738
zaprzyjaźnione, 141
zasoby, 741
znajdowanie, 751, 757
zgłaszanie wyjątków, 191
zmienne
iteracyjne, 184
lokalne, 72, 89, 796
wyjściowe, 32, 77
wzorcowe, 32, 128
zakresowe, 395
znaczniki
dokumentacyjne XML, 251
niestandardowe, 253

Kup ksi k   Pole ksi k

znajdowanie
zależności, 761, 765
zestawów, 748, 751
znak
&, 772
?, 64, 199
@, 42
znaki
interpunkcyjne, 43
sterujące, 960
zrównoleglanie
przetwarzania danych, 885
wykonywania zadań, 885
operatorów zapytań, 890
zużycie pamięci, 540, 556
zwalnianie zasobów, 533, 535

Kup ksi k Pole ksi k

PROGRAM PARTNERSKI

GRUPY HELION

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

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

http://program-partnerski.helion.pl

GRUPA Helion

Trzymaj tę książkę stale na swoim biurku!

C# jest uważany za flagowy produkt firmy Microsoft. Od początku był konsekwentnie rozwijany — z najwyższą troską o elastyczność i wszechstronność. W efekcie pozwala na pisanie bezpiecznego kodu. Wersja 9.0 jest ósmą poważną aktualizacją tego języka. Zapewnia wysokopoziomowe abstrakcje, takie jak wyrażenia, zapytania i kontynuacje asynchroniczne, ale także udostępnia niskopoziomowe mechanizmy pozwalające uzyskać maksymalną wydajność aplikacji. Ten cel jest osiągany dzięki takim konstrukcjom jak własne typy wartościowe programisty czy opcjonalne wskaźniki. Tych nowości trzeba się uczyć, ale czas na to poświęcony rekompensuje przyjemne tworzenie znakomitego kodu.

To zaktualizowane wydanie znakomitego podręcznika dla programistów. Zawiera zwięzłe i dokładne informacje na temat języka C#, Common Language Runtime (CLR), a także biblioteki klas .NET 5 Base Class Library (BCL). Nowe składniki języka C# 9.0 i związanej z nim platformy specjalnie oznaczono, dzięki czemu książka może też służyć jako podręcznik do nauki C# 8.0 i C# 7.0. Znalazły się tu precyzyjne opisy pojęć i przypadków użycia z naciskiem na praktyczność zastosowań. Sporo uwagi poświęcono dość trudnym tematom, jak współbieżność, bezpieczeństwo i dostęp do funkcji systemu operacyjnego. Ten zwięzły przewodnik sprawdzi się doskonale jako stała pomoc w codziennej pracy programisty C#.

W książce między innymi:

- składnia C#, definiowanie zmiennych, wskaźniki, domknięcia i wzorce
- tajniki LINQ i praca na danych
- programowanie współbieżne i asynchroniczne
- zaawansowane techniki pracy z wątkami i programowanie równoległe
- narzędzia platformy.NET i struktury Span oraz kryptografia

Joseph Albahari ma dwudziestoletnie doświadczenie jako architekt i programista tworzący aplikacje dla korporacji. Jest autorem cenionych książek o C#. Napisał też popularny program dla programistów do roboczego wypróbowywania zapytań LINQ — LINQPad.

Helion

helion.pl

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

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

KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-283-8198-8

INFORMATYKA W NAJLEPSZYM WYDANIU

Cena: 179,00 zł