

O'REILLY®

Wydanie VI



# C# 6.0 w pigułce

Helion 

Joseph Albahari, Ben Albahari

Tytuł oryginału: C# 6.0 in a Nutshell, 6th Edition

Tłumaczenie: Łukasz Piwko (wstęp, rozdz. 1 – 11, 20 – 24)  
Robert Górczyński (12 – 18, 26 – 27)  
Jakub Hubisz (rozdz. 19, 25)

ISBN: 978-83-283-2423-7

© 2016 Helion SA

Authorized Polish translation of the English edition C# 6.0 in a Nutshell, 6th Edition  
ISBN 9781491927069 © 2016 Joseph Albahari, Ben 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 jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

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

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

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

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

<http://helion.pl/user/opinie/c6pig6>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:

<ftp://ftp.helion.pl/przyklady/c6pig6.zip>

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

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

---

# Spis treści

<b>Wstęp .....</b>	<b>11</b>
<b>1 Wprowadzenie do C# i .NET Framework .....</b>	<b>17</b>
Obiektowość	17
Bezpieczeństwo typów	18
Zarządzanie pamięcią	19
Platformy	19
Powiązania C# z CLR	19
CLR i .NET Framework	20
C# i środowisko wykonawcze systemu Windows	21
Co nowego w C# 6.0	22
Co było nowego w C# 5.0	24
Co było nowego w C# 4.0	24
Co było nowego w C# 3.0	25
<b>2 Podstawy języka C# .....</b>	<b>27</b>
Pierwszy program w języku C#	27
Składnia	30
Podstawy typów	33
Typy liczbowe	42
Typ logiczny i operatory	49
Łańcuchy znaków i pojedyncze znaki	51
Tablice	53
Zmienne i parametry	57
Wyrażenia i operatory	65
Operatory null	70
Instrukcje	71
Przestrzenie nazw	79

<b>3 Tworzenie typów w języku C# .....</b>	<b>87</b>
Klasy	87
Dziedziczenie	101
Typ object	109
Struktury	113
Modyfikatory dostępu	114
Interfejsy	116
Wyliczenia	121
Typy zagnieżdżone	124
Typy generyczne	125
<b>4 Zaawansowane elementy języka C# .....</b>	<b>139</b>
Delegaty	139
Zdarzenia	147
Wyrażenia lambda	153
Metody anonimowe	157
Instrukcje try i wyjątki	158
Wyliczanie i iteratory	166
Typy wartościowe dopuszczające wartość null	171
Przeciążanie operatorów	177
Metody rozszerzające	180
Typy anonimowe	182
Wiązanie dynamiczne	183
Atrybuty	191
Atrybuty informacji wywołującego	193
Niebezpieczny kod i wskaźniki	194
Dyrektywy preprocesora	198
Dokumentacja XML	200
<b>5 Ogólny zarys platformy .....</b>	<b>205</b>
CLR i rdzeń platformy	207
Technologie praktyczne	212
<b>6 Podstawowe wiadomości o platformie .....</b>	<b>219</b>
Obsługa łańcuchów i tekstu	219
Data i godzina	232
Daty i strefy czasowe	239
Formatowanie i parsowanie	244
Standardowe łańcuchy formatu i flagi parsowania	250
Inne mechanizmy konwersji	257
Globalizacja	261
Praca z liczbami	262

Wyliczenia	266
Krotki	269
Struktura Guid	271
Sprawdzanie równości	271
Określanie kolejności	281
Klasy pomocnicze	284
<b>7 Kolekcje .....</b>	<b>289</b>
Przeliczalność	289
Interfejsy ICollection i IList	296
Klasa Array	300
Listy, kolejki, stosy i zbiory	308
Słowniki	316
Kolekcje i pośredniki z możliwością dostosowywania	322
Dołączanie protokołów równości i porządkowania	328
<b>8 Zapytania LINQ .....</b>	<b>335</b>
Podstawy	335
Składnia płynna	337
Wyrażenia zapytań	343
Wykonywanie opóźnione	347
Podzapytania	353
Tworzenie zapytań złożonych	356
Strategie projekcji	360
Zapytania interpretowane	362
LINQ to SQL i Entity Framework	368
Budowanie wyrażeń zapytań	381
<b>9 Operatory LINQ .....</b>	<b>387</b>
Informacje ogólne	388
Filtrowanie	391
Projekcja	395
Łączenie	406
Porządkowanie	413
Grupowanie	416
Operatory zbiorów	419
Metody konwersji	420
Operatory elementów	423
Metody agregacyjne	425
Kwantyfikatory	429
Metody generujące	430

<b>10 LINQ to XML .....</b>	<b>433</b>
Przegląd architektury	433
Informacje ogólne o X-DOM	434
Tworzenie drzewa X-DOM	437
Nawigowanie i wysyłanie zapytań	440
Modyfikowanie drzewa X-DOM	444
Praca z wartościami	447
Dokumenty i deklaracje	450
Nazwy i przestrzenie nazw	453
Adnotacje	458
Projekcja do X-DOM	459
<b>11 Inne technologie XML .....</b>	<b>465</b>
Klasa XmlReader	466
Klasa XmlWriter	474
Typowe zastosowania klas XmlReader i XmlWriter	476
XSD i sprawdzanie poprawności schematów	480
XSLT	483
<b>12 Zwalnianie zasobów i mechanizm usuwania nieużytków .....</b>	<b>485</b>
IDisposable, Dispose i Close	485
Automatyczne usuwanie nieużytków	491
Finalizatory	493
Jak działa mechanizm usuwania nieużytków?	498
Wycieki pamięci zarządzanej	503
Słabe odwołania	507
<b>13 Diagnostyka i kontrakty kodu .....</b>	<b>511</b>
Kompilacja warunkowa	511
Debugowanie i klasy monitorowania	515
Ogólne omówienie kontraktów kodu	518
Warunki początkowe	523
Warunki końcowe	527
Asercje i metody inwariantów obiektu	529
Kontrakty w interfejsach i metodach abstrakcyjnych	531
Rozwiązywanie problemów z awariami podczas użycia kontraktów	532
Selektywne egzekwowanie kontraktów	534
Statyczne sprawdzenie kontraktu	536
Integracja z debuggerem	538
Procesy i wątki procesów	539
Klasy StackTrace i StackFrame	540
Dziennik zdarzeń Windows	542

Liczniki wydajności	544
Klasa Stopwatch	549
<b>14 Współbieżność i asynchroniczność .....</b>	<b>551</b>
Wprowadzenie	551
Wątkowanie	552
Zadania	569
Reguły asynchroniczności	577
Funkcje asynchroniczne w języku C#	582
Wzorce asynchroniczności	598
Wzorce uznane za przestarzałe	606
<b>15 Strumienie i wejście-wyjście .....</b>	<b>611</b>
Architektura strumienia	611
Użycie strumieni	613
Adapter strumienia	626
Kompresja strumienia	634
Praca z plikami w postaci archiwum ZIP	636
Operacje na plikach i katalogach	637
Plikowe operacje wejścia-wyjścia w środowisku uruchomieniowym Windows	648
Mapowanie plików w pamięci	650
Odizolowany magazyn danych	653
<b>16 Sieć .....</b>	<b>661</b>
Architektura sieci	661
Adresy i porty	664
Adresy URI	665
Klasy po stronie klienta	667
Praca z HTTP	680
Utworzenie serwera HTTP	685
Użycie FTP	688
Użycie DNS	690
Wysyłanie poczty elektronicznej za pomocą Smtplib	691
Użycie TCP	692
Otrzymywanie poczty elektronicznej POP3 za pomocą TcpClient	695
TCP w środowisku uruchomieniowym Windows	697
<b>17 Serializacja .....</b>	<b>699</b>
Koncepcje serializacji	699
Serializacja kontraktu danych	703
Kontrakty danych i kolekcje	713
Rozszerzenie kontraktu danych	715

Serializacja binarna	718
Atrybuty serializacji binarnej	720
Serializacja binarna za pomocą ISerializable	724
Serializacja XML	727
<b>18 Podzespoły .....</b>	<b>737</b>
Co znajduje się w podzespołe?	737
Silne nazwy i podpisywanie podzespołu	742
Nazwy podzespołów	745
Technologia Authenticode	748
Global Assembly Cache	751
Zasoby i podzespoły satelickie	754
Wyszukiwanie i wczytywanie podzespołów	762
Wdrażanie podzespołów poza katalogiem bazowym	768
Umieszczenie w pojedynczym pliku wykonywalnym	769
Praca z podzespołami, do których nie ma odwołań	771
<b>19 Refleksje i metadane .....</b>	<b>773</b>
Refleksje i aktywacja typów	774
Refleksje i wywoływanie składowych	780
Refleksje dla podzespołów	792
Praca z atrybutami	793
Generowanie dynamicznego kodu	799
Emitowanie podzespołów i typów	805
Emitowanie składowych typów	809
Emitowanie generycznych typów i klas	814
Kłopotliwe cele emisji	816
Parsowanie IL	819
<b>20 Programowanie dynamiczne .....</b>	<b>825</b>
Dynamiczny system wykonawczy języka	825
Unifikacja typów liczbowych	827
Dynamiczne wybieranie przeciążonych składowych	828
Implementowanie obiektów dynamicznych	833
Współpraca z językami dynamicznymi	836
<b>21 Bezpieczeństwo .....</b>	<b>839</b>
Uprawnienia	839
Zabezpieczenia dostępu kodu	843
Dopuszczanie częściowo zaufanych wywołujących	846
Model transparentności	848



Ograniczanie innego zestawu	856
Zabezpieczenia systemu operacyjnego	860
Tożsamości i role	862
Kryptografia	864
Windows Data Protection	864
Obliczanie skrótów	865
Szyfrowanie symetryczne	867
Szyfrowanie kluczem publicznym i podpisywanie	871
<b>22 Zaawansowane techniki wielowątkowości .....</b>	<b>877</b>
Przegląd technik synchronizacji	878
Blokowanie wykluczające	878
Blokady i bezpieczeństwo ze względu na wątki	886
Blokowanie bez wykluczania	892
Sygnalizacja przy użyciu uchwytów zdarzeń oczekiwania	897
Klasa Barrier	905
Leniwa inicjalizacja	906
Pamięć lokalna wątku	909
Metody Interrupt i Abort	911
Metody Suspend i Resume	912
Zegary	913
<b>23 Programowanie równoległe .....</b>	<b>917</b>
Dlaczego PFX?	917
PLINQ	920
Klasa Parallel	933
Równoległe wykonywanie zadań	939
Klasa AggregateException	948
Kolekcje współbieżne	951
Klasa BlockingCollection<T>	954
<b>24 Domeny aplikacji .....</b>	<b>959</b>
Architektura domeny aplikacji	959
Tworzenie i likwidowanie domen aplikacji	961
Posługiwanie się wieloma domenami aplikacji	962
Metoda DoCallBack	964
Monitorowanie domen aplikacji	965
Domeny i wątki	965
Dzielenie danych między domenami	967

<b>25 Współpraca .....</b>	<b>973</b>
Odwołania do natywnych bibliotek DLL	973
Szeregowanie	974
Wywołania zwrotne z kodu niezarządzonego	977
Symulowanie unii C	977
Pamięć współdzielona	978
Mapowanie struktury na pamięć niezarządzaną	981
Współpraca COM	985
Wywołanie komponentu COM z C#	986
Osadzanie typów współpracujących	990
Główne moduły współpracujące	990
Udostępnianie obiektów C# dla COM	991
<b>26 Wyrażenia regularne .....</b>	<b>993</b>
Podstawy wyrażeń regularnych	994
Kwantyfikatory	998
Asercje o zerowej wielkości	999
Grupy	1002
Zastępowanie i dzielenie tekstu	1003
Receptury wyrażeń regularnych	1005
Leksykon języka wyrażeń regularnych	1008
<b>27 Kompilator Roslyn .....</b>	<b>1013</b>
Architektura Roslyn	1014
Drzewa składni	1015
Kompilacja i model semantyczny	1030
<b>Skorowidz .....</b>	<b>1041</b>



Platforma .NET Framework zapewnia standardowy zestaw typów do sortowania i obsługi kolekcji obiektów. Wśród nich można znaleźć listy o zmiennym rozmiarze, listy powiązane, sortowane i niesortowane słowniki oraz tablice. Z tych wszystkich typów jedynie tablice należą do języka C#. Pozostałe kolekcje są tylko klasami, których obiekty można tworzyć tak samo jak obiekty wszystkich innych klas.

Typy kolekcji platformy można podzielić na następujące kategorie:

- interfejsy definiujące standardowe protokoły kolekcji;
- gotowe do użycia klasy kolekcji (listy, słowniki itd.);
- klasy bazowe do pisania kolekcji specjalnie dostosowanych do potrzeb konkretnych aplikacji.

W tym rozdziale opisujemy wszystkie te kategorie oraz dodatkowo poświęcamy nieco miejsca typom wykorzystywanym do porównywania i porządkowania elementów.

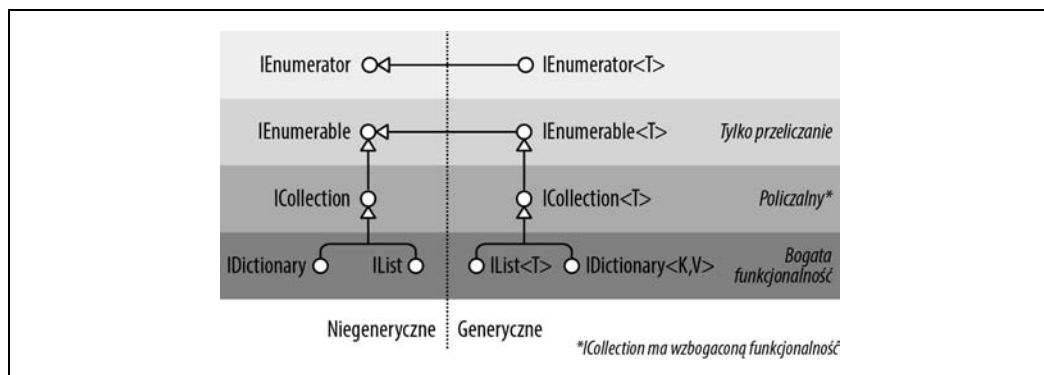
Oto wykaz przestrzeni nazw, w których znajdują się różne kolekcje.

Przeźrzeń nazw	Zawartość
System.Collections	Niegeneryczne klasy i interfejsy kolekcji
System.Collections.Specialized	Silnie typizowane niegeneryczne klasy kolekcji
System.Collections.Generic	Generyczne klasy i interfejsy kolekcji
System.Collections.ObjectModel	Klasy pośrednie i bazowe do tworzenia niestandardowych kolekcji
System.Collections.Concurrent	Kolekcje bezpieczne wątkowo (zob. rozdział 23.)

## Przeliczalność

Istnieje wiele różnych rodzajów kolekcji, od prostych struktur danych, przez tablice i listy powiązane po złożone struktury, takie jak drzewa czerwono-czarne i tablice skrótów. Choć konstrukcje te znacznie różnią się pod względem budowy zarówno wewnętrznych, jak i zewnętrznych cech, prawie wszystkie z nich można przeglądać. Platforma .NET zapewnia tę możliwość przez dwa interfejsy

(IEnumerable i IEnumerator oraz ich generyczne odpowiedniki), dzięki którym różne struktury danych udostępniają jednaki interfejs API do przeglądania ich zawartości. Wymienione interfejsy należą do szerszego zbioru przedstawionego na rysunku 7.1.



Rysunek 7.1. Interfejsy kolekcji

## Interfejsy IEnumerable i IEnumerator

Interfejs IEnumerator definiuje podstawowy niskopoziomowy protokół określający sposób przeglądania elementów kolekcji do przodu. Jego deklaracja wygląda tak:

```
public interface IEnumerator
{
    bool MoveNext();
    object Current { get; }
    void Reset();
}
```

Metoda MoveNext przesuwa bieżący element zwany kursorem o jedną pozycję dalej i zwraca fałsz, jeśli był to ostatni element tej kolekcji. Metoda Current zwraca bieżący element (zazwyczaj rzutowany z typu object na bardziej konkretny typ). Metoda MoveNext musi zostać wywołana przed pobraniem pierwszego elementu — zasadę tę wprowadzono, aby było możliwe tworzenie pustych kolekcji. Metoda Reset, jeśli jest zaimplementowana, przenosi kursor z powrotem na początek, aby można było od nowa przejrzeć kolekcję. Metoda ta znajduje zastosowanie głównie przy współpracy z technologią COM. Raczej nie wywołuje się jej bezpośrednio, ponieważ nie jest wszędzie obsługiwana (ani niezbędna, gdyż w większości przypadków równie dobrze można utworzyć nowy enumerator).

Tylko nieliczne kolekcje *implementują* enumeratory. Większość *udostępnia* enumeratory przez interfejs IEnumerable:

```
public interface IEnumerable
{
    IEnumerator GetEnumerator();
}
```

Dzięki definicji tylko jednej metody zwracającej enumerator interfejs IEnumerable zapewnia elastyczność umożliwiającą dostarczenie logiki iteracji w innej klasie. Ponadto dzięki temu kolekcję może przeglądać kilku konsumentów jednocześnie i w żaden sposób nie będą sobie przeszkadzać. Interfejs IEnumerable można traktować jak „IEnumeratorProvider” i jest to najbardziej podstawowy interfejs implementowany przez klasy kolekcji.

Poniżej znajduje się przykład niskopoziomowego wykorzystania interfejsów `IEnumerable` i `IEnumerator`:

```
string s = "Cześć";

// klasa string implementuje interfejs IEnumerable, więc możemy wywołać metodę GetEnumerator():
IEnumerator rator = s.GetEnumerator();

while (rator.MoveNext())
{
    char c = (char) rator.Current;
    Console.Write (c + ".");
}

// wynik: C.z.e.ś.ć.
```

Jednak taki bezpośredni sposób wywoływania metod na enumeratorach należy do rzadkości, ponieważ w języku C# istnieje wygodny skrót składniowy w postaci instrukcji `foreach`. Oto ten sam przykład napisany z użyciem tej właśnie instrukcji:

```
string s = "Cześć"; // klasa string implementuje interfejs IEnumerable

foreach (char c in s)
    Console.Write (c + ".");
```

## Interfejsy `IEnumerable<T>` i `IEnumerator<T>`

Interfejsy `IEnumerator` i `IEnumerable` są prawie zawsze implementowane w parze ze swoimi rozszerzonymi generycznymi wersjami:

```
public interface IEnumerator<T> : IEnumerator, IDisposable
{
    T Current { get; }
}

public interface IEnumerable<T> : IEnumerable
{
    IEnumerator<T> GetEnumerator();
}
```

Dzięki definicjom typizowanych wersji metod `Current` i `GetEnumerator` interfejsy te wzmacniają bezpieczeństwo typów, eliminują konieczność pakowania elementów typów wartościowych oraz są wygodniejsze w użyciu. Tablice automatycznie implementują interfejs `IEnumerable<T>` (gdzie `T` jest typem elementów przechowywanych w tablicy).

Dzięki zwiększonemu bezpieczeństwu dla typów wywołanie poniższej metody z tablicą znaków spowoduje błąd kompilacji:

```
void Test (IEnumerable<int> numbers) { ... }
```

W klasach kolekcji standardowo udostępnia się publicznie elementy interfejsu `IEnumerable<T>`, a „ukrywa” elementy niegenerycznego interfejsu `IEnumerable` przez zastosowanie jawnej implementacji tego pierwszego. Dzięki temu, jeśli programista bezpośrednio wywoła metodę `GetEnumerator()`, to otrzyma bezpieczny pod względem typów `IEnumerator<T>`. Zdarzają się jednak przypadki złamania tej zasady ze względu na zgodność ze starym kodem (typy generyczne wprowadzono dopiero w C# 2.0). Dobrym przykładem są tablice, które muszą zwracać niegeneryczny (można ładniej powiedzieć: klasyczny) `IEnumerator`, aby nie spowodować awarii starych partii kodu. Aby otrzymać generyczny `IEnumerator<T>`, należy dokonać rzutowania, by udostępnić jawny interfejs:

```
int[] data = { 1, 2, 3 };
var rator = ((IEnumerable<int>)data).GetEnumerator();
```

Na szczęście dzięki instrukcji `foreach` rzadko jest to konieczne.

## Interfejsy `IEnumerable<T>` i `IDisposable`

`IEnumerator<T>` dziedziczy po `IDisposable`. Dzięki temu enumeratory mogą przechowywać referencje do takich zasobów jak połączenia z bazą danych i zwalniać je po zakończeniu lub przerwaniu pracy. Instrukcja `foreach` „ma świadomość” tego szczegółu i tłumaczy taki kod:

```
foreach (var element in somethingEnumerable) { ... }
```

na następujący logiczny ekwiwalent:

```
using (var rator = somethingEnumerable.GetEnumerator())
while (rator.MoveNext())
{
    var element = rator.Current;
    ...
}
```

## Kiedy używać interfejsów niegenerycznych

Biorąc pod uwagę zwiększone bezpieczeństwo typowe generycznych interfejsów kolekcji, takich jak `IEnumerable<T>`, można zadać pytanie: czy niegeneryczna wersja `IEnumerable` (lub `ICollection` albo  `IList`) w ogóle jest do czegoś potrzebna?

Jeśli chodzi o interfejs `IEnumerable`, to musi być implementowany w połączeniu z `IEnumerable<T>`, ponieważ ten drugi korzysta z pierwszego. Jednak niezmiernie rzadko pisze się implementacje tych interfejsów od początku — najczęściej można wykorzystać metody iteracyjne, `Collection<T>` oraz LINQ.

A co z konsumentem? Prawie zawsze wystarczają interfejsy generyczne. Wprawdzie niegeneryczne wersje też są czasami przydatne, choć raczej do zapewnienia spójności typów wszystkich elementów kolekcji. Poniższa metoda np. **rekurencyjnie** liczy elementy w każdej kolekcji:

```
public static int Count (IEnumerable e)
{
    int count = 0;
    foreach (object element in e)
    {
        var subCollection = element as IEnumerable;
        if (subCollection != null)
            count += Count (subCollection);
        else
            count++;
    }
    return count;
}
```

Ponieważ w języku C# występuje kowariancja interfejsów generycznych, można się spodziewać, że ta metoda powinna przyjmować typ `IEnumerable<object>`. To jednak nie udałoby się z elementami typów wartościowych i ze starymi kolekcjami, które nie implementują interfejsu `IEnumerable<T>` — przykładem jest `ControlCollection` z `Windows Forms`.

Tak na marginesie: uważny czytelnik mógł zauważyć w naszym przykładzie potencjalny błąd — **cykliczne** referencje spowodują nieskończoną rekurencję i awarię metody. Problem ten można łatwo usunąć, używając kolekcji `HashSet` (zob. sekcję „Klasy `HashSet<T>` i `SortedSet<T>`”).

Blok usi ng zapewnia odpowiednie załatwienie zasobów — szerzej na temat interfejsu IDisposable piszemy w rozdziale 12.

## Implementowanie interfejsów przeliczeniowych

Interfejsy `IEnumerable` i `IEnumerable<T>` można zaimplementować z następujących powodów:

- aby umożliwić korzystanie z instrukcji `foreach`;
- aby zapewnić możliwość współpracy ze wszystkim, co oczekuje standardowej kolekcji;
- aby spełnić wymagania bardziej zaawansowanego interfejsu kolekcji;
- aby obsługiwać inicjalizatory kolekcji.

Aby zaimplementować interfejs `IEnumerable` lub `IEnumerable<T>`, należy dostarczyć enumerator. Można to zrobić na jeden z trzech sposobów:

- jeżeli klasa „opakowuje” inną kolekcję, można zwrócić enumerator tej opakowanej kolekcji;
- przez iterator za pomocą instrukcji `yield return`;
- tworząc własną implementację interfejsu `IEnumerable` lub `IEnumerable<T>`.



Można też utworzyć podklasę istniejącej kolekcji. Klasa `Collection<T>` służy właśnie do tego celu (zob. podrozdział „Kolekcje i pośredniki z możliwością dostosowywania”). Inną możliwością jest użycie operatorów zapytań LINQ, o których mowa w następnym rozdziale.

Zwrócenie enumeratora innej kolekcji jest zaledwie kwestią wywołania metody `GetEnumerator` na wewnętrznej kolekcji. Jednak takie rozwiązanie jest możliwe jedynie w najprostszyc h przypadkach, gdy elementy wewnętrznej kolekcji są dokładnie tym, czym powinny być. Bardziej elastyczne rozwiązanie polega na napisaniu iteratora przy użyciu instrukcji `yield return`. **Iterator** to element języka C# pomocny w pisaniu kolekcji na podobnej zasadzie, jak instrukcja `foreach` jest pomocna w ich konsumowaniu. Iterator automatycznie rozwiązuje kwestię implementacji interfejsów `IEnumerable` i `IEnumerator` lub ich generycznych wersji. Oto prosty przykład:

```
public class MyCollection : IEnumerable
{
    int[] data = { 1, 2, 3 };

    public IEnumerator GetEnumerator()
    {
        foreach (int i in data)
            yield return i;
    }
}
```

Zwróć uwagę na zawartą w tym kodzie „czarną magię” — metoda `GetEnumerator` nie wygląda tak, jakby miała zwracać enumerator! Kompilator, parsując instrukcję `yield return`, tworzy ukrytą zagnieźdżoną klasę enumeratora, a następnie refaktoryzuje metodę `GetEnumerator` w taki sposób, aby tworzyła i zwracała egzemplarz tej klasy. Iteratory są proste i potężne (i często znajdują zastosowanie w implementacji standardowych operatorów zapytań LINQ to Object).

Zgodnie z tą linią możemy też zaimplementować generyczny interfejs `IEnumerable<T>`:

```
public class MyGenCollection : IEnumerable<int>
{
    int[] data = { 1, 2, 3 };

    public IEnumerator<int> GetEnumerator()
    {
        foreach (int i in data)
            yield return i;
    }

    IEnumerator IEnumerable.GetEnumerator() //jawna implementacja
    { //ukrywa go
        return GetEnumerator();
    }
}
```

Jako że interfejs `IEnumerable<T>` dziedziczy po `IEnumerable`, musimy zaimplementować zarówno generyczną, jak i niegeneryczną wersję metody `GetEnumerator`. Dodatkowo zgodnie ze standardową praktyką zaimplementowaliśmy też jawnie niegeneryczną wersję. Może ona po prostu wywoływać generyczną metodę `GetEnumerator`, ponieważ interfejs `IEnumerator<T>` dziedziczy po `IEnumerator`.

Napisana przez nas klasa mogłaby zostać wykorzystana jako baza do napisania bardziej zaawansowanej kolekcji. Jeśli jednak potrzebna jest tylko prosta implementacja interfejsu `IEnumerable<T>`, instrukcja `yield return` umożliwia skorzystanie z prostszego rozwiązania. Zamiast pisać klasę, logikę iteracji można umieścić w metodzie zwracającej generyczną kolekcję typu `IEnumerable<T>` i resztę zostawić kompilatorowi. Oto przykład:

```
public class Test
{
    public static IEnumerable <int> GetSomeIntegers()
    {
        yield return 1;
        yield return 2;
        yield return 3;
    }
}
```

A oto przykład użycia naszej metody:

```
foreach (int i in Test.GetSomeIntegers())
    Console.WriteLine (i);

//wynik
1
2
3
```

Ostatnim sposobem napisania metody `GetEnumerator` jest napisanie klasy bezpośrednio implementującej interfejs `IEnumerator`. Dokładnie to robi kompilator niejawnie, przy rozpoznawaniu iteratorów. (Na szczęście nieczęsto trzeba posuwać się tak daleko we własnym kodzie). Poniżej znajduje się przykład kolekcji z wpisanymi na stałe wartościami 1, 2 i 3:

```
public class MyIntList : IEnumerator
{
    int[] data = { 1, 2, 3 };
}
```



```

public IEnumerator GetEnumerator()
{
    return new Enumerator (this);
}

class Enumerator : IEnumerator // definicja wewnętrznej klasy
                               // dla enumeratora
{
    MyIntList collection;
    int currentIndex = -1;

    public Enumerator (MyIntList collection)
    {
        this.collection = collection;
    }

    public object Current
    {
        get
        {
            if (currentIndex == -1)
                throw new InvalidOperationException ("Enumeracja nie została rozpoczęta!");
            if (currentIndex == collection.data.Length)
                throw new InvalidOperationException ("Za końcem listy!");
            return collection.data [currentIndex];
        }
    }

    public bool MoveNext()
    {
        if (currentIndex >= collection.data.Length - 1) return false;
        return ++currentIndex < collection.data.Length;
    }

    public void Reset() { currentIndex = -1; }
}
}

```



Implementacja metody `Reset` jest nieobowiązkowa — ewentualnie można zgłaszać wyjątek `NotSupportedException`.

Podkreślmy, że pierwsze wywołanie metody `MoveNext` powinno powodować przejście do pierwszego (a nie drugiego) elementu listy.

Aby uzyskać funkcjonalność zbliżoną do iteratora, musimy jeszcze dodatkowo zaimplementować interfejs `IEnumerator<T>`. Poniżej przedstawiamy przykład z pominięciem testów granic dla uproszczenia:

```

class MyIntList : IEnumerable<int>
{
    int[] data = { 1, 2, 3 };

    // Generyczny enumerator jest zgodny zarówno z IEnumerable, jak i IEnumerable<T>.
    // Implementujemy niegeneryczną metodę GetEnumerator jawnie, aby uniknąć konfliktów nazw.

    public IEnumerator<int> GetEnumerator() { return new Enumerator(this); }
    IEnumerator IEnumerable.GetEnumerator() { return new Enumerator(this); }
}

```

```

class Enumerator : IEnumerator<int>
{
    int currentIndex = -1;
    MyIntList collection;

    public Enumerator (MyIntList collection)
    {
        this.collection = collection;
    }

    public int Current => collection.data [currentIndex];

    object IEnumerator.Current => Current;

    public bool MoveNext() => ++currentIndex < collection.data.Length;

    public void Reset() => currentIndex = -1;
    // Jeśli nie jest potrzebna metoda Dispose, to dobrym zwyczajem jest jej
    // jawne zaimplementowanie tak, aby była niedostępna w interfejsie publicznym.
    void IDisposable.Dispose() {}
}
}

```

Przykład z użyciem typów generycznych jest szybszy, ponieważ metoda `IEnumerator<int>.Current` nie wymaga rzutowania z `int` na `object`, a więc eliminuje pakowanie.

## Interfejsy ICollection i IList

Choć interfejsy enumeracyjne stanowią protokół iteracji kolekcji tylko do przodu, nie zapewniają możliwości sprawdzania rozmiaru kolekcji, dostępu do składowych za pomocą indeksów, przeszukiwania struktur danych ani ich modyfikowania. Wszystkie te funkcje zapewniają interfejsy .NET Framework `ICollection`, `IList` oraz `IDictionary`. Każdy z nich występuje w wersji generycznej i niegenerycznej, choć te drugie istnieją głównie ze względu na zgodność ze starym kodem.

Hierarchia dziedziczenia tych interfejsów jest pokazana na rysunku 7.1. Najprościej można je podsumować w następujący sposób:

`IEnumerable<T>` (i `IEnumerable`)

Zapewnia minimalną funkcjonalność (tylko przeglądanie).

`ICollection<T>` (i `ICollection`)

Zapewnia średni poziom funkcjonalności (np. własność `Count`).

`IList <T>/IDictionary <K,V>` i ich niegeneryczne wersje

Zapewnia najwyższy poziom funkcjonalności (wliczając dostęp „swobodny” przy użyciu indeksów i kluczy).



Konieczność *implementowania* któregośkolwiek z tych interfejsów jest rzadkością. Gdy trzeba napisać klasę kolekcji, to prawie zawsze można wykorzystać do tego klasę bazową `Collection<T>` (zob. podrozdział „Kolekcje i pośredniki z możliwością dostosowywania”). W niektórych przypadkach inną możliwość zapewnia też technologia LINQ.

Różnice między wersjami generycznymi i niegenerycznymi są większe niż można się spodziewać, zwłaszcza w przypadku interfejsu `ICollection`. Przyczyny tego są w głównej mierze historyczne — typy generyczne powstały później, więc interfejsy generyczne tworzone z pewnym bagażem doświadczenia, dzięki czemu udało się dobrać inne (i lepsze) składowe. Dlatego właśnie interfejs `ICollection<T>` nie rozszerza interfejsu `ICollection`, `IList<T>` nie rozszerza interfejsu `IList`, a `IDictionary<TKey, TValue>` nie rozszerza interfejsu `IDictionary`. Oczywiście klasa kolekcji może implementować obie wersje interfejsu, jeśli jest to korzystne (a często jest).



Innym, mniej oczywistym powodem, dla którego `IList<T>` nie rozszerza interfejsu `IList`, jest to, że rzutowanie na `IList<T>` zwracałoby interfejs ze składowymi `Add(T)` i `Add(object)`. To z kolei oznaczałoby zniweczenie bezpieczeństwa typowego, ponieważ można byłoby wywołać metodę `Add` z obiektem dowolnego typu.

W tym podrozdziale są opisane interfejsy `ICollection<T>`, `IList<T>` i ich niegeneryczne wersje. Opis interfejsów słownikowych znajduje się w podrozdziale „Słowniki”.



W obrębie platformy .NET Framework słowa **kolekcja** i **lista** są używane bez *dającej się uchwycić* logiki. Na przykład interfejs `IList<T>` jest bardziej funkcjonalną wersją interfejsu `ICollection<T>`, więc można oczekiwać, że klasa `List<T>` będzie tak samo bardziej funkcjonalna niż klasa `Collection<T>`. Jednak tak nie jest. Dlatego terminy **kolekcja** i **lista** najlepiej traktować jako synonimy, chyba że chodzi o konkretny typ.

## Interfejsy `ICollection<T>` i `ICollection`

`ICollection<T>` to standardowy interfejs dla policzalnych kolekcji obiektów. Zapewnia możliwość sprawdzenia rozmiaru kolekcji (`Count`), sprawdzenia, czy dany element znajduje się w kolekcji (`Contains`), skopiowania kolekcji do tablicy (`ToArray`) oraz stwierdzenia, czy strukturę można modyfikować (`IsReadOnly`). W przypadku kolekcji z możliwością zapisu dostępne są także operacje `Add`, `Remove` i `Clear`. A ponieważ interfejs ten rozszerza `IEnumerable<T>`, kolekcje można przeglądać za pomocą instrukcji `foreach`:

```
public interface ICollection<T> : IEnumerable<T>, IEnumerable
{
    int Count { get; }

    bool Contains (T item);
    void CopyTo (T[] array, int arrayIndex);
    bool IsReadOnly { get; }

    void Add(T item);
    bool Remove (T item);
    void Clear();
}
```

Niegeneryczny interfejs `ICollection` także opisuje policzalną kolekcję, ale nie zapewnia funkcji do modyfikowania listy ani sprawdzania, czy zawiera określony element:

```
public interface ICollection : IEnumerable
{
    int Count { get; }
    bool IsSynchronized { get; }
```

```

    object SyncRoot { get; }
    void CopyTo (Array array, int index);
}

```

Niegeneryczny interfejs dodatkowo definiuje własności przydatne w synchronizacji (rozdział 14.) — wstawiono je do wersji generycznej, ponieważ bezpieczeństwo wątków nie jest już uważane za cechę wewnętrzną kolekcji.

Implementacja obu interfejsów jest łatwa. W przypadku implementacji interfejsu tylko do odczytu `ICollection<T>` metody `Add`, `Remove` i `Clear` powinny zgłaszać wyjątek `NotSupportedException`.

Z reguły interfejsy te implementuje się łącznie z `IList` lub `IDictionary`.

## Interfejsy `IList<T>` i `IList`

`IList<T>` to standardowy interfejs kolekcji indeksowanych pozycyjnie. Oprócz składników odziedziczonych z interfejsów `ICollection<T>` i `IEnumerable<T>` zawiera funkcje pozwalające odczytywać, zapisywać (za pomocą indeksatora), wstawiać oraz usuwać elementy wg pozycji:

```

public interface IList<T> : ICollection<T>, IEnumerable<T>, IEnumerable
{
    T this [int index] { get; set; }
    int IndexOf (T item);
    void Insert (int index, T item);
    void RemoveAt (int index);
}

```

Metody `IndexOf` wykonują przeszukiwanie liniowe listy i zwracają wartość `-1`, jeśli nie znajdują szukanego elementu.

Niegeneryczna wersja interfejsu `IList` ma więcej składowych, ponieważ mniej dziedziczy po `ICollection`:

```

public interface IList : ICollection, IEnumerable
{
    object this [int index] { get; set }
    bool IsFixedSize { get; }
    bool IsReadOnly { get; }
    int Add (object value);
    void Clear();
    bool Contains (object value);
    int IndexOf (object value);
    void Insert (int index, object value);
    void Remove (object value);
    void RemoveAt (int index);
}

```

Metoda `Add` niegenerycznego interfejsu `IList` zwraca liczbę całkowitą będącą indeksem nowo dodanego elementu. Dla porównania metoda `Add` interfejsu `ICollection<T>` ma typ zwrotny `void`.

Ogólna klasa `List<T>` jest typową implementacją interfejsów `IList<T>` i `IList`. W języku C# także tablice implementują zarówno generyczną, jak i niegeneryczną wersję (choć metody dodające i usuwające elementy są ukryte przez jawną implementację interfejsu i w razie wywołania zgłaszają wyjątek `NotSupportedException`).



Przy próbie uzyskania dostępu do wielowymiarowej tablicy za pomocą indeksatora interfejsu `IList` zgłaszany jest wyjątek `ArgumentException`. Jest to pułapka dla programistów piszących takie metody jak poniższa:

```
public object FirstOrNull (IList list)
{
    if (list == null || list.Count == 0) return null;
    return list[0];
}
```

Na pierwszy rzut oka może się wydawać, że to bardzo solidna funkcja, ale zgłosi wyjątek, jeśli ktoś wywoła ją na tablicy wielowymiarowej. W razie potrzeby w czasie działania programu można testować, czy argument nie jest tablicą wielowymiarową, za pomocą poniższego wyrażenia (szerzej na ten temat piszemy w rozdziale 19.):

```
list.GetType().IsArray && list.GetType().GetArrayRank()>1
```

## Interfejs `IReadOnlyList<T>`

W celu zapewnienia możliwości współpracy z kolekcjami tylko do odczytu Windows Runtime w .NET Framework 4.5 wprowadzono nowy interfejs kolekcji o nazwie `IReadOnlyList<T>`. Jest on jednak przydatny sam w sobie i można go traktować jak okrojoną wersję interfejsu `IList<T>` udostępniającą tylko składowe potrzebne do wykonywania operacji odczytu na listach:

```
public interface IReadOnlyList<out T> : IEnumerable<T>, IEnumerable
{
    int Count { get; }
    T this[int index] { get; }
}
```

Ponieważ parametr typu jest wykorzystywany wyłącznie na pozycjach wyjściowych, oznaczono go jako kowariantny. Dzięki temu np. listę kotów można traktować jako listę zwierząt tylko do odczytu. Natomiast w interfejsie `IList<T>` parametr `T` nie jest oznaczony jako kowariantny, ponieważ znajduje zastosowanie zarówno na pozycjach wejściowych, jak i wyjściowych.



Interfejs `IReadOnlyList<T>` reprezentuje **widok** listy tylko do odczytu. Nie oznacza to jednak, że podstawowa implementacja także musi być tylko do odczytu.

Zgodnie z logiką interfejs `IList<T>` powinien dziedziczyć po `IReadOnlyList<T>`. Jednak firma Microsoft nie mogła wprowadzić takiej zmiany, ponieważ wymagałaby ona przeniesienia składowych z `IList<T>` do `IReadOnlyList<T>`, co spowodowałoby problemy z programami działającymi pod kontrolą CLR 4.5 (konsumenci musieliby ponownie skompilować swoje programy, aby uniknąć błędów wykonawczych). Dlatego implementatorzy interfejsu `IList<T>` muszą ręcznie dodawać do swoich kolekcji implementację interfejsu `IReadOnlyList<T>`.

Interfejs `IReadOnlyList<T>` odpowiada typowi Windows Runtime `IVectorView<T>`.

# Klasa Array

Klasa Array to podstawa wszystkich jedno- i wielowymiarowych tablic oraz jeden z podstawowych typów implementujących standardowe interfejsy kolekcji. Ponieważ zapewnia unifikację typów, wszystkie tablice dysponują takim samym zestawem metod, niezależnie od ich deklaracji i typu elementów.

Ze względu na wielkie znaczenie tablic w języku C# utworzono specjalną składnię do ich deklarowania i inicjalizowania, której opis znajduje się w rozdziałach 2. i 3. Gdy programista deklaruje tablicę za pomocą składni C#, system CLR niejawnie generuje podtyp klasy Array, tworząc **pseudotyp** o odpowiednich wymiarach i typie elementów. Ten typ implementuje typizowane generyczne interfejsy kolekcji, np. `IList<string>`.

Ponadto system CLR traktuje typy tablicowe w specjalny sposób, przypisując tworzonym obiektom ciągiły obszar pamięci. Dzięki temu indeksowanie jest bardzo szybkie, ale za cenę braku możliwości zwiększenia rozmiaru struktury w późniejszym czasie.

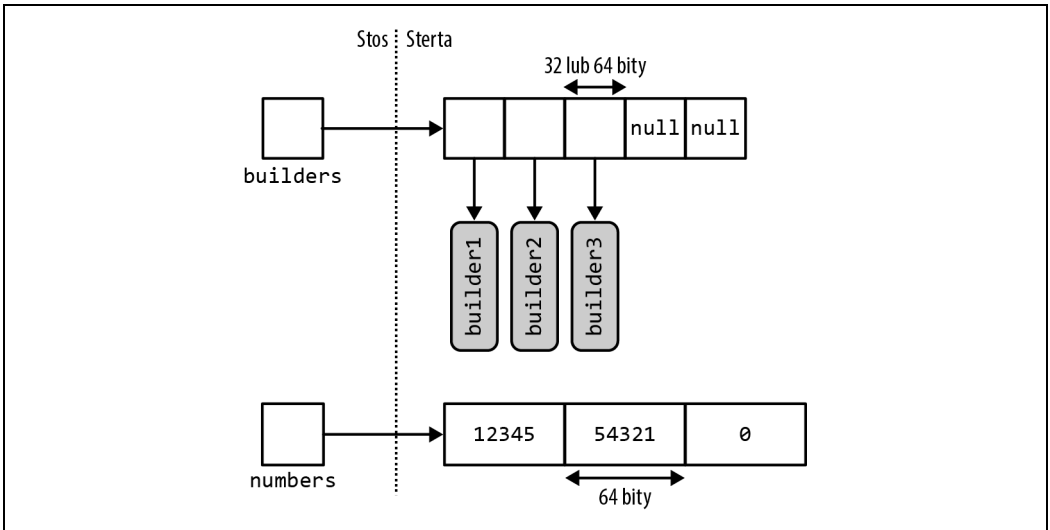
Klasa Array implementuje interfejsy kolekcji do `IList<T>` w wersjach generycznych i niegenerycznych. Natomiast interfejs `IList<T>` jest zaimplementowany jawnie w celu utrzymania porządku w interfejsie publicznym tablicy, który nie może udostępniać takich metod jak `Add` czy `Remove`, ponieważ zgłaszają one wyjątki, jeśli wywoła się je na kolekcji o niezmiennym rozmiarze. Wprawdzie klasa Array udostępnia statyczną metodę o nazwie `Resize` do zmieniania rozmiaru tablicy, ale jej działanie polega na utworzeniu nowej tablicy i skopiowaniu do niej wszystkich elementów. Nie dość, że ta operacja jest bardziej nieefektywna, to na dodatek wszystkie znajdujące się w różnych miejscach programu referencje nadal będą wskazywać starą strukturę. Jeśli potrzebny jest obiekt o zmiennym rozmiarze, to lepiej użyć klasy `List<T>` (opisanej w następnym podrozdziale).

W tablicy można przechowywać elementy typów zarówno wartościowych, jak i referencyjnych. Elementy wartościowe są przechowywane bezpośrednio w tablicy, więc struktura zawierająca trzy liczby całkowite (każda po 8 bajtów) zajmuje ciągiły obszar pamięci o rozmiarze 24 bajtów. Natomiast elementy typów referencyjnych zajmują w tablicy tylko tyle miejsca, ile potrzeba do przechowywania referencji (4 bajty w środowisku 32-bitowym i 8 bajtów w środowisku 64-bitowym). Na rysunku 7.2 pokazano, co dzieje się w pamięci po wykonaniu poniższego programu:

```
StringBuilder[] builders = new StringBuilder [5];
builders [0] = new StringBuilder ("builder1");
builders [1] = new StringBuilder ("builder2");
builders [2] = new StringBuilder ("builder3");

long[] numbers = new long [3];
numbers [0] = 12345;
numbers [1] = 54321;
```

Jako że Array to klasa, same tablice są typami referencyjnymi, niezależnie od rodzaju przechowywanych w nich elementów. Oznacza to, że wynikiem wykonania instrukcji `tablicaB = tablicaA` będzie powstanie dwóch zmiennych odnoszących się do tej samej tablicy. Jednocześnie test równości dwóch różnych tablic zawsze będzie miał wynik negatywny, chyba że programista użyje własnego



Rysunek 7.2. Tablice w pamięci

komparatora. Jeden taki komparator wprowadzono w .NET Framework 4.0, aby umożliwić porównywanie elementów w tablicach lub krotkach. Dostęp do niego można uzyskać przez typ `StructuralComparisons`:

```
object[] a1 = { "string", 123, true };
object[] a2 = { "string", 123, true };

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

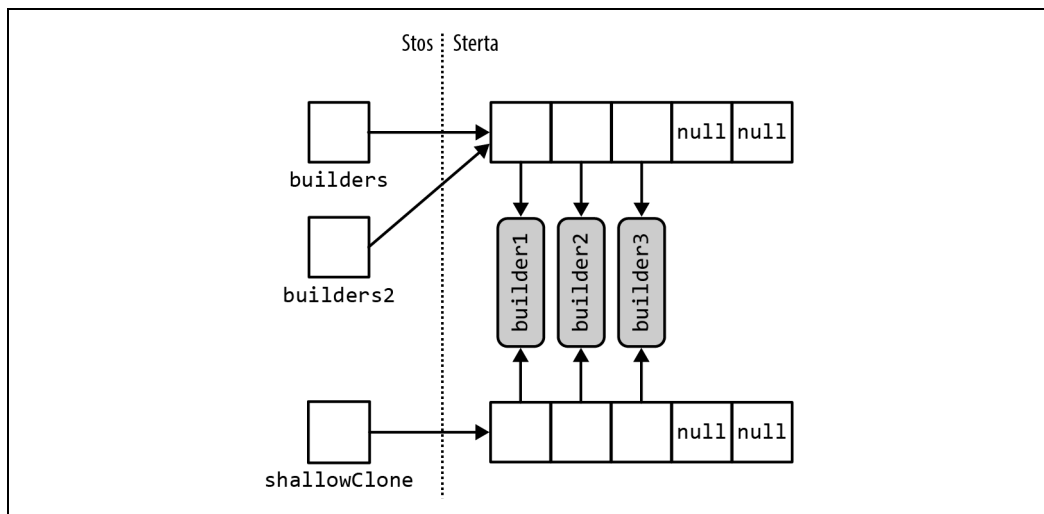
IStructuralEquatable se1 = a1;
Console.WriteLine (se1.Equals (a2,
    StructuralComparisons.StructuralEqualityComparer)); //prawda
```

Tablice można kopiować za pomocą metody `Clone`: `tablicaB = tablicaA.Clone()`. Jednak w ten sposób można wykonać kopię płytką, czyli obejmującą tylko reprezentowany przez tablicę obszar pamięci. Jeżeli struktura zawiera elementy typów wartościowych, to zostaną one skopiowane. Ale jeżeli tablica zawiera obiekty typów referencyjnych, to skopiowane zostaną tylko referencje (w efekcie powstaną dwie tablice, których składowe odnoszą się do tych samych obiektów). Na rysunku 7.3 przedstawiono skutek dodania do naszego przykładu poniższego kodu:

```
StringBuilder[] builders2 = builders;
StringBuilder[] shallowClone = (StringBuilder[]) builders.Clone();
```

Jeśli potrzebna jest kopia głęboka — z duplikatami wszystkich obiektów typu referencyjnego — należy przejrzeć tablicę za pomocą pętli i każdy element sklonować ręcznie. Te same zasady dotyczą także innych typów kolekcji .NET.

Choć klasa `Array` jest głównie przeznaczona do pracy z indeksatorami 32-bitowymi, to dzięki kilku metodom przyjmującym parametry `Int32` i `Int64` do pewnego stopnia obsługuje także indeksatory 64-bitowe (co teoretycznie zwiększa jej przestrzeń adresową do  $2^{64}$  elementów). W praktyce te przeciążone wersje metod są jednak bezużyteczne, ponieważ maksymalny rozmiar obiektu w systemie CLR — w tym także tablic — wynosi 2 GB (zarówno w środowisku 32-bitowym, jak i 64-bitowym).



Rysunek 7.3. Płytkie kopiowanie tablicy



Wiele z metod klasy `Array`, które mogą się wydawać metodami egzemplarzowymi, w istocie jest metodami statycznymi. Projektanci tej klasy podjęli dziwną decyzję pod tym względem i przez to programista szukający odpowiedniej metody musi sprawdzać zarówno wśród metod statycznych, jak i egzemplarzowych.

## Tworzenie i indeksowanie tablic

Najprostszym sposobem na utworzenie i zindeksowanie tablicy jest użycie konstrukcji języka C#:

```
int[] myArray = { 1, 2, 3 };
int first = myArray [0];
int last = myArray [myArray.Length - 1];
```

Ewentualnie tablicę można utworzyć dynamicznie za pomocą metody `Array.CreateInstance`, która pozwala na określenie typu elementów i liczby wymiarów oraz tworzenie tablic indeksowanych od innego numeru niż zero (przez określenie dolnej granicy). Tablice indeksowane w ten sposób są jednak niezgodne z CLS (ang. *Common Language Specification*).

Dostęp do elementów dynamicznie utworzonej tablicy zapewniają metody `GetValue` i `SetValue` (działają też na zwykłych tablicach):

```
// utworzenie tablicy łańcuchów o długości 2
Array a = Array.CreateInstance (typeof (string), 2);
a.SetValue ("Witaj,", 0); // → a[0] = "Witaj,";
a.SetValue ("świecie", 1); // → a[1] = "świecie";
string s = (string) a.GetValue (0); // → s = a[0];

// istnieje też możliwość rzutowania na tablice C#
string[] cSharpArray = (string[]) a;
string s2 = cSharpArray [0];
```

Tworzone dynamicznie tablice indeksowane od zera można rzutować na tablice C# takiego samego lub zgodnego typu (zgodnego wg zasad wariacji tablic). Jeśli np. klasa `Apple` jest pochodną klasy `Fruit`, to tablicę typu `Apple[]` można rzutować na typ `Fruit[]`. W tym momencie niektórzy mogą



się zastanawiać, dlaczego do unifikacji tablicy nie użyto typu `object[]`, tylko klasy `Array`. Przyczyną jest to, że typ `object[]` jest niezgodny z wielowymiarowymi tablicami typów wartościowych (a także indeksowanych nie od zera). Tablice typu `int[]` nie można rzutować na `object[]`. Dlatego potrzebujemy klasy `Array` w celu zapewnienia kompletnej unifikacji.

Metody `GetValue` i `SetValue` działają też na tablicach stworzonych przez kompilator oraz są przydatne przy pisaniu metod działających na tablicach dowolnego typu i rzędu. Dla tablic wielowymiarowych przyjmują *tablicę* indeksatorów:

```
public object GetValue (params int[] indices)
public void SetValue (object value, params int[] indices)
```

Poniższa metoda drukuje pierwszy element tablicy, niezależnie od tego, ile ma ona wymiarów:

```
void WriteFirstValue (Array a)
{
    Console.Write (a.Rank + "-dimensional; ");

    // Tablica indeksatorów automatycznie zainicjalizuje wszystkie wartości zerami, więc przekazanie
    // jej do metody GetValue lub SetValue spowoduje pobranie lub ustawienie pierwszego elementu tablicy.

    int[] indexers = new int[a.Rank];
    Console.WriteLine ("Pierwsza wartość to " + a.GetValue (indexers));
}

void Demo()
{
    int[] oneD = { 1, 2, 3 };
    int[, ] twoD = { {5,6}, {8,9} };

    WriteFirstValue (oneD); // jednowymiarowa; pierwsza wartość to 1
    WriteFirstValue (twoD); // dwuwymiarowa; pierwsza wartość to 5
}
```



Do pracy z tablicami nieznanego typu, ale znanego rzędu można zastosować prostsze i efektywniejsze rozwiązanie z wykorzystaniem typów generycznych:

```
void WriteFirstValue<T> (T[] array)
{
    Console.WriteLine (array[0]);
}
```

Metoda `SetValue` zgłasza wyjątek, jeżeli element jest nieodpowiedniego dla danej tablicy typu.

Gdy tworzona jest tablica, zarówno za pomocą składni języka, jak i metody `Array.CreateInstance`, jej elementy zostają automatycznie zainicjalizowane. Jeżeli elementy są typu referencyjnego, początkowo przypisywana jest im wartość `null`. Natomiast w przypadku elementów typów wartościowych zostaje wywołany konstruktor domyślny odpowiedniego typu (co w istocie oznacza wyzerowanie składowych). Ponadto klasa `Array` zawiera metodę `Clear`, za pomocą której w dowolnym momencie można wyzerować tablicę:

```
public static void Clear (Array array, int index, int length);
```

Metoda ta nie zmienia rozmiaru tablicy, co klóci się z jej typowym zastosowaniem (np. w `ICollection<T>.Clear`), polegającym na redukcji struktury do zera elementów.

## Przeglądanie zawartości tablicy

Zawartość tablicy można łatwo przejrzeć za pomocą instrukcji `foreach`:

```
int[] myArray = { 1, 2, 3};
foreach (int val in myArray)
    Console.WriteLine (val);
```

Ewentualnie można też użyć statycznej metody `Array.ForEach` o następującej definicji:

```
public static void ForEach<T> (T[] array, Action<T> action);
```

Wykorzystuje ona delegat `Action` o następującej sygnaturze:

```
public delegate void Action<T> (T obj);
```

Tak wygląda pierwszy przykład napisany z użyciem metody `Array.ForEach`:

```
Array.ForEach (new[] { 1, 2, 3 }, Console.WriteLine);
```

## Długość i liczba wymiarów tablicy

W klasie `Array` dostępne są następujące metody i własności do sprawdzania długości i liczby wymiarów tablicy:

```
public int GetLength (int dimension);
public long GetLongLength (int dimension);

public int Length { get; }
public long LongLength { get; }

public int GetLowerBound (int dimension);
public int GetUpperBound (int dimension);

public int Rank { get; } //zwraca liczbę wymiarów tablicy
```

Metody `GetLength` i `GetLongLength` zwracają długość wybranego wymiaru (0, jeśli tablica jest jednowymiarowa), natomiast metody `Length` i `LongLength` zwracają liczbę wszystkich elementów tablicy z wszystkich wymiarów łącznie.

Metody `GetLowerBound` i `GetUpperBound` służą do pracy z tablicami indeksowanymi od innego numeru niż zero. Metoda `GetUpperBound` zwraca wynik równy sumie wartości `GetLowerBound` i `GetLength` dla wybranego wymiaru.

## Przeszukiwanie tablic

Klasa `Array` udostępnia bogaty wybór metod do znajdowania elementów w tablicach jednowymiarowych:

### Metody `BinarySearch`

Do szybkiego szukania elementu w posortowanej tablicy.

### Metody `IndexOf/LastIndex`

Do szukania elementu w nieposortowanej tablicy.

### `Find`, `FindLast`, `FindIndex`, `FindLastIndex`, `FindAll`, `Exists`, `TrueForAll`

Do szukania w nieposortowanych tablicach elementów spełniających warunek `Predicate<T>`.

Żadna z metod przeszukujących nie zgłasza wyjątku, jeśli nie znajdzie szukanej wartości. Zamiast tego metody zwracające liczbę całkowitą zwracają -1 (dla tablic indeksowanych od zera), a metody zwracające typ generyczny zwracają domyślną wartość tego typu (np. 0 w przypadku `int` i `null` w przypadku `string`).

Metody przeszukiwania binarnego są szybkie, ale działają tylko na posortowanych tablicach i wymagają porównywania elementów pod względem *kolejności*, a nie *równości*. W efekcie metody te przyjmują obiekty typu `IComparer` i `IComparer<T>` określające definicję porządku w danym przypadku (zob. sekcję „Dołączanie protokołów równości i porządkowania”). Definicja ta musi być zgodna z komparatorem użytym pierwotnie do posortowania tablicy. Jeżeli nie zostanie przekazany żaden komparator, zostanie zastosowany domyślny algorytm porządkujący dla danego typu oparty na jego implementacji interfejsu `IComparable/IComparable<T>`.

Metody `IndexOf` i `LastIndexOf` służą do prostego przeglądania elementów tablicy i zwracają numer pierwszej (lub ostatniej) pozycji podanej wartości.

Metody działające na bazie predykatów przyjmują delegaty i wyrażenia lambda pozwalające stwierdzić, czy dana wartość „pasuje” do szukanej. Predykat to po prostu delegat przyjmujący obiekt i zwracający prawdę lub fałsz:

```
public delegate bool Predicate<T> (T object);
```

W poniższym przykładzie szukamy w tablicy łańcuchów imienia zawierającego literę a:

```
static void Main()
{
    string[] names = { "Robert", "Jacek", "Juliusz" };
    string match = Array.Find (names, ContainsA);
    Console.WriteLine (match); // Jacek
}
static bool ContainsA (string name) { return name.Contains ("a"); }
```

To jest ten sam kod, tylko skrócony dzięki użyciu metody anonimowej:

```
string[] names = { "Robert", "Jacek", "Juliusz" };
string match = Array.Find (names, delegate (string name)
    { return name.Contains ("a"); } );
```

Wyrażenie lambda pozwala jeszcze bardziej skrócić kod:

```
string[] names = { "Robert", "Jacek", "Juliusz" };
string match = Array.Find (names, n => n.Contains ("a")); // Jacek
```

Metoda `FindAll` zwraca tablicę wszystkich elementów spełniających warunek predykatu. Metoda ta, równoznaczna z `Enumerable.Where` z przestrzeni nazw `System.Linq`, zwraca tylko tablicę pasujących elementów, a nie obiekt typu `IEnumerable<T>`.

Metoda `Exists` zwraca prawdę, jeżeli którykolwiek element tablicy spełnia warunki danego predykatu, i jest równoważna metodzie `Any` z `System.Linq.Enumerable`.

Metoda `TrueForAll` zwraca prawdę, jeżeli wszystkie elementy spełniają warunek predykatu, i jest równoważna z `All` z `System.Linq.Enumerable`.

## Sortowanie

Klasa `Array` ma następujące wbudowane metody sortujące:

```
// do sortowania pojedynczej tablicy
```

```
public static void Sort<T> (T[] array);  
public static void Sort (Array array);
```

```
// do sortowania pary tablic
```

```
public static void Sort<TKey,TValue> (TKey[] keys, TValue[] items);  
public static void Sort (Array keys, Array items);
```

Każda z tych metod jest dodatkowo przeciążona i przyjmuje:

```
int index // indeks, od którego ma być rozpoczęte sortowanie  
int length // liczba elementów do posortowania  
IComparer<T> comparer // obiekt definiujący sposób porównywania  
Comparison<T> comparison // delegat definiujący sposób porównywania
```

Poniżej znajduje się najprostszy przykład użycia metody `Sort`:

```
int[] numbers = { 3, 2, 1 };  
Array.Sort (numbers); // teraz tablica ma wartości { 1, 2, 3 }
```

Metody przyjmujące pary tablic przekładają elementy w każdej z tablic łącznie i opierają decyzje porządkowe na pierwszej tablicy. W następnym przykładzie zarówno liczby, jak i odpowiadające im słowa są sortowane w porządku numerycznym:

```
int[] numbers = { 3, 2, 1 };  
string[] words = { "trzy", "dwa", "jeden" };  
Array.Sort (numbers, words);
```

```
// teraz kolejność elementów w tablicy numbers to { 1, 2, 3 }  
// teraz kolejność elementów w tablicy words to { "jeden", "dwa", "trzy" }
```

Metoda `Array.Sort` wymaga, aby znajdujące się w tablicy elementy implementowały interfejs `IComparable` (zob. podrozdział „Określanie kolejności” w rozdziale 6.). Oznacza to, że większość wbudowanych typów języka C# (takich jak użyte w poprzednim przykładzie liczby całkowite) można sortować. Jeżeli elementy nie mają określonego wewnętrznego porządku albo programista chce zmienić domyślną kolejność, konieczne jest przekazanie metodzie `Sort` własnego dostawcy `comparison` określającego względne ustawienie dwóch elementów. Można to zrobić na dwa sposoby:

- przy użyciu pomocniczego obiektu implementującego interfejs `IComparer/IComparer<T>` (zob. podrozdział „Dołączanie protokołów równości i porządkowania”);
- przy użyciu delegatu `Comparison`:

```
public delegate int Comparison<T> (T x, T y);
```

Delegat `Comparison` posługuje się taką samą semantyką jak metoda `IComparer<T>.CompareTo`: jeśli `x` jest przed `y`, zostaje zwrócona ujemna liczba całkowita; jeśli `x` jest za `y`, zostaje zwrócona dodatnia liczba całkowita; jeśli `x` i `y` zajmują to samo miejsce, zostaje zwrócone 0.

W poniższym przykładzie sortujemy tablicę liczb całkowitych w taki sposób, że najpierw ustawiane są liczby nieparzyste:

```
int[] numbers = { 1, 2, 3, 4, 5 };
Array.Sort (numbers, (x, y) => x % 2 == y % 2 ? 0 : x % 2 == 1 ? -1 : 1);
// zawartość tablicy numbers to { 1, 3, 5, 2, 4 }
```



Zamiast metody `Sort` można też użyć operatorów LINQ `OrderBy` i `ThenBy`. W odróżnieniu od metody `Array.Sort` operatory LINQ nie zmieniają pierwotnej tablicy, tylko wysyłają posortowane wyniki do nowej sekwencji `IEnumerable<T>`.

## Odwracanie kolejności elementów

Poniższe metody klasy `Array` odwracają kolejność wszystkich lub niektórych elementów tablicy:

```
public static void Reverse (Array array);
public static void Reverse (Array array, int index, int length);
```

## Kopiowanie

Klasa `Array` zawiera cztery metody do kopiowania płytkiego: `Clone`, `CopyTo`, `Copy` oraz `ConstrainedCopy`. Dwie pierwsze są metodami egzemplarzowymi, a pozostałe — statycznymi.

Metoda `Clone` zwraca nową tablicę (stanowiącą płytką kopię oryginału). Metody `CopyTo` i `Copy` kopiują ciągły podzbiór elementów tablicy. Kopiowanie wielowymiarowej prostokątnej tablicy wymaga przeprowadzenia mapowania wielowymiarowych indeksów na liniowe. Na przykład środkowy element (`position[1,1]`) w tablicy o wymiarach  $3 \times 3$  ma indeks 4, który wynika z obliczeń  $1 * 3 + 1$ . Zakresy źródłowy i docelowy mogą się bez problemu nakładać.

Metoda `ConstrainedCopy` wykonuje operację **atomową**, tzn. jeśli nie można skopiować wszystkich potrzebnych elementów (np. z powodu błędnego typu), to cała operacja jest wycofywana.

Ponadto klasa `Array` zawiera metodę `AsReadOnly`, która zwraca opakowanie uniemożliwiające ponowne przypisywanie wartości elementom.

## Konwertowanie i zmienianie rozmiarów tablic

Metoda `Array.ConvertAll` tworzy i zwraca nową tablicę typu `TOutput`. Wywołuje w tym celu przekazany jej delegat `Converter`. Jej definicja wygląda następująco:

```
public delegate TOutput Converter<TInput,TOutput> (TInput input)
```

Poniżej znajduje się przykład konwersji tablicy liczb zmiennoprzecinkowych na tablicę liczb całkowitych:

```
float[] reals = { 1.3f, 1.5f, 1.8f };
int[] wholes = Array.ConvertAll (reals, r => Convert.ToInt32 (r));
```

```
// zawartość tablicy wholes to { 1, 2, 2 }
```

Metoda `Resize` tworzy nową tablicę i kopiuje do niej wszystkie elementy, a następnie zwraca tę nową strukturę przez parametr referencyjny. Mimo to referencje do pierwotnej tablicy przechowywane w innych obiektach pozostają niezmienione.



Przestrzeń System.Linq zawiera dodatkowy asortyment metod do konwersji tablic. Wszystkie one zwracają obiekty typu `IEnumerable<T>`, który można przekonwertować na tablicę za pomocą metody `ToArray` z klasy `Enumerable`.

## Listy, kolejki, stosy i zbiory

Platforma zapewnia podstawowy zestaw konkretnych klas kolekcji implementujących opisane wcześniej interfejsy. W tym podrozdziale skupiamy się na kolekcjach **listowych** (których nie należy mylić ze strukturami słownikowymi opisanymi w podrozdziale „Słowniki”). Tak jak w przypadku opisanych wcześniej interfejsów, często mamy do wyboru zarówno generyczne, jak i niegeneryczne wersje każdego typu. Pod względem elastyczności i wydajności lepsze są klasy generyczne, przez co ich niegeneryczne odpowiedniki są w zasadzie niepotrzebne z wyjątkiem sytuacji, kiedy trzeba zapewnić zgodność ze starym kodem. Jest więc inaczej niż w przypadku interfejsów kolekcji, które bywają przydatne także w wersjach niegenerycznych.

Z opisanych w tym podrozdziale klas najczęściej używana jest generyczna klasa `List`.

### Klasy `List<T>` i `ArrayList`

Generyczna klasa `List` i niegeneryczna klasa `ArrayList` umożliwiają tworzenie tablic obiektów o dynamicznym rozmiarze i należą do najczęściej używanych klas kolekcji. Klasa `ArrayList` implementuje interfejs `ICollection`, podczas gdy `List<T>` implementuje zarówno `ICollection`, jak i `ICollection<T>` (jak również nową wersję tylko do odczytu o nazwie `ReadOnlyCollection<T>`). Inaczej niż w przypadku tablic, wszystkie interfejsy są zaimplementowane publicznie i zgodnie z oczekiwaniami metody takie jak `Add` czy `Remove` są dostępne.

Wewnątrz klasy `List<T>` i `ArrayList` utrzymują tablicę obiektów, którą w razie osiągnięcia maksimum pojemności zamieniają na większą. Dodawanie elementów na końcu to efektywna operacja (ponieważ zazwyczaj na końcu jest wolne miejsce), natomiast wstawianie elementów w środku pomiędzy innymi elementami może być powolne (ponieważ trzeba przesunąć wszystkie elementy znajdujące się za miejscem wstawiania). Jeśli chodzi o przeszukiwanie, to podobnie jak w przypadku tablic operacja jest efektywna, jeżeli przeprowadza się ją za pomocą metody `BinarySearch` na posortowanej liście. W innych sytuacjach wyszukiwanie jest mało wydajne, gdyż wymaga sprawdzenia każdego elementu po kolei.



Klasa `List<T>` jest do kilku razy szybsza od `ArrayList`, gdy `T` jest typem wartościowym, ponieważ `List<T>` nie wymaga pakowania ani odpakowywania elementów.

Klasy `List<T>` i `ArrayList` zawierają konstruktory przyjmujące istniejące kolekcje elementów — kopiują one wszystkie elementy z przekazanej kolekcji do nowej struktury:

```
public class List<T> : ICollection<T>, IReadOnlyCollection<T>
{
    public List ();
    public List (IEnumerable<T> collection);
    public List (int capacity);
}
```

```

// Add+Insert
public void Add (T item);
public void AddRange (IEnumerable<T> collection);
public void Insert (int index, T item);
public void InsertRange (int index, IEnumerable<T> collection);

// Remove
public bool Remove (T item);
public void RemoveAt (int index);
public void RemoveRange (int index, int count);
public int RemoveAll (Predicate<T> match);

// indeksowanie
public T this [int index] { get; set; }
public List<T> GetRange (int index, int count);
public Enumerator<T> GetEnumerator();

// eksportowanie, kopiowanie i konwertowanie
public T[] ToArray();
public void CopyTo (T[] array);
public void CopyTo (T[] array, int arrayIndex);
public void CopyTo (int index, T[] array, int arrayIndex, int count);
public ReadOnlyCollection<T> AsReadOnly();
public List<TOutput> ConvertAll<TOutput> (Converter <T,TOutput>
    converter);

// inne
public void Reverse(); // odwraca kolejność elementów listy
public int Capacity { get;set; } // wymusza rozszerzenie wewnętrznej tablicy
public void TrimExcess(); // obcina wewnętrzną tablicę do potrzebnego rozmiaru
public void Clear(); // usuwa wszystkie elementy, tak że Count=0
}
public delegate TOutput Converter <TInput, TOutput> (TInput input);

```

Oprócz tych składowych klasa `List<T>` zawiera egzemplarzowe wersje wszystkich metod przeszukiwania i sortowania klasy `Array`.

Poniżej znajduje się przykład demonstrujący sposób użycia własności i metod klasy `List`. Przykłady przeszukiwania i sortowania zamieściliśmy w podrozdziale „Klasa `Array`”.

```

List<string> words = new List<string>(); // nowa lista typu string

words.Add ("melon");
words.Add ("awokado");
words.AddRange (new[] { "banan", "pomarańcza" });
words.Insert (0, "liczi"); // wstawianie na początku
words.InsertRange (0, new[] { "pomelo", "nashi" }); // wstawianie na początku

words.Remove ("melon");
words.RemoveAt (3); // usunięcie czwartego elementu
words.RemoveRange (0, 2); // usunięcie dwóch pierwszych elementów

// usunięcie wszystkich łańcuchów zaczynających się literą 'n'
words.RemoveAll (s => s.StartsWith ("n"));

Console.WriteLine (words [0]); // pierwsze słowo
Console.WriteLine (words [words.Count - 1]); // ostatnie słowo
foreach (string s in words) Console.WriteLine (s); // wszystkie słowa
List<string> subset = words.GetRange (1, 2); // drugie i trzecie słowo

```

```
string[] wordsArray = words.ToArray(); // tworzy nową tablicę typizowaną

// kopiowanie pierwszych dwóch elementów na koniec istniejącej tablicy
string[] existing = new string [1000];
words.CopyTo (0, existing, 998, 2);

List<string> upperCastWords = words.ConvertAll (s => s.ToUpper());
List<int> lengths = words.ConvertAll (s => s.Length);
```

Niegeneryczna klasa `ArrayList` jest wykorzystywana głównie ze względu na zgodność z kodem napisanym dla platformy 1.x. Używający jej programista musi się posługiwać niezręcznymi rzutowaniami, jak pokazano w poniższym przykładzie:

```
ArrayList al = new ArrayList();
al.Add ("hello");
string first = (string) al [0];
string[] strArr = (string[]) al.ToArray (typeof (string));
```

Kompilator nie może zweryfikować takich operacji rzutowania, przez co np. poniższy kod przejdzie kompilację i spowoduje awarię podczas wykonywania programu:

```
int first = (int) al [0]; // wyjątek wykonawczy
```



Klasa `ArrayList` pod względem funkcjonalności przypomina `List<object>`. Obie przydają się, gdy potrzebna jest lista elementów różnego typu niemających wspólnego typu bazowego (innego niż `object`). Klasa `ArrayList` może być lepszym wyborem, gdy podczas pracy z listą używa się refleksji (rozdział 19.). Techniki refleksji łatwiej jest stosować w odniesieniu do niegenerycznego typu `ArrayList` niż do typu generycznego `List<object>`.

Jeśli do programu zaimportuje się przestrzeń nazw `System.Linq`, to listę `ArrayList` można przekonwertować na listę generyczną za pomocą metod `Cast` i `ToList`:

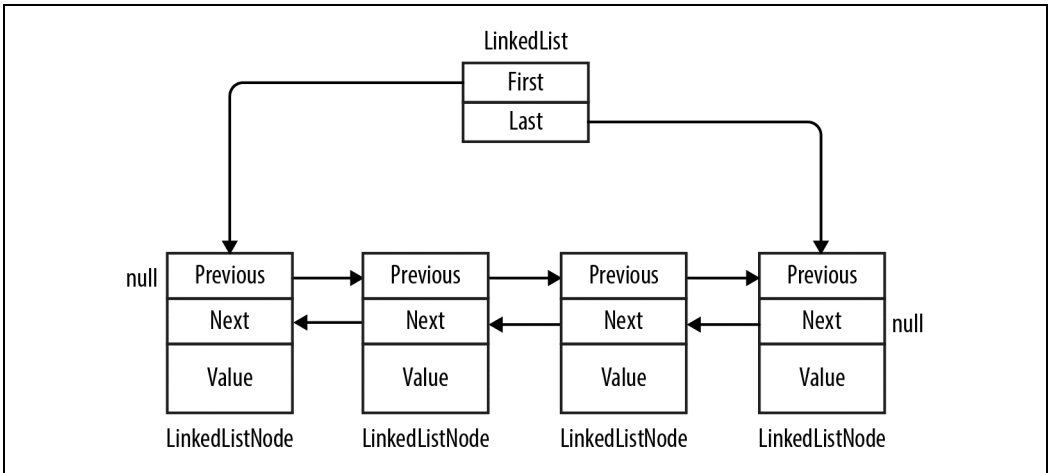
```
ArrayList al = new ArrayList();
al.AddRange (new[] { 1, 5, 9 } );
List<int> list = al.Cast<int>().ToList();
```

`Cast` i `ToList` to metody rozszerzeniowe klasy `System.Linq.Enumerable`.

## Klasa `LinkedList<T>`

Generyczna klasa `LinkedList<T>` stanowi implementację listy powiązanej dwustronnie (rysunek 7.4). Lista powiązana dwustronnie to łańcuch węzłów, w którym każdy węzeł zawiera referencję do poprzedniego i następnego węzła oraz do samego siebie. Największą zaletą tej struktury jest to, że każdy element można efektywnie wstawić w dowolnym miejscu listy, ponieważ wymaga to tylko utworzenia nowego węzła i zaktualizowania kilku referencji. Jednak znalezienie miejsca do wstawienia elementu może być czasochłonne, ponieważ nie istnieje żaden mechanizm bezpośredniego indeksowania takiej listy. Trzeba się przyjrzyć każdemu węzłowi i nie można korzystać z algorytmów wyszukiwania binarnego.





Rysunek 7.4. Klasa `LinkedList<T>`

Klasa `LinkedList<T>` implementuje interfejsy `IEnumerable<T>` i `ICollection<T>` (oraz ich niegeneryczne wersje). Nie implementuje natomiast interfejsu  `IList<T>`, ponieważ nie obsługuje dostępu do elementów za pomocą indeksów. Implementacja węzłów ma postać następującej klasy:

```
public sealed class LinkedListNode<T>
{
    public LinkedList<T> List { get; }
    public LinkedListNode<T> Next { get; }
    public LinkedListNode<T> Previous { get; }
    public T Value { get; set; }
}
```

Nowy węzeł można dodać na pozycji określonej względem innego węzła lub na początku albo na końcu listy. W klasie `LinkedList<T>` znajdują się następujące metody do wykonywania takich operacji:

```
public void AddFirst(LinkedListNode<T> node);
public LinkedListNode<T> AddFirst (T value);

public void AddLast (LinkedListNode<T> node);
public LinkedListNode<T> AddLast (T value);

public void AddAfter (LinkedListNode<T> node, LinkedListNode<T> newNode);
public LinkedListNode<T> AddAfter (LinkedListNode<T> node, T value);

public void AddBefore (LinkedListNode<T> node, LinkedListNode<T> newNode);
public LinkedListNode<T> AddBefore (LinkedListNode<T> node, T value);
```

Istnieje też podobny zestaw metod do usuwania elementów:

```
public void Clear();

public void RemoveFirst();
public void RemoveLast();

public bool Remove (T value);
public void Remove (LinkedListNode<T> node);
```

Klasa `LinkedList<T>` zawiera też kilka wewnętrznych pól do rejestrowania liczby przechowywanych elementów oraz referencji do pierwszego i ostatniego elementu. Pola te są dostępne w postaci następujących publicznych własności:

```
public int Count { get; } // szybkie
public LinkedListNode<T> First { get; } // szybkie
public LinkedListNode<T> Last { get; } // szybkie
```

Dodatkowo klasa `LinkedList<T>` udostępnia następujące metody wyszukiwania (każda wymaga, aby lista była wewnętrznie przeliczalna):

```
public bool Contains (T value);
public LinkedListNode<T> Find (T value);
public LinkedListNode<T> FindLast (T value);
```

Ponadto klasa `LinkedList<T>` obsługuje kopiowanie elementów do tablicy, aby można się było posługiwać indeksami, oraz umożliwia tworzenie enumeratorów, aby można się było posługiwać pętlą `foreach`:

```
public void CopyTo (T[] array, int index);
public Enumerator<T> GetEnumerator();
```

Oto przykład użycia klasy `LinkedList<string>`:

```
var tune = new LinkedList<string>();
tune.AddFirst ("do"); // do
tune.AddLast ("sol"); // do - sol

tune.AddAfter (tune.First, "re"); // do - re - sol
tune.AddAfter (tune.First.Next, "mi"); // do - re - mi - sol
tune.AddBefore (tune.Last, "fa"); // do - re - mi - fa - sol

tune.RemoveFirst(); // re - mi - fa - sol
tune.RemoveLast(); // re - mi - fa

LinkedListNode<string> miNode = tune.Find ("mi");
tune.Remove (miNode); // re - fa
tune.AddFirst (miNode); // mi - re - fa

foreach (string s in tune) Console.WriteLine (s);
```

## Klasy `Queue<T>` i `Queue`

Klasy `Queue<T>` i `Queue` to struktury danych typu FIFO (ang. *first-in, first-out* — pierwszy na wejściu, pierwszy na wyjściu) udostępniające m.in. metody `Enqueue` (do dodawania elementów na koniec kolejki) i `Dequeue` (do pobierania i usuwania elementów z początku kolejki). Ponadto dostępna jest metoda `Peek` zwracająca element z początku struktury bez jego usuwania oraz własność `Count` umożliwiająca sprawdzenie przed zdejmowaniem elementów, czy jakies w ogóle istnieją.

Choć kolejki są przeliczalne, nie implementują interfejsów `IList<T>` ani `IList`, ponieważ do składowych nie można odwoływać się bezpośrednio za pomocą indeksów. W razie potrzeby dostępna jest jednak metoda `ToArray` służąca do kopiowania elementów do tablicy zapewniającej swobodny dostęp do swojej zawartości:

```
public class Queue<T> : IEnumerable<T>, ICollection, IEnumerable
{
```

```

public Queue();
public Queue (IEnumerable<T> collection); // kopiuje istniejące elementy
public Queue (int capacity); // pozwala ograniczyć automatyczne zmiany rozmiaru
public void Clear();
public bool Contains (T item);
public void CopyTo (T[] array, int arrayIndex);
public int Count { get; }
public T Dequeue();
public void Enqueue (T item);
public Enumerator<T> GetEnumerator(); // aby można było korzystać z pętli foreach
public T Peek();
public T[] ToArray();
public void TrimExcess();
}

```

Poniżej przedstawiamy przykład użycia klasy `Queue<int>`:

```

var q = new Queue<int>();
q.Enqueue (10);
q.Enqueue (20);
int[] data = q.ToArray(); // eksport zawartości do tablicy
Console.WriteLine (q.Count); // "2"
Console.WriteLine (q.Peek()); // "10"
Console.WriteLine (q.Dequeue()); // "10"
Console.WriteLine (q.Dequeue()); // "20"
Console.WriteLine (q.Dequeue()); // spowoduje zgłoszenie wyjątku (pusta kolejka)

```

Wewnętrzna implementacja kolejek opiera się na tablicy, której rozmiar jest zmieniany zgodnie z zapotrzebowaniem — podobną implementację ma generyczna klasa `List`. Kolejka utrzymuje indeksy wskazujące bezpośrednio na pierwszy i ostatni element, dzięki czemu operacje dokładania i wyjmowania elementów są bardzo szybkie (z wyjątkiem sytuacji, gdy spowodują zmianę rozmiaru struktury).

## Klasy `Stack<T>` i `Stack`

Klasy `Stack<T>` i `Stack` to struktury danych typu LIFO (ang. *last-in, first-out* — ostatni na wejściu, pierwszy na wyjściu) udostępniające m.in. metody `Push` (do dodawania elementu na wierzchu stosu) i `Pop` (do pobierania i usuwania elementów z wierzchu stosu). Dostępne są też niedestrukcyjna metoda `Peek`, jak również własność `Count` i metoda `ToArray`, umożliwiające eksport danych do struktury pozwalającej na ich swobodne przeglądanie:

```

public class Stack<T> : IEnumerable<T>, ICollection, IEnumerable
{
    public Stack();
    public Stack (IEnumerable<T> collection); // kopiuje istniejące elementy
    public Stack (int capacity); // pozwala ograniczyć automatyczne zmiany rozmiaru
    public void Clear();
    public bool Contains (T item);
    public void CopyTo (T[] array, int arrayIndex);
    public int Count { get; }
    public Enumerator<T> GetEnumerator(); // aby można było korzystać z pętli foreach
    public T Peek();
    public T Pop();
    public void Push (T item);
    public T[] ToArray();
    public void TrimExcess();
}

```

Poniżej znajduje się przykład użycia klasy `Stack<int>`:

```
var s = new Stack<int>();
s.Push (1); // s = 1
s.Push (2); // s = 1,2
s.Push (3); // s = 1,2,3
Console.WriteLine (s.Count); // drukuje 3
Console.WriteLine (s.Peek()); // drukuje 3, s = 1,2,3
Console.WriteLine (s.Pop()); // drukuje 3, s = 1,2
Console.WriteLine (s.Pop()); // drukuje 2, s = 1
Console.WriteLine (s.Pop()); // drukuje 1, s = <empty>
Console.WriteLine (s.Pop()); // powoduje wyjątek
```

Wewnętrzna implementacja stosów bazuje na tablicy, której rozmiar zmienia się w razie potrzeby, podobnie jak w przypadku klas `Queue<T>` i `List<T>`.

## Klasa `BitArray`

Klasa `BitArray` to kolekcja wartości typu `bool` z możliwością dynamicznej zmiany rozmiaru. Pozwala efektywniej wykorzystać pamięć niż zwykła tablica lub struktura `List` wartości typu `bool`, ponieważ do przechowywania każdego elementu potrzebuje tylko jednego bitu, podczas gdy normalnie wartość typu `bool` zajmuje jeden bajt.

Indeksator struktury `BitArray` odczytuje i zapisuje pojedyncze bity:

```
var bits = new BitArray(2);
bits[1] = true;
```

Dostępne są cztery metody bitowe: `And`, `Or`, `Xor` oraz `Not`. Wszystkie oprócz ostatniej przyjmują obiekty typu `BitArray`:

```
bits.Xor (bits); // bitowe lub wykluczające na samym sobie
Console.WriteLine (bits[1]); // fałsz
```

## Klasy `HashSet<T>` i `SortedSet<T>`

Klasy `HashSet<T>` i `SortedSet<T>` to generyczne kolekcje, które wprowadzono odpowiednio w .NET Framework 3.5 i 4.0. Obie wyróżniają się następującymi cechami:

- Metody `Contains` charakteryzują się dużą szybkością działania dzięki posługiwaniu się algorytmem wyszukiwania wykorzystującym wartości skrótu.
- Nie przechowują duplikatów i niepostrzeżenie ignorują żądania dodania elementów takich samych jak elementy istniejące.
- Nie ma możliwości odwołania się do elementu po jego pozycji.

Klasa `SortedSet<T>` przechowuje elementy w określonym porządku, a `HashSet<T>` nie przechowuje.



Wspólne cechy tych dwóch klas są dziełem implementacji interfejsu `ISet<T>`.

Z powodów historycznych klasa `HashSet<T>` znajduje się w zestawie `System.Core.dll`, a klasa `SortedSet<T>` i interfejs `ISet<T>` w zestawie `System.dll`.

Implementację klasy `HashSet<T>` stanowi tablica skrótów przechowująca tylko klucze. Natomiast struktura `SortedSet<T>` jest zaimplementowana jako drzewo czerwono-czarne.

Obie kolekcje implementują interfejs `ICollection<T>` i udostępniają metody, jakich można się spodziewać, a więc np.: `Contains`, `Add` i `Remove`. Dodatkowo istnieje też wykorzystująca predykat metoda `RemoveWhere` do usuwania elementów.

Poniżej przedstawiamy przykład utworzenia zbioru `HashSet<char>` z istniejącej kolekcji, sprawdzenia, czy wybrany element jest dostępny w zbiorze, oraz jej przejrzenia (zwróć uwagę na usunięcie duplikatów):

```
var letters = new HashSet<char> ("gdyby kózka nie skakała");

Console.WriteLine (letters.Contains ('g')); //prawda
Console.WriteLine (letters.Contains ('j')); //fałsz

foreach (char c in letters) Console.Write (c); //gdymb kózkaniesl
```

(Przekazanie obiektu typu `string` do konstruktora klasy `HashSet<char>` nie spowodowało błędu, ponieważ klasa `string` implementuje interfejs `IEnumerable<char>`).

Najciekawsze metody to te operujące na zbiorach. Poniższe metody mają charakter **destrukcyjny** w tym sensie, że modyfikują zbiór:

```
public void UnionWith (IEnumerable<T> other); //dodaje
public void IntersectWith (IEnumerable<T> other); //usuwa
public void ExceptWith (IEnumerable<T> other); //usuwa
public void SymmetricExceptWith (IEnumerable<T> other); //usuwa
```

Natomiast te metody tylko sprawdzają coś w zbiorze, więc nie mają charakteru destrukcyjnego:

```
public bool IsSubsetOf (IEnumerable<T> other);
public bool IsProperSubsetOf (IEnumerable<T> other);
public bool IsSupersetOf (IEnumerable<T> other);
public bool IsProperSupersetOf (IEnumerable<T> other);
public bool Overlaps (IEnumerable<T> other);
public bool SetEquals (IEnumerable<T> other);
```

Metoda `UnionWith` dodaje wszystkie elementy z drugiego zbioru do pierwszego (wykluczając duplikaty). Metoda `IntersectWith` usuwa elementy, których nie ma w obu zbiorach. Z naszego zbioru znaków możemy np. wydobyc wszystkie samogłoski w następujący sposób:

```
var letters = new HashSet<char> ("gdyby kózka nie skakała");
letters.IntersectWith ("aeioou");
foreach (char c in letters) Console.Write (c); //yóai
```

Metoda `ExceptWith` usuwa określone elementy ze zbioru źródłowego. W poniższym przykładzie pozabawiamy nasz zbiór wszystkich samogłosek:

```
var letters = new HashSet<char> ("gdyby kózka nie skakała");
letters.ExceptWith ("aeioou");
foreach (char c in letters) Console.Write (c); //gdb kznsl
```

Metoda `SymmetricExceptWith` usuwa wszystkie elementy oprócz tych, które występują tylko w jednym lub drugim zbiorze:

```
var letters = new HashSet<char> ("gdyby kózka nie skakała");
letters.SymmetricExceptWith ("toby nogi nie zżamała");
foreach (char c in letters) Console.Write (c); // dkóstrom
```

Ponieważ klasy `HashSet<T>` i `SortedSet<T>` implementują interfejs `IEnumerable<T>`, ich metody przyjmują jako argumenty inne typy zbiorów (lub kolekcji).

Klasa `SortedSet<T>` ma wszystkie składowe klasy `HashSet<T>` oraz dodatkowo:

```
public virtual SortedSet<T> GetViewBetween (T lowerValue, T upperValue)
public IEnumerable<T> Reverse()
public T Min { get; }
public T Max { get; }
```

Ponadto konstruktor klasy `SortedSet<T>` opcjonalnie przyjmuje obiekt typu `IComparer<T>` (a nie *komparator równości*).

Oto jeden z efektów załadowania naszego przykładowego ciągu liter do zbioru `SortedSet<char>`:

```
var letters = new SortedSet<char> ("gdyby kózka nie skakała");
foreach (char c in letters) Console.Write (c); // abdegiknsyzół
```

Teraz możemy pobrać wszystkie litery z przedziału od f do j:

```
foreach (char c in letters.GetViewBetween ('f', 'j'))
Console.Write (c); // gi
```

## Słowniki

Słownik jest kolekcją, w której każdy przechowywany element stanowi parę klucz – wartość. Tego typu struktur danych najczęściej używa się do przeszukiwania i sortowania list.

Platforma .NET zawiera definicję standardowego protokołu słownikowego w postaci interfejsów `IDictionary` i `IDictionary<TKey, TValue>` oraz definicji kilku klas słownikowych ogólnego przeznaczenia. Różnią się one między sobą następującymi cechami:

- niektóre sortują elementy, a inne nie;
- niektóre pozwalają na dostęp do elementów wg pozycji (indeksu) i klucza, a inne nie;
- niektóre są generyczne, a inne nie;
- niektóre pozwalają na szybkie pobieranie elementów wg klucza dużych słowników, a inne nie.

W tabeli 7.1 przedstawiono zestawienie klas słownikowych i dzielących je różnic. Wartości czasowe są wyrażone w milisekundach dla 50 000 operacji na słowniku z kluczami i są wartościami całkowitoliczbowymi wykonywanymi w komputerze PC z procesorem 1,5 GHz. (Różnice między generycznymi i niegenerycznymi wersjami wynikają z konieczności stosowania pakowania w jednym przypadku i występują tylko w odniesieniu do typów wartościowych).

Tabela 7.1. Klasy słownikowe

Typ	Struktura wewnętrzna	Indeksowanie	Narzut pamięciowy (średnia liczba bajtów na element)	Szybkość — losowe wstawianie	Szybkość — wstawianie sekwencyjne	Szybkość — pobieranie elementów wg klucza
<b>Niesortowane</b>						
Dictionary <K,V>	Tablica skrótów	Nie	22	30	30	20
Hashtable	Tablica skrótów	Nie	38	50	50	30
ListDictionary	Lista powiązana	Nie	36	50 000	50 000	50 000
OrderedDictionary	Tablica skrótów + tablica	Tak	59	70	70	40
<b>Sortowane</b>						
SortedDictionary <K,V>	Drzewo czerwono-czarne	Nie	20	130	100	120
SortedList <K,V>	Dwie tablice	Tak	2	3300	30	40
SortedList	Dwie tablice	Tak	27	4500	100	180

W notacji wielkiego O czasy pobierania elementów wg klucza wynoszą:

- $O(1)$  dla Hashtable, Dictionary oraz OrderedDictionary;
- $O(\log n)$  dla SortedDictionary i SortedList;
- $O(n)$  dla ListDictionary (i typów niesłownikowych, takich jak List<T>).

Parametr  $n$  reprezentuje liczbę elementów w kolekcji.

## Interfejs IDictionary<TKey,TValue>

Interfejs IDictionary<TKey,TValue> to standardowy protokół dla wszystkich kolekcji przechowujących pary klucz – wartość. Stanowi on rozszerzenie interfejsu ICollection<T>, do którego dodaje metody i własności umożliwiające dostęp do elementów na podstawie kluczy dowolnego typu:

```
public interface IDictionary <TKey, TValue> :
    ICollection <KeyValuePair <TKey, TValue>>, IEnumerable
{
    bool ContainsKey (TKey key);
    bool TryGetValue (TKey key, out TValue value);
    void Add (TKey key, TValue value);
    bool Remove (TKey key);
}
```

```
TValue this [TKey key] { get; set; } // główny indeksator — wg klucza
ICollection<TKey> Keys { get; } // zwraca tylko klucze
ICollection<TValue> Values { get; } // zwraca tylko wartości
}
```



W .NET Framework 4.5 dodano interfejs o nazwie `IReadOnlyDictionary<TKey, TValue>` definiujący podzbiór składowych słownika tylko do odczytu. Odpowiada on typowi `IMapView<K,V>` Windows Runtime i został wprowadzony głównie jako jego odpowiednik.

Aby dodać element do słownika, należy wywołać metodę `Add` albo użyć metody dostępowej indeksu — ta druga dodaje element do słownika, jeśli nie ma w nim jeszcze takiego klucza (a jeśli jest, to dokonuje modyfikacji elementu). W żadnej implementacji słownika nie może się powtórzyć ani jeden klucz, więc dwukrotne wywołanie metody `Add` z takim samym kluczem powoduje wyjątek.

Aby pobrać element ze słownika, należy się posłużyć indeksatorem albo metodą `TryGetValue`. Jeżeli dany klucz nie istnieje, indeksator zgłasza wyjątek, podczas gdy metoda `TryGetValue` zwraca fałsz. Istnienie elementu w strukturze można sprawdzić za pomocą metody `ContainsKey`. Jeśli jednak później dany element trzeba pobrać, będzie to oznaczało konieczność wykonania dwóch takich samych operacji wyszukiwania.

Operacja enumeracji struktury typu `IDictionary<TKey, TValue>` zwraca sekwencję struktur `KeyValuePair`:

```
public struct KeyValuePair<TKey, TValue>
{
    public TKey Key { get; }
    public TValue Value { get; }
}
```

Istnieje możliwość przeglądania tylko kluczy lub tylko wartości przy użyciu własności `Keys` i `Values`.

W następnej sekcji przedstawiamy przykład użycia tego interfejsu w połączeniu z generyczną klasą `Dictionary`.

## Interfejs `IDictionary`

Niegeneryczny interfejs `IDictionary` jest bardzo podobny do generycznego `IDictionary<TKey, TValue>`, ale różni się od niego w dwóch ważnych aspektach. Należy wiedzieć o tych różnicach, ponieważ interfejs `IDictionary` występuje w starym kodzie (włącznie z samą platformą .NET Framework):

- Próba pobrania przez indeksator nieistniejącego klucza kończy się zwróceniem wartości `null` (a nie wyjątkiem).
- Do sprawdzania, czy dany element znajduje się w strukturze, służy metoda `Contains`, a nie `ContainsKey`.

Efektom przeliczania niegenerycznego `IDictionary` jest zwrot sekwencji struktur `DictionaryEntry`:

```
public struct DictionaryEntry
{
    public object Key { get; set; }
    public object Value { get; set; }
}
```



## Klasy Dictionary<TKey,TValue> i Hashtable

Generyczna klasa Dictionary należy do najczęściej używanych kolekcji (obok List<T>). Do przechowywania kluczy i wartości wykorzystuje tablicę skrótów (ang. *hashtable*), dzięki czemu jest szybka i efektywna.



Niegeneryczna wersja klasy Dictionary<TKey,TValue> nazywa się Hashtable. Nie istnieje niegeneryczna klasa o nazwie Dictionary. Dlatego pisząc Dictionary, mamy na myśli generyczną klasę Dictionary<TKey,TValue>.

Klasa Dictionary implementuje zarówno generyczny, jak i niegeneryczny interfejs IDictionary, przy czym generyczna wersja jest udostępniona publicznie. W istocie Dictionary to „podręcznikowa” implementacja generycznego interfejsu IDictionary.

Oto przykład jej użycia:

```
var d = new Dictionary<string, int>();

d.Add("Jeden", 1);
d["Dwa"] = 2; // dodaje element do słownika, ponieważ nie ma w nim jeszcze "dwa"
d["Dwa"] = 22; // modyfikuje słownik, ponieważ "dwa" już w nim jest
d["Trzy"] = 3;

Console.WriteLine (d["Dwa"]); // drukuje "22"
Console.WriteLine (d.ContainsKey ("Jeden")); // prawda (szybka operacja)
Console.WriteLine (d.ContainsValue (3)); // prawda (wolna operacja)
int val = 0;
if (d.TryGetValue ("jeden", out val))
    Console.WriteLine ("Brak"); // "Brak" (wielkość liter ma znaczenie)

// trzy różne sposoby przeglądania słownika:

foreach (KeyValuePair<string, int> kv in d) // Jeden; 1
    Console.WriteLine (kv.Key + "; " + kv.Value); // Dwa; 22
    // Trzy; 3

foreach (string s in d.Keys) Console.Write (s); // JedenDwaTrzy
Console.WriteLine();
foreach (int i in d.Values) Console.Write (i); // 1223
```

Wewnętrzna tablica skrótów konwertuje klucz każdego elementu na całkowitoliczbową wartość skrótu, która jest pseudoniepowtarzalna, a następnie za pomocą specjalnego algorytmu konwertuje ten skrót na klucz mieszający. Wewnętrznie za pomocą tego klucza wybierany jest „kubek”, do którego należy dany wpis. Jeśli kubek zawiera więcej niż jedną wartość, to jest przeszukiwany algorytmem liniowym. Dobra funkcja mieszająca nie dąży do zwracania wyłącznie niepowtarzalnych skrótów (co w większości przypadków byłoby niemożliwe), tylko do równomiernego rozmieszczenia skrótów w 32-bitowej przestrzeni całkowitoliczbowej. W ten sposób unika się groźby powstania bardzo małej liczby bardzo dużych (i nieefektywnych) kubków.

Klucze w słowniku mogą być każdego typu, pod warunkiem że można je porównywać i tworzyć z nich wartości skrótu. Domyślnie klucze porównuje się za pomocą metody klucza `object.Equals`, a pseudoniepowtarzalna wartość skrótu jest obliczana przez metodę klucza `GetHashCode`. Można to zmienić przez przesłonięcie tych metod lub dostarczenie obiektu implementującego interfejs

IEqualityComparer podczas tworzenia słownika. Możliwość tę wykorzystuje się często, gdy stosowane są klucze łańcuchowe w celu użycia komparatora nierozróżniającego wielkości liter:

```
var d = new Dictionary<string, int> (StringComparer.OrdinalIgnoreCase);
```

Szerzej na ten temat piszemy jeszcze w podrozdziale „Dołączanie protokołów równości i porządkowania”.

Tak jak w przypadku wszystkich innych typów kolekcji, wydajność słownika można nieco podnieść przez określenie w konstruktorze oczekiwanego rozmiaru struktury, co pozwala ograniczyć potrzebę wykonywania operacji zmiany rozmiaru (lub w ogóle jej uniknąć).

Niegeneryczna wersja nazywa się `Hashtable` i pod względem funkcjonalnym jest podobna do generycznej, tylko udostępnia niegeneryczny interfejs `IDictionary`, o którym była mowa wcześniej.

Wadą klas `Dictionary` i `Hashtable` jest to, że nie sortują przechowywanych elementów. Nie zachowują nawet kolejności dodawania wpisów. Jak wszystkie słowniki, struktury te nie pozwalają na zapisanie dwóch takich samych kluczy.



Gdy w .NET Framework 2.0 wprowadzono generyczne kolekcje, programiści CLR postanowili nadać im nazwy zgodne z tym, co reprezentują (`Dictionary`, `List`), a nie odnoszące się do ich wewnętrznej implementacji (`Hashtable`, `ArrayList`). Choć jest to dobra decyzja, dzięki której w przyszłości będzie można bez przeszkód zmieniać podstawową implementację, wybrane nazwy nie pozwalają się zorientować, jaką *wydajnością* (a informacja ta jest często najważniejszym kryterium wyboru) charakteryzują się poszczególne kolekcje.

## Klasa `OrderedDictionary`

`OrderedDictionary` to niegeneryczny słownik przechowujący elementy w kolejności ich dodawania. W tej strukturze elementy dostępne są zarówno wg indeksu, jak i wg klucza.



`OrderedDictionary` nie jest słownikiem *posortowanym*.

Klasa `OrderedDictionary` jest kombinacją klas `Hashtable` i `ArrayList`, tzn. zawiera całą funkcjonalność pierwszej i kilka dodatkowych funkcji, takich jak `RemoveAt` i indeksator całkowitoliczbowy. Ponadto struktura ta udostępnia własności `Keys` i `Values` zwracające elementy w pierwotnym porządku.

Klasę tę wprowadzono w .NET 2.0, ale wciąż nie wiadomo dlaczego nie ma jej wersji generycznej.

## Klasy `ListDictionary` i `HybridDictionary`

Klasa `ListDictionary` przechowuje dane w liście powiązanej jednostronnie. Nie sortuje elementów, ale zapisuje je w kolejności dodawania. Struktura ta działa bardzo wolno, gdy jest duża. Jedyne sens jej istnienia to wysoka wydajność dla bardzo małych list (zawierających mniej niż dziesięć elementów).

Klasa `HybridDictionary` to `ListDictionary` automatycznie konwertująca się na `Hashtable` po osiągnięciu określonego rozmiaru w celu uniknięcia problemów wydajnościowych. Chodzi o to, by jak

najoszczędniej operować pamięcią, gdy słownik jest mały, oraz by zachować dobrą wydajność, kiedy się powiększy. Biorąc jednak pod uwagę czas potrzebny na przeprowadzenie konwersji — oraz to, że klasa `Dictionary` też nie jest zbyt ciężka ani wolna — nikomu nie zaszkodzi użycie słownika `Dictionary` w zamian.

Obie te klasy występują tylko w wersji niegenerycznej.

## Słowniki sortowane

Platforma .NET zawiera dwie klasy słownikowe o wewnętrznej strukturze sprawiającej, że ich zawartość jest zawsze posortowana wg klucza:

- `SortedDictionary<TKey, TValue>`,
- `SortedList<TKey, TValue>`<sup>1</sup>.

(W tej sekcji skracamy zapis `<TKey, TValue>` do postaci `<, >`).

Implementacja klasy `SortedDictionary<, >` bazuje na drzewie czerwono-czarnym — strukturze danych, która charakteryzuje się taką samą wydajnością w każdej operacji wstawiania i pobierania elementu.

Implementację klasy `SortedList<, >` stanowi uporządkowana para tablic, co zapewnia szybkie pobieranie danych (za pomocą binarnego algorytmu wyszukiwania), ale mało efektywne wstawianie elementów (ponieważ trzeba przesunąć istniejące elementy, aby zrobić miejsce dla dodawanego).

Klasa `SortedDictionary<, >` jest znacznie szybsza od `SortedList<, >` we wstawianiu elementów w losowej kolejności (zwłaszcza gdy struktura danych jest duża). Za to `SortedList<, >` dodatkowo umożliwia odnośnienie się do elementów wg indeksu lub wg klucza. Mając posortowaną listę, można bezpośrednio przejść do *n*-tego elementu sekwencji (posługując się indeksatorem lub własnościami `Keys` i `Values`). Aby zrobić to samo ze strukturą `SortedDictionary<, >`, należy ręcznie przejrzeć *n* elementów. (Ewentualnie można napisać klasę łączącą posortowany słownik z listą).

Żadna z tych trzech kolekcji nie pozwala na duplikowanie kluczy (tak jest we wszystkich słownikach).

Poniżej przedstawiamy przykład, w którym za pomocą refleksji ładujemy wszystkie metody zdefiniowane w klasie `System.Object` do posortowanej listy, w której klucze stanowią nazwy metod, a następnie przeglądamy te klucze i ich wartości:

```
// MethodInfo znajduje się w przestrzeni nazw System.Reflection

var sorted = new SortedList<string, MethodInfo>();

foreach (MethodInfo m in typeof(object).GetMethods())
    sorted [m.Name] = m;

foreach (string name in sorted.Keys)
    Console.WriteLine (name);

foreach (MethodInfo m in sorted.Values)
    Console.WriteLine (m.Name + " zwraca obiekt typu " + m.ReturnType);
```

---

<sup>1</sup> stnieje też identyczna pod względem funkcjonalności niegeneryczna wersja o nazwie `SortedList`.

Oto wynik pierwszego przeglądu:

```
Equals  
GetHashCode  
GetType  
ReferenceEquals  
ToString
```

A to jest wynik drugiej pętli:

```
Equals zwraca obiekt typu System.Boolean  
GetHashCode zwraca obiekt typu System.Int32  
GetType zwraca obiekt typu System.Type  
ReferenceEquals zwraca obiekt typu System.Boolean  
ToString zwraca obiekt typu System.String
```

Zauważ, że zawartość wstawiliśmy do słownika za pomocą indeksatora. Gdybyśmy zamiast niego użyli metody `Add`, zgłosiłaby wyjątek, ponieważ klasa `object`, której dotyczy stosowana przez nas refleksja, przeciąża metodę `Equals`, a do słownika nie można dodać dwa razy takiego samego klucza. Dzięki użyciu indeksatora drugi wpis po prostu zastąpi poprzedni i nie wystąpi żaden błąd.



Jeśli trzeba zapisać kilka elementów pod takim samym kluczem, to elementy wartości można zdefiniować jako listy:

```
SortedList <string, List<MethodInfo>>
```

Wracając do naszego przykładu: poniższa instrukcja pobiera obiekt `MethodInfo`, którego klucz to `GetHashCode`. Dokładnie tak samo pracowalibyśmy ze zwykłym słownikiem:

```
Console.WriteLine (sorted ["GetHashCode"]); // Int32 GetHashCode()
```

Wszystko, co do tej pory zrobiliśmy, dałoby się wykonać także z klasą `SortedDictionary<,>`. Ale poniższe dwa wiersze kodu zadziałają tylko z listą posortowaną, ponieważ pobieramy w nich ostatni klucz i ostatnią wartość:

```
Console.WriteLine (sorted.Keys [sorted.Count - 1]); // ToString  
Console.WriteLine (sorted.Values[sorted.Count - 1].IsVirtual); // prawda
```

## Kolekcje i pośredniki z możliwością dostosowywania

Opisane w poprzednich podrozdziałach klasy kolekcji są wygodne w użyciu, ponieważ można bezpośrednio tworzyć ich egzemplarze. Niestety, nie ma możliwości kontrolowania sposobu dodawania ani usuwania elementów. Możliwość taka jest jednak potrzebna w aplikacjach, w których używane są ściśle typizowane kolekcje, np.:

- aby uruchomić zdarzenie w reakcji na dodanie lub usunięcie elementu;
- aby zaktualizować własności z powodu dodania lub usunięcia elementu;
- aby wykryć „niedozwoloną” operację dodawania lub usuwania i zgłosić wyjątek (gdy np. operacja łamie jakieś zasady biznesowe).

Platforma .NET Framework zawiera klasy kolekcji przeznaczone do użytku właśnie w takich sytuacjach. Znajdują się one w przestrzeni nazw `System.Collections.ObjectModel` i tak naprawdę są

pośrednikami lub opakowaniami implementującymi interfejs `IList<T>` lub `IDictionary<,>` przez przekazywanie metod do podstawowej kolekcji. Każda operacja `Add`, `Remove` i `Clear` zostaje przekierowana przez metodę wirtualną, która po przesłonięciu stanowi rodzaj „bramy”.

Klasy kolekcji z możliwością dostosowywania są powszechnie wykorzystywane do tworzenia publicznie dostępnych kolekcji, takich jak np. kolekcja kontrolek publicznych klasy `System.Windows.Form`.

## Klasy `Collection<T>` i `CollectionBase`

Klasa `Collection<T>` to modyfikowalne opakowanie klasy `List<T>`.

Implementuje ona interfejsy `IList<T>` i `IList` oraz definiuje cztery dodatkowe metody wirtualne i chronioną własność:

```
public class Collection<T> :
    IList<T>, ICollection<T>, IEnumerable<T>, IList, ICollection, IEnumerable
{
    //...
    protected virtual void ClearItems();
    protected virtual void InsertItem (int index, T item);
    protected virtual void RemoveItem (int index);
    protected virtual void SetItem (int index, T item);

    protected IList<T> Items { get; }
}
```

Metody wirtualne stanowią bramę, przez którą można „się podpiąć”, aby zmienić lub rozszerzyć normalną funkcjonalność listy. Chroniona własność `Items` umożliwia implementatorowi uzyskanie bezpośredniego dostępu do „wewnętrznej listy” — w ten sposób można dokonywać zmian wewnątrz bez uruchamiania metod wirtualnych.

Metody wirtualne nie muszą być przesłonięte. Można je zostawić nietknięte, aż nadejdzie potrzeba modyfikacji domyślnego działania listy. Poniżej znajduje się schematyczny przykład ilustrujący typowy sposób użycia klasy `Collection<T>`:

```
public class Animal
{
    public string Name;
    public int Popularity;

    public Animal (string name, int popularity)
    {
        Name = name; Popularity = popularity;
    }
}

public class AnimalCollection : Collection <Animal>
{
    // AnimalCollection to już w pełni funkcjonalna lista zwierząt.
    // Nie jest potrzebny żaden dodatkowy kod.
}

public class Zoo //klasa, która będzie udostępniać AnimalCollection,
{ //normalnie zawierałaby dodatkowe składowe
    public readonly AnimalCollection Animals = new AnimalCollection();
}
```

```

class Program
{
    static void Main()
    {
        Zoo zoo = new Zoo();
        zoo.Animals.Add (new Animal ("Kangur", 10));
        zoo.Animals.Add (new Animal ("Król Lew", 20));
        foreach (Animal a in zoo.Animals) Console.WriteLine (a.Name);
    }
}

```

W tej postaci klasa `AnimalCollection` nie przewyższa funkcjonalnością prostej klasy `List<Animal>`, ponieważ została stworzona tylko po to, by można ją było w przyszłości rozszerzyć. Dla przykładu do klasy `Animal` dodamy teraz własność `Zoo`, aby można było określić, w którym zoo mieszka dane zwierzę, i dodatkowo przesłonimy wszystkie metody wirtualne klasy `Collection<Animal>`, aby zapewnić automatyczną obsługę tej własności:

```

public class Animal
{
    public string Name;
    public int Popularity;
    public Zoo Zoo { get; internal set; }
    public Animal(string name, int popularity)
    {
        Name = name; Popularity = popularity;
    }
}

public class AnimalCollection : Collection <Animal>
{
    Zoo zoo;
    public AnimalCollection (Zoo zoo) { this.zoo = zoo; }
    protected override void InsertItem (int index, Animal item)
    {
        base.InsertItem (index, item);
        item.Zoo = zoo;
    }
    protected override void SetItem (int index, Animal item)
    {
        base.SetItem (index, item);
        item.Zoo = zoo;
    }
    protected override void RemoveItem (int index)
    {
        this [index].Zoo = null;
        base.RemoveItem (index);
    }
    protected override void ClearItems()
    {
        foreach (Animal a in this) a.Zoo = null;
        base.ClearItems();
    }
}

public class Zoo
{
    public readonly AnimalCollection Animals;
    public Zoo() { Animals = new AnimalCollection (this); }
}

```

Klasa `Collection<T>` zawiera też konstruktor przyjmujący obiekty implementujące interfejs `IList<T>`. Jednak inaczej niż jest w przypadku innych klas kolekcji, przekazana lista nie zostanie *skopiowana*, tylko nowy egzemplarz klasy `Collection<T>` będzie *pośrednikiem* do opakowywanej listy, więc wszelkie wprowadzane w niej zmiany będą widoczne przez tego pośrednika (choć *bez* uruchamiania metod wirtualnych klasy `Collection<T>`).

## Klasa `CollectionBase`

Klasa `CollectionBase` to niegeneryczna wersja klasy `Collection<T>`, którą wprowadzono w wersji 1.0 platformy. Zapewnia bardzo podobną funkcjonalność jak `Collection<T>`, tylko jest mniej wygodna w użyciu. Zamiast szablonowych metod: `InsertItem`, `RemoveItem`, `SetItem` i `ClearItem`, klasa `CollectionBase` zawiera metody „zaczepowe”, które podwajają tylko zestaw potrzebnych metod: `OnInsert`, `OnInsertComplete`, `OnSet`, `OnSetComplete`, `OnRemove`, `OnRemoveComplete`, `OnClear` oraz `OnClearComplete`. Jako że klasa `CollectionBase` nie jest generyczna, tworząc jej podklasę, należy dodatkowo zaimplementować metody typizowane — przynajmniej indeksator i metodę `Add`.

## Klasy `KeyedCollection<TKey,TItem>` i `DictionaryBase`

Klasa `KeyedCollection<TKey,TItem>` jest podklasą klasy `Collection<TItem>`. W pewnych obszarach rozszerza jej funkcjonalność, a w innych ogranicza. Jeśli chodzi o rozszerzenia, to dodaje możliwość odnoszenia się do elementów wg klucza, tak jak w słownikach. Likwiduje natomiast możliwość tworzenia pośredników listy wewnętrznej.

Kolekcja `KeyedCollection` przypomina w pewnym stopniu klasę `OrderedDictionary`, ponieważ także jest kombinacją liniowej listy i tablicy skrótów. Jednak w odróżnieniu od niej nie implementuje interfejsu `IDictionary` i nie rozpoznaje pojęcia *par* klucz – wartość. Klucze są tworzone z samych elementów przez abstrakcyjną metodę `GetKeyForItem`. Oznacza to, że kolekcję taką można przeglądać w taki sam sposób jak zwykłą listę.

Klasę `KeyedCollection<TKey,TItem>` można traktować jak `Collection<TItem>` z szybkim wyszukiwaniem po kluczach.

Jako że jest ona podklasą klasy `Collection<>`, dziedziczy wszystkie jej składniki z wyjątkiem możliwości podawania istniejącej listy konstruktorowi. Dodatkowe składowe, jakie definiuje, to:

```
public abstract class KeyedCollection <TKey, TItem> : Collection <TItem>
{
    // ...

    protected abstract TKey GetKeyForItem(TItem item);
    protected void ChangeItemKey(TItem item, TKey newKey);

    // szybkie wyszukiwanie wg klucza to dodatek do wyszukiwania indeksowego
    public TItem this[TKey key] { get; }

    protected IDictionary<TKey, TItem> Dictionary { get; }
}
```

Metodę `GetKeyForItem` należy przesłonić tak, aby prawidłowo obliczała klucze elementów utworzonego obiektu. Metodę `ChangeItemKey` należy wywołać, jeżeli zmieni się własność przechowująca klucz elementu, aby zaktualizować słownik wewnętrzny. Własność `Dictionary` zwraca wewnętrzny

słownik wykorzystywany do wyszukiwania, tworzony w chwili dodania pierwszego elementu. W razie potrzeby można zmienić tę zasadę w konstruktorze tak, aby utworzenie słownika następowało dopiero po przekroczeniu pewnego progu (do tego czasu elementy są wyszukiwane liniowo). Dobrym powodem do tego, by nie określać progu utworzenia słownika, jest to, że może on być przydatny do utworzenia kolekcji `ICollection<>` kluczy przy użyciu własności `Keys` klasy `Dictionary`.

Klasy `KeyedCollection<,>` najczęściej używa się do tworzenia kolekcji elementów dostępnych poprzez indeksy i nazwy. W ramach przykładu zaimplementujemy naszą klasę `AnimalCollection` jako `KeyedCollection<string,Animal>`:

```
public class Animal
{
    string name;
    public string Name
    {
        get { return name; }
        set {
            if (Zoo != null) Zoo.Animals.NotifyNameChange (this, value);
            name = value;
        }
    }
    public int Popularity;
    public Zoo Zoo { get; internal set; }

    public Animal (string name, int popularity)
    {
        Name = name; Popularity = popularity;
    }
}

public class AnimalCollection : KeyedCollection <string, Animal>
{
    Zoo zoo;
    public AnimalCollection (Zoo zoo) { this.zoo = zoo; }

    internal void NotifyNameChange (Animal a, string newName)
    {
        this.ChangeItemKey (a, newName);
    }

    protected override string GetKeyForItem (Animal item)
    {
        return item.Name;
    }

    // poniższe metody byłyby zaimplementowane, jak w poprzednim przykładzie
    protected override void InsertItem (int index, Animal item)...
    protected override void SetItem (int index, Animal item)...
    protected override void RemoveItem (int index)...
    protected override void ClearItems()...
}

public class Zoo
{
    public readonly AnimalCollection Animals;
    public Zoo() { Animals = new AnimalCollection (this); }
}
```



```

class Program
{
    static void Main()
    {
        Zoo zoo = new Zoo();
        zoo.Animals.Add (new Animal ("Kangur", 10));
        zoo.Animals.Add (new Animal ("Król Lew", 20));
        Console.WriteLine (zoo.Animals [0].Popularity); // 10
        Console.WriteLine (zoo.Animals ["Mr Sea Lion"].Popularity); // 20
        zoo.Animals ["Kangaroo"].Name = "Mr Roo";
        Console.WriteLine (zoo.Animals ["Mr Roo"].Popularity); // 10
    }
}

```

## DictionaryBase

DictionaryBase to nieregularna wersja klasy KeyedCollection. Jest to stara klasa, którą zbudowano w całkiem odmienny sposób niż nowsze — implementuje interfejs IDictionary i podobnie jak CollectionBase wykorzystuje nieporęczne metody zaczepowe: OnInsert, OnInsertComplete, OnSet, OnSetComplete, OnRemove, OnRemoveComplete, OnClear oraz OnClearComplete (i dodatkowo OnGet). Największą zaletą implementacji słownika IDictionary zamiast stosowania podejścia klasy Keyed ↪Collection jest to, że nie trzeba tworzyć podklasy, aby uzyskać klucze. Ponieważ jednak klasa DictionaryBase istnieje przede wszystkim po to, by utworzyć jej podklasę, ta zaleta nie ma znaczenia. Ulepszony model klasy KeyedCollection z pewnością powstał dlatego, że klasa ta została napisana kilka lat później, kiedy programiści mogli spojrzeć na nią z pewnej perspektywy. Dlatego też klasę DictionaryBase należy traktować głównie jako środek do zapewnienia zgodności ze starym kodem.

## Klasa ReadOnlyCollection<T>

Klasa ReadOnlyCollection<T> jest opakowaniem, albo *pośrednikiem*, zapewniającym widok tylko do odczytu kolekcji. Jest przydatna, gdy klasa ma publicznie udostępnić kolekcję, której nie można modyfikować z zewnątrz, a którą można zmieniać wewnętrznie.

Kolekcja tylko do odczytu przyjmuje kolekcję wejściową w konstruktorze, do którego ma stałą referencję. Nie tworzy ona statycznej kopii przekazanej kolekcji, więc wszelkie późniejsze zmiany kolekcji wejściowej są widoczne przez opakowanie.

W ramach przykładu powiedzmy, że chcemy utworzyć klasę publicznie udostępniającą listę łańcuchów tylko do odczytu o nazwie Names:

```

public class Test
{
    public List<string> Names { get; private set; }
}

```

To dopiero połowa pracy. Choć inne typy nie mogą nic przypisać do własności Names, nadal mogą wywoływać na liście metody Add, Remove i Clear. Rozwiązaniem problemu jest klasa ReadOnly ↪Collection<T>:

```

public class Test
{
    List<string> names;
    public ReadOnlyCollection<string> Names { get; private set; }
}

```

```

public Test()
{
    names = new List<string>();
    Names = new ReadOnlyCollection<string> (names);
}

public void AddInternally() { names.Add ("test"); }
}

```

Teraz tylko składowe klasy Test mogą zmieniać listę nazw:

```

Test t = new Test();

Console.WriteLine (t.Names.Count); // 0
t.AddInternally();
Console.WriteLine (t.Names.Count); // 1

t.Names.Add ("test"); // błąd kompilacji
((IList<string>) t.Names).Add ("test"); // NotSupportedException

```

## Dołączanie protokołów równości i porządkowania

W podrozdziałach „Sprawdzanie równości” i „Określanie kolejności” w rozdziale 6. opisaliśmy standardowe protokoły platformy .NET umożliwiające porównywanie typów i obliczanie ich skrótów. Typ implementujący te protokoły może prawidłowo funkcjonować w słowniku i liście posortowanej bez żadnego dodatkowego przygotowania. Mówiąc konkretniej:

- Typ, dla którego metody Equals i GetHashCode zwracają sensowne wyniki, może być wykorzystywany jako klucz w strukturach Dictionary i Hashtable.
- Typ implementujący interfejsy IComparable i IComparable<T> może być używany jako klucz we wszystkich *sortowanych* słownikach i listach.

Domyślna implementacja operacji sprawdzania równości i porównywania typu zazwyczaj odzwierciedla jego „naturalne” cechy. Ale programiście nie zawsze zależy na domyślnym znaczeniu tych operacji. Czasami potrzebny jest np. słownik, w którym klucze są typu string i nie ma znaczenia wielkość liter. Innym razem ktoś może potrzebować listy klientów posortowanej wg kodów pocztowych. Dlatego na platformie .NET Framework stworzono dodatkowy zestaw protokołów „dołączanych” o dwojakim przeznaczeniu:

- Dzięki nim można włączać alternatywne sposoby określania równości i kolejności elementów.
- Umożliwiają korzystanie ze słowników lub posortowanych kolekcji z typem kluczy, które w swej istocie nie są porównywalne.

Protokoły dołączane obejmują następujące interfejsy:

IEqualityComparer i IEqualityComparer<T>

- Odpowiada za *sprawdzanie równości i obliczanie skrótów*.
- Rozpoznawany przez Hashtable i Dictionary.

IComparer i IComparer<T>

- Odpowiada za *porównywanie pod względem kolejności*.
- Rozpoznawany przez słowniki i kolekcje posortowane; także Array.Sort.

Każdy interfejs występuje w wersji generycznej i niegenerycznej. Dodatkowo interfejsy IEquality i IEqualityComparer mają domyślną implementację w klasie o nazwie EqualityComparer.

Ponadto w .NET Framework 4.0 dodano dwa nowe interfejsy: IStructuralEquatable i IStructuralComparable, które umożliwiają wykonywanie porównań strukturalnych na klasach i tablicach.

## IEqualityComparer i EqualityComparer

Interfejs IEqualityComparer włącza niestandardowe operacje sprawdzania równości i obliczania skrótów i jest przeznaczony głównie do pracy z klasami Dictionary i Hashtable.

Przypomnimy pokrótce wymagania, jakie powinien spełniać słownik skrótów. Dla każdego klucza musi odpowiadać na dwa pytania:

- Czy jest taki sam jak inny?
- Jaka jest jego całkowitoliczbowa wartość skrótu?

Komparator sprawdzający równość odpowiada na te pytania przez implementację interfejsów IEqualityComparer:

```
public interface IEqualityComparer<T>
{
    bool Equals (T x, T y);
    int GetHashCode (T obj);
}

public interface IEqualityComparer // wersja niegeneryczna
{
    bool Equals (object x, object y);
    int GetHashCode (object obj);
}
```

Aby napisać własny komparator, należy zaimplementować jeden z tych interfejsów lub oba (implementacja obu zapewnia największą elastyczność). Ponieważ jednak jest to dość czasochłonne zajęcie, ewentualnie można też utworzyć podklasę abstrakcyjnej klasy EqualityComparer, której definicja wygląda następująco:

```
public abstract class EqualityComparer<T> : IEqualityComparer,
                                           IEqualityComparer<T>
{
    public abstract bool Equals (T x, T y);
    public abstract int GetHashCode (T obj);

    bool IEqualityComparer.Equals (object x, object y);
    int IEqualityComparer.GetHashCode (object obj);

    public static EqualityComparer<T> Default { get; }
}
```

Klasa EqualityComparer implementuje oba interfejsy, więc programiście pozostaje już tylko przesłonięcie dwóch abstrakcyjnych metod.

Semantyka metod `Equals` i `GetHashCode` jest taka sama jak metod `object.Equals` i `object.GetHashCode` opisanych w rozdziale 6. W poniższym przykładzie definiujemy klasę `Customer` z dwoma polami i piszemy komparator porównujący imiona i nazwiska:

```
public class Customer
{
    public string LastName;
    public string FirstName;

    public Customer (string last, string first)
    {
        LastName = last;
        FirstName = first;
    }
}
public class LastFirstEqComparer : EqualityComparer <Customer>
{
    public override bool Equals (Customer x, Customer y)
        => x.LastName == y.LastName && x.FirstName == y.FirstName;

    public override int GetHashCode (Customer obj)
        => (obj.LastName + ";" + obj.FirstName).GetHashCode();
}
```

Aby pokazać, jak to działa, utworzymy dwóch klientów:

```
Customer c1 = new Customer ("Barański", "Jan");
Customer c2 = new Customer ("Barański", "Jan");
```

Jako że nie przesłoniliśmy metody `object.Equals`, zastosowanie ma normalna semantyka referencyjna:

```
Console.WriteLine (c1 == c2); // falsz
Console.WriteLine (c1.Equals (c2)); // falsz
```

To samo dzieje się, gdy użyjemy naszych obiektów w słowniku, nie określając własnego komparatora:

```
var d = new Dictionary<Customer, string>();
d [c1] = "Jan";
Console.WriteLine (d.ContainsKey (c2)); // falsz
```

A teraz podajemy własny komparator:

```
var eqComparer = new LastFirstEqComparer();
var d = new Dictionary<Customer, string> (eqComparer);
d [c1] = "Jan";
Console.WriteLine (d.ContainsKey (c2)); // prawda
```

W tym przypadku musielibyśmy uważać, aby nie zmienić pól `FirstName` ani `LastName` klienta podczas używania obiektu w słowniku. W przeciwnym razie zmieniłaby się wartość skrótu i słownik przestałby działać.

## Metoda `EqualityComparer<T>.Default`

Metoda `EqualityComparer<T>.Default` zwraca ogólny komparator, którego można używać zamiast statycznej metody `object.Equals`. Jej zaletą jest to, że najpierw sprawdza, czy typ `T` implementuje interfejs `IComparable<T>`, i jeśli tak, to wywołuje tamtą implementację, pozwalając uniknąć pakowania. Jest to szczególnie przydatne w metodach generycznych:

```

static bool Foo<T> (T x, T y)
{
    bool same = EqualityComparer<T>.Default.Equals (x, y);
    ...
}

```

## IComparer i Comparer

Komparatory umożliwiają dołączanie do posortowanych słowników i kolekcji niestandardowej logiki porównawczej.

Dla nieposortowanego słownika, np. Dictionary i Hashtable, komparator jest bezużyteczny — w takim przypadku potrzebny jest IEqualityComparer, aby możliwe było obliczanie wartości skrótów. Analogicznie komparator równości jest nieprzydatny dla posortowanych słowników i kolekcji.

Oto definicje interfejsu IComparer:

```

public interface IComparer
{
    int Compare(object x, object y);
}
public interface IComparer <in T>
{
    int Compare(T x, T y);
}

```

Tak jak w przypadku komparatorów równości, istnieje klasa abstrakcyjna, którą można rozszerzyć zamiast pisać implementację interfejsów:

```

public abstract class Comparer<T> : IComparer, IComparer<T>
{
    public static Comparer<T> Default { get; }

    public abstract int Compare (T x, T y); // do implementacji przez programistę
    int IComparer.Compare (object x, object y); // do implementacji przez programistę
}

```

Poniżej znajduje się przykład ilustrujący klasę opisującą życzenia i komparator sortujący życzenia wg ważności:

```

class Wish
{
    public string Name;
    public int Priority;

    public Wish (string name, int priority)
    {
        Name = name;
        Priority = priority;
    }
}

class PriorityComparer : Comparer <Wish>
{
    public override int Compare (Wish x, Wish y)
    {
        if (object.Equals (x, y)) return 0; // test bezpieczeństwa
        return x.Priority.CompareTo (y.Priority);
    }
}

```

Test bezpieczeństwa metody `object.Equals` gwarantuje nam, że nigdy nie zaprzeczymy metodzie `Equals`. Wywołanie statycznej metody `object.Equals` jest w tym przypadku lepszym rozwiązaniem niż wywołanie `x.Equals`, ponieważ metoda ta zadziała także, gdy `x` będzie `null`!

Poniżej przedstawiamy przykład użycia naszego komparatora `PriorityComparer` do posortowania listy:

```
var wishList = new List<Wish>();
wishList.Add (new Wish ("Pokój", 2));
wishList.Add (new Wish ("Bogactwo", 3));
wishList.Add (new Wish ("Miłość", 2));
wishList.Add (new Wish ("3 kolejne życzenia", 1));

wishList.Sort (new PriorityComparer());
foreach (Wish w in wishList) Console.Write (w.Name + " | ");

// WYNIK: 3 kolejne życzenia | Bogactwo | Miłość | Pokój |
```

W następnym przykładzie klasa `SurnameComparer` pozwala na posortowanie nazwisk w kolejności odpowiedniej dla książki telefonicznej:

```
class SurnameComparer : Comparer <string>
{
    string Normalize (string s)
    {
        s = s.Trim().ToUpper();
        if (s.StartsWith ("MC")) s = "MAC" + s.Substring (2);
        return s;
    }

    public override int Compare (string x, string y)
        => Normalize (x).CompareTo (Normalize (y));
}
```

Oto przykład użycia tego komparatora w posortowanym słowniku:

```
var dic = new SortedDictionary<string,string> (new SurnameComparer());
dic.Add ("MacPhail", "drugi!");
dic.Add ("MacWilliam", "trzeci!");
dic.Add ("McDonald", "pierwszy!");

foreach (string s in dic.Values)
    Console.Write (s + " "); // pierwszy! drugi! trzeci!
```

## Klasa `StringComparer`

`StringComparer` to gotowa klasa do sprawdzania równości i porównywania łańcuchów, która dodatkowo umożliwia określenie języka i podjęcie decyzji, czy ma być rozpoznawana wielkość liter. Klasa ta implementuje interfejsy `IEqualityComparer` i `IComparer` (oraz ich generyczne wersje), więc można jej używać z każdym typem słownika lub posortowanej kolekcji:

```
// CultureInfo znajduje się w przestrzeni nazw System.Globalization

public abstract class StringComparer : IComparer, IComparer <string>,
    IEqualityComparer,
    IEqualityComparer <string>
{
```

```

public abstract int Compare (string x, string y);
public abstract bool Equals (string x, string y);
public abstract int GetHashCode (string obj);

public static StringComparer Create (CultureInfo culture,
                                     bool ignoreCase);
public static StringComparer CurrentCulture { get; }
public static StringComparer CurrentCultureIgnoreCase { get; }
public static StringComparer InvariantCulture { get; }
public static StringComparer InvariantCultureIgnoreCase { get; }
public static StringComparer Ordinal { get; }
public static StringComparer OrdinalIgnoreCase { get; }
}

```

Klasa `StringComparer` jest abstrakcyjna, więc egzemplarze otrzymuje się przez metody statyczne i właściwości. Metoda `StringComparer.Ordinal` odzwierciedla domyślny sposób sprawdzania równości łańcuchów, a metoda `StringComparer.CurrentCulture` dotyczy porównywania pod względem kolejności.

W poniższym przykładzie stworzymy porządkowy słownik nierozróżniający wielkości liter, taki że `dict["Jan"]` i `dict["JAN"]` znaczą to samo:

```
var dict = new Dictionary<string, int> (StringComparer.OrdinalIgnoreCase);
```

W następnym przykładzie sortujemy tablicę imion wg zasad australijskiej odmiany angielskiego:

```

string[] names = { "Tom", "HARRY", "sheila" };
CultureInfo ci = new CultureInfo ("en-AU");
Array.Sort<string> (names, StringComparer.Create (ci, false));

```

Ostatni przykład to rozpoznająca kultury wersja komparatora `SurnameComparer`, który napisaliśmy w poprzedniej sekcji (do porównywania nazwisk w sposób odpowiedni dla książki telefonicznej):

```

class SurnameComparer : Comparer <string>
{
    StringComparer strCmp;
    public SurnameComparer (CultureInfo ci)
    {
        // utworzenie rozpoznającego wielkość liter i kultury komparatora łańcuchów
        strCmp = StringComparer.Create (ci, false);
    }
    string Normalize (string s)
    {
        s = s.Trim();
        if (s.ToUpper().StartsWith ("MC")) s = "MAC" + s.Substring (2);
        return s;
    }

    public override int Compare (string x, string y)
    {
        // bezpośrednie wywołanie metody Compare na naszym komparatorze StringComparer
        return strCmp.Compare (Normalize (x), Normalize (y));
    }
}

```

## Interfejsy `IStructuralEquatable` i `IStructuralComparable`

Jak napisaliśmy w poprzednim rozdziale, struktury domyślnie implementują **porównywanie strukturalne**, które polega na tym, że dwie struktury zostają uznane za równe, jeśli wszystkie ich pola są sobie równe. Czasami jednak strukturalne porównywanie i równość są przydatne także w innych

typach, np. tablicach i krotkach, do których można je dodać jako protokoły dołączane. W wersji 4.0 platformy wprowadzono więc dwa nowe interfejsy:

```
public interface IStructuralEquatable
{
    bool Equals (object other, IEqualityComparer comparer);
    int GetHashCode (IEqualityComparer comparer);
}

public interface IStructuralComparable
{
    int CompareTo (object other, IComparer comparer);
}
```

Przekazywane komparatory `IEqualityComparer` i `IComparer` są stosowane do każdego elementu w obiekcie złożonym. Możemy to zademonstrować przy użyciu tablic i krotek, które implementują te interfejsy. W poniższym przykładzie sprawdzamy, czy dwie tablice są sobie równe, najpierw za pomocą domyślnej metody `Equals`, a następnie przy użyciu jej wersji z interfejsu `IStructuralEquatable`:

```
int[] a1 = { 1, 2, 3 };
int[] a2 = { 1, 2, 3 };
IStructuralEquatable sel = a1;
Console.WriteLine (a1.Equals (a2)); //fałsz
Console.WriteLine (sel.Equals (a2, EqualityComparer<int>.Default)); //prawda
```

Oto kolejny przykład:

```
string[] a1 = "gdyby kózka nie skakała".Split();
string[] a2 = "GDYBY KÓZKA NIE SKAKAŁA".Split();
IStructuralEquatable sel = a1;
bool isTrue = sel.Equals (a2, StringComparison.InvariantCultureIgnoreCase);
```

Krotki działają tak samo:

```
var t1 = Tuple.Create (1, "foo");
var t2 = Tuple.Create (1, "F00");
IStructuralEquatable sel = t1;
bool isTrue = sel.Equals (t2, StringComparison.InvariantCultureIgnoreCase);
IStructuralComparable sc1 = t1;
int zero = sc1.CompareTo (t2, StringComparison.InvariantCultureIgnoreCase);
```

Różnica, jeśli chodzi o krotki, jest taka, że ich *domyślne* implementacje operacji sprawdzania równości i porównywania mają charakter strukturalny:

```
var t1 = Tuple.Create (1, "F00");
var t2 = Tuple.Create (1, "F00");
Console.WriteLine (t1.Equals (t2)); //prawda
```



.ASMX Web Services, 217  
.NET Framework, 205

## A

abstrakcyjne

klasy, 105

składowe, 105

ACL, Access Control List, 861

adapter

StringReader, 632

StringWriter, 632

strumienia, 626

adaptery

binarne, 626, 632

tekstowe, 626, 627

XML, 627

adnotacje, 458

ADO.NET, 214

adres

IPv4, 664

IPv6, 664

SMTP, 691

URI, 665

agregacje bez ziarna, 428

akcesory widoku, 652

aksjomaty metody `object.Equals`,  
279

aktualizowanie danych, 446

aktywacja typów, 774

algorytm

MD5, 866

porównywania kulturowego,  
225

porównywania

porządkowego, 225

SHA, 866

SHA256, 866

SHA384, 866

algorytmy szyfrowania, 874

alias extern, 83

aliasy typów, 83

alternatywy, 1011

analiza

działających procesów, 539

poleceń, 1018

wątków w procesie, 539

wyrażeń, 1018

anonimowe wywoływanie

składowych, 790, 831

anulowanie zadań, 942

anulowanie zapytania PLINQ,  
927

API EF, 381

API COM, 987

API L2S, 381

aplikacje

typu WPF, 564

Windows Store, 775

APM, Asynchronous

Programming Model, 606

architektura

domeny aplikacji, 959

Roslyn, 1014

sieci, 661

strumienia, 611

archiwum ZIP, 636

argument, 28

NumberStyles, 253

argumenty

metody `GroupJoin`, 406

metody `Join`, 406

nazwane, 24, 64, 987

typów, 126

ASCII, 229

asercja, 515, 522, 529, 858

o zerowej długości, 999, 1010

asocjacje, 375

ASP.NET, 212

ASP.NET 5, 19

asynchroniczne wyrażenia

lambda, 593

asynchroniczność, 209, 551, 577,  
594

atak słownikowy, 866

atomowość, 882

atrapy, 673

atrybut, 191, 210, 793

[`ComVisible(false)`], 991

[`Conditional`], 513, 514

[`ContractVerification`], 537

[`DataMember`], 706

[`NonSerialized`], 720, 722

[`OnDeserialized`], 716, 721

[`OnDeserializing`], 716, 721

[`OnSerialized`], 722

[`OnSerializing`], 716, 722

[`OptionalField`], 723

[`SecurityTransparent`], 847,  
848

[`Serializable`], 717

[`SuppressUnmanagedSecurity`],  
853

[`SupressMessage`], 537

[`ThreadStatic`], 909

APTCA, 847

AttributeUsage, 795

atrybut  
Flags, 122  
LoaderOptimization, 963

atrybuty  
bitmapowe, 794  
debuggera, 538  
informacji wywołującego,  
193  
kompresji, 639  
nazwane, 192  
niestandardowe, 794  
podzespołu, 739  
pozycyjne, 192  
pseudoniestandardowe, 795  
serializacji binarnej, 720  
szyfrowania, 639  
warunkowe, 198

automatyczna konkatencja  
węzłów XText, 449

automatyczne usuwanie  
nieużytków, 491

awarie, 532

## B

bariera, 906

bariera wątku wykonawczego,  
905

bezpieczeństwo, 210, 839  
deklaratywne, 843  
dostępu kodu, 840  
imperatywne, 843  
pliku, 640  
typów, 18, 123  
wątków, 528, 558, 886–890

bezpieczne metody krytyczne,  
850

bezpośrednie osadzanie  
zasobów, 755

białe znaki, 1023

biblioteka, 29  
PCL, 19  
PEX, 918

biblioteki  
APTCA, 852  
DLL, 973  
natywne, 973  
WinRT, 21

binarny rewriter, 520, 521

BitTorrent, 692

blok  
catch, 158  
finally, 158, 161  
instrukcji, 28, 33  
kodu, 27  
try, 158  
try-catch-finally, 561  
try-finally, 486

blokady, 881, 886  
odczytu i zapisu, 893  
z możliwością uaktualnienia,  
895  
z podwójnym  
zatwierdzeniem, 907

bloki try-catch-finally, 169

blokowanie, 882  
bez wykluczania, 878, 892  
pętlowe, 919  
wykluczające, 878

błędy  
parsowania, 255  
zaokrąglania liczb  
rzeczywistych, 49

BMP, Basic Multilingual Plane,  
231

budowanie wyrażeń zapytań, 381

buforowanie, 508

bufory o stałym rozmiarze, 196

## C

CAS, code access security, 840

CCW, COM-callable wrapper, 991

cele  
atrybutów, 192  
delegatów, 142  
emisji, 816

certyfikat, 749  
witryny, 872

ciasteczka, *Patrz* cookies

ciąg tekstowy zapytania, 680

CLR, 19, 207, 501  
implementacja  
indeksatorów, 96  
implementacja  
własności, 95

COM, Component Object  
Model, 21, 985

COM+, 216

cookies, 682

cyfrowy certyfikat witryny, 872

czas  
bieżący, 236  
działania programu, 18  
kompilacji, 18  
letni, 243  
UTC, 244

czekanie na zadania, 942

czystość funkcyjna, 926

czyszczenie egzemplarzy  
nasłuchujących, 518

## D

dane  
statyczne, 132  
uwierzytelniające, 676

data i godzina, 232

deassembler, 819

debugger, 538  
atrybuty, 538  
punkty kontrolne, 538

debugowanie, 515

definicja równości, 277

definiowanie  
metod generycznych, 814  
przestrzeni nazw, 455  
treści, 438  
typów, 34  
typów generycznych, 815  
własnych atrybutów, 796

deklaracje, 450  
XML, 452

deklarowanie  
parametrów typów, 128, 135  
wielu pól naraz, 88

dekoratory strumieni, 612, 626

delegat, 18, 139, 382  
Action, 143  
Func, 143  
MatchEvaluator, 1004

delegaty  
asynchroniczne, 607  
multiemisji, 141  
zgodność, 145

- deserializacja, 699
  - deszyfrowanie wiadomości, 872
  - diagnostyka, 209, 511
  - diagnozowanie wycieku pamięci, 506
  - diagram UML, 355
  - DLL, Dynamic Link Libraries, 985
  - DLR, dynamiczny system wykonawczy języka, 825
  - DNS, Domain Name Service, 663, 690
  - dokumentacja XML, 200
  - dokumenty, 450
  - dołączane komparatory, 281
  - dołączanie debuggera, 538
  - DOM, 433
  - domeny aplikacji, 211, 959
    - architektura, 959
    - likwidowanie, 961
    - monitorowanie, 965
    - tworzenie, 961
    - używanie, 962
  - domyślna
    - inicjalizacja elementów, 54
    - wartość generyczna, 129
  - domyślne
    - przestrzenie nazw, 456
    - ustawienia przezroczystości, 852
  - dostawca
    - formatu, 245–247
    - WordyFormatProvider, 250
  - dostęp do składowych niepublicznych, 788
  - dostępność symbolu, 1034
  - dostosowanie kolekcji, 714
  - drobiazgi, 1023
    - niestrukuralne, 1025
    - strukuralne, 1025
  - drzewo
    - składni Roslyn, 1015
    - wyrażeń, 25 154, 362, 382, 386
    - wywołań, 579
    - wywołań asynchronicznych, 591
    - X-DOM, 435
  - dynamiczna kontrola typów, 111
  - dynamiczne
    - wybieranie przeciążonych składowych, 828
    - wywoływanie składowej, 786
  - dynamiczny
    - odbiorca, 188
    - system wykonawczy języka, 825
  - dyrektywa preprocesora, 198, 1024
    - #else, 512, 1024
    - #if, 511, 1024
    - #if LOGGINGMODE, 514
    - #line, 1024
    - fixed, 981
    - using, 80, 82
    - using static, 80
  - działanie
    - modelu transparentności, 849
    - programu, 860
    - wyliczeń, 269
    - zdarzeń, 148
  - dziedziczenie, 101, 107
  - dzielenie
    - całkowitoliczbowe, 45
    - danych między domenami, 967
    - łańcuchów, 223
    - na części, 930
    - przy użyciu skrótów, 929
    - tekstu, 1003
    - zakresowe, 930
  - dziennik zdarzeń
    - monitorowanie, 543
    - odczyt danych, 543
    - Windows, 542
    - zapis danych, 542
  - egzekwowanie
    - kontraktów, 534
  - egzemplarz, 35
    - nasłuchujący, 518
    - referencyjny, 39
  - elementy, 53
    - formatowania, 224
    - główne, 492, 493
    - opcjonalne, 470
    - podklasy kolekcji, 714, 734
    - puste, 470
    - składowe, 710, 711
  - eliminowanie pustych elementów, 461
  - emisja podzespołu, 1031
  - emitowanie
    - generycznych typów, 814
    - klas, 814
    - konstruktorów, 812
    - metod, 809
    - podzespołów i typów, 805
    - pól, 811
    - składowych typów, 809
    - właściwości, 811
  - encja, 468
  - Entity Framework, 368
  - enumerator, 166
- ## F
- fabryka abstrakcyjna, 948
  - FIFO, first-in, first-out, 312
  - filtr, 377
    - wyjątków, 23, 161
  - filtrowanie, 391
    - z indeksowaniem, 392
  - finalizatory, 29, 99, 493
    - wskrzeszanie, 496
    - wywoływanie metody Dispose(), 495
  - flagi parsowania, 250, 254
  - forma daty
    - długa, 238
    - krótka, 238
  - format XML, 208
  - formater, 702
    - SOAP, 722
    - XML, 705
  - formaterzy binarne, 706

formatowanie, 244  
  daty i godziny, 254, 255  
  wyliczeń, 257  
  złożone, 248  
formularz, 681  
fragmentacja, 501  
FTP, File Transfer Protocol, 663,  
  688  
funkcja, 18  
  CreateFileMapping, 979  
  GetComplexMessageHeaders,  
    513  
funkcje  
  asynchroniczne, 582, 589  
  klasy FileStream, 621  
  operatorowe, 177  
  wyrażeniowe, 23

## G

GAC, Global Assembly Cache,  
  751, 753  
generator  
  liczb losowych, 266  
  list z list, 343  
generowanie  
  dynamicznego kodu, 799  
  IL, 799  
  jednostek, 375  
  metod instancji, 810  
  zmiennych lokalnych, 802  
generyczne  
  kolekcje, 314  
  typy delegacyjne, 143  
globalizacja, 261  
globalny bufor zestawów, 22  
głębokie klonowanie, 439  
główne moduły współpracujące,  
  990  
granica słowa, 1001  
granice tablic, 57  
grupowanie, 416  
  sekwencji, 390  
  według wielu kluczy, 419

## H

hermetyzacja, 17  
hierarchie obiektów, 396

HTTP, Hypertext Transfer  
  Protocol, 663  
  ciąg tekstowy zapytania, 680  
  cookies, 682  
  dane formularza, 681  
  nagłówki, 680  
  tworzenie serwera, 685

## I

IANA, Internet Assigned  
  Numbers Authority, 230  
identyfikatory, 30  
IIS, Internet Information  
  Services, 663  
IL, Intermediate Language, 20,  
  737  
implementowanie  
  indeksatora, 96  
  interfejsów IComparable, 283  
  interfejsów przeliczeniowych,  
    293  
  obiektów dynamicznych, 833  
  własności w CLR, 95  
implikacje przekazywania przez  
  referencję, 63  
indeksatory, 29, 95, 988  
indeksowanie tablic, 302  
inferencja typów literalów  
  liczbowych, 43  
informacje  
  o operatorach zapytań, 388  
  o postępie, 600  
  o woluminie, 646  
  o wskaźnikach, 194  
inicjalizacja  
  elementów, 54  
  pól, 88, 108  
  tablic, 56  
inicjatory  
  indeksów, 23  
  kolekcji, 167  
  obiektów, 25, 91, 92, 360  
  własności, 23, 94  
initialization vector, IV, 868  
instalacja  
  certyfikatu, 749  
  typów, 778

instrukcja  
  break, 77, 936  
  continue, 78  
  fixed, 195  
  goto, 78  
  if, 73  
  lock, 879  
  return, 78, 168  
  switch, 74, 823  
  throw, 78  
  try, 158  
  using, 162, 486  
  yield break, 169  
instrukcje  
  deklaracji, 71  
  iteracyjne, 75  
  skoku, 77  
  wyboru, 73  
  wyrażeniowe, 72  
integracja z debuggerem, 538  
interakcja między  
  subskrybentami., 147  
interfejs, 17, 116  
  COM, 985  
  Comparer, 331  
  ICollection, 296, 297  
  ICollection<T>, 297  
  IComparable, 281–283  
  Comparer, 331  
  IConvertible, 258  
  ICustomFormatter, 249  
  IDictionary, 316, 318  
  IDictionary<TKey,TValue>,  
    317  
  IDispatch, 986  
  IDisposable, 292, 485, 489  
  IEnumerable, 290  
  IEnumerable<T>, 291, 292  
  IEnumerator, 290  
  IEnumerator<T>, 291, 295  
  IEqualityComparer, 320, 328  
  IEqualityComparer<T>, 328  
  IEquatable<T>, 276, 280  
  IFormatProvider, 249  
  IList, 296, 298  
  IList<T>, 298  
  IOrderedEnumerable, 415  
  IOrderedQueryable, 415

- interfejs
    - IPermission, 841
    - IProducerConsumer
      - ↳Collection<T>, 952
    - IProgress<T>, 601
    - IQueryable<T>, 362
    - IReadOnlyList<T>, 299
    - ISerializable, 724
    - IStructuralComparable, 333
    - IStructuralEquatable, 273, 333
    - IUnknown, 986
    - IXmlSerializable, 478, 702, 718, 734
    - Progress<T>, 601
  - interfejsy, 116
    - jawna implementacja, 117
    - konwersja struktury, 120
    - niegeneryczne, 292
    - przeliczeniowe, 293
    - reimplementacja, 118
    - rozszerzanie, 117
    - stosowanie, 120
    - tworzenie, 120
    - wirtualna implementacja
      - składowych, 118
  - interoperacyjność macierzysta, 211
  - interpolacja łańcuchów, 23, 53
  - IP, Internet Protocol, 663
  - iteratory, 168
    - komponowanie sekwencji, 170
  - izolacja
    - domeny, 654
    - podzespołu, 654
    - typów i zestawów, 969
- J**
- jawna implementacja interfejsu, 117
  - jawne określanie typów, 154
  - jednoczesne przetwarzanie
    - zadań, 551
  - jednolitość typów, 266
  - jednolity system typów, 17
  - jednostka, entity, 369
- język
    - pośredni IL, 20
    - XSLT, 483
  - języki
    - dynamiczne, 836
    - zarządzane, 20
  - JIT, just-in-time, 20
- K**
- katalog, 648
    - bazowy aplikacji, 619
    - specjalny, 644
  - kategoryzacja znaków, 221, 1009
  - klamry, 73
  - klasa, 17, 29, 87
    - Aes, 868
    - AggregateException, 948, 949
    - AppContext, 287
    - AppDomain, 970
    - Array, 300, 304
    - ArrayList, 308
    - Assembly, 741, 742, 767
    - AssemblyName, 746
    - AutoResetEvent, 897
    - BackgroundWorker, 608
    - Barrier, 905
    - BinaryReader, 632
    - BinaryWriter, 632
    - BitArray, 314
    - BitConverter, 261
    - BlockingCollection, 955
    - BlockingCollection<T>, 954
    - CodeAccessPermission, 840
    - Collection<T>, 323
    - CollectionBase, 325
    - ConcurrentBag<T>, 953
    - Console, 284
    - Contacts, 478
    - Contract, 521
    - Convert, 258
    - CountdownEvent, 901
    - CryptoStream, 868, 869
    - CSharpCompilationOptions, 1030
    - CSharpSyntaxRewriter, 1029
    - CSharpSyntaxWalker, 1022
    - DataContext, 371
    - DataContractSerializer, 704, 709
    - DataLoadOptions, 377
    - Debug, 515
    - DelegatingHandler, 674
    - Dictionary<TKey,TValue>, 319
    - DictionaryBase, 327
    - Directory, 641
    - DirectoryInfo, 642
    - Dns, 690
    - DynamicMethod, 799
    - DynamicObject, 833, 834
    - Encoding, 230, 631
    - Enumerable, 412
    - Environment, 285
    - Excel.Application, 987
    - ExpandoObject, 836
    - File, 619, 638
    - FileInfo, 642
    - FileIOPermission, 840
    - FileStream, 841
    - FileSystemWatcher, 647
    - HashSet<T>, 314
    - Hashtable, 319
    - HttpClient, 670, 676
    - HttpClientHandler, 675
    - HttpContent, 672
    - HttpListener, 686
    - HttpMessageHandler, 673
    - HybridDictionary, 320
    - IPAddress, 664
    - KeyedCollection<TKey, TItem>, 325
    - LambdaExpression, 384
    - Lazy<T>, 907
    - LazyInitializer, 908
    - LinkedList<T>, 310
    - List<T>, 308
    - ListDictionary, 320
    - ManualResetEvent, 900
    - Math, 262
    - Mutex, 885
    - NetDataContractSerializer, 704, 709
    - object, 112, 274
    - ObjectContext, 371
    - OpCodes, 820

klasa

- OrderedDictionary, 320
- Parallel, 919, 933
- ParallelLoopState, 936
- Path, 642
- PermissionSet, 842
- PrincipalPermission, 840
- Process, 286
- Publisher, 751
- Queue, 312
- Queue<T>, 312
- Random, 265
- ReaderWriterLockSlim, 893, 895
- ReadOnlyCollection<T>, 327
- RSA, 873
- RSACryptoServiceProvider, 875
- SemaphoreSlim, 892
- SerializationInfo, 725
- SharedMem, 982
- SmtpClient, 691
- SortedDictionary<, >, 321
- SortedList<, >, 321
- SortedList<T>, 314
- Stack, 313
- Stack<T>, 313
- StackFrame, 540
- StackTrace, 540
- Stopwatch, 549
- StorageFile podstawowa, 649
- StorageFolder, 648
- Stream, 613
- StreamReader, 629
- StreamWriter, 629
- StringBuilder, 228
- StringComparer, 332
- Supplier, 478
- SurnameComparer, 332
- SymbolInfo, 1034
- SyntaxNode, 1015
- SyntaxTree, 1019
- System.Exception, 164
- Task, 569
- TaskCompletionSource, 575, 577
- TaskFactory, 948
- TextReader, 627, 630
- TextWriter, 627, 630
- ThreadLocal<T>, 909, 910, 925
- TimeZone, 240
- TimeZoneInfo, 240
- Trace, 515
- TraceListener, 516
- Type, 777
- TypeInfo, 775, 1035
- WebClient, 667, 681
- WebRequest, 669
- WebResponse, 669
- XDocument, 450, 451
- XElement, 479, 480
- XmlConvert, 245, 259
- XmlReader, 465, 469, 479
  - odczytywanie elementów, 469
  - przedrostki, 473
  - przestrzeń nazw, 473
  - sprawdzanie poprawności, 481
  - wczytywanie atrybutów, 472
  - wczytywanie węzłów, 467
  - zastosowania, 476
- XmlReaderSettings, 466
- XmlSerializer, 730
- XmlWriter, 474, 480
  - przedrostki, 476
  - przestrzeń nazw, 476
  - wpisywanie atrybutów, 475
  - wpisywanie węzłów, 475
  - zastosowania, 476
- ZipArchive, 636
- ZipFile, 637
  - abstrakcyjne, 105
  - atrybutów, 191
  - bazowe, 108
  - dziedziczenie, 101
  - jednostek Entity Framework, 370
  - jednostek LINQ to SQL, 369
  - metod egzemplarza, 637
  - monitorowania, 515
  - po stronie klienta, 667
  - pochodne, 101
  - pomocnicze, 284
  - stacyczne, 36, 98, 637
  - tworzenie, 120

klauzula

- catch, 160
- else, 73
- from, 344
- inicjalizacji, 76
- into, 410
- iteracyjna, 76
- select, 396
- warunkowa, 76

klient, 552

- P2P, 662

klienty

- bogate, 212
- ubogie, 212

klonowanie głębokie, 439

klucz, 319

- prywatny, 871
- publiczny, 743, 871

kod

- IL, 737, 799, 819
- niebezpieczny, 195
- niezarządzany, 977
- programu hosta, 856
- transparentny, 851
- zarządzany, 20

kodowanie

- ASCII, 229, 630
- base64, 259
- Unicode, 229
- UTF-16, 231, 631
- UTF-8, 630

kolejka, 312

- FIFO, 312
- komunikatów, 565
- LIFO, 109
- typu producent-konsument, 954

kolejność elementów, 470, 923

- składowych, 711
- XML, 729
- inicjalizacji pól, 90, 98, 108
- wykonywania działań, 67

kolekcja, 208, 289, 322, 713

- blokująca, 954
- pokoleniowa, 499
- współbieżna, 951

- kombinacje zapytań, 366
  - komentarz, 27, 33
    - dokumentacyjny, 200
  - komparator, 331, 415
    - równości, 331, 419
  - kompilacja, 18, 29
    - warunkowa, 511, 512
  - kompilator
    - csc.exe, 29
    - JIT, 20, 95
    - Roslyn, 1013
  - kompiłowanie drzew wyrażań, 382
  - komponenty
    - bibliotek PFX, 918
    - COM, 986
  - komponowanie sekwencji, 170
  - kompozycja podzapytania, 355
  - kompresja
    - plików, 636
    - strumienia, 634
    - w pamięci, 636
  - komunikaty
    - odpowiedzi, 672
    - żądania, 672
  - konfiguracja Release, 535
  - konflikt nazw, 32
  - konkatenacja
    - łańcuchów, 52
    - węzłów XText, 449
  - konsola MMC, 544
  - konstrukcja
    - funkcyjna, 438
    - try-catch, 561
    - try-catch-finally, 561
  - konstrukcje
    - grupujące, 1010
    - sygnalizacyjne, 901
  - konstruktor bezparametrowy, 108
  - konstruktory, 29, 35, 107
    - egzemplarzy, 89
    - niejawne bez parametrów, 90
    - niepubliczne, 91
    - przeciążanie, 90
    - statyczne, 97
  - kontekst synchronizacji, 566
  - konteksty typizowane, 372
  - kontrakty, 531
    - danych, 713
    - kodu, 518, 521
      - binarny rewriter, 521
      - czystość, 522
      - kompilacja, 521
      - warunki początkowe, 523
    - sprawdzenie statyczne, 536
    - w konfiguracji Release, 535
  - kontrawariancja, 136, 145
  - Kontrola konta użytkownika, UAC, 860
  - kontrola typów
    - dynamiczna, 111
    - statyczna, 111
  - kontynuacje, 573, 579, 902
    - na jednym przodku, 947
    - warunkowe, 945
    - z wieloma przodkami, 947
    - zapytania, 358, 410
  - konwersje, 37
    - dynamiczne, 187, 259
    - dziesiętne, 45
    - jawne, 37
    - jawne niestandardowe, 178
    - liczb rzeczywistych, 258
    - liczbowe, 263
    - między typami
      - całkowitoliczbowymi, 44
      - między typami
        - zmiennoprzecinkowymi, 44
    - na format base64, 259
    - niejawne, 37
    - niejawne niestandardowe, 178
    - referencji, 102, 146
    - rozpakowania, 422
    - tablic, 307
    - typu logicznego, 49
    - uchwyty oczekiwania, 903
    - wyliczeń, 121, 267
      - na łańcuch, 268
      - na typy całkowitoliczbowe, 267
      - z liczb całkowitych, 268
    - znaków, 52
  - konwertery typów, 245, 260
  - kopiowanie
    - płytkie, 302, 307
    - tablicy, 302
  - kotwice, 1000
  - kowariancja, 133, 146
  - krotki, tuples, 269
    - porównywanie, 270
  - kryptografia, 864
  - kultura niezmienna, 254
  - kultury, 762
  - kwalifikatory aliasów, 84
  - kwantyfikatory, 390, 429, 998, 1009
    - \*, 998
    - leniwy, 999
    - zachłanny, 999
- ## L
- LAN, Local Area Network, 663
  - leksykon języka wyrażań regularnych, 1008
  - leniwa inicjalizacja, 906
  - liczby, 262
    - klasa Math, 262
    - klasa Random, 265
    - struktura BigInteger, 264
    - struktura Complex, 264
  - licznik wydajności, 544
    - odczyt danych, 546
    - tworzenie, 547
    - wyświetlanie, 545
    - zapis danych, 547
  - LIFO, last in, first out, 109, 313
  - likwidacja
    - domen aplikacji, 961
    - obiektów szyfrowania, 870
    - uchwyty oczekiwania, 899
  - LINQ, Language Integrated Query, 25, 208, 335
    - operatory, 387
  - LINQ to SQL, 368
  - LINQ to XML, 433
    - DOM, 434
  - lista, 297, 308
    - kontroli dostępu, ACL, 861

literały, 32  
całkowitoliczbowe, 42  
reprezentujące liczby  
rzeczywiste, 43  
lokalizacja, 261

## ł

łańcuch  
formatu, 53, 238, 246  
null, 221  
łańcuchowe wywoływanie metod,  
181  
łańcuchy  
dzielenie, 223  
formatowania daty i godziny,  
254, 255  
formatowania wycień, 257  
formatowanie, 244  
klasa `StringBuilder`, 228  
kodowanie tekstu, 229  
łączenie, 223  
łączenie dekoratorów, 351  
łączenie operatorów zapytań,  
337  
mechanizmy konwersji, 257  
metoda `string.Format`, 224  
metody statyczne, 221  
modyfikowanie, 223  
niestandardowe formatu, 250  
niestandardowe numeryczne  
formatu, 252  
numeryczne formatu, 250  
parsowanie, 244  
pobieranie znaków, 222  
połączenia z jednostką, 371  
porównywanie, 225, 393  
puste, 221  
przeszukiwanie, 222  
standardowe formatu, 250  
tworzenie, 220  
zapisywanie deklaracji, 452  
znaków, 51  
łączenie  
dekoratorów, 351  
łańcuchów, 223  
operatorów zapytań, 337  
sekwencji, 389

łącznik zadań, 603, 605  
łączność operatorów  
lewostronna, 67  
prawostronna, 67

## M

magazyn danych, 611  
komputera, 654  
kontenery, 655  
lokalny, 654  
odczyt i zapis, 655  
odizolowany, 650, 653  
podlegający roamingowi,  
654  
położenie, 657  
sprawdzenie dostępności,  
658  
strumieni, 617  
makro, 468  
manifest  
aplikacji .NET, 737–740  
podzespołu, 737, 738  
mapowanie  
plików w pamięci, 650, 651  
struktury, 981  
mechanizm  
cookies, 682  
konwersji, 257  
usuwania nieużytków, 485,  
492, 498–502  
`Close`, 485  
`Dispose`, 485  
dostrajanie, 503  
działanie, 498  
`IDisposable`, 485  
wymuszenie działania, 502  
metadane, 20  
składowych, 783  
metoda, 18, 28, 88  
`Abort`, 911  
`AddUsings()`, 1028  
`Aggregate`, 427  
`All`, 430  
`Any`, 430  
`AppendText()`, 629  
`Array.ConvertAll`, 307  
`Array.Sort`, 306  
`AsEnumerable`, 423  
`AsParallel`, 922  
`Assert`, 842  
`Average`, 426  
`BinarySearch`, 304  
`Cancel()`, 599, 600  
`Cast`, 421  
`Clone`, 307  
`Close()`, 487  
`CodeAccessPermission`.  
↳ `RevertAssert`, 859  
`Combine()`, 643  
`ComputeHash`, 865, 866  
`Concat`, 419  
`Connect()`, 692  
`ConstrainedCopy`, 307  
`Consume`, 957  
`Contains`, 430  
`ContinueWith`, 943  
`Contract.Assert()`, 529  
`Contract.Assume()`, 530  
`Contract.EndContractBlock()`,  
526  
`Contract.Ensures()`, 527  
`Contract.EnsuresOnThrow`,  
528  
`Contract.OldValue<T>`, 529  
`Contract.Requires()`, 523  
`Contract.Requires<T>Excepti  
on>`, 525  
`Contract.Result<T>`  
`Contract.ValueAtReturn<T>`,  
528  
`ConvertTime`, 241  
`Count`, 425  
`CreateDomain`, 859  
`CreateFileQueryWith`  
↳ `Options()`, 648  
`CreateText()`, 629  
`CreateViewAccessor()`, 652  
`DeepClone()`, 707, 709  
`DefaultIfEmpty`, 425  
`DefineMethodOverride`, 811  
`DefineType`, 806  
`Delay()`, 599  
`Delete()`, 638  
`Demand`, 841, 842  
`DescendantNodes()`, 1020



DisplayPrimeCounts(), 581  
 Dispose(), 486, 490, 495  
 Distinct, 394  
 DoCallBack, 964  
 Empty, 431  
 Enqueue, 956  
 EnsureInitialized, 908  
 EqualityComparer<T>.  
     ↳Default, 330  
 Equals, 274, 276, 279  
 Except, 420  
 ExceptWith, 315  
 ExecuteAssembly(), 771  
 FindAll, 305  
 FindAssembly(), 771  
 First, 424  
 FirstNode, 441  
 Flatten, 949  
 FlushFinalBlock, 870  
 Foo(), 599, 811  
 FormatOperand, 821  
 GC.ReRegisterForFinalize(),  
     497  
 get, 95  
 GetAccessControl(), 640  
 GetAsync(), 672  
 GetAwaiter(), 574  
 GetBankFee(), 882  
 GetBytes, 866  
 GetData, 910  
 GetDiagnostics(), 1017  
 GetEnumerator, 294  
 GetEnumerator(), 658  
 GetFileAsync(), 649  
 GetFooValue, 832  
 GetHashCode, 278, 319  
 GetHostAddresses(), 690  
 GetHostEntry(), 690  
 GetKeyForItem, 325  
 GetManifestResource  
     ↳Names(), 756  
 GetMembers, 780, 781  
 GetObjectData(), 724  
 GetPrimesCount(), 588  
 GetResult(), 574  
 GetType, 111  
 Go(), 592  
 GroupBy, 416  
 GroupJoin, 406, 409  
 Handle, 950  
 Increment, 881  
 IndexOf, 305  
 IndexOf/LastIndex, 304  
 Interrupt, 911  
 Intersect, 420  
 IsEquivalentTo(), 1017  
 IsMatch(), 994  
 Join, 406  
 Join(), 554  
 Kind(), 1017  
 Last, 424  
 LastIndexOf, 305  
 LastNode, 441  
 Load, 436  
 LoadFile(), 765, 766, 769  
 LoadFrom(), 765, 766, 769,  
     771  
 LongCount, 425  
 Matches(), 994  
 Max, 426  
 Min, 426  
 Monitor.Enter, 879, 880  
 Monitor.Exit, 879  
 Move(), 638  
 MoveNext, 290  
 Nodes, 441  
 object.Equals, 274, 332  
 Object.Equals, 273  
 object.ReferenceEquals, 275  
 OnCompleted(), 574  
 OpenText(), 629  
 OperationCompleted(), 595  
 OperationStarted(), 595  
 Parallel.For, 934  
 Parallel.ForEach, 934, 935  
 Parallel.Invoke, 933  
 Parse, 245, 436  
 ParseText(), 1018  
 PrintAnswerToLife(), 591  
 Queryable, 423  
 Range, 431  
 ReadElementString, 472  
 ReadString, 472  
 ReadSubtree, 472  
 ReadToDescendant, 472  
 ReadToFollowing, 472  
 ReadToNextSibling, 472  
 ReadXml, 477  
 ReadXml(), 735  
 Regex.Match(), 994  
 Regex.Replace(), 1003  
 Register(), 600  
 Repeat, 431  
 Replace(), 1026  
 Resume, 912  
 Save, 807  
 SaveChanges, 379  
 Select, 395  
 SelectMany, 399, 402, 403, 404  
 SendAsync(), 672, 674  
 SequenceEqual, 430  
 set, 95  
 SetAccessControl(), 640  
 SetData, 910  
 SetValue, 303  
 SignalAndWait, 904, 905  
 SignHash, 874  
 Single, 424  
 Skip, 393  
 SkipWhile, 394  
 Stop(), 487  
 string.Format, 224  
 SubmitChanges, 379  
 Sum, 426  
 Suspend, 912  
 SymmetricExceptWith, 315  
 Take, 393  
 TakeWhile, 394  
 Task.Delay(), 577  
 Task.Run(), 570  
 Task.WhenAny(), 603  
 ToArray, 422  
 ToDictionary, 422  
 ToList, 422  
 ToLookup, 422  
 ToString, 112, 238, 245, 437  
 ToString(), 1017  
 ToXmlString, 873  
 TryEnter, 880  
 TryGetSwitch, 287  
 TryParse, 245  
 TryTake, 952  
 Union, 419  
 UnionWith, 315

- metoda
    - Wait(), 570
    - WaitAll, 904, 942
    - WaitAny, 904
    - WhenAll(), 604
    - Where, 391
    - WindowFromPoint, 854
    - WithChangedText(), 1026
    - WriteAttributes(), 613
    - WriteLine(), 629
    - WriteValue, 475
    - WriteXml(), 735
  - metody
    - abstrakcyjne, 531
    - agregacji, 390
    - agregacyjne, 425
    - anonimowe, 157
    - asynchroniczne w WinRT, 594
    - częściowe, 26, 100
    - definicja, 100
    - implementacja, 100
    - destrukcyjne, 315
    - do kategoryzacji znaków, 221
    - dostępowe, 93, 95
    - dostępowe zdarzenia, 148
    - dostępowe zdarzeń, 152
    - dynamiczne, 801
    - egzemplarzowe, 181
    - egzemplarzy, 142
    - generujące, 430
    - generyczne, 128, 790
    - definiowanie, 814
    - pobieranie, 787
    - wywoływanie, 787
    - instancji, 803, 810
    - inwariantów obiektu, 520, 529, 530
    - klasy Array, 304
    - klasy DynamicObject, 834
    - klasy Expression, 384
    - klasy Math, 263
    - klasy TimeZoneInfo, 241
    - konwersji, 420
    - krytyczne
      - bezpieczne, 850
      - pod względem zabezpieczeń, 849
      - ze względu na wydajność, 849
    - niebezpieczne, 853
    - obiektu AssemblyName, 747
    - OfType, 421
    - przeciążanie, 89, 108
    - przeszukiwania binarnego, 305
    - ReadContentAsXXX, 471
    - ReadXXX, 471
    - rozszerzające, 180
      - łańcuchowe wywoływanie, 181
      - metody egzemplarzowe, 181
    - rozszerzeń, 25, 339
    - skrótów, 619
    - sortujące, 306
    - statyczne, 142
    - wczytujące, 471
    - wirtualne, 273
    - wtyczek, 140
    - wyrażeń, 89
  - miejsca wywołania, 826
  - MMC, Microsoft Management Console, 544
  - model
    - COM, 21
    - kontraktu danych, 701
    - obiektowy Reflection.Emit, 807
    - programowania
      - asynchronicznego, 606
    - semantyczny, 1030
    - transparentności, 848, 849, 850
    - wykonawczy PLINQ, 921
  - moduł sprawdzania pisowni, 924
  - moduły, 740, 793
    - współpracujące, 990
  - modyfikator
    - async, 583
    - internal, 114
    - out, 62
    - params, 63
    - private, 114
    - protected, 114
    - protected internal, 114
    - public, 114
  - readonly, 88
  - ref, 61
  - modyfikatory
    - dostępu, 114
    - zdarzeń, 153
  - modyfikowanie
    - drzewa X-DOM, 444
    - łańcuchów, 223
    - węzłów atrybutów, 445
    - węzłów potomnych, 445
    - domen aplikacji, 965
    - dziennika zdarzeń, 543
  - MSMQ, 216
  - multiemisja, 141
  - MVC, Model-View-Controller, 212
- ## N
- nadawca, 147
  - nadpisane metody, 527, 529
  - nadsubskrypcja procesora, 568
  - nakładanie blokad, 558
  - narzędzia
    - do generowania jednostek, 375
    - Visual Studio, 761
  - narzędzie signtool.exe, 750
  - natywne biblioteki DLL, 973
  - nawigacja, 440
    - do rodzica, 443
    - na tym samym poziomie, 444
    - po atrybutach, 444
  - nazwa
    - elementów, 714
    - kontraktu danych, 705
    - pliku, 619
    - podzespołów, 745
    - symbolu, 1036
    - typów generycznych, 776
    - typów osadzonych, 776
    - typów parametrów, 777
    - wskaźników, 777
  - niebezpieczny kod, 194
  - niegeneryczny słownik, 320
  - niejawne
    - określanie typów, 65
    - parametry ref, 988
    - typowane zmienne lokalne, 25

- niepowodzenie, 515
  - nieskończoność, 48
  - niestandardowe
    - komparatory równości, 419
    - łańcuchy formatu, 250
  - numeryczne łańcuchy formatu, 250
- ## 0
- obiekt, 58
    - AssemblyName, 746
    - CollectionDataContract, 714
    - CredentialCache, 677
    - HttpClient, 680
    - MailMessage, 691
    - MemberInfo, 781
    - szyfrowania, 870
    - typu DateTime, 235
    - typu DateTimeOffset, 236
    - WebClient, 678
    - WebRequest, 678
    - WebResponse, 678
  - obiektość, 17
  - obiekty
    - dynamiczne, 833
    - klasy Encoding, 230, 231
    - niezmienne, 891
    - potomne, 729
  - obliczanie skrótów, hashing, 864–866
  - obsługa
    - łańcuchów i tekstu, 219
    - wyjątków, 561, 678, 805
    - zdarzeń, 587
  - oczekiwanie, 583
    - na interfejs użytkownika, 585
    - na zadania, 903
  - odbiorca, 188
  - odczyt
    - danych licznika wydajności, 546
    - elementów, 469
    - plików .resources, 758
  - odizolowany magazyn danych, 653
  - odwołania
    - do bibliotek DLL, 973
    - do obiektu, 708
    - do składowych, 202
    - do typów, 202
    - wsteczne, 1011
  - ograniczanie
    - dostępności, 115
    - innego zestawu, 856
  - ograniczenia
    - dotyczące klasy bazowej, 130
    - dotyczące klasy i struktury, 131
    - dotyczące konstruktora bezparametrowego, 131
    - dotyczące typu nagiego, 131
    - kolekcja blokująca, 954
    - modyfikatorów dostępu, 116
    - PLINQ, 924
    - typów generycznych, 130
  - określanie obiektu stanu, 940
  - opakowanie, 327
    - zapytań, 358
  - opcja Baseline, 537
  - opcje
    - wyrażeń regularnych, 996, 1011
    - zmiennej statycznej, 512
  - operacje
    - asynchroniczne, 577
    - na plikach i katalogach, 637
    - synchroniczne, 577
    - wejścia-wyjścia, 209, 648, 844
  - operator, 33, 65, 68–70
    - &, 174
    - !=, 273
    - |, 174
    - <, 281, 283
    - ==, 273, 276
    - >, 283
    - as, 103
    - AsEnumerable, 367
    - AsQueryable, 383
    - is, 104
    - Join, 407
    - mnożenia, 29
    - nameof, 100
    - null, 70, 175
    - OrderBy, 414
    - OrderByDescending, 414
    - ThenBy, 414
    - ThenByDescending, 414
    - trójargumentowy, 51
    - typeof, 111, 129
    - warunkowy, 51
    - warunkowy null, 23, 70
    - wskaźnika do składowej, 196
    - zapytania, 335
  - operatory
    - agregacji, 343
    - arytmetyczne, 45
    - bitowe, 47
    - elementów, 342, 390, 423
    - inkrementacji i dekrementacji, 45
    - konwersji, 348
    - LINQ, 387
    - porównywania i równości, 49, 173
    - przypisania, 67
    - relacyjne, 174
    - sprawdzania przepełnienia całkowitoliczbowego, 46
    - warunkowe, 50
    - wyliczeń, 123
    - zbiorów, 390, 419
    - zmieniające kształt, 389
  - opóźnienie podpisania, 744
  - optymalizacja, 499, 596
    - PLINQ, 928
    - własnych agregacji, 931
    - z wartościami lokalnymi, 938
  - osadzanie typów współpracujących, 990
  - ostrzeżenia pragma, 199
- ## P
- pakiet NuGet, 1013
  - pakowanie, 110, 120
  - pamięć, 40
    - lokalna wątku, 909
    - niezarządzana, 981
    - współdzielona, 651, 978
  - paralelizm strukturalny, 918
  - parametr, 28, 57, 60
    - lockTaken, 880
    - out, 787
    - ref, 787, 988

- parametry
  - atrybutu, 192
  - metod, 786
  - nazwane, 192
  - opcjonalne, 24, 63, 92, 987
  - pozycyjne, 192
  - typów, 126, 128, 132
- parsowanie, 244, 255
  - argumentów, 801
  - IL, 819
  - liczb, 258
  - przez dostawcę formatu, 248
- pary zastępcze, 231
- PCL, Portable Class Libraries, 19
- pewność przypisania, 59
- pętla
  - do-while, 76
  - for, 76
  - foreach, 77
  - while, 76
- pętle
  - wewnętrzne, 935
  - zewnętrzne, 935
- PFX, 917
- PIA, primary interop assembly, 990
- piaskownica, 22
- pieczętowanie funkcji i klas, 106
- pierwszy program, 27
- planista
  - domyślny, 947
  - kontekstu synchronizacji, 947
  - zadań, 947
- planowanie zadań, 947
- platforma
  - .NET Framework, 19, 205
  - .NET Core, 19
- plik, 649
  - .resources, 757
  - .resx, 757
  - signtool.exe, 750
  - XSLT, 483
- pliki
  - .edmx, 370
  - .pdb, 540
  - .tlb, 991
  - .winmd, 737
  - metadanych, 22
  - ZIP, 636
- plikowe operacje
  - wejścia-wyjścia, 648, 650
- PLINQ, 918, 920
  - anulowanie zapytania, 927
  - kolejność elementów, 923
  - ograniczenia, 924
  - optymalizacja, 928
  - wykonywanie równoległe, 922
  - zastosowania, 926
- płytkie kopiowanie, 302
- pobieranie
  - atrybutów w czasie działania, 797
  - elementów, 441
  - elementów potomnych, 442
  - jednego elementu, 442
  - metadanych składowych, 783
  - metod generycznych, 787
  - typów osadzonych, 774
  - typów tablicowych, 775
  - wartości wyliczenia, 268
  - znaków, 222
- poczta elektroniczna
  - otrzymywanie, 695
  - wysyłanie, 691
- podklasy
  - obiektów potomnych, 731
  - typu głównego, 730
- podmiot zabezpieczeń, 862
- podnoszenie uprawnień, 846, 862
- podpisy cyfrowe, 874
- podpisywanie kodu, 750
- podstawowa przestrzeń
  - wielojęzyczna, BMP, 231
- podstawowe zestawy
  - międzyoperacyjne, 25
- podwyrażenia, 1002
- podzapytania, 353, 397
- podzespoły, 737, 738
  - atrybuty, 739
  - emitowanie, 805
  - ładowanie, 793
  - manifest aplikacji, 739
  - manifest podzespołu, 738
  - moduły, 740, 793
  - nazwa, 745
  - nazwa kwalifikowana, 746
  - opóźnienie podpisania, 744
  - podpisywanie, 742
  - refleksje, 792
  - repozytorium GAC, 753
  - satelickie, 754, 760
  - silne nazwy, 743
  - ustalanie, 763
  - użycie Authenticode, 749
  - w pojedynczym pliku, 769
  - wczytywanie, 762, 764
  - wdrażanie, 768
  - wersja informacyjna, 747
  - współpracujące COM, 986
  - wyszukiwanie, 762
  - zapisywanie, 807
- podział tekstu, 1004
- pola egzemplarzowe, 910
- pole, 35, 87
- polecenie, *Patrz instrukcja*
- polimorfizm, 101
  - jednokierunkowy, 831
  - wielokierunkowy, 831
- ponawianie zgłoszenia wyjątku, 163
- ponowne obliczanie, 348
- POP, Post Office Protocol, 663
- POP3, 695
- poprawa wydajności, 788
- poprawność pliku XML, 481
- porównywanie
  - krotek, 270
  - łańcuchów, 53, 225, 383
  - kulturowe, 225
  - pod względem kolejności, 227
  - pod względem równości, 226
  - porządkowe, 225, 282
- porty
  - TCP, 664
  - UDP, 664
- porządkowanie, 413
  - naturalne, 342
- pośrednik, 327

- potoki
    - anonimowe, 622, 624
    - nazwane, 622, 623
  - poziom zaufania, 846
  - pożyczanie operatorów, 173
  - praca zdalna, Remoting, 968
    - międzyprocesowa, 968
  - prefiksy adresów URI, 670
  - procesy, 539
  - profil referencyjny, 21
  - programowanie
    - asynchroniczne, 578, 579, 606
    - dynamiczne, 210, 825
    - funkcyjne, 18
    - równoległe, 211, 551, 917, 920
  - programy
    - typu klient, 552
    - wielowątkowe, 552
  - progresywne budowanie
    - zapytań, 356
  - projekcja, 394
    - do typów konkretnych, 398
    - do X-DOM, 459
    - z indeksowaniem, 396
  - protokoły
    - dołączane, 273
    - porządkowania, 328
    - równości, 273, 328
  - protokół
    - HTTP, 662, 680
    - MIDI, 978
    - POP3, 695
    - SetLastError, 980
    - TCP, 664, 692, 697
    - UDP, 664
  - przechwytywanie
    - stanu lokalnego, 584
    - zdarzeń, 647
    - zmiennych, 349, 560
    - zmiennych iteracyjnych, 156
    - zmiennych zewnętrznych, 155
  - przeciążanie
    - konstruktorów, 90
    - metod, 89, 108
    - operatorów, 177, 279
      - false, 179
    - porównywania, 178
  - równości, 178
    - true, 179
  - przedrostek, 454, 457
  - przeglądanie słownika, 319
  - przekazywanie
    - danych, 672
    - danych formularza, 681
    - przez referencję, 61, 89
    - przez wartość, 60, 89
    - nadmierne, 598
    - stanu, 837
    - wyjątku, 595
  - przeliczalność, 289
  - przeliczanie, 347
  - przełączanie kontekstu, 555
  - przepełnienie
    - całkowitoliczbowe, 45
  - przerwanie operacji, 598
  - przesłanie metody, 278, 279
  - przestrzenie nazw w XML, 453
  - przestrzeń nazw, 29, 79
    - aliasy, 83
    - globalna, 80
    - import, 83
    - kwalifikatory aliasów, 84
    - powtarzanie, 82
    - kontraktu danych, 706
    - System.ComponentModel, 566
    - System.Cryptography, 868
    - System.Diagnostics, 209
    - System.Reflection, 741
    - System.Reflection.Emit, 807
    - System.Runtime.Serialization, 703
    - System.Text, 208
    - System.Text.Regular
      - ↳ Expressions, 993
    - System.Xml, 465właściwości
      - zaawansowane, 83
    - zakres, 81
  - przeszukiwanie
    - drzewa, 1019
    - łańcuchów, 222
    - tablic, 304
  - przetwarzanie tekstu, 208
  - przewidywanie, 999
    - negatywne, 1000
    - pozytywne, 1000
    - wsteczne, 999
  - przypisywanie
    - atributów, 813
    - ról, 863
    - użytkowników, 863
    - wielu atrybutów, 192
  - przyrostki literałów liczbowych, 43
  - przysłanie metod, 810
  - pula wątków, 567
  - punkty kontrolne, 538
- ## R
- RCW, runtime-callable wrappers, 986
  - rdzeń platformy, 207
  - receptury wyrażeń regularnych, 1005
  - refaktoryzacja, 28
  - referencja this, 92
  - refleksje, 210, 773, 780
    - atributów, 798
    - dla podzespołów, 792
    - składowych, 781
  - reguły asynchroniczności, 577
  - reimplementacja interfejsu, 118
  - rekurencja blokowania, 897
  - Remoting, 217
  - repozytorium GAC, 752, 753
  - reprezentacja typu dynamic, 186
  - responsywny interfejs
    - użytkownika, 551
  - REST, REpresentational State Transfer, 663
  - rodzaje
    - kodowania tekstu, 229
    - serializatorów, 704
    - uprawnień, 840
    - węzłów, 475
  - role, 862
  - Roslyn
    - architektura, 1014
    - drzewa składni, 1015
    - kompilacja, 1030
    - model semantyczny, 1030
    - przestrzenie robocze, 1014

- rozgałęzianie, 803
  - rozpakowywanie, 110
  - rozpoznawanie, 108
  - rozszerzenie interfejsu, 117
  - rozszerzenie kontraktu danych, 715
  - równoległe wykonywanie zadań, 939
  - równoległość, 592
  - równość, 279
    - referencyjna, 272, 273
    - refleksyjna, 276
    - strukturalna, 272
    - wartościowa, 272, 273
  - równoważenie obciążenia, 930
  - równoważność typów, 990
  - rzutowanie, 37, 102, 679
    - w dół, 103
    - w górę, 102
- S**
- sekcja CDATA, 468
  - sekwencje, 171, 389
    - dekoracyjne, 350
    - filtrowanie, 389
    - grupowanie, 390
    - lokalne, 335
    - łączenie, 389
    - porządkowanie, 389
    - projekcja, 389
    - specjalne, 51
    - wyjściowe, 335
    - zagnieżdżone, 389
  - selektywne egzekwowanie kontraktów, 534
  - semafory, 892
  - semantyka iteratorów, 168
  - serializacja, 210, 437, 699
    - atributy, 720
    - binarna, 701, 718
    - jawna, 703
    - kolekcji, 732
    - kontraktu danych, 701, 703
    - niejawna, 703
    - obiektów potomnych, 730
    - odwołań do obiektów, 708
    - podklas, 707
    - tworzenie podklas, 726
    - typów generycznych, 722
    - XML, 727
    - za pomocą ISerializable, 724
    - zaczepy, 715
  - serializator
    - DataContractSerializer, 704
    - NetDataContractSerializer, 704
  - serializator kontraktu danych, 717
  - serwer
    - FTP, 688
    - HTTP, 685
    - POP3, 695
    - proxy, 675
    - SMTP, 691
  - serwerowe środowisko uruchomieniowe, 501
  - sieć, 209, 661
  - silne nazwy, 742
  - silnik XML, 702
  - silniki serializacji, 700
  - Silverlight, 214
  - sklep Windows Store, 650
  - skład zapytania interpretowanego, 365
  - składnia, 30
    - płynna, 337, 346
    - SQL, 346
    - zapytaniowa, 344, 346
  - składniki
    - PFX, 919
    - platformy .NET, 864
  - składowe
    - abstrakcyjne, 105
    - C#, 784
    - CLR, 784
    - egzemplarza, 35
    - funkcyjne, 18
    - interfejsu, 116
    - interfejsu generycznego, 790
    - klasy Assembly, 742
    - klasy object, 112
    - klasy Stream, 613
    - klasy TextReader, 628
    - klasy TextWriter, 628
    - niepubliczne, 788
    - odziedziczone, 105
    - prywatne, 36
    - styczne, 35, 889
    - typów generycznych, 785
    - typu, 35
  - skompilowane typy, 737
  - skrót, 278
  - skrypt, 837
  - słabe odwołania, 507, 508
  - słowa kluczowe, 30
    - kontekstowe, 32
  - słowa zarezerwowane, 30
  - słowniki, 316
    - sortowane, 321
  - słowo kluczowe
    - async, 582
    - async i await, 582
    - await, 583
    - base, 107, 108
    - Component, 759
    - dynamic, 827
    - extern, 83
    - fixed, 984
    - into, 358
    - let, 361
    - new, 106, 803
    - override, 106
    - public, 36
    - ref, 988
    - stackalloc, 196
    - static, 827
    - using, 1027
    - var, 65
    - virtual, 104
    - volatile, 983
  - SMTP, Simple Mail Transfer Protocol, 663, 691
  - sortowanie, 306, 415
  - spinning, 555
  - sprawdzanie
    - granic tablic, 57
    - modelu semantycznego, 1032
    - poprawności dokumentu, 481
    - poprawności drzewa X-DOM, 483
    - poprawności schematów, 480

- poziomu zaufania, 846
  - przepelnienia, 46
  - równości, 271, 277
  - sposobu kodowania, 230
  - typów, 971
  - dostępnych liczników, 545
  - kontraktu, 536
    - w miejscu wywołania, 535
  - SSL, 685
  - stała, 33, 96
  - stan
    - lokalny, 556, 584
    - współdzielony, 556
  - standardowe
    - łańcuchy formatu, 250
    - operatory zapytań, 335
  - standardowy wzorzec zdarzeń, 149
  - statyczna kontrola typów, 18, 111
  - statyczne sprawdzenie
    - kontraktu, 536
  - sterta, 58
    - ogromnych obiektów, 500
  - stos, 57, 313
    - ewaluacji, 800
    - LIFO, 313
  - stosowanie
    - blokady, 881
    - zasad dostępu kodu, 845
  - strategie
    - dziedziczenia, 370
    - projekcji, 360
  - strefy czasowe, 239, 242
  - struktura, 113
    - BigInteger, 264
    - Complex, 264
    - DateTime, 233, 239, 243
    - DateTimeOffset, 233, 239
    - DOM wyrażenia, 383
    - drzewa, 1019
    - Guid, 271
    - Nullable<T>, 172
    - SyntaxToken, 1015
    - SyntaxTree, 1015
    - SyntaxTrivia, 1016
    - TextSpan, 1022
    - TimeSpan, 232
  - strukturalna równość
    - wartościowa, 277
  - strukturalne zrównoleglenie
    - przetwarzania danych, 918
  - strumienie, 209
    - adapter, 626
    - architektura, 611
    - bezpieczeństwo wątków, 617
    - dekoracyjne, 612
    - kompresja, 634
    - limit czasu, 617
    - magazynu danych, 612, 617
    - obsługa odczytu i zapisu, 615
    - opróżnienie, 616
    - wyszukiwanie, 616
    - zamknięcie, 616
    - zamykanie adapterów, 633
  - strumieniowanie projekcji, 461
  - strumień
    - BufferedStream, 625
    - FileStream, 618
    - MemoryStream, 621
    - PipeStream, 622
    - XmlReader, 467
  - subkultury, 762
  - subskrybent, 147
  - surogat, 231
  - sygnalizacja, 878, 897, 898
  - sygnały dwustronne, 899
  - sygnatura metody, 88
  - sygnatury Func, 341
  - symbole, 1032
    - zadeklarowane, 1034
  - symetria typów
    - predefiniowanych, 35
  - symulowanie unii C, 977
  - synchronizacja, 594, 878, 880
  - system
    - CLR, 20
    - plików
      - CDFS, 640
      - FAT, 640
      - NTFS, 640
      - przechwytywanie
        - zdarzeń, 647
    - typów COM, 985
  - szeregowanie
    - In i Out, 976
    - klas i struktur, 975
    - typów wspólnych, 974
  - szzyfrowanie, 844
    - kluczem publicznym, 871
    - skrótów, 864
    - SSL, 685
    - symetryczne, 867
    - w pamięci, 868
    - wiadomości, 872
- ## Ś
- ściśła kontrola typów, 19
  - śledzenie obiektów, 373
  - środowisko
    - uruchomieniowe CLR, 501
    - wykonawcze systemu Windows, 21
- ## T
- tabela operatorów, 67
  - tablice, 53, 135, 196
    - bajtów, 231
    - długość, 304
    - indeksowanie, 302
    - konwertowanie, 307
    - kopiowanie, 307
    - liczba wymiarów, 304
    - nieregularne, 55
    - odwracanie kolejności
      - elementów, 307
    - prostokątne, 55
    - przeglądanie zawartości, 304
    - przeszukiwanie, 304
    - skrótów, hash tables, 278, 319
    - sortowanie, 306
    - tworzenie, 302
    - w pamięci, 301
    - wielowymiarowe, 55
    - zmienianie rozmiarów, 307
  - TAP, Task-based Asynchronous Pattern, 602
  - TCP, Transmission and Control Protocol, 663, 692, 697



- techniki
  - optymalizacji, 499
  - synchronizacji, 878
  - wielowątkowości, 877
- technologia
  - .ASMX Web Services, 217
  - ADO.NET, 214
  - ASP.NET, 212
  - Authenticode, 748
  - COM+, 216
  - EF, 379
  - L2S, 379
  - LINQ, 25, 208, 335
  - MSMQ, 216
  - PLINQ, 918
  - Silverlight, 214
  - WCF, 216
  - WPF, 213
  - XML, 465
- technologie
  - interfejsu użytkownika, 212
  - zapleczone, 214
- tekst, 208
- testowanie, 262
  - podzespołów satelickich, 761
- testy jednostkowe, 673
- token anulowania, 927
- tożsamości, 862
- transformacja drzewa składni, 1026
- transparentność, 848, 855
- tryb pliku, 619
- tworzenie
  - asercji, 515
  - deasemblera, 819
  - domen aplikacji, 961
  - drobiazgow, 1027
  - drzewa X-DOM, 437
  - egzemplarzy, 35
  - FileStream, 618
  - funkcji asynchronicznych, 589
  - instancji obiektów, 803
  - instancji typów, 778
  - kompilacji, 1030
  - liczników wydajności, 547
  - łańcuchów, 220
  - łańcuchów strumieni
    - szyfrowania, 869
  - obiektów typu DateTime, 235
  - obiektów typu
    - DateTimeOffset, 236
  - pliku .resx, 758
  - podklas typów generycznych, 131
  - podzespołu satelickiego, 760
  - serwera HTTP, 685
  - struktur, 113
  - tablic, 302
  - tokenów, 1027
  - typów, 87
  - uchwytu EventWaitHandle, 902
  - wątku, 552
  - węzłów, 1027
  - własnych fabryk zadań, 948
  - wyrażeń lambda, 340
  - wyrażeń zapytań, 357
  - zadań, 940
  - zapytań złożonych, 356
  - zasobu pack URI, 759
  - zbioru, 315
  - złączeń, 407
- typ, 33
  - bool, 174
  - char, 219
  - CultureInfo, 247
  - DateTimeFormatInfo, 247
  - decimal, 48
  - double, 47, 48
  - dynamic, 186, 187
  - float, 47
  - NumberFormatInfo, 247
  - object, 109, 186
  - string, 52, 220
  - TSource, 341
  - var, 187
  - wyliczeniowy
    - Environment.SpecialFolder, 644
    - RegexOptions, 995
    - UnmanagedType, 974
    - WebExceptionStatus, 678
- typizowanie elementów, 341
- typy
  - anonimowe, 25, 182, 360
  - argumenty, 126
  - bazowe, 777
  - całkowitoliczbowe
    - 16-bitowe, 47
    - 8-bitowe, 47
  - częściowe, 99
  - definiowane przez programistę, 277
  - delegacyjne, 139, 145
  - dopuszczające wartość null, 171
    - alternatywa, 176
    - konwersje jawne, 172
    - konwersje niejawne, 172
    - mieszanie operatorów, 174
    - operatory null, 175
    - pakowanie wartości, 172
    - rozpakowywanie wartości, 172
    - zastosowania, 175
  - generyczne, 125, 276, 776, 779
    - dane statyczne, 132
    - definiowanie, 815
    - delegacyjne, 143
    - niezwiązane, 129, 780
    - odwołania do samego siebie, 132
    - ograniczenia, 130
    - podklasy, 131
    - szablony C++, 137
    - zamknięte, 780, 816
  - izolacji, 653
  - logiczne, 49
  - liczbowe, 42, 827
  - osadzone, 775
  - otwarte, 126
  - parametry, 126, 128
  - platformy .NET Framework, 887
  - predefiniowane, 34
  - proste, 41
  - referencyjne, 37, 39, 41, 54
  - składowych, 782
  - statyczne, 189



systemowe, 207  
tablicowe, 135, 775  
wartościowe, 37, 38, 41, 54  
węzłów, 1017  
własne, 35  
wskaźnikowe, 194  
współpracujące, 990  
współpracujące COM, 986  
wyjątków, 164  
wyrażeń, 384  
X-DOM, 435  
zagnieżdżone, 124  
zamknięte, 126  
zwrotne, 28, 139

## U

UAC, User Account Control, 860  
uchwyt EventWaitHandle, 902  
uchwyty zdarzeń oczekiwania,  
897, 899, 902  
udostępnianie obiektów, 991  
UDP, Universal Datagram  
Protocol, 663  
ukończenie synchroniczne, 596  
ukrywanie  
nazw, 81  
odziedziczonych  
składowych, 105  
UNC, Universal Naming  
Convention, 663  
unia, 978  
Unicode, 229  
unifikacja, 109  
typów liczbowych, 827  
unikanie konfliktów nazw, 32  
uprawnienia, 839  
diagnostyczne, 844  
dla operacji wejścia i wyjścia,  
844  
dostępu kodu, 839  
dotyczące interfejsu  
użytkownika, 844  
dotyczące szyfrowania, 844  
dotyczące tożsamości, 845  
podstawowe, 843  
sieciowe, 844  
UIPermission, 854

URI, Uniform Resource  
Identifier, 663  
URL, Uniform Resource Locator,  
663  
uruchamianie zadań, 570, 940  
usługa, 370  
ustalenie typu, 763  
ustawianie stopnia  
zrównoleglenia, 927  
ustawienia przezroczystości, 852  
usuwanie  
elementów składowych, 490  
nieużytków, 485, 498  
automatyczne, 491  
Close, 485  
Dispose, 485  
IDisposable, 485  
obiektów, 373  
sekwencji atrybutów, 446  
sekwencji węzłów, 446  
UTF-16, 231  
uwierzytelnianie, 676  
na podstawie formularzy,  
684  
HttpClient, 678  
nagłówki, 678  
uzgadnianie klucza publicznego,  
872  
uzyskanie drzewa składni, 1018  
używanie  
Authenticode, 749  
biblioteki PFX, 920  
delegatów, 140  
delegatu multitemisji, 142  
DNS, 690  
FTP, 688  
klamer, 73  
klasy XmlWriter, 480  
kontraktów kodu, 519, 532  
PLINQ, 926  
punktów kontrolnych, 538  
serializatorów, 704  
strumieni, 613  
TCP, 692  
uchwytów zdarzeń  
oczekiwania, 897

## V

Voice over IP, 692

## W

wariancja, 134  
parametrów, 146  
typów, 24  
warstwy sekwencji dekoratorów,  
351  
wartości, 447  
domyślne, 59  
pobieranie, 448  
specjalne typów liczbowych,  
47  
ustawianie, 447  
zwrotne, 571  
wartość  
NaN, 48  
Null, 40, 171, 238, 712  
skrótów, hash code, 278  
warunki  
kontraktu  
wyjątki, 534  
końcowe, 527, 529  
początkowe, 523, 527  
wątek, 211, 552, 965  
bezpieczeństwo, 558  
blokowanie, 555  
dołączanie, 554  
priorytet, 563  
przekazywanie danych, 559  
sygnalizowanie, 564  
usypianie, 554  
wywłaszczenie, 553  
wątki  
działające w tle, 562  
aktywne, 562  
interfejsu użytkownika, 566  
procesów, 539  
pula, 567  
w aplikacjach WPF, 564  
wątkowanie, 552  
WCF, Windows Communication  
Foundation, 216  
wcześnie ładowanie, 378  
w Entity Framework, 379

- wczytywanie
  - atrybutów, 472
  - podzespołu, 764
  - węzłów, 467
- wdrażanie podzespołów, 768
- Web API, 217
- wektor inicjalizujący, IV, 868
- wersjonowanie, 723, 753
- weryfikacja Authenticode, 751
- węzły
  - atrybutów, 473
  - potomne, 441
  - XML, 467
  - XText, 449
  - z treścią mieszaną, 449
- wiązanie
  - dynamiczne, 24, 183, 989
  - językowe, 185
  - niestandardowe, 185
  - statyczne, 184
- wielowątkowość, 877
- Windows Data Protection, 864
- Windows Forms, 213
- Windows RT i Xamarin, 214
- Windows Store, 650
- Windows Workflow, 215
- WinRT, 21, 493
- wirtualizacja, 862
- wirtualne składowe funkcyjne, 104
- własne
  - atrybuty, 796
  - fabryki zadań, 948
  - liczniki wydajności, 548
  - łączniki zadań, 605
- własności, 18, 29, 93
  - automatyczne, 26, 94
  - klasy System.Exception, 164
  - obliczane, 94
  - otoczenia, 175
  - tylko do odczytu, 94
  - wyrazeniowe, 94
- właściwości adresu URI, 665
- właściwość
  - CodeBase, 767
  - IsGenericType, 780
  - IsGenericTypeDefinition, 780
  - Location, 767
  - MethodHandle, 782
  - Span, 1017
  - SyntaxTree, 1017
- WPF, Windows Presentation Foundation, 213
- wpisywanie
  - atrybutów, 475
  - węzłów, 475
- wskaznik, 777
  - do kodu niezarządzanego, 197
  - do składowej, 196
  - pusty, 197
- wskrzeszenie, 496
- współbieżność, 209, 551
  - drobnoziarnista, 579
  - gruboziarnista, 579, 588
  - w TCP, 694
- współpraca COM, 985
- wtyczka, 140
- wyбір
  - obiektu synchronizacji, 880
  - trybu pliku, 620
- wychodzenie z pętli, 936
- wyciek
  - blokady, 880
  - pamięci zarządzanej, 503
- wydajność, 788, 885, 901
- wyjatek, 158, 534, 572, 805
  - AggregateException, 949
  - ContractException, 522, 525
  - FormatException, 245
  - niezaobserwowany, 572
  - NullReferenceException, 274, 461
  - RuntimeBinderException, 186
  - TypeLoadException, 818
  - UnauthorizedAccess
    - ↳Exception, 861
  - WebException, 678
  - XmlException, 469
- wykonywanie
  - leniwe, 348
  - opóźnione, 347, 350, 356, 376
  - równoległe, 922
  - spekulatywne, 552
  - zapytania, 352
- wykorzystanie delegatów, 788
- wyliczenia, 121, 166
  - atrybut Flags, 122
  - konwersje, 121, 267
  - operatory, 123
- wyliczenie
  - BindingFlags, 789
  - DateStyles, 256
  - TaskCreationOptions, 941
- wymagania modelu
  - transparentności, 852
- wynik obliczeń, 28
- wyrażenia, 27, 65
  - dynamiczne, 188
  - inicjalizacji tablicy, 54, 56
  - lambda, 18, 25, 153, 340, 560, 593
  - jawne określanie typów, 154
  - przechwytywanie zmiennych zewnętrzných, 155
- podstawowe, 66
- przypisania, 66
- puste, 66
- regularne, 993
  - alternatywy, 1011
  - asercje, 999, 1010
  - granica słowa, 1001
  - grupy, 1002
  - kategorie znaków, 1009
  - kompilowane, 995
  - konstrukcje grupujące, 1010
  - konstrukcje różne, 1011
  - kotwice, 1000
  - kwantyfikatory, 1009
  - nazwane grupy, 1003
  - odwołania wsteczne, 1011
  - opcje, 996, 1011
  - receptury, 1005–1008
  - zastąpienia, 1010
  - zbiory znaków, 1009
  - zestawy znaków, 997
  - znaki sterujące, 996, 1008
- stałe, 66
- typy statyczne, 189
- zapytań, 18, 25, 337, 343, 357, 381

- wysyłanie zapytań, 440
  - wyszukiwanie
    - elementu potomnego, 1021
    - podzespółów, 762
    - symboli, 1036
  - wyświetlenie podpisu
    - Authenticode, 751
  - wywołania
    - anonimowe, 831
    - asynchroniczne, 591
    - dynamiczne, 188
    - komponentu COM, 986
    - konstruktorów bazowych, 813
    - metod generycznych, 787
    - metod instancji, 803
    - składowych, 780, 786
    - składowych interfejsu generycznego, 790
    - zwrotne, 977
  - wzorce
    - asynchroniczności, 598
    - oparte na zadaniu, 602
    - oparte na zdarzeniach, 607
    - przestarzałe, 606
  - wzorzec
    - metod TryXXX, 165
    - UnsafeXXX, 851
    - wizytator, 828
- X**
- X-DOM, 434
    - automatyczne głębokie klonowanie, 439
    - definiowanie przestrzeni nazw, 455
    - domyślne przestrzenie nazw, 456
    - klasa XmlReader, 479
    - klasa XmlWriter, 479
    - konstrukcja funkcyjna, 438
    - ładowanie, 436
    - modyfikowanie drzewa, 444
    - nawigacja do rodzica, 443
    - nawigacja na tym samym poziomie, 444
    - nawigacja po atrybutach, 444
  - nawigowanie, 440
  - parsowanie, 436
  - przekształcanie drzewa, 462
  - serializacja, 437
  - sprawdzanie poprawności, 483
  - tworzenie drzewa, 437
  - wysyłanie zapytań, 440
  - zapisywanie, 437
  - XML, 208, 465
    - dokumentacja, 200
  - XSD, XML Schema Definition, 480
  - XSLT, Extensible Stylesheet Language Transformations, 483
- Z**
- zabezpieczenia
    - dostępu kodu, 843
    - niebezpiecznych metod, 853
    - systemu operacyjnego, 860
  - zaczepy serializacji, 715
  - zadania, 569
    - autonomiczne, 572
    - anulowanie, 942
    - długo wykonywane, 571
    - kontynuacje, 943, 945
    - kontynuacje na jednym przodku, 947
    - kontynuacje warunkowe, 945
    - kontynuacje z wieloma przodkami, 947
    - planowanie, 947
    - potomne, 941, 945
    - równoległe wykonywanie, 939
    - sposób wykorzystania, 956
    - tworzenie, 940
    - uruchamianie, 570, 940
    - zimne, 956
  - zadanie Task<TResult>, 944
  - zagnieżdżanie
    - blokad, 883
    - dyrektywy using, 82
  - zakleszczenia, 884
  - zależności cykliczne, 817
  - zamykanie
    - adapterów, 633
    - egzemplarzy nasłuchujących, 518
  - zapisywanie deklaracji, 452
  - zapytania, 208
    - EF, 376
    - interpretowane, 362, 364
    - L2S, 376
    - LINQ, 335
    - o składni mieszanej, 347
    - złożone, 356
  - zarządzanie
    - kluczami, 871
    - pamięcią, 19
  - zasady
    - dostępu kodu, 845
    - zabezpieczeń, 855
  - zasoby, 755
  - zasób pack URI, 759
  - zastąpienia, 1010
  - zastępowanie tekstu, 1003
  - zastosowania
    - biblioteki PFX, 920
    - typów dopuszczających wartość null, 175
    - klasy XmlReader, 476
    - klasy XmlWriter, 476
  - zbiory, 314, 419
    - znaków, 1009
  - zdarzenia, 29, 147, 508
    - metody dostępowe, 152
    - modyfikatory, 153
    - wzorzec standardowy, 149
  - zdarzenia systemu plików, 647
  - zdarzenie, 18
    - AssemblyResolve, 764, 769
    - ContractFailed, 533
    - EntryWritten, 543
    - IncludeSubdirectories, 647
  - zegary, 505, 913
    - jednowątkowe, 915
    - wielowątkowe, 913
  - zestaw referencyjny, 22
  - zestawy, 210
    - zaprzyjaźnione, 115
    - znaków, 997

- zgłaszanie wyjątków, 162
- zgodność
  - delegatów, 145
  - parametrów, 145
  - typów, 145
  - typów zwrotnych, 146
- złączenia, 397, 406
  - krzyżowe, 407
  - nierównościowe, 407
  - płaskie zewnętrzne, 411
  - w składni płynnej, 409
  - według wielu kluczy, 409
  - wewnętrzne, 407
  - z widokami wyszukiwania,
    - 411
  - zewnętrzne, 404
  - zewnętrzne lewe, 407
- złożone
  - łańcuchy formatu, 224
  - operatory przypisania, 66
- zmienianie
  - definicji równości, 277
  - nazwy symbolu, 1036
  - ścieżki wykonywania, 73
- zmiennie, 33, 57
  - iteracyjne, 156
  - lokalne, 28, 65, 72, 557, 802
  - zakresowe, 344, 345, 358, 401
  - zewnętrzne, 155
- znaczniki
  - dokumentacyjne XML, 201
  - niestandardowe, 202
- znaki
  - interpunkcyjne, 32
  - podwójne, 231
  - sterujące, 996, 1008
- znakowanie czasowe, 750
- zrównoleganie, 927
  - przetwarzania danych, 918
  - wykonywania zadań, 918
- zużycie pamięci, 492
- zwalnianie zasobów, 485–488
  - na żądanie, 489
- zwrot egzemplarza, 590

## Ż

żądanie uprawnień, 857

# PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



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

Zmień swoją stronę WWW  
w działający bankomat!

**Dowiedz się więcej i dołącz już dzisiaj!**

<http://program-partnerski.helion.pl>

GRUPA WYDAWNICZA

 **Helion SA**



## C# jest szybki, efektywny, wygodny – to narzędzie w sam raz dla Ciebie!

C# jest jednym z najlepszych projektów firmy Microsoft. Język ten został od podstaw zaprojektowany jako obiektowy. Charakteryzuje się niezwykłą elastycznością i wszechstronnością. Udostępnia wysokopoziomowe abstrakcje, takie jak wyrażenia, zapytania i kontynuacje asynchroniczne, a równocześnie pozwala na korzystanie z niskopoziomowych mechanizmów, jak własne typy wartościowe programisty czy opcjonalne wskaźniki. C# w wersji 6.0 jest kolejną istotną aktualizacją języka. Programista piszący w C# powinien konsekwentnie poznawać te zmiany.

Niniejsza książka jest zwięzłym kompendium wiedzy o C#, CLR oraz o związanej z C# platformie. Opracowano je z myślą o programistach na co najmniej średnim poziomie zaawansowania. W zrozumiały, a równocześnie dogłębny sposób wyjaśniono takie trudne kwestie, jak współbieżność, bezpieczeństwo i domeny aplikacji. Informacje o nowych składnikach języka C# 6.0 i związanej z nim platformy zostały specjalnie oznaczone. Szczególnie istotny z punktu widzenia programisty może okazać się rozdział o nowym kompilatorze Roslyn, zwanym kompilatorem usługowym.

**Joseph Albahari** – jest autorem kilku książek dotyczących C# oraz LINQ. Napisał też LINQPad, popularny program do testowania zapytań LINQ.

**Ben Albahari** – był kierownikiem programowy w Microsoftzie; współtworzył takie projekty, jak .NET Compact Framework i ADO.NET. Jeden z założycieli firmy Genamics zajmującej się produkcją narzędzi dla programistów C# i J++ oraz oprogramowania do analizy DNA i sekwencjonowania białek. Jest autorem i współautorem kilku książek dotyczących C#.

### Najważniejsze zagadnienia ujęte w książce:

- składnia, typy oraz zmienne C#
- bezpieczeństwo kodu i dyrektywy preprocesora
- rdzenne technologie i techniki platformy .NET Framework, w tym LINQ, XML, kolekcje, kontrakty kodu, zarządzanie pamięcią, refleksja, programowanie dynamiczne
- kompilator Roslyn – jego architektura, struktura drzewa składni i model semantyczny

# Helion

księgarnia internetowa

<http://helion.pl>

zamówienia telefoniczne



0 801 339900



0 601 339900

Helion SA  
ul. Kościuszki 1c, 44-100 Gliwice  
tel.: 32 230 98 63  
e-mail: [helion@helion.pl](mailto:helion@helion.pl)  
<http://helion.pl>

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

ISBN 978-83-283-2423-7



9 788328 324237

cena: 129,00 zł

sięgnij po WIĘCEJ



KOD KORZYŚCI