

O'REILLY®



C# 7.0

Leksykon
kieszonkowy

Hellon 

Joseph Albahari
Ben Albahari

Tytuł oryginału: C# 7.0 Pocket Reference: Instant Help for C# 7.0 Programmers

Tłumaczenie: Przemysław Szeremiota

ISBN: 978-83-283-4077-0

© 2018 Helion SA

Authorized Polish translation of the English edition of C# 7.0 Pocket Reference
ISBN 9781491988534 © 2017 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

WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

<http://helion.pl/user/opinie/ch7lek>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

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

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

Spis treści

Konwencje typograficzne	5
Korzystanie z przykładowych programów	6
Pierwszy program w C#	7
Składnia	10
System typów	13
Typy liczbowe	22
Typ wartości logicznych i operatory logiczne	28
Znaki i ciągi znaków	30
Tablice	34
Zmienne i parametry	38
Operatory i wyrażenia	45
Operatory na typach z dopuszczalną wartością pustą	50
Instrukcje	52
Przestrzenie nazw	60
Klasy	64
Dziedziczenie	77
Typ object	85
Struktury	89
Modyfikatory dostępu	90
Interfejsy	92
Typy wyliczeniowe	95
Typy zagnieżdżone	97
Uogólnienia	98
Delegaty	105
Zdarzenia	111
Wyrażenia lambda	116
Wyrażenia lambda a metody lokalne	119
Metody anonimowe	120
Wyjątki i instrukcja try	121
Enumeratory i iteratory	129

Typy z dopuszczalną wartością pustą	134
Metody rozszerzające	138
Typy anonimowe	140
Krotki (C# 7)	141
LINQ	143
Wiązanie dynamiczne	165
Przeciążanie operatorów	172
Atrybuty	175
Atrybuty wywołania	179
Funkcje asynchroniczne	180
Wskaźniki i kod nienadzorowany	188
Dyrektywy preprocesora	192
Dokumentacja XML	194
Skorowidz	199

Wiązanie dynamiczne

Wiązanie dynamiczne oznacza przesunięcie momentu *wiązania* — procesu ustalania typów, składowych i wywołań — od czasu kompilacji do czasu wykonania programu. Wiązanie dynamiczne zostało wprowadzone do języka C# w wersji 4.0; ma zastosowanie, kiedy *programista* wie, że pewna metoda, składowa czy operacja istnieje, ale *kompilator* nie ma o niej informacji. Do takich sytuacji dochodzi często w ramach interoperacji z językami dynamicznymi (jak IronPython) i obiektami COM, a także w sytuacjach typowych dla zastosowań mechanizmów refleksji.

Typ dynamiczny jest deklarowany z kontekstowym słowem kluczowym `dynamic`:

```
dynamic d = GetSomeObject();  
d.Quack();
```

Typ dynamiczny nakazuje kompilatorowi rozluźnić kontrolę typów; programista oczekuje, że w czasie wykonania zmienna typu `d` będzie posiadała metodę `Quack`. Jedyną trudność w tym, że w czasie kompilacji nie można tego potwierdzić. Ponieważ `d` jest zmienną typu dynamicznego, kompilator opóźni wiązanie wywołania metody `Quack` na rzecz `d` do czasu wykonania programu. Aby lepiej zrozumieć, co to oznacza, należałoby zdefiniować rozróżnienie pomiędzy *wiązaniem statycznym* i *dynamicznym*.

Wiązanie statyczne a wiązanie dynamiczne

Klasycznym przykładem wiązania jest odwzorowanie nazwy występującej w kompilowanym wyrażeniu na konkretną metodę czy składową. Na przykład w poniższym wyrażeniu kompilator musi odszukać implementację metody o nazwie `Quack`:

```
d.Quack();
```

Załóżmy, że statyczny typ `d` to `Duck`:

```
Duck d = ...  
d.Quack();
```

W najprostszym przypadku kompilator dokonuje wiązania poprzez wyszukanie bezparametrowej metody `Quack` w klasie `Duck`. Jeśli to się nie powiedzie, kompilator rozszerzy poszukiwania na metody z parametrami opcjonalnymi, metody typów bazowych, a wreszcie metody rozszerzające, które przyjmują zmienną typu `Duck` w miejsce pierwszego argumentu wywołania. Jeśli i to nie doprowadzi do dopasowania wywołania do metody, kompilator zgłosi błąd kompilacji. Jak widać, niezależnie od tego, jaka metoda zostanie ostatecznie wybrana do realizacji wywołania, wybór ten jest dokonywany przez kompilator, a samo dopasowanie wywołania i metody bazuje wyłącznie na informacji dostępnej statycznie, to znaczy na widocznych w kodzie definicjach typów operandów (tutaj `d`). Tyle o *wiązaniu statycznym*.

Zmieńmy teraz statyczny typ zmiennej `d` na typ `object`:

```
object d = ...  
d.Quack();
```

Wywołanie `Quack` spowoduje teraz błąd kompilacji, ponieważ choć wartość przechowywana w `d` może zawierać metodę `Quack`, kompilator nie może jej znać, gdyż jedyną informacją, jaką dysponuje, jest najogólniejszy z możliwych typ zmiennej `object`. Co innego, jeśli typ `d` zostanie określony jako dynamiczny:

```
dynamic d = ...  
d.Quack();
```

Typ dynamiczny jest trochę jak typ `object`, to znaczy równie mało mówi o zmiennej `d`. Różnica polega na tym, że typ dynamiczny może być używany w sposób nieweryfikowalny w czasie kompilacji. Operacje na obiekcie deklarowanym jako `dynamic` będą rozstrzygane w oparciu o typ obiektu ustalony w czasie wykonania programu, a nie typ z czasu kompilacji. Kompilator, napotkawszy wyrażenie wiązane dynamicznie (czyli wyrażenie, w którym występuje dowolna wartość typu dynamicznego), ogranicza się jedynie do takiego obrobienia wyrażenia, żeby wiązanie mogło zostać rozstrzygnięte w czasie wykonania programu.

W czasie wykonania, jeśli okaże się, że obiekt dynamiczny implementuje interfejs `IDynamicMetaObjectProvider`, to właśnie ten interfejs zostanie wykorzystany do rozstrzygnięcia dopasowania; w innym przypadku środowisko wykonawcze rozstrzygnie wywołanie analogicznie, jak zrobiłby to kompilator w czasie kompilacji, to znaczy przeszuka aktualny typ obiektu, jego typy bazowe oraz metody rozszerzające. Rozstrzygnięcie w oparciu o implementację interfejsu `IDynamicMetaObject`

↳ Binding nazywamy *wiązaniem ze wskazania* albo *wiązaniem niestandardowym* (ang. *custom binding*), a rozstrzygnięcie w oparciu o dostępną w czasie wykonania wiedzę o typie i jego typach bazowych nazywamy *wiązaniem według reguł* albo *wiązaniem językowym* (ang. *language binding*).

Wiązanie ze wskazania

Wiązanie ze wskazania dotyczy przypadku, kiedy obiekt typu dynamicznego implementuje interfejs `IDynamicMetaObjectProvider` (IDMOP). Interfejs ten można co prawda implementować w typach pisanych w języku C#, ale najczęściej dotyczy to obiektów pozyskiwanych z innych języków dynamicznych z rodziny .NET, operujących w ramach środowiska wykonawczego `Dynamic Language Runtime` (DLR) (jak `IronPython` czy `IronRuby`). Obiekty pochodzące z tych języków niejawnie implementują interfejs IDMOP jako sposób bezpośredniego kontrolowania znaczenia operacji, które będą na tych obiektach wykonywane. Oto prosty przykład:

```
using System;
using System.Dynamic;

public class Test
{
    static void Main()
    {
        dynamic d = new Duck();
        d.Quack(); // wywołano Quack
        d.Waddle(); // wywołano Waddle
    }
}

public class Duck : DynamicObject
{
    public override bool TryInvokeMember (
        InvokeMemberBinder binder, object[] args,
        out object result)
    {
        Console.WriteLine ("wywołano " + binder.Name);
        result = null;
        return true;
    }
}
```

Klasa `Duck` w istocie nie posiada metody `Quack`, a jedynie wskazanie wiązania, dzięki któremu przechwytuje i interpretuje wywołania wszystkich metod na rzecz obiektów tej klasy.

Wiązania ze wskazania są omawiane szerzej w 20. rozdziale książki *C# 7.0 in a Nutshell*.

Wiązanie językowe

Z wiązaniem językowym mamy do czynienia w przypadku obiektów dynamicznych, które nie implementują interfejsu `IDynamicMetaObjectProvider`. Wiązanie językowe stosuje się przede wszystkim do obchodzenia niedogodności niedoskonale zaprojektowanych typów, a także niektórych ograniczeń właściwych systemowi typów platformy .NET. Typowym problemem jest na przykład brak jednolitego interfejsu typów liczbowych (wiemy już, że wiązanie dynamiczne dotyczy metod; dopowiedzmy, że dotyczy również operatorów):

```
static dynamic Mean (dynamic x, dynamic y) => (x + y) / 2;

static void Main()
{
    int x = 3, y = 4;
    Console.WriteLine (Mean (x, y));
}
```

W takich przypadkach zalety wiązania dynamicznego są oczywiste, bo nie trzeba powielać kodu wspólnych operacji obliczeniowych dla każdego z rozmaitych typów liczbowych. Wadą jest natomiast utrata statycznej kontroli typów, a więc i ryzyko zamiany błędów kompilacji na wyjątki czasu wykonania.

Uwaga

Wiązanie dynamiczne oznacza osłabienie kontroli typów, ale tylko statycznej — kontrola dynamiczna wciąż jest obecna. Inaczej niż w mechanizmach refleksji, wiązanie dynamiczne nie pozwala np. na ominięcie reguł widoczności składowych.

Dynamiczne wiązanie językowe celowo ma możliwie blisko odwzorowywać wiązanie statyczne (a więc działa tak, jak działałby kompilator, gdyby tylko typ dynamiczny był mu znany statycznie). Zachowanie programu z poprzedniego przykładu byłoby identyczne, gdybyśmy jawnie oprogramowali metodę `Mean` do operowania na typie `int`. Najbardziej znamieną różnicą pomiędzy wiązaniem dynamicznym i statycznym dotyczy metod rozszerzających; będzie o tym mowa w punkcie „Funkcje niedynamiczne”.

Uwaga

Wiązanie dynamiczne wiąże się z narzutem wydajności wykonania programu, ale dzięki mechanizmom cachowania DLR powtarzające się wykonania tych samych wyrażeń wiązanych dynamicznie są skutecznie optymalizowane, co pozwala stosunkowo wydajnie wykonywać wywołania dynamiczne na przykład w pętlach. Optymalizacja redukuje narzut rozstrzygnięcia prostego wyrażenia dynamicznego do około 100 ns (mowa o współczesnym komputerze).

Wyjątek `RuntimeBinderException`

Jeśli składowej nie uda się związać dynamicznie, środowisko wykonawcze zgłosi wyjątek `RuntimeBinderException`, będący odpowiednikiem błędu czasu kompilacji, ale wykrytym dopiero w czasie wykonania:

```
dynamic d = 5;
d.Hello(); // wywołanie zgłosi wyjątek RuntimeBinderException
```

Wyjątek zgłoszony w powyższym kodzie oznacza, że typ dynamiczny (tu `int`) nie posiada metody `Hello`.

Reprezentacja typu dynamicznego

Typ dynamiczny cechuje się znacznym podobieństwem do typu `object` — środowisko traktuje na przykład poniższe porównanie jako prawdziwe:

```
typeof (dynamic) == typeof (object)
```

Zasada ta rozciąga się również na typy konkretyzowane typem dynamicznym i typy tablicowe:

```
typeof (List<dynamic>) == typeof (List<object>)
typeof (dynamic[]) == typeof (object[])
```

Tak jak w przypadku referencji `object`, referencja `dynamic` może odnosić się do obiektu dowolnego typu (za wyjątkiem typów wskaźnikowych):

```
dynamic x = "ahoj";
Console.WriteLine (x.GetType().Name); // String

x = 123; // nie ma błędu (choć to ta sama zmienna)
Console.WriteLine (x.GetType().Name); // Int32
```

Pomiędzy referencjami `object` i referencjami `dynamic` strukturalnie nie ma żadnej różnicy. Referencja `dynamic` pozwala po prostu na dynamiczne wiązanie operacji na obiekcie, do którego się odnosi. Można nawet przekonwertować obiekt `object` na typ `dynamic` i następnie wykonać na nim dowolną operację dynamiczną:

```
object o = new System.Text.StringBuilder();
dynamic d = o;
d.Append ("ahoj");
Console.WriteLine (o); // ahøj
```

Konwersje typów dynamicznych

Typ dynamiczny daje się niejawnie skonwertować na dowolny inny typ, a także dowolny inny typ można niejawnie konwertować na typ dynamiczny. Aby jednak konwersja była skuteczna, właściwy typ obiektu dynamicznego musi faktycznie dać się skonwertować na docelowy typ statyczny.

Poniższy kod spowoduje wyjątek `RuntimeBinderException`, ponieważ typ `int` nie daje się niejawnie konwertować na typ `short`:

```
int i = 7;
dynamic d = i;
long l = d; // OK — niejawna konwersja działa
short j = d; // wyjątek RuntimeBinderException
```

var a dynamic

Typy `var` i `dynamic` wydają się podobne, ale w istocie zasadniczo się różnią:

- `var` oznacza: „niech *kompilator* określi faktyczny typ”;
- `dynamic` oznacza: „niech *środowisko wykonawcze* określi faktyczny typ”.

Spójrzmy:

```
dynamic x = "ahoj"; // statyczny typ to dynamic
var y = "ahoj"; // statyczny typ to string
int i = x; // wyjątek wykonania
int j = y; // błąd kompilacji
```

Wyrażenia dynamiczne

Wiązanie dynamiczne może dotyczyć pól, właściwości, metod, zdarzeń, konstruktorów, indeksów, operatorów i konwersji.

Nie można pobrać wyniku wyrażenia dynamicznego z typem zwracanym `void`; podobnie jest w przypadku wyrażeń statycznych. Różnica dotyczy momentu wystąpienia błędu (tutaj w czasie wykonania programu).

Wyrażenia dynamiczne to najczęściej takie wyrażenia, w których uczestniczą dynamiczne operandy, a to dlatego, że najczęściej niedostępność statycznej informacji o typie daje efekt kaskadowy:

```
dynamic x = 2;
var y = x * 3; // statyczny typ y to dynamic
```

Od tej reguły jest kilka oczywistych wyjątków. Po pierwsze, rzutowanie typu dynamicznego na typ statyczny daje wyrażenie statyczne. Po drugie, wywołanie konstruktora zawsze daje wyrażenie statyczne, nawet jeśli argumenty wywołania mają typy dynamiczne.

Istnieje też kilka przypadków brzegowych, w których wyrażenie zawierające argument dynamiczny samo jest statyczne; zaliczymy tu przekazanie indeksu do tablicy i wyrażenia tworzenia delegatów.

Rozstrzygnięcie przeciążeń składowych dynamicznych

Klasycznym przykładem użycia wiązania dynamicznego jest dynamiczny *odbiornik komunikatów*, to znaczy obiekt dynamiczny jako podmiot dynamicznego wywołania metody:

```
dynamic x = ...;
x.Foo (123); // x "odbiera" wywołania
```

Jednak zakres zastosowań typów dynamicznych jest szerszy i obejmuje także dynamiczne wiązanie argumentów wywołania metody, gdzie rozstrzygnięcie wywołania z dynamicznymi argumentami jest przesuwane z czasu kompilacji do czasu wykonania:

```
static void Foo (int x) => Console.WriteLine ("1");
static void Foo (string x) => Console.WriteLine ("2");
static void Main()
{
    dynamic x = 5;
    dynamic y = "arbuz";

    Foo (x); // 1
    Foo (y); // 2
}
```

Rozstrzygnięcie przeciążenia w czasie wykonania jest również określane mianem wielorozprowadzania albo *multimetody* (ang. *multiple dispatch*), a stosuje się je między innymi w implementacjach wzorca projektowego Wizytator.

W przypadku dynamicznych argumentów wywołania metody (ale bez dynamicznego podmiotu wywołania) kompilator może przeprowadzić podstawowe sprawdziany skuteczności wywołania dynamicznego: sprawdzane jest więc choćby istnienie funkcji z odpowiednią nazwą i liczbą parametrów. Jeśli takiej metody nie ma, wiadomo, że rozstrzygnięcie dynamiczne również będzie nieudane, więc kompilator zgłasza błąd kompilacji.

W przypadku wywołań z argumentami typów dynamicznych oraz statycznych ostateczny wybór metody będzie odzwierciedlał mieszaninę decyzji podjętych statycznie i dynamicznie:

```
static void X(object x, object y) => Console.Write("oo");
static void X(object x, string y) => Console.Write("os");
static void X(string x, object y) => Console.Write("so");
static void X(string x, string y) => Console.Write("ss");
static void Main()
{
    object o = "ahoj";
    dynamic d = "pa, pa";
    X (o, d); // os
}
```

Wywołanie $\lambda(o,d)$ jest wiązane dynamicznie, ponieważ jeden z jego argumentów ma typ dynamic. Ale skoro typ argumentu o jest statyczny, już w czasie kompilacji wiadomo, że w grę wchodzi jedynie dwa pierwsze przeciążenia λ . Ostatecznie wybór padnie na drugie z nich, a to ze względu na dokładnie dopasowany statyczny typ o i dokładnie dopasowany dynamiczny typ d . Innymi słowy, kompilator usiłuje maksymalnie wykorzystać statyczną informację o typach nawet w przypadku wyrażeń dynamicznych.

Funkcje niedynamiczne

Niektórych funkcji nie można wywołać dynamicznie. Dotyczy to:

- metod rozszerzających (ze składnią metod rozszerzających),
- dowolnych składowych interfejsu (w przypadku wywołań przez interfejs),
- składowych typów bazowych przykrytych typami pochodnymi.

Niemożność wynika z tego, że wiązanie dynamiczne potrzebuje dwóch informacji: nazwy metody do wywołania oraz obiektu, na rzecz którego ma się odbyć wywołanie. Ale w powyższych przypadkach uczestniczy *dotatkowy typ*, znany wyłącznie w czasie kompilacji. Język C# nie daje możliwości dynamicznego określenia dodatkowego typu.

W przypadku wywołania metody rozszerzającej tym dodatkowym typem jest klasa rozszerzająca, wybierana niejawnie na bazie dyrektyw `using` widocznych w kodzie źródłowym (a więc widocznych tylko w czasie kompilacji). Przy wywołaniach za pośrednictwem interfejsu dodatkowy typ jest komunikowany jawną albo niejawną konwersją (w przypadku implementacji jawnej nie da się wywołać składowej bez rzutowania na typ interfejsu). Tak samo w przypadku przykrywania składowych klasy bazowej: dodatkowy typ musi być określony albo poprzez rzutowanie, albo poprzez słowo kluczowe `base`, a oba są widoczne wyłącznie w czasie kompilacji.

Przeciążanie operatorów

Przeciążanie operatorów ma pozwalać na definiowanie bardziej naturalnej składni operacji na własnych typach danych. Przeciążanie operatorów jest najbardziej odpowiednie w definiowanych przez programistę strukturach reprezentujących stosunkowo proste, pierwotne typy danych. Na przykład znakomitym kandydatem do przeciążenia operatorów jest własny typ liczbowy.

Programista może przeciążać następujące operatory:

```
+ - * / ++ -- ! ~ % & | ^  
== != < << >> >
```

Możliwe jest również przesłanianie jawnych i niejawnych konwersji (ze słowem kluczowym `implicit` lub `explicit`), a także literalów `true` i `false` oraz jednoargumentowych operatorów `+` i `-`.

Operatory z przypisaniem (czyli np. `+=` czy `/=`) są przesłaniane automatycznie, jeśli przesłonięte zostały odpowiadające im operatory składowe (np. `+` bądź `/`).

Funkcje operatorów

Przeciążanie operatora odbywa się za pomocą deklaracji tak zwanej *funkcji operatora*. Funkcja operatora musi być statyczna, a przynajmniej jeden z operandów powinien być typu, dla którego przeciążamy operator.

W poniższym przykładzie definiujemy strukturę o nazwie `Note`, reprezentującą nutę muzyczną, i przeciążamy dla niej operator `+`:

```
public struct Note
{
    int value;
    public Note (int semitonesFromA)
        => value = semitonesFromA;

    public static Note operator + (Note x, int semitones)
    {
        return new Note (x.value + semitones);
    }
}
```

Takie przeciążenie pozwoli nam w poniższym kodzie bezproblemowo dodać do obiektu `Note` wartość typu `int`:

```
Note B = new Note();
Note CSharp = B + 2;
```

Powyższe przeciążenie operatora `+` pozwoli nam także na stosowanie obiektów `Note` w wyrażeniach z operatorem `+=`:

```
CSharp += 2;
```

Począwszy od C# 6, przeciążenia ograniczające się do pojedynczego wyrażenia mogą mieć zapis skrócony do wyrażenia (jak metody i właściwości):

```
public static Note operator + (Note x, int semitones)
    => new Note (x.value + semitones);
```

Przeciążanie operatorów porównania i relacji

Częstą praktyką jest również przeciążanie operatorów porównania i relacji, głównie dla struktur, znacznie rzadziej dla klas. Przy przeciążaniu tych operatorów obowiązują nas specjalne dodatkowe reguły i wymogi:

Parowanie

Kompilator języka C# wymaga, aby w przypadku przeciążania operatorów robić to parami. Takie obowiązkowe pary to: (==, !=), (<, >) oraz (<=, >=).

Przeciążenie Equals oraz GetHashCode

Jeśli przeciążymy operatory == i !=, zazwyczaj powinniśmy również przeciążyć dla danego typu metody Equals i GetHashCode dziedziczone po klasie object, tak aby obiekty typu można było skutecznie stosować w kolekcjach i tablicach mieszających.

Implementacja IComparable oraz IComparable<T>

Po przeciążeniu operatorów < i > implementuje się zazwyczaj dla danego typu również interfejsy IComparable i IComparable<T>.

W ramach rozszerzenia poprzedniego przykładu moglibyśmy uzupełnić typ Note o operatory porównania i relacji:

```
public static bool operator == (Note n1, Note n2)
{
    => n1.value == n2.value;
}
public static bool operator != (Note n1, Note n2)
{
    => n1.value != n2.value;
}
public override bool Equals (object otherNote)
{
    if (!(otherNote is Note)) return false;
    return this == (Note)otherNote;
}
public override int GetHashCode ()
{
    // w roli hasha możemy użyć hasha wartości przekazanej:
    public override int GetHashCode() => value.GetHashCode();
}
```

Własne konwersje jawne i niejawne

Konwersje, zarówno jawne, jak i niejawne, też są operatorami dającymi się przeciążać. Są zazwyczaj przeciążane w celu udostępnienia sposobu konwersji pomiędzy danym typem a innymi typami, silnie z nim powiązanymi (na przykład z typami liczbowymi) — tak aby operacje na tych typach były wygodne i naturalne.

Przy okazji omawiania podstaw systemu typów ustaliliśmy, że niejawne konwersje mają rację bytu, kiedy gwarantują skuteczność konwersji wartości bez utraty informacji. Kiedy to niemożliwe, należy zdefiniować konwersje jawne.

W następnym przykładzie zdefiniujemy konwersję pomiędzy naszym muzycznym typem `Note` a wartością typu `double` (która będzie reprezentować częstotliwość danego tonu):

```
...
// Konwersja na herce
public static implicit operator double (Note x)
{
    => 440 * Math.Pow (2, (double) x.value / 12);
}

// Konwersja z herców (z dokładnością do najbliższego półtonu)
public static explicit operator Note (double x)
{
    => new Note ((int) (0.5 + 12 * Math.Log(x/440) /
    Math.Log(2)) );
}
...
Note n = (Note)554.37;    // Konwersja jawna
double x = n;            // Konwersja niejawna
```

Uwaga

Powyższy przykład można by (zgodnie z wcześniejszymi wytycznymi) zrealizować również poprzez implementację konwersji za pośrednictwem jawnej metody `ToFrequency` i jawnej statycznej metody `FromFrequency`.

Własne konwersje nie są brane pod uwagę w operatorach `as` i `is`.

A

akcesor, 115, 116
właściwości, *Patrz:*
właściwość akcesor
aplikacja, 9
argument, *Patrz też:*
parametr
dynamiczny, 171
nazwany, 44
przekazywanie, 40
przez referencję,
40–42
przez wartość, 40–42
assembly, *Patrz:* zestaw
atrybut, 175
CLSCompliant, 177
obiekt docelowy, 177
ObsoleteAttribute, 176
odwołanie, 178
parametr, 176
nazwany, 176
pozycyjny, 176
własny, 177
wywołanie, 179
XmlElementAttribute,
176
mechanizm, 175
attribute target, *Patrz:*
atrybut obiekt
docelowy

B

biblioteka, 9
tworzenie, 10

blok
catch, 121–123
finally, 121–124
instrukcji, 7, 12, 52
try, 121
błędy zaokrąglania, 28
broadcaster, *Patrz:*
nadawca

C

caller info attribute,
Patrz: atrybut
wywołanie
ciąg
łączenie, 32
porównywanie, 33
przeszukiwanie, 33
znaków, 30, 31
interpolowany, 32
klasa, 9
constructor, *Patrz:*
konstruktor
custom binding, *Patrz:*
wiązanie
niestandardowe

D

dane
składowe, 16
statyczne, 103
wejściowe, 8, 65
wyjściowe, 8
dekonstruktor, 67

delegat, 105, 111
Action, 108
do metody
instancji, 108
statycznej, 108
Func, 108
metody-wtyczki, 106
typ, 105
kontrawariancja,
110
kowariancja, 110
uogólniony, 108
zgodność, 109
uogólniony, 110
wielokrotny, 107, 108
delegaty
modyfikowalność, 107
domknięcie, 118
dyrektywa
#define, 193
#elif, 193
#else, 193
#endif, 193
#endregion, 193
#error, 193
#if, 193
#line, 193
#pragma, 193, 194
#region, 193
#undef, 193
#warning, 193
preprocesora, 192, 193
using, 61
using static, 61
dziedziczenie, 77, 83, 92
po klasie, 77
dzielenie całkowite, 25

E

enumerator, 129
escape sequence, *Patrz:*
znak sterujący
event, *Patrz:* zdarzenie
exception filter, *Patrz:*
wyjątek filtr

F

finalizator, 8, 75, 89
finalizer, *Patrz:* finalizator
flaga, 96
fluent syntax, *Patrz:*
wyrażenie składnia
kaskadowa
fully qualified type name,
Patrz: typ nazwa
w pełni kwalifikowana
funkcja
asynchroniczna, 180,
184, 185
niedynamiczna, 172
przesyłanie, 80
składowa, 15, 82,
Patrz też: metoda
synchroniczna, 184
wirtualna, 80
wywoływanie, 39
asynchroniczne, 180
synchroniczne, 180

G

garbage collector, *Patrz:*
mechanizm
odśmieciania
generic method, *Patrz:*
metoda uogólniona
generic type, *Patrz:*
uogólnienie
grupowanie, 162

I

identyfikator, 10
interfejs, 167
indekser, 8, 72
implementowanie, 73
wirtualny, 81
indexer, *Patrz:* indekser
inferencja, 23
instancja, 15
składowa, 16
typu
wartościowego, 39
instantiation, *Patrz:*
konkretyzacja
instrukcja, 7, 52
blok, *Patrz:* blok
instrukcji
break, 59
continue, 59
deklaracji, 52
do-while, 57
fixed, 189, 190
for, 57
foreach, 35, 57, 129
goto, 59
if, 53
if..else, 53, 54
iteracyjna, *Patrz:*
instrukcja pętli
pętli, 57
return, 59, 132
skoku, 59
switch, 54, 59
throw, 59, 126, 127
try, 121, 124
using, 125
warunkowa, 53
while, 57
wyrażeniowa, 52, *Patrz*
też: wyrażenie
yield, 132
yield break, 132
interfejs, 86, 92
deklaracja, 92
dziedziczenie, 93

IComparable, 174
IDynamicMetaObject
↳Binding, 167
IDynamicMetaObject
↳Provider, 166, 168
IEnumerable, 35, 104,
129–132
IEnumerator, 104, 131
implementacja
jawna, 93
ponowna, 94
wirtualna, 94
INotifyCompletion.
↳OnCompleted,
183
rozszerzanie, 139
składowa, 172
System.Collections.
↳Generic.
↳IEnumerable,
132
System.Collections.
↳Generic.
↳IEnumerator,
129, 132
System.Collections.
↳IEnumerable,
130, 132
System.Collections.
↳IEnumerator,
129, 132
interpolated string, *Patrz:*
ciąg znaków
interpolowany
iterator, 130–132
sekwencja złożona, 133

J

jagged array, *Patrz:* tablica
wyszczerbiona

K

klasa, 9, 64, 92
abstrakcyjna, 81

- bazowa, 78, 82
- Console, 9
- definiowanie, 64
- dziedziczenie, *Patrz:*
 - dziedziczenie po klasie
- Enumerable, 152, 153
- nazwa, 10
- object, 86, 88
- pochodna, 78–83,
 - Patrz też:* podklasa
- statyczna, 75
- string, 34, 73
- System.Array, 35
- System.Console, 75
- System.EventArgs, 113
- System.Exception, 127
- System.Math, 75

klauzula

- inicjalizacji, 58
- iteracji, 58

kod

- nienadzorowany, 189
- XML, 194

kolejka

- LIFO, *Patrz:* stos

kolekcja, 130

komentarz

- dokumentujący, 194
- jednowierszowy, 13
- wielowierszowy, 13

kompilacja

- ręcznie, 10

kompilator, 9, 10, 194

komunikat

- ostrzeżenia, 194
- blokowanie, 194

konkretyzacja, 15

konstruktor, 8, 66, 83

- bezparametrowy, 67, 84, 89
- kolejność, 84
- niepubliczny, 67
- obiektu, 16
- przeciążanie, 66

- stacyczny, 74
- kontrawariancja, 103, 105, 110
- konwencja
 - wielbłądzia, 11
- konwersje
 - jawne, 17, 174
 - liczbowe, 24
 - niejawne, 17, 174
 - własne, 174
- kowariancja, 103, 105, 109, 110
- krotka, 141
 - dekonstrukcja, 142
 - elementy nazwane, 142
 - typ, 141
- kwalifikator, 151
 - global::, 63
- kwantyfikatory, 147, 148

L

- lambda expression, *Patrz:*
 - wyrażenie lambda
- language binding, *Patrz:*
 - wiązanie językowe
- Language Integrated Query, *Patrz:* LINQ
- LINQ, 143, 154
 - element, 143
 - operatory złączeń, 159
 - sekwencja, 143
- lista, 72
- literał, 8, 12
 - ciągu znaków, 32
 - dosłowny, 32
- krotki, 141
- liczb
 - całkowitych, 22
 - rzeczywistych, 22

M

- mechanizm
 - LINQ, *Patrz:* LINQ
 - odśmieciania, 39, 75

- metoda, 7, 15, 65
 - Aggregate, 151
 - All, 151
 - anonimowa, 120
 - Any, 151
 - AsEnumerable, 152
 - AsQueryable, 152
 - Average, 151
 - BinarySearch, 35
 - Cast, 152
 - CompareTo, 33
 - Concat, 151
 - Contains, 151
 - Copy, 35
 - Count, 151
 - CreateInstance, 35
 - częściowa, 76
 - definicja, 76
 - implementacja, 76
 - deklaracja, 40
 - Display, 126
 - Dispose, 125
 - Distinct, 150
 - ElementAt, 151
 - ElementAtOrDefault, 151
 - Empty, 152
 - Equals, 88, 89
 - Except, 151
 - Finalize, 75
 - Find, 35
 - Find LastIndex, 35
 - FindIndex, 35
 - First, 151
 - FirstOrDefault, 151
 - GetHashCode, 89
 - GetLength, 36
 - GetResult, 183
 - GetType, 87
 - GetValue, 35
 - GroupBy, 150
 - GroupJoin, 150
 - IndexOf, 35
 - Insert, 34
 - instancji, 108, 140
 - Intersect, 151

metoda

IsCapitalized, 139
Join, 150, 159
Last, 151
LastIndexOf, 35
LastOrDefault, 151
LongCount, 151
Main, 9
Max, 151
Min, 151
MoveNext, 147
nazwa, 10, 65
OfType, 152
OnComplete, 183
OrderBy, 150, 152
OrderByDescending, 150
PadLeft, 34
PadRight, 34
parametr, *Patrz:*
 parametr
przeciążanie, 65, 84
Range, 152
ReferenceEquals, 89
Remove, 34
Repeat, 152
Reverse, 150
rozszerzająca, 138,
 140, 144, 172
 wywołanie
 kaskadowe, 139
Select, 145, 150, 152
SelectMany, 150
SequenceEqual, 151
SetValue, 35
Single, 151
Skip, 150
SkipWhile, 150
skrótowa do
 wyrażenia, 65
Sort, 35
statyczna, 67
Substring, 34
Sum, 151
sygnatura, 65
Take, 150

TakeWhile, 150
ThenBy, 150
ThenByDescending, 150
ToArray, 152
ToDictionary, 152
ToList, 152
ToLookup, 152
ToLower, 34
ToString, 89
ToUpper, 34
Trim, 34
TrimEnd, 34
TrimStart, 34
Union, 151
uogólniona, 99
wartość
 zwracana, 8
Where, 150, 152
wirtualna, 81
WriteLine, 9
Zip, 150
metody lokalne, 65, 119
modyfikator
 async, 182, 187
 await, 182, 184
 dostępu
 internal, 90, 91
 private, 90
 protected, 90
 protected internal,
 90
 public, 90
 out, 40, 42
 params, 43
 readonly, 64
 ref, 40, 41, 175
 this, 138
 unsafe, 182
 virtual, 175
multimetoda, 171
multiple dispatch, *Patrz:*
 multimetoda

N

nadawca, 112
namespace, *Patrz:*
 przestrzeń nazw
nawiasy klamrowe, 12
nested type, *Patrz:* typ
 zagnieżdżony
null coalescing operator,
 Patrz: operator ??
nullable type, *Patrz:* typ
 z dopuszczalną
 wartością pustą
null-conditional operator,
 Patrz: operator ?.

O

obiekt, 38
 inicjalizator, 68
 instancja, *Patrz:*
 instancja
 konstruktor, *Patrz:*
 konstruktor
 obiektu
 nadzorowany, 189
 przeliczalny, 129
 System.Object, 85
 System.Type, 87
 tablicy, 35
 tworzenie, 38
 typ, *Patrz:* typ
 Type, 101
obliczenie
 nadzorowane, 25
odbiornik komunikatów,
 171
odpakowywanie, 86
ograniczenie
 do klasy bazowej, 102
 do typów
 referencyjnych, 102
operacje na liczbach
 całkowitych, 25
operand, 45

- operator, 8, 12, 45, 47,
 - Patrz też:* znak
 - !=
 - przeciążanie, 89
 - &, 26, 49, 97, 137, 172, 189
 - &&, 29
 - ?, 50
 - ?:, 49
 - ??, 50, 138
 - ^, 26, 49, 97, 137, 172
 - |, 26, 49, 97, 137, 172
 - ||, 29, 49
 - ~, 26, 97, 137
 - +, 32
 - <<, 26
 - ==, 135, 172
 - przeciążanie, 89
 - >>, 26
 - addytywny, 49
 - agregacji, 146, 148
 - All, 147
 - alternatywy dla null, 47
 - Any, 147
 - arytmetyczny, 24, 97
 - as, 49, 80
 - Average, 146
 - await, 48
 - bitowej sumy
 - wyłączającej, 49
 - bitowy, 26, 97
 - Cast, 164
 - checked, 25, 26, 48
 - Concat, 147
 - Contains, 147
 - Count, 146
 - default, 48
 - dekrementacji, 25
 - dostępu przez
 - wskaźnik, 189, 190
 - dwuargumentowy, 46
 - zapis wrostkowy, 46
 - elementowy, 145, 148, 151
 - Elvis, *Patrz:* operator ?.
 - Except, 147
 - First, 145, 146
 - FirstOrDefault, 146
 - funkcja, 173
 - główny, 48
 - GroupBy, 162, 163
 - GroupJoin, 159, 160
 - iloczynu bitowego, 49
 - iloczynu logicznego, 49
 - inkrementacji, 25
 - Intersect, 147
 - is, 49, 80
 - jednoargumentowy, 46, 48
 - Join, 159
 - konwersji, 148
 - lambda, 47, 50
 - Last, 145
 - łączność, 47
 - lewostronna, 47
 - prawostronna, 47
 - Max, 146
 - Min, 146
 - mnożenia, 12
 - multiplikatywny, 49
 - nameof, 48, 76
 - new, 16, 48
 - obliczenia
 - nadzorowanego, *Patrz:* operator checked
 - odwołania do
 - składowej, 12
 - OfType, 164
 - pierwszorzędny, 46
 - pobrania adresu, 189
 - porównania, 49, 88, 97, 136, 173
 - pożyczanie, 135, 136
 - priorytet, 47
 - przeciążanie, 172–174
 - przesunąć bitowych, 49
 - przypisania, 13, 46, 47, 50
 - relacji, 29, 49, 136, 173
 - Reverse, 145
 - równości, 29
 - SelectMany, 157
 - SequenceEquals, 147
 - Single, 145
 - sizeof, 48, 97
 - Skip, 145
 - stackalloc, 48
 - struktury Nullable, 135–137
 - sumy bitowej, 49
 - sumy logicznej, 49
 - Take, 145
 - ThenBy, 162
 - ToArray, 148
 - ToDictionary, 148
 - ToList, 148
 - ToLookup, 148
 - trójargumentowy, 46
 - typeof, 48, 101
 - typu wyliczeniowego, 97
 - unchecked, 26, 48
 - Union, 147
 - warunkowego dostępu do składowej, *Patrz:* operator ?.
 - warunkowy, 47, 49
 - Where, 144
 - wyłuskania, 189
 - XOR, 49
 - zapytania, 144, 147–151
 - kaskadowy, 152
 - zbiorów, 147
 - złączenia, 159

operatory

 - arytmetyczne, 24
 - bitowe, 26
 - logiczne, 28, 29
 - porównania, 29
 - relacji, 29

P

 - pakowanie, 86, 87
 - parametr, 8, 38, 40, 65, 117, *Patrz też:* argument

- parametr
 - opcjonalny, 43
 - typowy, 98
 - deklarowanie, 100
 - wartość domyślna, 101
- parowanie, 174
- partially qualified name,
 - Patrz:* przestrzeń nazw
 - nazwa częściowa
 - kwalifikacja
- pętla, *Patrz:* instrukcja pętli
- platforma
 - .NET Framework, 9
- pliki
 - .cs, 9
 - .dll, 9
 - .exe, 9
- podklasa, 78
- podzapytanie, 148
- pole, 38, 39, 64
 - deklarowanie, 64
 - inicjalizacja, 64
 - instancji, 38
 - publiczne, 70
 - styczne, 38
- polimorfizm, 78
- porządkowanie, 162
- preprocesor, 192, 193
 - symbol
 - #error, 193
 - #warning, 193
- primary operator, *Patrz:* operator
- operator pierwszorzędny
- programowanie
 - asynchroniczne, 180, 184, 185
- property, *Patrz:* właściwość
- protokół, 105
- przeciążanie operatorów, 173
- zapełnienie zakresu, 25
- przesłanie, 82

- przestrzeń nazw, 9, 60, 139
 - alias, 64
 - deklaracja, 63
 - globalna, 61
 - importowanie, 61, 63
 - nazwa
 - częściowa
 - kwalifikacja, 62
 - przesłanie, 63
 - zasięg, 62
 - System, 9, 14
 - System.Collections, 35
- przypisanie
 - oznaczone, 39
 - dekonstruujące, 68
- przyrostek
 - D, 23
 - L, 23
 - liczbowy, 23
 - U, 23

Q

- query expression, *Patrz:* wyrażenie zapytaniowe

R

- rectangular array, *Patrz:* tablica regularna
- referencja, 29, 41
 - dynamic, 169
 - object, 169
 - polimorficzna, 78,
 - Patrz też:* polimorfizm
 - pusta, 134
 - rzutowana
 - jawnie, 78
 - niejawnie, 78
 - w dół, 78, 79
 - w górę, 78
 - this, 67, 69

- rzutowanie typu
 - jawne, 17
 - niejawne, 17

S

- sealing, *Patrz:* zabezpieczenie implementacji
- sekwencje
 - składanie, 133
- serializacja, 175
- składanie sekwencji, 133
- składnia
 - kaskadowa, 153, 155
 - wyrażeniowa, 155
 - wyrażen
 - zapytaniowych, 154
- dane, 15
- słownik, 35, 72, 89, 130
- słowo kluczowe, 11
 - async, 180, 182, 187
 - await, 180, 182, 184
 - base, 83
 - class, 64
 - const, 74
 - default, 40, 101
 - delegate, 120
 - dynamic, 165
 - explicit, 173
 - fixed, 191
 - implicit, 173
 - internal, 60
 - into, 157
 - kontekstowe, 12
 - let, 156
 - namespace, 60
 - orderby, 162
 - partial, 75
 - private, 60
 - public, 17, 60
 - stackalloc, 191
 - static, 16, 75
 - struct, 18
 - this, 66, 69, 83
 - throw, 126

unsafe, 189
var, 45
virtual, 80
stała, 13, 45, 74
deklarowanie, 74
statement, *Patrz:*
instrukcja
sterta, 38
stos, 38, 85, 191
typ, 33
struktura, 89–92
 Nullable, 134
 operator, 135
strumień
 wejścia-wyjścia, 9
subclass, *Patrz:* podklasa
subscriber, *Patrz:*
 subskrybent
interfejs, 125

T

tablica, 34, 191
 deklaracja, 34
 dynamiczna, 35
 element, 38
 inicjalizacja, 35, 37, 39
 kopiowanie, 35
 liczb całkowitych, 35
 liczba wymiarów, 35
 mieszająca, 89
 nieposortowana, 35
 posortowana, 35
 regularna, 36
 rozmiar, 35
 sortowanie, 35
 tworzenie, 35
 dynamiczne, 35
typ, 36
wielowymiarowa, 36
wyszczerbiona, 36, 37
wyszukiwanie, 35
zagnieżdżona, 36, 37
typ, 13
 anonimowy, 140
 bazowy, 172

bool, 14, 21, 28, 40
byte, 21, 22, 26
całkowitoliczbowy, 8, 17
 przepełnienie, 25
char, 21, 30, 40
ciągu znaków, 21
częściowy, 75
decimal, 21–23, 28
delegatu, 105
 kontrawariancja, 110
 kowariancja, 110
 uogólniony, 108
 zgodność, 109
dookreślony, *Patrz:*
 typ zamknięty
dostępność, 91
double, 21–23, 28
dynamiczny, 169
 konwersja, 169
enum, 95
false, 40
float, 21–23, 28
int, 8, 14, 15, 21, 35
kontrola
 dynamiczna, 87
 statyczna, 87
konwersja, 24, 169, 174
 jawna, 24
 niejawna, 24, 26
liczbowy, 22, 40, 89
logiczny, 21, 28, 40
long, 17, 21–23
nazwa
 kolizja, 63
 w pełni
 kwalifikowana, 61
niedookreślony, *Patrz:*
 typ otwarty
niejawnie
 konwertowany, 23
niemodyfikowalny, 72
obiektowy, 21
object, 21, 85, 86, 169
ObsoleteAttribute, 176
odwołanie, 61

otwarty, 99, 101
predefiniowany, 13,
 14, 15, 21, 40
referencyjny, 19, 21,
 29, 31, 36, 38, 41,
 65, 134
sbyte, 21, 22, 26
short, 21, 22, 26
string, 14, 21, 31
uint, 21–23
ulong, 21–23
uogólniony, *Patrz:*
 uogólnienie
ushort, 21, 22, 26
void, 65, 108
wartościowy, 21, 31,
 39, 89, 188
wartość
 domyślna, *Patrz:*
 wartość
 domyślna
wbudowany, *Patrz:* typ
 predefiniowany
 własny, 14
wnioskowany, 23
wskaźnikowy, 188
wyliczeniowy, 40, 95
 konwersja, 95
z dopuszczalną
 wartością pustą, 134
 konwersja, 135, 137
 pakowanie, 135
zagnieżdżony, 97
zamknięty, 99
znakowy, 21, 40
type argument, *Patrz:*
 argument typowy
type parameter, *Patrz:*
 parametr typowy
operator, 87
argument, 98

U

uogólnienie, 98, 100
ograniczenia, 101
pochodne, 102

V

verbatim string literal,
Patrz: literal ciągu
znaków dosłowny
void expression, *Patrz:*
wyrażenie puste

W

wartość
domyślna, 40, 43, 64, 95
minus 0, 27
minus nieskończoność,
27
NaN, 27
null, 20, 36, 40, 107,
134
plus nieskończoność, 27
whitespace, *Patrz:* znak
biała spacja
wiązanie
dynamiczne, 165–168,
171
językowe, 167, 168
niestandardowe, 167
statyczne, 165
według reguł, 167
ze wskazania, *Patrz:*
wiązanie
niestandardowe
wielorozprowadzanie,
Patrz: multimetoda
wiersz poleceń, 10
właściwość, 8, 70
akcesor, 70, 71
get, 70, 71, 72
poziom dostępności,
72
set, 70, 71, 72

automatyczna, 71
inicjalizacja, 72
Length, 35
Rank, 35
skrótowa do wyrażenia,
71
this, 73
wirtualna, 81
wyłącznie do
odczytu, 71
zapisu, 71
wskaźnik
beztypowy, 191
współbieżność, 180, 185,
186
wyjątek, 86
czasu wykonania, 137,
169
DivideByZero
↳Exception, 122
filtr, 124
IndexOutOfRangeException
↳Exception, 35
NullReference
↳Exception, 51
OutOfMemory
↳Exception, 123
OverflowException, 25
RuntimeBinder
↳Exception,
169, 170
System.Argument
↳Exception, 128
System.Argument
↳NullException,
126, 128
System.Argument
↳OutOfRangeException
↳Exception, 128
System.Exception, 123
System.Invalid
↳Operation
↳Exception, 128
System.Not
↳Implemented
↳Exception, 128

System.NotSupported
↳Exception, 128
System.ObjectDisposed
↳Exception, 128
WebException, 124
zgłaszanie, 126
ponowne, 126, 127
wyrażenie, 45, 52, 65, 67
dynamiczne, 170
lambda, 116, 117
asynchroniczne, 187
zmienna
wciągnięta, 118,
119
zmienna
zewnętrzna, 118
logiczne, 29
przypisania, 46
puste, 46
składnia
kaskadowa, 153,
155
wyrażeniowa, 155
zapytaniowe, 153, 160
konstruktor, 67

X

XML, 194
znacznik
c, 196
code, 196
example, 196
exception, 196
include, 197
list, 196
para, 197
param, 195
paramref, 196
permission, 196
remarks, 195
returns, 195
see, 196
seealso, 196
summary, 195

Z

- zapięczętowanie
 - implementacji, 82
- zapytanie, 144, 153, 157
 - zintegrowane, 143,
 - Patrz też:* LINQ
- zasada
 - przypisań oznaczonych, 39
- zdarzenie, 8, 111
 - akcesor, *Patrz:* akcesor
 - EventArgs, 110
 - implementacja, 113
 - EventArgs, 110
 - MouseEventArgs, 110
 - nadawca, *Patrz:*
 - nadawca
 - subskrybent, *Patrz:*
 - subskrybent
 - wirtualne, 81
- zestaw, 9
 - zaprzyjaźniony, 91
- złączenia równościowe, 159
- złączenie
 - GroupJoin, 160
 - Join, 159
 - suwakowe, 161
 - Zip, 161
- zmienna, 13, 38, 45
 - deklaracja, 45
 - inicjalizacja, 45
 - lokalna, 38, 52, 117
 - nazwa, 10
 - zakresowa, 154
- znak
 - !, 48, 137, 172
 - !=, 29, 97, 136, 172
 - \$, 32
 - %, 24, 49, 137, 172
 - &, 26, 97, 137, 190
 - &&, 29, 49
 - &=, 50
 - (), 12, 48
 - *, 12, 48, 172, 190
 - */, 13
 - *=, 50
 - ., 12, 48
 - /, 24, 49, 137, 172
 - /*, 13
 - //, 13
 - /=, 50
 - ;, 12
 - ?, 134
 - ?., 48
 - ?:, 49
 - ??, 138
 - @, 11
 - [], 48
 - ^, 26, 49, 97, 137, 172
 - ^=, 50
 - {, 12
 - |, 26, 49, 97, 137, 172
 - ||, 29, 49
 - |=, 50
 - }, 12
 - ~, 26, 48, 97, 137, 172
 - +, 32, 48, 97, 137, 172
 - ++, 25, 48, 97, 172
 - +=, 50, 97
 - <, 29, 49, 97, 136, 172
 - <<, 26, 49, 137, 172
 - <<=, 50
 - <=, 29, 49, 97, 136
 - =, 12, 46, 50, 97
 - =, 50, 97
 - ==, 13, 29, 135, 172
 - =>, 50
 - >, 29, 49, 97, 136, 172
 - >, 48, 189, 190
 - >=, 29, 49, 97, 136
 - >>, 26, 49, 137, 172
 - >>=, 50
 - biała spacja, 34
 - interpunkcyjny
 - języka C, 12
 - sterujący, 30

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>

Zwięźle, prosto i w punkt: C#!

C# od dawna jest dojrzałym, uniwersalnym i efektywnym językiem programowania, dzięki któremu można sprawnie pisać bezpieczny, przejrzysty i wydajny kod. Twórcy C# jako założenie przyjęli obiektowość i kontrolę typów, jednak przede wszystkim język ten ma być prosty w stosowaniu. Mimo to nawet tak świetne narzędzie, jak C# w wersji 7.0, jest jeszcze wygodniejsze w pracy, jeśli programista ma pod ręką coś, co wspomogą jego codzienną pracę i we właściwym miejscu podsunie potrzebną informację.

Niniejsza książka jest świetnym, zwięzłym i wyjątkowo praktycznym kompendium. Zawiera dokładnie to, co powinna – bez nudnawych wywodów i rozdmuchanych przykładów. Może posłużyć jako podręcznik do nauki C# lub jako bardzo poręczna ściągawka, pozwalająca na szybkie znalezienie odpowiedzi. Jeśli tylko posiadasz podstawowe umiejętności programowania w Javie, C++ lub w poprzednich wersjach C# i chcesz bez większych problemów przystąpić do programowania w C# 7.0, to trzymasz w ręku właściwą książkę.

Joseph Albahari napisał kilka książek o programowaniu w języku C#. Jest twórcą LINQPad, popularnego narzędzia do prototypowania zapytań LINQ.

Ben Albahari pracował w Microsoftzie, gdzie był szefem wielu istotnych projektów, w tym *Entity Framework*. Jest współtwórcą serwisu Auditionist, obsługującego wirtualne castingi aktorów w Wielkiej Brytanii.

W leksykonie przedstawiono:

- podstawy języka C#
- nowości w C#: krotki, składnię dekonstrukcji krotek, dopasowywanie wzorców
- wiązania dynamiczne i funkcje asynchroniczne
- wskaźniki, atrybuty, dyrektywy preprocesora i wiele innych zagadnień

Helion 

 helion.pl

 **HELION SA**
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 53
helion@helion.pl

Sprawdź nasze szkolenia!

SZKOLENIA

AKADEMIA IT & BUSINESS

WWW.SZKOLENIA.HELION.PL

KOD KORZYŚCI
Sięgnij po więcej! ▶



ISBN 978-83-283-4077-0



9 788328 340770