O'REILLY®

C# 7.0

Leksykon kieszonkowy

Joseph Albahari
Ben Albahari

Helion

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

Tłumaczenie: Przemysław Szeremiota

ISBN: 978-83-283-4077-0

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.

Spis treści

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`. Jedyna 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 sprowokuje 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 dynamic:

```
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. Rozstrzyganie w oparciu o implementację interfejsu IdynamicMetaObject

↪`Binding` nazywamy *wiązaniem ze wskazania* albo *wiązaniem niestandardowym* (ang. *custom binding*), a rozstrzyganie 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 znamienna różnica 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);  // ahoj
```

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 sprowokuje 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, indekserów, 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.

Rozstrzyganie 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
}
```

Rozstrzyganie 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 `X(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 wchodzą jedynie dwa pierwsze przeciążenia `X`. 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 *dodatkowy 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 literałów `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`.

Skorowidz

L

M

N

O

P

Q

R

S

T

Z

O'REILLY®

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 wspomoże 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.

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

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

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

Helion

helion.pl

HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

ISBN 978-83-283-4077-0

Cena: 34,90 zł