

O'REILLY®

C# 9.0

Leksykon kieszonkowy



Joseph Albahari
Ben Albahari

Helion 

Tytuł oryginału: C# 9.0 Pocket Reference: Instant Help for C# 9.0 Programmers

Tłumaczenie: Przemysław Szeremiota

ISBN: 978-83-283-7976-3

© 2021 Helion S.A.

Authorized Polish translation of the English edition of C# 9.0 Pocket Reference

ISBN 9781098101138 © 2021 Joseph Albahari and 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 wydawca dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz wydawca nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Helion S.A.

ul. Kościuszki 1c, 44-100 Gliwice

tel. 32 231 22 19, 32 230 98 63

e-mail: helion@helion.pl

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

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

<http://helion.pl/user/opinie/ch9lek>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

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

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

Spis treści

C# 9.0. Leksykon kieszonkowy	5
Pierwszy program w C#	5
Składnia	9
System typów	12
Typy liczbowe	24
Typ wartości logicznych i operatory logiczne	32
Znaki i ciągi znaków	34
Tablice	39
Zmienne i parametry	44
Operatory i wyrażenia	54
Operatory na typach z dopuszczalną wartością pustą	56
Instrukcje	62
Przestrzenie nazw	72
Klasy	76
Dziedziczenie	94
Typ object	104
Struktury	109
Modyfikatory dostępu	111
Interfejsy	113
Typy wyliczeniowe	117
Typy zagnieżdżone	120
Uogólnienia	121
Delegaty	130
Zdarzenia	136
Wyrażenia lambda	142

Metody anonimowe	147
Wyjątki i instrukcja try	148
Enumeratory i iteratory	156
Typy z dopuszczalną wartością pustą	161
Zabezpieczanie pustych referencji	167
Metody rozszerzające	169
Typy anonimowe	171
Krotki	171
Rekordy (C# 9)	174
Wzorce	181
LINQ	185
Wiązanie dynamiczne	210
Przeciążanie operatorów	219
Atrybuty	222
Atrybuty wywołania	226
Funkcje asynchroniczne	227
Wskaźniki i kod nienadzorowany	238
Wskaźniki do funkcji (C# 9)	243
Dyrektywy preprocesora	244
Dokumentacja XML	247

Przykłady typów predefiniowanych

Typy predefiniowane (albo tak zwane typy wbudowane) to takie typy, które są przez kompilator obsługiwane w sposób specjalny, wyróżniony. Przykładem takiego typu może być `int` jako predefiniowany typ dla wartości liczbowych całkowitych, zajmujących w pamięci 32 bity, a więc pokrywających zakres od -2^{31} do $2^{31}-1$. Na wartościach typu `int`, czyli obiektach typu `int`, możemy przeprowadzać operacje, np. arytmetyczne:

```
int x = 12 * 30;
```

Innym predefiniowanym typem języka C# jest `string`. Typ `string` reprezentuje ciąg znaków, np. „.NET” czy „http://helion.pl”. Ciągami znaków możemy manipulować poprzez wywoływanie funkcji operujących na obiektach typu `string`:

```
string message = "Ahoj, przygodo";  
string upperMessage = message.ToUpper();  
Console.WriteLine (upperMessage); // AHOJ, PRZYGODO
```

```
int x = 2021;  
message = message + x.ToString();  
Console.WriteLine (message); // Ahoj, przygodo2021
```

Kolejny predefiniowany typ — `bool` — to typ dopuszczający dokładnie dwie wartości obiektu: `true` (prawda) i `false` (fałsz). Typ `bool` jest powszechnie stosowany do rozdzielania wykonania kodu w zależności od wyników instrukcji warunkowej `if`. Oto przykład:

```
bool simpleVar = false;  
if (simpleVar)  
    Console.WriteLine ("To się nie pojawi");
```

```
int x = 5000;  
bool lessThanAMile = x < 5280;  
if (lessThanAMile)  
    Console.WriteLine ("To się pojawi");
```

Wiele ważnych typów, które nie są typami predefiniowanymi (np. `Date` ↪ `Time`), w środowisku .NET umieszczono w przestrzeni nazw `System`.

Przykłady typów własnych

Tak jak z prostych funkcji możemy składać funkcje złożone, tak z typów prostych możemy składać typy złożone. W tym przykładzie zdefiniujemy własny typ o nazwie `UnitConverter` — klasę, która będzie definiować schematy konwersji jednostek miar:

```
using System;

UnitConverter feetToInches = new UnitConverter(12);
UnitConverter milesToFeet = new UnitConverter(5280);

Console.WriteLine (feetToInches.Convert(30)); // 360
Console.WriteLine (feetToInches.Convert(100)); // 1200
Console.WriteLine (feetToInches.Convert
    (milesToFeet.Convert(1))); // 63360
```

```
public class UnitConverter
{
    int ratio; // Pole

    public UnitConverter (int unitRatio) // Konstruktor
    {
        ratio = unitRatio;
    }

    public int Convert (int unit) // Metoda
    {
        return unit * ratio;
    }
}
```

Składowe typu

Typ może zawierać *dane składowe* i *funkcje składowe* (metody). W naszej klasie `UnitConverter` zdefiniowaliśmy *pole* danej składowej o nazwie `ratio` (dla współczynnika konwersji). Do funkcji składowych klasy `UnitConverter` zaliczymy metodę `Convert` i *konstruktor* obiektów klasy `UnitConverter`.

Symetria typów wbudowanych i typów własnych

Bardzo eleganckim aspektem języka C# jest to, że typy predefiniowane (wbudowane) i typy własne tylko nieznacznie różnią się między sobą. Na przykład wbudowany typ `int` służy jako wyznacznik cech wartości

będących liczbami całkowitymi. Przechowuje dane (32 bity) i udostępnia składowe do manipulowania tymi danymi, np. metodę `ToString` do zamiany wartości liczbowej na ciąg znaków. Podobnie nasz własny typ `UnitConverter`, który określa cechy obiektów klasy `UnitConverter`, definiuje miejsce przechowywania danych — współczynnika konwersji `ratio` — i udostępnia składowe do odwoływania się do danych.

Konstruktory a konkretyzacja typu

Instancje poszczególnych typów są powoływane do życia w programie w ramach procesu tak zwanej *konkretyzacji* (ang. *instantiation*), czyli procesu tworzenia instancji (obiektów) danego typu. Typy wbudowane mogą być konkretyzowane poprzez podanie wartości dla obiektu za pomocą literału takiego jak **12** czy *"Ahoj, przygodo"*.

Do tworzenia egzemplarza typu definiowanego przez użytkownika (a więc niewbudowanego) służy zawsze operator `new`, jak na początku ostatniego przykładu, który rozpoczynają dwa wywołania `new` tworzące dwa egzemplarze typu `UnitConverter`. Natychmiast po tym, jak operator `new` skonkretyzuje instancję klasy zwaną obiektem, wywoływany jest *konstruktor* obiektu w celu zainicjalizowania wartości nowej instancji klasy. Konstruktor jest definiowany w klasie jak zwyczajna metoda, z tym że nazwa tej metody i wartość zwracana są zredukowane do nazwy typu:

```
public UnitConverter (int unitRatio)    // Konstruktor
{
    ratio = unitRatio;
}
```

Składowe statyczne a składowe instancji

Dane i funkcje składowe klasy operujące na *instancji* klasy są tak zwanymi składowymi instancji. W przypadku klasy `UnitConverter` można tak powiedzieć o metodzie `Convert`, a w przypadku typu predefiniowanego `int` taki charakter ma metoda `ToString` — obie metody są składowymi instancji odpowiednich typów. Wszystkie składowe deklarowane w klasie są przez domniemanie składowymi instancji.

Z kolei dane i funkcje składowe, które operują nie na instancjach typu, ale na samym typie jako takim, muszą być specjalnie oznaczane w deklaracji — służy do tego słowo kluczowe `static`. W odwołaniach do statycznych

składowych z zewnątrz klasy używa się nazwy samej klasy, a nie instancji klasy. Mieliśmy przykłady takich składowych statycznych w postaci metody `Main` czy metody `Console.WriteLine`. W przypadku klasy `Console` mamy zresztą do czynienia z *klasą statyczną*, w której *wszystkie* składowe również są statyczne. Dzięki temu nie musieliśmy w ogóle tworzyć obiektu klasy `Console` — cała aplikacja korzysta ze wspólnej konsoli, więc nie trzeba tworzyć osobnych obiektów do obsługi konsoli w poszczególnych fragmentach programu.

Żeby zilustrować różnice pomiędzy składowymi statycznymi a składowymi instancji, przyjrzymy się klasie `Panda` z polem instancji o nazwie `Name`, odnoszącym się do pojedynczej pandy (a więc do pojedynczej instancji klasy), oraz z polem o nazwie `Population`, odnoszącym się do łącznej liczby obiektów `Panda`, a więc do klasy `Panda` jako takiej i równocześnie do wszystkich obiektów tej klasy. Poniżej stworzymy dwa obiekty klasy `Panda` i wypiszemy ich nazwy, a potem liczbę pand:

```
Panda p1 = new Panda ("Pan Daa");
Panda p2 = new Panda ("Ban Taa");

Console.WriteLine (p1.Name);    // Pan Daa
Console.WriteLine (p2.Name);    // Ban Taa

Console.WriteLine (Panda.Population);    // 2

public class Panda
{
    public string Name;           // Pole instancji
    public static int Population; // Pole statyczne (klasy)

    public Panda (string n)       // Konstruktor
    {
        Name = n;                 // Pole instancji
        Population = Population + 1; // Pole statyczne
    }
}
```

Próba odwołania się do `p1.Population` (dostęp do składowej statycznej przez referencję instancji) albo `Panda.Name` (dostęp do składowej instancji przez nazwę klasy) spowoduje błąd kompilacji programu.

Słowo kluczowe public

Słowo kluczowe `public` służy do eksponowania składowych klas tak, aby można się było do nich odwoływać z innych klas. Gdyby w naszym poprzednim przykładzie pole `Name` w klasie `Panda` nie było oznaczone jako publiczne, zostałyby uznane za prywatne, nie moglibyśmy się więc do niego odwoływać w klasie `Test` i nie zdołalibyśmy wypisać imion `pand`. Oznaczenie składowej klasy słowem `public` oznacza powiadomienie kompilatora, że dana składowa ma charakter publiczny, czyli powinna być widoczna dla typów zewnętrznych; wszystkie inne składowe (bez słowa `public`) są niedostępne na zewnątrz. W terminologii obiektowej powiemy, że za składowymi publicznymi ukrywamy składowe prywatne klasy.

Tworzenie przestrzeni nazw

Zwłaszcza w większych programach do skutecznego organizowania typów w programie niezbędne stają się własne przestrzenie nazw. Oto jak możemy umieścić klasę `Panda` w przestrzeni nazw `Animals`:

```
namespace Animals
{
    public class Panda
    {
        ...
    }
}
```

Więcej o przestrzeniach nazw opowiemy w osobnym podrozdziale „Prze-strzenie nazw”.

Metoda Main

Wszystkie dotychczasowe przykłady zawierały instrukcje pisane w tzw. głównym korpusie programu, co jest nowością w C# 9. W poprzednich wersjach C# analogiczny program musiałby być jawnie zamknięty w odpowiedniej strukturze z metodą `Main`:

```
class Program
{
    static void Main()    // Punkt wejścia do programu
    {
        int x = 12 * 30;
        Console.WriteLine (x);
    }
}
```

```
}  
}
```

Pod nieobecność instrukcji w głównym przebiegu programu kompilator szuka jawnego punktu wejścia do programu w postaci metody `Main`. Metoda `Main` może zostać zdefiniowana w dowolnej klasie (ale równocześnie może być tylko jedna taka metoda w całym programie). Jeśli w metodzie `Main` chcemy odwoływać się do prywatnych składowych którejs z klas programu, najwygodniej będzie zdefiniować `Main` wewnątrz tej właśnie klasy i w takim układzie jawna metoda `Main` będzie wygodniejsza niż kod pisany wprost w głównym przebiegu programu.

Metoda `Main` może opcjonalnie, zamiast `void`, zwracać do środowiska wykonawczego wartość typu całkowitego (liczbę). Metoda `Main` może też opcjonalnie w wywołaniu przyjmować argumenty w postaci tablicy ciągów znaków (wypełnianej argumentami, z którymi uruchomiono program). Oto przykład:

```
static int Main (string[] args) {...}
```

Uwaga

Tablica (jak `string[]` z powyższego przykładu) reprezentuje pewną ustaloną liczbę elementów o tym samym typie (zobacz też podrozdział „Tablice”).

Metoda `Main` może być też zadeklarowana jako asynchroniczna i zwracać wartość typu `Task` bądź `Task<int>` (zobacz podrozdział „Funkcje asynchroniczne”).

Instrukcje w bloku głównym programu (C# 9)

Możliwość pisania programu wprost w bloku głównym, na zewnątrz klas, uwalnia programistów od uciążliwości definiowania statycznej metody `Main` oraz klasy, w której ta metoda ma się znajdować. Plik z programem w bloku głównym składa się z trzech części (w takiej kolejności):

1. Dyrektywy `using` (opcjonalnie).
2. Serii instrukcji, ewentualnie przeplatanych z deklaracjami metod.
3. Deklaracji typów i przestrzeni nazw (opcjonalnie).

Wszystko, co znajduje się w części 2., zostanie przez kompilator potraktowane jako wnętrze niejawnej metody „main” generowanej automatycznie wewnątrz wygenerowanej automatycznie klasy. Oznacza to, że metody definiowane w głównym bloku programu są tak naprawdę metodami lokalnymi tej automatycznej klasy (zobacz też podpunkt „Metody lokalne”). Instrukcje pisane wprost w bloku głównym również mogą zakończyć się przekazaniem wartości (całkowitoliczbowej) jako wyniku wykonania programu, i tak samo jak wnętrze metody `Main` mają dostęp do „magicznej” zmiennej typu `string[]` o nazwie `args`, w której zapisane są argumenty uruchomienia programu.

Wiemy już, że program może posiadać tylko jeden punkt wejścia; z tego też powodu tylko jeden z plików w całym projekcie programu C# może zawierać kod pisany wprost w bloku głównym programu.

Typy i konwersje

W języku C# można korzystać z konwersji pomiędzy instancjami zgodnych typów. Konwersja zawsze owocuje utworzeniem nowej wartości na bazie wartości istniejącej. Konwersje mogą być *jawne* i *niejawne*; niejawne odbywają się automatycznie, a jawne wymagają, aby programista zastosował *rzutowanie* (ang. *cast*). W poniższym przykładzie dochodzi do *niejawnego* rzutowania (konwersji) typu `int` na typ `long` (to również typ prosty dla liczb całkowitych, o zwielokrotnionej pojemności w stosunku do `int`) oraz do *jawnego* rzutowania typu `int` na typ `short` (kolejny typ dla liczb całkowitych, o ograniczonym zakresie liczbowym w porównaniu do `int`):

```
int x = 12345;           // int to liczba całkowita zapisana na 32 bitach
long y = x;             // Niejawna konwersja na 64-bitowy typ long
short z = (short) x;   // Jawna konwersja na 16-bitowy typ short
```

Niejawne konwersje są dozwolone wtedy, gdy kompilator może zagwarantować skuteczność konwersji automatycznej *oraz* konwersja nie zagraża utratą informacji.

Z zasady niejawne konwersje są dozwolone wtedy, gdy kompilator może zagwarantować skuteczność konwersji bez ryzyka utraty informacji. W innych przypadkach konwersja pomiędzy (zgodnymi) typami musi być jawna.

Typy wartościowe a typy referencyjne

Typy języka C# można podzielić na *typy wartościowe* (ang. *value types*) i *typy referencyjne* (ang. *reference types*).

Zdecydowana większość typów wbudowanych zalicza się do *typów wartościowych* (w szczególności do tej kategorii należą wszystkie typy liczbowe, typ znakowy `char` oraz typ logiczny `bool`); do tej samej kategorii zalicza się definiowane przez programistę typy strukturalne (`struct`) i wyliczeniowe (`enum`). *Typy referencyjne* to wszystkie klasy, tablice, delegaty i interfejsy.

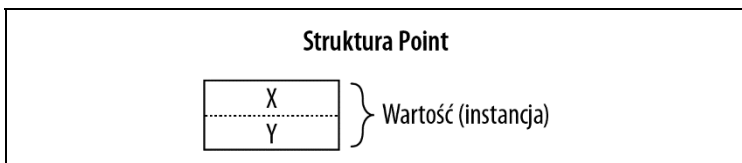
Podstawowa różnica pomiędzy typami wartościowymi a typami referencyjnymi sprowadza się do sposobu obsługi instancji typu w pamięci.

Typy wartościowe

Zawartość stałej albo zmiennej *typu wartościowego* to zwyczajna wartość (ogólnie typy wartościowe to typy przechowywane i przekazywane „przez wartość”). Na przykład zawartość obiektu wbudowanego typu `int` to 32 bity danych.

Własny typ wartościowy (zobacz rysunek 1.) można zdefiniować za pomocą słowa kluczowego `struct`, jak w poniższym kodzie:

```
public struct Point { public int X, Y; }
```



Rysunek 1. Instancja typu wartościowego w pamięci

Przypisanie zmiennej typu wartościowego powoduje każdorazowo wykonanie *kopii* instancji przypisywanego obiektu. Oto przykład:

```
Point p1 = new Point();  
p1.X = 7;
```

```
Point p2 = p1;    // Przypisanie powoduje wykonanie kopii
```

```
Console.WriteLine (p1.X);    // 7
```

```

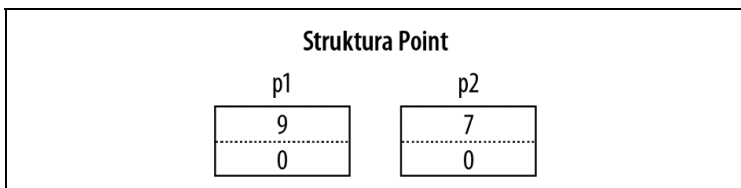
Console.WriteLine (p2.X);    // 7

p1.X = 9;                    // Zmiana wartości p1.X

Console.WriteLine (p1.X);    // 9
Console.WriteLine (p2.X);    // 7

```

Na rysunku 2. widać, że obiekty p1 i p2 dysponują osobną (każdy własną) pamięcią dla przechowywanych wartości.

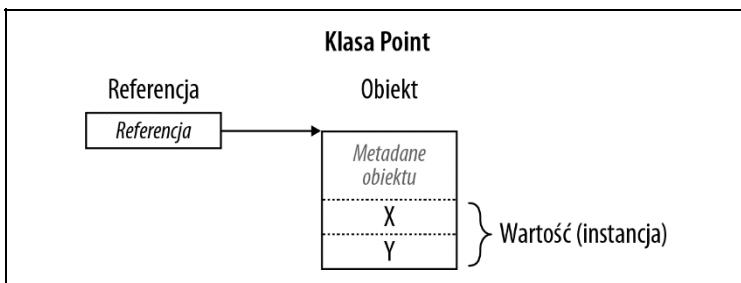


Rysunek 2. Przypisanie powoduje kopiowanie wartości typu wartościowego

Typy referencyjne

Typ referencyjny jest bardziej złożony niż typ wartościowy, bo jego obiekt składa się z dwóch części: właściwego *obiekту* oraz *referencji* do obiektu. Zawartość zmiennej bądź stałej typu referencyjnego odnosi się do obiektu przechowującego wartość. Przedstawimy to na przykładzie typu Point przepisanego na postać klasy — zobacz rysunek 3.

```
public class Point { public int X, Y; }
```



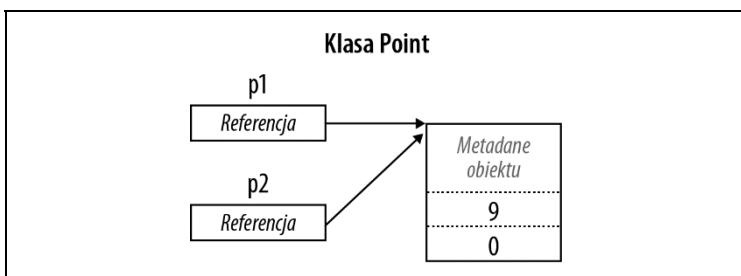
Rysunek 3. Instancja typu referencyjnego w pamięci

Przypisanie zmiennej typu referencyjnego oznacza zawsze kopiowanie samej referencji, a nie kopiowanie instancji. Dzięki temu można uzyskać

sytuację, w której do tego samej instancji (obiektu) odnosi się wiele zmiennych typu referencyjnego — w przypadku typów wartościowych jest to niemożliwe, bo każda zmienna odnosi się do własnego, pojedynczego obiektu. Powtórzmy poprzedni przykład dla typu `Point` jako klasy, a przekonamy się, że operacje na `p1` wpływają na wartość widzianą przez `p2`:

```
Point p1 = new Point();  
p1.X = 7;  
  
Point p2 = p1;           // Skopiowanie referencji p1  
  
Console.WriteLine (p1.X); // 7  
Console.WriteLine (p2.X); // 7  
  
p1.X = 9;               // Zmiana wartości p1.X  
Console.WriteLine (p1.X); // 9  
Console.WriteLine (p2.X); // 9
```

Na rysunku 4. ilustrujemy przyczynę takiego zachowania programu: `p1` i `p2` są tutaj referencjami odnoszącymi się do tej samej instancji (obiektu):



Rysunek 4. Przypisanie powoduje kopiowanie referencji

Wartość pusta

Do referencji można przypisać literał `null` oznaczający, że referencja nie odnosi się do żadnego obiektu. Na przykład dla klasy `Point`:

```
Point p = null;  
Console.WriteLine (p == null); // True  
  
próba odwołania do składowej za pośrednictwem pustej referencji spowoduje wyjątek czasu wykonania:  
  
Console.WriteLine (p.X);           // NullReferenceException
```

Uwaga

W podrozdziale „Zabezpieczanie pustych referencji” opisujemy mechanizm pozwalający zredukować liczbę przypadkowych wystąpień wyjątku `NullReferenceException`.

Z kolei — dla porównania — do zmiennej typu wartościowego nie można przypisać wartości pustej:

```
struct Point {...}
...
Point p = null;    // Błąd kompilacji
int x = null;     // Błąd kompilacji
```

Uwaga

W języku C# stosuje się specjalną konstrukcję o nazwie *nullable types* do reprezentowania wartości pustych typu wartościowego (zobacz podrozdział „Typy z dopuszczalną wartością pustą”).

Taksonomia typów predefiniowanych

W języku C# mamy do dyspozycji niżej wymienione typy predefiniowane (wbudowane).

Typy wartościowe:

- liczbowe:
 - typy liczb całkowitych ze znakiem (`sbyte`, `short`, `int`, `long`),
 - typy liczb całkowitych bez znaku (`byte`, `ushort`, `uint`, `ulong`),
 - typy liczb rzeczywistych (`float`, `double`, `decimal`);
- typ logiczny (`bool`);
- typ znakowy (`char`).

Typy referencyjne:

- typ ciągu znaków (`string`);
- typ obiektowy (`object`).

Predefiniowane typy języka C# stanowią aliasy dla typów zdefiniowanych w przestrzeni nazw `System`. Z tego względu poniższe dwa wiersze


kodu różnią się wyłącznie składnią, natomiast ich znaczenie i działanie są identyczne:

```
int i = 5;  
System.Int32 i = 5;
```

W środowisku wykonawczym CLR (ang. *Common Language Runtime*) mianem *typów prostych* (ang. *primitive types*) określamy podzbiór predefiniowanych typów *wartościowych*. Taka nazwa wywodzi się z faktu, że wartości tych typów są niepodzielne, czyli stanowią najmniejsze cegiełki danych w konkretnym języku i jako takie mają bezpośrednie odwzorowanie w kodzie maszynowym.

PROGRAM PARTNERSKI

— GRUPY HELION —

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

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

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

GRUPA
Helion 

Efektywne wsparcie każdego programisty C#!

C# jest wyjątkowo udanym dziełem Microsoftu — konsekwentnie rozwijanym, dojrzałym i wyjątkowo wszechstronnym obiektowym językiem programowania. Wersja C# 9.0 została dostosowana do współpracy ze środowiskiem uruchomieniowym Microsoft .NET 5. Język ten pozwala programistom na uzyskiwanie imponującej efektywności — dzięki prostocie, ekspresywności kodu i wydajności działania. Jednak nawet profesjonaliści, którym zależy na sprawnej pracy, potrzebują niekiedy poręcznej ściągawki.

W tym zwięzłym kompendium znajdziesz wszystko, co jest potrzebne do programowania w C#. Książka stanowi nieocenioną pomoc dla osób, które znają C# albo C++ lub Javę i postanowiły nabrać wprawy w pracy z najnowszą wersją C#. Poszczególne zagadnienia przedstawiono w przejrzysty, treściwy i równocześnie esencjonalny sposób. Wszystkie użyte tu listingi i fragmenty kodu są dostępne w postaci interaktywnych próbek dla platformy LINQPad, która pozwala na błyskawiczne prototypowanie i obserwowanie efektów wszelkich zmian w kodzie.

W tej książce:

- gruntowne podstawy języka z uwzględnieniem nowości C# w wersji 9.0
- zagadnienia zaawansowane, w tym przeciążanie operatorów, ograniczenia typów, iteratory, wzorce dopasowania typów, wyrażenia lambda i domknięcia
- funkcje asynchroniczne w C#
- kod nienadzorowany i wskaźniki

Joseph Albahari jest autorem kilku cenionych książek poświęconych programowaniu. Jest też twórcą LINQPaDa, popularnego narzędzia pomocnego w implementowaniu zapytań do baz danych w LINQ.

Ben Albahari jest współzałożycielem serwisu Auditionist. Przez pięć lat pracował w Microsoftzie, gdzie kierował projektami. Jest współautorem kilku książek o programowaniu w C#.

Helion 

 helion.pl

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

Sprawdź nasze szkolenia!



AKADEMIA IT & BUSINESS

HELIONSZKOLENIA.PL

KOD KORZYŚCI
Sięgnij po więcej! ▶



ISBN 978-83-283-7976-3



9 788328 379763

INFORMATYKA W NAJLEPSZYM WYDANIU

Cena: 49,00 zł