O'REILLY®

C# 12

Leksykon kieszonkowy

Natychmiastowa pomoc dla programistów C#

Joseph Albahari
Ben Albahari

Helion

Tytuł oryginału: C# 12 Pocket Reference: Instant Help for C# 12 Programmers

Tłumaczenie: Piotr Rajca, Przemysław Szeremiota

ISBN: 978-83-289-1222-9

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/c12lek
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

System typów

Typ wyznacza charakter wartości. W naszym przykładzie użyliśmy dwóch literałów typu całkowitego (`int`) o wartościach `12` i `30`. Zadeklarowali-śmy także zmienną typu `int` o nazwie `x`.

Zmienna reprezentuje obszar pamięci, w którym w czasie wykonania programu przechowywane są wartości danego typu; w miarę wykony-wania programu wartości te mogą się zmieniać. Z kolei *stałe* nie zmieniają swojej wartości w czasie wykonania programu (o stałych powiemy więcej nieco później).

Wszystkie wartości w programie C# są *obiektami* pewnego typu. Określenie typu determinuje znaczenie wartości i zestaw wartości dopuszczalnych dla danej zmiennej.

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 ciągi 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 = 2022;
message = message + x.ToString();
Console.WriteLine (message);          // Ahoj, przygodo 2022
```

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. `DateTime`), 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:

```
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 nie operują na instancjach typu, mogą zostać oznaczone jako statyczne; służy do tego słowo kluczowe `static`. W odwołaniach do statycznych składowych z zewnątrz klasy używa się nazwy samej *klasy*, a nie *instancji* klasy. Przykładem takiej składowej może być metoda `WriteLine` klasy `Console`. Ponieważ jest to

metoda statyczna, wywołujemy ją, używając zapisu `Console.WriteLine()`, a nie `new Console().WriteLine()`.

W poniższym przykładzie klasy `Panda` pole instancji o nazwie `Name` odnosi się do pojedynczej pandy (a więc do pojedynczej instancji klasy), natomiast pole `Population` odnosi się do całej populacji pand — zbioru wszystkich obiektów `Panda`. W poniższym przykładzie stworzymy dwa obiekty klasy `Panda` i wyświetlimy 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, zostałoby uznane za prywatne, nie moglibyśmy się więc do niego odwoływać w kodzie w klasie `Test`. Oznaczenie składowej klasy słowem `public` można by porównać ze stwierdzeniem: „Chciałbym, żeby ta składowa była widoczna dla innych klas — cała reszta to moje prywatne

szczegóły implementacyjne". 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 „Przestrzenie nazw".

Metoda Main

Wszystkie dotychczasowe przykłady zawierały instrukcje pisane w tzw. głównym korpusie programu. Taka możliwość została wprowadzona w C# 9; w poprzednich wersjach C# analogiczny program musiałby być jawnie zamknięty w odpowiedniej strukturze z metodą `Main`:

```
using System;
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).

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

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 z `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
```

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 strukturowe (`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.

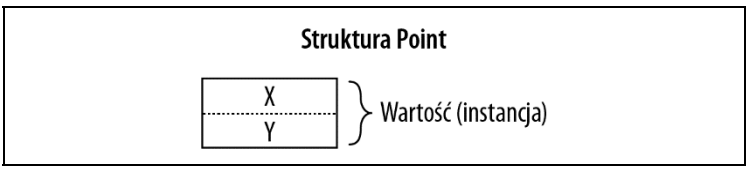
Typy wartościowe

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:

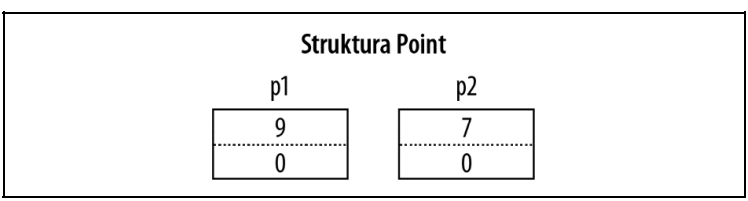
```
Point p1 = new Point();
p1.X = 7;

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.

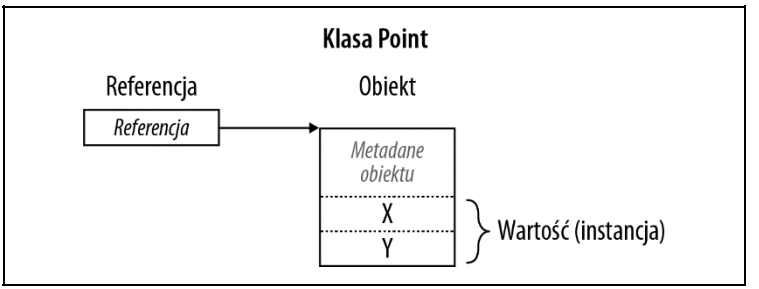


Rysunek 2. Przypisanie powoduje kopiowanie wartości typu wartościowego

Typy referencyjne

Typ referencyjny jest bardziej złożony niż typ wartościowy, bo jego obiekt składa się z dwóch części: właściwego *obiektu* 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 tej 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`:

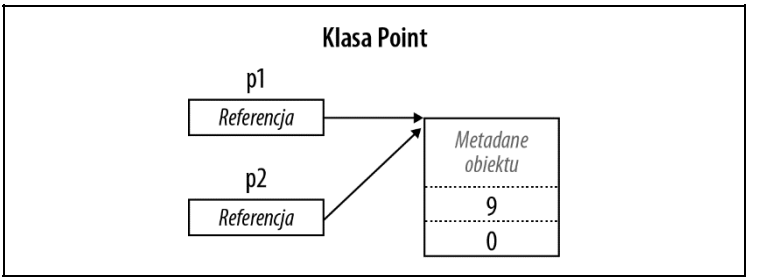
```
Point p1 = new Point();
p1.X = 7;

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 się 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
```

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

Kodujesz w C#?
Oto Twoje źródło czystej informacji!

C# to podstawowy element ekosystemu Microsoftu, napędzający aplikacje Windows, rozwój gier przy użyciu Unity i usługi backendowe z użyciem ASP.NET. Twórcy tego języka chcieli przede wszystkim zapewnić programistom jak największą efektywność, co znalazło odzwierciedlenie w jego prostocie, a także ekspresywności kodu i wydajności działnia. Wersja C# 12 została dostosowana do współpracy ze środowiskiem uruchomieniowym Microsoft .NET 8.

Jeśli szukasz źródła błyskawicznych odpowiedzi na pytania, jakie się pojawiają podczas pracy z C#, ta książka sprawdzi się idealnie! Została pomyślana tak, aby maksymalnie ułatwić przeglądanie i odnajdywanie potrzebnych treści — jest precyzyjnym, zwięzłym i niezwykle praktycznym przewodnikiem, szczególnie cenionym przez osoby, które znają już inne języki programowania, takie jak C++ czy Java. Wszystkie fragmenty kodu zostały udostępnione w programie LINQPad jako interaktywne przykłady. Można je edytować i od razu oglądać wyniki bez konieczności tworzenia projektów w Visual Studio. To książka, którą każdy programista C# powinien mieć pod ręką!

Najważniejsze zagadnienia:

- podstawy języka z uwzględnieniem nowych cech C# w wersji 12
- zaawansowane tematy, w tym przeciążanie operatorów, ograniczenia typów, typy akceptujące wartości puste, operator lifting
- domknięcia, wzorce i funkcje asynchroniczne
- LINQ: sekwencje, przetwarzanie opóźnione, standardowe operatory zapytań
- niebezpieczny kod
- niestandardowe atrybuty, dyrektywy preprocesora i generowanie dokumentacji XML

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

Ben Albahari pracował w Microsofcie, gdzie kierował kilkoma projektami, w tym .NET Compact Framework i Entity Framework. Jest współautorem książek o programowaniu w C#.

Helion

helion.pl

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

KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-289-1222-9

9 788328 912229

Cena: 49,90 zł