

IDŹ DO

PRZYKŁADOWY ROZDZIAŁ



SPIS TREŚCI

KATALOG KSIĄŻEK

KATALOG ONLINE

ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK

DODAJ DO KOSZYKA

CENNIK I INFORMACJE

ZAMÓW INFORMACJE
O NOWOŚCIACH

ZAMÓW CENNIK

CZYTELNIA

FRAGMENTY KSIĄŻEK ONLINE

C#. Leksykon kieszonkowy

Autorzy: Peter Drayton, Ben Albahari, Ted Neward

Tłumaczenie: Przemysław Steć

ISBN: 83-7361-082-0

Tytuł oryginału: [C# Language Pocket Reference](#)

Format: B5, stron: 146



Książka „C#. Leksykon kieszonkowy” dostarcza programistom zwięzłego opisu najbardziej innowacyjnego z języków środowiska .NET. Zaprojektowana jako poręczny, przenośny przewodnik do codziennego użytku, stanowi niezbędną pomoc dla programistów. Pomoże także Czytelnikowi przypomnieć sobie typowe wzorce składniowe, a ponadto ułatwi poruszanie się po środowisku .NET Framework.

Książka ta zawiera:

- Opis języka C# zawierający reguły składniowe dotyczące słów kluczowych, deklaracji i instrukcji
- Wprowadzenie do systemu typów, w tym opis mechanizmu opakowywania i odpakowywania pomiędzy typami referencyjnymi a typami wartościowymi
- Zestawienie opcji kompilatora C# oraz przewodnik po istotnych narzędziach środowiska .NET
- Tabele opisujące składnię wyrażeń regularnych, specyfikatory formatu oraz zestawienie przestrzeni nazw i odpowiadających im złożeń



Spis treści

<i>Identyfikatory oraz słowa kluczowe</i>	8
<i>Podstawowe elementy</i>	9
<i>Typy wartościowe i referencyjne</i>	10
Typy wartościowe	10
Typy referencyjne	11
<i>Typy predefiniowane</i>	16
Typy całkowite	16
Typy zmiennoprzecinkowe	18
Typ dziesiętny (decimal)	19
Typ znakowy (char)	20
Typ logiczny (bool)	21
Typ object	22
Typ łańcuchowy (string)	22
<i>Tablice</i>	23
Tablice wielowymiarowe	24
Lokalne deklaracje tablic i deklaracje tablic jako pól	25
Długość i rząd tablicy	25
Kontrola zakresów	25
Konwersje tablic	26
<i>Zmienne i parametry</i>	26
Określone przypisanie	26
Wartości domyślne	27
Parametry	28
<i>Wyrażenia i operatory</i>	31
Priorytet operatorów	31
Operatory kontroli wypełnienia arytmetycznego	34

Instrukcje	34
Instrukcje wyrażeniowe	35
Instrukcje deklaracyjne.....	35
Instrukcje wyboru	36
Instrukcje pętli	39
Instrukcje skoku	41
Przestrzenie nazw	44
Pliki	44
Wykorzystanie przestrzeni nazw	45
Klasy	48
Słowo kluczowe <code>this</code>	51
Pola.....	52
Stałe	52
Właściwości.....	54
Indeksatory	55
Metody	56
Konstruktory egzemplarzy	57
Konstruktory statyczne	59
Destruktry i finalizery	60
Typy zagnieżdżone.....	61
Modyfikatory dostępu	62
Ograniczenia modyfikatorów dostępu	64
Struktury	65
Interfejsy	66
Definiowanie interfejsu	67
Implementacja interfejsu	68
Wykorzystanie interfejsu	68
Rozszerzanie interfejsu.....	69
Jawna implementacja interfejsu	69
Ponowna implementacja interfejsu	70
Konwersje interfejsów	71
Wyliczenia	71
Operatory wyliczeń	73
Konwersje wyliczeń.....	73

Delegaty	74
Delegaty zbiorowe	74
Zdarzenia	76
Definiowanie delegata dla zdarzenia.....	76
Przechowywanie danych zdarzenia z wykorzystaniem klasy EventArgs.....	76
Deklaracja i wywoływanie zdarzenia	77
Reagowanie na zdarzenie z wykorzystaniem procedury obsługi zdarzenia	78
Aksesory zdarzenia	79
Przeciążanie operatorów	80
Implementacja równości wartości	80
Logiczne pary operatorów.....	82
Niestandardowe konwersje — jawne i niejawnie	82
Operatory przeciążalne pośrednio	83
Instrukcje try i wyjątki	84
Wyjątki.....	84
Klauzula catch	86
Blok finally	87
Główne właściwości klasy System.Exception.....	88
Atrybuty	89
Klasy atrybutów	90
Parametry nazwane i pozycyjne.....	90
Elementy docelowe atrybutów	91
Zastosowanie kilku atrybutów	92
Kod niebezpieczny i wskaźniki	92
Podstawowe informacje o wskaźnikach.....	92
Kod niebezpieczny	93
Instrukcja fixed	94
Operator wskaźnikowego dostępu do składowych	95
Słowo kluczowe stackalloc	95
Void*	96
Wskaźniki do kodu niezarządzanego.....	96
Dyrektywy preprocesora	97

<i>Opis biblioteki Framework Class Library</i>	98
Typy rdzenne.....	99
Tekst.....	100
Kolekcje.....	100
Strumienie i operacje wejścia-wyjścia	101
Praca sieciowa.....	101
Wątki.....	102
Bezpieczeństwo	102
Refleksje i metadane	103
Złożenia	104
Serializacja.....	104
Zdalne wykorzystanie obiektów	104
Usługi WWW	105
Dostęp do danych	106
XML.....	106
Grafika	107
Rozbudowane aplikacje klienckie	107
Aplikacje WWW	108
Globalizacja	109
Konfiguracja.....	109
Zaawansowane usługi komponentowe.....	109
Diagnostyka i testowanie.....	110
Współpraca z kodem niezarządzanym.....	110
Obsługa kompilatora i narzędzi	111
Funkcje środowiska uruchomieniowego.....	112
Rodzime funkcje systemu operacyjnego	112
<i>Przestrzenie nazw a złożenia</i>	112
<i>Wyrażenia regularne</i>	120
<i>Specyfikatory formatu</i>	124
Specyfikatory formatu wizualnego	126
Specyfikatory formatu typu DateTime	127
<i>Opcje kompilatora języka C#</i>	129
<i>Istotne narzędzia środowiska .NET</i>	134
<i>Skorowidz</i>	139

Opakowywanie i odpakowywanie typów wartościowych

Aby można było wykonywać wspólne operacje — przeprowadzane zarówno na typach referencyjnych, jak i wartościowych — każdy typ wartościowy posiada odpowiadający mu ukryty typ referencyjny. Zostaje on utworzony w przypadku przypisania typu wartościowego do egzemplarza typu `System.Object` lub do interfejsu. Proces ten zwany jest *opakowywaniem* (ang. *boxing*). Typ wartościowy można rzutować do klasy `object` (wyjściowej klasy bazowej dla wszystkich typów wartościowych i referencyjnych) lub do implementowanego przez niego interfejsu.

W zaprezentowanym tu przykładzie opakowujemy typ wartościowy `int` w odpowiadający mu typ referencyjny, a następnie odpakowujemy go:

```
class Test {
    static void Main () {
        int x = 9;
        object o = x; // opakuj typ int
        int y = (int)o; // odpakuj typ int
    }
}
```

Podczas opakowywania typu wartościowego tworzony jest nowy typ referencyjny, który przechowywać będzie kopię opakowywanego typu. Operacja odpakowania powoduje przekopiowanie wartości z typu referencyjnego z powrotem do typu wartościowego. Wymaga ona jawnego rzutowania, podczas którego wykonywane jest sprawdzenie, czy docelowy typ wartościowy jest zgodny z typem zawartym w typie referencyjnym. W przypadku niepomyślnego wyniku tego sprawdzenia zgłaszany jest wyjątek `InvalidCastException`. Nie musimy się martwić o to, co stanie się z opakowanymi obiektami, gdy przestaniemy z nich korzy-

stać — poradzi sobie z nimi za nas mechanizm odśmiecania pamięci.

Dobrym przykładem operacji opakowania i odpakowania jest zastosowanie klas kolekcji (ang. *collection classes*). W następującym fragmencie kodu wykorzystujemy klasę `Queue` w połączeniu z typami wartościowymi:

```
using System;
using System.Collections;
class Test {
    static void Main () {
        Queue q = new Queue ();
        q.Enqueue (1); // opakuj wartość typu int
        q.Enqueue (2); // opakuj wartość typu int
        Console.WriteLine ((int)q.Dequeue()); // odpakuj
                                                    ↪wartość typu int
        Console.WriteLine ((int)q.Dequeue()); // odpakuj
                                                    ↪wartość typu int
    }
}
```

Typy predefiniowane

Wszystkie typy predefiniowane w języku C# są nazwami zastępczymi typów występujących w przestrzeni nazw `System`. Na przykład, pomiędzy następującymi dwoma instrukcjami istnieje jedynie różnica składniowa:

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

Typy całkowite

Typy całkowite oraz ich właściwości przedstawia zamieszczona poniżej tabela.

W przypadku typów całkowitych bez znaku, o szerokości n bitów, możliwe wartości należą do zakresu od 0 do 2^n-1 . W przypadku typów całkowitych ze znakiem, o takiej samej szerokości

Typ w języku C#	Typ systemowy	Rozmiar	Ze znakiem
sbyte	System.SByte	1 bajt	Tak
short	System.Int16	2 bajty	Tak
int	System.Int32	4 bajty	Tak
long	System.Int64	8 bajtów	Tak
byte	System.Byte	1 bajt	Nie
ushort	System.UInt16	2 bajty	Nie
uint	System.UInt32	4 bajty	Nie
ulong	System.UInt64	8 bajtów	Nie

(n bitów), możliwe wartości mieszczą się w zakresie od -2^{n-1} do $2^{n-1}-1$. W literałach całkowitych stosować można notację dziesiętną lub szesnastkową:

```
int x = 5;
ulong y = 0x1234AF; // w przypadku liczb
↳szesnastkowych stosujemy przedrostek 0x
```

W przypadku, gdy dany literał całkowity jest poprawny dla kilku możliwych typów całkowitych, typ domyślny wybierany jest w następującej kolejności: `int`, `uint`, `long` oraz `ulong`. W celu jawnego określenia wybranego typu można użyć następujących przyrostków:

U
 uint lub ulong

L
 long lub ulong

UL
 ulong

Konwersje całkowite

Niejawna konwersja pomiędzy typami całkowitymi jest dozwolona w przypadku, gdy typ, na który chcemy konwertować, zawiera wszystkie możliwe wartości typu konwertowanego. W przeciwnym razie konieczna jest konwersja jawna. Możemy, na przykład, niejawnie skonwertować typ `int` na typ `long`, lecz typ `int` na typ `short` musimy już konwertować jawnie:

```
int x = 123456;
long y = x; // konwersja niejawna, nie ma utraty
            ↳ informacji
short z = (short)x; // konwersja jawna, wartość x
                  ↳ zostaje "obcięta"
```

Typy zmiennoprzecinkowe

Typ w języku C#	Typ systemowy	Rozmiar
<code>float</code>	<code>System.Single</code>	4 bajty
<code>double</code>	<code>System.Double</code>	8 bajtów

Typ `float` może pomieścić wartości z zakresu od około $\pm 1,5 \times 10^{-45}$ do około $\pm 3,4 \times 10^{38}$, z dokładnością do siedmiu cyfr znaczących.

Typ `double` może pomieścić wartości z zakresu od około $\pm 5,0 \times 10^{-324}$ do około $\pm 1,7 \times 10^{308}$, z dokładnością do 15 – 16 cyfr znaczących.

Typy zmiennoprzecinkowe mogą przechowywać wartości specjalnie `+0`, `-0`, `+∞`, `-∞`, `NaN` (ang. *not a number* — wartość nieliczbowa). Reprezentują one wyniki takich operacji matematycznych jak dzielenie przez zero. Typy `float` oraz `double` stanowią implementację specyfikacji typów formatu IEEE 754, obsługiwaną przez prawie wszystkie procesory, a zdefiniowaną przez organizację IEEE w witrynie <http://www.ieee.org>.

W literałach zmiennoprzecinkowych stosować można notację dziesiętną lub wykładniczą. Literał typu `float` wymaga dołączenia przyrostka `f` lub `F`. Do literału typu `double` można (lecz nie jest to konieczne) dodać przyrostek `d` lub `D`.

```
float x = 9.81f;  
double y = 7E-02; // 0.07
```

Konwersje zmiennoprzecinkowe

Niejawna konwersja typu `float` na typ `double` nie powoduje utraty informacji i jest dozwolona — lecz nigdy na odwrót. Dla zapewnienia czytelności dozwolona jest również konwersja typów `int`, `uint` i `long` na typ `float` oraz typ `long` na typ `double`:

```
int sila = 2;  
int przesuniecie = 3;  
float x = 9.53f * sila - przesuniecie;
```

Jeśli w przykładzie tym użyjemy większych wartości, może nastąpić utrata precyzji. Jednak możliwy zakres wartości nie jest okrojony, ponieważ najniższa i najwyższa wartość zarówno typu `float`, jak i `double`, przekraczają najniższą i najwyższą wartość typów `int`, `uint` czy `long`. Wszystkie pozostałe konwersje pomiędzy typami całkowitymi a zmiennoprzecinkowymi muszą być jawne:

```
float x = 3.53f;  
int przesuniecie = (int)x;
```

Typ dziesiętny (decimal)

Typ `decimal` może przechowywać wartości z zakresu od $\pm 1,0 \times 10^{-28}$ do około $\pm 7,9 \times 10^{28}$, przy użyciu 28 – 29 cyfr znaczących.

Typ `decimal` zawiera 28 cyfr oraz położenie przecinka oddzielającego części dziesiętne. W przeciwieństwie do wartości zmien-

noprzecinkowej typ ten charakteryzuje się większą dokładnością, lecz mniejszym zakresem. Przydatny jest on zwykle w obliczeniach finansowych, gdzie połączenie jego wysokiej dokładności oraz możliwości zapisania liczby dziesiętnej bez błędów zaokrąglenia jest bardzo cenne. Na przykład liczba 0,1 zostanie za pomocą typu `decimal` przedstawiona dokładnie, lecz jeśli użyjemy do jej zapisania typu zmiennoprzecinkowego, jej reprezentacją będzie liczba dwójkowa okresowa. W przypadku typu `decimal` nie istnieją pojęcia wartości $+0$, -0 , $+\infty$, $-\infty$ czy NaN.

Literał dziesiętny wymaga dołączenia przyrostka `m` lub `M`.

```
decimal x = 80603.454327m; // przechowuje dokładna  
                             ↪wartość
```

Konwersje typu decimal

Dozwolona jest niejawna konwersja wszystkich typów całkowitych na typ dziesiętny (`decimal`), ponieważ typ ten reprezentować może dowolną wartość całkowitą. Konwersja typu `decimal` na typ zmiennoprzecinkowy (i odwrotnie) wymaga konwersji jawnej, ponieważ typy zmiennoprzecinkowe posiadają większy zakres niż typ dziesiętny, ten zaś posiada większą dokładność niż typy zmiennoprzecinkowe.

Typ znakowy (char)

Typ w języku C#	Typ systemowy	Rozmiar
<code>char</code>	<code>System.Char</code>	2 bajty

Typ `char` reprezentuje znak w formacie Unicode. Literał typu `char` może być albo znakiem prostym, albo znakiem w formacie Unicode, albo znakiem sterującym (ang. *escape character*). We wszystkich tych przypadkach użyty jako taki literał znak powinien być ujęty w apostrofy:

```
'A' // znak prosty
'\u0041' // znak w kodzie Unicode
'\x0041' // znak w kodzie szesnastkowym unsigned
    ↳short
'\n' // znak sekwencji sterującej
```

Znaki sekwencji sterującej zebrano w tabeli 1.

Tabela 1. Znaki sekwencji sterującej

Znak	Znaczenie	Wartość
\'	Apostrof	0x0027
\"	Cudzysłów	0x0022
\\	Lewy ukośnik	0x005C
\0	Zero	0x0000
\a	Alarm	0x0007
\b	Znak cofania	0x0008
\f	Wysuw strony	0x000C
\n	Nowy wiersz	0x000A
\r	Powrót karetki	0x000D
\t	Tabulacja pozioma	0x0009
\v	Tabulacja pionowa	0x000B

Konwersje typu char

Niejawna konwersja typu `char` na większość typów numerycznych jest zwykle wykonywana, przy czym jest ona zależna od tego, czy dany typ numeryczny może pomieścić typ `short` bez znaku. Jeśli nie — konieczna jest konwersja jawna.

Typ logiczny (*bool*)

Typ w języku C#	Typ systemowy	Rozmiar
-----------------	---------------	---------

<code>bool</code>	<code>System.Boolean</code>	1 bajt/2 bajty
-------------------	-----------------------------	----------------

Typ `bool` jest wartością logiczną, której przypisać można literał `true` lub `false`.

Chociaż wartość boolowska wymaga tylko jednego bitu (0 lub 1), zajmuje ona 1 bajt pamięci, gdyż jest to minimalna porcja, która może podlegać adresowaniu w przypadku większości architektur procesorów. Każdy element w tablicy boolowskiej zajmuje dwa bajty pamięci.

Konwersje typu `bool`

Nie są możliwe jakiegokolwiek konwersje wartości boolowskich na typy numeryczne i odwrotnie.

Typ `object`

Typ w języku C#	Typ systemowy	Rozmiar
<code>object</code>	<code>System.Object</code>	0-bajtowy/8-bajtowy narzut

Klasa `object` jest wyjściowym typem bazowym zarówno dla typów wartościowych, jak i referencyjnych. Typy wartościowe nie zawierają narzutu pamięci pochodzącego od obiektu. Typy referencyjne, które przechowywane są na sterckie, z natury wymagają narzutu. W środowisku uruchomieniowym .NET egzemplarz typu referencyjnego posiada 8-bajtowy narzut, który zawiera typ obiektu, a także informacje tymczasowe, dotyczące na przykład stanu blokady synchronizacji lub tego, czy dany obiekt został zabezpieczony przed usunięciem przez mechanizm odświeżania pamięci. Warto wiedzieć, że każda referencja do egzemplarza typu referencyjnego zajmuje 4 bajty pamięci.