

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# 3.0. Leksykon kieszonkowy. Wydanie II

Autor: Joseph Albahari, Ben Albahari

Tłumaczenie: Przemysław Szeremiota

ISBN: 978-83-246-1827-9

Tytuł oryginału: [C# 3.0 Pocket Reference](#):

[Instant Help for C# 3.0 Programmers, 2nd Edition](#)

Format: 115x170, stron: 280



Poznaj nowości języka C# i podnieś wydajność programowania

- Co nowego w C# 3.0?
- Jak skrócić i usprawnić swój kod?
- Do czego służy mechanizm LINQ?

C# jest obiektowym językiem programowania zalecanym przez Microsoft dla platformy .NET Framework. Pozwala definiować wiele różnorodnych elementów składowych klas, nie tylko pola czy metody. Analiza struktury kodu umożliwia tworzenie wysoce uniwersalnych mechanizmów operujących na strukturze kodu nieznaney w czasie kompilacji. Wiedza programisty jest kluczem do wykorzystania wszystkich jego możliwości. Leksykon stanowi bogate kompendium nowych rozwiązań dostępnych w C# 3.0 oraz ich implementacji. Opisane zagadnienia, dotyczące mechanizmu LINQ (Language Integrated Query), pozwalają na pozyskanie praktycznej wiedzy niezbędnej we współczesnym programowaniu. Książka C# 3.0 Leksykon kieszonkowy. Wydanie II, poruszająca w sposób przejrzysty i rzeczowy całość pojęciowych zmian koniecznych do opanowania C#, jest idealną pozycją dla wszystkich programistów, którym nieobca jest Java, C++ lub poprzednie wersje C#.

Do najważniejszych cech wyróżniających język C# w wydaniu 3.0 zaliczamy:

- wyrażenia lambda,
- metody rozszerzające,
- niejawne typowanie zmiennych lokalnych,
- składnię ujmowania zapytań w kodzie,
- typy anonimowe,
- niejawne typowanie tablic,
- inicjalizatory obiektów,
- właściwości automatyczne,
- metody częściowe,
- drzewa wyrażień.

Nie trać czasu na szukanie informacji!

Programuj wydajnie i efektywnie z kieszonkowym leksykonem!



Spis treści

Nowości w C# 3.0	11
Pierwszy program w C#	15
Kompilacja	18
Składnia	19
Identyfikatory i słowa kluczowe	20
Literały, znaki interpunkcyjne i operatory	22
Komentarze do kodu	23
System typów	23
Przykłady typów predefiniowanych	24
Przykłady typów własnych	25
Konwersje	29
Typy wartościowe a typy referencyjne	30
Taksonomia typów predefiniowanych	34
Typy liczbowe	35
Literały liczbowe	36
Konwersje liczbowe	38
Operatory arytmetyczne	40
Operatory inkrementacji i dekrementacji	40
Specjalizowane operacje na liczbach całkowitych	41
Liczby całkowite 8- i 16-bitowe	43

Wyróżnione wartości zmiennoprzecinkowe	43
double kontra decimal	45
Błędy zaokrąglania liczb rzeczywistych	46
Typ wartości logicznych i operatory logiczne	47
Operatory porównania i relacji	47
Operatory logiczne	48
Znaki i ciągi znaków	49
Konwersje znaków	50
Typ string	51
Tablice	54
Domyślna inicjalizacja elementów tablic	55
Tablice wielowymiarowe	57
Wyrażenia uproszczonej inicjalizacji tablic	58
Kontrola zakresów	60
Zmienne i parametry	61
Stos i sarta	61
Przypisania oznaczone	63
Wartości domyślne	64
Parametry	64
Niejawne typowanie zmiennych lokalnych (C# 3.0) (var)	69
Operatory i wyrażenia	70
Wyrażenia proste	71
Wyrażenia bezwartościowe	71
Wyrażenia przypisania	71
Priorytety i łączność operatorów	72
Tabela operatorów	73

Instrukcje	77
Instrukcje deklaracji	77
Instrukcje wyrażeniowe	78
Instrukcje warunkowe	79
Instrukcje iteracyjne	83
Instrukcje skoku	86
Inne instrukcje	88
Przestrzenie nazw	89
Dyrektywa using	90
Reguły obowiązujące w przestrzeniach nazw	91
Synonimy typów i przestrzeni nazw	93
Klasy	93
Pola	94
Metody	95
Konstruktory instancji	96
Inicjalizatory obiektów (C# 3.0)	98
Referencja this	99
Właściwości	100
Indeksery	102
Stałe	104
Konstruktory statyczne	105
Klasy statyczne	106
Finalizatory	106
Klasy i metody częściowe	107
Dziedziczenie	108
Polimorfizm	110
Rzutowanie	110
Wirtualne funkcje składowe	112
Klasy i składowe abstrakcyjne	114
Widoczność składowych dziedziczonych	114

Pieczerowanie funkcji i klas	115
Słowo kluczowe base	116
Konstruktory a dziedziczenie	117
Przeciążanie a rozstrzyganie wywołań	118
Typ object	119
Pakowanie i odpakowywanie	120
Statyczna i dynamiczna kontrola typów	121
Wykaz składowych klasy object	122
Metoda GetType() i operator typeof	122
Equals, ReferenceEquals i GetHashCode	123
Metoda ToString	124
Struktury	125
Semantyka konstrukcji struktury	126
Modyfikatory dostępu	127
Przykłady	128
Narzucanie dostępności	128
Ograniczenia modyfikatorów dostępu	129
Interfejsy	129
Rozszerzanie interfejsu	131
Jawna implementacja interfejsu	131
Wirtualne implementowanie składowych interfejsów	132
Ponowna implementacja interfejsu w klasie pochodnej	133
Typy wyliczeniowe	135
Konwersje typów wyliczeniowych	136
Flagi	137
Operatory typów wyliczeniowych	138

Typy zagnieżdżone	139
Uogólnienia	140
Typy uogólnione	140
Po co nam uogólnienia	142
Metody uogólnione	143
Deklarowanie parametrów uogólnienia	144
Uogólnienia a typeof	145
Domyślna wartość uogólnienia	146
Ograniczenia uogólnień	146
Uogólnienia i kowariancja	148
Pochodne typów uogólnionych	149
Deklaracje uogólnień z odwołaniami do siebie	150
Dane statyczne	150
Inicjalizowanie kolekcji uogólnionych	151
Delegaty	151
Pisanie metod-wtyczek dla delegatów	153
Delegaty wielokrotne	153
Metody statyczne i metody instancji jako metody docelowe	155
Uogólnione typy delegatów	155
Zgodność delegatów	155
Zdarzenia	157
Standardowy schemat zdarzenia	159
Aksesory zdarzenia	163
Modyfikatory zdarzeń	164
Wyrażenia lambda (C# 3.0)	164
Jawne określanie typów parametrów lambda	166
Uogólnione wyrażenia lambda i delegaty Func	166
Zmienne zewnętrzne	167

Metody anonimowe	169
Wyjątki i instrukcja try	170
Klauzula catch	173
Blok finally	174
Zgłaszanie wyjątków	176
Najważniejsze właściwości klasy System.Exception	178
Najczęstsze wyjątki	178
Enumeratory i iteratory	180
Enumeracja	180
Iteratory	181
Semantyka iteratora	183
Składanie sekwencji	184
Typy z dopuszczalną wartością pustą	185
Podstawy	185
Pożyczanie operatorów	188
Typ bool?	190
Operator ??	191
Przeciążanie operatorów	191
Funkcje operatorów	192
Przeciążanie operatorów porównania i relacji	193
Własne konwersje jawne i niejawne	194
Metody rozszerzające (C# 3.0)	196
Kaskadowe wywołania metod rozszerzających	197
Niejednoznaczność i rozstrzygnięcie niejednoznaczności	197
Typy anonimowe (C# 3.0)	199

LINQ (C# 3.0)	200
Podstawy LINQ	200
Opóźnione wykonanie	205
Standardowe operatory zapytań	207
Kaskadowe operatory zapytań	213
Składnia zapytań	214
Słowo kluczowe let	218
Kontynuacje zapytań	219
Zapytania z wieloma generatorami	219
Złączenia	221
Porządkowanie	224
Grupowanie	225
Operatory OfType i Cast	228
Atrybuty	229
Klasy atrybutów	229
Parametry nazwane i pozycyjne	230
Obiekt docelowy atrybutu	231
Nadawanie wielu atrybutów	231
Definiowanie własnych atrybutów	232
Odwołania do atrybutów w czasie wykonania	233
Wskaźniki i kod nienadzorowany	234
Elementarz wskaźników	234
Kod nienadzorowany	235
Instrukcja fixed	235
Operator dostępu do składowej przez wskaźnik	237
Tablice	237
Typ void*	238
Wskaźniki do kodu nienadzorowanego	239

Dyrektywy preprocesora	239
Atrybuty warunkowe	241
Ostrzeżenia i pragma	242
Dokumentacja XML	243
Standardowe znaczniki dokumentacji XML	244
Przegląd infrastruktury i środowiska	247
Rdzeń infrastruktury	248
Technologie interfejsu użytkownika	256
Technologie zaplecza aplikacji	261
Technologie systemów rozproszonych	263
Skorowidz	267

C# 3.0.

Leksykon kieszonkowy. Wydanie II

C# to obiektowy język programowania ogólnego przeznaczenia, z kontrolą typów, w którym największe znaczenie ma produktywność programisty. Aby ją zwiększyć, w języku należało zrównoważyć prostotę, wyrazistość i wydajność. Język C# jest neutralny wobec platformy, ale najlepiej współpracuje z .NET Framework firmy Microsoft. Dla C# w wersji 3.0 najodpowiedniejsza jest platforma .NET Framework 3.5.

Nowości w C# 3.0

Nowości wprowadzone w C# 3.0 koncentrują się wokół mechanizmu integracji zapytań w kodzie — *Language Integrated Query*, w skrócie LINQ. Pozwala on na stosowanie zapytań w stylu SQL wprost w programie C#. Zapytania takie mają tę zaletę, że podlegają *statycznej* kontroli poprawności. Mogą być wykonywane na zbiorach lokalnych i zdalnych; platforma .NET Framework udostępnia interfejsy uwzględniające mechanizm lokalnych zapytań LINQ dla wszelkich kolekcji, zdalnych baz danych oraz do plików XML.

Do najważniejszych cech wyróżniających język C# w wydaniu 3.0 zaliczymy:

- wyrażenia lambda,
- metody rozszerzające,
- niejawne typowanie zmiennych lokalnych,
- składnię ujmowania zapytań w kodzie,

- typy anonimowe,
- niejawne typowanie tablic,
- inicjalizatory obiektów,
- właściwości automatyczne,
- metody częściowe,
- drzewa wyrażeń.

Wyrażenia *lambda* stanowią miniatury funkcji, definiowanych „w locie”, to jest w miejscu wystąpienia wyrażenia. Stanowią naturalne rozwinięcie metod anonimowych wprowadzonych do C# 2.0 i w zasadzie całkowicie wypierają funkcjonalność metod anonimowych. Oto przykład:

```
Func<int,int> sqr = x => x * x;
Console.WriteLine (sqr(3)); //9
```

Najważniejszym zastosowaniem wyrażeń *lambda* w języku C# są zapytania LINQ, takie jak poniższe:

```
string[] names = { "Arek", "Ala", "Gosia" };

// Tylko imiona o długości >= 4 znaki

IEnumerable<string> filteredNames =
    Enumerable.Where (names, n => n.Length >= 4);
```

Metody *rozszerzające* służą do uzupełniania już istniejących typów o nowe metody, bez zmieniania pierwotnej definicji typu. Pełnią rolę skrótowca składniowego, ponieważ ich działanie sprowadza się do upodobnienia wywołań metod statycznych do wywołań metod na rzecz instancji. Ponieważ operatory LINQ są zaimplementowane właśnie jako metody *rozszerzające*, możemy poprzednie zapytanie uprościć do następującej postaci:

```
IEnumerable<string> filteredNames =
    names.Where (n => n.Length >= 4);
```

Niejawne typowanie zmiennych lokalnych to zezwolenie na pominięcie typu zmiennej w instrukcji deklaracji i zdanie się na wnioskowanie typu przez kompilator. Ponieważ kompilator potrafi samodzielnie określić typ zmiennej `filteredNames`, możemy nasze zapytanie uprościć jeszcze bardziej:

```
var filteredNames = names.Where (n => n.Length >= 4);
```

Składnia zapytania ujętego w kodzie służy do stosowania składni LINQ upodobnionej do składni zapytań SQL zamiast zapytań konstruowanych z wywołań operatorów. Dzięki temu znacząco upraszcza się zapis wielu rodzajów zapytań; ponownie mamy do czynienia ze skrótowcem składniowym, tym razem dla zamaskowanych wyrażeń lambda. Oto poprzedni przykład, przepisany z użyciem składni ujęcia zapytania w kodzie:

```
var filteredNames = from n in names
                    where n.Length >= 4
                    select n;
```

Typy anonimowe to proste klasy tworzone w locie, wykorzystywane często do reprezentowania wyników zapytań:

```
var query = from n in names where n.Length >= 4
            select new {
                Name = n,
                Length = n.Length
            };
```

Oto prostszy przykład:

```
var dude = new { Name = "Robert", Age = 20 };
```

Niejawne typowanie tablic to mechanizm pozwalający na wyeliminowanie obowiązku określania typu elementów tablicy przy konstruowaniu i inicjalizowaniu tablicy za jednym zamachem:

```
var dudes = new[]
{
    new { Name = "Robert", Age = 20 }
    new { Name = "Roman", Age = 30 }
};
```

Inicjalizatory obiektów upraszczają konstruowanie instancji klas, pozwalając na ustawienie właściwości już przy wywołaniu konstruktora obiektu. Inicjalizatory obiektów można stosować z typami anonimowymi i z typami nazwanymi. Oto przykład:

```
Bunny b1 = new Bunny {  
    Name = "Bo",  
    LikesCarrots = true  
};
```

W języku C# w wydaniu 2.0 odpowiednikiem powyższego byłby taki kod:

```
Bunny b2 = new Bunny();  
b2.Name = "Bo";  
b2.LikesCarrots = true;
```

Właściwości automatyczne oszczędzają pisanie kodu właściwości sprowadzających się do prostych akcesorów ustawiających i odczytujących wartość prywatnego pola docelowego. W poniższym przykładzie kompilator automatycznie wygeneruje prywatne pole i akcesory dla właściwości X:

```
public class Stock  
{  
    public decimal X { get; set; }  
}
```

Metody częściowe pozwalają w klasach generowanych automatycznie na dodawanie ręcznych uzupełnień. Mechanizm LINQ do SQL wykorzystuje metody częściowe dla generowanych klas odwzorowania tabel SQL.

Drzewa wyrażeń to miniaturowe obiektowe modele wyrażeń, opisujące wyrażenia lambda. Kompilator C# w wersji 3.0 generuje drzewo wyrażenia dla każdego wyrażenia lambda przypisanego do instancji specjalnego typu `Expression<TDelegate>`:

```
Expression<Func<string,bool>> predicate =  
    s => s.Length > 10;
```

Drzewa wyrażień pozwalają na zdalne (np. na serwerze baz danych) wykonywanie zapytań LINQ, bo są dostępne do refleksji i tłumaczenia w czasie wykonania (można je więc tłumaczyć np. na zapytania SQL).

Pierwszy program w C#

Oto program wyliczający iloczyn $12 * 30$ i wypisujący wynik (360) na konsoli. Podwójne znaki ukośników umieszczone w kodzie oznaczają, że reszta danego wiersza to *komentarz do kodu*:

```
using System;                                // Import przestrzeni nazw

class Test                                    // Deklaracja klasy
{
    static void Main()                        // Deklaracja metody w klasie
    {
        int x = 12 * 30;                     // Instrukcja (1)
        Console.WriteLine (x);              // Instrukcja (2)
    }                                         // Koniec metody
}                                             // Koniec klasy
```

Sedno tego programu tkwi w dwóch *instrukcjach*. Instrukcje w języku C# są wykonywane sekwencyjnie (jedna po drugiej). Każda instrukcja kończy się znakiem średnika:

```
int x = 12 * 30;
Console.WriteLine (x);
```

Pierwsza z tych instrukcji oblicza wartość *wyrażenia* $12 * 30$ i zapisuje ustalony wynik w *zmiennej lokalnej* o nazwie *x*, typu całkowitego. Druga instrukcja wywołuje metodę `WriteLine` z klasy `Console`; metoda ta wypisuje wartość zmiennej *x* na konsolę, czyli tekstowe okno na ekranie.

Metoda jest miejscem wykonywania akcji w postaci szeregu instrukcji, zwanego *blokiem instrukcji* — z parą nawiasów klamrowych i dowolną liczbą (zerem albo więcej) instrukcji pomiędzy nimi. W naszym programie zdefiniowaliśmy prostą metodę o nazwie `Main`:

```

static void Main()
{
    ...
}

```

Kod programu można uprościć przez napisanie funkcji wysokopoziomowych, wywołujących funkcje niższego poziomu. Nasz program można by przerobić tak, aby wielokrotnie wykorzystywał zmodyfikowaną metodę obliczającą iloczyn argumentu i liczby 12, jak tutaj:

```

using System;

class Test
{
    static void Main()
    {
        Console.WriteLine (FeetToInches (30));           //360
        Console.WriteLine (FeetToInches (100));         //1200
    }

    static int FeetToInches (int feet)
    {
        int inches = feet * 12;
        return inches;
    }
}

```

Metoda może pobierać *dane wejściowe* od wywołującego za pośrednictwem zadeklarowanych *parametrów*, a także zwracać *dane wyjściowe* do wywołującego, za pośrednictwem zadeklarowanej *wartości zwracanej*. W powyższym przykładzie zdefiniowaliśmy metodę o nazwie `FeetToInches` (przeliczającą stopy na cale) z parametrem dla wejściowej liczby stóp i wartością zwracaną dla wynikowej liczby cali:

```

static int FeetToInches (int feet) {...}

```

Argumentami przekazywanymi do metody `FeetToInches` są *literały* (wartości liczbowe) 30 oraz 100. Metoda `Main` z naszego przykładu nie zawiera pomiędzy nawiasami w deklaracji żadnych

parametrów, ponieważ nie przyjmuje żadnych argumentów; z kolei typ wartości zwracanej w metodzie `Main` został ustalony jako `void`, ponieważ metoda nie zwraca żadnych wartości do wywołującego:

```
static void Main()
```

Język C# rozpoznaje metodę o nazwie `Main` jako domyślny punkt wejścia do programu, czyli miejsce rozpoczęcia wykonywania 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”).

W języku C# metody to jeden z kilku wyróżnionych rodzajów funkcji. W naszym przykładzie wykorzystaliśmy też funkcję innego rodzaju: *operator* `*`, zastosowany do obliczenia iloczynu. Do tego dochodzą jeszcze *konstruktory* (ang. *constructors*), *właściwości* (ang. *properties*), *zdarzenia* (ang. *events*), *indeksery* (ang. *indexers*) i *finalizatory* (ang. *finalizers*).

W naszym przykładzie obie zdefiniowane metody zostały ujęte we wspólnej klasie. *Klasa* to konstrukcja grupująca składowe — funkcje i dane — w obiektowy „klocek”. Na przykład klasa `Console` grupuje składowe wykorzystywane do obsługi tekstowych strumieni wejścia-wyjścia, w tym choćby metodę `WriteLine`. W naszej klasie `Test` zgrupowaliśmy dwie metody: metodę `Main` i metodę `FeetToInches`. Klasa definiuje *typ*, o którym powiemy więcej w podrozdziale „System typów”.

Na najbardziej zewnętrznym poziomie programu typy są zorganizowane w *przestrzeniach nazw* (ang. *namespaces*). Dyrektywa `using` wprowadziła do naszego programu przestrzeń nazw `System`, dzięki czemu mogliśmy się odwołać do zdefiniowanego w tej przestrzeni typu `Console` bez konieczności stosowania przedrostka `System` (`System.Console`). Ze swojej strony moglibyśmy wszystkie nasze klasy umieścić we wspólnej przestrzeni nazw `TestPrograms`, jak tutaj:

```
using System;

namespace TestPrograms
{
    class Test1 {...}
    class Test1 {...}
}
```

Cała platforma .NET Framework jest zorganizowana w zagnieżdżonych w sobie przestrzeniach nazw. Na przykład typy do obsługi tekstu zawiera przestrzeń nazw:

```
using System.Text;
```

Dyrektywa `using` jest stosowana jedynie dla wygody skróconego zapisywania następnych odwołań do elementów przestrzeni nazw. Nie jest obowiązkowa i można zamiast niej stosować kwalifikowane nazwy typów, a więc poprzedzać właściwe nazwy typów przedrostkami z nazwami przestrzeni nazw, w których te typy są definiowane, np. `System.Text.StringBuilder`.

Kompilacja

Kompilator języka C# kompiluje kod źródłowy, dany w postaci zbioru plików o rozszerzeniu `.cs`, w tak zwany *zestaw* (ang. *assembly*). Zestaw jest w .NET jednostką mechanizmu budowania i rozmieszczania, i może być zarówno aplikacją, jak biblioteką. Zwyczajna *aplikacja* przeznaczona dla środowiska okienkowego albo dla konsoli posiada metodę `Main` i ma postać pliku wykonywalnego z rozsze-

rzeniem *.exe*. Z kolei *biblioteka*, z rozszerzeniem *.dll*, jest odpowiednikiem aplikacji, tyle że bez wyróżnionego punktu wejścia do programu. Biblioteka ma służyć do realizacji wywołań (odwołań) z innych aplikacji i bibliotek. Sama platforma .NET Framework stanowi właśnie zestaw bibliotek.

Kompilator języka C# to plik wykonywalny o nazwie *csc.exe*. Można go stosować z wnętrza środowiska programistycznego takiego jak Visual Studio .NET, w którym kompilator jest wywoływany automatycznie, albo przeprowadzać kompilację ręcznie, z poziomu wiersza poleceń. Aby ręcznie skompilować program, należy najpierw zapisać kod źródłowy programu w pliku, np. *MyFirstProgram.cs*, a potem w wierszu poleceń wywołać polecenie *csc* (z `<Windows>\Microsoft .NET\Framework\<wersja platformy>`):

```
csc MyFirstProgram.cs
```

W taki sposób powstanie aplikacja o nazwie *MainFirstProgram.exe*; gdybyśmy chcieli utworzyć bibliotekę (*.dll*), powinniśmy wydać polecenie:

```
csc /target:library MyFirstProgram.cs
```

Składnia

Składnia C# wywodzi się ze składni języków C i C++. W tym podrozdziale opiszemy elementy składni języka C# na przykładzie poniższego programu:

```
using System;

class Test
{
    static void Main()
    {
        int x = 12 * 30;
        Console.WriteLine (x);
    }
}
```

Identyfikatory i słowa kluczowe

Identyfikatory są nazwami wybieranymi przez programistę jako nazwy dla klas, metod, zmiennych itd. W naszym kodzie przykładowym występują następujące identyfikatory (w kolejności, w jakiej pojawiają się w tekście programu):

```
System Test Main x Console WriteLine
```

Identyfikator musi mieć charakter słowa, to znaczy zawierać znaki Unicode oraz zaczynać się od litery, ewentualnie od znaku podkreślenia. W identyfikatorach języka C# wielkość liter jest istotna. Przyjęło się, że: 1) argumenty funkcji, zmienne lokalne i pola prywatne zapisuje się zgodnie z tak zwaną konwencją wielbłądzą (ang. *camel case*), czyli np. mojaZmienna (pierwszy wyraz składowy małą literą, pozostałe wyrazy wyróżniane wielką literą); 2) wszystkie pozostałe identyfikatory zapisuje się w konwencji pascalowej, np. MojaMetoda (wszystkie wyrazy składowe wielką literą).

Z kolei *słowa kluczowe* są nazwami zarezerwowanymi dla kompilatora; nie można ich wykorzystać jako identyfikatorów dla własnych elementów programu. Oto lista słów kluczowych występujących w naszym przykładowym kodzie:

```
using class static void int
```

A oto pełna lista słów kluczowych w języku C#:

abstract	class	event	if
as	const	explicit	implicit
base	continue	extern	in
bool	decimal	false	int
break	default	finally	interface
byte	delegate	fixed	internal
case	do	float	is
catch	double	for	lock
char	else	foreach	long
checked	enum	goto	namespace

new	readonly	struct	unsafe
null	ref	switch	ushort
object	return	this	using
operator	sbyte	throw	virtual
out	sealed	true	volatile
override	short	try	void
params	sizeof	typeof	while
private	stackalloc	uint	
protected	static	ulong	
public	string	unchecked	

Unikanie kolizji nazw

Jeśli koniecznie chcemy zastosować identyfikator, który koliduje z jednym ze słów kluczowych, możemy kwalifikować nasz identyfikator znakiem @. Oto przykład:

```
class class {...} // Niedozwolone
class @class {...} // Dozwolone
```

Symbol @ nie wchodzi formalnie w skład samego identyfikatora, więc @mojaZmienna to dla kompilatora to samo co mojaZmienna.

Kontekstowe słowa kluczowe

W języku C# występują też słowa, które nie są słowami kluczowymi w ścisłym znaczeniu, ale również mają specjalne znaczenie. Są to tak zwane *kontekstowe słowa kluczowe*, które można wykorzystywać również jako identyfikatory, i to bez symbolu @. Wśród kontekstowych słów kluczowych mamy:

add	get	let	set
ascending	global	on	value
by	group	orderby	var
descending	in	partial	where
equals	into	remove	yield
from	join	select	

Stosowanie kontekstowych słów kluczowych jako identyfikatorów jest dozwolone, pod warunkiem że w kontekście wystąpienia identyfikatora nie pojawi się niejednoznaczność co do jego charakteru.