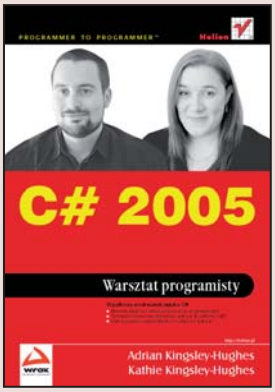
C# 2005. Warsztat programisty



Autor: Adrian Kingsley-Hughes, Kathie Kingsley-Hughes
Tłumaczenie:Wojciech Demski
ISBN: 978-83-246-1065-5
Tytuł oryginału: C# 2005 Programmers Reference
Format: B5, stron: 392
oprawa twarda

Wyjątkowy podręcznik języka C#

- Elementy języka i podstawowe koncepcje programistyczne
- Techniki tworzenia najwyższej klasy aplikacji dla platformy .NET
- Metody poprawy bezpieczeństwa i wydajności aplikacji

Język C#, pomimo stosunkowo krótkiej obecności na rynku, zyskał ogromną popularność i jest wykorzystywany przez wielu programistów. Trudno się temu dziwić – ten łączący w sobie najlepsze cechy Javy i C++ obiektowy język programowania jest niezwykle uniwersalny. Można wykorzystać go w niemal każdym projekcie – programów dla systemu Windows, dynamicznych witryn internetowych w technologii ASP.NET oraz aplikacji mobilnych dla platformy PocketPC. Prosta składnia, rozbudowane mechanizmy obsługi wyjątków i dostęp do ogromnej biblioteki klas .NET sprawiają, że C# jest doskonałym narzędziem.

Książka „C# 2005. Warsztat programisty" to połączenie kursu i podręcznika programowania w tym języku. Czytając ją, opanujesz zarówno podstawy C#, jak i zaawansowane zagadnienia związane z tworzeniem bezpiecznych i wydajnych aplikacji. Poznasz typy danych, zasady stosowania zmiennych oraz składnię instrukcji i wyrażeń. Przeczytasz o programowaniu obiektowym, typach generycznych, obsłudze wyjątków i technikach programowania w C#. Znajdziesz tu również omówienie gramatyki języka C#, konwencji nazewniczych i zasad umieszczania komentarzy w dokumentacji XML.

- Narzędzia programistyczne
- Struktura języka C#
- Typy danych
- Wyrażenia i instrukcje
- Przestrzenie nazw
- Klasy i obiekty
- Typy wyliczeniowe
- Obsługa wyjątków
- Typy generyczne

Poznaj nowoczesne metody tworzenia aplikacji

Wydawnictwo Helion
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

4

Struktura języka C#

Aby pisać dobre programy w C#, należy w pełni zrozumieć strukturę tego języka. W rozdziale tym omówimy językową, czyli leksykalną strukturę programów napisanych w C#.

Oto kolejność, w jakiej będziemy przedstawiać tematy:

- Programy C#
- Gramatyka
- Zakończenia wierszy
- Komentarze
- Białe znaki
- Tokeny
- Słowa kluczowe
- Dyrektywy

Programy C#

Każdy program C# tworzony jest z jednego bądź wielu **plików źródłowych**. Te pliki, zwane też jednostkami kompilacji, mogą być odrębnymi plikami tekstowymi lub też plikami zawartymi wewnątrz zintegrowanego środowiska IDE (ang. *Integrated Development Environment*), jak Visual Studio.

Jednostki kompilacyjne zawierają uporządkowane sekwencje znaków Unicode (okrężne określenie tekstu) i aby zapewnić maksymalną przenaszalność, wszystkie pliki źródłowe powinny być zakodowane w standardzie UTF-8. Korzystając z prostego edytora tekstowego (jakim jest Notatnik) lub środowiska programistycznego specyficznego dla C#, możemy mieć pewność, że kod otrzyma poprawny format.

Jednostka kompilacji składa się z:

- Zera lub więcej dyrektyw `using`
- Zera lub więcej atrybutów globalnych
- Zera lub więcej deklaracji składowych przestrzeni nazw

Atrybut jest obiektem reprezentującym dane, które zamierzamy powiązać z elementem programu, podczas gdy element, z którym wiążemy ten atrybut, nazywamy adresatem atrybutu.

Wszystkie elementy jednostki kompilacji pełnią określoną rolę:

- **Dyrektywy `using`.** Umożliwiają użycie przestrzeni nazw (które wykorzystuje się do logicznego rozmieszczania klas, struktur, interfejsów, wyliczeń i delegacji) oraz typów zdefiniowanych w innych przestrzeniach nazw. To wpływa na atrybuty i deklaracje składowych przestrzeni nazw jednostki kompilacji. Dyrektywa `using` zawarta w jednej jednostce kompilacji nie ma wpływu na pozostałe jednostki.
- **Atrybuty globalne.** Umożliwiają zdefiniowanie atrybutów dla całego projektu. Asemblacje i moduły pełnią rolę fizycznych pojemników na typy (lub jako miejsce dla kodu; nieco później powiemy o tym bardziej szczegółowo). Asemblacja może składać się z kilku odrębnych modułów lub, w przypadku prostszych projektów, z pojedynczego.
- **Deklaracje składowych przestrzeni nazw.** Tworzą jedną deklarację przestrzeni zwanej globalną przestrzenią nazw.

Gdy program C# jest kompilowany, wówczas wszystkie jednostki kompilacji przetwarzane są razem, co oznacza, że są powiązane zależnościami (jeżeli program składa się z więcej niż jednej jednostki kompilacji, to kompilator, aby być w stanie poprawnie skompilować kod źródłowy, musi mieć dostęp do wszystkich jednostek).

Kompilacja programu C# przebiega w trzech etapach:

- **Transformacja.** Ten etap polega na przekształceniu kodu na zbiór znaków Unicode (ze standardu, w którym kod zawarty w jednostkach kompilacji został zapisany i zakodowany).
- **Analiza leksykalna.** Jest to proces przekształcania znaków Unicode w strumień tokenów.
- **Analiza syntaktyczna.** Na tym etapie, poprzedzającym konwersję na postać wykonywalną, strumieniowi tokenów nadawany jest format Microsoft Intermediate Language (MSIL).

W C# wyróżnia się kilka rodzajów tokenów:

- Identyfikatory
- Słowa kluczowe
- Literały
- Operatory
- Separatory

Białe znaki i komentarze nie są tokenami.

Kompilator powinien być w stanie przekształcić jednostki kompilacji zapisane w formacie Unicode lub pliki źródłowe zakodowane w standardzie UTF-8 na postać sekwencji znaków Unicode. Może się zdarzyć, że kompilator będzie zdolny skompilować jednostki kompilacji zakodowane przy użyciu innych schematów (na przykład UTF-16 czy UTF-32), ale nie należy na to liczyć.

Gramatyka

W języku C# stosuje się gramatykę dwojakiego rodzaju:

- **Gramatykę leksykalną.** Określa sposób, w jaki znaki Unicode formują:
  - Zakończenia wierszy
  - Białe znaki
  - Komentarze
  - Tokeny
  - Dyrektywy preprocesora
- **Gramatykę syntaktyczną.** Ten typ gramatyki określa schemat łączenia tokenów, które powstały na podstawie reguł gramatyki leksykalnej, w programy C#.

Dwuznaczności gramatyczne

W każdym języku programowania można napotkać pewne dwuznaczności. Dla przykładu rozważmy takie oto wyrażenie:

```
F(X<Y, Z>(5));
```

To proste wyrażenie można zinterpretować na dwa sposoby:

1. Jako wywołanie `F` z dwoma argumentami: `X<Y` i `Z>(5)`
2. Jako wywołanie `F` z jednym argumentem, którym jest wywołanie metody generycznej `X` mającej dwa argumenty typu (czyli argumenty, które po prostu są typami) oraz jeden argument zwykły.

Na szczęście istnieją pewne reguły, według których postępuje kompilator, rozstrzygając takie niejednoznaczności. Napotykając kod, jak w powyższym przykładzie (gdzie mamy sekwencję tokenów tworzących listę argumentów typu), kompilator zwraca uwagę przede wszystkim na token znajdujący się bezpośrednio za znakiem zamykającym `>`. Jeśli jest to jeden z następujących tokenów:

- `(`
- `)`

- [
- :
- ;
- ,
- .
- ?
- ==
- !=

lista argumentów typu definiuje część prostej nazwy, dostęp do składowej lub dostęp do wskaźnika poprzedzającego i wszystkie dalsze opcje są odrzucane.

Jeśli następnym tokenem jest token, którego nie ma na powyższym wykazie, to lista argumentów typu nie definiuje części nazwy, dostępu do składowej ani dostępu do wskaźnika poprzedzającego.

> *Powyższa reguła nie odnosi się do parsowania listy argumentów typu w przestrzeni nazw czy nazwach typów.*

Wracając do naszego wcześniejszego, dość niejednoznacznego przykładu:

```
F(X<Y, Z>(5));
```

Zgodnie z regułami, które wymieniliśmy wcześniej, zapis ten zostanie zinterpretowany jako wywołanie `F` z jednym argumentem, który stanowi wywołanie metody generycznej `X`, o dwóch argumentach typów i jednym argumencie zwykłym.

Oto dwa następne przykłady wyrażeń, które zostaną zinterpretowane jako wywołanie `F` z dwoma argumentami:

```
F(X<Y, Z>5);
F((X<Y, Z>>5);
```

Omówmy inne wyrażenie:

```
X = F<Y> + Z;
```

Wyrażenie to z perspektywy użytych operatorów zostanie zinterpretowane jako:

- Operator „mniejszy niż” <
- Operator „większy niż” >
- Unarny operator +

Inna forma zapisu powyższego wyrażenia wygląda tak:

```
X = (F < Y) > (+Z);
```

Analiza leksykalna

Każdy plik źródłowy programu C# musi być zgodny z następującym gramatycznym wzorcem leksykalnym:

```
wejście:
   wejście-sekcjaopcj

wejście-sekcja:
   wejście-sekcja-część
   wejście-sekcja   wejście-sekcja-część

wejście-sekcja-część:
   elementy wejściowe opcj    nowy-wiersz
   dyrektywa-pp

elementy-wejściowe
   element-wejściowy
   elementy-wejściowe   wejście-element

wejście-element:
   biały znak
   komentarz
   token
```

Leksykalną strukturę jednostki kompilacji C# tworzy pięć elementów podstawowych. Są to:

- Zakończenia wierszy
- Białe znaki
- Komentarze
- Tokeny
- Dyrektywy preprocesora

Spośród tych elementów jedynie tokeny mają znaczenie z punktu widzenia gramatyki syntaktycznej dowolnego programu C# (z wyjątkiem sytuacji, gdy token > występuje łącznie z innym tokenem, tworząc z nim operator). Kiedy kompilator dokonuje leksykalnego przetwarzania jednostki kompilacji, to traktuje plik jako serię tokenów, które stanowią wejście dla późniejszego przetwarzania syntaktycznego. Zakończenia wierszy, białe znaki i komentarze oddzielające token y są elementami wyłącznie leksykalnymi i nie mają żadnego wpływu na składnię programu C#. Podobnie dyrektywy preprocesora służą tylko do omijania fragmentów kodu w pliku źródłowym i jeśli chodzi o składnię, także nie mają znaczenia.

Zawsze kiedy w wyniku przetwarzania leksykalnego możliwe jest uzyskanie kilku wyników, procesor leksykalny pobiera najdłuższy poprawny element leksykalny. Na przykład, jeśli kompilator napotyka następującą sekwencję znaków:

```
//
```

przetwarza je i interpretuje jako początek wiersza komentarza, a nie dwa wystąpienia tokena / (w takim przypadku nie oznaczałyby one początku komentarza). Podobnie dzieje się, kiedy napotkana zostaje sekwencja:

```
!=
```

Jest ona interpretowana jako operator porównania. Mając to na uwadze, łatwo zauważyć, jak proste błędy typograficzne w kodzie źródłowym mogą doprowadzić do nieprawidłowego działania programu. Zazwyczaj jednak powodują one błąd kompilacji.

Zakończenia wierszy

Zakończenie wiersza służy do dzielenia sekwencji znaków w pliku źródłowym C# na osobne wiersze.

Istnieje szereg różnych zakończeń wierszy:

- Znak powrotu karetki: *U+000D*
- Przesunięcie o wiersz: *U+000A*
- Powrót karetki, a po nim przesunięcie o wiersz: *U+000D U+000A*
- Przejście do następnego wiersza: *U+2085*
- Separator wiersza: *U+2028*
- Separator akapitu: *U+2029*

Aby zachować wysoki poziom kompatybilności z różnymi narzędziami do edycji kodu źródłowego, które umieszczają znaki zakończenia pliku, i zapewnić, że pliki źródłowe C# będą odczytywane jako sekwencja poprawnie zakończonych wierszy, są one poddawane dwóm przekształceniom:

- Jeśli ostatnim znakiem w źródłowym pliku C# jest *Ctrl+Z* (*U+001A*), to znak ten zostaje usunięty.
- Jeśli źródłowy plik C# nie jest pusty i jeśli jego ostatnim znakiem nie jest znak powrotu karetki (*U+000D*), wysunięcia wiersza (*U+000A*), przejścia do następnego wiersza (*U+2085*), separatora wiersza (*U+2028*) lub separatora akapitu (*U+2029*), to na końcu pliku umieszczony zostaje znak powrotu karetki (*U+000D*).

Komentarze

C# obsługuje dwa rodzaje komentarzy w plikach źródłowych:

- Komentarze wyodrębnione
- Komentarze jednowierszowe

W poniższych podpunktach omówimy te rodzaje komentarzy bardziej szczegółowo.

Komentarze wyodrębnione

Komentarz wyodrębniony zawsze rozpoczyna się znakami `/*`, a kończy znakami `*/`.

Komentarz taki może zajmować fragment wiersza:

```
/* Program testowy Hello World
*/ class Test
{
   static void Main() {
      System.Console.WriteLine("Hello, World!");
   }
}
```

Jeden wiersz:

```
/* Program testowy Hello World */
class Test
{
   static void Main() {
      System.Console.WriteLine("Hello, World!");
   }
}
```

Lub kilka wierszy:

```
/*
Program testowy Hello World
*/
class Test
{
   static void Main() {
      System.Console.WriteLine("Hello, World!");
   }
}
```

Komentarze wyodrębnione mogą być umieszczane w dowolnych punktach kodu, o ile tylko zajmują wydzielony wiersz. Na przykład komentarz przedstawiony poniżej jest wpisany poprawnie:

```
/*
Program testowy Hello World
*/
class Test
{
   static void Main() {
      System.Console.WriteLine("Hello, World!");
   }
}
```

W tym przypadku także:

```
class Test
{
/*
Program testowy Hello World
*/
   static void Main() {
      System.Console.WriteLine("Hello, World!");
   }
}
```

Podobnie jak w tym:

```
/*
Program testowy Hello World
*/
class Test
{
   static void Main() {
/*
Łańcuch znakowy wyświetlany na ekranie
*/
      System.Console.WriteLine("Hello, World!");
   }
}
```

To zaś przykład komentarza zapisanego niepoprawnie:

```
class Test
{
   static void Main() {
      System.Console. /*
Program testowy Hello World
*/
WriteLine("Hello, World!");
   }
}
```

Komentarze jednowierszowe

Komentarze jednowierszowe to jak sama nazwa wskazuje, komentarze zawarte w jednym wierszu. Zaczynają się od znaków // i biegną do końca wiersza:

```
class Test
{
   static void Main() {
      System.Console.WriteLine("Hello, World!"); // wyświetla "Hello, World!"
   }
}
```

Wewnątrz kodu można umieścić tyle komentarzy jednowierszowych, ile jest konieczne:

```
// wyświetla "Hello, World!"
class Test
{
   static void Main(){
      System.Console.WriteLine("Hello, World!"); // wyświetla "Hello, World!"
   }
}
```

Nie wolno jednak umieszczać komentarzy jednowierszowych wewnątrz wyrażeń. Oto przykład niepoprawnego wpisania komentarza:

```
// wyświetla "Hello, World!"
class Test
{
```

```
   static void Main(){
      System.Console.WriteLine // wyświetla "Hello, World!" ("Hello, World!");
   }
}
```

Ten kod także jest zapisany niewłaściwie:

```
class Test
{
   static void Main() {
      System.Console.WriteLine("Hello, World!");
   }
}
```

Zagnieżdżanie komentarzy

Nie można zagnieżdżać komentarzy. Trzeba tego unikać w imię czytelności kodu:

```
// /* Niewłaściwe zagnieżdżenie komentarza */

/* // Niewłaściwe zagnieżdżenie komentarza */
```

Zagnieżdżenie komentarzy nie powoduje błędu kompilacji; to jedynie niepoprawna forma, utrudniająca czytanie kodu.

Białe znaki

Białymi znakami są spacje, znaki tabulacji poziomej i pionowej, jak również znak przesunięcia strony.

Tokeny

Wyróżniamy pięć rodzajów tokenów:

- Identyfikatory
- Słowa kluczowe
- Literały
- Operatory
- Znaki interpunkcyjne

Białe znaki i komentarze nie są traktowane jak tokeny, a jedynie jako ich separatory.

Sekwencje specjalne Unicode

Sekwencje specjalne Unicode reprezentują znaki Unicode. Jedna taka sekwencja reprezentuje pojedynczy znak Unicode.

Sekwencje tworzone są przy użyciu znaków *\U* lub *\u*, po których następuje numer w formacie heksadecymalnym: *\Uxxxx* lub *\uxxxx*.

Sekwencja *\U0066* reprezentuje literę *f*. Jednakże *\U00660066* odpowiada *f0066*, a nie *ff*. Aby uzyskać dwuznak *ff*, należałoby użyć następującej sekwencji:

```
\U0066\U0066
```

Przedstawiony niżej kod stanowi przykład wykorzystania sekwencji specjalnych Unicode w praktyce:

```
class Test
{
   static void Main() {
      System.Console.WriteLine("\u0048\u0065\u006C\u006C\u006f, World!");
   }
}
```

Powyższy kod jest równoważny poniższemu:

```
class Test
{
   static void Main() {
      System.Console.WriteLine("Hello, World!");
   }
}
```

Sekwencje specjalne przetwarzane są, gdy kompilator odnajduje je w:

- Identyfikatorach
- Zwykłych literałach łańcuchowych
- Literałach znakowych

Sekwencje specjalne Unicode odnalezione w innych punktach kodu nie są przetwarzane.

Identyfikatory

Reguły rządzące użyciem identyfikatorów są dokładnie takie same jak zalecane w Unicode Standard Annex 15 (*http://www.unicode.org/reports/tr15/*), z następującymi wyjątkami:

- Dopuszczalne jest użycie znaku podkreślenia jako znaku początkowego, zgodnie z tradycją programowania w języku C.
- Dopuszcza się użycie sekwencji specjalnych Unicode jako identyfikatorów.
- Dopuszczalne jest użycie znaku `@` jako prefiksu umożliwiającego wykorzystanie słów kluczowych jako identyfikatorów. Bywa to szczególnie przydatne, gdy programuje się interfejsy C# do innych języków programowania. Gdy prefiks `@` zostaje użyty jako prefiks identyfikatora, wówczas identyfikator ten nazywany jest **identyfikatorem dosłownym**. I choć użycie prefiksu `@` z identyfikatorami niebędącymi słowami kluczowymi jest dopuszczalne, to jednak ze względów stylistycznych nie jest zalecane.

Oto przykłady składni identyfikatorów:

```
identyfikator:
   dostępny-identyfikator
   @   identyfikator-lub-słowo-kluczowe
```

```
dostępny-identyfikator:
   identyfikator-lub-słowo-kluczowe, które nie jest słowem kluczowym

identyfikator-lub-słowo-kluczowe:
znak-początkowy-identyfikatora   kolejne-znaki-identyfikatoraopcji

znak-początkowy-identyfikatora:
   znak-litera
   _ (znak podkreślenia U+005F)

kolejne-znaki-identyfikatora:
   kolejny-znak-identyfikatora
   kolejne-znaki-identyfikatora   kolejny-znak-identyfikatora

kolejny-znak-identyfikatora:
   znak-litera
   znak-cyfra-dziesiętna
   znak-łącznik
   znak-łączący
   znak-formatujący

znak-litera:
   znak Unicode klasy Lu, Ll, Lt, Lm, Lo lub Nl
sekwencja-specjalna-Unicode reprezentująca znak klasy Lu, Ll, Lt, Lm, Lo lub Nl

znak-łączący:
   znak Unicode klasy Mn lub Mc
   sekwencja-specjalna-Unicode reprezentująca znak klasy Mn lub Mc

znak-cyfra-dziesiętna:
   znak Unicode klasy Nd
   sekwencja-specjalna-Unicode reprezentująca znak klasy Nd

znak-łącznik:
   znak Unicode klasy Pc
   sekwencja-specjalna-Unicode reprezentująca znak klasy Pc

znak-formatujący:
   znak Unicode klasy Cf
   sekwencja-specjalna-Unicode reprezentująca znak klasy Cf
```

Przykładami poprawnych identyfikatorów mogą być:

- `identyfikator1`
- `_identyfikator`
- `@private`

Dwa identyfikatory uznawane są za identyczne, jeśli po wymienionych niżej przekształceniach (dokonanych w opisanej kolejności) stają się takie same:

- Usuwanie prefiksu `@` z identyfikatorów dosłownych.
- Przekształcenie każdej sekwencji specjalnej Unicode na znaki Unicode.
- Usuwanie wszelkich znaków formatujących..

Identyfikatory, w których wykorzystuje się dwa sąsiadujące znaki podkreślenia (_), są zastrzeżone do wykorzystania w przyszłości.

Słowa kluczowe

Słowo kluczowe podobne jest do identyfikatora, z tą różnicą, że jest zastrzeżone. Słowa kluczowe mogą być identyfikatorami tylko wówczas, kiedy poprzedza je prefiks `@`.

Oto lista słów kluczowych w C#:

```
abstract
as
base
bool
break
byte
case
catch
char
checked
class
const
continue
decimal
default
delegate
do
double
else
enum
event
explicit
extern
false
finally
fixed
float
for
foreach
goto
if
implicit
in
int
interface
internal
is
lock
long
namespace
new
null
object
operator
out
override
params
private
protected
public
readonly
ref
return
sbyte
sealed
short
sizeof
stackalloc
static
string
struct
switch
this
throw
true
try
typeof
uint
ulong
unchecked
unsafe
ushort
using
virtual
void
volatile
while
```

Literały

Zadanie literałów jest bardzo proste: reprezentują one wartości w kodzie źródłowym.

Istnieje szereg różnych literałów.

Literały logiczne

W tej grupie wyróżniamy dwa literały:

- `true`
- `false`

Typem literałów logicznych jest `bool`.

Literały całkowitoliczbowe

Literały całkowitoliczbowe stosuje się do zapisywania wartości następujących typów:

- `int`
- `uint`
- `long`
- `ulong`

Mogą one przyjmować jedną z dwóch form:

- Wartości dziesiętne
- Wartości szesnastkowe

Typ literału całkowitoliczbowego można określić w następujący sposób:

- Jeśli literał całkowitoliczbowy zapisany jest bez sufiksu, to jest to literał typu:

```
int
uint
long
ulong
```

- Jeśli literał całkowitoliczbowy zapisany jest z sufiksem `U` lub `u`, to jest to literał typu:

```
uint
ulong
```

- Jeśli literał całkowitoliczbowy zapisany jest z sufiksem `L` lub `l`, to jest to literał typu:

```
long
ulong
```

- Jeżeli literał całkowitoliczbowy zapisany jest z sufiksem `UL`, `uL`, `Ul`, `LU`, `lU` lub `Lu`, to jest to literał typu:

```
ulong
```

Jeżeli wartość dowolnego literału całkowitoliczbowego wykracza poza zakres wartości dla typu `ulong`, wówczas kompilator generuje błąd.

Literały rzeczywiste

Literały rzeczywiste służą do zapisywania wartości następujących typów:

- `float`
- `double`
- `decimal`

Te trzy typy zapisywane są z trzema różnymi sufiksami:

- `F` lub `f` dla typu `float`
- `D` lub `d` dla typu `double`
- `M` lub `m` dla typu `decimal`

Jeśli sufiks nie został użyty, wartość domyślnie przyjmuje typ `double`.

Literały znakowe

Literał znakowy reprezentuje pojedyncze znaki pomiędzy apostrofami, jak `'x'`.

Poniższa tabela wymienia specjalne sekwencje znaków w C#:

Sekwencja znaków	Nazwa znaku	Unicode
`\'`	Apostrof	0x0027
`\"`	Cudzysłów	0x0022
`\\`	Ukośnik lewy (backslash)	0x005C
`\a`	Alarm	0x0007
`\b`	Znak cofania	0x0008
`\f`	Wysuw strony	0x000C
`\n`	Nowy wiersz	0x000A
`\0`	Null	0x0000
`\r`	Powrót karetki	0x000D
`\t`	Tabulacja pozioma	0x0009
`\v`	Tabulacja pionowa	0x000B

Tak wygląda składnia literałów znakowych:

```
znak-literał:
   ' znak '

znak:
   pojedynczy-znak
```

```
prosta-sekwencja-specjalna
heksadecymalna-sekwencja-specjalna
sekwencja-specjalna-Unicode

pojedynczy-znak:
   Dowolny znak oprócz ' (U+0027), \ (U+005C) oraz znaku nowego wiersza

prosta-sekwencja-specjalna: jedna z
   \' \" \\ \0 \a \b \f \n \r \t \v

szesnastkowa-sekwencja-specjalna:
   \x cyfra-hex   cyfra-hex(opcj)   cyfra-hex(opcj)   cyfra-hex(opcj)
```

Znak następujący po lewym ukośniku (\) musi być jednym ze znaków wymienionych w powyższej tabeli, w przeciwnym razie podczas kompilacji pojawi się błąd.

Literały łańcuchowe

W języku C# zaimplementowano obsługę dwóch rodzajów literałów łańcuchowych:

- Zwykłych literałów łańcuchowych
- Dosłownych literałów łańcuchowych

Zwykły literał łańcuchowy jest łańcuchem składającym się z zera lub więcej znaków zapisanych pomiędzy cudzysłowami. Literały tego rodzaju mogą zawierać zarówno proste, jak i szesnastkowe sekwencje specjalne, jak również sekwencje Unicode.

```
string = "Hello, World!";
```

Dosłowny literał łańcuchowy składa się ze znaku `@` poprzedzającego cudzysłów, a następnie zero lub więcej znaków, i kończy się cudzysłowem.

```
string = @"Hello, World!";
```

Różnica pomiędzy zwykłym a dosłownym literałem łańcuchowym polega na tym, że znaki pomiędzy cudzysłowami zapisane w literale dosłownym interpretowane są dosłownie, a literał ten może rozciągać się na wiele wierszy kodu źródłowego.

Należy przy tym zwrócić uwagę, że w przypadku przetwarzania literałów dosłownych jedynym wyjątkiem jest dwuznak \", reprezentujący cudzysłów.

```
string = @"Hello
,
 World!";
```

Literały puste

Na temat literałów pustych nie da się powiedzieć zbyt wiele — po prostu są one puste.

Operatory i znaki przestankowe

W C# stosuje się wiele operatorów i znaków przestankowych.

- Operatory stosuje się w wyrażeniach w celu opisania operacji wykonywanych na jednym, dwóch lub większej liczbie operandów.
- Znaki przestankowe służą do grupowania i separowania.
  - {
  - }
  - [
  -]
  - (
  -)
  - .
  - ,
  - :
  - ;
  - +
  - -
  - *
  - /
  - %
  - &
  - |
  - ^
  - !
  - ~
  - =
  - >
  - <
  - ?
  - ??
  - ::
  - ++
  - --
  - &&
  - ||

- ->
- ==
- !=
- <=
- >=
- +=
- -=
- *=
- /=
- %=
- &=
- |=
- ^=
- <<
- <<=
- > > (przesunięcie w prawo, złożone z dwóch tokenów: > oraz >)
- > >= (przypisanie przesunięcia w prawo, składające się z dwóch tokenów, > oraz >=)

Dyrektywy preprocesora

Dyrektywy preprocesora zapewniają kodowi C# ogromną funkcjonalność. Dzięki nim można:

- Warunkowo pomijać fragmenty plików źródłowych.
- Raportować błędy.
- Raportować warunki ostrzegawcze.
- Wyznaczać sekcje kodu.

Słowo „preprocesor" pochodzi z języków C i C++ i ponieważ przetwarzanie wstępne w C# nie jest wykonywane, stosowane jest jedynie dla zachowania spójności z tymi językami.

W C# dostępne są następujące dyrektywy preprocesora:

- `#define` oraz `#undef`. Służą do definiowania i unieważniania warunkowych symboli kompilacyjnych.
- `#if`, `#elif`, `#else` oraz `#endif`. Służą do pomijania sekcji kodu.
- `#line`. Pozwala kontrolować numery wierszy, w których pojawiły się błędy bądź ostrzeżenia.

- `#error` oraz `#warning`. Służą do wyświetlania komunikatów o błędach i ostrzeżeń.
- `#region` oraz `#endregion`. Służą do oznaczania sekcji kodu.
- `#pragma`. Przekazuje kompilatorowi informacje kontekstowe.

Dyrektywy preprocesora nie są tokenami C# i nie stanowią elementu gramatyki syntaktycznej tego języka.

Każda dyrektywa preprocesora musi być wpisana w nowym wierszu kodu źródłowego. Oprócz tego każda z nich musi zaczynać się od znaku #, po którym następuje nazwa dyrektywy.

> *Warto pamiętać, że jakkolwiek przed znakiem #, a także pomiędzy nim i nazwą dyrektywy, może znajdować się biały znak, to jednak nie jest to zalecane, gdyż utrudnia czytanie kodu.*

Wiersz kodu zawierający dyrektywę `#define`, `#undef`, `#if`, `#elif`, `#else`, `#endif` lub `#line` może kończyć się jednowierszowym komentarzem. W wierszach takich nie dopuszcza się jednak umieszczania komentarzy wyodrębnionych.

Dyrektywy preprocesora mogą mieć ogromny wpływ na wyniki kompilacji źródłowego kodu C#.

Na przykład skompilowanie następującego kodu:

```
#define A
#undef B
#define C
#undef B

class D
{
#if A
   void E() {}

#else
   void F() {}

#endif

#if B
   void G() {}

#else
   void H() {}

#endif

#if C
   void I() {}

#else
   void J() {}
```

```
#endif

#if D
   void K() {}

#else
   void L() {}

#endif
}
```

jest równoważne zapisaniu kodu:

```
class D
{
   void E() {}
   void H() {}
   void I() {}
   void L() {}
}
```

Symbole kompilacji warunkowej

Funkcja kompilacji warunkowej wykorzystuje dyrektywy: `#if`, `#elif`, `#else` oraz `#endif`, a steruje się nimi przy użyciu wyrażeń preprocesora i symboli kompilacji warunkowej.

Symbol kompilacji warunkowej może przyjąć jeden z dwóch stanów:

- Zdefiniowany
- Niezdefiniowany

Początkowo symbol przyjmuje stan niezdefiniowany i zachowuje go do czasu jawnego zdefiniowania. Napotkana dyrektywa `#define` pozostaje w tym stanie do czasu napotkania dyrektywy `#undef` lub do chwili osiągnięcia końca pliku źródłowego.

Wyrażenia preprocesora

Wyrażenia preprocesora mogą być zawarte w dyrektywach `#if` oraz `#elif`. W wyrażeniach tych można stosować następujące operatory:

- `!`
- `==`
- `!=`
- `&&`
- `||`

Do grupowania operatorów można użyć nawiasów.

Wynikiem wyrażenia preprocesora zawsze jest wartość logiczna.

Dyrektywy deklaracji

Dyrektywy deklaracji służą do definiowania i unieważniania symboli kompilacji warunkowej.

Przetworzenie dyrektywy `#define` skutkuje zdefiniowaniem symbolu kompilacji warunkowej, obowiązującego od wiersza następującego bezpośrednio po dyrektywie.

Przetworzenie dyrektywy `#undef` powoduje unieważnienie symbolu kompilacji warunkowej, obowiązujące od wiersza następującego bezpośrednio po dyrektywie.

Dyrektywa `#define` może być też użyta do przedefiniowania symbolu, który został zdefiniowany wcześniej, bez konieczności uprzedniego stosowania dyrektywy `#undef` do tego symbolu.

Dyrektywy kompilacji warunkowej

Dyrektywy kompilacji warunkowej pozwalają na warunkowe uwzględnianie bądź pomijanie fragmentów plików źródłowych C#.

Gdy użyte są dyrektywy kompilacji warunkowej, wówczas przetwarzana jest tylko jedna sekcja.

Zasady przetwarzania są następujące:

- Dyrektywy `#if` oraz `#elif` są przetwarzane w kolejności, aż do uzyskania wartości `true`. Jeśli wynikiem wyrażenia jest `true`, wówczas dana sekcja kodu jest zaznaczana.
- Jeśli wszystkie dyrektywy zwracają wartość `false`, wówczas użyta zostaje dyrektywa `#else`, o ile jest obecna w kodzie.
- Jeżeli wszystkie dyrektywy zwracają wartość `false` i nie można odnaleźć dyrektywy `#else`, wówczas żadna sekcja kodu nie zostaje zaznaczona.

Pominięty kod nie jest uwzględniany podczas analizy leksykalnej.

Dyrektywy diagnostyczne

Zadaniem dyrektyw diagnostycznych jest generowanie komunikatów o błędach i ostrzeżeniach, które raportowane są w taki sam sposób jak inne błędy i ostrzeżenia w czasie kompilacji.

Zarówno:

```
#warning Sprawdź kod!
```

jak i:

```
#error Błąd kodu w tym miejscu
```

wygenerują błąd kompilacji i służą jako wskaźniki fragmentów wymagających wprowadzenia zmian.

Dyrektywy oznaczania fragmentów kodu

Dyrektywy te służą do oznaczania fragmentów kodu źródłowego. Fragmentom tym nie jest przypisywane żadne znaczenie semantyczne. Oznaczenia fragmentów wykorzystywane są wyłącznie przez programistów lub zautomatyzowane narzędzia.

Dyrektywy oznaczania fragmentów kodu zapisuje się w sposób następujący:

```
#region
…

#endregion
```

Zapis powyższy jest równoważny zapisowi:

```
#if true
…

#endif
```

Dyrektywy #line

Dyrektywy `#line` służą do zmiany numerów wierszy i nazw plików źródłowych zwracanych przez kompilator, na przykład w komunikatach o błędach i w ostrzeżeniach.

Jeśli w kodzie źródłowym nie ma żadnej dyrektywy `#line`, wówczas kompilator zwraca rzeczywiste numery wierszy i nazwy plików źródłowych.

Dyrektywy #pragma

Dyrektywa `#pragma` jest dyrektywą preprocesora, przy użyciu której dostarcza się kompilatorowi informacji kontekstowych.

Oto przykłady sytuacji, w których dyrektywy te bywają użyteczne:

- Włączanie i wyłączanie specyficznych ostrzeżeń.
- Wskazywanie informacji, które zostaną wykorzystane przez debuger.

Podsumowanie

W rozdziale tym omawialiśmy leksykalną strukturę C#, zwracając szczególną uwagę na programy C#, gramatykę, zakończenia wierszy, komentarze, tokeny, słowa kluczowe i dyrektywy. Mając na względzie reguły gramatyki leksykalnej C#, programista może oszczędzić sobie wiele pracy i — przez zmniejszenie liczby błędów — skrócić czas debugowania.

W rozdziale 5. omówione zostaną rozmaite koncepcje języka C#.