

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# 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 wyczerpieniowe
- Obsługa wyjątków
- Typy generyczne

Poznaj nowoczesne metody tworzenia aplikacji



Spis treści

| | |
|---|-----------|
| O autorach | 15 |
| Wprowadzenie | 17 |
| Rozdział 1. Czym jest C#? | 23 |
| Nazwa | 23 |
| Ogólny rzut oka na C# | 23 |
| Historia | 24 |
| C# i CLR | 24 |
| Wzmianka o .NET | 25 |
| Standardy | 25 |
| Inne implementacje | 25 |
| Przykładowy kod C# | 26 |
| Korzyści z nauki programowania w C# | 28 |
| Podsumowanie | 28 |
| Rozdział 2. Początki pracy z C# | 29 |
| Początki pracy z C# są tańsze, niż mogłoby się wydawać! | 29 |
| Tanie opcje | 29 |
| Jak pisać kod C#, stosując bezpłatne narzędzia | 32 |
| Tanie narzędzie, które ułatwia życie! | 35 |
| Alternatywne edytory tekstowe i narzędzia C# | 37 |
| Narzędzia komercyjne: Visual Studio i Visual C# | 37 |
| Podsumowanie | 38 |
| Rozdział 3. Ogólny rzut oka na C# | 39 |
| C# | 39 |
| Podstawy C# | 39 |
| Czytanie źródłowego kodu C# | 40 |
| Typy | 41 |
| Typy wartości | 41 |
| Typy referencyjne | 41 |
| Typy predefiniowane | 42 |
| Przeciążanie | 44 |
| Przekształcenia | 44 |
| Typy tablicowe | 45 |
| Zmienne i parametry | 45 |
| Wyrażenia | 47 |
| Instrukcje | 48 |

| | |
|--|-----------|
| Klasy | 49 |
| State | 50 |
| Pola | 50 |
| Metody | 50 |
| Właściwości | 51 |
| Zdarzenia | 51 |
| Operatory | 51 |
| Indeksatory | 51 |
| Konstruktory instancji | 51 |
| Destruktory | 52 |
| Konstruktory statyczne | 52 |
| Dziedziczenie | 52 |
| Klasy statyczne | 52 |
| Struktury | 53 |
| Interfejsy | 53 |
| Delegacje | 53 |
| Typy wyliczeniowe | 54 |
| Typy generyczne | 54 |
| Iteratory | 54 |
| Typy dopuszczające wartość pustą | 55 |
| Podsumowanie | 55 |
| Rozdział 4. Struktura języka C# | 57 |
| Programy C# | 57 |
| Gramatyka | 59 |
| Dwuznaczności gramatyczne | 59 |
| Analiza leksykalna | 61 |
| Podsumowanie | 77 |
| Rozdział 5. Konceptje języka C# | 79 |
| Uruchamianie aplikacji | 79 |
| Zamykanie aplikacji | 80 |
| Deklaracje w C# | 80 |
| Składowe | 83 |
| Składowe przestrzeni nazw | 83 |
| Składowe struktur | 83 |
| Składowe wyliczeniowe | 83 |
| Składowe klas | 83 |
| Składowe interfejsów | 84 |
| Składowe tablic | 85 |
| Składowe delegacji | 85 |
| Dostęp do składowych | 85 |
| Deklarowana dostępność | 85 |
| Sygnatury | 86 |
| Sygnatury indeksatorów | 86 |
| Sygnatury konstruktorów instancji | 86 |
| Sygnatury metod | 86 |
| Sygnatury operatorów | 87 |
| Sygnatury i przeciążanie | 87 |
| Zasięg | 88 |
| Nazwy przestrzeni nazw i typów | 89 |

| | |
|--|------------|
| Zarządzanie pamięcią w C# | 89 |
| Podsumowanie | 90 |
| Rozdział 6. Typy | 91 |
| Trzy typy typów | 91 |
| Różnice pomiędzy typami wartościowymi i referencyjnymi | 91 |
| System typów w C# | 92 |
| Typy wartościowe | 92 |
| System.ValueType | 94 |
| Konstruktory domyślne | 94 |
| Typy strukturalne | 94 |
| Typy proste | 95 |
| Typy całkowitoliczbowe | 96 |
| Określanie typów | 97 |
| Typy zmiennoprzecinkowe | 98 |
| Typy dziesiętne | 98 |
| Typ bool | 99 |
| Typy wyliczeniowe | 99 |
| Typy referencyjne | 99 |
| Typy klas | 100 |
| Typ object | 101 |
| Typ string | 101 |
| Typy tablicowe | 101 |
| Typy delegacji | 101 |
| Typ null | 101 |
| Pakowanie i odpakowywanie | 101 |
| Typy nullable | 102 |
| Podsumowanie | 103 |
| Rozdział 7. Zmienne | 105 |
| Czym są zmienne? | 105 |
| Nie wszystkie zmienne tworzy się w ten sam sposób | 105 |
| Kategorie zmiennych | 106 |
| Zmienne statyczne | 107 |
| Elementy tablicy | 107 |
| Zmienne instancji | 108 |
| Parametry wartości | 109 |
| Parametry referencyjne | 109 |
| Parametry wyjściowe | 110 |
| Zmienne lokalne | 111 |
| Wartości domyślne | 111 |
| Przypisania niewątpliwe | 111 |
| Zmienne zainicjalizowane | 112 |
| Zmienne niezainicjalizowane | 112 |
| Podsumowanie | 121 |
| Rozdział 8. Przekształcenia | 123 |
| Przekształcenia niejawne | 123 |
| Przekształcenia tożsamościowe | 124 |
| Niejawne przekształcenia liczbowe | 124 |
| Niejawne przekształcenia wyliczeniowe | 125 |
| Niejawne przekształcenia referencyjne | 125 |

| | |
|--|------------|
| Przekształcenia typu opakowywanie | 126 |
| Niejawne przekształcenia parametrów typów | 127 |
| Niejawne przekształcenia wyrażeń stałych | 127 |
| Niejawne przekształcenia definiowane przez użytkownika | 127 |
| Przekształcenia jawne | 128 |
| Jawne przekształcenia liczbowe | 128 |
| Jawne przekształcenia wyliczeniowe | 130 |
| Jawne przekształcenia referencyjne | 131 |
| Przekształcenia typu odpakowywanie | 132 |
| Jawne przekształcenia parametrów typu | 132 |
| Jawne przekształcenia definiowane przez użytkownika | 132 |
| Przekształcenia standardowe | 132 |
| Standardowe przekształcenia niejawne | 133 |
| Standardowe przekształcenia jawne | 133 |
| Przekształcenia definiowane przez użytkownika | 133 |
| Przekształcenia metod anonimowych | 134 |
| Przekształcenia grup metod | 134 |
| Przekształcenia typu null | 135 |
| Przekształcenia dopuszczające wartość pustą | 135 |
| Podsumowanie | 136 |
| Rozdział 9. Wyrażenia | 137 |
| Klasyfikacja wyrażeń | 137 |
| Wyniki wyrażeń | 138 |
| Wartości wyrażeń | 138 |
| Wyrażenia i operatory | 138 |
| Trzy rodzaje operatorów | 139 |
| Priorytet i łączność operatorów | 139 |
| Przeciążanie operatorów | 141 |
| Operatory rozszerzone | 144 |
| Wyszukiwanie składowej | 146 |
| Typy bazowe | 147 |
| Składowe funkcyjne | 147 |
| Wyrażenia podstawowe | 151 |
| Literały | 152 |
| Nazwy proste | 152 |
| Wyrażenia nawiasowe | 152 |
| Dostęp do składowej | 152 |
| Wyrażenia wywołania | 153 |
| Dostęp do elementu | 154 |
| Wyrażenie wartości domyślnej | 157 |
| Metody anonimowe | 157 |
| Wyrażenia jednoargumentowe | 158 |
| Wyrażenia rzutowania | 158 |
| Operatory arytmetyczne | 158 |
| Operatory przesunięcia | 159 |
| Operatory relacyjne | 159 |
| Operatory logiczne | 160 |
| Logiczne operatory warunkowe | 161 |
| Operator null coalescing | 161 |
| Operatory przypisania | 162 |

| | |
|--|------------|
| Wyrażenie | 163 |
| Wyrażenia stałe | 163 |
| Wyrażenia boolowskie | 165 |
| Podsumowanie | 166 |
| Rozdział 10. Instrukcje | 167 |
| Czym są instrukcje? | 167 |
| Instrukcje C# | 169 |
| Punkt końcowy i osiągalność | 170 |
| Punkt końcowy | 170 |
| Osiągalność | 170 |
| Bloki kodu | 172 |
| Listy instrukcji | 172 |
| Instrukcje puste | 173 |
| Instrukcje etykietowane | 173 |
| Instrukcje deklaracyjne | 174 |
| Deklaracje zmiennych lokalnych | 174 |
| Deklaracje statycznych lokalnych | 175 |
| Instrukcje wyrażeniowe | 176 |
| Instrukcje wyboru | 176 |
| Instrukcje iteracyjne | 182 |
| Instrukcje skoku | 184 |
| Instrukcja using | 187 |
| Instrukcja yield | 188 |
| Podsumowanie | 189 |
| Rozdział 11. Przestrzenie nazw | 191 |
| Czym są przestrzenie nazw? | 191 |
| Organizowanie klas | 191 |
| Kontrolowanie zasięgu | 192 |
| Jednostki kompilacji | 192 |
| Deklaracje przestrzeni nazw | 193 |
| Dyrektywy użycia synonimu zewnętrznego | 194 |
| Dyrektywy użycia | 195 |
| Składowe przestrzeni nazw | 196 |
| Deklaracje typów | 196 |
| Kwalifikowane składowe synonimów | 197 |
| Podsumowanie | 199 |
| Rozdział 12. Klasy | 201 |
| Czym są klasy? | 201 |
| Deklaracje klas | 201 |
| Modyfikatory klas | 202 |
| Wskazanie klasy bazowej | 203 |
| Klasy bazowe | 203 |
| Implementacje interfejsów | 203 |
| Ciało klasy | 204 |
| Deklaracje częściowe | 204 |
| Składowe klasy | 204 |
| Dziedziczenie | 206 |
| Modyfikator new | 206 |
| Modyfikatory dostępu | 206 |

| | |
|---|------------|
| Składowe statyczne i instancje | 206 |
| State | 207 |
| Pola | 208 |
| Pola statyczne i instancje | 209 |
| Pola readonly | 209 |
| Metody | 210 |
| Parametry metody | 211 |
| Metody statyczne i instancje | 212 |
| Metody wirtualne | 212 |
| Metody przesłaniające | 213 |
| Metody ostateczne | 213 |
| Metody abstrakcyjne | 213 |
| Ciało metody | 213 |
| Właściwości | 214 |
| Właściwości statyczne i instancje | 215 |
| Aksesory | 215 |
| Aksesory wirtualne, ostateczne, przesłaniające i abstrakcyjne | 216 |
| Zdarzenia | 216 |
| Zdarzenia jako pola | 217 |
| Zdarzenia statyczne i instancje | 217 |
| Aksesory wirtualne, ostateczne, przesłaniające i abstrakcyjne | 218 |
| Indeksatory | 218 |
| Operatory | 220 |
| Operatory jednoargumentowe | 221 |
| Operatory dwuargumentowe | 222 |
| Operatory przekształcenia | 222 |
| Konstruktory instancje | 222 |
| Konstruktory statyczne | 223 |
| Destruktry | 224 |
| Podsumowanie | 224 |
| Rozdział 13. Struktury | 225 |
| Czym są struktury? | 225 |
| Deklaracje struktur | 226 |
| Modyfikatory struktury | 227 |
| Interfejsy struktury | 227 |
| Ciało struktury | 227 |
| Składowe struktury | 228 |
| Różnice między klasą a strukturą | 228 |
| Semantyka wartości | 229 |
| Dziedziczenie | 229 |
| Przypisania | 230 |
| Wartości domyślne | 230 |
| Opakowywanie i rozpakowywanie | 230 |
| this | 231 |
| Inicjalizatory pól | 231 |
| Konstruktory | 231 |
| Destruktry | 231 |
| Konstruktory statyczne | 232 |
| Kiedy korzystać ze struktur | 232 |
| Podsumowanie | 232 |

| | |
|--|------------|
| Rozdział 14. Tablice | 233 |
| Czym jest tablica? | 233 |
| Typy tablicowe | 235 |
| Typ System.Array | 236 |
| Tworzenie tablic | 236 |
| Dostęp do elementów tablicy | 237 |
| Składowe tablic | 237 |
| Kowariancja tablic | 237 |
| Inicjalizatory tablic | 238 |
| Podsumowanie | 240 |
| Rozdział 15. Interfejsy | 241 |
| Czym jest interfejs? | 241 |
| Definiowanie interfejsu | 242 |
| Deklaracje interfejsów | 242 |
| Modyfikatory | 243 |
| Jawne interfejsy bazowe | 243 |
| Ciało interfejsu | 244 |
| Składowe interfejsu | 244 |
| Metody interfejsu | 244 |
| Właściwości interfejsu | 244 |
| Zdarzenia interfejsu | 245 |
| Podsumowanie | 245 |
| Rozdział 16. Typy wyliczeniowe | 247 |
| Deklaracje typów wyliczeniowych | 248 |
| Modyfikatory typów wyliczeniowych | 249 |
| Składowe typów wyliczeniowych | 250 |
| Przeostrożność przed referencjami cyklicznymi | 251 |
| System.Enum | 251 |
| Wartości i operacje typów wyliczeniowych | 252 |
| Podsumowanie | 252 |
| Rozdział 17. Delegacje | 253 |
| Delegacje w działaniu | 253 |
| Deklaracje delegacji | 254 |
| Modyfikatory | 254 |
| Deklarowanie delegacji | 255 |
| Lista wywoławcza | 255 |
| Tworzenie instancji delegacji | 256 |
| Podsumowanie | 257 |
| Rozdział 18. Wyjątki | 259 |
| Zgłaszanie wyjątków | 259 |
| System.Exception | 260 |
| Typowe klasy wyjątków | 260 |
| Obsługa wyjątków | 261 |
| Co się dzieje, gdy klauzula catch nie zostaje odnaleziona? | 261 |
| Podsumowanie | 262 |

| | |
|--|------------|
| Rozdział 19. Atrybuty | 263 |
| Wprowadzenie do atrybutów | 263 |
| Klasy atrybutów | 264 |
| Parametry pozycyjne i nazwane | 264 |
| Użycie atrybutów | 264 |
| Typy parametrów atrybutu | 265 |
| Specyfikacja atrybutów | 266 |
| Instancje atrybutów | 269 |
| Kompilacja atrybutów | 269 |
| Odczytywanie instancji atrybutu podczas wykonywania programu | 269 |
| Atrybuty zastrzeżone | 270 |
| Atrybut Conditional | 270 |
| Podsumowanie | 272 |
| Rozdział 20. Typy generyczne | 275 |
| Typy generyczne w C# i szablony w C++ | 275 |
| Zalety typów generycznych | 276 |
| Deklaracje klas generycznych | 276 |
| Parametry typu | 277 |
| Różnice parametrów typu | 278 |
| Typ instancji | 279 |
| Składowe klas generycznych | 279 |
| Pola statyczne w klasach generycznych | 280 |
| Konstruktory statyczne w klasach generycznych | 280 |
| Dostęp do składowych chronionych | 281 |
| Przeciążanie w klasach generycznych | 281 |
| Operatory w klasach generycznych | 281 |
| Deklaracje struktur generycznych | 282 |
| Deklaracje interfejsów generycznych | 282 |
| Jawne implementacje składowych interfejsu | 283 |
| Deklaracje delegacji generycznych | 283 |
| Typy konstruowane | 283 |
| Argumenty typu | 284 |
| Typy otwarte i zamknięte | 284 |
| Składowe typów konstruowanych | 284 |
| Stosowanie dyrektyw użycia synonimów | 285 |
| Metody generyczne | 285 |
| Gdzie nie używa się typów generycznych | 287 |
| Ograniczenia | 288 |
| Podsumowanie | 291 |
| Rozdział 21. Iteratory | 293 |
| Blok iteratora | 294 |
| Bloki iteratora i błędy czasu kompilacji | 295 |
| Interfejsy enumeratora | 295 |
| Interfejsy wyliczeniowe | 295 |
| Typ yield | 296 |
| This | 296 |
| Obiekty enumeratora | 296 |
| Metoda MoveNext | 297 |
| Przerwanie wykonywania kodu | 298 |

| | |
|--|------------|
| Właściwość Current | 298 |
| Metoda Dispose | 299 |
| Obiekty przeliczalne | 300 |
| Metoda GetEnumerator | 300 |
| Podsumowanie | 301 |
| Rozdział 22. Kod nienadzorowany | 303 |
| Czym jest kod nienadzorowany? | 303 |
| Zalety i wady kodu nienadzorowanego | 304 |
| Zalety kodu nienadzorowanego | 304 |
| Wady kodu nienadzorowanego | 304 |
| Konteksty kodu nienadzorowanego | 305 |
| Podstawowe informacje o wskaźnikach | 306 |
| Wskaźniki void | 307 |
| Operatory wskaźnikowe | 307 |
| Kod nienadzorowany w działaniu | 308 |
| Stosowanie modyfikatora fixed | 309 |
| Operator sizeof | 310 |
| Użycie słowa kluczowego stackalloc | 311 |
| Kompilowanie kodu nienadzorowanego | 312 |
| Podsumowanie | 312 |
| Dodatek A Gramatyka języka C# | 313 |
| Dodatek B Konwencje nazewnictwa | 349 |
| Dodatek C Biblioteka standardowa | 357 |
| Dodatek D Przenaszalność | 371 |
| Dodatek E Komentarze dokumentacyjne XML | 375 |
| Skorowidz | 379 |

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 (określone 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. Asemlacja i moduły pełnią rolę fizycznych pojemników na typy (lub jako miejsce dla kodu; nieco później powiemy o tym bardziej szczegółowo). Asemlacja 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 dwojakiemu 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.

Powyzsza regula 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 wejścioweopcj    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

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

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

Literały znakowe reprezentuje pojedyncze znaki pomiędzy apostrofami, jak `'x'`.

Poniższa tabela wymienia specjalne sekwencje znaków w C#:

| Sekwencja znaków | Nazwa znaku | Unicode |
|------------------|--------------------------|---------|
| <code>\'</code> | Apostrof | 0x0027 |
| <code>\"</code> | Cudzysłów | 0x0022 |
| <code>\\</code> | Ukośnik lewy (backslash) | 0x005C |
| <code>\a</code> | Alarm | 0x0007 |
| <code>\b</code> | Znak cofania | 0x0008 |
| <code>\f</code> | Wysuw strony | 0x000C |
| <code>\n</code> | Nowy wiersz | 0x000A |
| <code>\0</code> | Null | 0x0000 |
| <code>\r</code> | Powrót karetki | 0x000D |
| <code>\t</code> | Tabulacja pozioma | 0x0009 |
| <code>\v</code> | 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 zera 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 debugger.

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#.