helion.pl

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

# Visual C# .NET. Encyklopedia




Autor: Microsoft Corporation
Tłumaczenie: Tomasz Miszkiel
ISBN: 83-7197-819-7
Tytuł oryginału: Microsoft Visual C# .Net Language Reference
Format: B5, stron: 390

Książka zawiera oficjalną dokumentację języka Visual C# .NET. C# jest jednym z podstawowych języków dostępnych na platformie .NET, łączącym w sobie największe zalety Javy i C++. „Visual C#. NET. Encyklopedia" to doskonałe uzupełnienie dokumentacji dostępnej w formie elektronicznej, niezbędne dla każdego programisty korzystającego z C#. Poza szczegółowym opisem języka, a także kompilatora C#, zawiera ona wiele przykładów gotowych do wykorzystania w konkretnych aplikacjach. Jest to książka, do której będziesz często powracał.

Omówiono:

- Typy
- Modyfikatory
- Polecenia
- Przestrzenie nazw
- Operatory
- Przekształcenia
- Dyrektywy preprocesora
- Atrybuty
- Tworzenie dokumentacji w XML

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

# Spis treści


**Rozdzial 1. Wprowadzenie do języka C# 9**

Programowanie w C# ... 9
Ogólna struktura programu napisanego w języku C# ... 10

**Rozdzial 2. Slowa kluczowe języka C# 13**

Słowa kluczowe ... 13
Modyfikatory ... 14
Modyfikatory dostępu ... 15
abstract ... 25
const ... 27
event ... 29
extern ... 35
override ... 36
readonly ... 38
sealed ... 40
static ... 40
unsafe ... 43
virtual ... 44
volatile ... 46
Przekształcenia ... 47
explicit ... 47
implicit ... 49
operator ... 50
Słowa kluczowe dostępu ... 52
base ... 52
this ... 54
Słowa kluczowe instrukcji ... 56
Instrukcje wyboru ... 56
Instrukcje iteracyjne ... 62
Instrukcje skoku ... 74
Instrukcje obsługi wyjątków ... 80
Instrukcje checked i unchecked ... 87
Instrukcja fixed ... 93
Instrukcja lock ... 95

Słowa kluczowe literałów ... 97
null ... 97
true ... 97
false ... 98
Słowa kluczowe operatorów ... 99
as ... 100
is ... 101
new ... 103
sizeof ... 107
typeof ... 108
true ... 110
false ... 111
stackalloc ... 112
Słowa kluczowe parametrów ... 113
params ... 114
ref ... 115
out ... 116
Słowa kluczowe przestrzeni nazw ... 117
namespace ... 117
using ... 119
Słowa kluczowe typów ... 122
Typy bezpośrednie ... 123
Opakowywanie i rozpakowywanie ... 149
Typy referencyjne ... 152
Tabele typów ... 163

**Rozdział 3. Język C# 171**

Operatory ... 171
Operatory przeciążalne ... 172
Operator [ ] ... 173
Operator () ... 174
Operator ... 175
Operator + ... 176
Operator - ... 177
Operator * ... 178
Operator / ... 179
Operator % ... 180
Operator & ... 180
Operator | ... 181
Operator ^ ... 182
Operator ! ... 183
Operator ~ ... 184
Operator = ... 184
Operator < ... 185
Operator > ... 186
Operator ?: ... 187
Operator ++ ... 188
Operator -- ... 189
Operator && ... 190
Operator || ... 191

Operator << 193
Operator >> 194
Operator == 195
Operator != 196
Operator <= 197
Operator >= 198
Operator += 199
Operator -= 200
Operator *= 200
Operator /= 201
Operator %= 202
Operator &= 203
Operator |= 204
Operator ^= 205
Operator <<= 206
Operator >>= 207
Operator -> 208
Atrybuty 209
Wprowadzenie do atrybutów 209
AttributeUsage 216
Conditional 217
Obsolete 219
Przewodnik po atrybutach 220
Deklaracje 226
Składniki 229
Składniki przestrzeni nazw 229
Składniki struktur 230
Składniki klas 230
Inne składniki 231
Dostęp do składników 231
Dostępność deklarowana 231
Dziedziny dostępności 232
Dostęp chroniony do składników 235
Ograniczenia dostępności 236
Dyrektywy preprocesora 237
#if 238
#else 239
#elif 239
#endif 240
#define 240
#undef 241
#warning 242
#error 242
#line 243
#region 244
#endregion 244
Tablice 245
Tablice jednowymiarowe 245
Tablice wielowymiarowe 247
Tablice nieregularne 248

Przekazywanie tablic za pomocą ref i out ... 250
Przewodnik po tablicach ... 252
Konstruktory i destruktory ... 257
Konstruktory klas ... 257
Konstruktory struktur ... 264
Destruktory ... 265
Indeksatory ... 268
Deklaracja indeksatora ... 268
Porównanie indeksatorów z właściwościami ... 271
Indeksatory interfejsu ... 272
Przewodnik po indeksatorach ... 275
Właściwości ... 278
Deklaracja właściwości ... 279
Akcesory ... 281
Właściwości interfejsów ... 287
Przewodnik po właściwościach ... 289
Sygnatury i przeciążanie ... 295
Zasięg ... 296
Ogólne reguły rządzące zasięgami ... 296
Ukrywanie nazw ... 299
Ukrywanie przez zagnieżdżenie ... 299
Ukrywanie przez dziedziczenie ... 300
Przekazywanie parametrów ... 302
Metoda Main ... 307
Wartości zwracane ... 308
Argumenty z wiersza poleceń ... 309
Przewodnik po metodach warunkowych ... 310
Przewodnik po delegacjach ... 313
Przewodnik po zdarzeniach ... 319
Dokumentacja XML ... 326
Znaczniki w komentarzach ... 326
Przetwarzanie pliku XML ... 339

**Dodatek A Opcje kompilatora języka C# 343**

Budowanie programu z wiersza poleceń ... 343
Opcje kompilatora C# — wykaz z podziałem na kategorie ... 345
Opcje kompilatora C# — wykaz alfabetyczny ... 347
@ (określenie pliku odpowiedzi) ... 349
/addmodule (import metadanych) ... 350
/baseaddress (określenie adresu bazowego biblioteki DLL) ... 350
/bugreport (tworzenie informacji o wystąpieniu problemu) ... 351
/checked (kontrola przepełnienia arytmetycznego) ... 352
/codepage (określenie strony kodowej dla plików z kodem źródłowym) ... 353
/debug (generowanie informacji diagnostycznych) ... 353
/define (definicja preprocesora) ... 354
/doc (przetwarzanie komentarzy dokumentacji) ... 355
/filealign (określenie rozmiaru sekcji) ... 356
/fullpaths (określenie pełnej ścieżki dostępu do pliku) ... 357
/help, /? (wyświetlenie informacji dotyczących opcji kompilatora) ... 357
/incremental (umożliwienie kompilacji przyrostowej) ... 358

/lib (określenie lokalizacji odwołania do złożeń) ... 359
/linkresource (odnośnik do zasobu .NET Framework) ... 360
/main (określenie lokalizacji metody Main) ... 361
/noconfig (ignorowanie pliku csc.rsp) ... 362
/nologo (usunięcie informacji o kompilatorze) ... 363
/nostdlib (pominięcie biblioteki standardowej) ... 363
/nowarn (pominięcie określonych ostrzeżeń) ... 364
/optimize (włączenie lub wyłączenie optymalizacji) ... 364
/out (określenie nazwy pliku wyjściowego) ... 365
/recurse (przeszukanie podkatalogów w celu odnalezienia plików źródłowych) ... 366
/reference (import metadanych) ... 367
/resource (umieszczenie zasobu w pliku wyjściowym) ... 368
/target (określenie formatu pliku wyjściowego) ... 369
/unsafe (uaktywnienie trybu nienadzorowanego) ... 372
/utf8output (wyświetlenie komunikatów kompilatora w kodzie UTF-8) ... 373
/warn (określenie poziomu ostrzeżeń) ... 373
/warnaserror (traktowanie ostrzeżeń na równi z błędami) ... 374
/win32icon (importowanie pliku .ico) ... 375
/win32res (importowanie pliku zasobów Win32) ... 376

**Skorowidz** **377**

# Słowa kluczowe instrukcji

Instrukcje są wykonywane w trakcie działania programu. Poza nielicznymi wyjątkami opisanymi w niniejszej książce, instrukcje są wykonywane kolejno, zgodnie z ich zapisem w pliku programu.

| **Kategoria** | **Słowa kluczowe *C#*** |
|---|---|
| **Instrukcje wyboru** | `if`, `else`, `switch`, `case` |
| **Instrukcje iteracyjne** | `do`, `for`, `foreach`, `in`, `while` |
| **Instrukcje skoków** | `break`, `continue`, `default`, `goto`, `return` |
| **Instrukcje obsługi wyjątków** | `throw`, `try-catch`, `try-finally` |
| **Instrukcje** `checked` **i** `unchecked` | `checked`, `unchecked` |
| **Instrukcja** `fixed` | `fixed` |
| **Instrukcja** `lock` | `lock` |

## Instrukcje wyboru

Instrukcja wyboru sprawia, że program jest wykonywany w określonym miejscu, które zależy od wartości wyrażenia warunkowego (**true** lub **false**).

W instrukcjach wyboru korzystamy z następujących słów kluczowych:

- `if`,
- `else`,
- `switch`,

- case.

## if-else

Instrukcja **if** jest instrukcją kontroli przebiegu programu. Za pomocą tej instrukcji wykonywany jest określony fragment kodu, jeżeli wyrażenie warunkowe przyjmie wartość logiczną **true**.

```
if (wyrażenie)
     instrukcja1
[else
    instrukcja2]
```

gdzie:

*wyrażenie*

Wyrażenie, które może być niejawnie przekształcone w typ **bool** lub inny typ zawierający przeciążone operatory **true** i **false**.

*instrukcja1*

Instrukcja wewnętrzna (lub zestaw instrukcji), która zostanie wykonana, jeśli *wyrażenie* ma wartość **true**.

*instrukcja2*

Instrukcja wewnętrzna (lub zestaw instrukcji), która zostanie wykonana, jeśli *wyrażenie* ma wartość **false**.

### Uwagi

Jeśli *wyrażenie* przyjmie wartość **true**, to wykonywana jest *instrukcja1*.Jeśli istnieje opcjonalna sekcja **else**, a *wyrażenie* przyjmuje wartość **false**, to wykonywana jest *instrukcja2*. Po wykonaniu instrukcji warunkowej **if** następuje przejście do następnej instrukcji programu.

W instrukcji warunkowej **if** można umieścić bloki kodu, które są wykonywane w zależności od wartości wyrażenia warunkowego (aby instrukcja **if** została wykonana, wyrażenie warunkowe musi przyjąć wartość **true** lub **false**). W blokach można umieścić wiele instrukcji, których wykonanie zależy od wartości wyrażenia warunkowego.

Wewnątrz wyrażenia warunkowego można umieścić dowolne instrukcje. Wyrażeniem może być kolejna zagnieżdżona instrukcja warunkowa **if**. Należy przy tym pamiętać, że sekcja **else** odnosi się do ostatniej (najbardziej zagnieżdżonej) instrukcji **if**, na przykład:

```
if (x > 10)
   if (y > 20)
      Console.Write("Instrukcja_1");
   else
      Console.Write("Instrukcja_2");
```

Jeżeli warunek (y>20) nie będzie spełniony (przyjmie wartość **false**), to na ekranie zostanie wyświetlony tekst Instrukcja_2. Aby powiązać blok Instrukcja_2 z warunkiem (x>10), trzeba skorzystać z nawiasów klamrowych:

```
if (x > 10)
{
   if (y > 20)
      Console.Write("Instrukcja_1");
}
else
   Console.Write("Instrukcja_2");
```

W tym przypadku łańcuch Instrukcja_2 jest wyświetlany, jeśli warunek (x>10) przyjmuje wartość **false**.

### Przyklad

W niniejszym przykładzie znajduje się program, który na podstawie wprowadzonego przez użytkownika znaku z klawiatury sprawdza, czy znak ten jest literą alfabetu. Jeśli wprowadzono literę alfabetu, program sprawdza, czy jest to mała, czy wielka litera. W każdym przypadku na ekranie pojawia się odpowiedni komunikat.

```
// Instrukcje_if_else.cs
//przykład zastosowania if-else
using System;
public class IfTest
{
   public static void Main()
   {
      Console.Write("Proszę wprowadzić znak: ");
      char c = (char) Console.Read();
      if (Char.IsLetter(c))
         if (Char.IsLower(c))
            Console.WriteLine("Wprowadzono małą literę.");
         else
            Console.WriteLine("Wprowadzono wielką literę.");
      else
         Console.WriteLine("Wprowadzony znak nie jest literą alfabetu.");
   }
}
```

### Przykladowe dane:

```
2
```

### Przykladowy wynik:

```
Proszę wprowadzić znak: 2
Wprowadzony znak nie jest literą alfabetu.
```

Poniżej przedstawiono kilka innych wyników działania tego programu:

### Uruchomienie 2

```
Proszę wprowadzić znak: A
Wprowadzono wielką literę.
```

### Uruchomienie 3

```
Proszę wprowadzić znak: h
Wprowadzono małą literę.
```

Dzięki konstrukcji **else-if** można rozszerzyć instrukcję warunkową o większą liczbę warunków, na przykład:

```
if (Warunek_1)
   Instrukcja_1;
else if (Warunek_2)
   Instrukcja_2;
else if (Warunek_3)
   Instrukcja_3;
...
else
   Instrukcja_n;
```

### Przyklad

Poniższy program służy do sprawdzania, czy użytkownik wprowadził małą literę, wielką literę, czy cyfrę. Inne znaki nie należą do grupy znaków alfanumerycznych. W programie wykorzystano konstrukcję **else-if**.

```
// Instrukcje_if_else2.cs
// else-if
using System;
public class IfTest
{
   public static void Main()
   {
      Console.Write("Proszę wprowadzić znak: ");
      char c = (char) Console.Read();

      if (Char.IsUpper(c))
         Console.WriteLine("Wprowadzono wielką literę.");
      else if (Char.IsLower(c))
         Console.WriteLine("Wprowadzono małą literę.");
      else if (Char.IsDigit(c))
         Console.WriteLine("Wprowadzono cyfrę.");
      else
         Console.WriteLine("To nie jest znak alfanumeryczny");
   }
}
```

### Przykladowe dane

```
E
```

### Przykladowy wynik:

```
Proszę wprowadzić znak: E
Wprowadzono wielką literę.
```

### Uruchomienie 2

```
Proszę wprowadzić znak: e
Wprowadzono małą literę.
```

### Uruchomienie 3

```
Proszę wprowadzić znak: 4
Wprowadzono cyfrę.
```

### Uruchomienie 4

```
Proszę wprowadzić znak: $
To nie jest znak alfanumeryczny.
```

### Zobacz także:

- **switch**

## switch

Instrukcja **switch** należy do grupy instrukcji sterujących przebiegiem programu. Za pomocą tej instrukcji można wykonać odpowiedni blok kodu źródłowego w zależności od wartości wyrażenia sterującego. Oto składnia instrukcji **switch**:

```
switch (wyrażenie)
{
case wyrażenie-stałe
         instrukcja
         instrukcja-skoku
[default:
          instrukcja
          instrukcja-skoku]
}
```

gdzie:

*wyrażenie*

Wyrażenie całkowitoliczbowe.

*instrukcja*

Wewnętrzna instrukcja (lub blok instrukcji), która jest wykonywana, gdy wyrażenie przyjmie jedną z wartości wymienionych w instrukcji **switch** (jest to blok zawarty w sekcji **case**). Gdy wyrażenie nie przyjmie żadnej wartości wymienionej w instrukcji **switch**, wykonywany jest kod zawarty w sekcji **default**.

*instrukcja-skoku*

Instrukcja, która powoduje wyjście z danej sekcji **case**.

*wyrażenie-stałe*

W zależności od wartości tego wyrażenia wykonywany jest blok instrukcji znajdujący się w odpowiedniej sekcji **case**.

### Uwagi

Po wyznaczeniu wartości wyrażenia instrukcji **switch** następuje wykonywanie bloku kodu tej sekcji **case**, której *wyrażenie stałe* ma taką samą wartość jak *wyrażenie*. W instrukcji **switch** można umieścić dowolną liczbę sekcji **case**, ale nie można zastosować

dwóch sekcji **case**, których wartość wyrażenia stałego jest taka sama. Wykonywanie instrukcji **switch** rozpoczyna się od wybranego bloku, a kończy na instrukcji skoku, która powoduje wyjście z sekcji **case**.

Należy zauważyć, że odpowiednia instrukcja skoku jest wymagana na końcu bloku kodu w każdej sekcji **case** oraz w sekcji **default**. W przeciwieństwie do odpowiednika instrukcji **switch** w języku C++, C# nie zapewnia domyślnego przejścia z jednej etykiety **case** do następnej. Aby umożliwić przejście pomiędzy określonymi etykietami, należy skorzystać z instrukcji **goto** w odpowiednim bloku **case** lub **default**.

Jeśli *wyrażenie* ma wartość, która nie pasuje do wartości żadnego wyrażenia stałego etykiet **case**, sterowanie przechodzi do instrukcji zawartych w sekcji oznaczonej etykietą **default** (umieszczenie tej etykiety w instrukcji **switch** jest opcjonalne). Jeśli instrukcja **switch** nie zawiera etykiety **default**, następuje wyjście z instrukcji.

### Przyklad

```
// instrukcje_switch.cs
using System;
class SwitchTest
{
   public static void Main()
   {
      Console.WriteLine("Kawa: 1=Mała 2=Średnia 3=Duża");
      Console.Write("Proszę wybrać kawę: ");
      string s = Console.ReadLine();
      int n = int.Parse(s);
      int cost = 0;
      switch(n)
      {
         case 0:
         case 1:
            cost += 25;
            break;
         case 2:
            cost += 25;
            goto case 1;
         case 3:
            cost += 50;
            goto case 1;
         default:
            Console.WriteLine("Nieprawidłowy wybór. Można wybrać 1, 2 lub 3.");
            break;
       }
       if (cost != 0)
          Console.WriteLine("Proszę wrzucić {0} groszy.", cost);
       Console.WriteLine("Dziękujemy za skorzystanie z naszej usługi.");
   }
}
```

### Przykladowe dane:

```
2
```

### Przykladowy wynik

```
Kawa: 1=Mała 2=Średnia 3=Duża
Proszę wybrać kawę: 2
Proszę wrzucić 50 groszy.
Dziękujemy za skorzystanie z naszej usługi.
```

W powyższym przykładzie jako *wyrażenia* użyto zmiennej typu całkowitego o nazwie `n`. Można również skorzystać bezpośrednio ze zmiennej typu **string** o nazwie `s`, wówczas instrukcja **switch** przyjmie postać:

```
switch(s)
{
   case "1":
      ...
   case "2":
      ...
}
```

### Zobacz także:

- **if**

## Instrukcje iteracyjne

Za pomocą instrukcji iteracyjnych można tworzyć pętle. Instrukcje iteracyjne wykonują instrukcję wewnętrzną określoną ilość razy, aż do wystąpienia warunku kończącego działanie pętli. Instrukcje wykonywane są w porządku, w jakim wystąpiły w tekście programu (wyjątkiem jest instrukcja skoku).

Poniższe słowa kluczowe są wykorzystywane w instrukcjach iteracyjnych:

- `do`,
- `for`,
- `foreach`,
- `in`,
- `while`.

## do

Instrukcja **do** służy do wielokrotnego wykonywania instrukcji wewnętrznej (lub bloku instrukcji), aż do momentu, gdy określone wyrażenie przyjmie wartość **false**. Oto składnia instrukcji **do**:

```
do instrukcja while (wyrażenie);
```

gdzie:

*wyrażenie*

Jest to wyrażenie, które może być niejawnie przekształcone w wyrażenie boolowskie lub w typ zawierający przeciążone operatory **true** i **false**.

Wyrażenie to służy do sprawdzania, czy warunek kończący działanie pętli został osiągnięty.

*instrukcja*

Wewnętrzna instrukcja (lub blok instrukcji) do wykonania.

### Uwagi

W przeciwieństwie do instrukcji **while**, w instrukcji **do** wartość wyrażenia jest obliczana po wykonaniu wewnętrznego bloku instrukcji, co sprawia, że blok ten jest wykonywany przynajmniej raz.

### Przyklad

```
// instrukcje_do.cs
using System;
public class TestDoWhile
{
   public static void Main ()
   {
      int x;
      int y = 0;

      do
      {
         x = y++;
         Console.WriteLine(x);
      }

      while(y < 5);
   }
}
```

### Wynik

```
0
1
2
3
4
```

### Przyklad

W tym przykładzie pokazano sytuację, w której wyrażenie ma zawsze wartość **false**, lecz mimo to pętla zostanie wykonana tylko raz.

```
// statements_do2.cs
using System;
class DoTest {
   public static void Main()
   {
      int n = 10;
      do
      {
         Console.WriteLine("Wartość zmiennej n wynosi {0}", n);
         n++;
```

```
        } while (n < 6);
    }
}
```

### Wynik

```
Wartość zmiennej n wynosi 10
```

## for

Pętle **for** służą do wielokrotnego wykonywania wewnętrznej instrukcji (lub bloku instrukcji), aż do momentu, gdy określone wyrażenie przyjmie wartość **false**. Oto składnia pętli **for**:

```
for ([inicjatory]; [wyrażenie]; [iteratory]) instrukcja
```

gdzie:

*inicjatory*

Szereg wyrażeń lub instrukcji przypisania inicjujący liczniki pętli. Wyrażenia i instrukcje przypisania są oddzielone od siebie za pomocą przecinka.

*wyrażenie*

Jest to wyrażenie, które może być niejawnie przekształcone w wyrażenie boolowskie lub w typ zawierający operatory przeciążone **true** i **false**. Wyrażenie to służy do sprawdzania, czy warunek kończący działanie pętli został osiągnięty.

*iteratory*

Instrukcja (lub blok instrukcji) służąca do inkrementacji lub dekrementacji wartości liczników pętli.

*instrukcja*

Wewnętrzna instrukcja (lub blok instrukcji) danej pętli.

### Uwagi

Instrukcja **for** jest wykonywana w następujący sposób:

- Najpierw są wyznaczane wartości *inicjatorów*.
- Następnie sprawdzana jest wartość logiczna *wyrażenia*. Jeśli wyrażenie ma wartość **true**, następuje wykonanie wewnętrznego bloku instrukcji, po czym wyznaczana jest wartość iteratorów.
- Jeśli *wyrażenie* ma wartość logiczną **false**, następuje wyjście z pętli.

W instrukcji **for** najpierw sprawdzana jest wartość logiczna wyrażenia, co sprawia, że blok wewnętrzny pętli może w ogóle nie zostać wykonany.

Wszystkie wyrażenia sterujące pętlą **for** są opcjonalne, dzięki czemu można tworzyć pętle nieskończone, na przykład:

```
for (;;) {
...
```

```
}
```

### Przyklad

```
// instrukcje_for.cs
// pętla for
using System;
public class ForLoopTest
{
   public static void Main()
   {
      for (int i = 1; i <= 5; i++)
         Console.WriteLine(i);
   }
}
```

### Wynik

```
1
2
3
4
5
```

## foreach, in

Instrukcja **foreach** służy do wielokrotnego wykonywania grupy instrukcji wewnętrznych dla każdego elementu znajdującego się w tablicy lub w obiekcie kolekcji. W instrukcji **foreach** jest przeprowadzana iteracja po elementach kolekcji w celu uzyskania określonych informacji. Instrukcja ta nie powinna być wykorzystywana do zmiany zawartości tablicy lub obiektu kolekcji, ponieważ dokonywanie takich zmian może mieć nieprzewidywalne skutki. Oto składnia instrukcji **foreach**:

```
foreach (typ identyfikator in wyrażenie) instrukcja
```

gdzie:

*typ*

Typ identyfikatora.

*identyfikator*

Zmienna iteracyjna reprezentująca element kolekcji.

*wyrażenie*

Kolekcja obiektów lub wyrażenie typu tablicowego. Typ elementów kolekcji musi być przekształcalny do typu zmiennej iteracyjnej.

*instrukcja*

Wewnętrzna instrukcja (lub blok instrukcji) do wykonania.

### Uwagi

Instrukcja wewnętrzna jest wykonywana dla każdego elementu znajdującego się w tablicy lub kolekcji. Po wykonaniu iteracji po wszystkich elementach kolekcji, sterowanie przekazywane jest do następnej instrukcji umieszczonej poniżej bloku **foreach**.

Więcej informacji na temat słowa kluczowego **foreach** (łącznie z przykładami zastosowań) znajduje się w podpunktach:

- Wykorzystanie instrukcji **foreach** z tablicami.
- Wykorzystanie instrukcji **foreach** z kolekcjami.

### Wykorzystanie instrukcji foreach z tablicami

Przy korzystaniu z instrukcji foreach i tablicy, wewnętrzna instrukcja (lub blok instrukcji) jest wykonywana dla każdego elementu tablicy.

### Przyklad

W niniejszym przykładzie znajduje się program, który służy do wyznaczania liczby wszystkich liczb parzystych i nieparzystych zawartych w tablicy. Każdy typ (parzysty i nieparzysty) posiada swój własny licznik, przechowujący liczbę wystąpień danego typu.

```
// statements_foreach_arrays.cs
// Wykorzystanie foreach z tablicami
using System;
class MainClass
{
   public static void Main()
   {
      int odd = 0, even = 0;
      int[] arr = new int [] {0,1,2,5,7,8,11};

      foreach (int i in arr)
      {
         if (i%2 == 0)
            even++;
         else
            odd++;
      }

      Console.WriteLine("Znaleziono {0} liczb nieparzystych i {1} liczb parzystych.",
                        odd, even) ;
   }
}
```

### Wynik

```
Znaleziono 4 liczb parzystych i 3 liczb parzystych .
```

### Zobacz także:

- **foreach**, **in** (strona 65); Tablice (strona 245).

### Wykorzystanie instrukcji foreach z kolekcjami

Aby było możliwe przeprowadzenie iteracji po elementach kolekcji, kolekcja musi spełnić określone wymagania. Na przykład w następującej instrukcji:

```
foreach (TypElementu element in MojaKolekcja)
```

`MojaKolekcja` musi spełnić następujące wymagania:

- Typ kolekcji:
  - Musi być jednym z typów: **interface**, **class** lub **struct**.
  - Musi zawierać metodę o nazwie `GetEnumerator()`, zwracającą typ, na przykład `Enumerator` (wyjaśniono poniżej).
- Typ `Enumerator` (klasa lub struktura) musi zawierać:
  - Właściwość o nazwie `Current`, zwracającą `TypElementu` lub inny typ, który może być przekształcony w `TypElementu`. Właściwość zwraca aktualny element kolekcji.
  - Metodę boolowską `MoveNext`, która służy do przeprowadzania inkrementacji licznika elementów i zwraca wartość logiczną **true**, jeśli po inkrementacji występują jeszcze elementy w kolekcji.

Istnieją trzy sposoby tworzenia i wykorzystywania kolekcji:

1. Utworzenie kolekcji zgodnie z powyższymi instrukcjami. Taka kolekcja może być wykorzystana tylko w programach napisanych w języku C#.
2. Utworzenie kolekcji w postaci ogólnej zgodnie z powyższymi zaleceniami, z dodaniem implementacji interfejsu `IEnumerable`. Taka kolekcja może być wykorzystana w innych językach programowania, na przykład w Visual Basic®.
3. Skorzystanie z jednej z predefiniowanych kolekcji, które znajdują się w klasach kolekcji.

Poniższe przykłady przedstawiają wyżej wymienione trzy sposoby tworzenia i wykorzystywania kolekcji.

| Przyklad | Przedstawia: | Komentarz |
|---|---|---|
| Przykład 1 | kolekcję dla programów napisanych w języku C#. | W tym przykładzie utworzono kolekcję zgodnie z powyższymi zaleceniami. |
| Przykład 2a | kolekcję ogólnodostępną. | W tym przykładzie utworzono kolekcję zgodnie z powyższymi zaleceniami oraz zaimplementowano interfejsy `IEnumerable` i `IEnumerator`. |
| Przykład 2b | kolekcję ogólnodostępną zawierającą metodę `Dispose`. | Jest to kolekcja podobna do zdefiniowanej w przykładzie 2a (różnica polega na zastosowaniu w wyliczeniu zdefiniowanym przez użytkownika metody `Dispose`, odziedziczonej z interfejsu `IDisposable`). |
| Przykład 3 | wykorzystanie jednej z predefiniowanych | W tym przykładzie utworzono egzemplarz tablicy mieszającej (za pomocą klasy `Hashtable`), której |

| | | |
|---|---|---|
| | klas kolekcji. | składniki wykorzystano do manipulacji kolekcją. Klasa `Hashtable` reprezentuje słownikową strukturę danych opartą o relację klucz-wartość Struktura ta jest implementowana jako tablica mieszająca. |

Więcej informacji dotyczących interfejsów `IEnumerator` i `IEnumerable` oraz klasy `Hashtable` znajduje się w „System.Collections" w .NET Framework SDK.

### Przyklad 1

```
// instrukcje_foreach_kolekcje.cs
// Wykorzystanie foreach z kolekcją dla C#:
using System;

// Deklaracja kolekcji:
public class MyCollection
{
   int[] items;

   public MyCollection()
   {
      items = new int[5] {12, 44, 33, 2, 50};
   }

   public MyEnumerator GetEnumerator()
   {
      return new MyEnumerator(this);
   }

   // Deklaracja klasy wyliczenia:
   public class MyEnumerator
   {
      int nIndex;
      MyCollection collection;
      public MyEnumerator(MyCollection coll)
      {
         collection = coll;
         nIndex = -1;
      }

      public bool MoveNext()
      {
         nIndex++;
         return(nIndex < collection.items.GetLength(0));
      }

      public int Current
      {
         get
         {
            return(collection.items[nIndex]);
         }
      }
   }
}

public class MainClass
```

```
{
   public static void Main()
   {
      MyCollection col = new MyCollection();
      Console.WriteLine("Wartości w kolekcji:");

      // Wyświetlenie elementów kolekcji:
      foreach (int i in col)
      {
         Console.WriteLine(i);
      }
   }
}
```

### Wynik

```
Wartości w kolekcji:
12
44
33
2
50
```

### Przyklad 2a

W tym przykładzie powtórzono algorytm poprzedniego przykładu, ale zastosowano ogólnodostępną kolekcję, która może być wyliczana w programach napisanych w innych językach, na przykład w Visual Basic. Ogólnodostępny typ kolekcji musi zawierać implementację interfejsu `IEnumerable` znajdującego się w przestrzeni nazw `System.Collections`.

```
// instrukcje_foreach_ogólnodostępna_kolekcja.cs
// Wykorzystanie foreach z ogólnodostępną kolekcją
using System;
using System.Collections;

// Deklaracja kolekcji i implementacja interfejsu IEnumerable:
public class MyCollection: IEnumerable
{
   int[] items;
   public MyCollection()
   {
      items = new int[5] {12, 44, 33, 2, 50};
   }

   public MyEnumerator GetEnumerator()
   {
      return new MyEnumerator(this);
   }

   // Implementacja metody GetEnumerator():
   IEnumerator IEnumerable.GetEnumerator()
   {
      return GetEnumerator();
   }

   // Deklaracja wyliczenia i implementacja interfejsu IEnumerator:
   public class MyEnumerator: IEnumerator
```

```
    {
        int nIndex;
        MyCollection collection;
        public MyEnumerator(MyCollection coll)
        {
            collection = coll;
            nIndex = -1;
        }

        public void Reset()
        {
            nIndex = -1;
        }

        public bool MoveNext() {
            nIndex++;
            return(nIndex < collection.items.GetLength(0));
        }

        public int Current
        {
            get
            {
                return(collection.items[nIndex]);
            }
        }

        // Właściwość Current interfejsu IEnumerator:
        object IEnumerator.Current
        {
            get
            {
                return(Current);
            }
        }
    }
}

public class MainClass
{
    public static void Main(string [] args)
    {
        MyCollection col = new MyCollection();
        Console.WriteLine("Wartości w kolekcji:");

        // Wyświetlenie elementów kolekcji:
        foreach (int i in col)
        {
            Console.WriteLine(i);
        }
    }
}
```

### Wynik

```
Wartości w kolekcji:
12
44
```

```
33
2
50
```

### Przyklad 2b

Ten przykład jest podobny do poprzedniego. Różni się tylko tym, że wykorzystano tu metodę `Dispose`, której implementacja znajduje się w wyliczeniu zdefiniowanym przez użytkownika. Wyliczenie dziedziczy składniki interfejsu `IDisposable`.

Instrukcja **foreach** wspiera wyliczenia wykorzystywane jednokrotnie. Jeśli wyliczenie zawiera implementację interfejsu `IDisposable`, to — niezależnie od sposobu przerwania pętli w stosunku do wyliczenia — zostanie wywołana metoda `Dispose`.

```
// instrukcje_foreach_ogólnodostępne_kolekcje2.cs
// Wykorzystanie foreach z kolekcja ogólnodostępną
using System;
using System.Collections;

// Deklaracja kolekcji i implementacja interfejsu IEnumerable:
public class MyCollection: IEnumerable
{
   int[] items;
   public MyCollection()
   {
      items = new int[5] {12, 44, 33, 2, 50};
   }

   public MyEnumerator GetEnumerator()
   {
      return new MyEnumerator(this);
   }

   // Implementacja metody GetEnumerator():
   IEnumerator IEnumerable.GetEnumerator()
   {
      return GetEnumerator();
   }

   // Deklaracja wyliczenia i implementacja interfejsów IEnumerator
   // i IDisposable
   public class MyEnumerator: IEnumerator, IDisposable
   {
      int nIndex;
      MyCollection collection;
      public MyEnumerator(MyCollection coll)
      {
         collection = coll;
         nIndex = -1;
      }

      public void Reset()
      {
         nIndex = -1;
      }
      public bool MoveNext()
      {
```

```
            nIndex++;
            return (nIndex < collection.items.GetLength(0));
        }

        public int Current
        {
            get
            {
                return (collection.items[nIndex]);
            }
        }

        // Właściwość Current interfejsu IEnumerator:
        object IEnumerator.Current
        {
            get
            {
                return (Current);
            }
        }

        public void Dispose()
        {
            Console.WriteLine("Zerowanie kolekcji");
            collection = null;
        }
    }
}

public class MainClass
{
    public static void Main(string [] args)
    {
        MyCollection col = new MyCollection();
        Console.WriteLine("Wartości w kolekcji:");

        // Wyświetlenie elementów kolekcji:
        foreach (int i in col)
        {
            Console.WriteLine(i);
        }
    }
}
```

## Wynik

```
Wartości w kolekcji:
12
44
33
2
50
Zerowanie kolekcji
```

### Przyklad 3

W tym przykładzie użyto predefiniowanej klasy `Hashtable`. Dzięki wykorzystaniu w programie przestrzeni nazw `System.Collections` mamy dostęp do klasy `Hashtable` i jej składników. Aby dodać elementy do obiektu klasy `Hashtable`, należy skorzystać z metody `Add`.

```
// instrukcje_foreach_hashtable.cs
// Wykorzystanie klasy kolekcji Hashtable
using System;
using System.Collections;
public class MainClass
{
   public static void Main(string [] args)
   {
      // Deklaracja obiektu klasy Hashtable:
      Hashtable ziphash = new Hashtable();

      // Dodanie elementów za pomocą metody Add():
      ziphash.Add("98008", "Bellevue");
      ziphash.Add("98052", "Redmond");
      ziphash.Add("98201", "Everett");
      ziphash.Add("98101", "Seattle");
      ziphash.Add("98371", "Puyallup");

      // Wyświetlenie zawartości tablicy mieszającej:
      Console.WriteLine("Kod pocztowy   Miasto");
      foreach (string zip in ziphash.Keys)
      {
         Console.WriteLine(zip + "          " + ziphash[zip]);
      }
   }
}
```

### Wynik

```
Kod pocztowy   Miasto
98201          Everett
98052          Redmond
98101          Seattle
98008          Bellevue
98371          Puyallup
```

### Zobacz także:

- **foreach**, **in** (strona 65).

## while

Instrukcja **while** służy do wielokrotnego wykonywania określonego bloku instrukcji, aż do momentu, w którym wyrażenie kontrolne przyjmie wartość **false**. Oto składnia instrukcji **while**:

```
while (wyrażenie) instrukcja
```

gdzie:

*wyrażenie*

Jest to wyrażenie kontrolne, które może być niejawnie przekształcone w typ **bool** albo w inny typ zawierający przeciążone operatory **true** i **false**. Wyrażenie kontrolne służy do sprawdzania, czy warunek kończący działanie pętli został osiągnięty.

*instrukcja*

Wewnętrzna instrukcja (lub blok instrukcji) do wykonania.

### Uwagi

Wartość wyrażenia kontrolnego jest wyznaczana przed wykonaniem wewnętrznego bloku instrukcji, co sprawia, że pętla może w ogóle nie zostać wykonana.

Wykonywanie pętli może być przerwane za pomocą instrukcji **break**, **goto**, **return** lub **throw**, które powodują wyjście z pętli. Aby przekazać sterowanie do następnej iteracji, należy skorzystać z instrukcji **continue**.

### Przyklad

```
// instrukcje_while.cs
using System;
class WhileTest
{
   public static void Main()
   {
      int n = 1;

      while (n < 6)
      {
         Console.WriteLine("Aktualna wartość n wynosi {0}", n);
         n++;
      }
   }
}
```

### Wynik

```
Aktualna wartość n wynosi 1
Aktualna wartość n wynosi 2
Aktualna wartość n wynosi 3
Aktualna wartość n wynosi 4
Aktualna wartość n wynosi 5
```

## Instrukcje skoku

Instrukcje skoku służą do rozgałęziania programu. Za pomocą tych instrukcji można przekazać sterowanie w inne miejsce programu. Poniżej znajduje się wykaz słów kluczowych wykorzystywanych w instrukcjach skoku:

- `break`,
- `continue`,

- default,
- goto,
- return.

## break

Instrukcja **break** powoduje przerwanie wykonywania „najciaśniejszej" pętli lub instrukcji warunkowej, w której się znajduje. Sterowanie jest przekazywane do najbliższej instrukcji znajdującej się poniżej instrukcji przerwanej. Oto składnia instrukcji **break**:

```
break;
```

### Przyklad

W tym przykładzie instrukcja sterująca pętli **for** zawiera licznik, którego wartość powinna się zmieniać od 1 do 100. Zastosowano jednak instrukcję **break**, która przerywa wykonywanie pętli, gdy licznik osiągnie wartość 5.

```
// statements_break.cs
using System;
class BreakTest
{
   public static void Main()
   {
      for (int i = 1; i <= 100; i++)
      {
         if (i == 5)
            break;
         Console.WriteLine(i);
      }
   }
}
```

### Wynik

```
1
2
3
4
```

### Przyklad

W tym przykładzie zastosowano instrukcję **break** wewnątrz instrukcji **switch**.

```
// instrukcje_break2.cs
// break i switch
using System;
class Switch
{
   public static void Main()
   {
      Console.Write("Wprowadź 1, 2, lub 3: ");
      string s = Console.ReadLine();
      int n = Int32.Parse(s);
```

```
            switch(n)
            {
               case 1:
                  Console.WriteLine("Aktualna wartość wynosi {0}", 1);
                  break;
               case 2:
                  Console.WriteLine("Aktualna wartość wynosi {0}", 2);
                  break;
               case 3:
                  Console.WriteLine("Aktualna wartość wynosi {0}", 3);
                  break;
               default:
                  Console.WriteLine("Wprowadzono nieprawidłową wartość.");
                  break;
            }
         }
      }
```

### Wprowadzając:

```
1
```

### Przykladowy wynik

```
Wprowadź 1, 2, lub 3: 1
Aktualna wartość wynosi 1
```

Jeśli wprowadzimy wartość 4, wówczas otrzymamy:

```
Wprowadź 1, 2, lub 3: 4
Wprowadzono nieprawidłową wartość.
```

## continue

Instrukcja **continue** jest wykorzystywana podczas przekazywania sterowania do następnej iteracji występującej w danej instrukcji iteracyjnej (pętli). Oto składnia instrukcji **continue**:

```
continue;
```

### Przyklad

W poniższym przykładzie znajduje się pętla **for** z licznikiem, którego wartości zmieniają się od 1 do 10. Zastosowanie instrukcji **continue** umieszczonej w warunku `(i<9)` spowodowało, że blok programu umieszczonego pomiędzy instrukcją **continue** a końcem pętli **for** nie został wykonany.

```
// instrukcje_continue.cs
using System;
class ContinueTest
{
   public static void Main()
   {
      for (int i = 1; i <= 10; i++)
      {
```

```
            if (i < 9)
               continue;
            Console.WriteLine(i);
         }
      }
   }
```

### Wynik

```
9
10
```

## goto

Za pomocą instrukcji **goto** można przekazać sterowanie bezpośrednio do instrukcji oznaczonej etykietą, czyli w określone miejsce programu. Oto składnia instrukcji **goto**:

```
goto identyfikator;
goto case wyrażenie-stałe;
goto default;
```

gdzie:

*identyfikator*

Identyfikatorem jest etykieta.

*wyrażenie-stałe*

Jest to etykieta instrukcji **case** wewnątrz instrukcji **switch**.

### Uwagi

W tabeli przedstawiono trzy postaci omawianej instrukcji. W pierwszej z nich *identyfikator* oznacza etykietę znajdującą się w bieżącym bloku programu, w tym samym zasięgu leksykalnym lub w zasięgu zawierającym instrukcję **goto**.

Instrukcja **goto** jest zwykle wykorzystywana do przekazywania sterowania do określonej etykiety **case** instrukcji **switch** albo do etykiety **default** instrukcji **switch**.

Za pomocą instrukcji **goto** można opuścić głęboko zagnieżdżone pętle.

Jeśli etykieta, do której odwołuje się instrukcja **goto**, nie była wcześniej wykorzystywana w programie, wówczas w trakcie kompilacji może wystąpić ostrzeżenie.

### Przyklad

Przykład wykorzystania instrukcji **goto** w celu przekazania sterowania do określonej etykiety **case** instrukcji **switch** znajduje się w punkcie „switch” na stronie 60.

### Przyklad

Poniższy przykład przedstawia wykorzystanie instrukcji **goto** w celu opuszczenia zagnieżdżonych pętli.

```
// instrukcje_goto.cs
```

```
// Zagnieżdżone pętle służące do przeszukiwania zawartości tablicy
using System;
public class GotoTest1
{
   public static void Main()
   {
      int x = 200, y = 4;
      int[,] myArray = new int[x,y];

      // inicjalizacja tablicy:
      for (int i = 0; i < x; i++)
         for (int j = 0; j < y; j++)
            myArray[i,j] = ++i;

      // Odczyt wprowadzonej przez użytkownika wartości:
      Console.Write("Wprowadź liczbę którą należy znaleźć: ");

      // wprowadzenie znaku
      string s = Console.ReadLine();
      int myNumber = Int32.Parse(s);

      // Przeszukiwanie tablicy dwuwymiarowej:
      for (int i = 0; i < x; i++)
         for (int j = 0; j < y; j++)
            if (myArray[i,j] == myNumber)
               goto Found;

      Console.WriteLine("Liczby {0} nie znaleziono.", myNumber);
      goto Finish;
   Found:
      Console.WriteLine("Liczba {0} została znaleziona.", myNumber);

   Finish:
      Console.WriteLine("Przeszukiwanie zakończone.");
   }
}
```

### Źródlo

```
44
```

### Przykladowy wynik

```
Wprowadź liczbę którą należy znaleźć: 44
Liczba 44 została znaleziona.
Przeszukiwanie zakończone.
```

### Przyklad

```
// instrukcje_goto2.cs
// Wystąpi błąd o numerze CS0159, bo
// etykiety znajdują się poza zasięgiem.
using System;
class UnreachableCode
{
   public static void Main()
   {
      int x = 55;
```

```
        Console.WriteLine("x = {0}", x);

        if (x == 55)
        {
           x = 135;
           goto A;   // Błąd
        }

        x = x + 1;

        for (int i=1; i<=5; i++)
        {
           A: Console.WriteLine(i);
        }

        Console.WriteLine("x = {0}", x);
     }
  }
```

W tym przykładzie instrukcja **goto** odwołuje się do etykiety `A`, która jest poza zasięgiem instrukcji **goto**. W trakcie kompilacji wystąpi błąd:

```
No such label 'A' within the scope of the goto statement
```

co oznacza, że nie istnieje etykieta `A` w zasięgu instrukcji **goto**.

Do etykiety `A` nie odwołano się wcześniej w programie, wobec czego w trakcie kompilacji wystąpi ostrzeżenie.

Jeśli etykietę `A` umieścimy na początku instrukcji rozpoczynającej pętlę **for**, będziemy mogli skompilować i uruchomić program.

```
A: for (int i=1; i<=5; i++) {// Teraz będzie można skompilować program.
```

## return

Instrukcja **return** powoduje przerwanie wykonywania metody, w której się znajduje i powrót sterowania do miejsca, w którym metoda została wywołana. Za pomocą tej instrukcji można również zwrócić wartość określonego *wyrażenia*. Jeśli dana metoda jest typu **void**, wówczas nie trzeba korzystać z instrukcji **return**. Oto składnia instrukcji **return**:

```
return [wyrażenie];
```

gdzie:

*wyrażenie*

Wartość zwrócona przez daną metodę. *Wyrażenie* nie jest wykorzystywane w metodach typu **void**.

### Przyklad

W poniższym przykładzie metoda `CalculateArea(int r)` zwraca wartość pola powierzchni koła za pomocą zmiennej `area` typu **double**.

```
// instrukcje_return.cs
using System;
```

```
class ReturnTest
{
   static double CalculateArea(int r)
   {
      double area;
      area = r*r*Math.PI;
      return area;
   }

   public static void Main()
   {
      int radius = 5;
      Console.WriteLine("Pole powierzchni wynosi {0:0.00}", CalculateArea(radius));
   }
}
```

### Wynik

```
Pole powierzchni wynosi 78.54
```

## Instrukcje obsługi wyjątków

Język C# posiada wbudowany system obsługi sytuacji wyjątkowych, zwanych krótko wyjątkami (ang. *exceptions*), które mogą wystąpić w trakcie wykonywania programu. Wyjątki są obsługiwane w specjalnych, wydzielonych blokach kodu źródłowego, do których sterowanie ma dostęp tylko wtedy, gdy wystąpi sytuacja wyjątkowa. Do obsługi wyjątków wykorzystuje się słowa kluczowe: **try**, **catch**, **throw** i **finally**.

Podrozdział składa się z następujących punktów:

- `throw`,
- `try-catch`,
- `try-finally`,
- `try-catch-finally`.

### throw

Instrukcja **throw** służy do zgłaszania sytuacji wyjątkowych, które wystąpiły podczas wykonywania programu. Oto składnia instrukcji **throw**:

**throw** [*wyrażenie*];

gdzie:

*wyrażenie*

Jest to obiekt wyjątku. Wyrażenie można opuścić, jeśli ponownie zgłaszany jest wyjątek bieżącego obiektu w bloku **catch**.

### Uwagi

Zgłoszony wyjątek jest obiektem klasy pochodnej względem **System.Exception**, na przykład:

```
class MyException : System.Exception {}
throw new MyException();
```

Instrukcja **throw** najczęściej wykorzystywana jest z instrukcjami obsługi wyjątków **try-catch** lub **try-finally**. Gdy dany wyjątek jest zgłaszany, następuje poszukiwanie odpowiedniego bloku **catch**, który może obsłużyć ten wyjątek.

### Przyklad

W poniższym przykładzie pokazano sposób wykorzystania instrukcji **throw** w celu zgłoszenia wyjątku.

```
// przykład użycia instrukcji throw
using System;
public class ThrowTest
{
   public static void Main()
   {
      string s = null;

      if (s == null)
      {
         throw(new ArgumentNullException());
      }

      Console.Write("W zmiennej s typu String znajduje się wartość null");
      ➥// instrukcja nie zostanie wykonana
   }
}
```

### Wynik

Wynikiem działania programu będzie wystąpienie wyjątku:

```
System.ArgumentNullException
```

### Przyklad

Inne przykłady wykorzystania instrukcji **throw** znajdują się na stronie 80 („try-catch”, „try-finally” i „try-catch-finally”).

## try-catch

Instrukcja **try-catch** składa się z bloku **try** oraz jednego lub kilku bloków **catch**, które określają odpowiednie sekcje obsługi wyjątków. Oto składnia instrukcji **try-catch**:

```
try blok-try
catch (deklaracja-wyjątku-1) blok-catch-1
catch (deklaracja-wyjątku-2) blok-catch-2
```

```
...
try blok-try catch blok-catch
```

gdzie:

*blok-try*

Zawiera segment kodu źródłowego, w którym może wystąpić sytuacja wyjątkowa.

*deklaracja-wyjątku, dekaracja-wyjątku-1, deklaracja-wyjątku-2*

Deklaracja obiektów wyjątku.

*catch-blok, catch-blok-1, catch-blok-2*

Bloki zawierają kod obsługi wyjątków.

## Uwagi

W bloku *blok-try* znajduje się kod źródłowy, którego wykonanie może spowodować wystąpienie wyjątku. Dany kod jest wykonywany aż do momentu zgłoszenia wyjątku lub do chwili pomyślnego wykonania całego bloku *blok-try*. Na przykład w poniższy fragment jest próbą konwersji jawnej obiektu **null** w wartość zmiennej typu **int**, która zakończy się wystąpieniem wyjątku **NullReferenceException**:

```
object o2 = null;
try
{
   int i2 = (int) o2;   // Błąd
}
```

Klauzula **catch** może być wykorzystana bez argumentów. Przechwytuje ona wówczas wyjątek każdego typu. Taką klauzulę nazywamy ogólną klauzulą **catch**. Klauzuli **catch** można także przekazać argument odziedziczony po klasie **System.Exception**, wówczas klauzula zawiera kod obsługi określonego wyjątku, na przykład:

```
catch (InvalidCastException e)
{
}
```

W jednej instrukcji **try-catch** można wykorzystać kilka klauzuli **catch**, przeznaczonych do przechwytywania określonych wyjątków. Jeśli instrukcja zawiera kilka klauzuli **catch**, należy umieścić je w odpowiednim porządku, bo w trakcie wystąpienia wyjątku klauzule **catch** są sprawdzane po kolei. Bardziej specyficzne wyjątki powinny być wychwytywane wcześniej od wyjątków ogólnych.

W bloku **catch** można wykorzystać instrukcję **throw**. Wykorzystanie tej instrukcji umożliwia ponowne zgłoszenie wyjątku, który wcześniej został przechwycony przez daną instrukcję **catch**, na przykład:

```
catch (InvalidCastException e)
{
   throw (e);   // Ponowne zgłoszenie wyjątku e
}
```

Jeśli chcemy zgłosić wyjątek, który właśnie jest obsługiwany w bezparametrowym bloku **catch**, możemy użyć instrukcji **throw** nie zawierającej argumentów, na przykład:

```
catch
{
   throw;
}
```

Nie należy inicjować zmiennych wewnątrz bloku **try**, ponieważ nie ma pewności, że blok ten zostanie w całości wykonany w trakcie działania programu. Wyjątek może wystąpić przed wykonaniem wszystkich instrukcji znajdujących się w danym bloku. Przykładem jest poniższy fragment kodu źródłowego, w którym następuje inicjalizacja zmiennej `x` wewnątrz bloku **try**. Próba wykorzystania tej zmiennej na zewnątrz bloku **try** w instrukcji `Write(x)` spowoduje wystąpienie błędu podczas kompilacji:

```
Use of unassigned local variable,
```

co oznacza, że nastąpiła próba wykorzystania zmiennej lokalnej, której nie przypisano żadnej wartości.

```
public static void Main()
{
   int x;
   try
   {
      x = 123;   //   Tak nie należy robić.
      // ...
   }
   catch
   {
      // ...
   }
   Console.Write(x);   // Błąd: Use of unassigned local variable 'x'.
}
```

Więcej informacji na temat bloku **catch** znajduje się w punkcie „try-catch-finally” na stronie 86.

## Przyklad

W niniejszym przykładzie w bloku **try** wywoływana jest metoda `MyFn()`, która może spowodować wystąpienie wyjątku. Klauzula **catch** zawiera kod obsługi wyjątku, którego zadaniem jest wyświetlenie komunikatu na ekranie. Gdy wewnątrz metody `MyFn()` wywoływana jest instrukcja **throw**, następuje poszukiwanie sekcji **catch**, w której znajduje się odpowiedni kod obsługi wyjątku. Na ekranie pojawi się komunikat `Wyjątek przechwycony`.

```
// Ponowne zgłaszanie wyjątków:
using System;
class MyClass
{
   public static void Main()
   {
      MyClass x = new MyClass();
      try
      {
```

```
            string s = null;
            x.MyFn(s);
         }

         catch (Exception e)
         {
            Console.WriteLine("{0} Wyjątek przechwycono.", e);
         }
      }

      public void MyFn(string s)
      {
         if (s == null)
            throw(new ArgumentNullException());
      }
}
```

### Wynik

Podczas wykonywania programu wystąpi wyjątek:

```
System.ArgumentNullException
```

### Przyklad

W poniższym przykładzie wykorzystano dwie instrukcje **catch**. W pierwszej kolejności umieszczono blok obsługi najbardziej specyficznego wyjątku.

```
// Uporządkowane klauzule catch
using System;
class MyClass
{
   public static void Main()
   {
      MyClass x = new MyClass();
      try
      {
         string s = null;
         x.MyFn(s);
      }

      // Najbardziej szczególny wyjątek:
      catch (ArgumentNullException e)
      {
         Console.WriteLine("{0} Przechwycono pierwszy wyjątek.", e);
      }

      // Najbardziej ogólny wyjątek:
      catch (Exception e)
      {
         Console.WriteLine("{0} Przechwycono drugi wyjątek.", e);
      }

   }

   public void MyFn(string s)
   {
if (s == null)
```

```
            throw new ArgumentNullException();
        }
    }
```

### Wynik

Podczas wykonywania programu wystąpi wyjątek:

```
System.ArgumentNullException
```

Gdybyśmy umieścili klauzule **catch** w odwrotnej kolejności, w trakcie wykonywania programu wystąpiłby błąd:

```
A previous catch clause already catches all exceptions of this or a super type
➥('System.Exception')
```

który oznacza, że poprzednia klauzula **catch** (w naszym przypadku pierwsza) przechwytuje wszystkie wyjątki typu `System.Exception` (lub typu pochodnego).

Aby wychwycić najbardziej ogólny wyjątek, zamiast instrukcji **throw** należy umieścić:

```
throw new Exception();
```

### Zobacz także:

- **throw** (strona 80), **try-finally** (strona 85).

## try-finally

Blok **finally** jest użyteczny, gdy istnieje potrzeba zwolnienia zasobów przydzielonych w bloku **try**. Jest to spowodowane tym, że sterowanie zawsze jest przekazywane do bloku **finally**, niezależnie od wykonania bloku **try**. Instrukcja **try-finally** ma postać:

```
try blok-try finally blok-finally
```

gdzie:

*blok-try*

Zawiera fragment kodu, w którym może wystąpić wyjątek.

*blok-finally*

Zawiera kod obsługi wyjątku i kod porządkujący system.

### Uwagi

Klauzula **catch** służy do przechwytywania określonych wyjątków. Kod zawarty w bloku **catch** jest wykonywany tylko wtedy, gdy dany wyjątek wystąpi. Instrukcja **finally** obejmuje blok kodu źródłowego, który jest wykonywany niezależnie od wystąpienia jakichkolwiek wyjątków w bloku **try**.

### Przyklad

W poniższym przykładzie umieszczono instrukcję dokonującą nieprawidłowej konwersji, która powoduje wystąpienie wyjątku. Po uruchomieniu programu na ekranie pojawi

się komunikat o wystąpieniu błędu, a następnie, po umieszczeniu instrukcji w bloku **finally**, zostanie wyświetlony wynik działania.

```
// try-finally
using System;
public class TestTryFinally
{
   public static void Main()
   {
      int i = 123;
      string s = "Łańcuch znaków";
      object o = s;

      try
      {
         // Zabroniona konwersja; o zawiera łańcuch znaków, a nie wartość typu int
         i = (int) o;
      }

      finally
      {
         Console.Write("i = {0}", i);
      }
   }
}
```

### Wynik

Podczas działania programu wystąpi następujący wyjątek:

```
System.InvalidCastException
```

Pomimo wystąpienia, przechwycenia i obsługi wyjątku, na ekranie pojawi się wartość zmiennej `i` po umieszczeniu w bloku **finally** instrukcji wyświetlającej tą wartość.

```
i =123
```

Więcej informacji na temat słowa kluczowego **finally** znajduje się w punkcie „try-catch-finally".

### Zobacz także:

- **throw** (strona 80), **try-catch** (strona 81).

## try-catch-finally

Instrukcje **catch** i **finally** najczęściej są stosowane razem w celu rezerwacji i wykorzystania zasobów w bloku **try**, przechwycenia i obsługi ewentualnych wyjątków za pomocą klauzuli **catch**, a także w celu zwolnienia zasobów w bloku **finally**.

### Przyklad

```
// try-catch-finally
using System;
public class EHClass
{
```

```
    public static void Main ()
    {
       try
       {
          Console.WriteLine("Wykonywanie instrukcji bloku try.");
          throw new NullReferenceException();
       }

       catch(NullReferenceException e)
       {
          Console.WriteLine("{0} Przechwycono wyjątek #1.", e);
       }

       catch
       {
          Console.WriteLine("Przechwycono wyjątek #2.");
       }

       finally
       {
          Console.WriteLine("Wykonywanie bloku finally.");
       }
    }
}
```

### Wynik

```
Wykonywanie instrukcji bloku try.
System.NullReferenceException: Attempted to dereference a null object reference
at EHClass.Main() Przechwycono wyjątek #1.
Wykonywanie bloku finally.
```

### Zobacz także:

- **throw** (strona 80).

## Instrukcje checked i unchecked

Instrukcje języka C# mogą być wykonywane w kontekście z kontrolą przepełnienia arytmetycznego (**checked**) lub bez takiej kontroli (**unchecked**). W trybie **checked** przepełnienie arytmetyczne powoduje wystąpienie wyjątku. W trybie **unchecked** przepełnienie arytmetyczne jest ignorowane przez kompilator, a wynik operacji jest dostosowany do typu zmiennej.

- `checked` — określa kontekst kontrolowany,
- `unchecked` — określa kontekst niekontrolowany.

Jeśli w stosunku do danego bloku instrukcji nie użyjemy ani **checked**, ani **unchecked**, wówczas domyślny kontekst zależy od czynników zewnętrznych, takich jak np. parametry kompilatora.

Kontekst kontroli przepełnienia dotyczy następujących operacji:

- Wyrażenia wykorzystujące predefiniowane operatory w stosunku do typów całkowitych:

  ++ -- - (jednoargumentowe) + - * /

- Jawne przekształcenia liczbowe pomiędzy typami całkowitymi.

Opcja kompilatora /**checked** umożliwia określenie kontekstu kontroli przepełnienia dla wszystkich instrukcji arytmetycznych, które nie znajdują się jawnie w zasięgu działania słowa kluczowego **checked** lub **unchecked**.

## checked

Słowo kluczowe **checked** służy do sterowania kontrolą przepełnienia podczas działań arytmetycznych i przekształceń typów całkowitych. Słowo kluczowe **checked** jest wykorzystywane jako operator lub instrukcja.

Instrukcja **checked**:

```
checked blok
```

Operator **checked**:

```
checked (wyrażenie)
```

gdzie:

*blok*

Blok instrukcji zawierający wyrażenia wyliczane w kontekście kontrolowanym.

*wyrażenie*

Wyrażenie wyliczane w kontekście kontrolowanym. Należy zauważyć, że wyrażenie musi być umieszczone w nawiasach okrągłych.

### Uwagi

Jeśli w wyniku obliczenia wyrażenia w kontekście kontrolowanym powstanie wartość wykraczająca poza zakres typu docelowego, wynik zależy od tego, czy dane wyrażenie jest stałe, czy nie ma ono stałej wartości. W przypadku wyrażeń stałych wystąpi błąd w trakcie kompilacji, a wyrażenia nie mające stałej wartości spowodują wystąpienie wyjątku w trakcie działania programu.

Jeśli określone wyrażenie nie jest oznaczone jako **checked** lub **unchecked**, to w przypadku wyrażenia stałego domyślnie przeprowadzana jest kontrola przepełnienia w trakcie kompilacji, co oznacza, że używany jest operator **checked**. Gdy wyrażenie nie ma stałej wartości, kontrola przepełnienia zależy od czynników zewnętrznych, takich jak opcje kompilatora czy konfiguracja środowiska.

Poniżej przedstawiono trzy przykłady pokazujące zastosowanie operatorów **checked** i **unchecked** w stosunku do wyrażeń nie mających wartości stałej. We wszystkich przykładach posłużono się tym samym algorytmem, ale innymi kontekstami kontroli. Kontrola przepełnienia następuje w trakcie działania programu.

- Przykład 1: Wykorzystanie wyrażenia z operatorem **checked**.
- Przykład 2: Wykorzystanie wyrażenia z operatorem **unchecked**.
- Przykład 3: Wykorzystanie domyślnego operatora przepełnienia.

Tylko w pierwszym przykładzie wystąpi wyjątek przepełnienia w trakcie działania programu i w tym wypadku będziemy mieli do wyboru przerwanie wykonywania programu lub przejście w tryb poprawiania i kontroli przebiegu programu (debugging mode). W dwóch pozostałych przykładach otrzymamy wartości „przycięte".

Inne przykłady znajdują się w punkcie poświęconym instrukcji **unchecked**.

### Przyklad 1

```
// instrukcje_checked.cs
// Przepełnienie wyrażenia niestałego jest kontrolowane w trakcie działania programu
using System;

class OverFlowTest
{
   static short x = 32767;   // Maksymalna wartość typu short
   static short y = 32767;

   // Wykorzystanie operatora checked
   public static int myMethodCh()
   {
      int z = 0;

      try
      {
        z =  checked((short)(x + y));
      }
      catch (System.OverflowException e)
      {
        System.Console.WriteLine(e.ToString());
      }
      return z;   // Zgłoszenie wyjątku OverflowException
   }

   public static void Main()
   {
      Console.WriteLine("Kontrolowana wartość wynikowa wynosi: {0}", myMethodCh());
   }
}
```

### Przykladowy wynik

Po uruchomieniu programu wystąpi wyjątek `OverflowException`. Można albo debugować program, albo przerwać jego wykonywanie.

```
System.OverflowException: An exception of type System.OverflowException was thrown.
   at OverFlowTest.myMethodCh()
Kontrolowana wartość wynikowa wynosi:  0
```

### Przyklad 2

```
// instrukcje_checked2.cs
```

```
// Wykorzystanie operatora unchecked
// Przepełnienie wyrażenia niestałego jest kontrolowane w trakcie działania programu
using System;

class OverFlowTest
{
   static short x = 32767;   // Maksymalna wartość typu short
   static short y = 32767;

   public static int myMethodUnch()
   {
      int z =  unchecked((short)(x + y));
      return z;   // Zwraca -2
   }

   public static void Main()
   {
      Console.WriteLine("Wartość niekontrolowana wynosi: {0}", myMethodUnch());
   }
}
```

### Wynik

```
Wartość niekontrolowana wynosi: -2
```

### Przyklad 3

```
// instrukcje_checked3.cs
// Wykorzystanie domyślnej kontroli przepełnienia
// Przepełnienie wyrażenia niestałego jest kontrolowane w trakcie działania programu
using System;

class OverFlowTest
{
   static short x = 32767;   // Maksymalna wartość typu short
   static short y = 32767;

   public static int myMethodUnch()
   {
      int z =  (short)(x + y);
      return z;   // Zwraca -2
   }

   public static void Main()
   {
      Console.WriteLine("Wynikowa wartość wynosi: {0}", myMethodUnch());
   }
}
```

### Wynik

```
Wynikowa wartość wynosi: -2
```

### Zobacz także:

- **unchecked**

## unchecked

Słowo kluczowe **unchecked** służy do sterowania kontrolą przepełnienia podczas działań arytmetycznych i przekształceń typów całkowitych. Słowo kluczowe **unchecked** jest wykorzystywane jako operator lub instrukcja.

Instrukcja **unchecked**:

```
unchecked blok
```

Operator **unchecked**:

```
unchecked (wyrażenie)
```

gdzie:

*blok*

Blok instrukcji zawierający wyrażenia wyliczane w kontekście niekontrolowanym.

*wyrażenie*

Wyrażenie wyliczane w kontekście niekontrolowanym. Należy zauważyć, że wyrażenie musi być umieszczone w nawiasie okrągłym.

### Uwagi

Jeśli w wyniku obliczenia wyrażenia w kontekście niekontrolowanym powstanie wartość wykraczająca poza zakres typu docelowego, wartość ta jest przycinana.

Jeśli określone wyrażenie nie jest oznaczone jako **checked** lub **unchecked**, to w przypadku wyrażenia stałego domyślnie przeprowadzana jest kontrola przepełnienia w trakcie kompilacji, co oznacza, że używany jest operator **checked**. Gdy wyrażenie nie ma stałej wartości, kontrola przepełnienia zależy od czynników zewnętrznych, takich jak opcje kompilatora czy konfiguracja środowiska.

Poniżej przedstawiono trzy przykłady pokazujące zastosowanie operatorów **checked** i **unchecked** w stosunku do wyrażeń nie mających wartości stałej. We wszystkich przykładach posłużono się tym samym algorytmem, ale innymi kontekstami kontroli. Kontrola przepełnienia następuje w trakcie działania programu.

- Przykład 1: Wykorzystanie wyrażenia z operatorem **unchecked**.
- Przykład 2: Wykorzystanie domyślnego operatora przepełnienia.
- Przykład 3: Wykorzystanie wyrażenia z operatorem **checked**.

Tylko w pierwszym przykładzie otrzymamy w wyniku „obciętą" wartość. W dwóch pozostałych wystąpi błąd podczas kompilacji.

Inne przykłady znajdują się w punkcie poświęconym instrukcji **checked**.

### Przyklad 1

```
// instrukcje_unchecked.cs
// Wykorzystanie instrukcji unchecked z wyrażeniem stałym
```

```
// Przepełnienie jest sprawdzane w trakcie kompilacji programu
using System;

class TestClass
{
   const int x = 2147483647;   // Maksymalna wartość typu int
   const int y = 2;

   public int MethodUnCh()
   {
      // Instrukcja unchecked:
      unchecked
      {
         int z = x * y;
         return z;   // Zwraca -2
      }
   }

   public static void Main()
   {
      TestClass myObject = new TestClass();
      Console.WriteLine("Wynikowa wartość niekontrolowana wynosi : {0}",
         myObject.MethodUnCh());
   }
}
```

### Wynik

```
Wynikowa wartość niekontrolowana wynosi : -2
```

### Przyklad 2

```
// instrukcje_unchecked2.cs
// Oczekiwany błąd o numerzeCS0220
// Wykorzystanie domyślnej kontroli przepełnienia w stosunku do wyrażenia stałego
// Kontrola przepełnienia następuje w trakcie kompilacji
using System;

class TestClass
{
   const int x = 2147483647;   // Maksymalna wartość typu int
   const int y = 2;

   public int MethodDef()
   {
      // Wykorzystanie domyślnej kontroli przepełnienia:
      int z = x * y;
      return z;   // Błąd kompilatora
   }

   public static void Main()
   {
      TestClass myObject = new TestClass();
      Console.WriteLine("Wartość kontrolowana domyślnie wynosi: {0}",
         myObject.MethodDef());
   }
}
```

### Wynik

**Błąd w trakcie kompilacji**: The operation overflows at compile time in checked mode.

### Przyklad 3

```
// instrukcje_unchecked3.cs
// Oczekiwany błąd CS0220
// Wykorzystanie instrukcji checked w stosunku do wyrażenia stałego
// Kontrola przepełnienia następuje w trakcie kompilacji
using System;

class TestClass
{
   const int x = 2147483647;   // Maksymalna wartość typu int
   const int y = 2;

   public int MethodCh()
   {
      // Instrukcja checked:
      checked
      {
         int z = (x * y);
         return z;   // Błąd kompilatora
      }
   }

   public static void Main()
   {
      TestClass myObject = new TestClass();
      Console.WriteLine("Wartość kontrolowana wynosi: {0}", myObject.MethodCh());
   }
}
```

### Wynik

**Błąd w trakcie kompilacji**: The operation overflows at compile time in checked mode.

### Zobacz także:

- **checked** (strona 87).

## Instrukcja fixed

Instrukcja **fixed** uniemożliwia zmianę adresu określonej zmiennej przez „zbieracza śmieci" (ang. *garbage collector*).

```
fixed (typ* wskaźnik = wyrażenie) instrukcja
```

gdzie:

*typ*

Typ niezarządzany albo pusty (**void**).

*wskaźnik*

Nazwa wskaźnika.

*wyrażenie*

Wyrażenie, które jest niejawnie konwertowane w *typ**.

*instrukcja*

Instrukcja lub blok instrukcji do wykonania.

## Uwagi

Korzystanie z instrukcji **fixed** jest dozwolone tylko w kontekście nienadzorowanym (ang. *unsafe context*).

Instrukcja **fixed** powoduje ustawienie wskaźnika na danej zmiennej i zablokowanie jej adresu na czas wykonywania *instrukcji*. Bez wykorzystania instrukcji **fixed** wskaźniki zmiennych ruchomych są mało użyteczne, bo mechanizm przywracania pamięci może w sposób dowolny relokować położenie obiektów w pamięci. Kompilator języka C# nie zezwala na użycie wskaźnika w stosunku do zarządzanej zmiennej bez wykorzystania instrukcji **fixed**.

```
// zakładamy istnienie klasy Point { public int x, y; }
Point pt = new Point();    // pt jest zmienną zarządzaną, przedmiotem g.c.
fixed ( int* p = &pt.x ){  // należy skorzystać z fixed aby otrzymać adres pt.x i
    *p = 1;                //zablokować pt w trakcie wykorzystywania wskaźnika
}
```

Wskaźnik można inicjalizować adresem tablicy lub łańcucha znaków:

```
fixed (int* p = arr) ...  // równoważne z p = &arr[0]
fixed (char* p = str) ... // równoważne z p = &str[0]
```

Można inicjalizować wiele wskaźników, jeśli wszystkie są tego samego typu:

```
fixed (byte* ps = srcarray, pd = dstarray) {...}
```

Aby zainicjować wskaźniki różnych typów, należy zagnieździć instrukcje **fixed**:

```
fixed (int* p1 = &p.x)
   fixed (double* p2 = &array[5])
      // wykorzystanie wskaźników p1 i p2
```

Wskaźniki zainicjowane w instrukcjach **fixed** nie mogą być modyfikowane.

Po wykonaniu *instrukcji* wszystkie zablokowane zmienne związane z daną instrukcją są odblokowywane i mogą być przedmiotem „zbieracza śmieci", wobec czego nie należy używać wskaźników do tych zmiennych poza instrukcją **fixed**.

W kodzie nienadzorowanym można alokować pamięć na stosie i w tym przypadku „zbieracz śmieci" nie ingeruje w tworzony stos, co sprawia, że nie trzeba blokować położenia bloków pamięci na stosie. Więcej informacji na ten temat znajduje się w punkcie „stackalloc" na stronie 112.

## Przyklad

```
// instrukcje_fixed.cs
// Należy skompilować z: /unsafe
using System;
```

```
class Point {
   public int x, y;
}

class FixedTest
{
   // metoda nienadzorowana: przyjmuje wskaźnik na zmienną typu int
   unsafe static void SquarePtrParam (int* p)
   {
      *p *= *p;
   }

   unsafe public static void Main()
   {
      Point pt = new Point();
      pt.x = 5;
      pt.y = 6;
      // zablokowanie pt
      fixed (int* p = &pt.x)
      {
         SquarePtrParam (p);
      }
      // pt jest już odblokowany
      Console.WriteLine ("{0} {1}", pt.x, pt.y);
   }
}
```

### Wynik

```
25 6
```

### Zobacz także:

- **unsafe** (strona 43).

## Instrukcja lock

Za pomocą słowa kluczowego **lock** określony blok instrukcji jest oznaczany jako sekcja krytyczna.

```
lock (wyrażenie) blok_instrukcji
```

gdzie:

*wyrażenie*

Określa obiekt, który ma być zablokowany. *Wyrażenie* musi być typu referencyjnego.

Najczęściej używanym wyrażeniem jest **this**, gdy chcemy uchronić zmienną egzemplarza, lub **typeof** (*klasa*), jeśli chcemy uchronić zmienną statyczną (albo gdy sekcja krytyczna pojawi się w metodzie statycznej w danej klasie).

*blok_instrukcji*

Instrukcje zawarte w sekcji krytycznej.

## Uwagi

Instrukcja **lock** zapewnia synchronizację wątków. Dzięki niej mamy pewność, że jeśli jeden wątek znajduje się w sekcji krytycznej, to inny wątek nie ma do niej dostępu.

## Przyklad 1

Przykład przedstawia wykorzystanie wątków w języku C#.

```
// instrukcje_lock.cs
using System;
using System.Threading;

class ThreadTest
{
   public void runme()
   {
      Console.WriteLine("wywołano runme");
   }

   public static void Main()
   {
      ThreadTest b = new ThreadTest();
      Thread t = new Thread(new ThreadStart(b.runme));
      t.Start();
   }
}
```