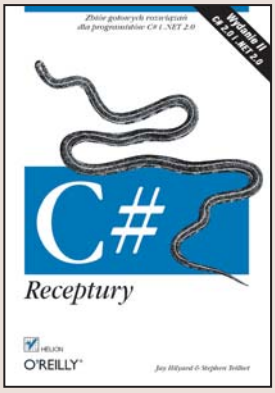
# C#. Receptury. Wydanie II




Autorzy: Jay Hilyard, Stephen Teilhet
Tłumaczenie: Radosław Meryk (przedmowa, rozdz. 1 – 11), Daniel Kaczmarek (rozdz. 12 – 20)
ISBN: 83-246-0476-6
Tytuł oryginału: C# Cookbook
Format: B5, stron: 1064

**Zbiór gotowych rozwiązań dla programistów C# i .NET 2.0**

- Przetwarzanie liczb i tekstów
- Obsługa błędów i wyjątków
- Aplikacje sieciowe

C# to jeden z języków programowania przeznaczonych dla platformy .NET. Został tak skonstruowany, że programiści, którzy wcześniej korzystali z języków Java bądź C++, bez problemu opanują zasady programowania w C#. Według twórcy C# – firmy Microsoft – język ten jest nowatorskim narzędziem programowania na platformie .NET, niesprawiającym kłopotów programistom znającym inne języki, a jednocześnie zapewniającym większą kontrolę nad działającym kodem w fazie wykonywania. W nowej wersji platformy .NET, oznaczonej numerem 2.0, wprowadzono również nową wersję języka C#, oferującą dodatkowe możliwości.

Książka „C#. Receptury. Wydanie II" to zbiór ponad 300 porad, które programistom C# pomogą rozwiązać zadania programistyczne, z jakim spotykają się w codziennej pracy. Przedstawiono w niej metody przetwarzania danych tekstowych i liczbowych, zasady korzystania z wyrażeń regularnych oraz typów generycznych – nowości w C# 2.0. Omówiono sposoby przetwarzania plików XML, obsługę wyjątków oraz reguły tworzenia aplikacji sieciowych ASP.NET i aplikacji dla systemu Windows.

- Operacje na liczbach
- Przetwarzanie ciągów znaków
- Typy generyczne
- Kolekcje i iteratory
- Dzienniki zdarzeń
- Obsługa zdarzeń
- Korzystanie z wyrażeń regularnych
- Operacje na systemie plików
- Tworzenie aplikacji sieciowych
- Zabezpieczanie kodu

**Przyspiesz tempo swojej pracy – korzystaj z gotowych rozwiązań**

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

# Spis treści

ROZDZIAŁ 1.

# Liczby i typy wyliczeniowe

## 1.0. Wprowadzenie

**Typy proste** to podzbiór wbudowanych typów w języku C#. W rzeczywistości typy te zdefiniowano w bibliotece *.NET Framework Class Library* (.NET FCL). Na typy proste składa się kilka typów liczbowych oraz typ `bool`. Do typów liczbowych należy typ dziesiętny (`decimal`), dziewięć typów liczb całkowitych (`byte`, `char`, `int`, `long`, `sbyte`, `short`, `uint`, `ulong`, `ushort`) oraz dwa typy zmiennoprzecinkowe (`float`, `double`). Typy proste wraz z ich pełną, kwalifikowaną nazwą na platformie .NET Framework zestawiono w tabeli 1.1.

*Tabela 1.1. Proste typy danych*

| Pełna nazwa | Alias | Zakres wartości |
|---|---|---|
| `System.Boolean` | `bool` | true lub false |
| `System.Byte` | `byte` | 0 – 255 |
| `System.SByte` | `sbyte` | –128 – 127 |
| `System.Char` | `char` | 0 – 65535 |
| `System.Decimal` | `decimal` | –79 228 162 514 264 337 593 543 850 335 – 79 228 162 514 264 337 593 543 950 335 |
| `System.Double` | `double` | –1,79769313486232e308 – 1,79769313486232e308 |
| `System.Single` | `float` | –3,40282347E+38 – 3,40282347E+38 |
| `System.Int16` | `short` | –32 768 – 32 767 |
| `System.Uint16` | `ushort` | 0 – 65535 |
| `System.Int32` | `int` | –2 147 483 648 – 2 147 483 647 |
| `System.UInt32` | `uint` | 0 – 4 294 967 295 |
| `System.Int64` | `long` | –9 223 372 036 854 775 808 – 9 223 372 036 854 775 807 |
| `System.UInt64` | `ulong` | 0 – 18 446 744 073 709 551 615 |

Słowa kluczowe w języku C# dla różnych typów danych to jedynie aliasy ich pełnych, kwalifikowanych nazw. Nie ma znaczenia, czy w kodzie wykorzystamy pełną nazwę typu czy też słowo kluczowe: kompilator języka C# w obu przypadkach wygeneruje identyczny kod.

Należy zwrócić uwagę, że następujące nazwy typów: `sbyte`, `ushort`, `uint` i `ulong`, nie są zgodne z ogólną specyfikacją języka (ang. *Common Language Specification* — CLS). W efekcie

mogą być nieobsługiwane w innych językach wchodzących w skład platformy .NET. Brak obsługi tych typów może ograniczyć możliwości komunikacji kodu C# z kodem napisanym w innych językach zgodnych z CLS, na przykład w języku Visual Basic .NET.

Typy wyliczeniowe niejawnie dziedziczą własności po typie `System.Enum`, który z kolei dziedziczy własności po typie `System.ValueType`. Typy wyliczeniowe mają jedno zastosowanie: opisanie elementów określonego zbioru; na przykład kolory czerwony (ang. *red*), niebieski (ang. *blue*) i żółty (ang. *yellow*) można zdefiniować jako elementy typu wyliczeniowego `ValidShapeColor`. W podobny sposób można zdefiniować figury: kwadrat (ang. *square*), okrąg (ang. *circle*) i trójkąt (ang. *triangle*) jako elementy typu wyliczeniowego `ValidShape`. Definicja tych typów może mieć następującą postać:

```
enum ValidShapeColor
{
    Red, Blue, Yellow
}
enum ValidShape
{
    Square = 2, Circle = 4, Triangle = 6
}
```

Każdy element typu wyliczeniowego otrzymuje wartość liczbową niezależnie od tego, czy się ją jawnie przypisze czy nie. Ponieważ kompilator automatycznie nadaje elementom typów wyliczeniowych wartości począwszy od zera i zwiększa je o jeden dla każdego kolejnego elementu, zdefiniowany wcześniej typ wyliczeniowy `ValidShapeCreator` można by równie dobrze zdefiniować w sposób następujący:

```
enum ValidShapeColor
{
    Red = 0, Blue = 1, Yellow = 2
}
```

Typy wyliczeniowe to dobre narzędzia do dokumentowania kodu. Na przykład o wiele bardziej intuicyjnie zrozumiały będzie kod zapisany w sposób następujący:

```
ValidShapeColor currentColor = ValidShapeColor.Red;
```

niż zapisany tak:

```
int currentColor = 0;
```

Jedna i druga wersja jest prawidłowa, ale kod zapisany pierwszą metodą lepiej się czyta i jest bardziej zrozumiały. Ma to szczególne znaczenie w przypadku, gdy programista jest zmuszony do czytania kodu, który napisał ktoś inny. Kod w pierwszej wersji jest również **bezpieczny pod względem typów** (ang. *type-safe*), podczas gdy kod, w którym zastosowano wartości `int`, nie charakteryzuje się taką własnością.

# 1.1. Określanie przybliżonej równości pomiędzy wartością ułamkową a zmiennoprzecinkową

## Problem

Trzeba porównać ułamek z wartością typu `double` lub `float` po to, by określić, czy ich wartości są sobie bliskie. Weźmy na przykład wynik porównania wyrażenia 1/6 z wartością 0,16666667.

Wydaje się, że liczby te są sobie równe, jednak liczba 0,16666667 jest wyrażona z dokładnością jedynie do 8 pozycji po przecinku, natomiast dokładność wyrażenia 1/6 jest ograniczona maksymalną liczbą cyfr po przecinku, jakie można zapisać w określonym typie danych.

## Rozwiązanie

W celu porównania przybliżonej równości pomiędzy wartością ułamkową a zmiennoprzecinkową sprawdzimy, czy różnica pomiędzy obiema wartościami mieści się w dopuszczalnych granicach:

```
using System;
// Wersja, w której wykorzystano wartość epsilon typu System.Double.Epsilon
public static bool IsApproximatelyEqualTo(double numerator,
                                          double denominator,
                                          double dblValue)
{
    return IsApproximatelyEqualTo(numerator, denominator, dblValue, double.Epsilon);
}

// Wersja, w której można wprowadzić wartość epsilon
// innego typu niż System.Double.Epsilon
public static bool IsApproximatelyEqualTo(double numerator,
                                          double denominator,
                                          double dblValue,
                                          double epsilon)
{
    double difference = (numerator/denominator) - dblValue;

    if (Math.Abs(difference) < epsilon)
    {
         // Dobre przybliżenie.
         return true;
    }
    else
    {
        // To NIE jest dobre przybliżenie.
        return false;
    }
}
```

Wystarczy zastąpić typ `double` typem `float`, aby stwierdzić, czy ułamek jest w przybliżeniu równy wartości typu `float`.

## Analiza

Wartości ułamkowe można przedstawić, dzieląc licznik przez mianownik, jednak czasami konieczne jest ich zapisywanie w postaci liczb zmiennoprzecinkowych. W takim przypadku pojawiają się problemy związane z zaokrągleniami, które utrudniają porównywanie. Wyrażanie wartości w postaci ułamkowej (np. 1/6) zapewnia maksymalną dokładność. Wyrażanie wartości w postaci zmiennoprzecinkowej (np. 0,1667) ogranicza tę dokładność. W takim przypadku dokładność zależy od liczby cyfr, jaką programista użyje po prawej stronie kropki dziesiętnej.

W niektórych przypadkach trzeba stwierdzić, czy dwie wartości są w przybliżeniu sobie równe. Takie porównanie można osiągnąć poprzez zdefiniowanie wartości (epsilon), która reprezentuje najmniejszą dodatnią wartość dopuszczalnej różnicy pomiędzy dwiema liczbami, aby w dalszym ciągu były uznawane za równe. Mówiąc inaczej, wystarczy wyliczyć wartość bezwzględną

różnicy pomiędzy wartością ułamkową (`nominator/denominator`) a wartością zmiennoprzecinkową (`dblValue`) i porównać ją z predefiniowaną wartością przekazaną jako argument `epsilon`, aby stwierdzić, czy wartość zmiennoprzecinkowa jest dobrym przybliżeniem wartości ułamkowej.

Przeanalizujmy porównanie pomiędzy ułamkiem 1/7 a wartością zmiennoprzecinkową 0,14285714285714285. Poniższe wywołanie metody `IsApproximatelyEqualTo` wskazuje na to, że w wartości zmiennoprzecinkowej jest zbyt mało cyfr po prawej stronie kropki dziesiętnej, aby było ono dobrym przybliżeniem (jest sześć cyfr, a potrzebnych jest siedem):

```
bool Approximate = Class1.IsApproximatelyEqualto(1, 7, .142857, .0000001);
// Approximate == false
```

Po wprowadzeniu kolejnej cyfry dokładności do trzeciego parametru metody okazało się, że uzyskaliśmy dobre przybliżenie ułamka 1/7:

```
bool Approximate = Class1.IsApproximatelyEqualto(1, 7, .1428571, .0000001);
// Approximate == true
```

## Zobacz również

- Tematy *Double.Epsilon Field* i *Single.Epsilon Field* w dokumentacji MSDN.

# 1.2. Konwersja stopni na radiany

## Problem

W przypadku korzystania z funkcji trygonometrycznych klasy `Math` wszystkie jednostki są wyrażone w radianach. Często mamy do czynienia z kątami mierzonymi w stopniach i aby skorzystać z metod klasy, musimy przekształcić je na radiany.

## Rozwiązanie

Aby przekształcić wartość wyrażoną w stopniach na radiany, należy ją pomnożyć przez współczynnik $\pi/180$:

```
using System;

public static double ConvertDegreesToRadians (double degrees)
{
    double radians = (Math.PI / 180) * degrees;
    return (radians);
}
```

## Analiza

We wszystkich statycznych metodach trygonometrycznych klasy `Math` kąty wyrażane są w radianach. Procedury konwersji pomiędzy stopniami a radianami są bardzo przydatne, zwłaszcza gdy użytkownik wprowadza dane w stopniach, a nie w radianach. Pomimo wszystko lepiej rozumiemy stopnie od radianów.

Równanie konwersji stopni na radiany ma następującą postać:

```
radians = (Math.PI / 180) * degrees
```

Statyczne pole `Math.PI` zawiera stałą π.

# 1.3. Konwersja radianów na stopnie

## Problem

W przypadku korzystania z funkcji trygonometrycznych klasy `Math` stopnie są wyrażone w radianach. Często zdarza się, że wyniki obliczeń trzeba podawać w stopniach.

## Rozwiązanie

Aby przekształcić wartość wyrażoną w radianach na stopnie, należy ją pomnożyć przez współczynnik 180/π:

```
using System;

public static double ConvertRadiansToDegrees (double radians)
{
    double degrees = (180 / Math.PI) * radians;
    return (degrees);
}
```

## Analiza

We wszystkich statycznych metodach trygonometrycznych klasy `Math`, kąty wyrażane są w radianach. Procedury konwersji pomiędzy stopniami a radianami są bardzo przydatne. Wyświetlanie wyników w stopniach jest bardziej zrozumiałe dla użytkowników.

Równanie przekształcenia radianów na stopnie ma następującą postać:

```
degrees = (180 / Math.PI) * radians
```

Statyczne pole `Math.PI` zawiera stałą π.

# 1.4. Zastosowanie operatora negacji bitowej do danych różnych typów

## Problem

Operator negacji bitowej (~) można przeciążyć w taki sposób, by można go było wykorzystać bezpośrednio z danymi typu `int`, `uint`, `long`, `ulong`, a także z danymi typów wyliczeniowych składających się z typów `int`, `uint`, `long` i `ulong`. Często jednak występuje potrzeba wykonywania operacji negacji bitowej na danych innych typów liczbowych.

## Rozwiązanie

Aby skorzystać z bitowego operatora negacji dla danych dowolnego typu, należy dokonać konwersji uzyskanej wartości operacji bitowej na typ, który chcemy wykorzystywać. W poniższym kodzie zaprezentowano tę technikę dla typu danych `byte`:

```
byte y =1;
byte result = (byte)~y;
```

W efekcie wykonania powyższej operacji do zmiennej `result` przypisywana jest wartość 254.

## Analiza

Poniższy kod pokazuje nieprawidłowe wykorzystanie operatora negacji bitowej z danymi typu `byte`:

```
byte y = 1;
Console.WriteLine("~y = " + ~y);
```

W wyniku uruchomienia tego kodu nieoczekiwanie wyświetla się liczba –2.

Najwyraźniej wynik wykonania operacji negacji bitowej dla zmiennej typu `byte` jest nieprawidłowy. Prawidłowy wynik to 254. W rzeczywistości `byte` jest typem danych bez znaku, a zatem nie może przyjmować wartości ujemnych. Jeśli zapiszemy kod w następujący sposób:

```
byte y = 1;
byte result = ~y;
```

uzyskamy błąd kompilacji *Cannot implicitly convert type 'int' to 'byte'*[1]. Ten komunikat o błędzie trochę wyjaśnia, dlaczego wcześniejsza operacja nie zadziałała tak, jak się spodziewaliśmy. Aby rozwiązać problem, trzeba jawnie przeprowadzić konwersję wyniku na typ `byte` przed przypisaniem jej do zmiennej `result`, tak jak pokazano poniżej:

```
byte y = 1;
byte result = (byte)~y;
```

Powyższa konwersja jest konieczna, ponieważ operatory bitowe są przeciążane do działania tylko na sześciu typach danych: `int`, `uint`, `long`, `ulong`, `bool` oraz typach wyliczeniowych. Jeśli operator bitowy zostanie użyty w odniesieniu do danych innego typu, zostaną one przekształcane na obsługiwany typ danych, najbardziej zbliżony do typu, dla którego próbowano wykonać operację. Dlatego właśnie przed wykonaniem negacji bitowej dane typu `byte` zostały przekształcone na `int`:

```
        0x01 // byte y = 1;
0xFFFFFFFE // Wartość 01h jest przekształcana na int, a następnie
           // jest dla niej wykonywana negacja bitowa.
           // Taka kombinacja bitów dla danych typu int odpowiada liczbie –2.
        0xFE // Dla uzyskanej wartości int wykonywana jest konwersja na typ byte.
```

Zwróćmy uwagę na to, że `int`, w odróżnieniu od `byte`, jest typem danych ze znakiem. Dlatego właśnie w wyniku uzyskaliśmy –2 zamiast spodziewanej wartości 254. Konwersję typu danych `byte` na najbliższy odpowiednik określa się terminem **promocji numerycznej** (ang. *numeric promotion*). Mechanizm ten wykorzystuje się również w przypadku zastosowania danych różnych typów z operatorami dwuargumentowymi (w tym także bitowymi).

[1] Nie można niejawnie przekształcać danych typu int na byte — *przyp. tłum.*

Mechanizm promocji numerycznej szczegółowo opisano w specyfikacji języka C# (*C# Language Specification*) w punkcie 7.2.6 (dokument jest dostępny pod adresem *http://msdn.microsoft.com/vcsharp/programming/language*). Dokładne zrozumienie działania promocji numerycznej ma istotne znaczenie w przypadku wykorzystania operatorów na danych różnych typów, a także w przypadku zastosowania operatorów z takimi typami danych, dla których nie istnieje przeciążona wersja operatora obsługująca je. Dzięki znajomości mechanizmu promocji numerycznej można zaoszczędzić wiele godzin debugowania.

# 1.5. Sprawdzenie, czy liczba jest parzysta czy nieparzysta

## Problem

Potrzebna jest szybka metoda umożliwiająca stwierdzenie, czy wartość liczbowa jest parzysta czy nieparzysta.

## Rozwiązanie

Rozwiązanie można zaimplementować w postaci dwóch metod. W celu sprawdzenia, czy liczba jest parzysta, można wykorzystać następującą metodę:

```
public static bool IsEven(int intValue)
{
    return ((intValue % 2) == 0);
}
```

Aby sprawdzić, czy liczba jest nieparzysta, można zastosować następującą metodę:

```
public static bool IsOdd(int intValue)
{
    return((intValue % 2) == 1);
}
```

## Analiza

W każdej liczbie nieparzystej najmniej znaczący bit zawsze ma wartość 1. Właśnie dlatego, aby stwierdzić, że określona liczba jest nieparzysta, wystarczy skontrolować, czy jej najmniej znaczący bit ma wartość 1. Z kolei poprzez sprawdzenie, czy najmniej znaczący bit ma wartość 0, można stwierdzić, czy liczba jest parzysta.

W celu sprawdzenia, czy wartość jest parzysta, wykonujemy operację `AND` tej wartości z liczbą 1, a następnie kontrolujemy, czy wynik tej operacji jest zerem. Jeśli tak, możemy stwierdzić, że testowana liczba jest parzysta, a jeśli nie, jest nieparzysta. Opisaną powyżej operację wykonano w metodzie `IsEven`.

Z kolei aby sprawdzić, czy wartość jest nieparzysta, należy podobnie jak poprzednio wykonać operację `AND` tej wartości z liczbą 1 i skontrolować, czy uzyskany wynik ma wartość 1. Jeśli tak, możemy stwierdzić, że testowana liczba jest nieparzysta, w przeciwnym przypadku jest parzysta. Opisywaną operację wykonano w metodzie `IsOdd`.

Należy zwrócić uwagę, że w danej aplikacji nie ma potrzeby implementacji obu metod — `IsOdd` i `IsEven` — choć implementacja ich obydwu z pewnością zwiększy czytelność kodu. Bez trudu można zaimplementować jedną z metod, używając drugiej. Oto przykład implementacji metody `IsOdd` za pomocą metody `IsEven`:

```
public static bool IsOdd(int intValue)
{
    return (!IsEven(intValue));
}
```

Pokazane powyżej metody można stosować wyłącznie wobec 32-bitowych liczb całkowitych. Aby zastosować je w odniesieniu do innych typów liczbowych, wystarczy je przeciążyć dla dowolnego potrzebnego typu. Na przykład, aby stwierdzić, czy 64-bitowa liczba całkowita jest parzysta, można zmodyfikować metodę `IsEven` w sposób następujący:

```
public static bool isEven(long longValue)
{
    return ((longValue & 1) == 0);
}
```

Wystarczyło zmodyfikować typ danych na liście parametrów.

# 1.6. Uzyskanie bardziej znaczącego i mniej znaczącego słowa liczby

## Problem

Mamy 32-bitową liczbę całkowitą zawierającą informacje zarówno w mniej znaczących, jak i w bardziej znaczących 16 bitach. Potrzebujemy metody, która odczytałaby z tej liczby bardziej znaczące słowo (pierwsze 16 bitów) i (lub) mniej znaczące słowo (ostatnie 16 bitów).

## Rozwiązanie

Aby uzyskać bardziej znaczące słowo liczby całkowitej, należy wykonać bitową operacją `AND` tej liczby z wartością `0xFFFF << 16`, tak jak pokazano poniżej:

```
public static int GetHighWord(int intValue)
{
    return (intValue & (0xFFFF << 16));
}
```

Aby uzyskać mniej znaczące słowo liczby, można skorzystać z następującej metody:

```
public static int GetLowWord(int intValue)
{
    return (intValue & 0x0000FFFF);
}
```

Technikę tę można z łatwością zmodyfikować do zastosowania z liczbami całkowitymi innych rozmiarów (8-bitowych, 16-bitowych bądź 64-bitowych). Sposoby wykonania tych działań omówiono w punkcie „Analiza”.

## Analiza

Aby wyznaczyć wartość bardziej znaczącego słowa liczby, wystarczy skorzystać z następującej bitowej operacji AND:

```
uint intValue = Int32.MaxValue;
uint MSB = intValue & (OxFFFF << 16);

// MSB == 0xFFFF0000
```

Powyższa metoda wykonuje operację AND liczby z inną liczbą — taką, w której wszystkie bity bardziej znaczącego słowa są ustawione na 1. Wykonanie tej metody spowoduje wyzerowanie wszystkich bitów mniej znaczącego słowa, natomiast bity słowa bardziej znaczącego pozostaną bez zmian.

Aby wyznaczyć wartość mniej znaczącego słowa liczby, można skorzystać z następującej operacji AND:

```
uint intValue = Int32.MaxValue;
uint LSB = intValue & OxOOOOFFFF;

// LSB == 0x0000FFFF
```

Powyższa metoda wykonuje operację AND liczby z inną liczbą — taką, w której wszystkie bity mniej znaczącego słowa są ustawione na 1. Wykonanie tej metody spowoduje wyzerowanie wszystkich bitów bardziej znaczącego słowa, natomiast bity mniej znaczącego słowa pozostaną bez zmian.

Pokazane powyżej metody można stosować wyłącznie do 32-bitowych liczb całkowitych. Aby zastosować je w odniesieniu do innych typów liczbowych, wystarczy je przeciążyć dla dowolnego potrzebnego typu. Na przykład, aby wyznaczyć mniej i bardziej znaczące słowa liczby 16-bitowej, można skorzystać z metody o podobnej strukturze do metody GetHighWord:

```
public static short GetHighByte(short shortValue)
{
    return (short)(shortValue & (OxFF << 8));
}
```

Metodę GetLowWord można zmodyfikować następująco:

```
public static short GetLowByte(short shortValue)
{
    return (short)(shortValue & (short)OxFF);
}
```

# 1.7. Konwersja liczby z innego systemu liczbowego na dziesiętny

## Problem

Mamy ciąg znaków zawierający liczbę wyrażoną w systemie dwójkowym, ósemkowym, dziesiętnym lub szesnastkowym. Chcemy przekształcić ten ciąg znaków na odpowiadającą mu liczbę całkowitą i wyświetlić w postaci dziesiętnej.

## Rozwiązanie

Aby przekształcić liczbę wyrażoną w innym systemie liczbowym na system dziesiętny, wystarczy skorzystać z przeciążonej wersji statycznej metody `Convert.ToInt32` należącej do klasy `Convert`:

```
string base2 = "11";
string base8 = "17";
string base10 = "110";
string base16 = "11FF";

Console.WriteLine("Convert.ToInt32(base2, 2) = " +
                  Convert.ToInt32(base2, 2));

Console.WriteLine("Convert.ToInt32(base8, 8) = " +
                  Convert.ToInt32(base8, 8));

Console.WriteLine("Convert.ToInt32(base10, 10) = " +
                  Convert.ToInt32(base10, 10));

Console.WriteLine("Convert.ToInt32(base16, 16) = " +
                  Convert.ToInt32(base16, 16));
```

Wykonanie powyższego kodu spowoduje wyświetlenie następującego wyniku:

```
Convert.ToInt32(base2, 2) = 3
Convert.ToInt32(base8, 8) = 15
Convert.ToInt32(base10, 10) = 110
Convert.ToInt32(base16, 16) = 4607
```

## Analiza

Dla statycznej metody `Convert.ToInt32` istnieje przeciążona wersja pobierająca ciąg zawierający liczbę oraz liczbę całkowitą definiującą podstawę systemu liczbowego, w jakim tę liczbę wyrażono. Następnie metoda przekształca ciąg zawierający liczbę na wartość całkowitą typu `integer`, po czym metoda `Console.WriteLine` przekształca liczbę na postać dziesiętną i ją wyświetla.

Dla innych statycznych metod klasy `Convert`, takich jak `ToByte`, `ToInt64` oraz `ToInt16`, istnieją podobne przeciążone wersje, które pobierają liczbę w postaci ciągu znaków oraz podstawę systemu liczbowego, w którym wyrażono tę liczbę. Niestety, metody te jedynie przekształcają znakowe reprezentacje liczb wyrażonych w systemach dwójkowym, ósemkowym, dziesiętnym i szesnastkowym na postać dziesiętną. Nie pozwalają na konwersję liczby na ciąg znaków wyrażony w innym systemie liczbowym niż dziesiętny. Na taką konwersję pozwalają jednak metody `ToString` dostępne dla różnych typów liczbowych.

## Zobacz również

- Tematy *Convert Class* oraz *Converting with System.Convert* w dokumentacji MSDN.

# 1.8. Sprawdzenie, czy ciąg znaków reprezentuje prawidłową liczbę

## Problem

Mamy ciąg znaków, który być może zawiera wartość liczbową, Chcemy się przekonać, czy tak jest.

## Rozwiązanie

Można zastosować metodę `TryParse` dowolnego typu liczbowego. Na przykład, aby sprawdzić, czy ciąg zawiera wartość typu `double`, można zastosować następującą metodę:

```
string str = "12.5";
double result = 0;
if(double.TryParse(str,
       System.Globalization.NumberStyles.Float,
       System.Globalization.NumberFormatInfo.CurrentInfo,
       out result))
{
     // liczba jest typu double!
}
```

## Analiza

W tej recepturze pokazano, w jaki sposób stwierdzić, czy ciąg znaków zawiera wyłącznie wartość liczbową. Metoda `TryParse` zwraca `true` w sytuacji, gdy ciąg znaków zawiera poprawną liczbę, bez zgłaszania wyjątku właściwego dla metody `Parse`. Ponieważ metoda `TryParse` nie zgłasza wyjątków, działa wydajniej w przypadku jej zastosowania dla szeregu ciągów znaków, z których niektóre nie zawierają liczb.

## Zobacz również

- Tematy *Parse* i *TryParse* w dokumentacji MSDN.

# 1.9. Zaokrąglanie wartości zmiennoprzecinkowych

## Problem

Trzeba zaokrąglić liczbę do wartości całkowitej bądź do określonej liczby miejsc po przecinku.

## Rozwiązanie

W celu zaokrąglenia dowolnej liczby do najbliższej liczby całkowitej, można zastosować przeciążoną wersję statycznej metody `Round`, która pobiera pojedynczy argument:

```
int x = (int)Math.Round(2.5555); // x == 3
```

Aby zaokrąglić wartość zmiennoprzecinkową do określonej liczby miejsc po przecinku, można skorzystać z przeciążonej wersji metody `Math.Round` pobierającej dwa argumenty:

```
decimal x = Math.Round(2.5555, 2); // x == 2.56
```

## Analiza

Metoda `Round` jest łatwa w użyciu, podczas jej stosowania trzeba jednak pamiętać, w jaki sposób działa operacja zaokrąglania. Metoda `Round` jest zgodna z punktem 4. standardu IEE 754. Oznacza to, że jeśli zaokrąglana liczba znajduje się w połowie pomiędzy dwoma liczbami, operacja `Round` zawsze zaokrągli ją do wartości parzystej. Oto przykład:

```
decimal x = Math.Round(1.5); // x == 2
decimal y = Math.Round(2.5); // y == 2
```

Zwróćmy uwagę, że liczbę 1,5 zaokrąglono w górę do najbliższej całkowitej liczby parzystej, natomiast liczbę 2,5 zaokrąglono w dół do najbliższej całkowitej liczby parzystej. Należy o tym pamiętać podczas korzystania z operacji `Round`.

## Zobacz również

- Temat *Math Class* w dokumentacji MSDN.
- Receptury 1.1 i 1.22.

# 1.10. Wybór algorytmu zaokrąglania

## Problem

W przypadku zastosowania metody `Math.Round` liczba 1,5 zostanie zaokrąglona do wartości 2, podobnie jak liczba 2,5. Często zdarza się, że w takiej sytuacji chcemy zaokrąglać w górę (tzn. 2,5 po zaokrągleniu ma wynosić 3, a nie 2) lub w dół (tzn. 1,5 po zaokrągleniu ma mieć wartość 1).

## Rozwiązanie

Aby zawsze zaokrąglać w górę w przypadku, gdy wartość liczby leży dokładnie w połowie pomiędzy dwiema liczbami całkowitymi, należy skorzystać z metody `Math.Floor`:

```
public static double RoundUp(double valueToRound)
{
    return (Math.Floor(valueToRound + 0.5));
}
```

Aby zawsze zaokrąglać w dół liczby leżące dokładnie w połowie pomiędzy dwiema wartościami całkowitymi, można zastosować następującą technikę:

```
public static double RoundDown(double valueToRound)
{
    double floorValue = Math.Floor(valueToRound);
    if ((valueToRound - floorValue) > .5)
    {
        return (floorValue + 1);
    }
```

```
        else
        {
            return (floorValue);
        }
    }
```

## Analiza

Statyczna metoda `Math.Round` zaokrągla liczby w górę do najbliższej liczby parzystej (więcej informacji można znaleźć w recepturze 1.9). Czasami jednak są sytuacje, kiedy nie chcemy zaokrąglać liczb w ten sposób. Aby zastosować inny sposób zaokrąglania, można skorzystać ze statycznej metody `Math.Floor`.

Należy zwrócić uwagę, że metody zastosowane do zaokrąglania liczb w tej recepturze nie zaokrąglają do określonej liczby miejsc dziesiętnych, ale do najbliższej liczby całkowitej.

## Zobacz również

- Temat *Math Class* w dokumentacji MSDN.
- Receptury 1.9 i 1.22.

# 1.11. Zamiana stopni Celsjusza na stopnie Fahrenheita

## Problem

Odczytaliśmy temperaturę w stopniach Celsjusza i chcemy zamienić ją na stopnie Fahrenheita.

## Rozwiązanie

```
public static double CtoF(double celsius)
{
    return(((0.9/0.5) * celsius) +32);
}
```

Aby wygenerować wynik typu `double` z zachowaniem tego samego współczynnika jak dla liczb całkowitych (9 do 5), w obliczeniach wykorzystano liczby 0.9 i 0.5.

## Analiza

W niniejszej recepturze wykorzystano następujące równanie konwersji stopni Celsjusza na stopnie Fahrenheita:

```
TempFahrenheit = ((9 / 5) * TempCelsius) + 32
```

Skalę temperatury Fahrenheita powszechnie stosuje się w Stanach Zjednoczonych. W większości pozostałej części świata zazwyczaj stosuje się stopnie Celsjusza.

# 1.12. Zamiana stopni Fahrenheita na stopnie Celsjusza

## Problem

Odczytaliśmy temperaturę w stopniach Fahrenheita i chcemy zamienić ją na stopnie Celsjusza.

## Rozwiązanie

```
public static double FtoC(double fahrenheit)
{
    return(((0.5/0.9) * fahrenheit) - 32);
}
```

## Analiza

W niniejszej recepturze wykorzystano następujące równanie konwersji stopni Fahrenheita na Celsjusza:

```
TempCelsius = (0.5 / 0.9) * (TempFahrenheit – 32)
```

Skalę temperatury Fahrenheita powszechnie stosuje się w Stanach Zjednoczonych. W większości pozostałej części świata zazwyczaj stosuje się stopnie Celsjusza.

# 1.13. Bezpieczna konwersja liczb do mniejszego rozmiaru

## Problem

Trzeba dokonać konwersji liczby o większym rozmiarze do mniejszego formatu bez utraty informacji. Na przykład konwersja liczby typu `long` na `int` powoduje utratę informacji tylko wtedy, gdy liczba typu `long` poddawana konwersji jest większa od `int.MaxSize`.

## Rozwiązanie

Najprostsze rozwiązanie problemu polega na skorzystaniu ze słowa kluczowego `checked`. W zamieszczonej poniżej metodzie pobrano dwa argumenty typu `long` i podjęto próbę dodania ich do siebie. Wynik ma być zwrócony jako liczba całkowita. Jeśli nastąpi przepełnienie, metoda zgłosi wyjątek `OverflowException`:

```
using System;

public static void UseChecked(long lhs, long rhs)
{
    int result = 0;

    try
    {
        result = checked((int)(lhs + rhs));
    }
```

```
    catch (OverflowException e)
    {
        // Obsługa wyjątku przepełnienia.
    }
}
```

Jest to najprostsza metoda. Jeśli jednak nie chcemy ponosić kosztów zgłaszania wyjątków i ujmować kodu dużej objętości w bloku `try-catch` w celu obsłużenia przepełnienia, możemy wykorzystać pola `MaxValue` i `MinValue` dla każdego typu danych. Przed konwersją można przeprowadzić test z wykorzystaniem tych pól i się upewnić, czy nie nastąpi utrata informacji. Jeśli tak, można przekazać do aplikacji komunikat, że wykonywana konwersja doprowadzi do utraty informacji. Aby stwierdzić, czy wartość `sourceValue` da się bezpiecznie (bez utraty informacji) poddać konwersji na typ danych `short`, można skorzystać z następującej instrukcji warunkowej:

```
// Deklaracja i zainicjowanie dwóch zmiennych.
int sourceValue = 34000;
short destinationValue = 0;

// Sprawdzenie, czy konwersja zmiennej sourceValue na typ short doprowadzi do utraty informacji.
if (sourceValue <= short.MaxValue && sourceValue >= short.MinValue)
{
    destinationValue = (short)sourceValue;
}
else
{
    // Poinformowanie aplikacji o utracie informacji.
}
```

## Analiza

**Konwersja zawężająca** (ang. *narrowing conversion*) zachodzi w przypadku, gdy dana typu o większym rozmiarze jest przekształcana na typ o mniejszym rozmiarze. Może tak się zdarzyć, na przykład, w przypadku konwersji wartości typu `Int32` na `Int16`. Jeśli wartość typu `Int32` jest mniejsza bądź równa wartości pola `Int16.MaxValue`, natomiast wartość `Int32` jest większa bądź równa wartości pola `Int16.MinValue`, konwersja przebiegnie bez błędów oraz bez utraty informacji. Utrata informacji wystąpi w przypadku, gdy wartość typu `Int32` będzie większa od wartości pola `Int16.MaxValue` lub wartość `Int32` będzie mniejsza niż wartość pola `Int16.MinValue`. W każdym z tych przypadków najbardziej znaczące bity wartości `Int32` zostaną obcięte i odrzucone, co spowoduje, że wartość po konwersji będzie inna niż przed konwersją.

Jeśli w kodzie działającym w **kontekście niekontrolowanym** (ang. *unchecked context*) wystąpi utrata informacji, nie zostanie zgłoszony żaden wyjątek i aplikacja nie wykryje takiej sytuacji. Problem ten może spowodować powstanie błędów bardzo trudnych do wyśledzenia. Aby im zapobiec, należy sprawdzić, czy wartość, która ma być poddana konwersji, mieści się pomiędzy dolnym a górnym limitem typu, na który ma nastąpić konwersja. Jeśli wartość leży poza tym zakresem, można napisać kod, który obsłuży tę sytuację. Taki kod pozwala na zrezygnowanie z konwersji i (lub) poinformowanie aplikacji o potencjalnym problemie. Zaprezentowane rozwiązanie może zapobiec wystąpieniu w aplikacji trudnych do wykrycia błędów arytmetycznych.

Należy zwrócić uwagę, że obie techniki pokazane w punkcie „Rozwiązanie" są prawidłowe. Wykorzystana technika zależy jednak od tego, czy spodziewamy się częstych przypadków przepełnienia czy tylko pojedynczych. Jeśli sytuacje przepełnienia będą występować często,

lepiej wybrać drugą technikę polegającą na ręcznym sprawdzaniu wartości liczbowej. W innym przypadku lepiej skorzystać ze słowa kluczowego `checked`, tak jak pokazano w pierwszej z technik.

W języku C# kod może działać w **kontekście kontrolowanym** (ang. *checked*) bądź **niekontrolowanym** (ang. *unchecked*). Domyślnie kod działa w kontekście niekontrolowanym. W kontekście kontrolowanym wszystkie obliczenia arytmetyczne i operacje konwersji z wykorzystaniem typów całkowitych są sprawdzane pod kątem przepełnienia. Jeśli wystąpi przepełnienie, kod zgłasza wyjątek `OverflowException`. W kontekście niekontrolowanym w przypadku przepełnienia wyjątek `OverflowException` nie będzie zgłoszony.

Kontekst kontrolowany można włączyć za pomocą opcji kompilatora `/checked{+}`, ustawiając właściwość projektu *Check for Arithmetic Overflow/Underflow* na wartość `true` lub za pomocą słowa kluczowego `checked`. Kontekst niekontrolowany można włączyć za pomocą opcji kompilatora `/checked-`, ustawiając właściwość projektu *Check for Arithmetic Overflow/Underflow* na wartość `false` lub za pomocą słowa kluczowego `unchecked`.

Zwróćmy uwagę, że w kodzie obsługującym konwersję na typy całkowite zaprezentowanym w tej recepturze nie uwzględniono typów zmiennoprzecinkowych. Jest tak dlatego, ponieważ konwersja z dowolnego typu całkowitego na typy `float`, `double` lub `decimal` nigdy nie powoduje utraty informacji. W związku z tym wprowadzanie dodatkowych testów jest zbędne.

Podczas wykonywania konwersji dodatkowo należy zwrócić uwagę na następujące zagadnienia:

- Konwersja danych typu `float`, `double` lub `decimal` na typy całkowite powoduje obcięcie części ułamkowej liczby. Co więcej, jeśli część całkowita liczby przekracza wartość pola `MaxValue` typu docelowego, uzyskana wartość będzie niezdefiniowana, chyba że konwersja jest wykonywana w kontekście kontrolowanym. W takim przypadku nastąpi zgłoszenie wyjątku `OverflowException`.
- Konwersja danych typu `float` lub `double` na `decimal` powoduje zaokrąglenie danych typu `float` lub `double` do 28 miejsc po przecinku.
- Konwersja danych z typu `double` na `float` powoduje zaokrąglenie wartości typu `double` do najbliższej wartości typu `float`.
- Konwersja danych typu `decimal` na `float` lub `double` powoduje zaokrąglenie danych typu `decimal` do typu wyniku (`float` lub `double`).
- Konwersja z typów `int`, `uint` lub `long` do typu `float` może spowodować utratę dokładności, ale nigdy rzędu wielkości.
- Konwersja z typu `long` na `double` może spowodować utratę dokładności, ale nigdy rzędu wielkości.

## Zobacz również

- Tematy *Checked Keyword* oraz *Checked and Unchecked* w dokumentacji MSDN.

# 1.14. Wyznaczanie długości dowolnego z trzech boków trójkąta prostokątnego

## Problem

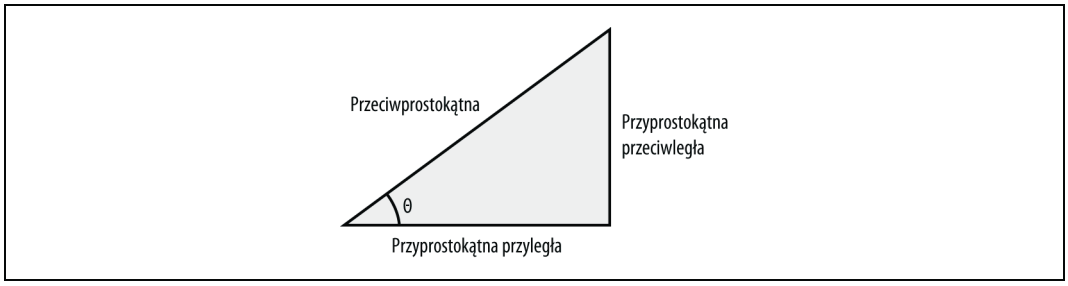
Trzeba obliczyć długość jednego z boków trójkąta w przypadku, gdy jest znana długość dwóch boków bądź jeden z kątów i długość jednego boku.

## Rozwiązanie

Aby znaleźć długość boku, można skorzystać z metod `Math.Sin`, `Math.Cos` i `Math.Tan` klasy `Math`. Oto równania, w których wykorzystamy wymienione wyżej metody:

```
double theta = 30;
double hypotenuse = 5;
double oppositeSide = 0;
double adjacentSide = 0;
oppositeSide = Math.Sin(theta) * hypotenuse;
oppositeSide = Math.Tan(theta) * adjacentSide;
adjacentSide = Math.Cos(theta) * hypotenuse;
adjacentSide = oppositeSide / Math.Tan(theta);
hypotenuse = adjacentSide / Math.Sin(theta);
hypotenuse = adjacentSide / Math.Cos(theta);
```

W powyższych wzorach `theta` oznacza znany kąt, zmienna `oppositeSide` ma wartość równą przyprostokątnej przeciwległej do kąta `theta`, natomiast zmienna `adjacentSide` odpowiada długości boku przyległego do kąta `theta`. Zmienna `hypotenuse` odpowiada długości przeciwprostokątnej trójkąta (rysunek 1.1).




*Rysunek 1.1. Trójkąt prostokątny*

Oprócz wymienionych trzech statycznych metod długość przeciwprostokątnej trójkąta prostokątnego można obliczyć z twierdzenia Pitagorasa. Zgodnie z nim długość przeciwprostokątnej trójkąta prostokątnego jest równa pierwiastkowi kwadratowemu z sumy kwadratów pozostałych dwóch boków. Powyższe równanie można rozwiązać za pomocą metody statycznej `Math.Sqrt`, w następujący sposób:

```
double hypotenuse = Math.Sqrt((xSide * xSide) + (ySide * ySide));
```

gdzie `xSide` i `ySide` są długościami przyprostokątnych trójkąta.

## Analiza

Obliczenie długości boku trójkąta prostokątnego jest proste w przypadku, gdy znany jest jeden kąt oraz długość dowolnego boku. Wykorzystując funkcje trygonometryczne: sinus, kosinus i tangens, można wyznaczyć długości nieznanych boków. Oto równania dla sinusa, kosinusa i tangensa:

```
sin(theta) = oppositeSide / hypotenuseSide
cos(theta) = adjacentSide / hypotenuseSide
tan(theta) = oppositeSide / adjacentSide
```

gdzie `theta` oznacza wartość znanego kąta. Po przekształceniu powyższych równań otrzymujemy:

```
oppositeSide = Math.Sin(theta) * hypotenuse;
oppositeSide = Math.Tan(theta) * adjacentSide;
adjacentSide = Math.Cos(theta) * hypotenuse;
adjacentSide = oppositeSide / tan(theta);
hypotenuse = adjacentSide / sin(theta);
hypotenuse = adjacentSide / cos(theta);
```

Powyższe równania umożliwiają wyznaczenie długości dowolnego z boków trójkąta dwoma metodami.

Jeśli nie znamy żadnego kąta, ale znamy długości dwóch boków, możemy skorzystać z twierdzenia Pitagorasa w celu wyznaczenia długości pozostałego boku. Twierdzenie Pitagorasa można opisać za pomocą następującego równania:

```
Math.Sqrt(hypotenuse * hypotenuse) = Math.Sqrt((xSide * xSide) +
                                               (ySide * ySide))
```

Po uproszczeniu tego równania do postaci składni języka C#, uzyskujemy następujący kod:

```
double hypotenuse = math.Sqrt((xSide * xSide) +
                              (ySide * ySide));
```

gdzie zmienna `hypotenuse` ma wartość odpowiadającą długości przeciwprostokątnej, natomiast zmienne `xSide` i `ySide` odpowiadają długościom przyprostokątnych.

## Zobacz również

- Temat *Math.Class* w dokumentacji MSDN.

# 1.15. Obliczanie kątów trójkąta prostokątnego

## Problem

Trzeba obliczyć kąty trójkąta prostokątnego w sytuacji, gdy znane są długości dwóch boków.

## Rozwiązanie

Można skorzystać ze statycznych metod `Math.Atan`, `Math.Acos` lub `Math.Asin` klasy `Math`. Kąt `theta` wyrażony w radianach można uzyskać za pomocą następującego kodu:

```
double theta = Math.Atan(OppositeSide / AdjacentSide);
theta = Math.Acos(AdjacentSide / Hypotenuse);
theta = Math.Asin(OppositeSide / Hypotenuse);
```

Aby uzyskać kąt w stopniach, można skorzystać z następującego kodu:

```
double theta = Math.Atan(oppositeSide / adjacentSide) * (180 / Math.PI);
theta = Math.Acos(adjacentSide / hypotenuse) * (180 / Math.PI);
theta = Math.Asin(oppositeSide / hypotenuse) * (180 / Math.PI);
```

gdzie `theta` ma wartość znanego kąta, `oppositeSide` jest równa długości przyprostokątnej przeciwległej do kąta, natomiast `adjacentSide` — długości przyprostokątnej przyległej do kąta. Zmienna `hypotenuse` ma wartość odpowiadającą długości przeciwprostokątnej trójkąta. Graficzną reprezentację trójkąta prostokątnego zaprezentowano na rysunku 1.1 w recepturze 1.14, natomiast sposoby konwersji stopni na radiany i odwrotnie opisano w recepturach 1.2 i 1.3.

## Analiza

Zdarza się, że trzeba wyznaczyć kąt trójkąta prostokątnego w sytuacji, kiedy znane są długości tylko dwóch boków. Na podstawie tych danych można wyznaczyć dowolny kąt w trójkącie prostokątnym, wykorzystując jedną z trzech funkcji trygonometrycznych: arcus sinus, arcus cosinus lub arcus tangens. Odpowiednie działania trygonometryczne można wykonać, posługując się statycznymi metodami `Math.Atan`, `Math.Acos` i `Math.Asin` klasy `Math`.

## Zobacz również

- Temat *Math Class* w dokumentacji MSDN.
- Receptura 1.14.

# 1.16. Wyświetlanie wartości typu wyliczeniowego w postaci tekstowej

## Problem

Trzeba wyświetlić wartość tekstową lub liczbową typu wyliczeniowego.

## Rozwiązanie

Aby wyświetlić wartość typu wyliczeniowego w postaci tekstowej, można wykorzystać metodę `ToString`, którą każdy element typu wyliczeniowego dziedziczy po typie `System.Enum`.

W przykładowym typie wyliczeniowym `ValidShape`, którego deklarację zamieszczono poniżej, można uzyskać zarówno liczbowe, jak tekstowe wartości poszczególnych elementów typu.

```
enum ValidShape
{
    Square, Circle, Cylinder, Octagon
}
```

Za pomocą metody `ToString` typu wyliczeniowego `ValidShape` można bezpośrednio uzyskać określoną wartość typu wyliczeniowego `ValidShape`:

```
Console.WriteLine(ValidShape.ToString());
Console.WriteLine(ValidShape.ToString("G"));
Console.WriteLine(ValidShape.ToString("D"));
Console.WriteLine(ValidShape.ToString("F"));
Console.WriteLine(ValidShape.ToString("X"));
```

Wykonanie tego kodu spowoduje wyświetlenie następującego wyniku:

```
Circle
Circle
1
Circle
00000001
```

W podobny sposób można uzyskać wartości typu wyliczeniowego w przypadku posługiwania się zmiennymi typu `ValidShape`:

```
ValidShape shapeStyle = ValidShape.Cylinder;

Console.WriteLine(shapeStyle.ToString());
Console.WriteLine(shapeStyle.ToString("G"));
Console.WriteLine(shapeStyle.ToString("D"));
Console.WriteLine(shapeStyle.ToString("F"));
Console.WriteLine(shapeStyle.ToString("X"));
```

Wykonanie kodu spowoduje wyświetlenie następującego wyniku:

```
Cylinder
Cylinder
2
Cylinder
00000002
```

## Analiza

Uzyskanie tekstowej bądź liczbowej reprezentacji wartości typu wyliczeniowego jest proste, jeśli skorzysta się z metody `ToString` typu `Enum`. Metoda pobiera argument w postaci znaku reprezentującego typ formatowania do zastosowania dla wartości typu wyliczeniowego. Może to być jeden ze znaków: `G`, `g`, `D`, `d`, `X`, `x`, `F` lub `f`. Znaczenie poszczególnych typów formatowania opisano w tabeli 1.2.

*Tabela 1.2. Typy formatowania*

| Typ formatowania | Nazwa | Opis |
|---|---|---|
| `G` lub `g` | Ogólny (ang. *general*) | Wyświetla tekstową reprezentację wartości typu wyliczeniowego. |
| `F` lub `f` | Flaga (ang. *flag*) | Wyświetla tekstową reprezentację wartości typu wyliczeniowego. Wartość typu wyliczeniowego jest interpretowana tak, jakby była polem bitowym. |
| `D` lub `d` | Dziesiętny (ang. *decimal*) | Wyświetla dziesiętny odpowiednik wartości typu wyliczeniowego. |
| `X` lub `x` | Szesnastkowy (ang. *hexadecimal*) | Wyświetla szesnastkowy odpowiednik wartości typu wyliczeniowego. |

Wartości typu wyliczeniowego z atrybutem `Flags` są wyświetlane z uwzględnieniem wykonanych operacji `OR` dla więcej niż jednej wartości. W efekcie, w zależności od wybranego formatowania, uzyskujemy wszystkie wartości typu wyliczeniowego poddane operacji `OR` wyświetlone

w postaci ciągów znaków rozdzielonych przecinkami lub w postaci liczbowego wyniku operacji OR. Jako przykład rozważmy sytuację, w której do typu wyliczeniowego ValidShape dodano atrybut Flags w następujący sposób:

```
[Flags]
enum ValidShape
{
    Square = 0, Circle = 1, Cylinder = 2, Octagon = 4
}
```

Jeśli teraz zmodyfikujemy kod naszej receptury w sposób następujący:

```
ValidShape shapeStyle = ValidShape.Circle | ValidShape.Cylinder;

Console.WriteLine(shapeStyle.ToString());
Console.WriteLine(shapeStyle.ToString("G"));
Console.WriteLine(shapeStyle.ToString("D"));
Console.WriteLine(shapeStyle.ToString("F"));
Console.WriteLine(shapeStyle.ToString("X"));
```

to uzyskamy poniższy wynik:

```
Circle, Cylinder
Circle, Cylinder
3
Circle, Cylinder
00000003
```

Pokazana technika to elastyczny sposób wyświetlania flag zastosowanych dla typu wyliczeniowego.

## Zobacz również

- Tematy *Enum.ToString Method* oraz *Enumeration Format Strings* w dokumentacji MSDN.

# 1.17. Konwersja zwykłego tekstu na odpowiedniki w postaci wartości typu wyliczeniowego

## Problem

Mamy tekstową reprezentację wartości typu wyliczeniowego, na przykład z bazy danych bądź pliku. Tę wartość tekstową chcemy przekształcić na typ wyliczeniowy.

## Rozwiązanie

Aby przekształcić tekstową reprezentację elementu typu wyliczeniowego na typ wyliczeniowy, można skorzystać z metody Parse klasy Enum, na przykład:

```
try
{
    Language proj1Language = (Language)Enum.Parse(typeof(Language), "VBNET");
    Language proj2Language = (Language)Enum.Parse(typeof(Language), "UnDefined");
}
catch (ArgumentException e)
{
```

```
        // Obsługa nieprawidłowej wartości tekstowej,
        // (na przykład ciągu "UnDefined").
    }
```

przy czym typ wyliczeniowy `Language` zdefiniowano w sposób następujący:

```
enum Language
{
    Other = 0, CSharp = 1, VBNET = 2, VB6 = 3
}
```

## Analiza

Statyczna metoda `Enum.Parse` przekształca tekst na określoną wartość typu wyliczeniowego. Technika ta przydaje się w przypadku, kiedy wyświetlamy użytkownikowi listę wartości zdefiniowanych wewnątrz typu wyliczeniowego. Kiedy użytkownik wybierze element z tej listy, za pomocą metody `Enum.Parse` można z łatwością przekształcić wybrany tekst z reprezentacji tekstowej na odpowiednią wartość typu wyliczeniowego. Metoda zwraca obiekt, który następnie trzeba przekształcić na docelowy typ wyliczeniowy.

Oprócz przekazywania do metody `Enum.Parse` pojedynczej wartości typu wyliczeniowego w postaci tekstowej, można również przekazać do niej tę wartość w postaci liczbowej. Na przykład, przeanalizujmy następujący kod:

```
Language proj1Language = (Language)Enum.Parse(typeof(Language),"VBNET");
```

Instrukcję tę można zapisać również w następującej postaci:

```
Language proj1Language = (Language)Enum.Parse(typeof(Language),"2");
```

W tym przypadku założono, że wartość typu wyliczeniowego `Language.VBNET` jest równa 2. Inną interesującą własnością metody `Parse` jest możliwość przekazywania do niej listy nazw lub wartości elementów typu wyliczeniowego rozdzielonej przecinkami. W takim przypadku dla elementów typu wyliczeniowego zostanie wykonana logiczna operacja `OR`. W poniższym przykładzie utworzono typ wyliczeniowy i wykonano operację `OR` dla wartości `VBNET` i `CSharp`:

```
Language proj1Language = (Language)Enum.Parse(typeof(Language),"CSharp, VBNET");
```

Przed wykonaniem operacji OR dla każdego elementu typu wyliczeniowego są obcinane spacje, zatem dodanie spacji pomiędzy poszczególnymi elementami na liście nie ma znaczenia.

## Zobacz również

- Temat *Enum.Parse Method* w dokumentacji MSDN.

# 1.18. Sprawdzanie poprawności wartości typu wyliczeniowego

## Problem

W przypadku przekazywania wartości liczbowej do metody, która spodziewa się elementu typu wyliczeniowego, może się zdarzyć, że przekazana wartość nie istnieje w typie wyliczeniowym. Przed wykorzystaniem liczby jako argumentu należy przeprowadzić test, by sprawdzić, czy rzeczywiście jest jednym z elementów typu.

## Rozwiązanie

Aby zapobiec wystąpieniu tego problemu, można skontrolować wartość przekazywaną jako parametr za pomocą instrukcji `switch`.

Jako przykład przeanalizujmy typ wyliczeniowy `Language` o następującej definicji:

```
enum Language
{
    Other = 0, CSharp = 1, VBNET = 2, VB6 = 3
}
```

Załóżmy, że mamy metodę, która pobiera parametr typu wyliczeniowego `Language`, na przykład:

```
public void HandleEnum(Language language)
{
    // Wykorzystanie wartości typu wyliczeniowego...
}
```

Potrzebujemy metody zdefiniowania wartości numerycznych, które można przekazywać do metody `HandleEnum`. Oto jej kod:

```
public static bool CheckLanguageEnumValue(Language language)
{
    switch (language)
    {
        // Należy wymienić wszystkie prawidłowe wartości typu wyliczeniowego.
        // Oznacza to, że metoda będzie akceptowała tylko te wartości typu, które
        // wymienimy. Inne wartości będą nieprawidłowe.
        case Language.CSharp:
        case Language.Other:
        case Language.VB6:
        case Language.VBNET:
            break;
        default:
            Debug.Assert(false, language + " nie jest prawidłową wartością typu
                 wyliczeniowego akceptowaną przez procedurę.");
            return false;
    }
    return true;
}
```

## Analiza

Chociaż klasa `Enum` zawiera statyczną metodę `IsDefined`, nie należy jej używać. Metoda `IsDefined` wykorzystuje odbicia, co powoduje obniżenie wydajności. Obsługa wersji typu wyliczeniowego również nie jest najlepsza. Rozważmy sytuację, w której w kolejnej wersji naszej aplikacji dodajemy do typu wyliczeniowego `Language` wartość `MgdCpp` (Managed C++). Jeśli do sprawdzenia argumentów skorzystamy z metody `IsDefined`, wartość `MgdCpp` będzie uznana za prawidłową, ponieważ zdefiniowano ją w deklaracji typu wyliczeniowego pomimo tego, że kod, dla którego będziemy sprawdzali poprawność parametru, nie obsługuje tej wartości. Dzięki jawnemu wyszczególnieniu wartości w instrukcji `switch` użytej wewnątrz metody `CheckLanguageEnumValue` odrzucamy wartość `MgdCpp`, przez co eliminujemy sytuację działania kodu w nieprawidłowym kontekście, a oto przecież nam chodziło.

Sprawdzenie poprawności elementów typu wyliczeniowego należy zastosować zawsze wtedy, gdy metoda jest widoczna dla obiektów zewnętrznych. Obiekt zewnętrzny może wywoływać metody z widocznością publiczną. Dlatego wartości typu wyliczeniowego przekazane do metody powinny być sprawdzane przed ich wykorzystaniem.

Metody o widoczności `internal`, `protected` oraz `internal protected` mają znacznie węższy zasięg od metod publicznych, ale również dotyczą ich problemy właściwe dla tych pierwszych. Metody o widoczności `private` nie zawsze wymagają tego dodatkowego poziomu ochrony. Należy samodzielnie ocenić, czy warto zastosować metodę typu `CheckLanguageEnumValue` w celu sprawdzenia poprawności parametrów przekazywanych do metod prywatnych.

Metodę `HandleEnum` można wywoływać na kilka różnych sposobów. Oto trzy z nich:

```
HandleEnum(Language.CSharp)
HandleEnum((Language)1)
HandleEnum((Language)someVar)      // Gdzie zmienna someVar jest typu int
```

Każde z tych wywołań jest prawidłowe. Niestety, poniższe wywołania metody również są prawidłowe:

```
HandleEnum((Language)100)

int someVar = 100;
HandleEnum((Language)someVar)
```

Takie wywołania metody również skompilują się bez błędów, ale w przypadku, gdy wewnątrz metody `HandleEnum` nastąpi próba wykorzystania przekazanej do niej wartości (w tym przypadku liczby 100), kod będzie działał w sposób trudny do przewidzenia. W wielu przypadkach nie zostanie nawet zgłoszony wyjątek. Metoda `HandleEnum` otrzyma wartość 100 jako argument, tak jakby był prawidłowym elementem typu wyliczeniowego.

Aby zapobiec takiej sytuacji, można zastosować metodę `CheckLanguageEnumValue`, która sprawdza, czy do metody są przekazywane poprawne elementy typu wyliczeniowego. Poniżej zaprezentowano zmodyfikowaną wersję metody `HandleEnum`:

```
public void HandleEnum(Language language)
{
    if (CheckLanguageEnumValue(language))
    {
        // Wykorzystanie języka...
    }
    else
    {
        // Obsługa nieprawidłowej wartości języka...
    }
}
```

## Zobacz również

Sposób testowania prawidłowej wartości typu wyliczeniowego w przypadku oznaczenia typu atrybutem `Flags` opisano w recepturze 1.19.

# 1.19. Sprawdzanie poprawności typu wyliczeniowego z atrybutem Flags

## Problem

Trzeba sprawdzić, czy określona wartość jest poprawnym elementem typu wyliczeniowego bądź poprawną kombinacją elementów typu wyliczeniowego (na przykład flagami bitowymi poddanymi operacji `OR` w przypadku typu wyliczeniowego oznaczonego atrybutem `Flags`).

## Rozwiązanie

Aby można było kontrolować, czy określona wartość jest poprawnym elementem typu wyliczeniowego bądź pewną kombinacją elementów typu wyliczeniowego, należy wprowadzić element `All` do typu wyliczeniowego równy wszystkim elementom typu poddanym operacji `OR`. Następnie można skorzystać z metody `HandleFlagsEnum` do przeprowadzenia testu.

W przypadku typu wyliczeniowego oznaczonego atrybutem `Flags` występują problemy z zastosowaniem metody `Enum.IsDefined`. Rozważmy typ wyliczeniowy `Language` o następującej deklaracji:

```
[Flags]
enum Language
{
    CSharp = 1, VBNET = 2, VB6 = 4
}
```

Prawidłowe wartości typu `Language` to liczby `{1, 2, 3, 4, 5, 6, 7}`. Wartości `3`, `5`, `6` i `7` nie występują jednak jawnie w definicji typu. Wartość `3` uzyskujemy poprzez wykonanie operacji `OR` dla elementów `CSharp` i `VBNET`, z kolei wartość `7` uzyskamy, jeśli wszystkie elementy typu poddamy operacji `OR`. W przypadku wartości `3`, `5`, `6` i `7` wywołanie metody `Enum.IsDefined` zwróci `false`, co wskazuje na to, że nie są to prawidłowe wartości, podczas gdy w rzeczywistości są prawidłowe. Potrzebujemy sposobu sprawdzenia, czy do metody przekazano prawidłowy zbiór flag.

Aby rozwiązać problem, można wprowadzić nowy element typu wyliczeniowego `Language`, który definiuje wszystkie wartości prawidłowe. W naszym przypadku deklarację typu wyliczeniowego `Language` należy zapisać w następujący sposób:

```
[Flags]
enum Language
{
    CSharp = 1, VBNET = 2, VB6 = 4,
    All = (CSharp | VBNET | VB6)
}
```

Nowy element typu wyliczeniowego `All` jest równy wszystkim pozostałym elementom typu poddanym operacji `OR`. Aby teraz sprawdzić poprawność flag typu `Language`, można wykorzystać metodę w następującej postaci:

```
public bool HandleFlagsEnum(Language language)
{
    if ((language != 0) && ((language & Language.All) == language))
    {
        return (true);
    }
    else
    {
        return (false);
    }
}
```

## Analiza

Aby skorzystać z metody `HandleFlagsEnum` w przypadku definicji typu `Language` z atrybutem `Flags`, wystarczy dodać element `All`. Powinien on być równy wszystkim elementom typu poddanym operacji `OR`.

Metoda `HandleFlagsEnum` wykorzystuje element `All` do sprawdzenia, czy element typu wyliczeniowego jest prawidłowy. Aby to zrobić, wykonuje operację `AND` wartości parametru `language` z elementem `Language.All`, a następnie sprawdza, czy wynik odpowiada oryginalnej wartości parametru `language`.

W celu obsługi typu wyliczeniowego w pokazanej postaci można również przeciążyć metodę `HandleFlagsEnum` (w tym przypadku zdefiniowano metodę dla typu `integer` zamiast typu wyliczeniowego `Language`). Poniższy kod kontroluje, czy zmienna typu `integer` zawiera poprawną wartość elementu typu wyliczeniowego `Language`:

```
public static bool HandleFlagsEnum(int language)
{
    if ((language!=0) && ((language & (int)Language.All) == language))
    {
         return (true);
    }
    else
    {
        return (false);
    }
}
```

Przeciążona wersja metody `HandleFlagsEnum` zwraca `true` w przypadku, gdy parametr `language` jest prawidłowy, i `false` w przeciwnym razie.

## Zobacz również

Sposób testowania prawidłowej wartości typu wyliczeniowego bez atrybutu `Flags` opisano w recepturze 1.18.

# 1.20. Zastosowanie elementów typu wyliczeniowego w masce bitowej

## Problem

Chcemy, aby elementy typu wyliczeniowego były interpretowane jako flagi bitowe, które można wykorzystać do tworzenia kombinacji wartości (flag).

## Rozwiązanie

Należy oznaczyć typ wyliczeniowy za pomocą atrybutu `Flags`:

```
[Flags]
enum Language
{
    CSharp = 0x0001, VBNET = 0x0002, VB6 = 0x0004, Cpp = 0x0008
}
```

Aby skorzystać z kombinacji elementów tego typu wyliczeniowego, wystarczy zastosować dla nich bitową operację `OR` (`|`), na przykład:

```
Language lang = Language.CSharp | Language.VBNET;
```

## Analiza

Dodanie atrybutu `Flags` do deklaracji typu wyliczeniowego powoduje, że określone elementy typu są traktowane jako indywidualne flagi bitowe, dla których można wykonywać bitową operację `OR`. Zastosowanie typu wyliczeniowego oznaczonego atrybutem `Flags` niczym nie różni się od stosowania standardowego typu wyliczeniowego. Należy zwrócić uwagę, że brak oznaczenia typu wyliczeniowego atrybutem `Flags` nie spowoduje zgłoszenia wyjątku ani powstania błędu kompilacji nawet wtedy, gdy elementy typu wyliczeniowego będą wykorzystane jako flagi bitowe.

Dodanie atrybutu `Flags` daje dwie korzyści. Po pierwsze, w przypadku oznaczenia typu wyliczeniowego atrybutem `Flags` metody `ToString` oraz `ToString("G")` zwracają ciąg znaków składający się z nazw stałych rozdzielonych przecinkami. W innym razie te dwie metody zwracają liczbową reprezentację elementu typu wyliczeniowego. Należy zwrócić uwagę, że metoda `ToString("F")` zwraca ciąg znaków składający się z nazw stałych rozdzielonych przecinkami, niezależnie od tego, czy określony typ wyliczeniowy oznaczono atrybutem `Flags` czy nie. Aby uzyskać wskazówkę dotyczącą sposobu działania tej metody, można zapoznać się z opisem typu formatowania `"F"` w tabeli 1.2 zamieszczonej w recepturze 1.16.

Druga korzyść polega na tym, że w przypadku natknięcia się na typ wyliczeniowy w analizowanym kodzie, można łatwiej ocenić, do czego programista chciał go wykorzystać. Jeśli jawnie zdefiniował go z atrybutem `Flags`, możemy stosować jego wartości jako maski bitowe.

Typ wyliczeniowy oznaczony atrybutem `Flags` można interpretować jako wartość pojedynczą lub jako kilka wartości połączonych w jedną. Jeśli chcemy jednocześnie przekazać kilka języków, możemy wykorzystać następujący kod:

```
Language lang = Language.CSharp | Language.VBNET;
```

Zmienna `lang` jest teraz równa bitowym wartościom dwóch elementów typu wyliczeniowego po poddaniu ich operacji `OR`. W wyniku wykonania tej operacji uzyskamy wartość 3, tak jak pokazano poniżej:

```
Language.CSharp    0001
Language.VBNET.....0010
Po wykonaniu OR    0011
```

Elementy typu wyliczeniowego przekształcono na liczby dwójkowe, a następnie wykonano operację `OR`. W jej wyniku uzyskano wartość 0011, czyli 3 w systemie dziesiętnym. Dla kompilatora ta wartość to zarówno dwa elementy typu wyliczeniowego po wykonaniu operacji `OR` (`Language.CSharp | Language.VBNET`), jak pojedyncza wartość 3.

Aby sprawdzić, czy w zmiennej typu wyliczeniowego włączono określoną flagę, można skorzystać z bitowego operatora `AND` (`&`) w następujący sposób:

```
Language lang = Language.CSharp | Language.VBNET;

if((lang & Language.CSharp) == Language.CSharp)
    Console.WriteLine("Zmienna zawiera flagę odpowiadającą językowi C#");
else
    Console.WriteLine("Zmienna nie zawiera flagi odpowiadającej językowi C#");
```

Wykonanie powyższego kodu spowoduje wyświetlenie tekstu *Zmienna zawiera flagę odpowiadającą językowi C#*. Wykonanie operacji `AND` dla zmiennej `lang` i elementu `Language.CSharp` zwróci zero, jeśli w zmiennej `lang` nie włączono flagi odpowiadającej językowi C#, lub wartość

`Language.CSharp` w przypadku, gdy w zmiennej `lang` włączono tę flagę. Wykonanie operacji `AND` dla dwójkowych reprezentacji obu wartości można zapisać w sposób następujący:

```
Language.CSharp | Language.VBNET    0011
Language.CSharp                     0001
Wynik operacji AND                  0001
```

Dokładniej problem ten opisano w recepturze 1.21.

W niektórych przypadkach definicja typu wyliczeniowego rozrasta się do sporych rozmiarów. W przypadku typu `Language` można wprowadzić odpowiedniki wielu różnych języków, tak jak pokazano poniżej:

```
[Flags]
enum Language
{
    CSharp = 0x0001, VBNET = 0x0002, VB6 = 0x0004, Cpp = 0x0008,
    FortranNET = 0x0010, JSharp = 0x0020, MSIL = 0x0080
}
```

Aby utworzyć wartość typu wyliczeniowego `Language`, która reprezentuje wszystkie języki, trzeba by wykonać operację `OR` dla wszystkich wartości typu:

```
Language lang = Language.CSharp | Language.VBNET | Language.VB6;
```

Zamiast tego można dodać nową wartość `All` do typu wyliczeniowego obejmującą wszystkie języki. Można to zrobić w następujący sposób:

```
[Flags]
enum Language
{
    CSharp = 0x0001, VBNET = 0x0002, VB6 = 0x0004, Cpp = 0x0008,
    FortranNET = 0x0010, JSharp = 0x0020, MSIL = 0x0080,
    All = (CSharp | VBNET | VB6 |  Cpp | FortranNET | JSharp | MSIL)
}
```

Mamy teraz nową wartość typu wyliczeniowego — `All`, która obejmuje wszystkie jego wartości. Zwróćmy uwagę na to, że istnieją dwa sposoby reprezentacji wszystkich wartości typu wyliczeniowego. Druga z nich jest znacznie bardziej czytelna. Niezależnie od wybranej metody w przypadku dodania lub usunięcia elementów typu wyliczeniowego trzeba odpowiednio zmodyfikować wartość reprezentującą wszystkie wartości.

W podobny sposób można wprowadzić wartości odpowiadające określonym podzbiorom wartości typu wyliczeniowego, na przykład:

```
[Flags]
enum Language
{
    CSharp = 0x0001, VBNET = 0x0002, VB6 = 0x0004, Cpp = 0x0008,
    CobolNET = 0x000F, FortranNET = 0x0010, JSharp = 0x0020,
    MSIL = 0x0080,
    All = (CSharp | VBNET | VB6 | Cpp | FortranNET | JSharp | MSIL),
    VBOnly = (VBNET | VB6),
    NonVB = (CSharp | Cpp | FortranNET | JSharp | MSIL)
}
```

Mamy teraz dwa dodatkowe elementy typu wyliczeniowego: jeden odpowiadający wyłącznie językom rodziny Visual Basic (`Language.VBNET` oraz `Language.VB6`) i drugi, który odpowiada innym językom niż VB).

# 1.21. Sprawdzanie, czy ustawiono jedną czy kilka flag w danych typu wyliczeniowego

## Problem

Trzeba sprawdzić, czy w zmiennej typu wyliczeniowego składającej się z flag bitowych włączono jedną lub kilka określonych flag. Na przykład dla następującej deklaracji typu wyliczeniowego:

```
[Flags]
enum language
{
    CSharp = 0x0001, VBNET = 0x0002, VB6 = 0x0004, Cpp = 0x0008
}
```

trzeba sprawdzić, używając logiki Boole'a, czy w zmiennej `lang` zdefiniowanej poniżej włączono flagi `Language.CSharp` i (lub) `Language.Cpp`:

```
Language lang = Language.CSharp | Language.VBNET;
```

## Rozwiązanie

Aby sprawdzić, czy w zmiennej włączono określoną flagę bitową, można wykorzystać następującą instrukcję warunkową:

```
if((lang & Language.CSharp) == Language.CSharp)
{
    // W zmiennej lang włączono co najmniej wartość Language.CSharp.
}
```

Aby sprawdzić, czy w zmiennej włączono wyłącznie określoną flagę bitową, można wykorzystać następującą instrukcję warunkową:

```
if(lang == Language.CSharp)
{
    // W zmiennej lang włączono tylko wartość Language.CSharp.
}
```

Aby sprawdzić, czy w zmiennej włączono zbiór wszystkich flag bitowych, można skorzystać z następującej instrukcji warunkowej:

```
if((lang & (Language.CSharp | Language.VBNET)) ==
    (Language.CSharp | Language.VBNET))
{
    // W zmiennej lang włączono co najmniej elementy Language.CSharp oraz Language.VBNET.
}
```

Aby sprawdzić, czy w zmiennej włączono jedynie zbiór określonych flag bitowych, można skorzystać z następującej instrukcji warunkowej:

```
if((lang | (Language.CSharp | Language.VBNET)) ==
    (Language.CSharp | Language.VBNET))
{
    // W zmiennej lang włączono jedynie elementy Language.CSharp oraz Language.VBNET.
}
```

## Analiza

W przypadku typów wyliczeniowych zdefiniowanych jako flagi bitowe z wykorzystaniem atrybutu `Flags` zazwyczaj występuje potrzeba wykonania określonych instrukcji warunkowych. Zwykle wykorzystuje się w nich operatory bitowe `AND` (`&`) oraz `OR` (`|`).

Sprawdzenie, czy w zmiennej ustawiono określone flagi bitowe, można wykonać za pomocą następującej instrukcji warunkowej:

```
if((lang & Language.CSharp) == Language.CSharp)
```

gdzie `lang` jest zmienną typu wyliczeniowego `Language`.

Operator `&` w połączeniu z maską bitową wykorzystuje się w celu stwierdzenia, czy określony bit ustawiono na jedynkę. Wynik operacji `AND` dwóch bitów jest jedynką tylko wtedy, gdy oba bity mają wartość 1. W innym przypadku wynik jest zerem. Operację można wykorzystać do sprawdzenia, czy w liczbie zawierającej określone flagi bitowe ustawiono określone bity. Jeśli wykonamy operację `AND` zmiennej `lang` z określoną flagą bitową (w tym przypadku `Language.CSharp`), możemy wyodrębnić tę flagę bitową. Wyrażenie (`lang & Language.CSharp`) w przypadku, gdy zmienna `lang` jest równa wartości `Language.CSharp`, można rozwiązać w sposób następujący:

```
Language.CSharp      0001
lang                 0001
Wynik operacji AND   0001
```

Jeśli zmienna `lang` jest równa innej wartości, na przykład `Language.VBNET`, wyrażenie można rozwiązać w następujący sposób:

```
Language.CSharp      0001
lang                 0010
Wynik operacji AND   0000
```

Zwróćmy uwagę, że wykonanie operacji `AND` w pierwszym przypadku zwraca wartość `Language.CSharp`, natomiast w drugim wartość `0x0000`. Porównanie wyniku operacji z poszukiwaną wartością (`Language.CSharp`) pozwala stwierdzić, czy określony bit włączono.

Zaprezentowana metoda doskonale nadaje się do sprawdzania wybranych bitów. Co jednak zrobić, by się dowiedzieć, czy tylko jeden wskazany bit jest włączony (a wszystkie pozostałe wyłączone) lub wyłączony (a wszystkie pozostałe włączone)? Aby sprawdzić, czy w zmiennej `lang` jest włączony wyłącznie bit `Language.CSharp`, można wykorzystać następującą instrukcję warunkową:

```
if(lang == Language.CSharp)
```

Rozważmy sytuację, w której w zmiennej `lang` włączono wyłącznie wartość `Language.CSharp`. Wyrażenie wykorzystujące operator `OR` będzie miało następującą postać:

```
lang = Language.CSharp;
if ((lang != 0) && (Language.CSharp == (lang | Language.CSharp)))
{
    // Dzięki zastosowaniu operacji OR znaleziono wartość CSharp
}

Language.CSharp      0001
lang                 0001
Wynik operacji OR    0001
```

Spróbujmy teraz włączyć jedną lub dwie wartości w zmiennej `lang` i wykonać dla niej tę samą operację:

```
lang = Language.CSharp | Language.VB6 | Language.Cpp;
if ((lang != 0) && (Language.CSharp == (lang | Language.CSharp)))
{
    // Dzięki zastosowaniu operacji OR znaleziono wartość CSharp
}

Language.CSharp      0001
lang                 1101
Wynik operacji OR    1101
```

Wynikiem pierwszego wyrażenia jest ta sama wartość, z którą sprawdzamy zmienną. Wynikiem drugiego wyrażenia jest znacznie większa wartość od elementu `Language.CSharp`. Wskazuje to na fakt, że w zmiennej `lang` w pierwszym wyrażeniu włączono wyłącznie wartość `Language.CSharp`, podczas gdy w drugim wyrażeniu oprócz `Language.CSharp` włączono również wartości odpowiadające innym językom (a być może wcale nie włączono wartości `Language.CSharp`).

Za pomocą wersji formuły z operacją `OR` można testować wiele bitów w celu stwierdzenia, czy są włączone, a wszystkie pozostałe wyłączone. Można to zrobić za pomocą następującej instrukcji warunkowej:

```
if ((lang !=0) && ((lang | (Language.CSharp | Language.VBNET))==
    (Language.Csharp | Language.VBNET)))
```

Zwróćmy uwagę, że w celu przeprowadzenia testu dla więcej niż jednego języka wystarczy poddać operacji `OR` wartości tych języków. Zmieniając pierwszy operator `|` na `&`, można stwierdzić, czy włączone są co najmniej te bity, które sprawdzamy. Oto odpowiednia instrukcja warunkowa:

```
if ((lang !=0) && ((lang & (Language.CSharp | Language.VBNET))==
    (Language.CSharp | Language.VBNET)))
```

W przypadku przeprowadzania testu dla wielu elementów typu wyliczeniowego warto dodać element, którego wartość jest równa wynikowi operacji `OR` wszystkich elementów, które chcemy testować. Na przykład w przypadku testu dla wszystkich języków oprócz `Language.CSharp` potrzebna instrukcja warunkowa mocno by się rozrosła. Aby temu zapobiec, można dodać element typu wyliczeniowego `Language`, którego wartość jest równa wynikowi operacji `OR` dla wszystkich języków oprócz `Language.CSharp`. Deklaracja typu wyliczeniowego w zmodyfikowanej formie ma teraz następującą postać:

```
[Flags]
enum Language
{
    CSharp = 0x0001, VBNET = 0x0002, VB6 = 0x0004, Cpp = 0x0008,
    AllLanguagesExceptCSharp = VBNET | VB6 | Cpp
}
```

natomiast instrukcja warunkowa będzie miała postać:

```
if ((lang !=0) && ((lang | (Language.AllLanguagesExceptCSharp)==
    (Language.AllLanguagesExceptCsharp)
```

Instrukcja jest znacznie bardziej zwięzła, łatwiejsza do poprawiania i analizy.

Aby sprawdzić, czy jeden lub więcej bitów ustawiono na jedynkę, należy zastosować operator AND.

Aby skontrolować, czy jeden lub więcej bitów ustawiono na zero, należy wykorzystać operator OR.

# 1.22. Wyznaczanie części całkowitej zmiennej typu decimal lub double

## Problem

Chcemy znaleźć część całkowitą liczby typu `decimal` lub `double`.

## Rozwiązanie

Część całkowitą liczby typu `decimal` bądź `double` można wyznaczyć poprzez obcięcie jej do liczby całkowitej bliższej zeru. W tym celu można skorzystać z przeciążonej wersji statycznej metody `System.Math.Truncate`, która pobiera jako argument wartość typu `decimal` bądź `double` i zwraca ten sam typ:

```
decimal pi = (decimal)System.Math.PI;
decimal decRet = System.Math.Truncate(pi); // decRet = 3

double trouble = 5.555;
double dblRet = System.Math.Truncate(trouble);
```

## Analiza

Metodę `Truncate` wprowadzono w wersji 2.0 platformy .NET w celu usprawnienia możliwości zaokrąglania liczb. Efekt uboczny metody `Truncate` polega na odrzucaniu części ułamkowej liczby i zwracaniu części całkowitej. Kiedy liczby zmiennoprzecinkowe przekroczą określony rozmiar, nie mają części ułamkowej, a jedynie przybliżenie części całkowitej.

## Zobacz również

- Receptura 1.9.
- Temat *System.Math.Truncate Method* w dokumentacji MSDN.