

**Zawartość:**

- Enum
- Decimal vs Double/Float
- String
- Typy referencyjne i wartościowe
- Kopia i kopia głęboka

## Enum – typ wyliczeniowy

Sama nazwa typ wyliczeniowy mówi bardzo niewiele. Enum to definicja, grupy wartości:

```
1  enum Direction{
2      Left = 0,
3      Right = 1,
4      Bottom = 2,
5      Top = 3,
6  }
7
8  function moveObject(steps: number, dir: Direction): void{
9      //some logic
10 }
11
12 moveObject(2, Direction.Left);
```

Co nam to dało??? Ograniczyliśmy drugi parametr do konkretnych czterech wartości i do tego mamy je nazwane. Nie da się wpisać w drugi parametr wartości z poza enuma. Przykład z C/C++ wyjaśni więcej:

```
1  enum Direction{
2      Left = 0,
3      Right = 1,
4      Bottom = 2,
5      Top = 3,
6  };
7
8  void moveObject2(int step, Direction dir){
9      if(dir == Left){
10         //left logic
11     }
12     else if(dir == Right){
13         //right logic
14     }
15     else if(dir == Bottom){
16         //bottom logic
17     }
18     else if(dir == Top){
19         //top logic
20     }
21 }
22
23 void moveObject1(int step, int dir){
24     if(dir == 0){
25         //left logic
26     }
27     else if(dir == 1){
28         //right logic
29     }
30     else if(dir == 2){
31         //bottom logic
32     }
33     else if(dir == 3){
34         //top logic
35     }
36 }
37
```

Powyżej widać dwa podejścia. Stare bardzo proste podejście przyjmujące, że konkretna liczba oznacza jakiś kierunek oraz podejście bazujące na enum. Jaka jest główna wada podejścia bazującego na intach? Ano taka, że ktoś może w parametrze podać wartość 4 albo 2mln i nic się nie stanie. W przypadku enuma, podanie złej wartości skończy się błędem kompilacji. Ogarnięte środowiska programowania, dzięki wykorzystaniu enuma, będą wiedziały co nam podpowiedzieć w momencie wywołania funkcji.

```
new Font("Microsoft Sans Serif", 16.0f, FontStyle.Regular);
Font("Mi
new Fc
Font("Mi
FontType.Line;
FontDashStyle.Dot;
```

Font.Font(string familyName, float emSize, FontStyle style) (+ 12 overloads)  
Initializes a new Font using a specified size and style.

Exceptions:  
ArgumentException

Przykład z C#. Klasa Font w konstruktorze przyjmuje trzy wartości, gdzie trzecia to enum, oznaczający styl czcionki. Autor klasy przewidział enuma, zawierającego wszystkie dostępne rodzaje stylów:

```
public enum FontStyle
{
    //
    // Summary:
    //     Normal text.
    Regular = 0x0,
    //
    // Summary:
    //     Bold text.
    Bold = 0x1,
    //
    // Summary:
    //     Italic text.
    Italic = 0x2,
    //
    // Summary:
    //     Underlined text.
    Underline = 0x4,
    //
    // Summary:
    //     Text with a line through the middle.
    Strikeout = 0x8
}
```

W pythonie enum to klasa dziedzicząca z klasy Enum:

```
from enum import Enum
class Color(Enum):
    RED = 1
    GREEN = 2
    BLUE = 3
```

## DECIMAL vs DOUBLE

Typy double oraz float są bardzo podstawowe i bazują na standardzie IEEE 754. System ten jest bardzo niedoskonały i nie radzi sobie z pewnymi liczbami np. 0.9 może zostać zapisane jako 0.900000001 albo 0.8999999991. **DLATEGO NIGDY NIE IMPLEMENTUJE SIĘ NA NICH OPERACJI NA PIENIĄŻKACH 😊**

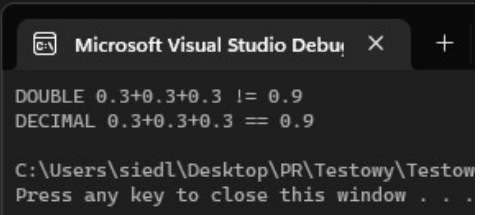
W C# istnieje coś takiego jak typ decimal, który jest wysoko poziomą implementacją liczby zmiennie przecinkowych:

```
double a = 0.3;
double b = 0.3;
double c = 0.3;

if(a+b+c == 0.9)
{
    Console.WriteLine("DOUBLE 0.3+0.3+0.3 == 0.9");
}
else
{
    Console.WriteLine("DOUBLE 0.3+0.3+0.3 != 0.9");
}

decimal d = 0.3M;
decimal e = 0.3M;
decimal f = 0.3M;

if (d + e + f == 0.9M)
{
    Console.WriteLine("DECIMAL 0.3+0.3+0.3 == 0.9");
}
else
{
    Console.WriteLine("DECIMAL 0.3+0.3+0.3 != 0.9");
}
```



Decimal to zwykła klasa, które oferuje różne możliwości, różne konstruktory budujące liczbę decimal na bazie stringa/doubla/floata.

W C/C++ istnieje wiele implementacji typu decimal. W innych językach wyższego poziomu powyższym problemem nie trzeba się przejmować bo liczby double/float są zaimplementowane tak by zapis był poprawny.

Co mam na myśli mówiąc wysoko poziomą implementacją??? Wyobraźmy sobie, że chcemy zapisać bardzo dużą liczbę w zmiennej np. 30 cyfrowego inta. No, żaden typ liczbowy tego nie pomieści. Co można zrobić? Można napisać klasę, która przyjmuje tą liczbę w konstruktorze jako napis, a później rozбивa na kilka intów, które są mnożone \*10000 itp.

## String

String to inaczej łańcuch znaków. Nazwa ta nie wzięła się z niczego. Pierwotnie w języku C stringi były po prostu tablicami znaków:

```
9  int main(){
10     char napis[] = { 'N','A','P','I','S', 0};
11     std::cout<<napis<<std::endl;
12
13     return 0;
14 }
15
```

PROBLEMS OUTPUT DEBUG CONSOLE **TERMINAL** JUPYTER ...

```
PS C:\Users\siedl\Desktop\Korki> .\main.exe
NAPIS
PS C:\Users\siedl\Desktop\Korki> |
```

0 na końcu sygnalizuje językowi koniec stringa. Brak 0 na końcu spowoduje, że język poleci dalej po pamięci do napotkania pierwszego 0. Z tego często biorą się krzaczkę na ekranie, ponieważ, program próbuje interpretować napotkane dane w pamięci jako napis.

```
9  int main(){
10     char napis[] = { 'N','A','P','I','S'};
11     std::cout<<napis<<std::endl;
12
13     return 0;
14 }
15
```

PROBLEMS OUTPUT DEBUG CONSOLE **TERMINAL** JUPYTER ... powershell

```
PS C:\Users\siedl\Desktop\Korki> .\main.exe
NAPISd, v||E
PS C:\Users\siedl\Desktop\Korki> |
```

W późniejszym czasie w C++ dodali `std::string`, który jest abstrakcyjną nakładką na powyższego stringa. W językach wyższego poziomu stringi też są zaimplementowane wysoko poziomowo, ale jakby się w te implementacje zagłębić to prędzej czy później trafimy na tablice znaków.

```
// Represents text as a sequence of UTF-16
[DefaultMember("Chars")]
public sealed class String : IEnumerable<char>,
{
    //
```

W C# klasa `String` dziedziczy z `IEnumerable<Char>`. Jest to typ definiujący kolekcję znaków.

## Typy referencyjne oraz typy wartościowe:

W językach wyższego poziomu takich jak Python, C#, Java typy zmiennych można podzielić na powyższe dwie kategorie. Jest to bardzo ważne zagadnienie.

Typ wartościowy – jeżeli zmienna jest typu wartościowego oznacza to, że przechowuje ona dane bezpośrednio w sobie. Wartości typów wartościowych zapisywane są na stosie chyba, że jest częścią typu referencyjnego.

Typ referencyjny – jeżeli zmienna jest typu referencyjnego oznacza to, że wartość zapisana jest/będzie gdzieś w pamięci (na stercie) a sama zmienna przechowuje referencje (uproszczony wskaźnik) na to miejsce. Wartości typów referencyjnych zapisywane są na stercie (heap).

Przykłady z C#:

Typy wartościowe – bool, int, byte, char, double, enum, long, **typy powstałe na strukturach**

Typy referencyjne – delegaty, tablice, kolekcje, **typy powstałe na klasach**

```
0 references
internal class Program
{
    0 references
    static void Main(string[] args)
    {
        int a = 10; //typ wartościowy
        double b = 1.1; //typ wartościowy

        String asd = "ASD"; //typ referencyjny
        List<int> myList = new List<int>(); //typ referencyjny
    }
}
```

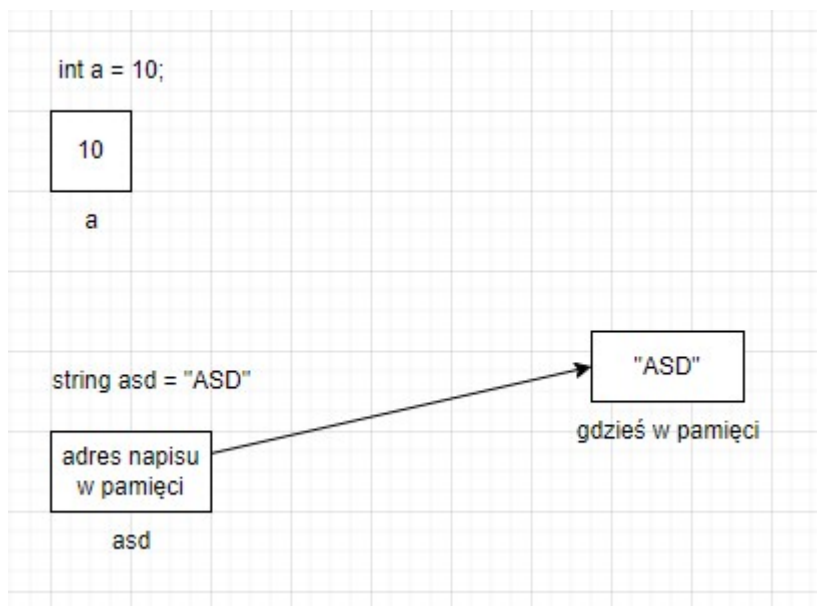
Definicja własnych typów:

```
//typ prosty
3 references
struct ASD_S
{
    public int a;
}

//typ referencyjny
3 references
class ASD_C
{
    public int a;
}
```

Struktura to typ wartościowy a klasa to typ referencyjny.

Tak to wygląda mniej więcej:



Co nam to daje??? Spróbujmy zwiększyć wartość a w powyższej klasie i strukturze za pomocą funkcji:

```
1 reference
static void InkrementujStruct(ASD_S target, int ile)
{
    target.a += ile;
}

1 reference
static void InkrementujClass(ASD_C target, int ile)
{
    target.a += ile;
}
```

```
ASD_S struct_instance = new ASD_S();
struct_instance.a = 10;
InkrementujStruct(struct_instance, 2);
Console.WriteLine(struct_instance.a);

ASD_C class_instance = new ASD_C();
class_instance.a = 10;
InkrementujClass(class_instance, 2);
Console.WriteLine(class_instance.a);
```

Podając do funkcji parametr (x\_instance), funkcja tworzy sobie jego kopię i na nim pracuje. W teorii oryginał powinien zostać bez zmian. W praktyce pierwszy WriteLine wypisze 10 a drugi 12. Dlaczego??? Zgodnie z schematem wyżej funkcja zrobiła sobie kopie struct\_instance do zmiennej target i zwiększyła a w zmiennej target. Tutaj wszystko jest proste i naturalne. W drugim przypadku również została wykonana kopia, ale jest to kopia



adresu. Mamy więc dwie zmienne z adresem wskazującym ten sam cel, który jest modyfikowany. Dlatego bez użycia zwracania z funkcji widzimy zmiany po powrocie z funkcji.

```
1 reference
static void Inkrementuj(int a, int ile)
{
    a += ile;
}

1 reference
static void InkrementujList(List<int> lista, int ile)
{
    for(int i = 0; i < lista.Count; i++)
    {
        lista[i] += ile;
    }
}
```

Int jest typem wartościowym, Lista jest typem referencyjnym.

```
int a = 10;
Inkrementuj(a, 2);
Console.WriteLine(a);

List<int> list = new List<int>() { 1, 2, 3, 4 };
InkrementujList(list, 2);
Console.WriteLine(list[0]);
```

Pierwszy WriteLine wypisze oczywiście 10. Drugi wypisze 3.

**Typ wartościowy staje się typem referencyjnym jeżeli jest częścią typu referencyjnego:**

```
//typ referencyjny
3 references
class ASD_C
{
    public int a;
}
```

Ten int mimo, że jest typem wartościowym staje się typem referencyjnym ponieważ stanowi część typu referencyjnego.

Analogiczna sytuacja jest w Pythonie czy Javie. W C/C++ nie ma czegoś takiego. W momencie tworzenia zmiennej decydujemy gdzie będzie zapisana. Jeżeli chcemy uzyskać efekt referencji opisany wyżej należy posłużyć się wskaźnikami lub referencjami &.

Generalnie jak da się coś obsłużyć typem wartościowym to dobrze jest go użyć bo jest szybszy. Typy wartościowe to zmienne lokalne więc znikają w momencie wyjścia np. z funkcji. Typy referencyjne są alokowane globalnie i są usuwane z pamięci przez Garbage Collector gdy ten zobaczy, że są już nie używane. Proces ten kosztuje nas wydajność.



## Kopia oraz kopia głęboka

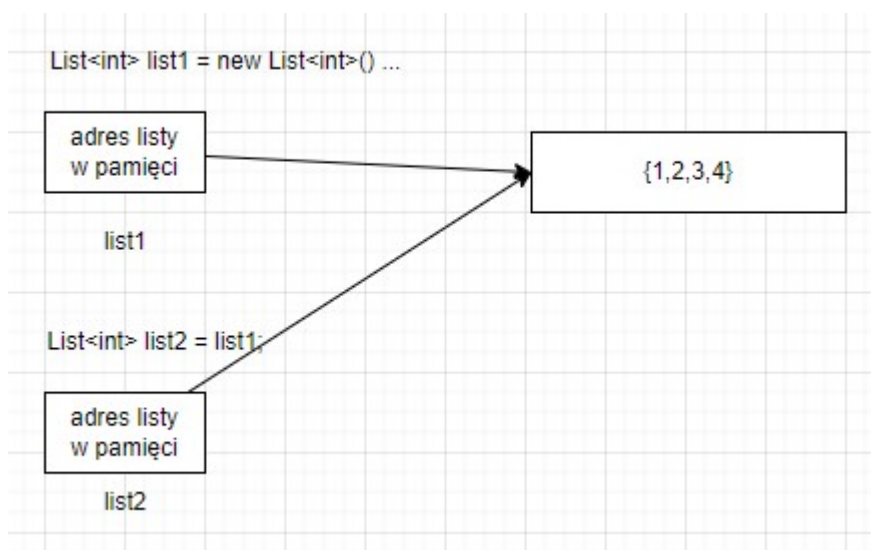
Temat ten nawiązuje do opisanego wyżej tematu typów referencyjnych i wartościowych.

```
int zmienna1 = 2;  
int zmienna2 = zmienna1;
```

Powyżej została wykonana kopia typu wartościowego. Oba inty mają tę samą wartość, ale są to dwie osobne liczby w pamięci.

```
List<int> list1 = new List<int>() { 1, 2, 3, 4 };  
List<int> list2 = list1;
```

Tutaj sytuacja jest inna. Mamy gdzieś w pamięci listę intów oraz zmienną z adresem (referencją) na tę listę. Wykonując standardową kopię jak wyżej wykonujemy kopię adresu a nie listy. Mamy więc tak naprawdę dwie zmienne z tym samym adresem, który oczywiście wskazuje na to samo miejsce. Jeżeli zwiększymy o 1 wartość list1[0] to list2[0] również się zwiększy bo obie te zmienne (list1 oraz list2) wskazują na to samo miejsce.



Jak zrobić prawdziwą kopię? Trzeba zaalokować drugą listę i przepisać wartości do niej.

```
ASD_C b = new ASD_C();  
ASD_C c = b;
```

Tutaj sytuacja jest analogiczna. ASD\_C jest klasą a więc typem referencyjnym. Aby zrobić kopię należy stworzyć drugi obiekt i po prostu przepisać pola.

```
ASD_S e = new ASD_S();  
ASD_S g = e;
```

ASD\_S jest strukturą, a więc typem wartościowym. Tutaj kopia zadziała normalnie tak jak w przypadku inta.

### KOPIA GŁĘBOKA

Bardzo często sytuacja wygląda tak, że mamy typ referencyjny. A w nim wartości również typu referencyjnego. A w tych wartościach również mogą być wartości referencyjne.

Przykład to lista obiektów jakiejś klasy.

```
List<ASD_C> m = new List<ASD_C>();
```

Lista jest typem referencyjnym. Tworzenie kopi polega na stworzeniu nowej i przepisaniu elementów. Ale tutaj mamy haczyk. Każdy element też jest typem referencyjnym. Więc zrobienie kopi tylko listy spowoduje, że elementy będą tymi samymi adresami na ten samo obiekt. Tutaj trzeba też odpowiednio skopiować każdy obiekt.

Sytuacja taka może zachodzić w głąb bardzo daleko stąd też nazwa głęboka kopia.