

**Zawartość:**

- typ wartościowy i referencyjny
- jak rozpoznać co jest jakim typem
- kopie typów referencyjnych
- przekazywanie do funkcji parametrów wartościowych i referencyjnych
- przekazywanie do funkcji typów wartościowych przez referencję

## O co chodzi z tymi typami wartościowymi i referencyjnymi?

Każdy zdaje sobie sprawę co to jest zmienna? No właśnie nie do końca. Zmienna to w teorii pojemnik na dane, ale jakie? W praktyce zmienna może przechowywać jedną z dwóch rzeczy:

- Właściwą wartość np. liczbę 564, znak 'A', wartość logiczną true/false itp.
- Wskaźnik/Referencję na wartość znajdującą się gdzieś w pamięci

Jeżeli zmienna znajduje się w drugiej kategorii nazywamy ją zmienną referencyjną. To czy zmienna jest referencyjna czy wartościowa zależy od typu zmiennej i typu przypisanej do niej wartości.

W językach takich jak C#/Java/Python jest to bardzo istotny temat. Bardzo często osoby mniej doświadczone mogą się całkiem nieźle zdziwić, że ich kod działa inaczej niż to sobie wyobrażali.

Tak prezentuje się lista typów wartościowych. Jeżeli utworzymy zmienną typu znajdującego się na poniższej liście zmienna ta będzie w sobie przechowywać bezpośrednią wartość.

- bool
- byte
- char
- decimal
- double
- enum
- float
- int
- long
- sbyte
- short
- struct
- uint
- ulong
- ushort
- typy utworzone ze struktur

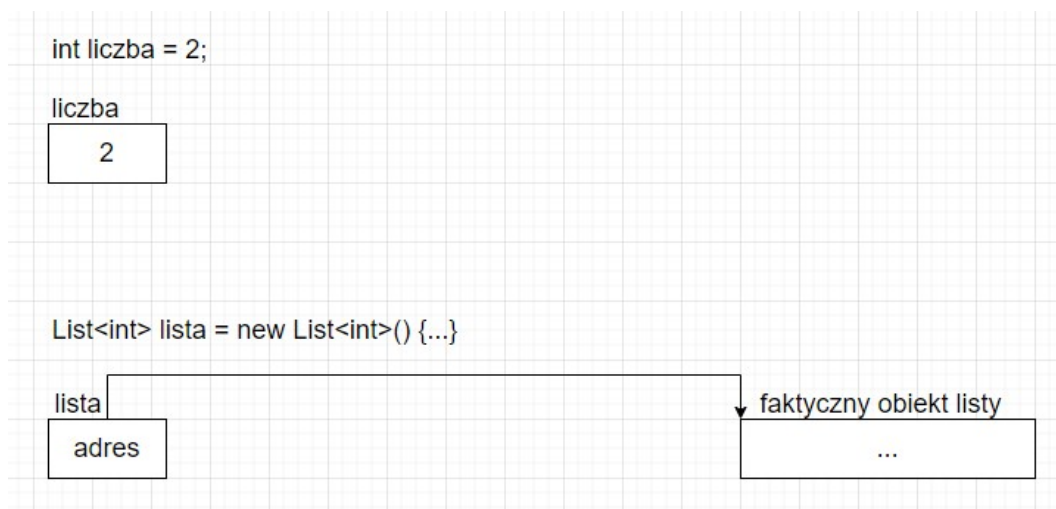
Każdy inny typ zmiennej w tym typy własne utworzone z klas są typami referencyjnymi.

Zobaczmy następujący kod:

```
0 references
static void Main(string[] args)
{
    int liczba = 2;

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

Widzimy tutaj dwie zmienne. Pierwsza typu „int” przechowuje w sobie liczbę 2. Druga typu „List<int>” przechowuje w sobie adres/referencje/wskaźnik na miejsce w pamięci gdzie znajduje się lista.



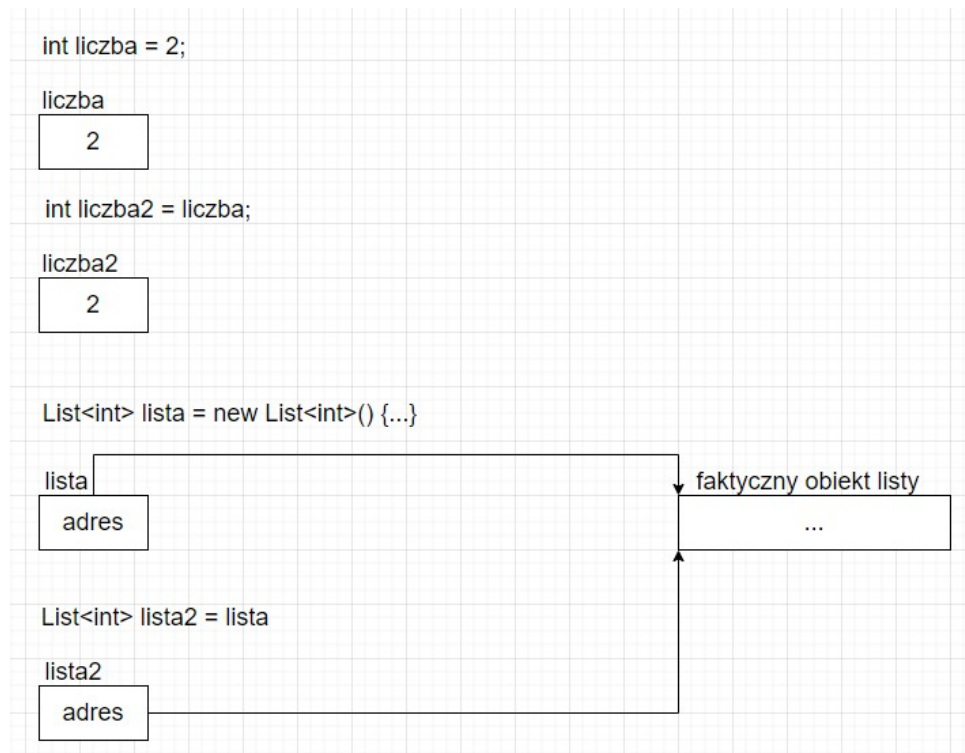
Powyższy schemat przedstawia jak wygląda sytuacja.

Dobrze zadać sobie pytanie co się stanie jeżeli zrobimy kopię zmiennej „liczba” do innej zmiennej typu „int” a co się stanie jeżeli zrobimy kopię zmiennej „lista” do innej zmiennej typu „List<int>”.

```
0 references
static void Main(string[] args)
{
    int liczba = 2;
    int liczba2 = liczba;

    List<int> lista = new List<int>() { 1, 2, 3 };
    List<int> lista2 = lista;
}
```

Jak powyższy kod zapiszemy w postaci schematu.



Jak widać w przypadku liczby „int” stworzyliśmy nowy pojemnik i wpisaliśmy tam taką samą wartość jak przechowuje oryginalna zmienna „liczba”. Wszystko jasne i oczywiste.

W drugim przypadku sytuacja jest inna. Wykonując kopię operatorem przypisania do nowego pojemnika przypisaliśmy adres znajdujący się w oryginalnej zmiennej. W wyniku czego mamy dwie zmienne z tym samym adresem wskazującym na ten sam obiekt. Odwołując się do „lista2” zmieniamy obiekt, który znajduje się także pod zmienną „lista”.

Sytuacja ta prowadzi do wielu dziwnych zachowań, o których nie każdy wie.

Ale zanim o nich jeszcze inny przykład:

```
string s1 = "TEST";
string s2 = "TEST";

s1[0] = 'B';
```

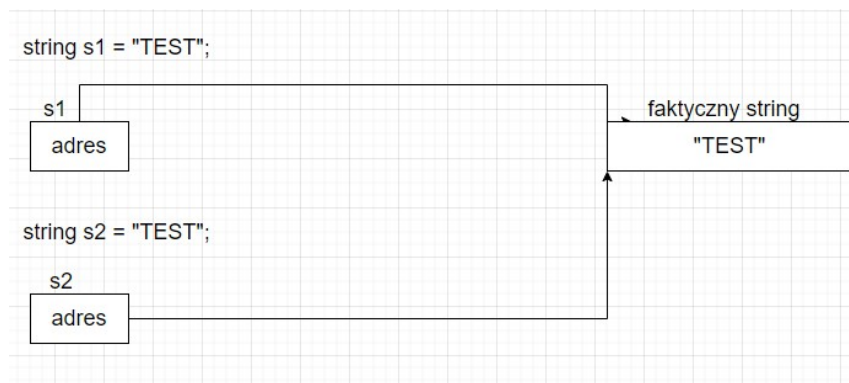
(local variable) string s1

CS0200: Property or indexer 'string.this[int]' cannot be assigned to -- it is read only

Dlaczego powyższy kod podświetla się na czerwono? Ponieważ ktoś kiedyś opracowując ten język ustalił, że takie stringi są „niemutowalne”. Dlaczego? Dla optymalizacji.

Na czym polega ta optymalizacja? Obie zmienne w teorii zawierają identyczny string. Po co więc go dublować w pamięci jak obie zmienne mogą zawierać adres/wskaźnik/referencję na jedną i tą samą kopię napisu „TEST”.

I teraz skąd bierze się błąd? Jeżeli język pozwolił by zmodyfikować ten jeden uniwersalny napis „TEST” to zmianie uległy by wszystkie inne odnośniki do niego. Tutaj mamy dwa, więc sprawa jest jeszcze do przeżycia. Ale co jeżeli w 15 miejscach programu tworzona jest zmienna z napisem „TEST”. Zmiana w jednym miejscu niosła by za sobą zmianę w każdym innym. Dlatego stringi są read-only.



Tak wygląda sytuacja. Jest ona dziwniejsza niż ta wyżej, ponieważ tutaj nie robimy nawet kopii operatorem przypisania. Po prostu autorzy języka taką przyjęli konwencję.

Jeżeli chcemy mieć pewność, że każda instancja stringu wskazuje inną kopię napisu musimy napisać tak:

```
String s1 = new String("TEST");
String s2 = new String("TEST");
```

Tutaj c# alokuje dwa stringi w pamięci z tą samą zawartością. Oczywiście zmienne s1 oraz s2 przechowują adres/referencję/wskaźnik na te dwa obiekty.

Wróćmy do kodu z intem i listą. Pierwsza konsekwencja istnienia typów referencyjnych została już opisana. Tworzenie kopii zmiennych za pomocą operatora przypisania nie tworzy kopii obiektu tylko kopię adresu.

Zobaczmy inną konsekwencję. Co się dzieje ze zmienną gdy podajemy ją do funkcji jako parametr? Tworzona jest jej kopia.

```
1 reference
static void Inkrementuj(int value)
{
    value++;
}

0 references
static void Main(string[] args)
{
    int liczba = 2;
    Inkrementuj(liczba);

    //liczba równa się ciągle 2
}
```

Funkcja wykona kopię zmiennej „liczba” do zmiennej „value”. Następnie wykona zwiększenie wartości w zmiennej „value”. Wartość oryginalna zostanie bez zmian. Wszystko jest jasne i proste.

Co się stanie gdy do funkcji podamy zmienną typu referencyjnego?

```
1 reference
static void Inkrementuj(List<int> value)
{
    for(int i = 0; i < value.Count; i++)
    {
        value[i] += 1;
    }
}

0 references
static void Main(string[] args)
{
    List<int> lista = new List<int> { 1, 2, 3 };
    Inkrementuj(lista);

    //lista zawiera liczbę {2,3,4}
}
```

Tutaj sytuacja jest taka sama, jednak tworzona jest kopia zmiennej z adresem. Zmienna „value” zawiera ten sam adres co zmienna „lista”. Obie wskazują ten sam obiekt. Funkcja zmieni więc ten obiekt i zmiany będą widoczne pod oryginalną zmienną po wyjściu z funkcji.

**Typy wartościowe możemy przekazywać do funkcji przez referencję dodając do argumentu słowo kluczowe „ref” lub „out”.**

```

1 reference
static void Inkrementuj(ref int value)
{
    value++;
}

0 references
static void Main(string[] args)
{
    int liczba = 2;
    Inkrementuj(ref liczba);

    //liczba równa się ciągle 3
}

```

```

1 reference
static void Inkrementuj(out int value)
{
    //value++; //error gdyż nie było wartości na start
    value = 3;
}

0 references
static void Main(string[] args)
{
    int liczba;
    Inkrementuj(out liczba);

    //liczba równa się 3
}

```

Słowo kluczowe „out” pozwala przekazać zmienną przez referencję do funkcji jeżeli ta nie ma wartości. Jak widać zmienna „liczba” nie ma wartości przed podaniem do funkcji. Oczywiście blokuje to niektóre operacje takie jak ++ bo skąd program ma wiedzieć co zwiększyć o 1 jeżeli w zmiennej nie ma wartości?

Trzecim mniej popularnym sposobem przekazywania zmiennej do funkcji na zasadach referencji jest „boxing”. **Każdy typ wartościowy staje się typem referencyjnym jeżeli jest częścią typu referencyjnego.**

```

0 references
class ASD
{
    public int A;
}

```

Klasa definiuje typ referencyjny. „int” jako składowa typu referencyjnego jest typem referencyjnym. Boxing działa na takiej zasadzie. Można utworzyć taką tymczasową klasę. Umieścić w niej zmienną wartościową, przekazać do funkcji i po powrocie odczytać z obiektu tejże klasy.

Zobaczmy taki przykład:

```
2 references
class MailSender
{
    1 reference
    public void SendMail() { }
}

1 reference
class MailMessenger
{
    private MailSender sender;

    0 references
    public MailMessenger(MailSender sender)
    {
        this.sender = sender;
    }

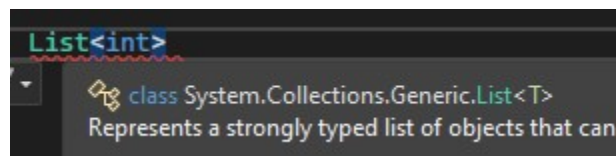
    0 references
    public void MakeMail() { }
    0 references
    public void SendMail() { sender.SendMail(); }
}
```

Ustawiamy tutaj pole, podając zależność za pomocą konstruktora. Zależnością jest obiekt. Dzięki mechanizmowi referencji cały proces wstrzykiwania operuje na tym samym obiekcie a nie jest wykonywana za każdym razem kopia obiektu MailSender.

### Jak sprawdzić czy typ jest wartościowy czy referencyjny?

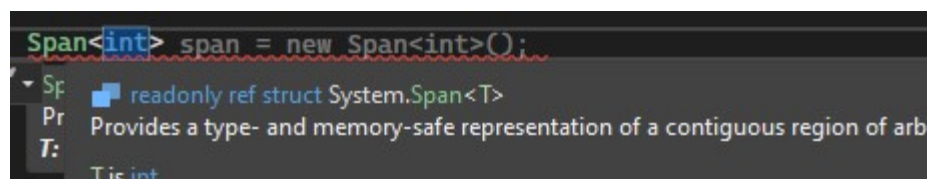
Każdy typ powstały na strukturze jest typem wartościowym.

Każdy typ powstały na bazie klasy jest typem referencyjnym.



```
List<int>
class System.Collections.Generic.List<T>
Represents a strongly typed list of objects that can b
```

Jak widać w odpowiedzi List<T> zostało zdefiniowane za pomocą klasy.



```
Span<int> span = new Span<int>();
readonly ref struct System.Span<T>
Provides a type- and memory-safe representation of a contiguous region of arb
T is int
```

Typ Span<T> jest typem wartościowym, ponieważ został zdefiniowany za pomocą struktury.



## **Kopia płytka i kopia głęboka**

Co jeżeli chcemy zrobić kopię typu referencyjnego?

Musimy napisać funkcję lub kawałek kodu, który alokuje nowy obiekt typu referencyjnego oraz przepisuje jego składowe jedna po drugiej. Tworzymy w ten sposób nowy identyczny obiekt. Jest to kopia płytka.

Co jeżeli zmienna referencyjna zawiera w sobie elementy referencyjne, a te elementy referencyjne kolejne elementy referencyjne?

Przykładem może być lista obiektów klasy. W takim przypadku nasza funkcja musi wykonać kopię na każdym szczeblu w sposób rekurencyjny. Takie coś nazywamy kopią głęboką.