

**ZPR PWr – Zintegrowany Program Rozwoju Politechniki Wrocławskiej**

# Aplikacje webowe na platformę .NET

W06 – Język C#: elementy obiektowe

# Syllabus

- Prosta klasa
- Pola instancji
- Metody instancji
- Tworzenie instancji i dostęp do pól instancji
- Słowo kluczowe **this**
- Właściwości
- Konstruktory
- Inicjatory i finalizatory
- Dekonstruktor
- Składowe statyczne
- konstruktor statyczny, klasa statyczna
- **const, readonly**
- Metody rozszerzające
- Klasy zagnieżdżone
- Klasy częściowe
- Metody częściowe
- Klasy anonimowe
- Typ dynamiczny
- Tworzenie klas pochodnych
- Modyfikatory dostępu w ramach dziedziczenia
- Słowo kluczowe **base** w konstruktorach
- Klasy abstrakcyjne
- Polimorfizm
- Operatory **is** oraz **as**
- Rzutowanie
- Klasa Object
- Interfejs
- Składowe z jawnie/niejawnie podanym interfejsem
- Dziedziczenie interfejsów (po wielu interfejsach)
- Typy bezpośrednie
- Różnica między strukturą z obiektem
- Typy bezpośrednie/wartościowe – przykłady
- Struktury
- Różnica między strukturą z obiektem w argumentach metod:
  - przekazywane przez wartość
  - przekazywane przez referencję
  - przekazywane jako wyjściowe
- Typy wyliczeniowe - **enum**
- Wyjątki
  - Rzucanie wyjątku
  - Obsługa wyjątku
  - Wolny mechanizm
  - Reguły używania wyjątków
- Klasy generyczne
- Typy generyczne
- Interfejsy generyczne
- Metody generyczne

C#

# KLASY I OBIEKTY

## Tworzenie kodu klasy

- W C# klasa musi być zadeklarowana w jakiejś przestrzeni nazw (**namespace** <nazwaPrzestrzeniNazw> { ... }).
- Nie licząc możliwych wcześniejszych **modyfikatorów** definicja klasy zaczyna się od słowa kluczowego **class** oraz pary nawiasów klamrowych zawierających definicję klasy.

```
namespace Classes
{
    class Person1
    {
        // body of class
        ...
    }
}
```

## Pola instancji (obiektu)

- W ramach ciała klasy można definiować pola instancji tejże klasy.
- Ich deklaracja wygląda jak deklaracja zmiennych lokalnych w ciele metody
- W deklaracji pól instancji określona jest widoczność tychże (**public**, **private** i inne)
- Deklaracje pól mogą być przemieszane z metodami.
- Pola, nawet jeśli są zapisane na końcu klasy, widoczne i dostępne są od początku ciała klasy

```
namespace Classes
{
    class Person1
    {
        public string name;
        public double salary;
        // the rest of class
        . . .
    }
}
```

## Metody instancji

- W ciele klasy definiowane są również metody instancji.
- Przed zwracanych typem rezultatu metody znajdują się modyfikatory, w tym modyfikatory dostępu (**public**, **private** i inne).
- Uwaga: w dalszych kodach pomijane będą elementy wcześniej zaprezentowane w ramkach

```
class Person1{
    public void SalaryUp(int howMany) {
        if (howMany > 0)
            salary += howMany;}

    public void SalaryDown(int howMany) {
        if (howMany > 0)
            SalaryDownInner(howMany);}

    private void SalaryDownInner(int howMany) {
        if (howMany > 100)
            howMany = 100;
        salary -= howMany;
        if(salary<1000)
            salary = 1000;}
}
```

# Tworzenie instancji i dostęp do pól instancji

- Nową instancję klasy czyli obiekt klasy tworzy się poprzez operator **new** z nazwą klasy i para nawiasów okrągłych.
- Operator ten wywołuje **konstruktor** (szczegóły będą dalej) **bezparametryczny**, zwany **konstruktorem domyślnym**. Jeśli twórca klasy nie napisze żadnego konstruktora, konstruktor domyślny zostanie wytworzony przez kompilator.
- **Klasa** to typ **referencyjny**, więc **zmienna** do operowania na obiektach to zmienna pamiętająca **referencję**.
- Do operowania na polach instancji oraz metodach danego obiektu służy **operator kropki**.
- Poza ciałem klasy można używać tylko składowych publicznych.

```
namespace Classes
{
    class Program
    {
        static void Person1Test()
        {
            Person1 person = new Person1();
            person.name = "Kowalski";
            person.salary = 3000;
            person.SalaryUp(100);
            //person.SalaryDownInner(100);
            Console.WriteLine($"name={person.name}, salary={person.salary}");
        }
    }
}
```

name=Kowalski, salary=3100

## Słowo kluczowe **this**

- Słowo kluczowe **this** to referencja na bieżący obiekt.
- Jednym z zastosowań słowa kluczowego **this** jest dostęp do pól instancji klasy w przypadku przesłonięcia ich przez parametry metody.
- Słowo **this** można też użyć, gdy chcemy użyć bieżącego obiektu jako parametru jakiejś metody.

```
class Person1{
    public string DescriptionStr()
    {
        return Program.GetSentence(this);
    }
}
class Program
{
    public static string GetSentence(Person1 person)
    {
        return $"{person.name} has a salary of {person.salary} ";
    }
    static void Person1TestThis()
    {
        Person1 person = new Person1();
        person.name = "Kowalski";
        person.salary = 3000;
        Console.WriteLine(person.DescriptionStr());
    }
}
```

Kowalski has a salary of 3000



# Modyfikatory dostępu

Kilka modyfikatorów dostępu

- **public** – dostęp do pola/metody/właściwości w kodzie dowolnej klasy
- **private** – dostęp tylko w kodzie danej klasy
  - Domyślny (gdy brak modyfikatora)
- **protected**, będzie omówione później przy dziedziczeniu.
- Istnieją również **internal**, **protected internal**

# Właściwości

- **Idea:** zamiast metod `getX()`, `setX()` możliwość używania pola `x` jakby było publiczną składową, jednak z możliwością kontroli jak w getterze i setterze.
- Rozwiązanie: **właściwości**, czyli getter i setter ale zapisywany *jak pole*.
- Pozwala to decydować o obecności lub nie gettera/settera i jego mniejszej widoczności niż ogólnie właściwości.
- Zalecane używanie właściwości zamiast pól publicznych.
- W setterze używa się kontekstowego słowa kluczowego **value**, które jest domyślnym parametrem.

```
class Person2{
    private string name;    // private field name
    private double salary;  // private field salary
    public string Name      //public property Name
    {
        get { return name; } // public getter
        set { name = value; } // public setter
    }
    public double Salary    //public property Salary
    {
        // private setter
        private set { if (value > 0) salary = value; else salary = 0; }
        get {return salary;} // public getter
    }
}
```

## Właściwości c.d.

- Właściwości – zapis pascalowy (bo w sumie to są metody)
- Właściwości zatem nie można używać jako parametrów z modyfikatorem **ref** i **out**.
  - trzeba przepisać wartość do zmiennej, użyć w wywołaniu metody tej zmiennej i po powrocie z wykonania metody z powrotem przepisać wartość do właściwości.
- Właściwości **automatyczne**: piszemy tylko słowa `get` lub `set` ze średnikiem. Wytwarza się ukryte pole instancyjne.
- Mogą istnieć wirtualne właściwości, które nie są powiązane prostą relacją z polem
  - `CenaNetto`, `CenaBrutto`
  - `Imie`, `Nazwisko`, `ImieNazwisko`

## Właściwości – przykład 2

```
class Person3
{
    private double salary;
    public string Name { get; set; } // automatic property
    public double Salary
    {
        private set { if (value > 0) salary = value; else salary = 0; }
        get { return salary; }
    }
}

class Program
{
    static void Person3Test()
    {
        Person3 person = new Person3();
        person.Name = "Kowalski";
        //person.Salary = 3000; // compilation error, not a public setter
        person.Name += "-Nowak";
        Console.WriteLine($"name={person.Name}, salary={person.Salary}");
    }
}
```

name=Kowalski-Nowak, salary=0

# Konstruktory

- Deklaracja konstruktora: podobna do metody, ale nic nie zwraca (nawet **void**) i nazwa musi być taka sama jak nazwa klasy.
- Jeśli **nie ma żadnego konstruktora** napisanego wprost, **kompilator wytworzy** konstruktor **bezparametryczny** (domyślny) wypełniający pola wartościami **domyślnymi**.
- Napisanie **dowolnego konstruktora** powoduje, że kompilator **nie wytworzy** kod dla konstruktora domyślnego.

```
class Person4{
    public string Name { get; set; }
    public double Salary { get; set; }
    public Person4(string name, int salary=3000)
    {
        Name = name;
        Salary = salary;
    }
}
class Program{
    static void Person4Test()
    {
        // compiler do NOT create default constructor
        // Person4 person = new Person4();
        Person4 person = new Person4("Kowalski");
        person.Salary += 500;
        Console.WriteLine($"name={person.Name}, salary={person.Salary}");
    }
}
```

name=Kowalski, salary=3500

## Wartość domyślna dla pola lub właściwości

- Pole (lub właściwość automatyczna) może posiadać domyślną wartość.
  - Jest to niemożliwe dla nie-automatycznych właściwości (byłyby dwa miejsca dla takiej wartości)
- Po prostu po nazwie pola lub zamykającym nawiasie klamrowym znak równości oraz wyrażenie zakończone średnikiem.

```
class Person3
{
    private double salary = 12000.4;
    public string Name { get; set; } = "Kowalski";

    public double Salary
    {
        private set { if (value > 0) salary = value; else salary = 0; }
        get { return salary; }
    }
    // = 1234.5; // compilation error
}
```

# Inicjatory i finalizatory

- **Inicjatory** to skrót notacyjny pozwalający w jednej instrukcji stworzyć obiekt oraz przypisać wartości do pól i właściwości nowostworzonego obiektu.
  - Środowisko programistyczne podpowiada nazwy pól i właściwości
  - Jest to rozbijane na ciąg przypisań.
- Zapis: po argumentach konstruktora w nawiasach klamrowych sekwencja przypisań do pól lub właściwości rozdzielona przecinkami.

```
class Program{  
    static void Person4Test2()  
    {  
        int value = 100;  
        Person4 person = new Person4("Kowalski") { Salary = 1400+value, Name="Nowak" };  
        Console.WriteLine($"name={person.Name}, salary={person.Salary}");  
    }  
}
```

name=Nowak, salary=1500

- **Finalizatory** – uruchamiane przy zwalnianiu obiektu z pamięci. Maszyna wirtualna decyduje, kiedy będzie zwalniać pamięć (jeśli w ogóle).
- Finalizatory mają sens, gdy tworzy się klasy z wykorzystaniem metod rodzimym z języków o dostępie bezpośrednim do pamięci (np. C++).

## Przeciążanie konstruktorów

- Konstruktory, tak jak metody, **można przeciążać**.
- Nieraz potrzeba wywołać działanie innego konstruktora (tylko jednego) i tylko dopisać kilka nowych instrukcji.
- Aby wywołać inny konstruktor podczas uruchamiania bieżącego należy po nawiasie zamykającym parametry konstruktora napisać dwukropek, słowo **this** oraz w nawiasach argumenty wywołania innego konstruktora.
- Oczywiście wywoływany konstruktor może wywołać jeszcze inny itd. Tworzy się łańcuch wywołań konstruktorów

```
class Person4
{
    public string Name { get; set; }
    public double Salary { get; set; }
    public Person4(string name, int salary=3000)
    {
        Name = name;
        Salary = salary;
    }
    public Person4(int salary, string name):this(name, salary)
    {
        // any further instructions
    }
}
```



# Dekonstruktor

- **Dekonstruktor** „przerabia” obiekt na zestaw wartości lub krotkę wartości.
- Należy napisać metodę `Deconstruct` z minimum dwoma parametrami.
- Może być użyta jak zwykła metoda, ale można też ją odpowiednio użyć w notacji z krotką.
- Dekonstruktory można **przeciążać**.

```
class Person5
{
    public string Name { get; set; }
    public double Salary { get; set; }
    public int Age { get; set; }
    public Person5(string name, double salary, int age)
    {
        Name = name;
        Salary = salary;
        Age = age;
    }
    public void Deconstruct(out string name, out double salary, out int age)
    {
        (name, salary, age) = (Name, Salary, Age);
    }
    public void Deconstruct(out string name, out double salary)
    {
        (name, salary) = (Name, Salary);
    }
}
```

## Dekonstruktor – użycie

- Parametry muszą być oznaczone jako **out**.
- Można użyć '\_' jako „dzokera” na oznaczenie tych składników krotki, których wartość nie jest dla nas ważna.

```
static void Person5Test()  
{  
    Person5 person = new Person5("Kowalski",2500,33);  
    person.Deconstruct(out _, out double salary1, out int age1);  
    string name2;  
    double salary2;  
    person.Deconstruct(out name2, out salary2);  
    (name2, salary2) = person; // the same  
}
```

## Składowe statyczne

- **Składowe statyczne** są powiązane z klasą (a nie instancją klasy).
- Deklaruje się je podobnie jak składowe statyczne, jednak poprzedzone słowem kluczowym **static**.
- **Dostęp** uzyskuje się przez **nazwę klasy** i **operator kropki**, po którym występuje składowa statyczna.
- Nie jest potrzebna żadna instancja klasy, aby używać składowej statycznej.
- Jeśli **nie zainicjujemy** pola statycznego (właściwości automatycznej), to będzie miało **wartość domyślną**. Podczas inicjacji można używać innych składowych statycznych (już zainicjowanych).

Metody statyczne, konstruktor statyczny, klasa statyczna

- **Metody statyczne** nie mogą używać referencji **this**.
- Istnieje możliwość definicji **jednego bezparametrycznego konstruktora statycznego**.
  - Będzie uruchomiony tylko raz przy pierwszym skorzystaniu ze składowej statycznej lub przy pierwszej konstrukcji obiektu.
- Jeśli **klasa** posiada **tylko** składowe **statyczne** można ją zadeklarować jako **statyczną**:
  - Staje się automatycznie **nierozszerzalna** i nie można utworzyć instancji takiej klasy (kompilator automatycznie dodaje cechy **sealed** i **abstract**)

## Hermetyzacja danych

- Modyfikator **const** oznacza pole, którego wartość jest **stała** i musi być wyznaczona **w czasie kompilacji**. Takie pole automatycznie staje się polem statycznym.
- Modyfikator **readonly** używa się do pól, których wartość będzie wyznaczana **podczas konstrukcji wartości pola** (instancyjnego lub statycznego).
- Od wersji C#6.0 preferowane są właściwości automatyczne tylko do odczytu niż pola z modyfikatorem **readonly**.

## static, const, readonly - przykłady

```
class CloneCounter
{
    const int START=1;
    readonly int localStart;
    static int lastNumber = START;
    static string globalName;
    static CloneCounter(){
        globalName = "Global counter at "+DateTime.Now;
        //START = 2;  // compile error
    }
    public CloneCounter()
    {
        localStart = lastNumber;
        lastNumber++;
    }
}
```

## Metody rozszerzające

- Idea: Gdy klasa X jest **zapięczętowana (nierozszerzalna)** chciałoby się jednak dodać pewne metody tak, aby można używać **notacji z kropką** na rzecz obiektu klasy X. Stąd powstał mechanizm dodawania metod, które można w ten sposób wywołać.
- Metoda rozszerzającą typ X :
  - Musi być w klasie statycznej
  - Musi być metodą statyczną
  - Pierwszy parametr musi być typu X
  - Pierwszy parametr musi być poprzedzony słowem kluczowym **this**
  - Trzeba importować przestrzeń nazw klasy z metodą rozszerzającą klasę X albo wywoływać metodę w tej samej przestrzeni nazw
    - Nie wystarczy przestrzeń nazw klasy X
- Mechanizm ten bywa mylący, więc zalecane jest używanie tylko gdy to jest konieczne.
- ASP .NET często używa w kodzie strony WWW metody rozszerzające jak również w plikach `Program.cs` i `Startup.cs`.
- Mechanizm ten jest również często używany w klasach-kolekcjach.

## Metody rozszerzające - przykład

- Klasę **string** nie można dziedziczyć, ale można utworzyć metodę rozszerzającą tę klasę.

```
public static class MyStringExtension
{
    public static string AddStars(this string str, int howMany)
    {
        while (howMany > 0)
        {
            str += "*";
            howMany--;
        }
        return str;
    }
}

class Program
{
    static void AddStarsTest()
    {
        string line = "any word ";
        Console.WriteLine(MyStringExtension.AddStars(line, 5));
        Console.WriteLine(line.AddStars(5));
        Console.WriteLine(" this is also possible ".AddStars(3));
    }
}
```

```
any word *****
any word *****
this is also possible ***
```



## Klasy zagnieżdżone

- Składową klasy może być inna klasa.
- Najczęściej jest to klasa prywatna (i tylko wtedy klasa może być prywatna), czyli dostępna tylko dla klasy nadrzędnej.
- **Klasa zagnieżdżona** może używać wszystkich składowych klasy nadrzędnej
- **Klasa nadrzędna** może używać tylko składowych publicznych klasy podrzędnej
- Słowo **this** w klasie zagnieżdżonej to dostęp do instancji tejże klasy zagnieżdżonej
- Nie ma bezpośredniej możliwości dostępu do instancji klasy nadrzędnej:
  - Można przekazać jako parametr
  - Dygresja – jest to inne rozwiązanie niż w C++/Java.
  - Uwaga – więcej niż jeden stopień zagnieżdżenia jest w zasadzie nieprzydatny i wprowadza tylko bałagan

## Klasy częściowe

- **Klasy częściowe** są przydatne, gdy szkielet klasy generuje jakiś wizzard i generacja taka może się zdarzać wiele razy (wtedy zmiany w pliku dokonane przez programistę byłyby tracone).
- Implementację klasy można rozbić na dwa i więcej plików.
- Deklaracje (definicje) metod, pól, własności itd. mogą znajdować się w różnych plikach, a ich użycie w całym innych.
  - Deklaracje/definicje te MUSZĄ w którymś pliku istnieć

## Metody częściowe

- **Metody częściowe** (raczej nieadekwatna nazwa) to mechanizm dla klas częściowych.
- Pozwala w jednym pliku z klasą częściową zadeklarować, że będzie prawdopodobnie w innym pliku z tą klasą implementacja pewnej metody. Kompilator zezwoli zatem na wywoływanie tej metody.
- W innym pliku powinna się znajdować implementacja takiej metody.
  - Jeśli w procesie kompilacji nie znajdzie się implementacja metody częściowej, kompilator zignoruje wywołanie tej metody (stąd wymagania jak poniżej)
  - Nie mogą być dwie implementacje takiej metody
- Metoda musi być typu **void**
- Nie może zawierać parametru typu **out**.
- Może zawierać parametry typu **ref**.

# Klasy częściowe i metody częściowe - przykład

- Definicja klasy `Probe` rozbita na 3 pliki.

```
public partial class Probe
{
    public string Name { get; set; } = "Kowalski";
    partial void ShowName();
    partial void ShowAge();
    public void ShowIt()
    {
        NormalMethod();
        Console.WriteLine($"Before {Age}");
        ShowName();
        ShowAge();
        Console.WriteLine("After");
    }
}
```

```
//public partial class Probe
//{
//    partial void ShowAge()
//    { Console.WriteLine($"age={Age}"); }
//}
```

```
public partial class Probe
{
    public int Age { get; set; } = 30;
    partial void ShowName()
    { Console.WriteLine($"name={Name}"); }
    public void NormalMethod()
    { Console.WriteLine("Normal"); }
}
```

Normal  
Before 30  
name=Kowalski  
After

Po odkomentowaniu

Normal  
Before 30  
name=Kowalski  
age=30  
After

# Klasy anonimowe

- Klasy anonimowe (od C# 4) to klasa bez nazwy tworzona dla obiektu poprzez operator **new** i deklaracje pól z wartościami w nawiasach klamrowych
- Ponieważ klasa jest anonimowa może być przypisana do zmiennej z **var** jako określenie typu.
- Podczas kompilacji wygenerowana będzie niewidoczna klasa dla takich obiektów

```
static void AnonTypeTest()  
{  
    var person = new  
    {  
        Name = "Smidth",  
        Age = 30  
    };  
    System.Console.WriteLine(person.Age);  
    System.Console.WriteLine(person);  
}
```

```
30  
{ Name = Smidth, Age = 30 }
```

```
static void AnonTypeTest()  
{  
    var person = new  
    {  
        AnonymousType 'a'  
        Typy anonimowe:  
        'a jest new { string Name, int Age }  
    };  
    System.Console.WriteLine(person);  
}
```

## Typy anonimowe – możliwości i ograniczenia

- Jeśli nazwy właściwości są takie same i w tej samej kolejności, to kompilator (w ramach tego samego podzespołu) utworzy jedną klasę.
- Typy anonimowe są niemodyfikowalne, zatem nie można zmienić wartości pól
- Nie posiadają konstruktorów i żadnych metod czy właściwości
- Nie posiadają składowych statycznych
- Można stworzyć tablicę elementów obiektów typu anonimowego.
- Typy anonimowe są silnie typizowane, kompilator rozpoznaje, jaki typ jest używany w argumencie lub w wyniku.
- Domyślnie dziedziczą po **object**.
- W zasadzie dany typ anonimowy powinno się używać **w ramach jednej metody**, czyli nie powinien być zwracany jako wynik, ani pobierany jako argument (nie można używać jako oznaczenia typu słowa **var**).
- Powyższa zasada może być złamana jeśli użyjemy typów **dynamicznych**.

## Określenie typu jako **dynamic**

- Określenie typu zmiennej jako **dynamic** oznacza, że wszelkie operacje na danej zmiennej będą sprawdzane dopiero podczas wykonania
  - Kompilator zatem nie sprawdza składni wyrażenia z taką zmienną (oprócz nawiasów, poprawności identyfikatorów itp.)
- Posiada metody do sprawdzania, czy dane pole lub metoda istnieje w ramach właściwego typu.
  - Działa mechanizm refleksji

```
19 static void AnonTypeArg(dynamic arg)
20 {
21     System.Console.WriteLine(arg.Age);
22     System.Console.WriteLine(arg.Salary);
23 }
24
25 static void AnonTypeArgTest()
26 {
```

Nieobsługiwany wyjątek

Microsoft.CSharp.RuntimeBinder.RuntimeBinderException: '<>f\_\_AnonymousType0<string,int>' does not contain a definition for 'Salary'

```
static void AnonTypeArg(dynamic arg)
{
    System.Console.WriteLine(arg.Age);
    System.Console.WriteLine(arg.Salary);
}

static void AnonTypeArgTest()
{
    var person = new
    {
        Name = "Smidth",
        Age = 30
    };
    AnonTypeArg(person);
}
```

```
30
Unhandled exception. Microsoft.CSharp.RuntimeBinder.RuntimeBinderException: '<>f__AnonymousType0<string,int>' does not contain a definition for 'Salary'
at CallSite.Target(Closure, CallSite, Object)
at System.Dynamic.UpdateDelegates.UpdateAndExecute1[T0,TRet](CallSite site, T0 arg0)
at AnonType.Program.AnonTypeArg(Object arg) in C:\Users\dariu\source\repos\AnonType\AnonType\Program.cs:line 22
at AnonType.Program.AnonTypeArgTest() in C:\Users\dariu\source\repos\AnonType\AnonType\Program.cs:line 32
at AnonType.Program.Main(String[] args) in C:\Users\dariu\source\repos\AnonType\AnonType\Program.cs:line 38
```

## Przeładowanie operatorów

- Część operator można przeładować dla implementowanej klasy (struktury)
- Metody takie muszą być **publiczne** i **statyczne**
- „nazwa” metody to „**operator** <operator>”, i musi posiadać odpowiednią liczbę parametrów z których minimum jeden musi być typu klasy w której jest zaimplementowany np.:  
**public static** Fraction **operator** +(Fraction a, Fraction b)
- Przykładowe operatory możliwe do przeładowania:
  - Arytmetyczne: +, -, /, \* itd.
  - Bitowe: &, | itd.
  - Porównywania: ==, !=, < itd.
  - Indeksator: a[]
  - Unarne: -x, !x, ++ itd.
  - Stałe: true, false
- Jeśli przeładowane zostanie np. +, to operator += stanie się automatycznie złączeniem wykonania + i następnie =
- Przeładowania stałych true/false będzie wykorzystywane w operator logicznych typu &&.
- Więcej: <https://docs.microsoft.com/pl-pl/dotnet/csharp/language-reference/operators/operator-overloading>



C#

# DZIEDZICZENIE

## Tworzenie klas pochodnych, dostęp do składowych

- Słownictwo:
  - Nadklasa, klasa bazowa, klasa dziedziczona, klasa rozszerzana, klasa ogólna
  - Podklasa, klasa pochodna, klasa dziedzicząca, klasa rozszerzająca, klasa wyspecjalizowana
- Klasę B, pochodną od klasy A, tworzymy poprzez zapisanie po nazwie klasy B dwukropka i nazwy klasy A.
- Jeśli nie ma podanej klasy po której dziedziczy dana klasa, dziedziczy ona po klasie **object**.
- Można dziedziczyć tylko po jednej klasie
  - Nie ma wielodziedziczenia jak w C++
- Dziedziczone są **wszystkie** składowe (również prywatne) **oprócz konstruktorów**.
  - Konstruktor domyślny może wytworzyć kompilator (jeśli istnieje konstruktor domyślny nadklasy)

## Modyfikatory dostępu i przykład

- W klasie dziedziczącej można korzystać z elementów z modyfikatorami **public** oraz **protected**, natomiast nie można ze składowych typu **private**.

```
class Person
{
    public string Name { get; set; }
    public int YearOfBirth { get; set; }
    protected string Info { get; set; }
    private string hidden;
    public Person(string name, int year, string info, string hidden)
    {
        Name = name;
        YearOfBirth = year;
        Info = info;
        this.hidden = hidden;
    }
    public virtual void showAll()
    {
        Console.WriteLine($"{Name}, {YearOfBirth}, {Info},{hidden}");
    }
}
```

# Modyfikatory dostępu - przykład

```
class Worker : Person
{
    public double Salary { get; set; }
    public Worker(string name, int year, string info, double salary):base(name, year,
        info, "worker")
    {
        Salary = salary;
    }
    public void ShowProtected()
    {
        Console.WriteLine($"{Info}");
        //Console.WriteLine($"{hidden}");
    }
}
```

```
Schmidt, 1999, all is OK,agent
Olaf, 2000, good person,worker
3000
good person
```

```
class Program
{
    public static void PersonTest()
    {
        Person person = new Person("Kowal", 1999, "all is OK", "agent");
        //person.Info = " anything";
        //person.hidden = " anything";
        person.Name = "Schmidt";
        person.showAll();
        Worker worker= new Worker("Nowak", 2000, "good person", 3000);
        worker.Name = "Olaf";
        //worker.Info = " anything";
        worker.showAll();
        worker.ShowProtected();
    }
}
```

## Wywoływanie konstruktorów nadklasy - **base**

- Konstruktor nadklasy najpierw wywołuje konstruktor podklasy (i kaskadowo dalej).
- Jeśli nie podamy, który konstruktor chcemy wywołać, uruchomiony będzie konstruktor domyślny nadklasy.
- Jeśli takiego konstruktora nie ma, lub chcemy wywołać inny konstruktor, należy za nawiasem zamykającym parametry konstruktora, napisać dwukropek oraz słowo kluczowe **base** i następnie w nawiasach argumenty wybranego konstruktora.
- Parametrem może być oczywiście dowolne wyrażenie zgodne typem.

```
class Worker : Person
{
    //...
    public Worker(string name, int year, string info, double salary):base(name, year,
        info, "worker")
    {
        Salary = salary;
    }
    // ...
}
```

## Klasy zamknięte

- Część klas z różnych względów (również bezpieczeństwa) nie powinny być dziedziczone.
- W języku C# do oznaczenia takiej klasy używane jest słowo kluczowe **sealed** (*ang. zapieczętowane*).
- W bibliotece C# istnieje wiele takich klas, jedną z nich jest np. klasa **string**.
- Dla klas zamkniętych, celem skrócenia zapisu, można używać metod rozszerzających. Nie jest to jednak tworzenie nowej klasy, ani nawet rzeczywiste rozszerzenie klasy.

## Modyfikator **virtual/override/new/sealed**

- Tworząc klasę pochodną istnieje potrzeba przeddefiniowania działania części metod obecnych w klasie nadrzędnej. Aby to umożliwić należy dodać w klasie bazowej przed nagłówkiem metody słowo kluczowe **virtual**.
- W klasie pochodnej pisząc metodę o tym samym nagłówku należy dopisać słowo **override**.
- Powyższe nazywa się mechanizmem wirtualizacji metod. Niekiedy istnieje potrzeba odejścia od tego mechanizmu (rzadko). Wtedy nową metodę trzeba poprzedzić słowem kluczowym **new**.
- Powyższe dotyczy oczywiście również właściwości.
- Metody rozszerzające nie są prawdziwymi składowymi klasy, więc nie podlegają tym regułom.
- Dodanie modyfikatora **sealed** do metody kończy mechanizm wirtualizacji danej metody na danej klasie.

# virtual/override/new - przykład

```
public class ClassA
{
    public void DisplayName() { Console.WriteLine("Class A"); }
}
public class ClassB : ClassA
{
    // warning that B is hiding an inherited method recommends using the word new
    public virtual void DisplayName() { Console.WriteLine("Class B"); }
}
public class ClassC : ClassB
{
    public override void DisplayName() { Console.WriteLine("Class C"); }
}
public class ClassD : ClassC
{
    public new void DisplayName() { Console.WriteLine("Class D"); }
}
```

```
public static void InheritanceTest()
{
    ClassD objD = new ClassD();
    ClassC objC = objD;
    ClassB objB = objD;
    ClassA objA = objD;

    objD.DisplayName();
    objC.DisplayName();
    objB.DisplayName();
    objA.DisplayName();
}
```

Class D  
Class C  
Class C  
Class A



## Wywołanie metody z nadklasy - **base**

- Pisząc nową metodę wirtualną (właściwość) w podklasie często istnieje potrzeba wywołania tejże metody z nadklasy. Aby to dokonać należy użyć słowa kluczowego **base**, które zachowuje się analogicznie jak **this**, ale oznacza referencję do „obiektu nadklasy”.

```
class Person {  
    //...  
    public virtual void showAll()  
    {  
        Console.WriteLine($"{Name}, {YearOfBirth}, {Info},{hidden}");  
    }  
}  
class Worker : Person {  
    //...  
    public override void showAll()  
    {  
        base.showAll();  
        Console.WriteLine($"{Salary}");  
    }  
}
```

```
public static void TestBaseAndPolimorphizm()
```

```
{  
    Person person = new Person("Kowal", 1999, "all is OK", "agent");  
    Worker worker = new Worker("Nowak", 2000, "good person", 3000);  
    person.showAll();  
    worker.showAll();  
    person = worker;  
    Console.WriteLine("after assignment person = worker");  
    person.showAll();  
}
```

```
Kowal, 1999, all is OK,agent  
Nowak, 2000, good person,worker  
3000  
after assignment person = worker  
Nowak, 2000, good person,worker  
3000
```

## Klasy abstrakcyjne

- Klasy abstrakcyjne to klasy, które są oznaczone słowem kluczowym **abstract**.
- Nie można tworzyć instancji klas abstrakcyjnych
  - Jednak może posiadać konstruktory!
- Można (trzeba?) dziedziczyć po klasach abstrakcyjnych.
- Można tworzyć zmienne referencyjne typu abstrakcyjnego.
- Klasy abstrakcyjne mogą posiadać **metody abstrakcyjne**, czyli tylko nagłówki metod bez ich implementacji
  - Jeżeli klasa ma chociaż jedną metodę abstrakcyjną (również odziedziczoną) jest wtedy klasą abstrakcyjną i **musi** być poprzedzona słowem **abstract**.
- Metody abstrakcyjne są automatycznie wirtualne i muszą w klasach pochodnych być nadpisane (**override**).
- Właściwości też mogą być abstrakcyjne.
- Klasy nie-abstrakcyjne nazywamy **klasami konkretnymi**.

# Klasy abstrakcyjne-przykład

```
public abstract class Animal {  
    string Name { get; set; }  
    public Animal(string name)  
    { Name = name; }  
    public abstract string GiveVoice(int howMany);  
}
```

```
public class Cat : Animal {  
    public string Color { get; set; }  
    public Cat(string name, string hair) : base(name)  
    { Color = hair; }  
    public override string GiveVoice(int howMany)  
    { return "Miau"; }  
}
```

```
Miau  
hau, hau,
```

```
public class Dog : Animal{  
    public int Height { get; set; }  
    public Dog(string name, int height) : base(name)  
    { Height = height; }  
    public override string GiveVoice(int n)  
    {  
        string retStr = "";  
        for (int i = 0; i < n; i++)  
            retStr += "hau, ";  
        return retStr;  
    }  
}
```

```
public static void AnimalTest() {  
    // Animal Animal = new Animal ();  
    Animal zw1 = new Cat("Mrau", "white");  
    Animal zw2 = new Dog("Dingo", 50);  
    Console.WriteLine(zw1.GiveVoice(2));  
    Console.WriteLine(zw2.GiveVoice(2));  
}
```

# Polimorfizm

- Polimorfizm z greki to wielopostaciowość.
- Pojęcie związane z mechanizm metod wirtualnych.
- Podczas wywoływania metody dla danej zmiennej referencyjnej uruchamiana jest ta, która jest właściwa dla rzeczywistego typu obiektu.
  - Przykład z klasami `ClassA`, `ClassB`, `ClassC`, `ClassD`
- W połączeniu z klasami abstrakcyjnymi (i interfejsami) pozwala pisać ogólne algorytmy jeszcze bez implementacji metod abstrakcyjnych.

```
public static void AllVoices(Animal[] arrAnim, int howMany)
{
    foreach (Animal zw in arrAnim)
        Console.WriteLine(zw.GiveVoice(howMany));
}

public static void PolimorphizmTest()
{
    Animal[] arr= { new Cat("Mrau", "white"),
        new Dog("Dingo", 50),
        new Dog("Rex", 30),
        new Cat("Garfild", "red")};
    AllVoices(arr, 4);
}
```

```
Miau
hau, hau, hau, hau,
hau, hau, hau, hau,
Miau
```

## Rzutowanie i operatory: **is** oraz **as**

- W ramach zmiennych/wyrażeń typów referencyjnych rzutowanie **w górę** hierarchii dziedziczenia jest wykonywane **automatycznie**.
- Rzutowanie **w dół** musi być wpisane **przez programistę wprost**, gdyż może być niepoprawne i wtedy wygeneruje wyjątek
- Zamiast korzystać z mechanizmu wyjątków można najpierw sprawdzić, czy zmienna referencyjna jest określonego typu poprzez operator **is**.
- Innym sposobem jest rzutowanie za pomocą operatora **as**, który w przypadku niepoprawnego typu zwrócić **null** zamiast wyjątku.

# Operator is i as - przykład

```
public static void CheckPerson(Person pers)
{
    if (pers is Worker)
        System.Console.WriteLine("This is a worker");
    else
        System.Console.WriteLine(" This is NOT a worker ");
}

public static void CheckAsWorker(Person pers)
{
    Worker man = pers as Worker;
    if (man != null)
        System.Console.WriteLine($"Salary={man.Salary}");
    else
        System.Console.WriteLine("null");
}

static void testIsAndAs()
{
    Person person = new Person("Kowal", 1999, "all is OK", "agent");
    Worker worker = new Worker("Nowak", 2000, "good person", 3000);
    CheckPerson(person);
    CheckPerson(worker);
    CheckAsWorker(person);
    CheckAsWorker(worker);
    worker = (Worker)person;
}
```

```
This is NOT a worker
This is a worker
null
Salary=3000
Unhandled exception. System.InvalidCastException: U
e.Worker'.
    at Inheritance.Program.testIsAndAs() in C:\Users
    at Inheritance.Program.Main(String[] args) in C:
```

# Rzutowanie

- Jeśli klasa B dziedziczy (bezpośredni lub pośrednio) po klasie A, to:
  - Jeśli mamy referencję na klasę B i chcemy ją użyć jako referencję do klasy A to nastąpi automatyczne rzutowanie na klasę A
  - Jeśli mamy referencję na klasę A i chcemy ją użyć jako referencję do klasy B to należy wykonać wprost rzutowanie tej referencji na klasę B poprzez:  
`(B) referencjaNaA`
- W drugim przypadku, jeśli rzutowanie jest niepoprawne rzucony zostanie wyjątek `InvalidCastException`.
- Ponieważ rzutowanie ma niski priorytet częste jest otaczanie wyrażenia rzutowania nawiasami aby np. wywołać metodę z klasy B:  
`((B) referencjaNaA).MetodaKlasyB()`
- Operator `nameof()` – zamienia identyfikator klasy/metody/zmiennej itp. na reprezentujący ją string:
  - Na etapie kompilacji sprawdzana poprawność identyfikatora.
  - Intellisense podpowiada uzupełnienie.
  - Przydatne w mechanizmie odbicia.

# Operator **is**

- Operator **is** pozwala sprawdzić, czy wyrażenie jest instancją pewnego typu – zwraca wartość logiczna **true**/**false**.
- Ponieważ mechanizm wyjątków jest dużo wolniejszy (do 30.000 razy) niż sprawdzenie prostego warunku logicznego, przy możliwości niepoprawnego rzutowania lepiej sprawdzić taką możliwość operatorem **is** niż łapać wyjątki.

```
static void WrongCastingException()
{
    System.Console.WriteLine("Start: "+nameof(WrongCastingException));
    Stopwatch stopWatch = new Stopwatch();
    stopWatch.Start();
    const int HOWMANY = 1000;
    for (int i = 0; i < HOWMANY; i++)
    {
        Person person = new Person("Kowal", 1999, "all is OK", "agent");
        try
        {
            Worker worker = (Worker)person;
            System.Console.WriteLine(worker);
        }
        catch(InvalidCastException)
        {
            // ignore...
        }
    }
    stopWatch.Stop();
    // Get the elapsed time as a TimeSpan value.
    TimeSpan ts = stopWatch.Elapsed;
    System.Console.WriteLine("Finish: " + ts.TotalSeconds + "s");
}
```



```
static void WrongCastingIs()
{
    System.Console.WriteLine("Start: " + nameof(WrongCastingIs));
    Stopwatch stopWatch = new Stopwatch();
    stopWatch.Start();
    const int HOWMANY = 1000;
    for (int i = 0; i < HOWMANY; i++)
    {
        Person person = new Person("Kowal", 1999, "all is OK", "agent");
        if (person is Worker )
        {
            Worker worker = (Worker)person;
            System.Console.WriteLine(worker);
        }
    }
    stopWatch.Stop();
    // Get the elapsed time as a TimeSpan value.
    TimeSpan ts = stopWatch.Elapsed;
    System.Console.WriteLine("Finish: "+ts.TotalSeconds + "s");
}
```

30.000 razy wolniej

```
Start: WrongCastingIs
Finish: 0,0003508s
Start: WrongCastingException
Finish: 14,9879607s
```

# Klasa **object**

- Jeśli nie podamy po jakiej klasie dziedziczy implementowana klasa, to automatycznie dziedziczy po klasie **object**.
- Klasa **object** posiada kilka metod. Kilka z nich jest wirtualnych, więc warto je zaimplementować inaczej (właściwie) dla swoich klas.
  - `public virtual bool Equals(object o)`
  - `public virtual int GetHashCode()`
  - `public virtual string ToString()`
- Istnieją domyślne implementacja tych metod, najczęściej nie działające zgodnie z oczekiwaniami, gdyż operują na referencji (adresie) a nie zawartości obiektów.

# Własna implementacja metod z **object** - przykład

```
class PersonPure
{
    public string Name { get; set; }
    public int Salary { get; set; }
    public PersonPure(string name, int salary) { Name = name; Salary = salary; }
}

class PersonRich
{
    public string Name { get; set; }
    public int Salary { get; set; }

    public PersonRich(string name, int salary) { Name = name; Salary = salary; }

    public override bool Equals(object obj)
    {
        PersonRich p = obj as PersonRich;
        return p==null?false:this.Name.Equals(p.Name);
    }
    public override int GetHashCode()
    {
        return Name.GetHashCode();
    }
    public override string ToString()
    {
        return $"PersonRich(Name={Name}, Salary={Salary})";
    }
}
```

# Metody object - przykład

```
public static void PersonPureRich()  
{  
    PersonPure pp1 = new PersonPure("Nowak", 2000);  
    PersonPure pp2 = new PersonPure("Nowak", 2000);  
    PersonRich pr1 = new PersonRich("Kowal", 3000);  
    PersonRich pr2 = new PersonRich("Kowal", 5555);  
    System.Console.WriteLine(pp1);  
    System.Console.WriteLine(pr1);  
    System.Console.WriteLine(pp1.Equals(pp2));  
    System.Console.WriteLine(pr1.Equals(pr2));  
    System.Console.WriteLine($"{pp1.GetHashCode()} != {pp2.GetHashCode()}");  
    System.Console.WriteLine($"{pr1.GetHashCode()} == {pr2.GetHashCode()}");  
}
```

```
Inheritance.PersonPure  
PersonRich(Name=Kowal, Salary=3000)  
False  
True  
58225482 != 54267293  
-2125116991 == -2125116991
```

C#

# INTERFEJSY

# Interface

- Interfejs to (trochę w uproszczeniu) określenie zestawu funkcji abstrakcyjnych.
- Klasa w swojej deklaracji może zapewniać implementację danego (jednego lub więcej) interfejsu.
- Deklaracje umieszcza się za dwukropkiem, po przecinkach za nazwą klasy z której dana klasa dziedziczy (lub bezpośrednio po dwukropku, jeśli w sposób niejawni dziedziczy po klasie **object**).
- Jeśli klasa obiecuje implementować interfejs, ale nie napiszemy implementacji wszystkich metod abstrakcyjnych, klasa musi być oznaczona jako abstrakcyjna.
- Jeśli klasa A implementuje interfejs X, to wszystkie klasy dziedziczące po klasie A również muszą implementować ten interfejs (ale już się tego w nagłówku klasy nie zapisuje)

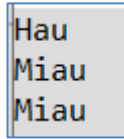
# Interfejs - przykład

```
public interface ISound
{
    string GiveVoice(int howMany);
}
public interface IWater
{
    int HowManyFins { get; set; }
}
public class Cat : ISound
{
    string ISound.GiveVoice(int howMany)
    {
        return "Miau";
    }
}
public class Dog : ISound
{
    string ISound.GiveVoice(int howMany)
    {
        return "Hau";
    }
}
public class Diver : IWater
{
    int IWater.HowManyFins { get ; set; }
}
public class Whale : IWater
{
    public int HowManyFins { get; set; }
}
```

# Interfejs - przykład

```
public static void Sounds(ISound[] arr)
{
    foreach (ISound sound in arr)
        Console.WriteLine(sound.GiveVoice(3));
}

public static void SoundsTest()
{
    ISound[] arr = new ISound[] { new Dog(), new Cat(), new Cat() };
    Sounds(arr);
}
```



Hau  
Miau  
Miau



## Składowe z jawnie/niejawnie podanym interfejsem

- Interfejs można implementować z jawnie podanym interfejsem (klasy `Diver`, `Dog`, `Cat`). Nie może być to wtedy publicznie dostępna składowa.
  - Aby skorzystać z metody takiej implementacji trzeba najpierw **jawnie rzutować** obiekt na interfejs.
- Interfejs można też implementować bez jawnie podanego interfejsu (klasa `Whale`). Wówczas musi to być publiczna metoda.
  - Nie ma wtedy potrzeby rzutować obiekt na odpowiedni interfejs.

```
public static void WaterTest()
{
    Diver diver = new Diver();
    Whale whale = new Whale();
    //diver.HowManyFins = 2; // compilation error
    IWater waterFish = (IWater)diver;
    waterFish.HowManyFins = 2;
    whale.HowManyFins = 4;
}
```

## Dziedziczenie interfejsów (po wielu interfejsach)

- Interfejsy można budować również poprzez dziedziczenie po innych interfejsach.
- W przeciwieństwie do dziedziczenia w ramach klas, interfejs może dziedziczyć po wielu interfejsach.
- Zapis jest analogiczny jak dla dziedziczenia w ramach klas:

```
public interface IWaterLoud: ISound, Iwater
{
    // additional abstract methods
}
```

## Domyślna implementacja (C# 8.0)

- Od C#8.0 istnieje możliwość domyślnej implementacji metody interfejsu.
- Jest ona traktowana jak implementacja z jawnie podanym interfejsem (przykład z klasą `Diver`).
  - Trzeba rzutować na interfejs

```
public interface IShow
{
    void ShowMe() => Console.WriteLine($"My type is {this.GetType().Name}");
}
```

```
public class ClassWithDefaultImpl: IShow
{
}

```

```
public static void TestDefaultImpl()
{
    ClassWithDefaultImpl cl = new ClassWithDefaultImpl();
    (cl as IShow).ShowMe();
}
```

My type is ClassWithDefaultImpl

## Interfejs - Informacje różne

- Interfejsy są często zestawiane z klasami abstrakcyjnymi, ale to tylko zestaw abstrakcyjnych metod
  - Niektóre języki programowania zezwalają na szkielet metod nieabstrakcyjnych oraz tworzenie stałych w ramach interfejsu.
- Zmienna referencyjna o typie interfejsu oznacza referencję na **obiekt** klasy, **która zapewnia implementację tego interfejsu**.
- Operatory **as** i **is** działają również dla interfejsów.
- Interfejs nie dziedziczy po **object**, ani po żadnym ogólnym interfejsie, to nie jest klasa.
  - Klasy tworzą drzewko dziedziczenia z **object** jako korzeń.
  - Interfejsy tworzą graf skierowany (bo interfejs może dziedziczyć po wielu interfejsach)
- Istnieją interfejsy bez metod – służą do szybkiego oznaczania cechy danej klasy (ang. marking/tagging interface) lub jej metod (np. efektywności)
  - Obecnie zaleca się używanie atrybutów (adnotacji)

# **TYPY BEZPOŚREDNIE, RÓŻNICE WZGLĘDEM TYPÓW REFERENCYJNYCH**

# Typy bezpośrednie

- Zmienne typów bezpośrednich przechowują wartość bezpośrednio.
  - Zmienne typów referencyjnych przechowują referencję (jakby adres komórki) w której dopiero są przechowywane wartości
- Wartości te są zatem przechowywane na stosie.
- Można konstruować własne typy bezpośrednie np. poprzez słowo **struct** zamiast **class**

```
class PointClass
{
    public int x;
    public int y;
    public PointClass(int x, int y){ this.x = x; this.y = y; }
    public override string ToString()
    {
        return $"{nameof(PointClass)}({x},{y})";
    }
}
```

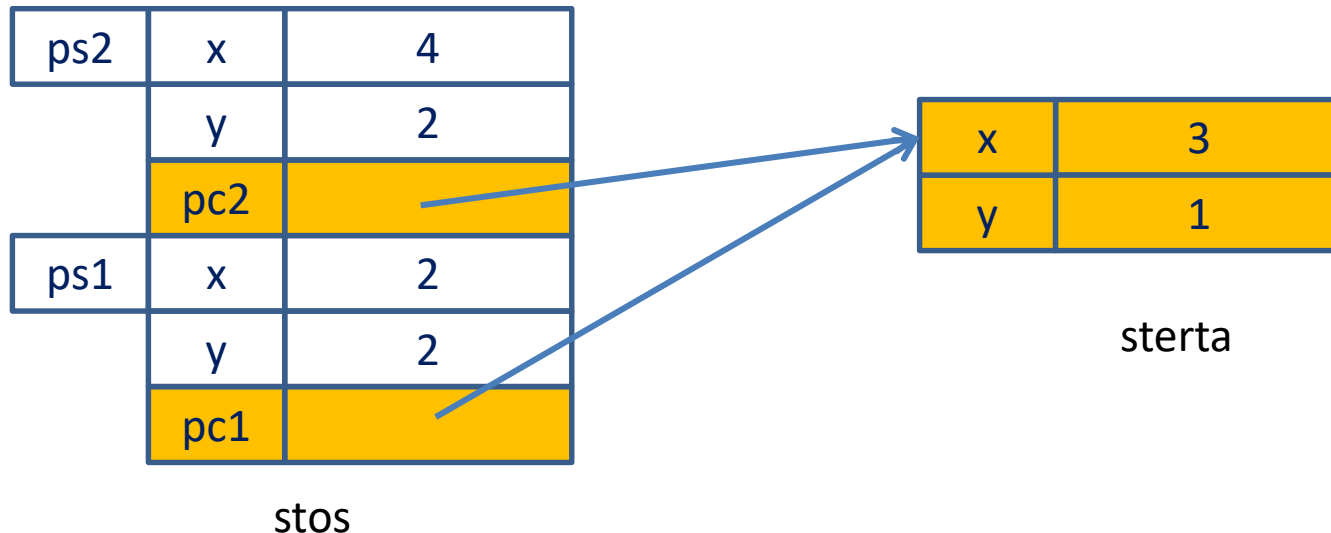
```
struct PointStruct
{
    public int x;
    public int y;
    public PointStruct(int x, int y) ){ this.x = x; this.y = y; }
    public override string ToString()
    {
        return $"{nameof(PointStruct)}({x},{y})";
    }
}
```

# Różnica w działaniu

```
static void DeclarationOfPoints()  
{  
    PointClass pc1= new PointClass(1,1);  
    PointStruct ps1 = new PointStruct(2, 2);  
    PointClass pc2 = pc1;  
    PointStruct ps2 = ps1;  
    pc2.x = 3;  
    ps2.x = 4;  
    System.Console.WriteLine($"{pc1}, {pc2}");  
    System.Console.WriteLine($"{ps1}, {ps2}");  
}
```

PointClass(3,1), PointClass(3,1)  
PointStruct(2,2), PointStruct(4,2)

- Sytuacja po ostatnim podstawieniu



## Typy bezpośrednie/wartościowe

- W języku C# typy bezpośrednie/wartościowe to:
  - Typy proste: **byte**, **int**, **bool**, **float**, **decimal** itp.
  - Typy wyliczeniowe: definiowane jako **enum** NazwaTypu { }
  - Struktury: definiowane jako **struct** NazwaTypu { }
  - Krotki
- Typy referencyjne to:
  - Typy klasowe, w tym **object** i **string**
  - Typy interfejsowe
  - Typy tablicowe
  - Typy delegata
  - Typy null-owalne
    - Opakowanie typów bezpośrednich

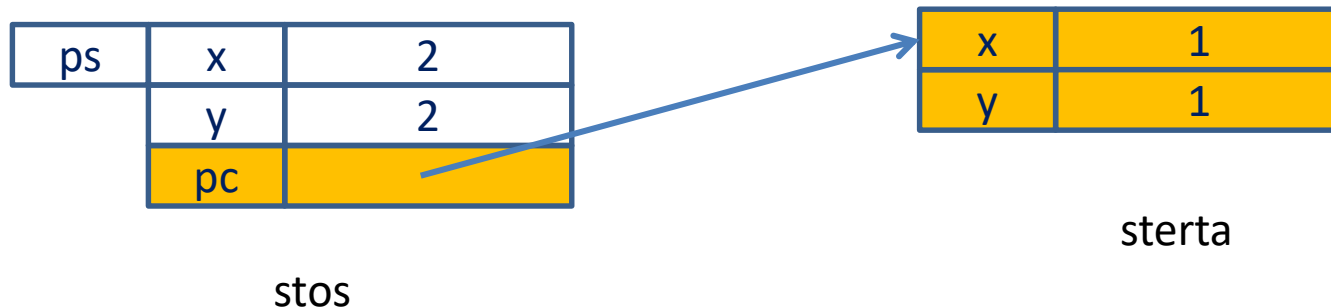


## Struktury

- Struktury to typy bezpośrednie
- Mają ograniczone możliwości klas
- Nie można inicjalizować pól bezpośrednio
- Można tworzyć konstruktory.
- Prawie wszystkie są pochodne od `System.ValueType` (tylko typ wyliczeniowy dziedziczy po `System.Enum`)
- Są niejawnie typami zamkniętymi (**sealed**)
- Dopóki wszystkie pola nie zostaną zainicjowane, nie można używać **this**.
- Struktury mogą implementować interfejsy.
  - Wbudowane typy mają najczęściej zaimplementowane interfejsy `Comparable` oraz `Formattable`.
- Warto przeciążać operatory porównujące i metodę `HashCode()` w celach wydajnościowych.

## Różnica działania jako parametry

- Typy wartościowe i referencyjne inaczej się zachowują przy przekazywaniu ich jako parametry do metody oraz przy zwracaniu jako wynik.
- Analiza zachowania dla klas `PointClass` i `PointStruct` przy przekazywaniu przez wartość, przez referencje i jako parametr wyjściowy.
- Po skończeniu przetwarzania w pamięci zostaną tylko te komórki, które leżą na żółtej „plamie”:
  - Pozostałe komórki ze sterty zostanie odśmieciona, gdy będzie brakować pamięci
  - Pozostałe komórki ze stosu będą uznane za śmieciowe wartości, gdyż stos obniży swoją wysokość.
- Sytuacja przed wykonaniem dwóch pierwszych testów:



## Argumenty przekazywane przez wartość - st1

- Metoda otrzymuje kopie referencji oraz wartości bezpośrednich

```
static void TestWithArgs(PointClass pcArg,  
                        PointStruct psArg)  
{  
    pcArg.x = 11;  
    psArg.x = 12; // st 1  
    pcArg = new PointClass(100, 100); //st 2  
    psArg = new PointStruct(200, 200); //st 3  
}
```

```
static void TestArgByValue()  
{  
    PointClass pc = new PointClass(1, 1);  
    PointStruct ps = new PointStruct(2, 2);  
    TestWithArgs(pc, ps);  
    System.Console.WriteLine($"{pc}");  
    System.Console.WriteLine($"{ps}");  
}
```

psArg	x	12
	y	2
	pcArg	

ps	x	2
	y	2
	pc	

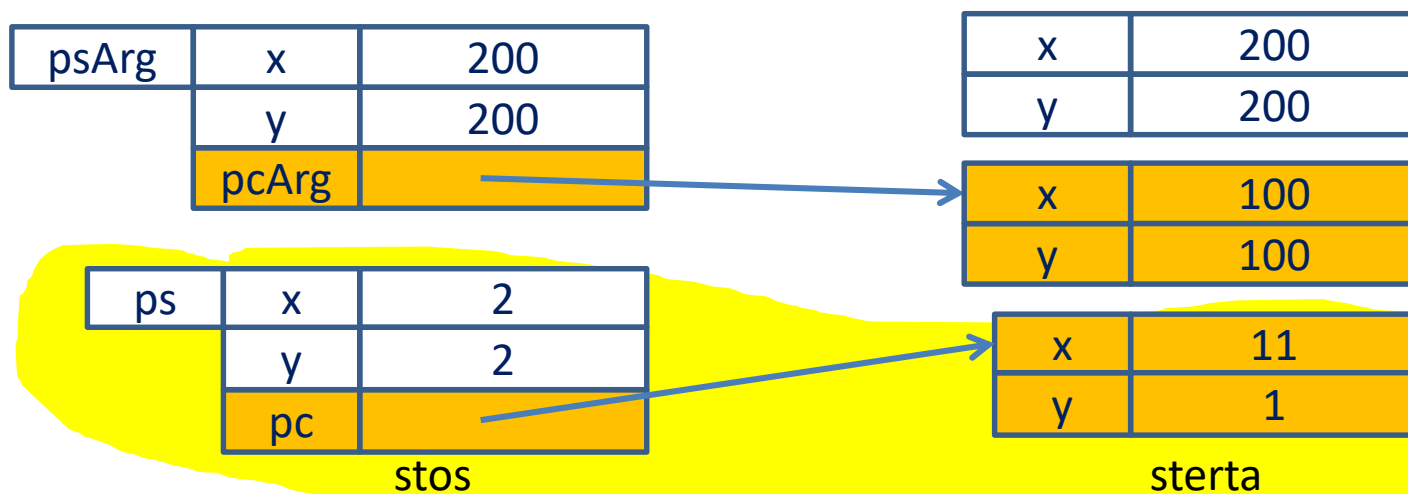
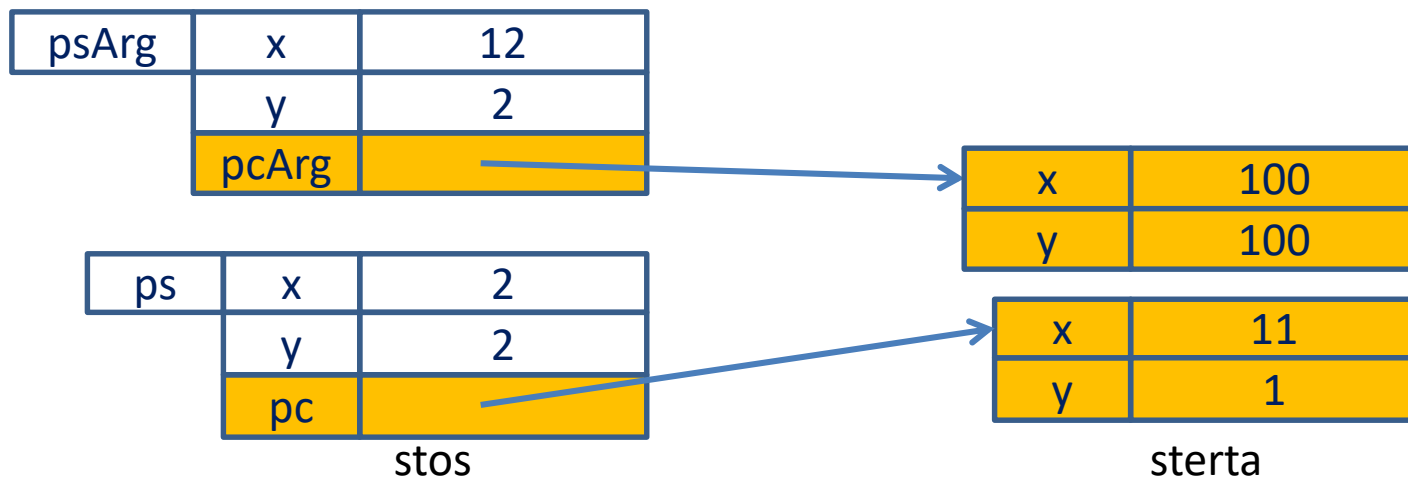
stos

x	11
y	1

sterta

PointClass(11,1)  
PointStruct(2,2)

## Argumenty przekazywane przez wartość – st2 i st3



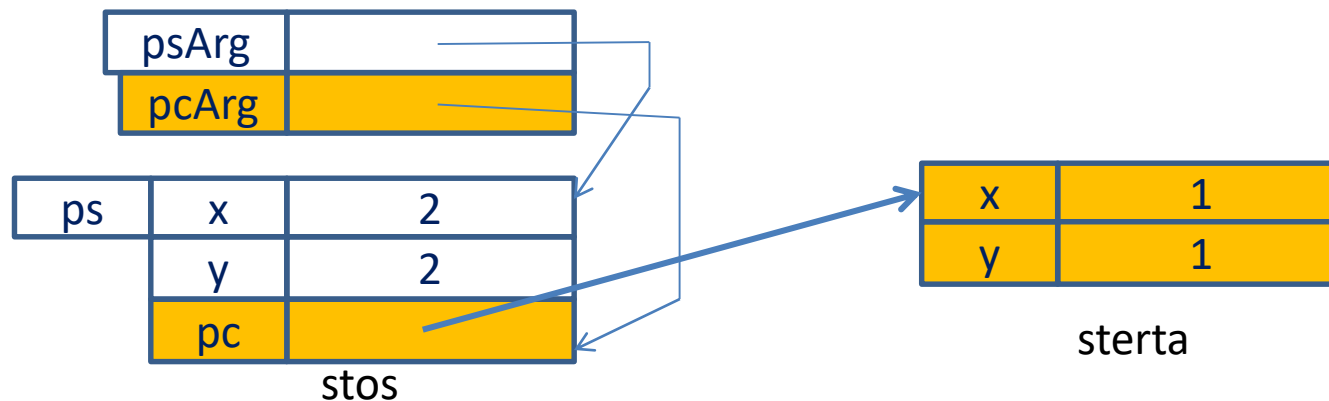
## Argumenty przekazywane przez referencję - st1

- Metoda otrzymuje referencję na referencję oraz referencję na wartości bezpośrednie.

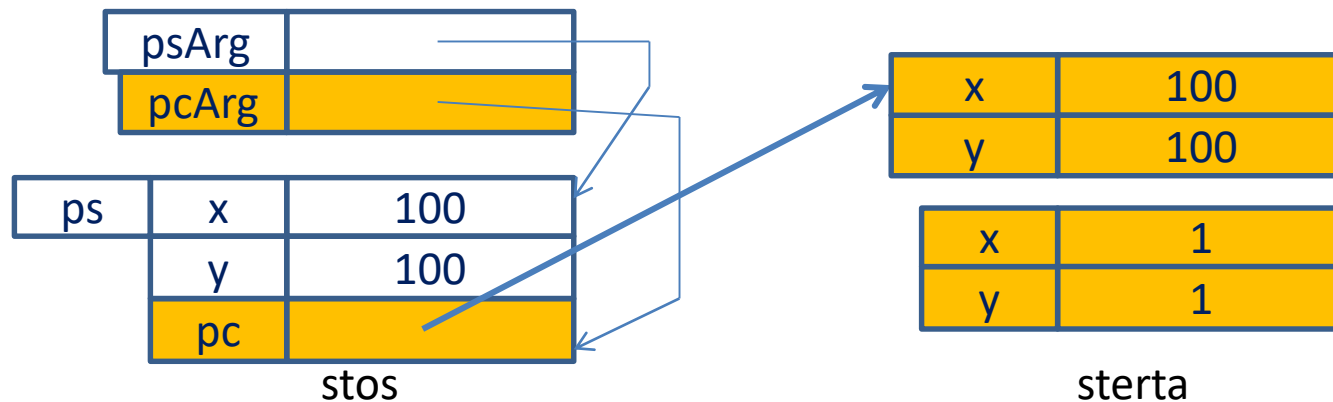
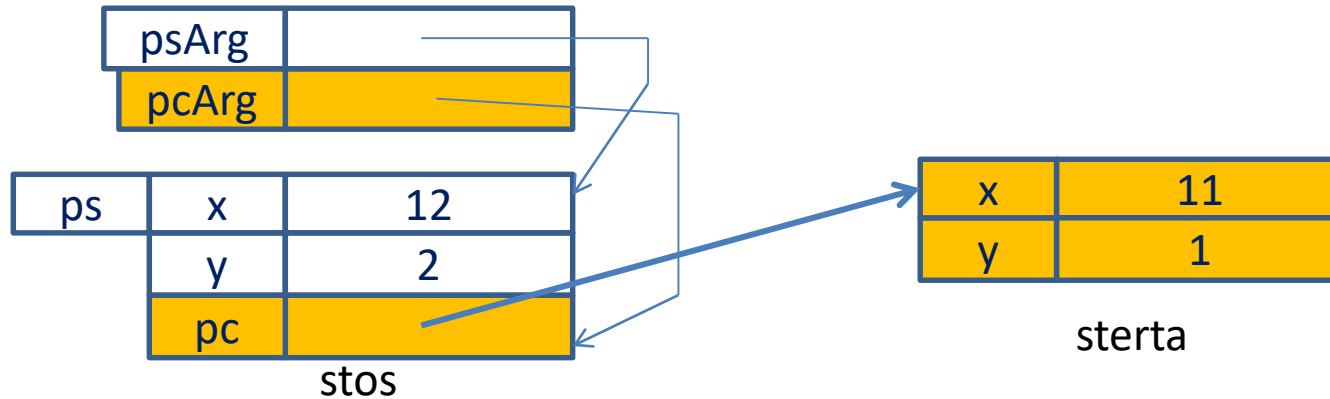
```
static void TestWithRef(ref PointClass pcArg,  
                        ref PointStruct psArg)  
{  
    // st1  
    pcArg.x = 11;  
    psArg.x = 12; // st2  
    pcArg = new PointClass(100, 100); //st3  
    psArg = new PointStruct(200, 200); //st4  
}
```

```
static void TestArgByRef()  
{  
    PointClass pc = new PointClass(1, 1);  
    PointStruct ps = new PointStruct(2, 2);  
    TestWithArgs(ref pc, ref ps);  
    System.Console.WriteLine($"{pc}");  
    System.Console.WriteLine($"{ps}");  
}
```

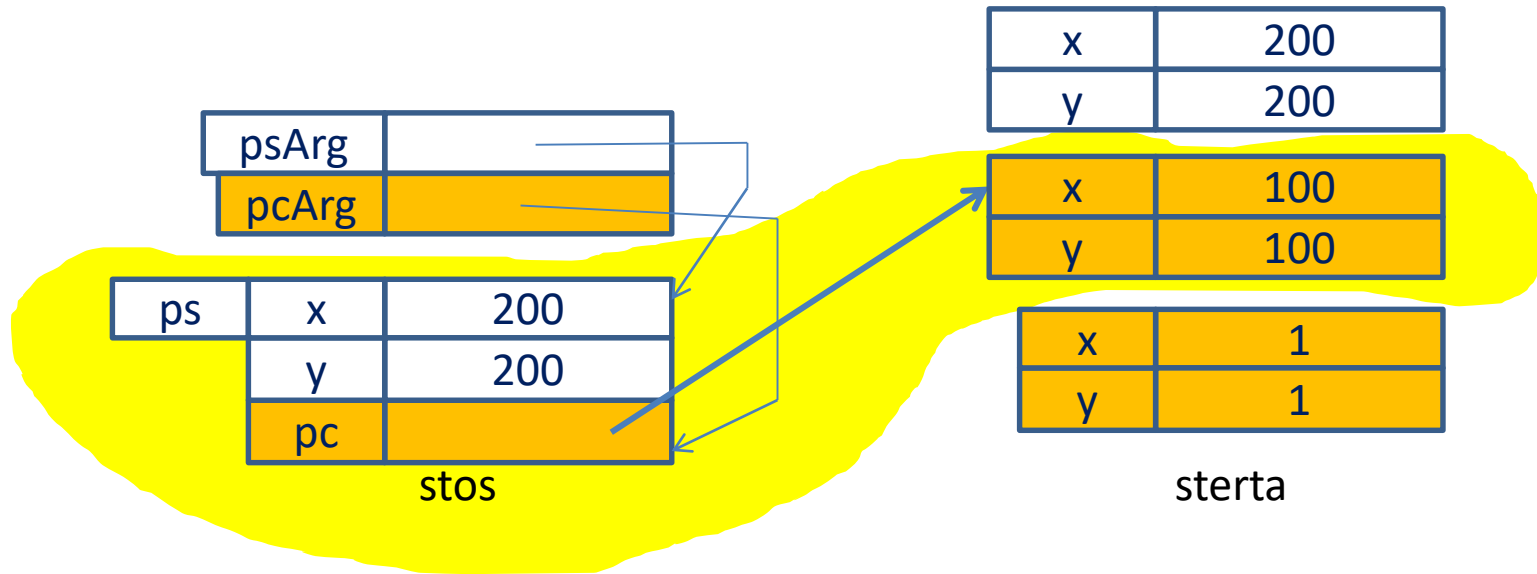
PointClass(100,100)  
PointStruct(200,200)



## Argumenty przekazywane przez referencję – st2 i st3



## Argumenty przekazywane przez referencję - st4



## Argumenty przekazywane jako wyjściowe – st1

- Metoda również otrzymuje referencję na referencję oraz referencję na wartości bezpośrednie.

```
static void TestWithout(out PointClass pcArg, out PointStruct psArg)
{
    //st1
    //pcArg.x = 11; // błąd kompilacji - parametr pc może być niezainicjowany
    //psArg.x++; błąd kompilacji, pole x struktury ps może być niezainicjowane
    psArg.x = 12; //st2, to jest typ wartościowy, pole ps musi "istnieć", jak i cała struktura ps
    pcArg = new PointClass(100, 100); //st3
    psArg = new PointStruct(200, 200); //st4, nadpisana wartość 12 w polu x
}
```

```
static void TestArgByOut()
{
    PointClass pc;
    PointStruct ps;
    TestWithout(out pc, out ps);
    System.Console.WriteLine($"PointClass: {pc.x} {pc.y}");
    System.Console.WriteLine($"PointStruct: {ps.x} {ps.y}");
}
```

PointClass(100,100)  
PointStruct(200,200)

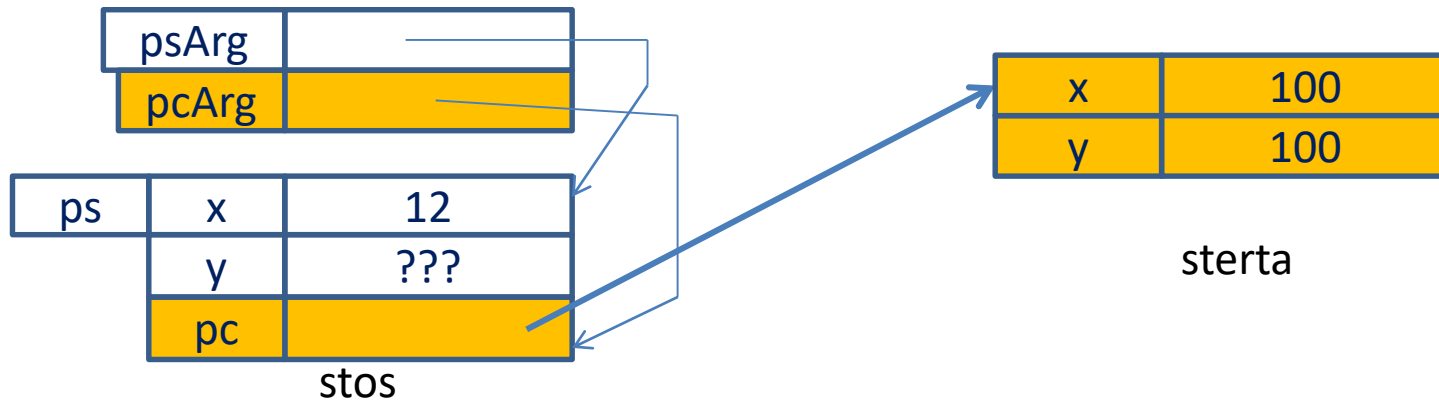
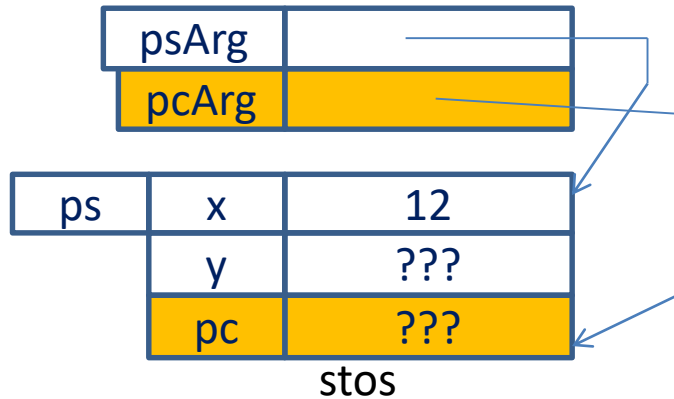
- st1

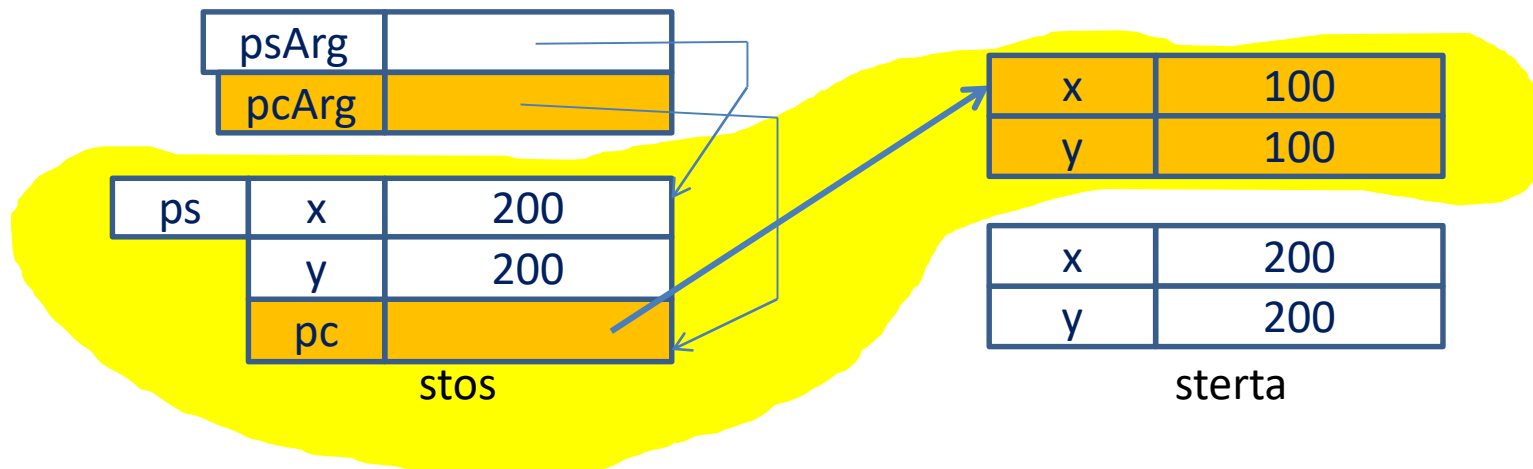
ps	x	???
	y	???
	pc	???

stos



## Argumenty przekazywane jako wyjściowe – st2 i st3





- Od C# 9.0 istnieje też pojęcie typu rekordu (`record`, `record struct` lub `record class`)
  - Podobne do `struct`, ale ma wbudowane operacje typu porównywania, dekonstrukcji itd.
  - Powinien być używany do wartości niezmiennych po stworzeniu
  - Posiada specjalny operator `with`

## Typy bezpośrednie - informacje różne

- Nie twórz typów bezpośrednich większych niż 16 bajtów:
  - Przekazywanie referencji jako parametrów jest szybsze.
  - Przekazywanie zmiennych typów bezpośrednich przez referencję często powoduje pakowanie ich do tymczasowych obiektów, czyli tworzenie zmiennych referencyjnych, a następnie wypakowywanie przy powrocie (np. gdy mają implementację interfejsów).
  - Dla przekazywania dużych typów bezpośrednich, których wartość nie będzie modyfikowana w metodzie przydatny będzie modyfikator parametrów **in**.
    - Do metody będzie przekazane coś w rodzaju referencji do adresu na stosie (każdy dostęp w metodzie do pola takiej struktury wymaga adresacji pośredniej w kodzie assemblera, ale za to nie trzeba kopiować całej struktury podczas wywoływania metody)
- Zalecane jest używanie typów bezpośrednich niemodyfikowalnych (we właściwościach jest tylko **get**).
  - Typ bezpośredni jest nazywany też typem wartościowym.
  - Zamiast modyfikacji można zawsze stworzyć nową strukturę z nowymi wartościami pól (podejście jak dla klasy **string**).

# Typy wyliczeniowe

- **enum** <nazwaTypu> {...}
- **enum** <nazwaTypu>:<typBazowy> {...}
- W klamrach kolejne identyfikatory stałych całkowitoliczbowych
  - Standardowo zaczynają się od zera i kolejne co 1
  - Można poprzez znak '=' nadać inną wartość (kolejne nieprzypisane wartości będą inkrementowane co 1)
  - Można wielu identyfikatorom nadać te same wartości
  - Istnieje atrybut [Flag], że typ wyliczeniowy jest bitowa flagą. Zmienia to działanie metody ToString() rozbijając argument do wyświetlenia na bity i wyświetlająca wszystkie identyfikatory z bitem 1.
  - Standardowym typem bazowym jest **int**.

```
enum State: short {  
    NotKnown,           // 0  
    New,                 // 1  
    PreRun=10,          // 10  
    Running,            // 11  
    Executing=Running,  // 11  
    Finishing,          // 12  
    Finished,           // 13  
    Dead=20+2*5+New     // 31  
}
```

```
[Flags] enum Direction  
{  
    None,  
    Up = 1,  
    Down = 2,  
    Left = 4,  
    Right = 8  
}
```

# Typ enum – zastosowanie i możliwości

- Zamiast wielu stałych typu `int`
- Zebrane w typ nie mieszają się z innymi stałymi typu `int` (`State.NotKnown=0` oraz `Direction.None=0`, ale to dwa różne typy)
- Można rzutować na `int` i odwrotnie.

```
public static void TestEnumValue(State state)
{
    switch (state)
    {
        case State.NotKnown:
            System.Console.WriteLine("?!");
            break;
        case State.New:
            System.Console.WriteLine(state);
            break;
        case State.Dead:
            System.Console.WriteLine((int)state);
            break;
        default:
            System.Console.WriteLine($"default block for: {state}");
            break;
    }
}
```

```
New
?!
31
default block for: Running
default block for: 100
False
Flags: Up, Down
Flags: Up, Down, Left, Right
Flags: 100
```

```
public static void EnumTest()
{
    State state = State.New;
    TestEnumValue(state);
    TestEnumValue(State.NotKnown);
    TestEnumValue(State.Dead);
    TestEnumValue((State)11);
    TestEnumValue((State)100); // nie będzie błędu ani wyjątku
    System.Console.WriteLine(System.Enum.IsDefined(typeof(State), (short)100));
    System.Console.WriteLine($"Flags: {Direction.Down | Direction.Up}");
    System.Console.WriteLine($"Flags: {(Direction)15}");
    System.Console.WriteLine($"Flags: {(Direction)100}");
}
```

C#

# WYJĄTKI

# Mechanizm wyjątków

- Mechanizm wyjątków służy do zakończenia metody (rzadziej bloku kodu) nie poprzez zwrócenie wyniku (lub sterowania poprzez return).
- Wyjątkiem może być obiekt klasy `System.Exception` (lub jego pochodne)
- Rzucanie wyjątku:
  - **throw** <objectException>
- Można tworzyć własne wyjątki:

```
public class MyTestException : Exception
{
}
}
```

## Blok **try/catch/finally**

- Obsługa wyjątku następuje w bloku **try/catch/finally**.

```
public static void TestException(int arg)
{
    System.Console.WriteLine("Before try/catch/finally");
    try
    {
        System.Console.WriteLine("Before method call");
        MakeException(arg);
        System.Console.WriteLine("After method call");
    }
    catch (IOException)
    {
        System.Console.WriteLine("Problem with IO");
        System.Console.WriteLine("Exception solved !");
    }
    catch (ArgumentOutOfRangeException exp)
    {
        System.Console.WriteLine("Problem "+exp);
        System.Console.WriteLine("Dont know how to solve");
        throw; // rethrow exception
    }
    finally
    {
        System.Console.WriteLine("Anyway i will to this!");
    }
    System.Console.WriteLine("After try/catch/finally");
}
```



## Działanie bloku **try/catch/finally**

- Działanie (ogólnie):
  - Jeśli nie wystąpi wyjątek, to blok **try** wykona się do końca, a następnie (jeśli jest) wykona się kod bloku **finally**
  - Jeśli nastąpi wyjątek to:
    - Jeśli istnieje blok **catch** zgodny co do tego wyjątku, nastąpi przejście do wykonania tego bloku (pierwszego, który pasuje) a następnie (jeśli jest) do wykonania bloku **finally**
    - Jeśli taki nie istnieje, wykona się blok **finally** a następnie wykonanie zakończy się rzuceniem wyjątku.
- W ramach bloku **catch** można obsługiwany wyjątek rzucić mimo wszystko wyżej (gdy po podjęciu próby rozwiązania problemu nadal występuje)
  - **throw;** // bez argumentu, na wyższym poziomie zostanie zachowana informacja o oryginalnym miejscu wystąpienia
- Blok **catch** bez nawiasów z argumentem jest ogólnym blokiem **catch**.
  - Bardziej przydatny przy wykorzystaniu oprogramowania natywnego.

# Testowanie wyjątków - kody

```
public static void TestWholeCases()
{
    System.Console.WriteLine("----- case 0");
    TestException(0);
    System.Console.WriteLine("----- case 1");
    TestException(1);
    System.Console.WriteLine("----- case 2");
    try
    {
        TestException(2);
    }
    catch
    {
        System.Console.WriteLine("unhandled exception");
    }
    System.Console.WriteLine("----- case 3");
    try
    {
        TestException(3);
    }
    catch
    {
        System.Console.WriteLine("unhandled exception");
    }
}
```

```
public static void MakeException(int arg)
{
    if (arg == 0)
        return;
    if (arg == 1)
        throw new IOException();
    if (arg == 2)
        throw new ArgumentOutOfRangeException();
    if (arg == 3)
        throw new MyTestException();
}
```

# Testowanie wyjątków - wynik

```
----- case 0
Before try/catch/finally
Before method call
After method call
Anyway i will to this!
After try/catch/finally
```

```
----- case 1
Before try/catch/finally
Before method call
Problem with IO
Exception solved !
Anyway i will to this!
After try/catch/finally
```

```
----- case 3
Before try/catch/finally
Before method call
Anyway i will to this!
unhandled exception
```

```
----- case 2
Before try/catch/finally
Before method call
Problem System.ArgumentOutOfRangeException: Specified argument was out of the range of valid values.
   at Exceptions.Program.MakeException(Int32 arg) in C:\Users\dariu\source\repos\Exceptions\Exceptions\Program.cs:line 24
   at Exceptions.Program.TestException(Int32 arg) in C:\Users\dariu\source\repos\Exceptions\Exceptions\Program.cs:line 34
Dont know how to solve
Anyway i will to this!
unhandled exception
```

# Wyjątki – czyszczenie stosu

- Wyjątki nie są prostym mechanizmem:
  - Co jeśli w bloku **catch** nastąpi nowy wyjątek? Który powinien być wysłany wyżej? A może oba?
  - Co jeśli w bloku **finally** wystąpi nowy wyjątek, gdy wykonujemy **finally** po pojawieniu się nieobsłużonego wyjątku?
- Konstruując wyjątek można podać np. komentarz (**string**), ale również inny wyjątek jako źródło tego właśnie tworzonego (aby wyższa metoda podjęła działanie)
- Ponieważ wyjście z metody z wyjątkiem zaburza mechanizm wykorzystania stosu do komunikacji z instrukcją wywołania rozkazu (przekazanie argumentów, odebranie wyników i przechowywanie zmiennych lokalnych) należy stos poprawnie uporządkować.
  - W językach typu C++ trzeba wręcz *odwikłać stos*.
- Standardowo przekroczenie zakresów typów liczbowych nie generuje wyjątków (kod w języku pośrednim jest prostszy i szybszy). Można włączyć mechanizm wyjątków umieszczając niepewny kod w bloku ze słowem **checked**.

```
public static void TestNotChecked() {  
    int n = int.MaxValue;  
    n = n + 1;  
    System.Console.WriteLine(n);  
}
```

-2147483648

```
public static void TestChecked() {  
    checked {  
        int n = int.MaxValue;  
        n = n + 1;  
        System.Console.WriteLine(n);  
    }  
}
```

Unhandled exception. System.OverflowException  
at Exceptions.Program.TestChecked() in  
at Exceptions.Program.Main() in C:\Use

# Mechanizm wyjątków - szybkość

- Mechanizm wyjątków jest wolny, nie używaj go, gdy problem można łatwo rozwiązać bez niego.

```
public static int TestArrNormal()  
{  
    int[] arr = new int[] {1,2};  
    int sum = 0;  
    for (int i = 0; i < arr.Length; i++)  
        sum += arr[i];  
    return sum;  
}
```

```
public static void TestTime(int howMany, Func<int> action)  
{  
    Stopwatch sw = new Stopwatch();  
    sw.Start();  
    int global = 0;  
    for (int i = 0; i < howMany; i++)  
        global+=action();  
    sw.Stop();  
    Console.WriteLine($"result={global}, time elapsed={sw.Elapsed}.");  
}
```

```
public static void TestTime()  
{  
    int howMany = 1000;  
    TestTime(howMany, TestArrNormal);  
    TestTime(howMany, TestArrException);  
}
```

```
public static int TestArrException()  
{  
    int[] arr = new int[] { 1, 2};  
    int sum = 0;  
    try  
    {  
        for (int i = 0; ; i++)  
            sum += arr[i];  
    }  
    catch(IndexOutOfRangeException )  
    {  
    }  
    return sum;  
}
```

```
result=3000, time elapsed=00:00:00.0001698.  
result=3000, time elapsed=00:00:15.0700570.
```

88.000 razy wolniej

?

- **using** (zasób) { }
- **using** (zasób) ;

## Reguły korzystania z wyjątków

- Nie generuj wyjątków w konstruktorach
- Nie generuj wyjątków w finalizatorach
- Przechwytuj tylko te wyjątki, które potrafisz obsłużyć
- Nie ukrywaj/ignoruj wyjątków, które nie są w pełni obsługiwane
- Rzadko korzystaj z ogólnego bloku `catch` lub wyłapywania `System.Exception`
- Unikaj informowania/rejestrowania o wyjątkach na niskich poziomach wywołań
- Kolejne reguły – już bardziej szczególne przypadki

## Wyjątki - informacje różne

- W przeciwieństwie do Javy, w języku C# nie podaje się, jakie wyjątki może zgłosić dana metoda
  - w Javie to ograniczenie zgłaszania wyjątków przez metodę i tak można obejść przez późne wiązanie z mechanizmem refleksji.



C#

# GENERYCZNOŚĆ

# Klasy generyczne

- Często w programowaniu jest potrzeba posiadania pewnej struktury danych, ale typ pamiętanych wewnętrznie danych zmienia się w zależności od zastosowania:
  - Stos `int`-ów, stos obiektów klasy `Person`, stos obiektów klasy `Frame` itd.
- Aby nie używać złej techniki kopiuj-wklej-zamień i pisać wiele implementacji stosu dla różnych typów można napisać jedną implementację dla najbardziej ogólnej klasy **`object`**
- Ma to jednak dwie wady:
  - Podczas „wyjmowania” obiektów z takiej kolekcji trzeba zrobić rzutowanie
  - Można wstawić do kolekcji inny obiekt niż oczekiwano (do stosu obiektów klasy `Person` wstawić obiekt klasy `Frame`).
- Rozwiązaniem jest zastosowanie klas **generycznych**, które jako jeden (lub więcej) parametr deklaracji klasy mają nieokreślony typ klasowy.
- Typ generyczny, oprócz nazwy klasy generycznej posiada między znakami ‘<’ i ‘>’ nazwę typu będącego parametrem tej klasy:  
`klasaGeneryczna<typKlasy>`. Np.
  - `Stack<int>`
  - `Stack<Person>`
  - `Stack<Frame>`
- Parametr klasy generycznej używa się jak znany typ (w deklaracjach pól, właściwości, parametrów i zmiennych lokalnych)
- Klasa generyczna może być użyta wewnątrz innej klasy generycznej np. w deklaracji typu:
  - `Stack<Stack<Frame>>`, stos stosów ramek

# Typ generyczny – użycie - przykład

- `Stack<Type>` - stos elementów typu `Type`.

```
using System.Collections.Generic;
using System.Drawing;
class Program
{
    public static void StackTest()
    {
        Stack<int> stackInt = new Stack<int>();
        Stack<Point> stackPoint = new Stack<Point>();
        Stack<object> stackObject = new Stack<object>();
        stackInt.Push(4);
        stackInt.Push(5);
        stackInt.Push(6);
        Console.WriteLine(stackInt.Pop());
        Console.WriteLine(stackInt.Pop());
        Console.WriteLine(stackInt.Pop());
        stackPoint.Push(new Point(2, 3));
        stackPoint.Push(new Point());
        stackPoint.Push(new Point(1, 1));
        Point p1 = stackPoint.Pop();
        Console.WriteLine(p1);
        Console.WriteLine(stackPoint.Pop());
        Console.WriteLine(stackPoint.Pop());
        stackObject.Push(new Point(2, 3));
        stackObject.Push(2);
        stackObject.Push(new Point());
        p1 = (Point)stackObject.Pop(); // musi być rzutowanie
        int x=(int)stackObject.Pop();
        x=(int)stackObject.Pop(); // będzie wyjątek, tam jest Punkt nie int
    }
}
```

```
6
5
4
{X=1,Y=1}
{X=0,Y=0}
{X=2,Y=3}
Unhandled exception. System.InvalidCastException: Un-
nt32'.
   at GenericType.Program.StackTest() in C:\Users\d
   at GenericType.Program.Main(String[] args) in C:
```

# Własna klasa generyczna - przykład

```
public class Pair<T,S>
{
    public T First { get; set; }
    public S Second { get; set; }
    public Pair(T first, S second)
    {
        First = first;
        Second = second;
    }
    public void set(T first, S second)
    {
        First = first;
        Second = second;
    }
    public override string ToString()
    {
        return $"({First},{Second})";
    }
}
```

```
(1,one)
(2,two)
(1,one)
0
```

```
public static void PairTest()
{
    Pair<int, string> para = new Pair<int, string>(2,"one");
    para.First = 1;
    Console.WriteLine(para);
    Stack<Pair<int, string>> pairStack = new Stack<Pair<int, string>>();
    pairStack.Push(para);
    pairStack.Push(new Pair<int, string>(2, "two"));
    Console.WriteLine(pairStack.Pop());
    Console.WriteLine(pairStack.Pop());
    Console.WriteLine(pairStack.Count);
}
```

## Klasy generyczne – ograniczenia na parametr

- Po deklaracji klasy mogą nastąpić ograniczenia na typ parametru. Przed klamrą za pomocą słowa kluczowego **where** dopisuje się ograniczenia. Np. Klasa `Parking` ma działać tylko na obiektach zapewniających implementację interfejsu `IVehicle`:

```
class Parking<T> where T:IVehicle {...}
```

Wstawiając ograniczenie do konkretnej klasy oznacza tą klasę lub jej pochodne

- Można wymusić użycie tylko klas z domyślnym konstruktorem
- Można wymusić, że można używać tylko struktury lub tylko klasy itp.
- Można nałożyć kilka ograniczeń, oddzielonych przecinkiem.
- Jeśli jest więcej parametrów klasy generycznej to warunki na kolejne typy parametryzujące oddziela się białym znakiem
- Przy większej liczbie parametrów można je powiązać warunkami.
- Ograniczenie tego typu można również użyć w interfejsach i metodach generycznych

```
class Base { }  
class Test<T, U>  
    where U : struct  
    where T : Base, new()  
{ }
```

```
//Type parameter V is used as a type constraint.  
public class SampleClass<T, U, V> where T : V { }
```

## Klasa generyczna `Nullable<T>` i typy nullowalne

- Specjalny zapis pozwala zmiennym typu bezpośredniego/wartościowego nadać wartość **null**. Poprzez dopisanie '?' na końcu typu:  
**int?** x=**null**;
- W rzeczywistości tworzy się zmienną typu generycznego `Nullable<T>` np.:  
`Nullable<int>` x=**null**;
- Kompilator w momencie używania takiej zmiennej będzie wstawiał uruchomienie odpowiednich właściwości `Value` oraz `HasValue`.
  - Powoduje to pewien dodatkowy narzut czasowy
- Z kodu poniżej widać, że można tą strukturę używać tylko do typów wartościowych
- W C# 8.0 są **nullowalne typy referencyjne**:
  - W zasadzie na odwrót: na poziomie kompilacji możemy dopuszczać lub nie wartość **null** w referencjach
  - Pozwala to już na etapie kompilacji zapewnić, że zmienna referencyjna nie będzie miała wartości **null**.

```
public struct Nullable<T> where T : struct{  
  
    public bool HasValue { get; private set; }  
    public T Value { get; private set; }  
    ...  
}
```

## Określanie wartości domyślnej

- W przypadku pól/właściwości typu klasy parametrycznej można użyć wartości domyślnej danego typu.
- Słowo kluczowe: **default**
- Szczegóły – typy danych
- Przykład stworzenia konstruktora domyślnego dla klasy `Pair<T, S>`:

```
public class Pair<T,S>
{
    ...
    public Pair()
    {
        First = default;
        Second = default;
    }
    ...
}
```

# Interfejs generyczny

- Deklaruje się analogicznie jak klasę generyczną.
- Te same możliwości dopisania ograniczeń
- Implementacja interfejsu generycznego może być generyczna lub niegeneryczna
- Interfejsy generyczne też mogą mieć domyślną implementację metod (od C# 8.0)

```
public interface IPair<T, S>
{
    public T First { get; set; }
    public S Second { get; set; }
    public void Set(T first, S second);
}
```

```
public class GenPair<T, S> : IPair<T, S>
{
    public T First { get; set; }
    public S Second { get; set; }
    public GenPair(T first, S second)
    {
        First = first;
        Second = second;
    }
    public void Set(T first, S second)
    {
        First = first;
        Second = second;
    }
    public override string ToString()
    {
        return $"({First},{Second})";
    }
}
```

```
public class PairIntString : IPair<int, string>
{
    public int First { get; set; }
    public string Second { get; set; }
    public PairIntString(int first, string second)
    {
        First = first;
        Second = second;
    }
    public void Set(int first, string second)
    {
        First = first;
        Second = second;
    }
    public override string ToString()
    {
        return $"({First},{Second})";
    }
}
```



## Wewnętrzne klasy generyczne

- Klasa wewnętrzna może być generyczna
- Wewnętrzna klasa generyczna ma swoje własne parametry
  - Nawet jeśli mają takie same nazwy jak klasa zewnętrzna
  - Może to być błąd logiczny powiązań między klasami

```
public class List<T>{  
  
    class Element<T>{  
        T value;  
        Element<T> next;  
    }  
  
    Element<T> head;  
    ...  
}
```

- Klasa `Element` ma w sumie dwa parametry generyczne: `T` z deklaracji klasy `List` (przesłonięty) oraz `T` z deklaracji klasy `Element`
- Pole `head` korzysta z tego pierwszego

```
public class List<T>{  
  
    class Element{  
        T value;  
        Element next;  
    }  
  
    Element head;  
    ...  
}
```

- Klasa `Element` ma dostęp do `T` z deklaracji klasy `List`

## Zalety klas/interfejsów generycznych

1. Bezpieczeństwo ze względu na typ
  - Nie można użyć innego typu niż podany (lub jego pochodny) podczas tworzenia obiektu klasy generycznej.
2. Sprawdzanie typów na etapie kompilacji
  - Zmniejszenie częstości wyjątków `InvalidCastException`
3. Typy bezpośrednie nie są opakowywane
  - Dla każdego typu wartościowego jest generowana oddzielna kompilacja typu generycznego (w Javie jest użycie klasy opakowującej)
  - Dla typów referencyjnych jeden kod z miejscem pamiętającym dla jakiego typu uruchamiany jest kod
4. Zmniejszenie ilości kodu
5. Wzrost wydajności (z powodu 2 i 3)
6. Użycie mniej pamięci (z powodu 3)
7. Czytelność kodu wzrasta (z powodu 4)
8. Intellisense wykrywa typy składowe obiektów generycznych i odpowiednio podpowiada możliwe operacje.

## Metody generyczne

- W ramach zwykłych klas można napisać metody generyczne
- Tworzenie: dodanie oznaczenia typów parametryzujących po nazwie metody.
- Dodanie ograniczeń na parametr: za nawiasem zamykającym zwykłe parametry metody.

```
public static T Max<T>(T[] tab) where T : IComparable<T>
{
    T max = tab[0];
    for (int i = 1; i < tab.Length; i++)
    {
        if (tab[i].CompareTo(max) > 0)
            max = tab[i];
    }
    return max;
}
```

- W ramach klas/interfejsów generycznych wszystkie metody mogą używać ich parametrów generycznych
  - Można też napisać metodę generyczną z własnymi typami parametryzowanymi, uważać wtedy na przesłanianie nazw typów (problem jak w klasach wewnętrznych)