.NET
Klasy, obiekty

Klasa i obiekt

- Każdy obiektowy język programowania daje programiście możliwość tworzenia nowych typów danych. Nowy typ danych definiuje się poprzez zdefiniowanie klasy.

Co to jest klasa?

Z filozoficznego punktu widzenia klasa to rodzaj klasyfikacji, czyli próby zdefiniowania cech wspólnych określonego obiektu (opis czegoś, co istnieje).

- Z programistycznego punktu widzenia klasa stanowi typ danych, który odwzorowuje wspólne cechy jakiegoś obiektu.

Co to jest obiekt

- Obiekt jest instancją klasy, czyli rzeczywistym tworem o określonych cechach i zachowaniu (coś, co istnieje). Przykładowo weźmy pod uwagę kota. Kot jest obiektem (jest czymś, co istnieje), natomiast jego opis znajdujący się np.: w encyklopedii jest klasą, (czyli opisem istniejącego obiektu).

- Każdy obiekt charakteryzuje jego:

  - **niepowtarzalność** - obiekty różnią się od siebie, przez co są niepowtarzalne (np. dwa koty należące do tej samej rasy nie są identyczne);

  - **zachowanie** - każdy obiekt może wykonywać zestaw określonych czynności (np. kot: miauczy, łasi się, etc);

  - **stan** - każdy obiekt przechowuje informacje o swoim stanie, informacja ta wpływa na jego zachowanie (np.: jeżeli kot jest głodny jego zachowanie się zmieni: będzie szukał jedzenia, będzie się upominał, etc, jeżeli zaspokoi głód, jego zachowanie znów się zmieni: będzie leżał, będzie mruczeć, etc.)

Definiowanie klas

- Składnia definicji klasy jest następująca:

[modyfikatory] class Identyfikator [lista_bazowa]
{
Składniki_klasy
}

```
class Kot
{
 // składowe klasy
}
```

Definiowanie klas cd.

[modyfikatory] class Identyfikator [lista_bazowa]

- ***modyfikatory***
  dozwolone są modyfikatory: *abstract, sealed* oraz cztery modyfikatory dostępu (opcjonalne),

- ***Identyfikator***
  nazwa klasy (wymagane),

- ***lista _bazowa***
  lista zawierająca nazwę jednej klasy bazowej oraz nazwy interfejsów oddzielone przecinkami (opcjonalne)

- ***składniki_klasy***
  deklaracja składowych klasy

Wewnątrz klas można zagnieżdżać inne klasy

```
class Samochod
{
 class Kolo
 {
  class Opona
  {
  }
 }
}
```

Modyfikatory

- Modyfikatory służą do zmiany zachowania typów pól składowych klasy.
- W języku C# można wyróżnić następujące modyfikatory:
  - ***abstract -*** określa, że klasa jest abstrakcyjna (nie można utworzyć instancji takiej klasy), klasa abstrakcyjna może być bazową dla innej klasy;
  - ***const -*** określa, że wartość składowej nie może być zmieniana;
  - ***event*** - do deklarowania zdarzeń;
  - ***override -*** służy nadpisywaniu metod wirtualnych odziedziczonych z klasy bazowej;
  - ***readonly -*** określa, że polu można przypisywać wartości jedynie w deklaracji oraz konstruktorze tej samej klasy;

Modyfikatory cd.

W języku C# można wyróżnić następujące modyfikatory cd.:

- ***sealed -*** określa, że klasa nie może być bazową klasą dla żadnej innej (stanowi klasę ostateczną);
- ***static -*** określa, że składnik klasy należy do typu a nie do konkretnego obiektu;
- ***virtual-*** służy do deklaracji metod wirtualnych, czyli metod, które mogą być nadpisywane w klasach dziedziczących z klasy, w której się znajdują;
- ***volatile -*** określa, że pole może zostać zmodyfikowane przez system operacyjny, urządzenie lub inny wątek,

oraz modyfikatory dostępu opisane dalej...

Modyfikatory dostępu

- Modyfikatory dostępu to słowa kluczowe służące do określania dostępności składnika danej klasy.

- Zastosowanie określonego modyfikatora dostępu niesie pewne ograniczenia dla składnika klasy, którego on dotyczy.

Modyfikatory dostępu cd.

Modyfikator dostępu	Ograniczenia dla składnika
public	Nie ma ograniczeń. Składnik posiadający ten modyfikator jest dostępny dla dowolnej klasy.
protected	Składnik posiadający ten modyfikator jest dostępny jedynie w obrębie klasy, w której się znajduje oraz klas dziedziczących.
private	Składnik posiadający ten modyfikator jest dostępny jedynie w obrębie klasy, w której się znajduje.
internal	Składnik posiadający ten modyfikator jest dostępny jedynie w obrębie klasy, w której się znajduje oraz klas w obrębie składnicy (pliku exe, dll itp.).
internal protected	Składnik posiadający ten modyfikator jest dostępny jedynie w obrębie klasy, w której się znajduje, klas w obrębie składnicy oraz klas dziedziczących.

Modyfikatory dostępu - przykłady

```
class Komputer
{
 private class Procesor
 {
 }

 public class Monitor
 {
 }
}
```

Tworzenie obiektu klasy

- Aby utworzyć instancję zdefiniowanej wcześniej klasy, należy zaalokować pamięć dla obiektu tej klasy (służy do tego operator ***new),***
- następnie zainicjować go (wskazując określony konstruktor klasy).
- Dostęp do tego obiektu będzie możliwy jedynie wtedy, gdy utworzymy odpowiednie wskazanie referencyjne (utworzenie odnośnika).

Komputer komp = new Komputer();

Atrybuty klasy - pola

- Każda klasa może zawierać pola, czyli zmienne dowolnego typu, które są dostępne w obrębie tej klasy.

- W zależności od modyfikatora znajdującego się przy deklaracji pola, może ono być dostępne lub nie dostępne dla innej klasy.

Składnia deklaracji pola

[modyfikatory] typ Identyfikator [= wartość_początkowa];

- ***modyfikatory***
  dozwolone są modyfikatory: *static, readonly, volatile* oraz modyfikatory dostępu *public, private* itd. (opcjonalne),

- ***typ***
  typ danych pola (wymagane),

- ***Identyfikator***
  unikalna w obrębie klasy nazwa pola (wymagane),

- ***wartość _początkowa***
  początkowa wartość pola

Pola klas cd.

- Przy deklarowaniu pól klasy zazwyczaj określa się również sposób dostępu do tego pola stosując jeden z modyfikatorów dostępu.
- Jeżeli pole nie będzie miało określonego modyfikatora dostępu, zostanie przyjęty domyślnie modyfikator ***private.***

Pola klas - przykład

```
class Osoba
  {
    public string Imie;
    private short Wiek;
    short Waga;
  }
```

W powyższym przykładzie w klasie ***Osoba*** zadeklarowano trzy pola: publiczne pole ***Imie*** oraz prywatne pola ***Wiek*** i ***Waga.***

Konstruktor

- Konstruktor jest specjalną metodą klasy, która jest wywoływana zawsze po utworzeniu obiektu danej klasy.

- Każdy konstruktor musi mieć taką samą nazwę jak nazwa klasy, w której został zdefiniowany i nie może zwracać żadnych wartości (w przeciwieństwie do standardowych metod).

- Klasa może zawierać wiele konstruktorów muszą się jednak różnić między sobą ilością argumentów.

Konstruktor domyślny

- Jeżeli nie zdefiniujemy żadnego konstruktora, kompilator zrobi to za nas. Konstruktor utworzony przez kompilator nazywa się konstruktorem domyślnym.

- Konstruktor domyślny posiada następujące właściwości:
  - jest widoczny na zewnątrz klasy (ma modyfikator dostępu ***public);***
  - ma taką samą nazwę jak nazwa klasy;
  - nie zwraca żadnych wartości;
  - nie posiada żadnych argumentów;
  - inicjuje wartości pól klasy zerem (w przypadku pól typu wartości za wyjątkiem ***bool), false*** (w przypadku typu ***bool)*** oraz ***null*** w przypadku pól typu referencyjnego.

Konstruktor domyślny - przykład

```
class Komputer
{
}


class Test
{
 static void Main()
 {
  Komputer komp = new Komputer();
  // ...
 }
}
```

- Mimo, że w klasie nie zdefiniowaliśmy żadnego konstruktora, kompilator wygenerował konstruktor domyślny, który posłużył do zainicjowania utworzonego obiektu.

- Jeżeli zdefiniujemy sami jakikolwiek konstruktor (z argumentami lub bez) domyślny konstruktor nie zostanie utworzony

Własne konstruktory 1

Ze względu na to, że domyślnie każdy konstruktor klasy jest prywatny (wyjątek stanowi jedynie klasa abstrakcyjna) musimy pamiętać o umieszczeniu modyfikatora ***public*** w jego definicji, aby późniejsze utworzenie obiektu było w ogóle możliwe:

```
class Komputer
{
  Komputer()  {  }
}

class Test
{
    static void Main()
    {
        Komputer komp = new Komputer (); // BŁĄD: konstruktor niedostępny
    }
}
```

Inicjalizacja pól 1

- W odróżnieniu od zmiennych lokalnych, które wymagają inicjacji, pola składowe klasy nie muszą być inicjowane, gdyż zajmie się tym konstruktor.

- Wszystkie pola, które nie posiadają przypisanej wartości, zostaną domyślnie zainicjowane wartościami 0, ***false*** lub ***null.***

Inicjalizacja pól 2

```
class Data
{
 private short Rok, Miesiac, Dzien;
}
```

=

```
class Data
{
 public Data()
 {
  Rok = 0;
  Miesiac = 0;
  Dzien = 0;
 }

 private short Rok, Miesiac, Dzien;
}
```

Inicjalizacja pól - konstruktory 1

- Programista może chcieć sam zainicjować pola składowe klasy jakimiś innymi wartościami początkowymi. W takim wypadku musi sam zdefiniować konstruktor i przypisać wartości poszczególnym polom składowym wewnątrz tego konstruktora:

```
class Data
{
 public Data()
 {
  Rok = 2004;
  Miesiac = 11;
  // pole Dzien niezainicjowane, więc domyślnie przyjmie wartość 0
 }

 private short Rok, Miesiac, Dzien;
}
```

Inicjalizacja pól - konstruktory 2

```
class Data
{
 public Data()
 {
 }

 private short Rok = 2004, Miesiac = 11, Dzien = 12;
}
```

- Wartości można inicjować również w taki sam sposób jak zmienne lokalne, czyli przypisanie wartości w deklaracji pola

Inicjalizacja pól - konstruktory 3

```
class Data
{
 public Data(short sRok, short sMiesiac, short sDzien)
 {
  Rok = sRok;
  Miesiac = sMiesiac;
  Dzien = sDzien;
 }

 private short Rok, Miesiac, Dzien;
}
```

- Aby móc wpływać na wartości pól klasy w czasie tworzenia obiektów, musimy zdefiniować konstruktor zawierający argumenty, które posłużą do zainicjowania pól klasy:

Inicjalizacja pól - konstruktory przykład użycia

- Aby móc wpływać na wartości pól klasy w czasie tworzenia obiektów, musimy zdefiniować konstruktor zawierający argumenty, które posłużą do zainicjowania pól klasy.

```
Data data = new Data(2004, 11, 12), innadata = new Data(2004, 12, 12);
```

Przeciążanie konstruktora

- Dla jednej klasy można zdefiniować wiele różnych konstruktorów, lecz każdy z nich musi być niepowtarzalny.

- Technika definiowania konstruktorów o identycznej nazwie, które różnią się między sobą argumentami, nazywa się przeciążaniem konstruktora.

- Przeciążając konstruktor należy pamiętać, że jest on rozróżniany na podstawie typu danych argumentów oraz ich ilości (nazwy argumentów nie są brane pod uwagę)

Przeciążanie konstruktora - przykłady

```
// OK: inny typ danych niż powyżej
public Data(short sRok)
{
 Rok = sRok;
}

public Data(short sRok, short sMiesiac, short sDzien)
{
 Rok = sRok;
 Miesiac = sMiesiac;
 Dzien = sDzien;
}
```

```
// BŁĄD: już zdefiniowano konstruktor
public Data(short sMiesiac)
{
 Miesiac = sMiesiac;
}
```

Konstruktor kopiujący 1

- Wśród konstruktorów szczególną rolę pełni konstruktor kopiujący.

- Jego zadaniem jest kopiowanie wartości pól istniejącego obiektu, do innego obiektu tego samego typu, który jest powoływany.

- W języku C# kompilator domyślnie nie tworzy konstruktora kopiującego, dlatego musimy go zdefiniować sami (jeżeli będzie nam potrzebny).

Konstruktor kopiujący 2

- Konstruktor ten charakteryzuje się tym, że ma tylko jeden argument, którym jest nazwa klasy, w której go zdefiniowano.

public Data(Data jakiś_obiekt)

Konstruktor kopiujący - przykład

```
public Data(Data jakiś_obiekt)
   {
        Rok = jakiś_obiekt.Rok;
        Miesiąc = jakiś_obiekt.Miesiąc;
        Dzień = jakiś_obiekt.Dzień;
   }

Data data_1 = new Data(2,  11,  12);
Data data_2 = new Data(data_1);
```

Niszczenie obiektu klasy 1

- W języku C# nie ma możliwości bezpośredniego zniszczenia utworzonego wcześniej obiektu (zajmie się tym **garbage collector *„kolekcjoner nieużytków").***

- Dzięki takiemu podejściu unika się wielu błędów:
  - niezwolnienia pamięci (tzw. problem ***„wycieków pamięci"),***
  - próby ponownego zwolnienia pamięci zajmowanej (próba zniszczenia nieistniejącego obiektu) przez obiekt,
  - czy też zniszczenia aktywnego obiektu (obiekt może być jeszcze potrzebny, jeżeli zostanie zwolniony wcześniej, późniejsze odwołania do niego spowodują pojawienie się błędu).

- Oznacza to również, że nie ma możliwości kontrolowania kolejności, w jakiej obiekty będą niszczone (można jedynie kontrolować kolejność ich tworzenia).

Niszczenie obiektu klasy 3

- **Garbage collector** jest procesem automatycznym, który gwarantuje:
  - zniszczenie obiektu (nie jest jednak określone kiedy dokładnie nastąpi jego zniszczenie);
  - obiekt zostanie zniszczony tylko raz;
  - jedynie obiekty dłużej nieużywane zostaną zniszczone (obiekt jest niszczony tylko, jeżeli żaden inny obiekt nie przechowuje referencji do niego, funkcja wyszukiwania obiektów, do których nie ma żadnych referencji jest czasochłonna, więc ***„kolekcjoner nieużytków"*** wykonuje tą czynność tylko, gdy wielkość dostępnej pamięci jest niska).

Niszczenie obiektu klasy 4

- „Ręczne” wywołanie Garbage collector’a

System.GC.Collect();
System.GC.WaitForPendingFinalizers();

Metoda jest niezalecana o ile jej użycie nie jest konieczne.

Niszczenie obiektu klasy 5

- Niszczenie obiektu w języku C# przebiega w dwóch etapach:
  - deinicjalizacja obiektu - czynności wykonywane przez destruktor, które polegają na tzw. „sprzątaniu" oraz zaznaczeniu, że pamięć zajmowana przez obiekt nie będzie już używana (proces ten można kontrolować poprzez dodanie do klasy własnego destruktora);
  - zwolnienie nieużywanej pamięci - czynności wykonywane przez „kolekcjonera nieużytków", które polegają na zwracaniu do puli nieużywanej już pamięci (proces ten nie może być kontrolowany).

Niszczenie obiektu klasy - destruktor

- W przypadku, gdy obiekt wykorzystuje jakiś zasób niezarządzalny (np.: uchwyt do pliku), który nie może być zniszczony przez „kolekcjonera nieużytków", należy samemu zwolnić pamięć.

- Do kontrolowania tego typu zasobów służy destruktor, który jest wywoływany automatycznie przez „kolekcjonera nieużytków", kiedy obiekt jest niszczony.

Destruktor

- Destruktor jest specjalną metodą klasy, która wykonywana jest jako ostatnia.

- Charakteryzuje się on tym, że podobnie jak konstruktor ma taką samą nazwę jak nazwa klasy, w której go zdefiniowano, z tą jednak różnicą, że przed nazwą umieszcza się znak tyldy ~ .

- Destruktor nie posiada modyfikatora dostępu, nie zwraca żadnej wartości i nie przyjmuje żadnych argumentów

Destruktor - przykład

```
class Data
{
  ~Data()
  {
    // deinicjacja
  }
}
```

- Destruktora należy unikać, o ile nie jest niezbędny (głównie w przypadku zwalniania pamięci zajmowanej przez zasoby niezarządzalne).
- Jeżeli chodzi o zasoby zarządzalne (obiekty), nie ma potrzeby definiowania w klasie destruktora.

Słowo kluczowe ***this***

- Słowo kluczowe ***this*** określa odwołanie się do aktywnej instancji klasy.

- Stosuje się je głównie, gdy nazwy argumentów pokrywają się z nazwami pól klasy (bez ***this*** nie możemy przypisać polu wartości przekazywanej przez argument, gdyż kompilator nie będzie wiedział, że chodzi o pole klasy).

Słowo kluczowe ***this*** – przykład zastosowania

```
class Data
{
 Data(short sRok, short sMiesiac, short sDzien)
 {
  this.sRok = sRok;
  this.sMiesiac = sMiesiac;
  this.sDzien = sDzien;
 }

 private short sRok, sMiesiac, sDzien;
}
```

Metody klasy

- W języku C# każda metoda jest składnikiem jakiejś klasy.

- Składnia definicji metody wygląda następująco:

```
[modyfikatory]  typ Identyfikator([lista_argumentów])
{
   [ciało_metody]
}
```

Składnia definicji metody

[modyfikatory] typ Identyfikator([lista_argumentów])
{
    [ciało_metody]
}

- ***modyfikatory***
  modyfikatory określające zachowanie i dostępność metody (opcjonalne),

- ***typ***
  typ danych zwracany przez metodę (wymagane),

- ***Identyfikator***
  nazwa metody musi być różna od nazwy klasy, w której ją zdefiniowano (wymagane),

- ***lista _argumentów***
  lista argumentów przekazywanych do metody (opcjonalne),

- ***cialo_metody***
  kod realizujący zadaną funkcjonalność (opcjonalne, gdy typ zwracany przez metodę jest ***void).***

Przykład definiowania metody

```
class Data
{
 // konstruktory, inne metody, itp.
 // ...
 public void WyswietlRok()
 {
  Console.WriteLine("Rok  A.D.",sRok);
 }

 private short sRok, sMiesiac, sDzien;
}
```

Definiując metodę należy pamiętać o tym, że jeżeli nie zdefiniujemy żadnego modyfikatora dostępu, domyślnie metoda będzie prywatna

Przykład definiowania metody

```
public void Liczenie()
{
 int iLiczba1, iLiczba2 = 100, iWynik = 0;

 for (iLiczba1 = 0; iLiczba1 < iLiczba2; iLiczba1++)
 {
  if ((iLiczba1 % 10) == 0)
   iWynik += iLiczba1;
 }

 Console.WriteLine(iWynik);
}
```

Wewnątrz metod można umieszczać: deklaracje zmiennych lokalnych, szeregi instrukcji i wyrażeń oraz wywołania innych metod.

Zwracanie wartości

```
public int Kwadrat(int a)
   {
    return a * a;
   }
```

- Każda metoda może zwracać wartość zgodną z typem umieszczonym w definicji. Do zwracania wartości służy słowo kluczowe ***return.*** Po zastosowaniu ***return*** następuje natychmiastowe zwrócenie wartości i wyjście z metody (wszystko co znajdzie się za słowem ***return*** nie zostanie wykonane).

Metody nie zwracające wartości

```
void Pokaz()
{
 Console.WriteLine("Test");
}
```

- Jedynie w przypadku metod nie zwracających wartości ***(void)*** nie wymaga się umieszczania słowa kluczowego ***return.***

```
void Pokaz(bool bPokaz)
{
 if (!bPokaz)
  return;

 Console.WriteLine("Test");
}
```

- Słowo ***return*** można wykorzystać do opuszczenia metody, gdy zaistnieje taka potrzeba.

Argumenty

- Argumenty stanowią informacje, które mogą być przekazywane do oraz z metody. W metodzie umieszcza się je w okrągłych nawiasach oddzielając przecinkami. Każdy argument przyjmuje następującą postać:

  [modyfikator_argumentu] typ Identyfikator

- ***modyfikator_argumentu***
  dopuszczalne są modyfikatory: ***params, out, ref*** (opcjonalne, jeżeli nie zostanie podany domyślnie, przyjmowane jest, iż chodzi o argument wejściowy przekazywany przez wartość);

- ***typ***
  typ danych argumentu (wymagane);

- ***Identyfikator***
  nazwa argumentu (wymagane).

Argumenty przykład

```
public void Dodaj(int Liczba1, int Liczba2, out int Wynik)
{
 Wynik = Liczba1 + Liczba2;
}
```

Każdy argument przekazywany do metody jest traktowany jak zmienna lokalna (argumenty dostępne są w obrębie całej metody, ale nie poza nią). Ze względu na ten fakt należy uważać, aby nie deklarować zmiennych o tej samej nazwie co nazwy argumentów.

Metody przekazywania argumentów

- Istnieją trzy metody przekazywania argumentów:
  - przez wartość - tzw. argumenty wejściowe, ponieważ dane mogą być przekazywane jedynie do metody (nie można ich zwracać z metody);
  - przez referencję - tzw. argumenty wejściowo/wyjściowe, ponieważ dane mogą być przekazywane w obu kierunkach (do oraz z metody);
  - przez wyjście - tzw. argumenty wyjściowe, ponieważ dane mogą być przekazywane jedynie z metody (nie można ich przekazywać do metody).

Argumenty przekazywane przez wartość

- W języku C# przekazywanie argumentów do metody przez wartość jest domyślnym sposobem, więc nie stosuje się w tym przypadku żadnych modyfikatorów argumentu.

- Typ wartości lub zmiennej przekazywanej do metody jako argument, musi być zgodny z zadeklarowanym typem argumentu (lub typem, który może zostać skonwertowany w sposób niejawny).

- W przypadku argumentów przekazywanych przez wartość mamy do czynienia jedynie z kopią danych.

Argumenty przekazywane przez wartość - przykład

```
public void Licz(int Liczba)
{
 Liczba++; // inkrementacja dotyczy kopii
}
```

```
Licz(10);
```

```
int Liczba = 100;
Licz(Liczba);
```

Jaki będzie wynik działania metody?

```
Licz(Abs(-4));
```

Argumenty przekazywane przez referencję

- Argumenty przekazywane przez referencję nie przechowują wartości, tylko takie samo wskazanie do danych, jakie zostanie przekazane do metody.

- Operując na argumencie referencyjnym, operujemy na oryginalnych danych, które zostały przekazane do metody.

- Argument referencyjny deklaruje się poprzez dodanie słowa ***ref*** zarówno w deklaracji jak i wywołaniu metody.

Argumenty przekazywane przez referencję - przykład

```
public void Licz(ref int Liczba)
{
 Liczba++;
}

int Liczba = 10;  // oryginalne dane
Licz(ref Liczba); // liczba będzie miała wartość 11
```

Argumenty przekazywane przez wyjście

- Argumenty przekazywane przez wyjście służą tylko do przekazywania danych z metody.

- Operują one na kopii danych, które muszą zostać zainicjowane jakąś wartością (podobnie jak niezainicjowane zmienne).

- Wartość zostanie przypisana danym oryginalnym, przekazanym do metody. Oznacza to, że nie ma możliwości odczytania danych przekazanych do metody, bowiem w pierwszej kolejności musi nastąpić operacja przypisania wartości.

Argumenty przekazywane przez wyjście 2

- Argument przekazywany przez wyjście, deklaruje się przez dodanie słowa ***out*** zarówno w deklaracji jak i wywołaniu metody.

- Typ argumentu musi być zgodny z typem danych przekazywanych do metody.

Argumenty przekazywane przez wyjście - przykład

```
public void Licz(out int Liczba)
{
 // nie ma możliwości odczytu kopii danych
 Liczba = 0;
 // można odczytać wartość kopii
}
```

```
int Liczba = 10;  // oryginalne dane
Licz(out Liczba); // zmiennej Liczba przypisana zostanie wartość kopii
```

Lista argumentów o zmiennej długości

- W niektórych sytuacjach istnieje potrzeba zdefiniowania metody pozwalającej przekazywać zmienną liczbę argumentów wejściowych.

- Lista argumentów pozwala na określenie wspólnego typu danych, więc wszystkie argumenty przekazywane do metody muszą być zgodne z typem listy lub typem, który może zostać niejawnie skonwertowany do typu listy.

- Do deklarowania listy argumentów o zmiennej długości służy słowo ***params***

Lista argumentów o zmiennej długości - przykład

```
public int Sumuj(params int[] Lista)
{
 int iSuma = 0;

 for (int i = 0; i < Lista.Length; i++)
  iSuma += Lista[i];

 return iSuma;
}

int Suma1 = Sumuj(1,2,3);
int Suma2 = Sumuj(2,3,4,4,5,6,6);
int Suma3 = Sumuj(new int[]{1,2,3,4,5});
```

Przeciążanie

- Wewnątrz klasy może być zdefiniowanych kilka różnych metod mających taką samą nazwę, ale muszą się różnić między sobą listą argumentów.

- Definiowanie kilku metod o takiej samej nazwie i różnej liście argumentów, nazywa się przeciążaniem.

Przeciążanie - przykład

```
public void Pokaz(int Liczba)
{
  ...
}

public void Pokaz(string Napis)
{
  ...
}
 public void Pokaz(int Liczba, string Napis)
 {
   ...
 }
```

Przeciążanie cd.

- Przy przeciążaniu metody, typ zwracanej wartości nie jest brany pod uwagę, więc zdefiniowanie dwóch metod, różniących się jedynie typem zwracanej wartości jest błędem.

```
class Test
{
 public void Pokaz()
 {
  ...
 }
```

```
// BŁĄD: już zdefionowano Pokaz z taką listą
public string Pokaz()
{
 ...
}
```

Statyczne składniki klasy

- Statyczne składniki klasy związane są z klasą, a nie z konkretną instancją obiektu tej klasy.
- Składnik taki jest częścią wspólną dla każdej instancji obiektu tej klasy i istnieje niezależnie od tego, czy utworzymy jakąkolwiek instancję obiektu czy nie.
- Składową statyczną tworzy się umieszczając modyfikator ***static*** przed typem zwracanej wartości składnika.
- Pola statyczne tworzy się, gdy istnieje potrzeba przechowywania wspólnej informacji dla każdej instancji obiektu danej klasy.

Statyczne składniki klasy - przykład

```
class Test
{
 public static int Liczba;     // pole statyczne

 public static void Zwieksz() // metoda statyczna
 {
  Liczba++;
 }
}

Test.Liczba = 100; // odwołanie do pola statycznego
Test.Zwieksz();    // Wywołanie metody statycznej
```

Statyczne składniki klasy cd.

- W przypadku składowych statycznych należy pamiętać, że skoro nie są one związane z instancją klasy, nie możemy korzystać ze słowa kluczowego ***this.***

- Jeżeli chcemy odwołać się do składowej, musimy skorzystać z operatora odwołania do składowej.

Statyczne składniki klasy – przykład 2

```
class Lokata
{
 private static float Procent = 6.5f;

 public static void Zmien(float Procent)
 {
  Lokata.Procent = Procent; // nie możemy korzystać z this
 }
}
```

Statyczne składniki klasy cd.

- Metody statyczne można zgrupować w klasie oferującej jakąś funkcjonalność (np.: klasa **Console** zawiera kilka znanych nam metod statycznych oferujących podstawową funkcjonalność obsługi strumieni wejścia wyjścia).

- Metody zdefiniowane w takiej funkcjonalnej klasie, mogą być dostępne w dowolnym momencie dla dowolnej klasy.

Właściwości

- Właściwości są mechanizmem wspierania hermetyzacji danych wewnątrz klasy. Mechanizm ten pośredniczy w zmianie wartości pól klasy.

- Definiując właściwość możemy zdefiniować rodzaj dostępu do właściwości jako: tylko do odczytu, tylko do zapisu lub do odczytu i zapisu.

- Służą do tego słowa ***get*** (odczyt) i ***set*** (zapis), które umieszcza się na początku bloku instrukcji wewnątrz definicji właściwości

Właściwości cd.

- Jeżeli właściwość zawiera jedynie blok ***get,*** traktowana jest jako tylko do odczytu. Jeżeli zawiera tylko blok ***set*** traktowana jest jako tylko do zapisu. Jeżeli zawiera oba bloki traktuje się ją jako do zapisu i odczytu.

- W przypadku bloku ***get*** należy użyć słowa ***return*** (tak jak w przypadku metod), aby zwrócić wartość właściwości.

- W przypadku bloku ***set,*** polu do którego odwołuje się właściwość, przypisuje się wartość ***value*** (inaczej niż w przypadku metod, gdzie korzystać można z listy argumentów).

Właściwości - składnia

```
[modyfikatory] typ Identyfikator
{
 deklaracja_dostępu
}
```

- ***modyfikatory***
  dopuszczalne są modyfikatory: new, ***static, virtual, abstract, override*** oraz modyfikatory dostępu (opcjonalne);

- ***typ***
  typ danych właściwości (wymagane);

- ***Identyfikator***
  nazwa właściwości (wymagane);
- ***deklaracja_dostęp***
  blok ***get*** lub/i ***set,*** dla bloku ***get*** wymaga się zwrócenia wartości za pomocą ***return,*** dla bloku ***set,*** wartość przypisania przechowywana jest w ***value*** (wymagany co najmniej jeden z bloków).

Właściwości - przykład

```
class Klasa
{
        private int pole;
         public int WlasciwoscW1                    // właściwość
         {
                        get                         // blok gettera
                        {
                                return pole;
                        }
                        set                         // blok settera
                        {
                                pole=value;
                        }
         }
}
```

Właściwości cd 1

- Do właściwości można odwoływać się w taki sam sposób, jak odwołuje się do pól (odwołanie zostanie przetłumaczone przez kompilator na odpowiednie wywołanie ***get*** lub ***set)***

```
Liczba l = new Liczba();

l.WartoscWl = 100;

int Wartosc = l.WartoscWl;
```

Właściwości cd. 2

- Właściwości wykazują podobieństwo do pól klasy, ze względu na składnię odwołania.
- W odróżnieniu od pól, nie mogą być przekazywane do metod jako argumenty ani przez referencje, ani przez wyjście.
- Właściwości mogą być przekazywane przez wartość:

```
Przelicz(l.WartoscWl); // ok. można
Przelicz(ref l.WartoscWl); //BŁĄD:
Przelicz(out l.WartoscWl); //BŁĄD:
```

Właściwości cd. 3

- Skrócony zapis właściwości

```
class Klasa
{
    public int Property { get; set; }
}
```

- W rzeczywistości kompilator sam tworzy zmienną typu ***int***, do której właściwość zapisuje swoje dane

Właściwości cd. 4

- Właściwości można definiować jako statyczne, ale muszą podobnie jak metody statyczne, odwoływać się do statycznych pól klasy.

Koniec męczarni