Laboratorium 7

Programowanie w C# (semestr zimowy 2019/2020)

Temat: Powtórzenie wiadomości i umiejętności (cz. 2)

Prowadzący: Piotr Pięta

1. Poruszone zagadnienia

1.1 Definicje metod za pomocą **wyrażeń lambda (ang. *lambda expressions*)**

[C# 6.0] Konstrukcja programistyczna pozwalająca na *skrócony* zapis metod:

```
using System;

public class Program
{

    public static double Dodaj(double arg1, double arg2 )
    {
        return arg1 + arg2 ;
    }

    public static void Main()
    {
        Console.WriteLine(Dodaj(0.97, 0.6));
    }
}
```

Zamiast tego można użyć operatora **lambda**, który ma postać =>.

Po lewej stronie takiego operatora występuje pewien parametr, a po prawej stronie wyrażenie lub blok instrukcji.

Oznacza to, że metoda może też wyglądać tak:

```
public double Dodaj(double arg1 , double arg2 ) => arg1 + arg2;
```

1.2 **Obiekt** jako **argument metody**

a) **Argumenty metody (**inaczej ***parametry*****) = dane, które są przekazywane danej metodzie**
b) **Metoda może posiadać dowolną liczbę argumentów – są one umieszczone w nawiasach okrągłych za nazwą metody (stosujemy przecinek między kolejnymi argumentami)**
c) **Słowo kluczowe void umieszczone przed nazwą metody oznacza, że metoda nic nie zwraca (w wyniku, podobnie nie używamy słowa kluczowego return)**
d) **Argumentem metody może być m.in. obiekt (*De facto* referencja do obiektu)**

Prosta ilustracja: (I)

```
void NazwaMetody(Punkt punkt)
{
        x = punkt.x ;
        y = punkt.y ;
}
```

(II)

```
Punkt NazwaMetody2 ( )
{
        Punkt punkt = new Punkt ( ) ; //?
        punkt.x = X ;
        punkt.y = y ;
        return punkt ;

}
```

1.3 Przeciążanie metod (ang. *methods overloading*)

a) Taka sama nazwa metod, ale różne argumenty, jakie mogą przyjąć (gdy się nie różnią?)
b) Mogą, ale nie muszą różnić się zwracanym typem

Prosta ilustracja (I)

```
void Metoda(int x, int y)
{
        arg_x = x;
        arg_y = y;
}

void Metoda(Punkt punkt)
{
        arg_x = punkt.x;
        arg_y = punkt.y;
}
```

c) Istnienie wielu metod o tej samej nazwie (przeciążonych) w klasie => oczywiście tu (powyżej) realizują to samo zadanie, jednak wcale nie muszą

1.4. Konstruktor, destruktor, czyli chwilowy powrót do obiektowości

Po utworzeniu obiektu (operator new) w pamięci wszystkie jego pola zawierają wartości domyślne (default)

[1] https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/default-values-table

Dla wygody programisty możemy dokonać inicjalizacji pól za pomocą specjalnej metody (Konstruktora).

a) Nigdy nie zwraca wyniku (nie umieszczamy słowa kluczowego *void*)
b) Nazwa Konstruktora musi być zgodna z nazwą klasy
c) To, **co będzie robił** konstruktor (jakie zadania wykona) zależy tylko i wyłącznie od nas, programistów 😊 What?! Intuicja a praktyka, Zaawansowani na pewno wiedzą:

Otóż konstruktor jest to specjalna metoda, która jest wywoływana zawsze w trakcie tworzenia obiektu w pamięci i jako taka nadaje się doskonale do jego zainicjowania.

d) Konstruktor nie musi być bezargumentowy
e) Jeśli konstruktor **przyjmuje** argumenty, to przy **tworzeniu** obiektu należy je podać:

Zamiast zapisu: *nazwaKlasy zmienna = new nazwaKlasy ( )*, zastosujemy:

nazwaKlasy zmienna = new nazwaKlasy(argumenty Konstruktora)

f) Konstruktor (metoda) oczywiście można przeciążać
g) Przeciążone konstruktory muszą różnić się argumentami

Prosta ilustracja (I)

```
//*************************************************************

        class Punkt
        {
        int x;
        int y;
        /************************************************/
        Punkt() {          Punkt(int xx, int yy) {      Punkt (Punkt punkt) {
            x = 1;             x = xx;                  x = punkt.x;
            y = 1;             y = yy;                  y = punkt.y;
            }                  }                            }
        }
```

Co *zyskujemy*?

Osobne wywołanie każdego z trzech konstruktorów, w zależności od tego, który z nich jest najbardziej odpowiedni w danej sytuacji

☺ *Zadanie*: Aby przekonać się, że tak jest rzeczywiście stwórz w metodzie *Main*() trzy obiekty typu Punkt – każdy z tych obiektów będzie tworzony za pomocą (innego) konstruktora

h) Jeżeli w klasie nie został zdefiniowany (jakikolwiek) konstruktor – zostaje automatycznie tworzony konstruktor bezparametrowy (domyślne wartości)
i) Jeżeli w klasie jest zdefiniowany co najmniej jeden konstruktor z parametrem, to należy samodzielnie zdefiniować konstruktor bez parametrów (nie jest tworzony automatycznie w takim przypadku)

new = utworzenie obiektu = zarezerwowanie dla danego obiektu miejsca w pamięci

Po wykorzystaniu obiektu możemy chcieć **zwolnić** pamięć → zrzucenie tego zadania na programistów (w przeszłości) powodowało liczne błędy → obecnie, zwalnianie pamięci (odpowiedzialność: środowisko) – nie mamy nad tym kontoli (poza wyjątkami, np. wywołanie metody *GC.Collect()* wymusza zainicjowanie procesu odzyskiwania pamięci)

Garbage collector – z ang. *odśmiecacz* – czuwa nad optymalnym wykorzystaniem pamięci i uruchamia proces jej odzyskiwania (w momencie, kiedy wolna ilość oddana do dyspozycji programu

zbytnio się zmniejszy). Problem → narzut czasowy, bowiem proces odśmiecania musi zająć czas procesora, my (programiści) się tym „nie przejmujemy"

Środowisko .NET jest w stanie automatycznie zarządzać wykorzystaniem pamięci (ulokowanej standardowo, czyli za pomocą operatora new).

a) **Destruktor – metoda wykonywana zawsze, kiedy obiekt jest niszczony, usuwany z pamięci.**
b) W ciele destruktora możemy wykonać dowolne instrukcje sprzątające.
c) W deklaracji nazwę destruktora poprzedzamy znakiem tyldy – nazwa destruktora tożsama z nazwą konstruktora, nie przekazuje żadnych wartości
d) W przeciwieństwie do konstruktora, w danej klasie może być zdefiniowany tylko jeden destruktor

1.5 Słowo kluczowe **this**

Odwołanie do obiektu bieżącego = *niejako* referencja do aktualnego obiektu = autoreferencja

Prosta ilustracja (I)

```
Punkt(int x, int y) { //treść konstruktora, (argumenty nazwane) inaczej niż xx, yy itp.}

Punkt(int x, int y) { //*** formalnie zapis jest poprawny

    x = x;      //*** problem: w jaki sposób kompilator kompilator ma ustalić,

    y = y; }    //*** kiedy chodzi nam o argument konstruktora/pole klasy?
```

Rozwiązaniem jest zastosowanie **this** → zaznaczamy, czy chodzi nam o składową (klasy): pole, metodę:

this.nazwaPola lub **this.nazwaMetody(argumenty)**

Prosta ilustracja (I)

```
Punkt(int x, int y)
{
    this.x = x; //(*)
    this.y = y; //(**)
}
```

Jak *zinterpretować* powyższy zapis?

(*) Przypisz polu x wartość przekazaną jako argument o nazwie x

(**) Przypisz polu y wartość przekazaną jako argument o nazwie y

a) Ponadto słowo kluczowe *this* pozwala na wywołanie konstruktora z wnętrza innego konstruktora (kiedy w danej klasie mamy *kilka przeciążonych konstruktorów*, a zakres

wykonywanego przez nie kodu pokrywa się – jednak nie zawsze takie wywołanie jest możliwe i niezbędne)

b) Jeżeli za jednym z konstruktorów umieścimy dwukropek, a za nim słowo kluczowe this i listę argumentów umieszczonych w nawiasie okrągłym:

```
class JKlasa {

    JKlasa(argumenty) : this(arg1, arg2, …, arg(N-1), argN)
    {
    }
    //POZOSTAŁE konstruktory
},
```

to przed widocznym konstruktorem zostanie wywołany ten, którego argumenty pasują do wymienionych w nawiasie po *this*. Jest to tzn. zastosowanie **inicjalizatora** (jeszcze inaczej: **listy inicjalizacyjnej**)

Przykład (wywołanie konstruktora z wnętrza innego konstruktora):

```
class Punkt {//deklaracje itp.

    Punkt(int x, int y) {this.x = x; this.y = y;}

    Punkt():this(1,1) {}

    public static void Main() {

        Punkt punkt1 = new Punkt(100, 200);

        Punkt punkt2 = new Punkt();

        Console.WriteLine(//wyświetl pierwszą składową punktu1)

        Console.WriteLine(//wyświetl drugą składową punktu1)

        Console.WriteLine(//wyświetl pierwszą składową punktu2)

        Console.WriteLine(//wyświetl drugą składową punktu2)

    }
```

a) Klasa Punkt ma dwa konstruktory, w tym bezargumentowy (konstruktor bezargumentowy) próbujący przypisać polom x, y stosowne wartości

b) Wykorzystujemy listę inicjalizacyjną, wnętrze konstruktora (drugiego) jest puste Punkt():this(1,1) – dzięki temu jest wywoływany konstruktor, którego argumenty są zgodne z podanymi na liście, a więc konstruktor przyjmujący dwa argumenty typu *int*

2 Zadania do samodzielnej realizacji

2.1

Napisz klasę, której zadaniem będzie przechowywanie wartości typu int. Dołącz jednoargumentowy konstruktor przyjmujący argument typu int. Polu klasy nadaj nazwę *liczba*, tak samo nazwij argument konstruktora.

a) Dodaj do klasy przeciążony konstruktor bezargumentowy ustawiający jej pole na wartość -1

2.2

Napisz klasę zawierającą dwa pola: pierwsze typu *double* i drugie typu *char*.

Dopisz cztery przeciążone konstruktory: pierwszy, przyjmujący jeden argument typu double, drugi przyjmujący jeden argument typu char, trzeci przyjmujący dwa argumenty – double i char, czwarty – przyjmujący również dwa argumenty: typu char i typu double (drugi argument)

2.3

Zmodyfikuj wybrane własne programy wzbogacając je o poznane konstrukcje programistyczne (konstruktory, destruktory, this)