Laboratorium 6

Programowanie w C# (semestr zimowy 2019/2020)

Temat: Powtórzenie i utrwalenie wiadomości (cz. 1)

Prowadzący: Piotr Pięta

1. Poruszone zagadnienia

1.1 Operator warunkowy

(wyrażenie logiczne)? wyrażenie1 : wyrażenie2;

```
static void Main(string[] args)
{
      int y, x = 1;s
      y = (x > 0) ? ++x : --x;
      Console.WriteLine(y);
      Console.ReadKey();
}
```

Ograniczenia: brak możliwości wykorzystania metod, które nic nie zwracają, brak możliwości definiowania bloków instrukcji

Kwestia: czy każdą instrukcję warunkową *if – else* można niejako zamienić na operator warunkowy (postać powyżej)?

1.2 Przestrzeń nazw

Przestrzeń nazw (ang. *namespace*) jest **kontekstem**, w ramach którego wszystkie nazwy powinny być unikatowe (niepowtarzalne). Stosowanie przestrzeni nazw pomaga porządkować kod projektów. Nie musimy się martwić, że w jednym programie (pisanym przez zespoły programistów) pojawią się klasy o tej samej nazwie i spowoduje to konflikt.

Przykład z naszych laboratoriów:

System (przestrzeń nazw) → using System;
System.Console.WriteLine(); *vs.* Consolee.WriteLine();

1.3 Typowe błędy z pętlą for() + dodatkowe możliwości

```
static void Main(string[] args)
{
int i, j, k;

for (i = 0, j = -10, k = 1; i > (j + k); i--, j++, k++)
{
     Console.WriteLine(" i = {0} j = {1} k = {2}", i, j, k);
}
Console.ReadKey();
}
```

Dobre praktyki programistyczne a kod powyżej…

1.4 Instrukcja *break* vs. *continue*

```
static void Main(string[] args)
{
     for (int i = 1; i <= 3; i++)
     {
          Console.WriteLine("Liczby w {0} wierszu:", i);
          for (int j = 1; j <= 5; j++)
          {
               if (j == 3) break; //zwróć uwagę na ten moment
                    Console.Write(j + ",");
          }
          Console.WriteLine();
     }
     Console.ReadKey();
}
```

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

static void Main(string[] args)
{
  for (int i = 1; i <= 6; i++)
  {
       if (i == 4)
            continue; //zwróć uwagę na ten moment
       Console.WriteLine(i);
  }
  Console.ReadKey();
   }
```

1.5 Typy wartościowe i typy referencyjne w C#

Typy będące **wartościami** - to wszystkie typy danych liczbowych (int, float, double itd.) oraz typy wyliczeniowe i struktury.
W przypadku typów będących wartościami, przy przypisywaniu jednej zmiennej do drugiej tworzy się lokalna kopia bitowa. Takie zmienne przechowywane są na stosie i usuwane z pamięci gdy wychodzą poza zdefiniowany zasięg (np. poza zasięg metody, w której zostały zadeklarowane).

Typy **referencyjne** to klasy i interfejsy. Kopiowanie typu referencyjnego (obiektu klasy lub interfejsu) daje w efekcie wiele referencji (odniesień) wskazujących na to samo miejsce w pamięci. Takie zmienne przechowywane są na stercie i usuwane z pamięci gdy wykonywane jest automatyczne czyszczenie pamięci zarządzanej sterty (ang. *garbage collection*).

Wszystkie typy danych (będące wartościami lub referencjami) wywodzą się ze wspólnej klasy bazowej Object z przestrzeni nazw System.

Konwersja typu wartościowego na typ referencyjny to tzn. **opakowywanie** (ang. boxing).

```
short i = 5; //zwykła wartość
object obiekt = i ; // opakowanie wartości - zwracana
        // jest referencja do utworzonego obiektu
        // (obiekt musi być zainicjalizowany żeby
        // istniała referencja do niego)
```

Konwersja typu referencyjnego na typ wartościowy jest nazywana **odpakowywaniem** (ang. *unboxing*)

```
short j = (short)obiekt; //odpakowanie referencji oraz
        //przywrócenie typu short
```

1.6 Typy wyliczeniowe

Typy wyliczeniowe stosowane są w przypadku kiedy wartości liczbowe wygodniej jest zastąpić zbiorem symbolicznych nazw. Deklarację typu wyliczeniowego w języku C# należy umieścić **poza** metodą klasy. Ma ona postać np.:

```
enum Dni { Pn, Wt, Śr, Czw, Pt, So, Nd };
```

Odwołanie się do konkretnego elementu jest trywialne (nie trzeba tworzyć dodatkowych obiektów):

Console.WriteLine(Dni.Pn);

Domyślnie każdy element jest mapowany na liczbę całkowitą.

Klasą podstawową typu wyliczeniowego jest *System.Enum*

1.7 Struktury w C#

Struktury to zorganizowany zbiór danych, *niekoniecznie* tego samego typu. Struktury w języku C#, podobnie jak klasy mogą posiadać pola, metody a nawet konstruktor. Strukturę można przekazać jako parametr metody.

struct Student

{

public string **Nazwisko**;

public long **Tel**;

public byte **Wiek**;

// konstruktor struktury z parametrami nazw, telefon, w:

// wartości tych parametrów zostaną przypisane polom zmiennej

// typu struktury Student

public Student(string nazw, long telefon, byte w)

{

Nazwisko = nazw;
Tel = telefon;
Wiek = w;

}

}

/*****************************/

```
class Program
{
static void Main(string [] args)
{
        // Utworzenie zmiennej s1 typu Student poprzez
        //konstruktor:
        Student s1 = new Student( "Jan Kowalski", 666666666,
        23);
        // Utworzenie zmiennej s2 typu Student poprzez
        //konstruktor:
        Student s2 = new Student( "Piotr Kowalski",
        555555555, 43);
        Console.ReadKey();
}
}
```

Przykładowe struktury:

System.Drawing.Point – przechowuje dane punktu o określonych współrzędnych

System.Drawing.Color – przechowuje informacje o kolorze

Najważniejsze *różnice* pomiędzy klasą a strukturą: *brak* możliwości dziedziczenia struktur (nie mogą być dziedziczone, ani same nie dziedziczą), struktury są przechowywane w pamięci jako *wartości* (nie jest to typ referencyjny!), *brak* możliwości inicjacji pól (zmienne struktury nie mogą być inicjalizowane tak, jak zmienne klasy), brak destruktorów

2. Zadania do samodzielnej realizacji

2.1
Z wykorzystaniem **operatorów warunkowych** napisz program pobierający od użytkownika dwie liczby całkowite. Program powinien (tylko) wypisać parzyste liczby znajdujące się pomiędzy podanymi wartościami.

2.2
a) napisać program, który ustali szczęśliwy numerek na dowolną wartość całkowitą >=0 oraz poprosi użytkownika o podanie liczby. Jeśli użytkownik poda liczbę **różną** od szczęśliwego numerka wyświetlić komunikat „*Spróbuj jeszcze raz*” i proces zgadywania powtarzać w pętli aż do uzyskania poprawnej liczby (pętla do while!). Jeśli użytkownik zgadnie szczęśliwy numerek wyświetlić komunikat „Brawo”;

b) zmodyfikować program tak, aby zliczał i dodatkowo wyświetlał informację za którym razem podano poprawną odpowiedź;
c) zmodyfikować program tak, aby możliwa było tylko 4- krotna próba i po czwartym razie - program kończy działanie komunikatem *„Niestety do czterech/trzech razy sztuka”*;
program ma dodatkowo generować losową liczbę z przedziału <1,9>;
d) **po realizacji punktu c)** pytać użytkownika: "Gramy dalej (t/n)?" i jeśli użytkownik potwierdzi dalszą gre to ponownie realizować zadania z punktu c (zastosować pętlę while!).

Przypomnienie – losowanie liczb w C# (.NET)

```
// tworzenie nowej instancji (obiektu) klasy:
        Random Oran = new Random();
// wylosowanie liczby z zakresu 0-999:
        in t x = Oran.Next(1000);
```

2.3

Napisać program, który znajduje w podanym **z klawiatury ciągu liczb** zawartych pomiędzy 0 a 20 **największą** i **najmniejszą** oraz poda **ile razy** każda z tych dwóch liczb wystąpiła. Zakładamy, że ciąg o nieznanej z góry długości będzie zakończony liczbą -1. W rozwiązaniu nie wolno stosować tablic. W razie podania liczby spoza przedziału nie należy jej uwzględniać a należy poprosić o powtórne podanie liczby z właściwego przedziału <0, 20>.

Wynik działania programu powinien wyglądać następująco:

Podaj liczbę (-1 kończy program): 12

Podaj liczbę (-1 kończy program): 8

Podaj liczbę (-1 kończy program): 18

Podaj liczbę (-1 kończy program): 7

Podaj liczbę (-1 kończy program): 11

Podaj liczbę (-1 kończy program): 12

Podaj liczbę (-1 kończy program): 7

Podaj liczbę (-1 kończy program): 9

Podaj liczbę (-1 kończy program): -1

Maksimum równe 18, liczb a powtórzeń 1

Minimum równe 7, liczb a powtórzeń 2.