Platforma .NET – Wykład 3
Programowanie w C# – Część 1/3

Osoba prowadząca wykład, laboratorium i projekt:
dr hab. inż. Marek Sawerwain, prof. UZ

Instytut Sterowania i Systemów Informatycznych
Uniwersytet Zielonogórski

e-mail : M.Sawerwain@issi.uz.zgora.pl
tel. (praca) : 68 328 2321,
pok. 328a A-2,
ul. Prof. Z.Szafrana 2,
65-246 Zielona Góra

Ostatnia kompilacja pliku: Monday 5th June, 2023, t: 23:02

Spis treści

Plan wykładu – spotkania tydzień po tygodniu

(1) Informacje o wykładzie, pojęcie platformy, podstawowe informacje o platformie .NET

(2) Składowe platformy .NET: CLR, CTS, języki programowania, biblioteki klas, pojęcie podzespołu (ang. assembly)

(3) Programowanie w C# – środowisko VS, MonoDevelop, syntaktyka C#, wyjątki, współpraca z DLL

(4) Programowanie w C# – model obiektowy, typy uogólnione, lambda wyrażenia

(5) Programowanie w C# – aplikacje „okienkowe", programowanie wielowątkowe

(6) Programowanie w F# – podstawy, przetwarzanie danych tekstowych,

(*) "Klasówka I", czyli egzamin cześć pierwsza

(7) Dostęp do baz danych

Plan wykładu – tydzień po tygodniu

(8) Język zapytań LINQ, Entity Framework
(9) Obsługa standardu XML
(10) Technologia ASP.NET 1/2
(11) Technologia ASP.NET 2/2
(12) Model widok i kontroler – Model View Controller
(13) Tworzenie usług sieciowych SOAP i WCF (komunikacja sieciowa)
(14) Wykład monograficzny .NET 1
(15) Wykład monograficzny .NET 2
(*) "Klasówka II", czyli egzamin cześć druga

Plan wykładu

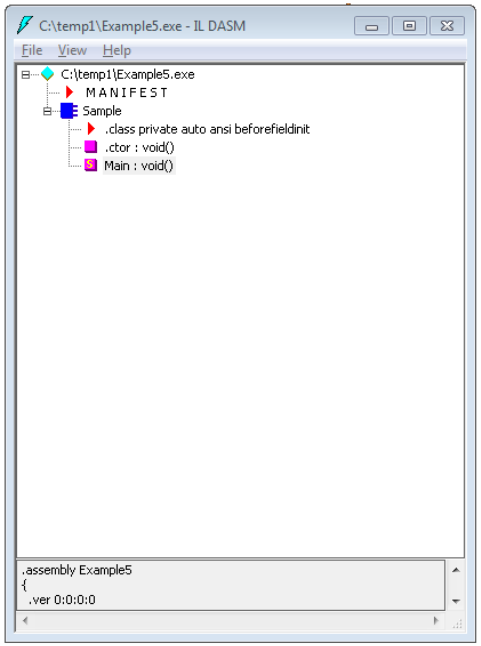
1. Narzędzia dla programistów związane z Platformą .NET
   1. narzędzia do IL,
   2. zarządzanie GAC'kiem, disco,
   3. „dotfuscator", „snippets code"
2. Wszystko co trzeba o wiedzieć o C#, a boimy się zapytać – część I,
   1. metoda Main,
   2. typy danych string, tablica,
   3. instrukcje warunkowe.
3. A już za tydzień, min:
   1. model obiektowy,
   2. wyrażenia lambda,
   3. typy uogólnione.

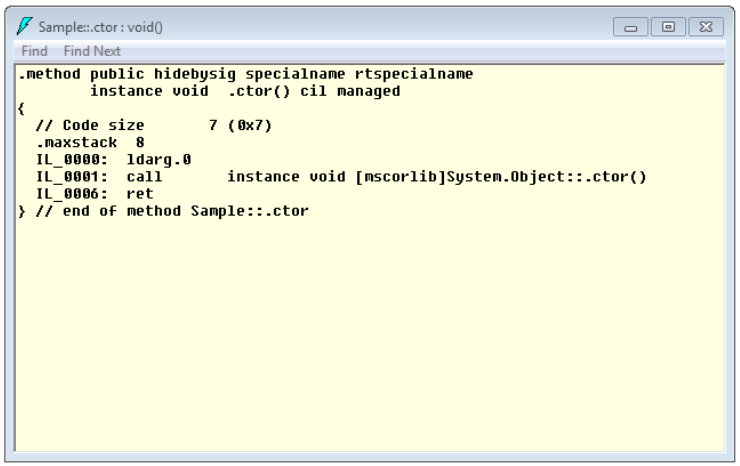
Narzędzia dodatkowe

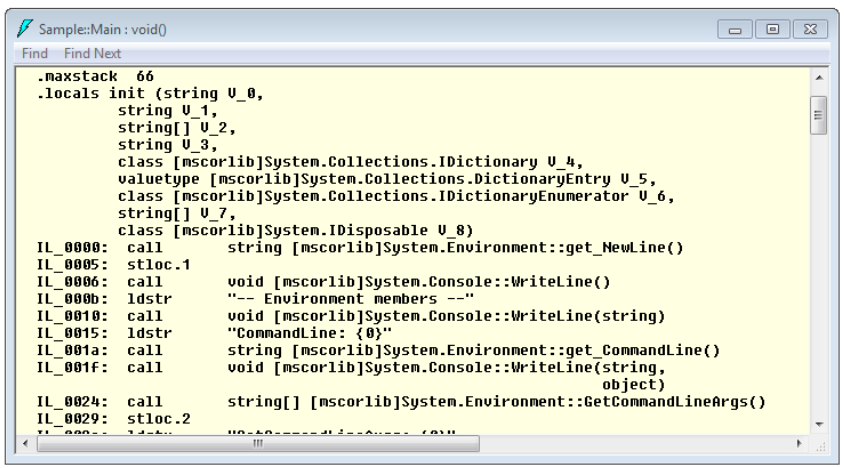
Oprócz głównych środowisk jak Visual Studio, SharpDevelop, MonoDevel w ramach platformy .NET obecne min. następujące narzędzia:

1. ildasm, monodis,
2. MAGE – manifest manager,
3. disco – „odkrywca" usług sieciowych,
4. gacutil, zarządzenie podzespołami,
5. ikvm – maszyna wirutualna Javy w środowisku .NET

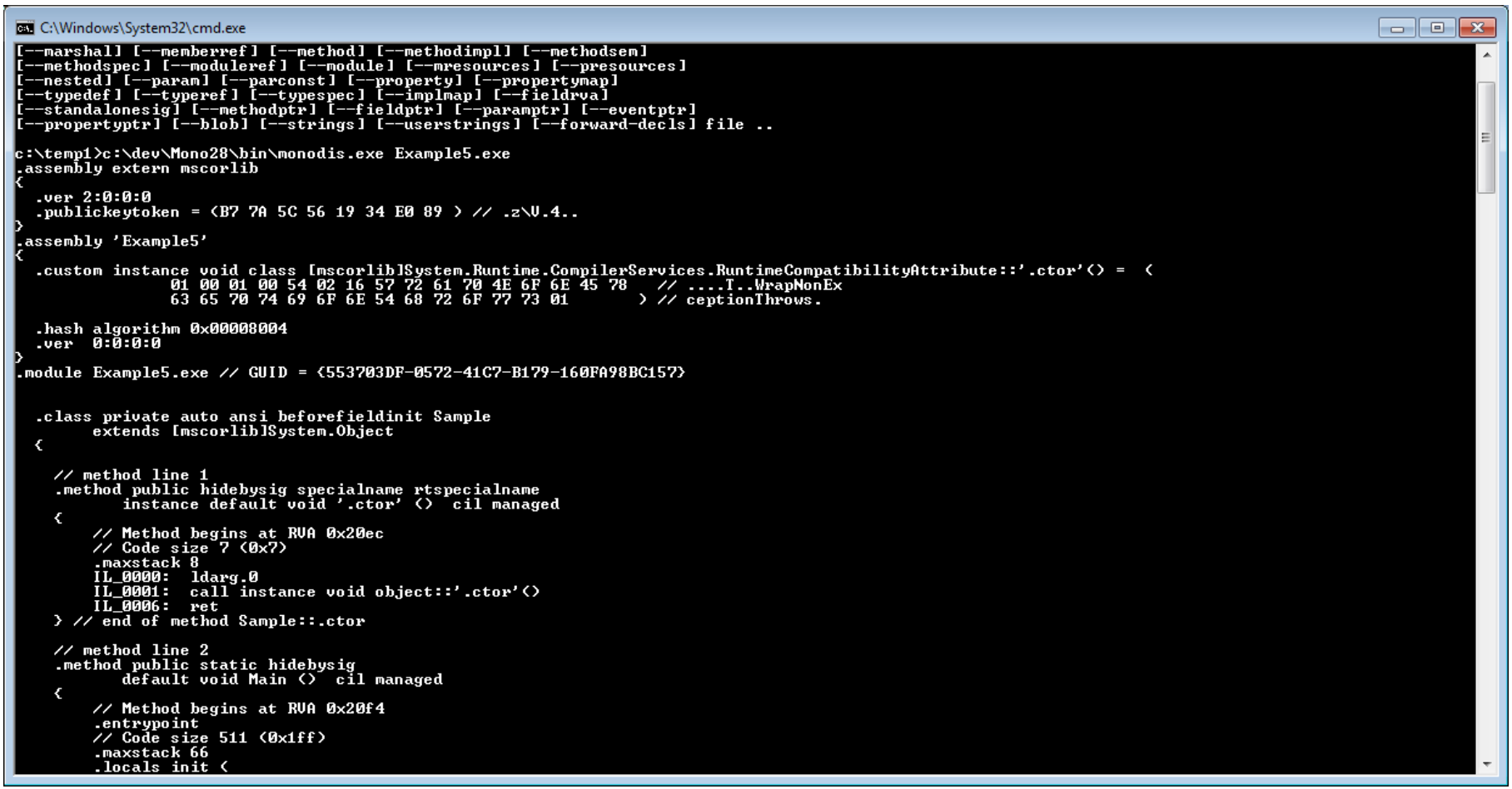
Program ILDASM



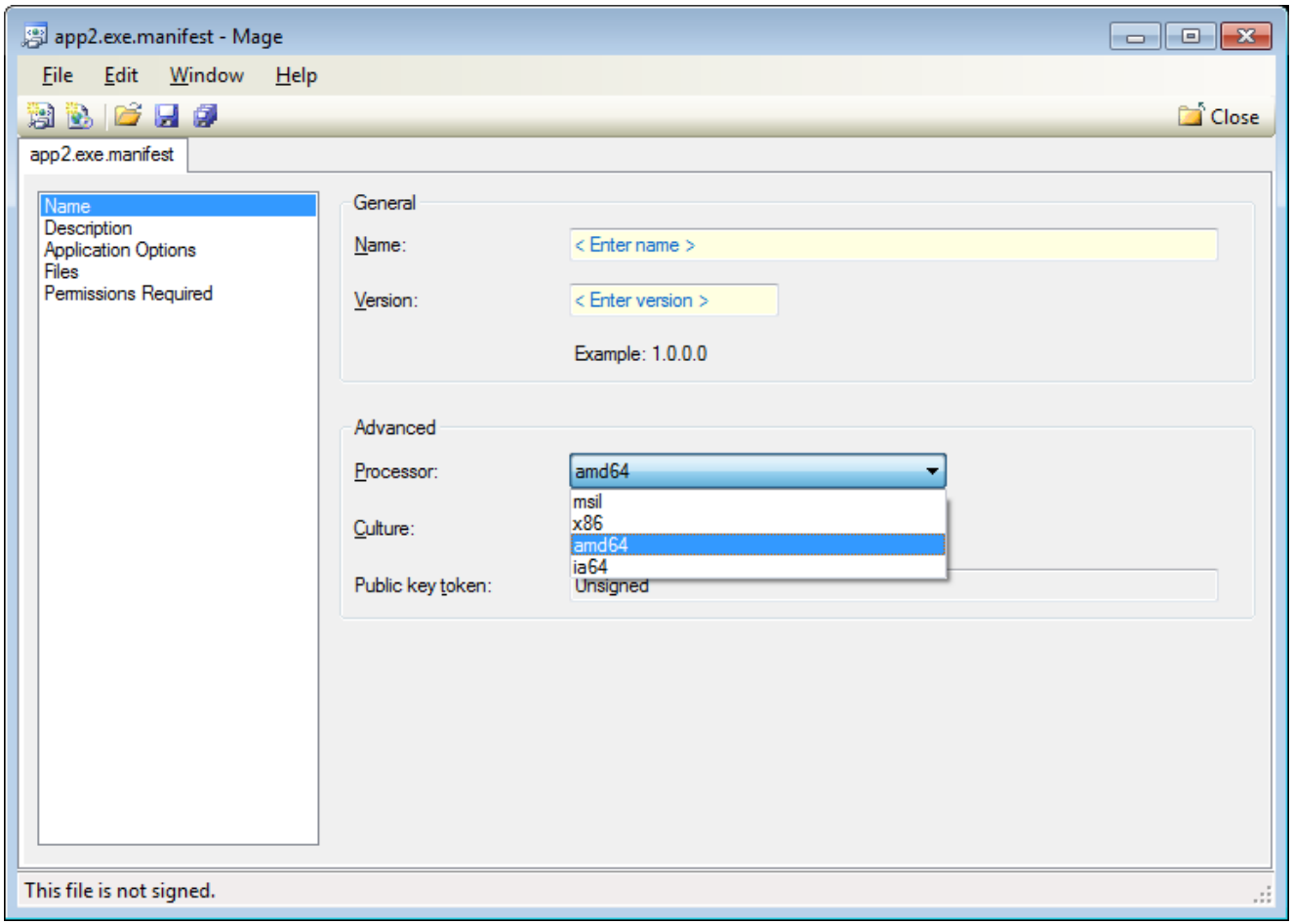




Program monodis

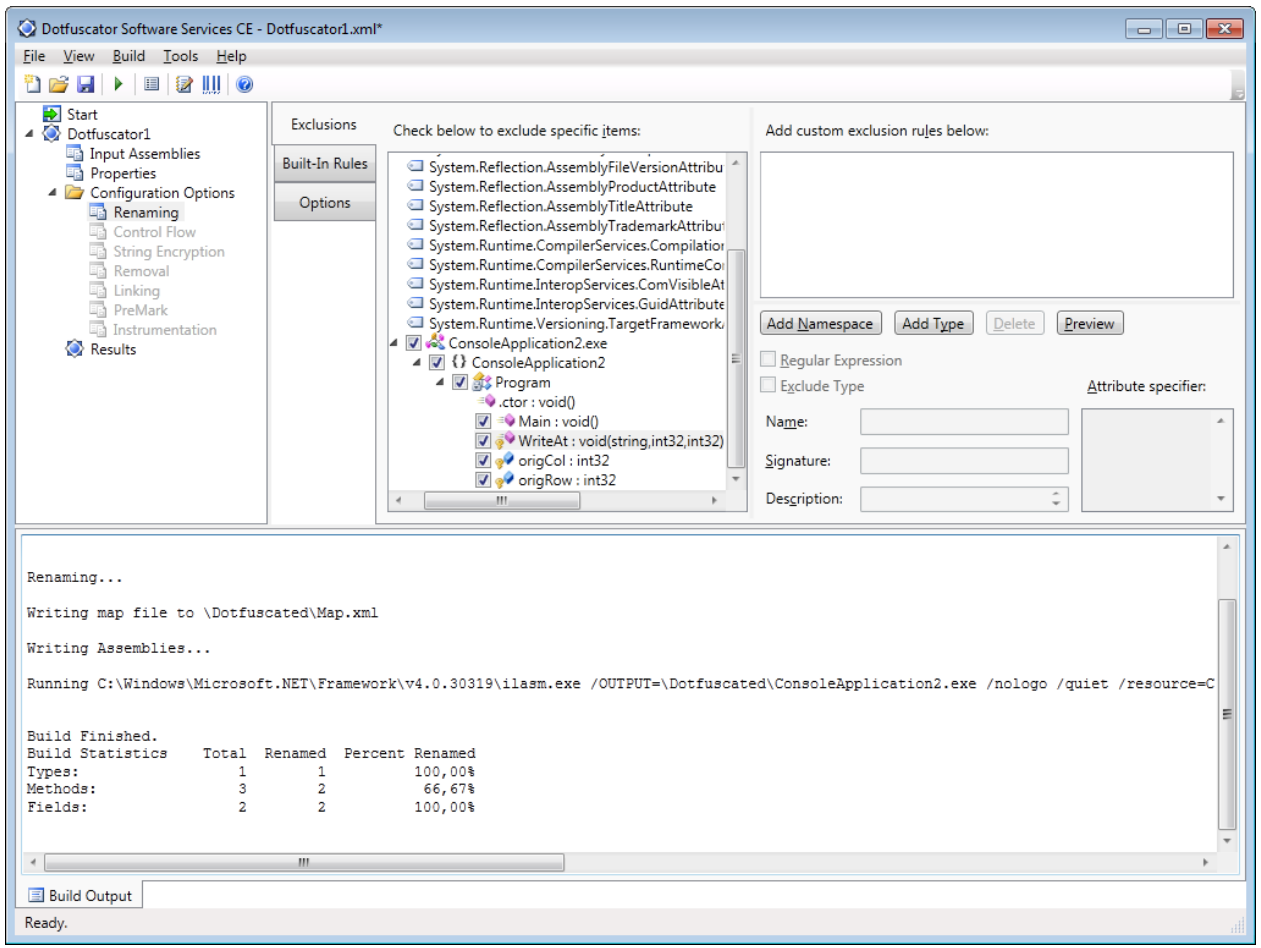


Manifest manager



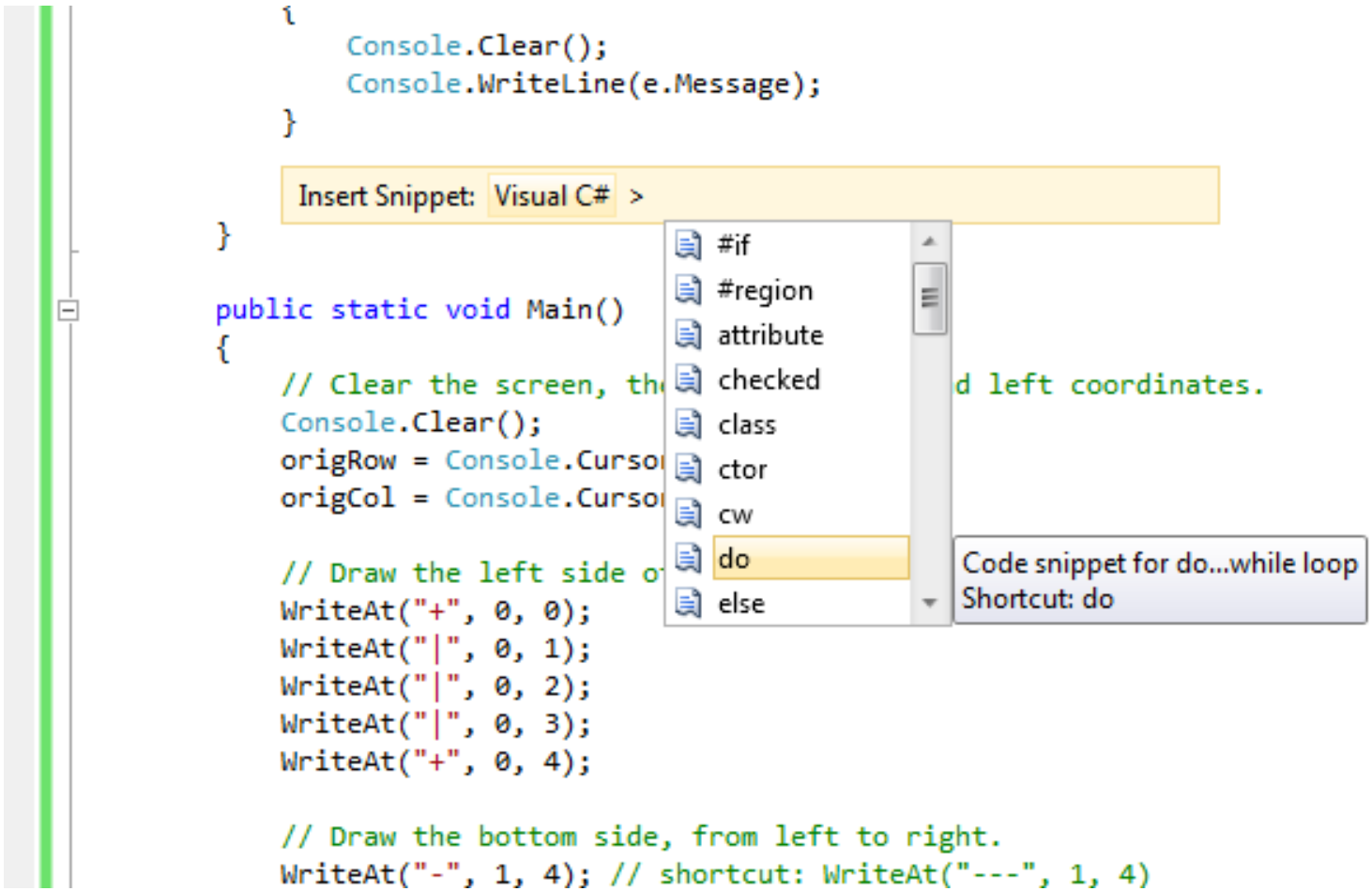
Dotfuscator

Zadaniem tego programu jest zniekształcenie kodu IL, celem utrudnienia jego analizy:



Code snippets – „szablony” kodu

Wstawienie szablonu, wywołuje się za pomocą kombinacji CTRL-K, CTRL-X:



Rozwój pojęć języka C# – 1/5

1. C# 1.0
   1. Type System
   2. Memory Management
   3. Syntatic Sugar
   4. C# 1.1
2. C# 2.0
   1. Generic Types
   2. Partial Types
   3. Static Classes
   4. Iterators
   5. Anonymous Methods
   6. Delegate Inference
   7. Delegate Covariance and Contravariance
   8. Nullable Types
   9. Property Accessors
   10. Null – Coalesce Operator
   11. Namespace Aliases

Rozwój języka C# – 2/5

1. C# 3.0
   1. Local Variable Type Inference
   2. Extension Methods
   3. Anonymous Types
   4. Lambda Expressions
   5. Query Expressions
   6. Expression Trees
   7. Automatic Properties
   8. Object Initializers
   9. Collection Intializers
   10. Partial Methods
2. C# 4.0
   1. Interoperability
   2. Dynamic Lookup
   3. Named and Optional Parameters
   4. COM Interoperability
   5. Generics, co- and contravariance

Rozwój języka C# – 3/5

1. C# 5.0
   1. Asynchronous methods
   2. Caller info attributes
2. C# 6.0
   1. Compiler-as-a-service (project Roslyn)
   2. Import of static type members into namespace
   3. Exception filters
   4. Await in catch/finally blocks
   5. Auto property initializers
   6. Default values for getter-only properties
   7. Expression-bodied members
   8. Null propagator (null-conditional operator, succinct null checking)
   9. String Interpolation
   10. nameof operator
   11. Dictionary initializer

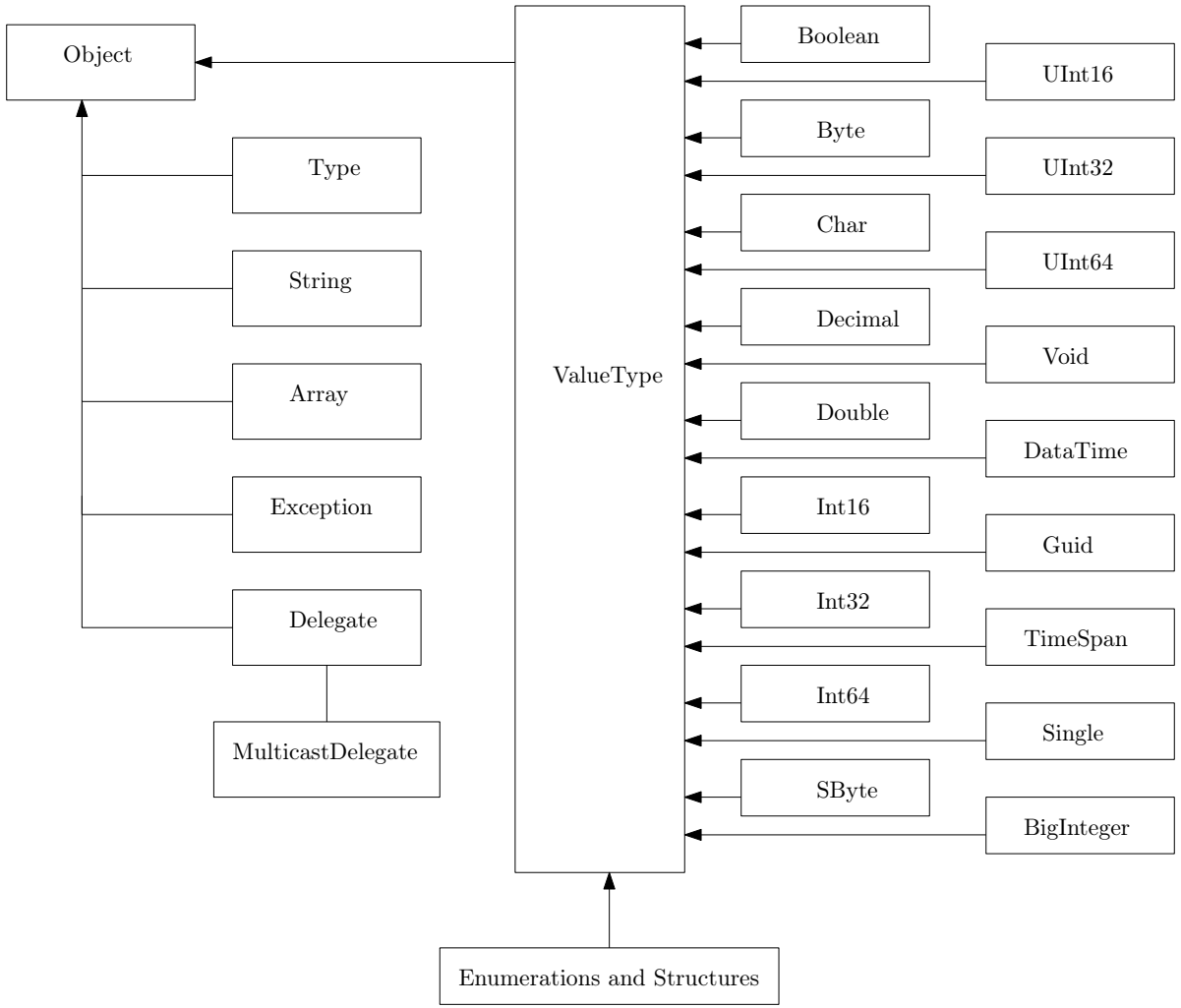
Rozwój języka C# – 4/5

1. C# 7.0
    1. Binary Literals
    2. Digit Separators
    3. Local Functions
    4. Type switch
    5. Ref Returns
    6. Named tuples
    7. Out var
    8. Arbitrary async returns
    9. Expression bodied getters and setters
    10. Expression bodied constructors and finalizers

Rozwój języka C# – 5/5

1. C# 7.0 +1
    1. Pattern Matching
    2. Records
    3. With expressions
    4. Non-null Reference Types
    5. Async Main
    6. Address of Static
    7. Bestest Betterness
    8. private protected
    9. Source Generation

Typy zmiennych



Typy wartościowe/proste są przechowywane na stosie, natomiast typy referencyjne (obiekty) są przechowywane na stercie.

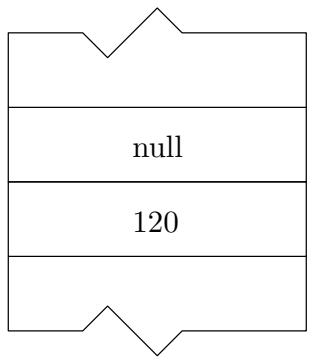
Rodzaje kontekstów dla zmiennych

Typ zmiennych	Kontekst stosowania
non-nullable value	wartość dokładna typu
Nullable value	wartość null lub wartość dokładna
object	referencja do null, referencja do innego obiektu, referencja do „pudełka”
Interface	referencja do null, referencja do instancji klasy (lub pudełka) która implementuje określony interfejs
Array	referencja do null, referencja do instancji tablicy o takim samym typie, referencja do instancji o kompatybilnym typie
Delegate	referencja do null, referencja do instancji delegatu

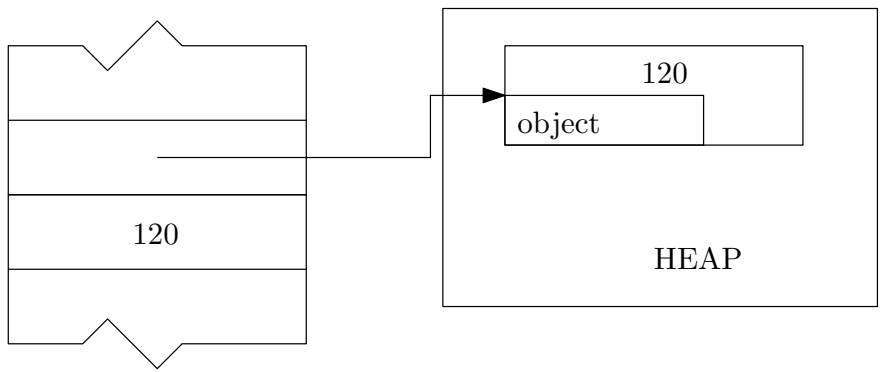
„Pakowanie” / „Pudełkowanie” zmiennych

Wszystkie typy C# dziedziczą po typie **object**. Jednak ze względu na efektywność wartości dla typów podstawowych jak np: int, long, float są przechowywane na stosie bez odwoływania się do obiektu typu **object**. W przypadku, jeśli komponent obiektowy jest potrzebny, można stosować technikę pakowania/pudełkowania zmiennych do typu **object**.

```
int x = 120;
object ox = null;
```




```
ox = i;
```

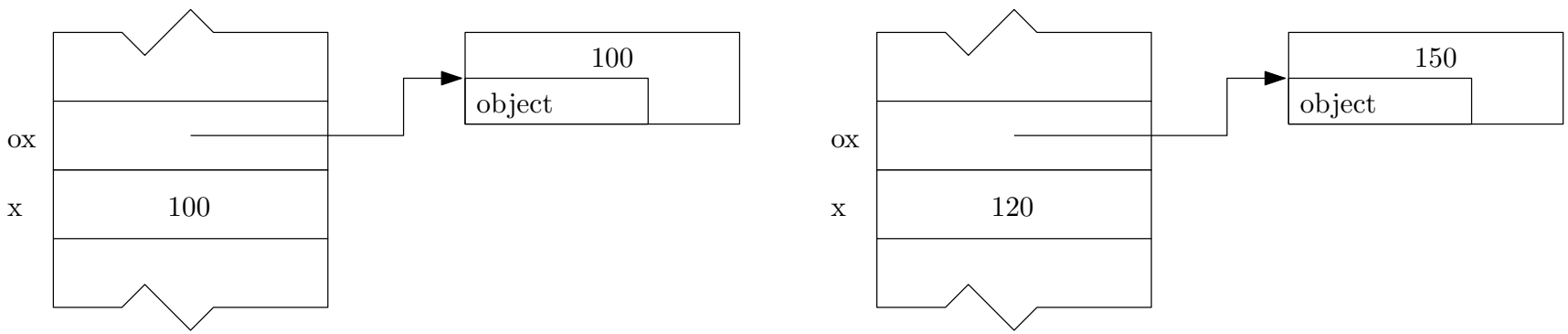


„Pakowanie” / „Pudełkowanie” zmiennych

Stosowanie techniki pudełkowania zawsze wykonuje kopię wartości, ale zachowuje typ.

```
int x = 100; object ox = x;
Console.WriteLine("x: {0}, ox: {1}", x, ox);
i = 12; oi = 15;
Console.WriteLine("x: {0}, ox: {1}", x, ox);
```

Ilustracja dla powyższego kodu:



Rezultat to:

```
x: 10, ox: 10
x: 12, ox: 15
```

Zmienne typu „nullable”

Główne zadanie zmienne typu „nullable”, to dopuszczenie do przyjęcia wartości null, co pozwala na np.: sprawdzenie czy określona zmienna została zainicjalizowana.

```
int? nullableInt = 10;
double? nullableDouble = 3.14;
bool? nullableBool = null;
char? nullableChar = 'a';
int?[] arrayOfNullableInts = new int?[10];
string? s = "oops"; <-- typ łańuchowy jest typem referencyjnym
```

Lepszym rozwiązaniem jest stosowanie wzorca `System.Nullable<T>`:

```
Nullable<int> nullableInt = 10;
Nullable<double> nullableDouble = 3.14;
Nullable<bool> nullableBool = null;
Nullable<char> nullableChar = 'a';
Nullable<int>[] arrayOfNullableInts = new int?[10];
```

Zmienne typu „nullable” – operator ??

Operator ?? (ang. null-coalescing operator) określa wartość domyślną w przypadku otrzymania wartości null:

```
int? x = null;
int y = x ?? -1;
int i = o.GetHeight() ?? default(int);
```

może być stosowany do konwersji typu akceptującego null do typu nieakceptujących wartości null.
Operator ?? działa także z typami referencyjnymi:

```
string s = GetStringValue();
Console.WriteLine(s ?? "wartość nieokreślona");
```

Rodzaje komentarzy

Podstawowy komentarz to:

```
/* początek
     i dalej
        i dalej
koniec komentarza */
```

Komentarz jedno liniowy:

```
// początek i koniec komentarza
```

Komentarz dla dokumentacji:

```
/// <summary>
/// This class does...
/// </summary>
class Program {
...
}
```

Polecenia preprocesora

Kompilator C# nie posiada oddzielnego preprocesora, a także nie pozwala na tworzenie mak
przeznaczeniem dostępnych poleceń jest przede wszystkim kompilacja warunkowa:

1. **#if**, **#else**, **#elif**, **#endif**,
2. **#define**, **#undef**, (polecenie kompilatora **/define**)
3. **#warning**, **#error**,
4. **#line**, **#region**, **#endregion**,
5. **#pragma**, **#pragma warning**, **#pragma checksum**.

Hello World!!!

```
using System;

namespace Example1 {
    class Program {
            static void Main() {
            Console.WriteLine("Witojcie!!!");
        }
    }
}
```

Uwaga

Domyślnie metoda **Main** jest deklarowana jako statyczna (nie trzeba tworzyć obiektu danej klasy), a także prywatna. W ten sposób inny proces nie będzie mógł uruchomić aplikacji, ponieważ punkt wejścia reprezentowany przez metodę **Main** nie jest publiczny.

Klasy i obiekty

Pojęcie klasy i obiektu jest kluczowe w języku w C#, co oznacza że nie np.: zmiennych globalnych, zmienne muszą być umieszczane wewnątrz innych klas.

Ogólna struktura programu C#

Schemat struktury programów tworzonych w języku C#

```
using ...
using System;
using ...

namespace PrzestrzeńNazw
{
    class Klasa
    {
    }

    struct Struktura
    {
    }

    interface IInterfejs
    {
    }

    delegate int Delegat();
```

```
enum TypEnum
{
}

namespace ZagnieżdżonaPrzestrzeńNazw
{
    struct Struktura
    {
    }
    ...
}

class KlasaGłówna
{
    static void Main(string[] args)
    {
        ...
    }
}
}
```

Parametr powrotny metody Main

```
using System;

namespace Example2 {
    class Program {
            static int Main() {
            Console.WriteLine("+---------------+");
            Console.WriteLine("|  Witojcie!!!  |");
            Console.WriteLine("+---------------+");

            Console.ReadLine();

            return -1;
        }
    }
}
```

Uwaga

Wartość powrotna równa zero oznacza iż program zakończył swoje działanie z sukcesem, inna powinna oznaczać kod błędu. Zakłada się iż wartości ujemne reprezentują błędy.

Odczytanie wartości kodu powrotnego – Windows

```
@echo off
Example2-app.exe
@if "%ERRORLEVEL%" == "0" goto success
:fail
    echo Error!
    echo return value = %ERRORLEVEL%
    goto end
:success
    echo Succees!
    echo return value = %ERRORLEVEL%
    goto end
:end
    echo All Done.
```

Uwaga

Odczytanie wartości jest możliwe dzięki zmiennej środowiskowej **%ERRORLEVEL%**.

Odczytanie wartości kodu powrotnego – BASH

```
#! /bin/sh

Example2-fail

ret_value=$?
if [ $ret_value -eq 0 ] ; then
   echo "Success!"
echo $ret_value
else
    echo "Error!"
 echo $ret_value
fi
```

Uwaga

Odczytanie wartości parametru powrotnego polega na odczytaniu predefiniowanej zmiennej **$?**.

Argumenty przekazane do programu

```
using System;

namespace Example3 {
    class Program {
        static int Main(string[] args) {
            Console.WriteLine("Argumenty: ");

            foreach(string arg in args)
                Console.WriteLine("     {0}", arg);

            Console.ReadLine();

            return 0;
        }
    }
}
```

Uwaga

Ponieważ argumenty przekazywane do programy są zapisane w tablicy args, elementy tej tablicy można odczytać za pomocą instrukcji **foreach**.

Metody w klasie (funkcje/procedury)

Parametry metody mogą być opatrzone trzema dodatkowymi słowami kluczowymi: **params**, **ref**, **out**. Słowo kluczowe **params** oznacza dowolną liczbę parametrów:

```
public static void UseParamsKeyword(params int[] list)
{
    for (int i = 0; i < list.Length; i++)
    {
    }
}
...
UseParamsKeyword(1,3,5,7)
```

Możliwe jest także przekazanie parametrów w postaci tablicy:

```
int[] intTab = { 5, 6, 7, 8, 9 };
UseParamsKeyword(intTab);
```

Metody w klasie (funkcje/procedury)

Typ poszczególnych parametrów może być dowolny:

```
public static void UseParamsKeyword(params object[] list)
{
    for (int i = 0; i < list.Length; i++)
    {
    }
}
...
UseParamsKeyword(1, 's', 5, "ciąg znaków")
```

Słowo kluczowe **ref** oznacza, że wartość jest przekazywana w postaci referencji:

```
static void Method(ref int i)
{
...
}
```

Podobnie do **ref** działa słowo kluczowe **out**, ale **ref** wymaga zmiennej zainicjalizowanej.

Klasa System.Environment

```
string[] theArgs = Environment.GetCommandLineArgs();
    foreach(string arg in theArgs)
        Console.WriteLine("     {0}", arg);
```

Uwaga

Tabela z argumentami w klasie **Environment** obejmuje także tzw. argument zerowy, czyli nazwę aplikacji.

Kilka podstawowych informacje o systemie:

```
foreach (string drive in Environment.GetLogicalDrives())
    Console.WriteLine("Drive: {0}", drive);

Console.WriteLine("OS: {0}", Environment.OSVersion);
Console.WriteLine("Number of processors: {0}", Environment.ProcessorCount);
Console.WriteLine(".NET Version: {0}", Environment.Version);
```

Wybrane własności klasy **System.Environment**

Własności klasy **System.Environment** pozwalają na odczytanie podstawowych informacji

Własność	Opis
ExitCode	Kod powrotny aplikacji
Is64BitOperatingSystem	Czy obecny jest system 64bitowy
MachineName	Nazwa komputera
NewLine	Reprezentacja symbolu nowej linii
ProcessorCount	Liczba dostępnych procesorów
StackTrace	Ślad stosu w trakcie wykonywania aplikacji
SystemDirectory	Katalog systemu
UserName	Nazwa użytkownika

Podstawianie zmiennych

Metoda **Environment.ExpandEnvironmentVariables(str)** pozwala na podstawianie zmiennych środowiska jak np.: **%SystemDrive%**:

```
String str;
String nl = Environment.NewLine;

String query = "System drive is %SystemDrive% and system root is %SystemRoot%";
str = Environment.ExpandEnvironmentVariables(query);
Console.WriteLine("ExpandEnvironmentVariables: {0}  {1}", nl, str);
```

Formatowanie ciągu znaków

Wyrażenia {0}, {1} pozwalają na wstawianie wartości zmiennych/wartości przekazywanych po przecinku, w metodach z rodziny **Write**.

Klasa **System.Console**

Kilka wybranych metod i własności z klasy **System.Console**:

Metody/Własności	Opis
Beep(int32, int32)	Wydaje dźwięk
BackgroundColor	Kolor tła
Clear()	Kasowanie zawartości konsoli
ForegroundColor	Kolor czcionki
BufferHeight, BufferWidth	Wielkość bufora konsoli
Title	Tytuł okna konsoli
WindowHeight, WindowWidth	Wymiary okna
WindowTop, WindowLeft	Lewy górny róg
ReadKey, ReadKey(Boolean)	Odczytanie znaku z konsoli/klawiatury
SetCursorPosition	Ustalenie pozycji kursora na ekranie
Write(...), WriteLine(...)	Wysłanie tekstu do konsoli/na ekran

Przykład z kolorami

Zmiana koloru czcionki i przywrócenie oryginalnego koloru:

```
Console.Write("Pierwszy: ");
string u1 = Console.ReadLine();
Console.Write("Drugi: ");
string u2 = Console.ReadLine();

ConsoleColor prevColor = Console.ForegroundColor;
Console.ForegroundColor = ConsoleColor.Yellow;

Console.WriteLine("Pierwszy {0} drugi {1} jakiś tekst.", u1, u2);

Console.ForegroundColor = prevColor;
```

Metoda WriteXY

Wyświetlenie ciągu znaków pod wskazanymi współrzędnymi:

```
protected static int origRow;
protected static int origCol;

static void Init()
{
    origRow = Console.CursorTop;
    origCol = Console.CursorLeft;
}

static void WriteXY(string s, int x, int y)
{
    Console.SetCursorPosition(origCol+x, origRow+y);
    Console.Write(s);
}
```

Formatowanie wyjścia

Podstawowa konstrukcja do formatowania wyjścia, to wyrażenie { a }, gdzie *a* to indeks parametru:

```
Console.WriteLine("{0}, tekst {0}, drugi tekst {0}", 9); // 9, tekst 9, drugi tekst 9
```

Kolejność jest dowolna:

```
Console.WriteLine("{1}, {0}, {2}", 10, 20, 30);
```

Uwaga

Przekroczenie liczby parametrów, czyli podanie większego numeru w nawiasie sześciennym powoduje zgłoszenie wyjątku w czasie wykonania aplikacji.

Formatowanie danych liczbowych

Ogólnie wyrażenie formatowania ma postać: {index[,alignment][:formatString]}:

Znak formatowania	Opis
C lub c	formatowanie walutowe
D lub d	liczba dziesiętna
E lub e	format wykładniczy
F lub f	format stałoprzecinkowy
G lub g	format ogólny
N lub n	format numeryczny
X lub x	liczba szesnastkowa

Wyrażenie „D9” określa pole o 9 znakach, dopełnienie zerami.

```
Console.WriteLine("c format: {0:c}", 99999);
Console.WriteLine("d9 format: {0:d9}", 99999);
Console.WriteLine("f3 format: {0:f3}", 99999);
Console.WriteLine("n format: {0:n}", 99999);
```

```
Wartość 99999 w różnym formatowaniu:
c format: 99 999,00 zł
d9 format: 000099999
f3 format: 99999,000
n format: 99 999,00
E format: 9,999900E+004
e format: 9,999900e+004
X format: 1869F
x format: 1869f
```

Wyrównywanie wyrażenia tekstowego

Wyrównanie do wielkości pola znakami spacji np.: pole walutowe o wielkości szesnastu znaków {0,16:c}.

```
const double tipRate = 0.18;
double billTotal = 45.0;
double tip = billTotal * tipRate;
Console.WriteLine();
Console.WriteLine("Kwota:\t{0,16:c}", billTotal);
Console.WriteLine("Podatek:\t{0,4:c} ({1:p1})", tip, tipRate);
Console.WriteLine(("").PadRight(32, '-'));
Console.WriteLine("Razem:\t{0,16:c}", billTotal + tip);
```

Powtarzanie znaków np.: formowanie linii podsumowania:

```
Console.WriteLine(("").PadRight(32, '-'));
```

Zakresy typów podstawowych

Odczytanie zakresu wybranych typów podstawowych:

```
Console.WriteLine("Max of int: {0}", int.MaxValue);
Console.WriteLine("Min of int: {0}", int.MinValue);
Console.WriteLine("Max of int64: {0}", Int64.MaxValue);
Console.WriteLine("Min of int64: {0}", Int64.MinValue);
Console.WriteLine("Max of double: {0}", double.MaxValue);
Console.WriteLine("Min of double: {0}", double.MinValue);
Console.WriteLine("double.Epsilon: {0}", double.Epsilon);
Console.WriteLine("double.PositiveInfinity: {0}", double.PositiveInfinity);
Console.WriteLine("double.NegativeInfinity: {0}", double.NegativeInfinity);
```

Wynik działania jest następujący:

```
Max of int: 2147483647
Min of int: -2147483648
Max of int64: 9223372036854775807
Min of int64: -9223372036854775808
Max of double: 1,79769313486232E+308
Min of double: -1,79769313486232E+308
double.Epsilon: 4,94065645841247E-324
double.PositiveInfinity: +nieskończoność
double.NegativeInfinity: -nieskończoność
```

Metody dla typów bool i char

Analogicznie dla typu logicznego można odczytać wartości prawdy i fałszu:

```
Console.WriteLine("bool.FalseString: {0}", bool.FalseString);
Console.WriteLine("bool.TrueString: {0}", bool.TrueString);
```

Analogicznie istnieją metody dla znaków sprawdzające czy znak jest np.: cyfrą **IsDigit**:

```
Console.WriteLine("char.IsDigit('a'): {0}", char.IsDigit(myChar));
Console.WriteLine("char.IsLetter('a'): {0}", char.IsLetter(myChar));
Console.WriteLine("char.IsWhiteSpace('Ciąg ze spacją', 5): {0}", char.IsWhiteSpace("Ciąg ze spacją", 5));
Console.WriteLine("char.IsWhiteSpace('Ciąg ze spacją', 6): {0}", char.IsWhiteSpace("Ciąg ze spacją", 6));
Console.WriteLine("char.IsPunctuation('?'): {0}", char.IsPunctuation('?'));
```

Metoda **Parse** zamiana ciągu znaków na wartość:

```
bool b = bool.Parse("True");
double d = double.Parse("99.884");
int i = int.Parse("8");
char c = Char.Parse("w");
```

Zamiana znaku na duża i odpowiednią małą literę: `char.ToUpper( . )`, `char.ToLower( . )`. Metoda `ToString()` zamiana wartości na reprezentację znakową.

Ciągi znaków

W języku C# ciąg znaków (w kodowanych w standardzie unicode) jest reprezentowany przez klasę String, dla uproszczenia wprowadzono słowo kluczowe string. Przykłady deklaracji ciągów znaków:

```
string message1;
string message2 = null;
string message3 = System.String.Empty;
string oldPath = "c:\\Program Files\\Microsoft Visual Studio 8.0";
string newPath = @"c:\Program Files\Microsoft Visual Studio 9.0";
System.String greeting = "Hello World!";
var temp = "I'm still a strongly-typed System.String!";
const string message4 = "You can't get rid of me!";
char[] letters = { 'A', 'B', 'C' };
string alphabet = new string(letters);
```

Ważna właściwość typu string

Ciągi znaków typu string są niezmienne (ang. immutable), co oznacza iż zmodyfikowane ciągi są zapisywane pod innymi adresami niż oryginalne, nadmiarowe kopie są zarządzane przez „garbage collector".

Sekwencje „escape”

Escape sequence	Character name	Unicode encoding
\’	Single quote	0x0027
\"	Double quote	0x0022
\\	Backslash	0x005C
\0	Null	0x0000
\a	Alert	0x0007
\b	Backspace	0x0008
\f	Form feed	0x000C
\n	New line	0x000A
\r	Carriage return	0x000D
\t	Horizontal tab	0x0009
\U	Unicode escape sequence for surrogate pairs	\Unnnnnnnn
\u	Unicode escape sequence	\u0041 = „A”
\v	Vertical tab	0x000B
\x	Unicode escape sequence similar to u except with variable length	\x0041 = „A”

Wybrane metody

Łączenie dwóch ciągów za pomocą operatora „+”, równość/nierówność operatory „==”, „!=”, wybrany spis metod:

Nazwa	Opis
Length	Długość ciągu znaków
Compare()	porównanie dwóch łańcuchów
Contains()	sprawdzenie obecności ciągu znaków
Contact()	łączenie ciągów znaków
Copy()	powielenie ciągu znaków
Equals()	równość ciągów znaków
Format()	formatowanie ciągu znaków
IndexOf()	indeks wskazanego znaku bądź ciągu
Insert()	wstawienie ciągu znaków
PadLeft()	pole z wyrównaniem do lewej strony
PadRight()	pole z wyrównaniem do prawej strony
Remove()	usunięcie wskazanego podciągu znaków
Replace()	zmiana wskazanej zawartości na inną
Split()	rozdział wg. separatora, tworzy tablicę ciągów znaków
ToUpper()	konwersja do dużych liter
ToLower()	konwersja do małych liter
Trim()	usunięcie poprzedzających/uzupełniających białych znaków

Klasa System.Text.StringBuilder

Klasa wspomagające tworzenie i formatowanie ciągów znaków we wskazanym buforze (ang. mutable):

```
StringBuilder sb = new StringBuilder("ciąg początkowy");
sb.Append("\n");
sb.AppendLine("kolejna linia 1");
sb.AppendLine("kolejna linia 2");
sb.AppendLine("kolejna linia 3");
sb.AppendLine("kolejna linia 4");
Console.WriteLine(sb.ToString());
```

Inny przykład, pokazujący zmienną wielkość bufora:

```
StringBuilder sb = new StringBuilder("ABC", 50);
sb.Append(new char[] { 'D', 'E', 'F' });
sb.AppendFormat("GHI{0}{1}", 'J', 'k');
Console.WriteLine("{0} chars: {1}", sb.Length, sb.ToString());

sb.Insert(0, "Alphabet: ");
sb.Replace('k', 'K');
Console.WriteLine("{0} chars: {1}", sb.Length, sb.ToString());
```

W rezultacie otrzymuje się:

```
11 chars: ABCDEFGHIJk
21 chars: Alphabet: ABCDEFGHIJK
```

Zadania typu wyliczeniowego

O typie wyliczeniowym można powiedzieć, że

- Tego rodzaju typy zawierające zbiór nazwanych stałych (np.: nazwy dni),
- stosowanie tego typu pozwala na łatwiejsze czytanie kodu – wartości o znaczących nazwach,
- ułatwia pisanie/tworzenie kodu – nowoczesne środowiska podpowiadają listę możliwych wartości,
- ułatwiają także zadanie utrzymania kodu, gdyż typ wyliczeniowy określa zbiór stałych a zmienne, które przyjmują wartości tylko z tego zbioru.

```
enum Color { Red, Green, Blue };

Color a = Color.Red;
Color b = Color.Green;
Color c = Color.Blue;

Console.WriteLine("Values of Color type: ");
foreach(string s in Enum.GetNames(typeof(Color))) {
    Console.WriteLine(s);
}
Console.WriteLine("Is Blue value of Color type: {0}", Enum.IsDefined(typeof(Color),"Blue"));
Console.WriteLine("Is Yellow value of Color type: {0}", Enum.IsDefined(typeof(Color), "Yellow"));
```

Tablice w C#

Tablice zawierają elementy o takim samym typie, podstawowe własności tablic są następujące:

1. tablice mogą być jedno-, wielowymiarowe, dostępne są także tablice-„jagged”,
2. domyślna wartość tablicy o elemencie numerycznym zero, dla typów referencyjnych domyślna wartość to null,
3. tablice-jagged są tablicami tablic, i elementy są inicjalizowane wartościami null,
4. tablice są indeksowane od zera do wartości n-1,
5. elementami tablicy mogą być dowolnego typu, również tablice,
6. tablice są typem referencyjnym i dziedziczą z abstrakcyjnego typu Array, implementuje interfejs IEnumerable, co pozwala na współpracę z konstrukcją foreach.

```
class TestArraysClass {
    static void Main() {
        int[] array1 = new int[5];
        int[] array2 = new int[] { 1, 3, 5, 7, 9 };
        int[] array3 = { 1, 2, 3, 4, 5, 6 };
        int[,] multiDimensionalArray1 = new int[2, 3];
        int[,] multiDimensionalArray2 = { { 1, 2, 3 }, { 4, 5, 6 } };
        int[][] jaggedArray = new int[6][];
        jaggedArray[0] = new int[4] { 1, 2, 3, 4 };
    }
}
```

„Jagged”-tablice

„Poszarpane” tablice, to tablice o elementach typu tablicowego, elementy tego typu tablicy mogą posiadać różne rozmiary i wymiary.

```
int[][] arr = new int[2][];

// inicjalizacja elementów:
arr[0] = new int[5] { 1, 3, 5, 7, 9 };
arr[1] = new int[4] { 2, 4, 6, 8 };

// wyświetlenie zawartości tablicy:
for (int i = 0; i < arr.Length; i++)
{
    System.Console.Write("Element({0}): ", i);

    for (int j = 0; j < arr[i].Length; j++)
    {
        System.Console.Write("{0}{1}", arr[i][j],
                    j == (arr[i].Length - 1) ? "" : " ");
    }
    System.Console.WriteLine();
}
```

Rezultat jest następujący:

```
Element(0): 1 3 5 7 9
Element(1): 2 4 6 8
```

Przykłady operacji na tablicach

Metody Copy, Clone, CopyTo klasy Array:

```
int[] tab1 = {  1, 2, 3, 4, 5, 6, 7, 8, 9 };
int[] tab2 = { 11,12,13,14,15,16,17,18,19 };

Array.Copy(tab1,2,tab2,4,4);

foreach (int i in tab2) {
    Console.Write("{0}, ",i);
}
```

Rezultat:

```
11, 12, 13, 14, 3, 4, 5, 6, 19,
```

Sortowanie elementów:

```
Array.Sort(tab1);
foreach (int i in tab1) {
    Console.Write("{0}, ", i);
}
```

Inne metody to: Reverse, Exists, FindLast, FindAll, FindIndex, FindLastIndex.

Blok instrukcji, if

Rola bloku instrukcji { ... } jest taka sama jak w języku C,C++ czy Java. Zasada konstrukcji warunku if również jest ogólnie podobna, jednak ze względu na fakt iż C# jest silnie typowalny, to warunek zawsze musi przyjmować wartość logiczną:

```
if(stringData.Length > 0)
{
Console.WriteLine("ciąg nie jest długości zero");
}
```

Nie można określać warunku w stylu C:

```
if(stringData.Length)
{
  ...
}
```

Instrukcja switch

Instrukcja switch działa podobnie jak wersja w C/C++ ale pozwala także na podawanie zmiennych innego typu niż typy numeryczne:

```
string langChoice = Console.ReadLine();
switch (langChoice) {
    case "Abc":
        Console.WriteLine(" tekst 1");
    break;
    case "def":
        Console.WriteLine(" tekst 2");
    break;
    default:
        Console.WriteLine(" tekst domyślny");
    break;
}
```

for, while, do ... while

Pętle typu for, while, do ... while funkcjonują tak samo jak w języku C/C++, przy czym obowiązuje uwaga o wartości logicznej we warunku:

```
int n = 5;
while (++n < 6)
{
    Console.WriteLine("Current value of n is {0}", n);
}
```

Uwaga

Pętle tego typu można przerwać za pomocą słów kluczowych: break, goto, return bądź throws (zgłoszenie wyjątku).

foreach

Konstrukcja językowa ułatwiające współpracę z typami tablicowymi, kolekcjami oraz dowolnym typem użytkownika, który implementuje interfejs System.Collections.IEnumerable lub `System.Collections.Generic.IEnumerable<T>`:

```
int[] fibarray = new int[] { 0, 1, 2, 3, 5, 8, 13 };
foreach (int i in fibarray)
{
    System.Console.WriteLine(i);
}
```

Konstrukcja foreach naturalnie uwzględnia strukturę zmiennej:

```
int[,] num2D = new int[3, 2] { { 9, 99 }, { 3, 33 }, { 5, 55 } };
foreach (int i in num2D)
{
    System.Console.Write("{0} ", i);
}
```

Wyrażenia regularne – klasa Regex

Przestrzeń **System.Text.RegularExpressions** zawiera klasy przeznaczone do obsługi wyrażeń regularnych. Klasa główna do obsługi wyrażeń to **Regex**.
Przykłady języka wyrażeń:

- ^\d{3} – słowo musi rozpoczynać się od trzech cyfr np.: „123-”,
- \d*.\d – cyfra z kropką lub liczba kropka i cyfra np.: „.0”, „12.12”, „120.123”,
- \d{3,5} – liczby składające się od trzech do pięciu cyfr,
- `ab(e|is|at)` – słowa abe, abis, abat, pionowa kreska (ang. pipe) oznacza alternatywę,
- `(?<duplicateWord>\w+)\s\k<duplicateWord>\W(?<nextWord>\w+)` albo `\b(?<word>\w+)\s+(\k<word>)\b` – wykrywa powtarzające się słowa.

Usuwanie formy grzecznościowej

Krótki przykład dla metody **Replace**, która odczytuje tylko dopasowany wzorzec:

```
using System;
using System.Text.RegularExpressions;

public class Example11
{
   public static void Main()
   {
      string pattern = "(Pan |Pani |Panna )";

      string[] names = { "Pan Henry Łowca",
                         "Pana Syliwa Szumowna",
                         "Pan Abraham Adamowski",
                         "Pani Nicole Czerwiec" };

      foreach (string name in names)
         Console.WriteLine(Regex.Replace(name, pattern, String.Empty));
   }
}
```

Poprawność adresu e-mail

Poniższy przykład pochodzi z dokumentacji .NET 4.0 i jest przeznaczony do sprawdzania poprawności adresu e-mail:

```
using System;
using System.Text.RegularExpressions;

public class Example12
{
    public class RegexUtilities
    {
        public static bool IsValidEmail(string strIn)
        {
            return Regex.IsMatch(strIn,
                @"^(?(""")(""".+?""@)|(([0-9a-zA-Z]((\.(?!\.))|[-!#\$%&'\*\+/=\?\^`\{\}\|~\w])*)(?<=[0-9a-zA-Z])@))" +
                @"(?(\[)(\[(\d{1,3}\.){3}\d{1,3}\])|(([0-9a-zA-Z][-\w]*[0-9a-zA-Z]\.)+[a-zA-Z]{2,6}))$");
        }
    }

   public static void Main()
   {
      string[] emailAddresses = { "imie.nazwisko@nazwa.com",
                                  "i..m@server1.albonazwa2..com" };
      foreach (string emailAddress in emailAddresses)
      {
         if (RegexUtilities.IsValidEmail(emailAddress))
            Console.WriteLine("Poprawny: {0}", emailAddress);
         else
            Console.WriteLine("Niepoprawny: {0}", emailAddress);
      }
   }
}
```

Nowy element platformy .NET 4.0, są to tzw. duże liczby całkowite, zawarte w przestrzeni **System.Numerics**.

```
BigInteger biggy = BigInteger.Parse("999999999...99999999999999");
Console.WriteLine("Wartość {0}", biggy);
Console.WriteLine("Parzystość: {0}", biggy.IsEven);
Console.WriteLine("Is biggy a power of two?: {0}", biggy.IsPowerOfTwo);
BigInteger reallyBig = BigInteger.Multiply(biggy,
                                BigInteger.Parse("88888888888...888888"));
Console.WriteLine("Value of reallyBig is {0}", reallyBig);
```

Konwersja do typu BigInteger:

```
try {
    posBigInt = BigInteger.Parse(positiveString);
    Console.WriteLine(posBigInt);
}
catch (FormatException)
{
    Console.WriteLine("Bład konwersji '{0}' to a BigInteger value.",
                posStr);
}

if (BigInteger.TryParse(negativeString, out negBigInt))
    Console.WriteLine(negBigInt);
else
    Console.WriteLine("Bład konwersji '{0}' to a BigInteger value.",
                      negStr);
```

Silnia

```
static BigInteger bi_factorial(BigInteger n)
{
    BigInteger i, r = 1;
    for (i = 1; i <= n; i++) r = r * i;

    return r;
}

static BigInteger bi_NaiveBinomial(BigInteger n, BigInteger k)
{
    var fn = bi_factorial(n);
    var fk = bi_factorial(k);
    var fnk = bi_factorial(n - k);

    return fn / (fk * fnk);
}
```

Prosty test dla 12! oraz 121!:

```
silnia 12 = 479001600
silnia 121 = 809429852527344373968162284544935082997082306309701607045776233628 4
  976604266405217133917739979101827382870741850789049568566634393183827450477162 14
  841147650721760223072092160000000000000000000000000000
```

Przykład obsługi wyjątku

Typowy przykład z dzieleniem przez zero:

```
int x = 0;
try {
    int y = 100 / x;
}
catch (ArithmeticException e) {
    Console.WriteLine("Obsługa wyjątku arytmetycznego: {0}", e.ToString());
}
catch (Exception e) {
    Console.WriteLine("Obsługa wyjątku ogólnego: {0}", e.ToString());
}
```

Uwaga

Deklaracja przechwycenia wyjątków szczegółowych powinna pojawić się wcześniej przed deklaracją wyjątku ogólnego.

Lepsza, bezpieczniejsza wersja WriteXY

```
static void WriteXY(string s, int x, int y)
{
try
    {
    Console.SetCursorPosition(origCol+x, origRow+y);
    Console.Write(s);
    }
catch (ArgumentOutOfRangeException e)
    {
    Console.Clear();
    Console.WriteLine(e.Message);
    }
}
```

Wywołanie funkcji **PtInRect**

Współpraca z kodem niezarządzanym jest istotna dla wydajności, niestety kosztem bezpieczeństwa.

```
using System.Runtime.InteropServices;

[StructLayout(LayoutKind.Sequential)]
public struct Point {
    public int x;
    public int y;
}

[StructLayout(LayoutKind.Explicit)]
public struct Rect {
    [FieldOffset(0)] public int left;
    [FieldOffset(4)] public int top;
    [FieldOffset(8)] public int right;
    [FieldOffset(12)] public int bottom;
}
```

```
class Win32API {
    [DllImport("User32.dll")]
    public static extern bool PtInRect(ref Rect r, Point p);
}
```

Program Czas i Data w WinAPI

Przykład pochodzi z dokumentacji:

```
[StructLayout(LayoutKind.Sequential)]
public class MySystemTime {
    public ushort
               wYear;
    public ushort
               wMonth;
    public ushort
               wDayOfWeek;
    public ushort
               wDay;
    public ushort
               wHour;
    public ushort
               wMinute;
    public ushort
               wSecond;
    public ushort
               wMilliseconds;
}
class Win32API {
    [DllImport("Kernel32.dll")]
    public static extern void GetSystemTime(MySystemTime st);

    [DllImport("user32.dll", CharSet=CharSet.Auto)]
     public static extern int MessageBox(IntPtr hWnd,
         string text, string caption, int options);
}
```

Program Czas i Data w WinAPI

Przykład pochodzi z dokumentacji:

```
public class TestPlatformInvoke
{
    public static void Main()
    {
        MySystemTime sysTime = new MySystemTime();
        Win32API.GetSystemTime(sysTime);

        string dt;
        dt = "System time is: \n" +
              "Year: " + sysTime.wYear + "\n" +
              "Month: " + sysTime.wMonth + "\n" +
              "DayOfWeek: " + sysTime.wDayOfWeek + "\n" +
              "Day: " + sysTime.wDay;
        Win32API.MessageBox(IntPtr.Zero, dt, "Platform Invoke Sample", 0);
    }
}
```

W następnym tygodniu między innymi

Zagadnienia poruszane na następnym wykładzie:

1. model obiektowy,
2. obiekty i klasy,
3. dziedziczenie, słowo kluczowe sealed
4. typy uogólnione, kolekcje,
5. lambda wyrażenia,
6. konwersje pomiędzy typami, operatory „is” oraz „as”

Proponowane tematy prac pisemnych:

1. opracowanie odpowiednika narzędzia ildasm dla platformy MONO – raport z projektu ,
2. ”brudzenie kodu” – narzędzia, przykłady, przegląd technik i aplikacji stosowanych w tym celu,
3. problem współpracy z kodem niezarządzanym, np.: pakiet OGRE3D .NET.

Dziękuje za uwagę!!!