Jazyk C# a platforma .NET

Katedra softwarového inženýrství
Fakulta informačních technologií
České vysoké učení technické v Praze

Další rozšíření jazyka C#
BI-DNP

```
using System;
using System.Collections.Generic;
using System.Windows.Forms;
// pridano!!
using System.Linq.Expressions;

namespace DalsiRozsireni
{
    public partial class Form1 : Form
    {
        public Form1() { InitializeComponent(); }

        private void btnProved_Click(object sender, EventArgs e)
        {
            // inicializace objektu, neni-li vhodny konstruktor:
            // (trida definovana dale)
            Employee objEmp = new Employee() {
                      FirstName = "Franta", LastName = "Novak"};

            txtVystup.Text = objEmp.Display();

            // lze uzit i v kombinaci s (existujicim) konstruktorem:
            // Employee objEmp2 = new Employee("Novak")
                                    { FirstName = "Franta" };

            // inicializace kolekci (museji implementovat
            // IEnumerable a mit metodu Add)
            // Potom misto:
            List<int> cisla = new List<int>();
            cisla.Add(1);
            cisla.Add(2);  // atd.

            // lze psat:
            List<int> cisla2 = new List<int>() { 1, 2, 3, 4, 5, 6 };

            // lze pouzit i pro vlastni typy
            List<Employee> lide = new List<Employee>(){
            new Employee(){FirstName = "Franta", LastName = "Novak"},
            new Employee(){FirstName = "Venca", LastName = "Cerny"},
            new Employee(){FirstName = "Eva", LastName = "Cerna"}
            };
```

```
// "typ" var

// inicializace povinna; stane se stringem:
var x = "Ahoj!";
txtVystup.Text = x.GetType().Name; // string
// x = 1;  // nelze - bude stringem navzdy!!

var z = 1;
// z = "Ahoj!"; // opet nelze

// u ciselnych typu lze presneji specifikovat
// typ pomoci suffixu:
var dec = 10M; //decimal
var lng = 10L; //long


// anonymni typy (tridy)
// kompilator prideli jmeno typu, ktere se ale nedozvime

// odvozeno primo od Object, vlastnosti ReadOnly;
// pouziva se v LINQ apod.
var anonymni = new { ID = 1, Jmeno = "Franta" };
// anonymni.ID = 5;  // nelze - read  only
txtVystup.Text = anonymni.ID + "; " + anonymni.Jmeno;


// extension methods - pridani metod k existujicimu typu
// (tride), aniz bychom ji redefinovali nebo dedili
// lze pouzit i u sealed trid
// definuje se jako zcela samostatna staticka metoda
// (viz dale), ktera se na "rozsirenem" typu pouziva
// jako instancni metoda
// extension method i rozsirovany typ museji by ve
// stejnem namespace nebo pouzit using

// pouziti extension metody:
string s = "ABCD";
txtVystup.Text = s.ToQuotedString();

// pro null nepada!!
s = null;
txtVystup.Text = s.ToQuotedString();
```

```
// Expression trees: kod se prevede na stromovou
// strukturu (pracuje se s nim jako s daty)
// jiny kod jej muze analyzovat, upravit nebo opet
// prevest na kod a spustit

// Pro tvorbu expression trees muzeme vyuzit statickych
// tovarnich metody tridy Expression z namespace
// System.Linq.Expressions. V tomto namespace jsou take
// definovany typy, predstavujici jednotlive listy
// stromu, jako napr.:
// ParameterExpression pro vytvoreni vstupního parametru
// nebo MethodCallExpression pro vytvoreni volani metody.

// Nasledujici priklad ukazuje sestaveni expression
// tree predstavujici lambda vyraz pro secteni dvou
// celych cisel. Sestaveni se provadi pomoci metod tridy
// Expression.

// tvorba parametru pro lambda vyraz
ParameterExpression par1 =
                Expression.Parameter(typeof(int), "a");
ParameterExpression par2 =
                Expression.Parameter(typeof(int), "b");

// tvorba vlastního lambda vyrazu pro secteni dvou cisel
Expression<Func<int, int, int>> addExpression =
          Expression.Lambda<Func<int, int, int>>
             (Expression.Add(par1, par2),
              new ParameterExpression[] { par1, par2 });

// Nastesti existuje i snazsi zpusob tvorby expression
// tree a to vyuzitim sluzeb kompilatoru,
// ktery je schopen na zakladě instance delegata,
// (vytvoreneho napr. pomoci lambda vyrazu), vytvorit
// expression tree automaticky. Kompilator totiz v
// pripade pouziti typu Expression<T>, kde T je typ
// delegata, pozna, ze nema s kodem zachazet jako se
// spustitelnym, ale že ma vytvorit expression tree.

Expression<Func<int, int, int>> addExpr = (a, b) => a+b;

//spusteni zkompilovaneho delegata
txtVystup.Text = addExpression.Compile()(2,3).ToString();
```

```
            // podobne lze i exp. tree pomoci jeho verejnych
            // vlastnosti snadno analyzovat a zjistit tak vsechny
            // informace o danem vyrazu. Nasledujici kod zobrazi
            // parametry a telo výrazu, ale moznosti analyzy
            // expression trees jsou mnohem sirsi

            Expression<Func<int, int, int>> exprTree = (a, b) => a+b;
            txtVystup.Text = "Parameters:\r\n";

            //vypsani vstupnich argumentu a jejich typu
            foreach (ParameterExpression parExpr in
                                        exprTree.Parameters) {
               txtVystup.Text += parExpr.Name + " - " +
                                 parExpr.Type + "\r\n";
            }
        }
    }

    // Auto-implemented properties
    // (tato trida je pouzita i pri ukazce inicializace objektu)
    class Employee
    {
        // autoimplemented properties museji obsahovat jak get,
        // tak set; ale mohou byt private - viz dale
        public string FirstName {
            get;
            set;
        }

        public string LastName;  // pro ukazku neni property

        public int ID {
            get;
            private set;  // private!!
        }

        public Employee(){
            // v konstruktoru (a metodach tridy) lze vlozit
            // do privatni property, zvenku ne
            // (.Next vraci int >= 0 a < 1000)
            ID = new Random().Next(1000);
        }

        public string Display() {  // jen pro vypis
            return LastName + " " + FirstName;
        }
    }
```

```
// castecne (partial) metody: jen v partial types (classes) nize
// neumoznuji modifikatory pristupu (jsou vzdy private) – volani
// jen v ramci typu
// pouzivano napr. pro graficke generovani casti kodu, ke kteremu
// je treba "rucne" dopsat zbytek (a nenarusit generovane)

 // Soubor 1:

 partial class PartialMethods
 {
      public void SomeMethod()
      {
          PartialMethodOne();
          PartialMethodTwo();
      }

      // metoda je naimplementovana v druhem souboru castecneho
      // (partial) typu
      partial void PartialMethodOne();

      //metoda neni implementovana
      partial void PartialMethodTwo();
 }

 // Soubor 2:
 partial class PartialMethods
 {
      //implementace castecne metody
      partial void PartialMethodOne(){
          Console.WriteLine("Partial method one ...");
      }
 }
```

```
    // definice extension method
    internal static class Extensions
    {
        // this u prvniho vstupniho argumentu metody (NUTNE)
        // oznacuje typ, ktery se rozsiruje (nikoli bezne this)

        // casto object misto string
        internal static string ToQuotedString(this string str) {
            return "\"" + str + "\"";
        }

        // mohou byt pretezovane se "standardnimi" parametry
        internal static string ToQuotedString(this string str,
                                              char chrQuote) {
            return chrQuote + str + chrQuote;
        }
    }
}
```