

Programovací jazyk C# Verze jazyka 3.0

Ing. Marek Běhálek
Katedra informatiky FEI VŠB-TUO
A-1018 / 597 324 251
<http://www.cs.vsb.cz/behalek>
marek.behalek@vsb.cz

V přednášce jsou použity části prezentace stažené z:
<http://mif.netstudent.cz>, autor: Tomáš Petříček

Náplň kapitoly

- Základní informace o LINQu
- Stručný úvod do funkcionálního programování
- Nové vlastnosti jazyka C# 3.0
 - Lambda Výrazy
 - Extension metody
 - Dedukce typu lokální proměnné
 - Inicializátory objektů
 - Anonymní typy
 - Dotazovací výrazy
 - Výrazové stromy

C# 3.0

2

Nové vlastnosti jazyka C#

- V této kapitole budete seznámeni s rozšířeními jazyka C# 3.0
 - Koncepce LINQ projektu
 - Dominantní technologií je OOP
 - Jsou už používané více než 20 let
 - Stále existují neobjektové zdroje dat a informací
 - Relační databáze, XML data, ...
 - V současné době velmi nepohodlná integrace
 - Přibýly ale i jiné vlastnosti na úrovni programovacího jazyka
 - Přejata řada prostředků z funkcionálních jazyků
 - Umožňuje jiný „styl“ programování.
 - „Ad hoc“ přístup při přidávání pro zajištění zamýšlené funkcionality.

C# 3.0

3

Tradiční přístup k práci s daty

```
using(SqlConnection conn = new SqlConnection
("server=localhost;database=mojeDatabase;uid=sa;pwd=");
{
    conn.Open();
    SqlCommand cmd =new SqlCommand(@"
SELECT * FROM [users] WHERE login=@login
AND passwr=@pass",conn);
cmd.Parameters.Add(@"@login",SqlDbType.VarChar);
cmd.Parameters.Add(@"@heslo",SqlDbType.VarChar);
cmd.Parameters["@login"].Value = login;
cmd.Parameters["@pass"].Value = heslo;

// ...
cmd.ExecuteReader()
}
```

C# 3.0

4

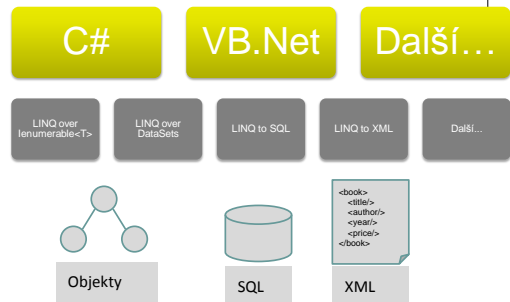
Základní vlastnosti LINQ

- Cíle projektu LINQ
 - Typová kontrola při překladu
 - Eliminace běhových chyb
 - Jednodušší zápis, který je součástí jazyka
 - Není nutné učit se další jazyk
 - Větší bezpečnost
 - Odpadají problémy s SQL injection
 - Rozšíření .NET jazyků pro práci s neobjektovými daty
- Rozšíření jazyka
 - Základem jsou dotazovací operátory
 - filtrování, projekce, seskupování, atd.. - lze používat na libovolné .NET kolekce
 - Rozšiřitelnost LINQ dotazů
 - Pro vlastní objekty lze operátory reimplementovat
 - Technicky je LINQ projekt postavený na .NET 2.0 (plná zpětná kompatibilita)

C# 3.0

5

LINQ - Language integrated query



C# 3.0

6

Funkcionální jazyky – Jazyk a architektura počítače

- Omezení jazyku na to, co lze efektivně implementovat na současných procesorech.
- Von Neumannova architektura
 - Model klasických procesorů
 - Základ klasických jazyků
- Funkcionální jazyky
 - Backus (1978, Turing Award) – kritika přístupu „od architektury k jazyku“
 - Funkcionální jazyky jsou efektivnější než imperativní
 - Lze dokazovat vlastnosti programů
 - Jednoduše je lze paralelizovat
 - Založeno na algebraických pravidlech
 - Malá efektivita implementace – možné optimalizace

C# 3.0

7

Funkcionální jazyky – Rozdíly mezi imperativními a deklarativními jazyky

- Imperativní jazyky
 - Program má implicitní stav, který se modifikuje konstrukcemi programovacího jazyka.
 - Explicitní pojem „pořadí“ příkazů
 - Vyjadřuje, jak se má program vyhodnocovat
 - Vychází z aktuální (Von Neumannovy) architektury počítače
 - Jednoduchá a efektivní realizace
- Deklarativní jazyky
 - Program nemá implicitní stav.
 - Program je tvořen výrazy, ne příkazy.
 - Popisujeme co se má spočítat, ne jak.
 - Není dáno pořadí příkazu.
 - Program je vyhodnocen redukčním systémem
- **Funkcionální jazyky x Relační (logické) jazyky**

C# 3.0

8

Funkcionální jazyky– Funkcionální programovací jazyky (2)

- Umožňují nové algebraické přístupy
 - **Lazy evaluation** (**x eager evaluation**)
 - Možnost používat potenciaálně nekonečné struktury.
 - Možnost oddělení dat od řízení – nemusíme se starat o to, jak proběhne vyhodnocení.
- Umožňuje nové přístupy k vývoji programů
 - Možnost dokazovat programy
 - Možnost transformovat program na základě algebraických vlastností
- Umožňuje lepší využití paralelního provádění programů
 - Jednoduchá dekompozice programů na části, které lze vyhodnocovat paralelně.
 - Potenciaálně příliš mnoho paralelismů.
 - Možnost kompozice dvou paralelních úloh jednoduchou kompozicí funkcí.

C# 3.0

9

Funkcionální jazyky - λ -kalkul

- 1930 Alonzo Church
 - netypaný λ -kalkul
 - matematická teorie funkcí
 - Proměnné
 - x, y, z, f, g, \dots
 - λ -abstrakce
 - $(\lambda x . e)$
 - Aplikace
 - $(e_1 e_2)$
 - Konvence pro závorky
 - $\lambda x . \lambda y . e_1 e_2 = (\lambda x . (\lambda y . e_1 e_2))$
 - $e_1 e_2 e_3 = ((e_1 e_2) e_3)$

C# 3.0

10

Funkcionální jazyky– Příklady

- Vytvoření seznamu druhých mocnin
 - $dm [] = []$
 $dm (x:xs) = sq\ x : dm\ xs$
 where $sq\ x = x * x$
- Seřazení seznamu (quicksort)
 - $qs [] = []$
 $qs (x:xs) =$
 let $ls = filter (< x)\ xs$
 $rs = filter (>=x)\ xs$
 in $qs\ ls ++ [x] ++ qs\ rs$

C# 3.0

11

Přehled novinek v C# 3.0

- Lambda Výrazy `c => c.Name`
- Extension metody `static void Dump(this object o);`
- Dedukce typu lokální proměnné `var x = 5;`
- Inicializátory objektů `new Point { x = 1, y = 2 }`
- Anonymní typy `new { c.Name, c.Phone }`
- Dotazovací výrazy `from ... where ... select`
- Výrazové stromy `Expression<T>`

12

Lambda výrazy

- Zobecněná syntaxe funkce
 - $\lambda x. x + 1$
 - v C# 3.0 je $x \Rightarrow x + 1$
- Vycházejí z anonymních delegátů
 - `delegate(int x) { return x + 1; }`
- Rozšiřují delegáty z .NET
 - Kromě možnosti předat lambda výraz jako delegát je možné předat lambda výraz jako „expression tree“.
- Jednoduchý způsob jak psát funkce, které lze předávat jako argumenty k vyhodnocení.

C# 3.0

13

Lambda výrazy

```
public delegate bool Predicate<T>(T obj);
```

```
public class List<T>
```

```
{  
    > FindAll(Predicate<T> test) {
```

Explicitně
typový

Příkazy

```
        List<Customer> customers =  
        GetCustomerList();
```

```
        List<Customer> x = customers.FindAll(  
        delegate(Customer c) { return c.State == "WA"; }  
        );
```

Lambda výraz

```
        List<Customer> x = customers.FindAll(c => c.State == "WA");
```

Implicitně typový seznam
Lambda parametru
(využívá inferenci typu)

C# 3.0

14

Lambda výrazy

```
Func<Customer, bool> test = c => c.State == "WA";
```

```
double factor = 2.0;  
Func<double, double> f = x => x * factor;
```

```
Func<int, int, int> f = (x, y) => x * y;
```

```
Func<int, int, int> comparer =  
(int x, int y) => {  
    if (x > y) return 1;  
    if (x < y) return -1;  
    return 0;  
};
```

C# 3.0

15

Čím se liší lambda výraz a delegát?

- Lambda výraz lze přeložit
 - Jednak do podoby kódu (jako delegát)
 - Jednak do podoby dat
Výrazový strom (Expression Tree)
- Na data lze aplikovat za běhu optimalizaci, překlad do jiného jazyka, ...
- Díky výrazovým stromům může LINQ to SQL překládat dotazy na SQL výrazy

Expression trees

```
public class Northwind: DataContext
{
    public Table<Customer> Customers;
    public Table<Order> Orders;
    ...
}
```

```
Northwind db = new Northwind(...);
var query = from c in db.Customers where c.State == "WA" select c;
```

```
Northwind db = new Northwind(...);
var query = db.Customers.Where(c => c.State == "WA");
```

```
public class Table<T>: IEnumerable<T>
{
    public Table<T> Where(Expression<Func<T, bool>> predicate);
    ...
}
```

```
System.Expressions.
Expression<T>
```

Jak se to však spustí?

Metoda potřebuje výrazový strom

17

Kompilace lambda výrazu

- Z lambda výrazu lze získat kód

```
Func<Customer, bool> test = c => c.State == "WA";
```

- Nebo výrazový strom (data)

```
Expression<Func<Customer, bool>> test = c => c.State == "WA";
```

```
ParameterExpression c =
    Expression.Parameter(typeof(Customer), "c");
Expression expr =
    Expression.EQ(
        Expression.Property(c,
            typeof(Customer).GetProperty("State")),
        Expression.Constant("WA"));
Expression<Func<Customer, bool>> test =
    Expression.Lambda<Func<Customer, bool>>(expr, c);
```

18

Query expressions

- Dotazy se překládají do volání metod
 - Where, Select, SelectMany, OrderBy, GroupBy

```
from c in customers
where c.State == "WA"
select new { c.Name, c.Phone };
```

```
customers
.Where(c => c.State == "WA")
.Select(c => new { c.Name, c.Phone });
```

Query expressions

- Převod výrazu na volání jednotlivých metod je vždy syntakticky korektní
- Nemusí však být sémanticky
 - Neexistující metody, chybné typy parametrů, nefunguje inference typů (generické metody)
 - Poznává se to při překladač!

Extension methods

- Extenzní metody umožňují rozšířit veřejné rozhraní typů
 - V současné době pouze o metody
- Technická realizace
 - Jsou definovány jako statické typy
 - V metadatach označeny atributem [System.Runtime.CompilerServices.Extension]
 - První parametr je deklarován modifikátorem this
- V případě LINQ se přidávají operátory
 - Ke všem objektům implementujícím IEnumerable<T>
 - (metody Select, Where, atd...)

C# 3.0

21

Extension metody

```
namespace System.Query
{
    public static class Sequence
    {
        public static IEnumerable<T> Where<T>(this IEnumerable<T> source,
            Func<T, bool> predicate) { ... }

        public static IEnumerable<S> Select<T, S>(this IEnumerable<T>
            source,
            Func<T, S> selector) { ... }
    }
    ...
}
using System.Query;

IEnumerable<string> contacts =
    customers.Where(c => c.State == "WA").Select(c =>
        c.Name);
```

Extension metody

Přinese extensions do rozsahu

obj.Foo(x, y)
↓
XXX.Foo(obj, x, y)

IntelliSense!

C# 3.0

22

Extension methods

```
public static class Extensions
{
    public static int ToInt32(this string s) {
        return Int32.Parse(s);
    }
    public static T[] Slice<T>(this T[] source,
        int index, int count) {
        if (index < 0 || count < 0 ||
            source.Length - index < count)
            throw new ArgumentException();
        T[] result = new T[count];
        Array.Copy(source, index, result, 0, count);
        return result;
    }
}
```

C# 3.0

23

Extension methods

```
using N1;
namespace N1
{
    public static class E
    {
        public static void F(this object obj, int i) { }
        public static void F(this object obj, string s) { }
    }
}
class A { }
class B
{
    public void F(int i) { }
}
class C
{
    public void F(object obj) { }
```

C# 3.0

24

Extension methods

```
static void Main(string[] args)
{
    A a=new A(); B b=new B(); C c=new C();
    a.F(1); a.F("hello");
    b.F(1); b.F("hello");
    c.F(1); c.F("hello");
    Console.ReadLine();
}
```

• Výstup:
E F 1
E F hello
B F 1
E F hello
C F 1
C F hello

C# 3.0

25

Object initialization expressions

```
public class Point
{
    private int x, y;
    public int X { get { return x; } set { x = value; } }
    public int Y { get { return y; } set { y = value; } }
}
```

Přifazení položky
nebo vlastnosti

```
Point a = new Point { X = 0, Y = 1 };
```

```
Point a = new Point();
a.X = 0;
a.Y = 1;
```

C# 3.0

26

Object initialization expressions

```
public class Rectangle
{
    private Point p1 = new Point();
    private Point p2 = new Point();
    public Point P1 { get { return p1; } }
    public Point P2 { get { return p2; } }
}
```

Vložené
objekty

Read-only
vlastnosti

```
Rectangle r = new Rectangle {
    P1 = { X = 0, Y = 1 },
    P2 = { X = 2, Y = 3 }
};
```

Ne "new Point"

```
Rectangle r = new Rectangle();
r.P1.X = 0;
r.P1.Y = 1;
r.P2.X = 2;
r.P2.Y = 3;
```

27

C# 3.0

Collection initialization

```
List<int> powers = new List<int> { 1, 10, 100, 1000, 10000 };
```

Musí implementovat
ICollection<T>

```
List<int> powers = new List<int>();
powers.Add(1);
powers.Add(10);
powers.Add(100);
powers.Add(1000);
powers.Add(10000);
```

28

C# 3.0

Anonymní typy & klíčové slovo var

- Co když v projekci potřebuju vrátet složený typ?
 - Potřeba snadno vytvářet typy s danými vlastnostmi
 - Nepotřebujeme ale znát jméno typu...
- Anonymní typy
 - POZOR, neplést s typem „object“ a dynamickým voláním
 - Typ je známý, má známé vlastnosti, pouze nemá jméno
- Klíčové slovo „var“
 - Typ výrazu bude odvozený z kontextu

C# 3.0

29

Type inference pomocí var

```
int i = 5;
string s = "Hello";
double d = 1.0;
int[] numbers = new int[] { 1, 2, 3 };
Dictionary<int,Order> orders = new Dictionary<int,Order>();
```

```
var i = 5;
var s = "Hello";
var d = 1.0;
var numbers = new int[] { 1, 2, 3 };
var orders = new Dictionary<int,Order>();
```

„var“ znamená stejný typ jako inicializátor

30

C# 3.0

Anonymní typy

```
public class Customer
{
    public string Name;
    public Address Address;
    public string Phone;
    public List<Order> Orders;
    ...
}

public class Contact
{
    public string Name;
    public string Phone;
}

Customer c = GetCustomer(...);
Contact x = new Contact { Name = c.Name, Phone = c.Phone };

Customer c = GetCustomer(...);
var x = new { Name = c.Name, Phone = c.Phone };

Customer c = GetCustomer(...);
var x = new { c.Name, c.Phone };
```

Inicializace s využitím projekce

31

C# 3.0

Anonymní typy

```
var contacts =
    from c in customers
    where c.State == "WA"
    select new { c.Name, c.Phone };

IEnumerable<??>

class ???
{
    public string Name;
    public string Phone;
}

var contacts =
    customers
    .Where(c => c.State == "WA")
    .Select(c => new { c.Name, c.Phone });

foreach (var c in contacts) {
    Console.WriteLine(c.Name);
    Console.WriteLine(c.Phone);
}
```

C# 3.0

32

Omezení anonymních typů

- Pouze lokální proměnné
- Deklarace musí obsahovat inicializaci
 - Inicializátor musí být výraz
 - Nesmí být objekt nebo kolekce objektů
- Typ inicializačního výrazu nesmí být null

Automatic properties

```
public class Product
{
    string name;
    decimal price;

    public string Name {
        get { return name; }
        set { name = value; }
    }

    public decimal Price {
        get { return price; }
        set { price = value; }
    }
}
```

private string □;
public string Name {
 get { return □; }
 set { □ = value; }
}

```
public class Product
{
    public string Name { get; set; }
    public decimal Price { get; set; }
}
```

Musí obsahovat
jak get tak set

C# 3.0

34

Partial Methods

```
partial class Customer
{
    public string Name {
        get { ... }
        set {
            OnNameChanging(value);
            _name = value;
            OnNameChanged();
        }
    }
    partial void OnNameChanging(string value);
    partial void OnNameChanged();
}
```

Partial Method Call

Partial Method Definition

C# 3.0

35