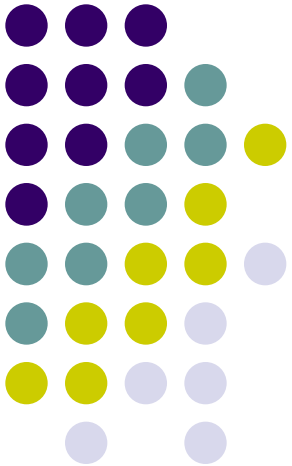
Programovací jazyk C#

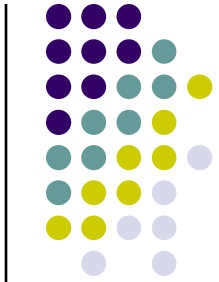
Verze jazyka 3.0

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



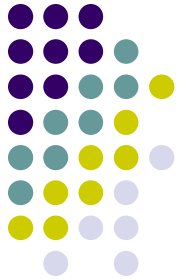
V přednášce jsou použity části prezentace stažené z:
http://mff.netstudent.cz, autor: Tomáš Petříček

Náplň kapitoly



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

Nové vlastnosti jazyka C#

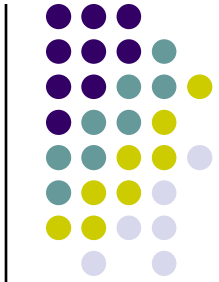


- V této kapitole budete seznámeni s rozšířeními jazyka C# 3.0
  - Koncepce LINQ projektu
    - Dominantní technologií je OOP
      - Jsou už používané více než 20 let
    - Stále existují neobjektové zdroje dat a informací
      - Relační databáze, XML data, ...
      - V současné době velmi nepohodlná integrace
  - Přibyly 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.

Tradiční přístup k práci s daty

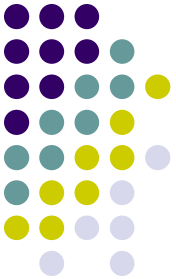
```
using(SqlConnection conn = new SqlConnection

("server=localhost;database=mojeDatabaze;uid=sa;pwd=");
 {
      conn.Open();
      SqlCommand cmd =new SqlCommand(@"
        SELECT * FROM [users] WHERE login=@login
           AND passwrd=@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()
}
```

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)

LINQ - Language integrated query



C#

VB.Net

Další…

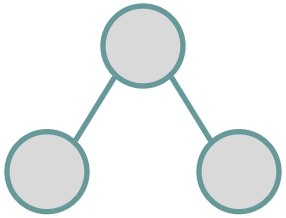
LINQ over Ienumerable<T>

LINQ over DataSets

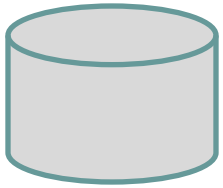
LINQ to SQL

LINQ to XML

Další...



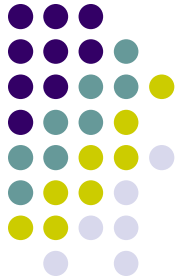
Objekty



SQL

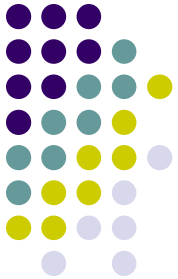
XML

Funkcionální jazyky – Jazyk a architektura počítače



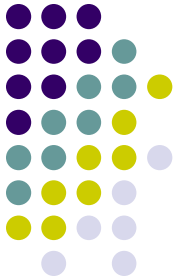
- Omezení jazyku na to, co lze efektivně implementovat na současných procesorech.
- Von Neumannova architektura
  - Model klasických procesorů
  - Základ klasických jazyků
- Funkcionální jazyky
  - Backus (1978, Turing Adward) – kritika přístupu „od architektury k jazyku“
  - Funkcionální jazyky jsou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

Funkcionální jazyky– Rozdíly mezí 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 Neummanovy) architektury počítačů
    - 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**

Funkcionální jazyky– **Funkcionální programovací jazyky (2)**

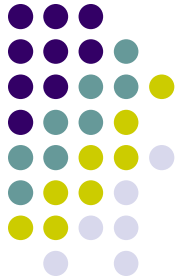


- Umožňují nové algebraické přístupy
  - **Lazy evaluation** (x **eager evaluation**)
    - Možnost používat potencionálně nekonečné struktury.
  - Možnost oddělení dat od řízení – nemusíme se starat o to, jak proběhne vyhodnocení.
- Umožňuje nové přístupy k vývoji programů
  - Možnost dokazovat programy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

Funkcionální jazyky - λ-kalkul

- 1930 Alonzo Church
  - netypovaný λ-kalkul
  - matematická teorie funkcí
  - Proměnné
    - x, y, z, f, g, …
  - λ-abstrakce
    - (λx . e)
  - Aplikace
    - (e_1 e_2)
  - Konvence pro závorky
    - λx . λy . e_1 e_2 = (λx . (λy . e_1 e_2))
    - e_1 e_2 e_3 = ((e_1 e_2) e_3)

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
    ```

Přehled novinek v C# 3.0

- Lambda Výrazy
- Extension metody
- Dedukce typu lokání proměnné
- Inicializárory objektů
- Anonymní typy
- Dotazovací výrazy
- Výrazové stromy

```
c => c.Name
```

```
static void Dump(this object o);
```

```
var x = 5;
```

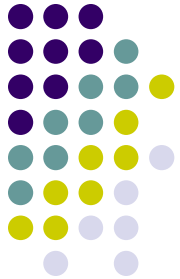
```
new Point { x = 1, y = 2 }
```

```
new { c.Name, c.Phone }
```

```
from … where … select
```

```
Expression<T>
```

Lambda výrazy



- Zobecněná syntaxe funkce
  - λ x . x + 1
  - v C# 3.0 je `x => 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.

Lambda výrazy

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

public class List<T>
{
    > FindAll(Predicate<T> test) {
```

```
    omer> customers =
    GetCu  merList();
```

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

Explicitně typový

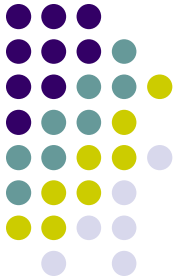
Příkazy

Lamda výraz

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

Implicitně typový seznam Lambda parametrů (využívá inferenci typu)

Lambda výrazy



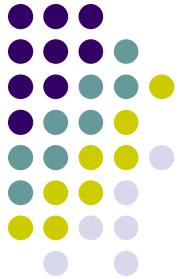
```
Func<Customer, bool> test = c => c.State == "WA";
```

```
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;
    };
```

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);
    …
}
```

Jak se to však spustí?

Metoda potřebuje výrazový strom

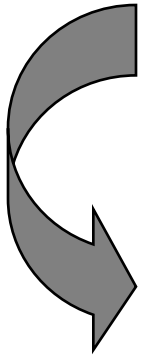
System.Expressions.
Expression<T>

Kompilace lambda výrazu

- Z lambda výrazu lze získat kód

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

- Nebo výrazový strom (data)

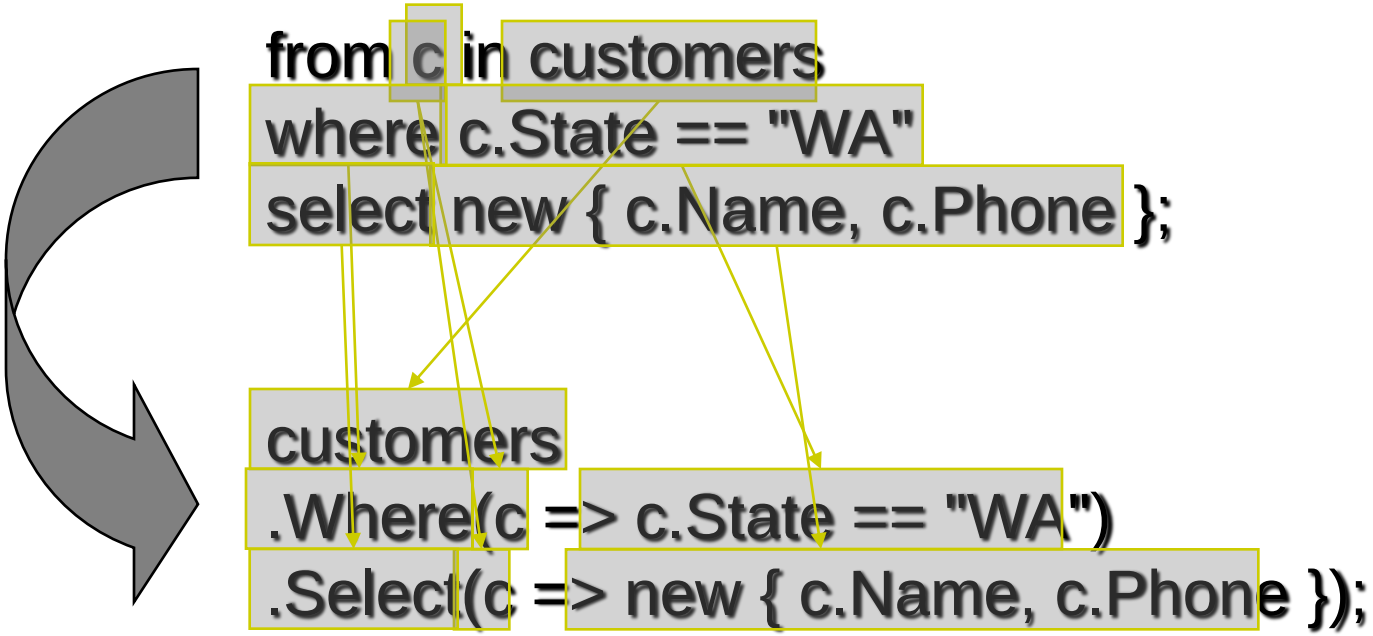


```
Expression<Func<Customer, bool>> test = c => c.State == "WA";
```

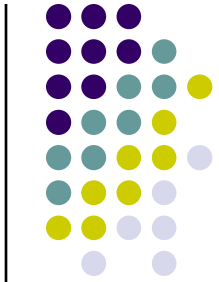
```
ParameterExpression c =
    Expression.Parameter(typeof(Customer), "c");
Expression expr =
    Expression.EQ(
        Expression.Property(c,
typeof(Customer).GetProperty("State")),
        Expression.Constant("WA")
    );
Expression<Func<Customer, bool>> test =
    Expression.Lambda<Func<Customer,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

Query expressions

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



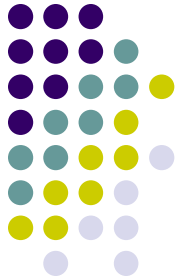
Query expressions



- Převod výrazu na volání jednotlivých metod je vždy syntakticky korektní

- Nemusí však být sémanticky
  - Neexistující metody, chybné typy parametrů, nefunguje inference typů (generické metody)
  - Pozná se to při překladu!

Extension methods



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

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) { … }
        …
    }
}
```

Extension metody

```
using System.Query;
```

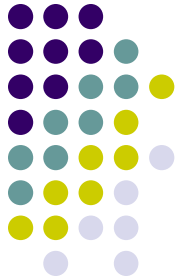
Přinese extensions do rozsahu

obj.Foo(x, y)
↓
XXX.Foo(obj, x, y)

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

IntelliSense!

Extension methods

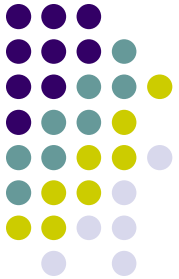


```
public static class Extensions
  {
    public static int ToInt32(this string s) {
       return Int32.Parse(s);
    }
    public static T[] Slice<T>(this T[] source,
                               int index, int count) {
      if (index < 0 || count < 0 ||
          source.Length - index < count)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

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) { }
}
```

Extension methods

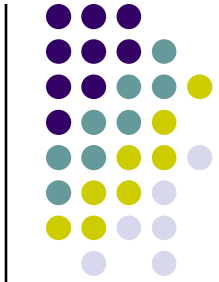


```
static void Main(string[] args)
        {
            A a=new A(); B b=new B(); C c=new C();
            a.F(1); a.F("hello");
            b.F(1); b.F("hello");
            c.F(1); c.F("hello");
            Console.ReadLine();
        }
```

- Výstup:

```
E F 1
E F hello
B F 1
E F hello
C F 1
C F hello
```

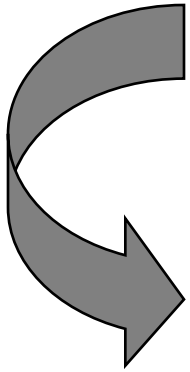
Object initialization expressions



```
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; } }
}
```

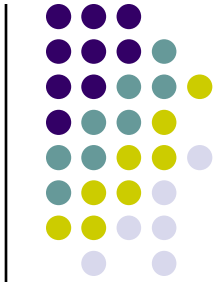
Přiřazení položky nebo vlastnosti

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



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

Object initialization expressions



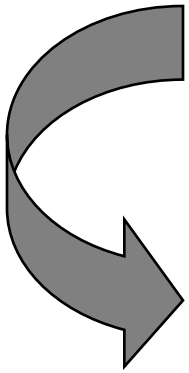
```
public class Rectangle
{
    private Point p1 = new Point();
    private Point p2 = new Point();

    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”

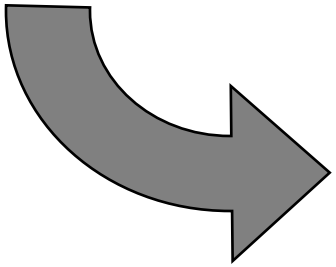


```
Rectangle r = new Rectangle();
r.P1.X = 0;
r.P1.Y = 1;
r.P2.X = 2;
r.P2.Y = 3;
```

Collection initialization

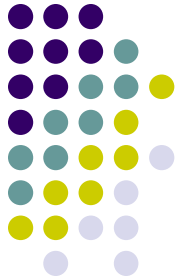
Musí implementovat ICollection<T>

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



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

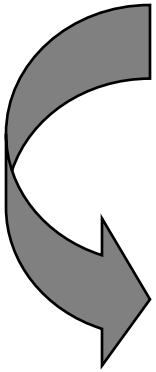
Anonymní typy & klíčové slovo var

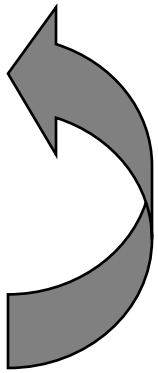


- Co když v projekci potřebuju vracet složený typ?
  - Potřeba snadno vytvářet typy s danými vlastnostmi
  - Nepotřebujeme ale znát jméno typu...
- Anonymní typy
  - POZOR, neplést s typem „object“ a dynamickým voláním
  - Typ je známý, má známé vlastnosti, pouze nemá jméno
- Klíčové slovo „var“
  -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

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

Anonymní typy

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

```
public class Contact
{
    pub
    pub
}
```

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

```
Customer c = GetCustomer(…);
Contact x = new Contact { Name = c.Name, Pho          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

Anonymní typy

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

IEnumerable<???>

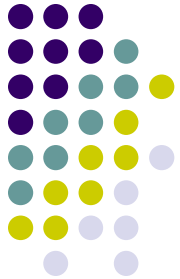
```
class ???
{
    public string Name;
    public string Phone;
}
```

```
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);
}
```

Omezení anonymních typů



- Pouze lokální proměnné
- Deklarace musí obsahovat inicializaci
  - Inicializátor musí být výraz
  - Nesmí být objekt nebo kolekce objektů
- Typ incializačního 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; }
    }
}
```

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

```
private string □;

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

Musí obsahovat jak get tak set

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