Nové prvky v C# 3.0

David Keprt

revize Aleš Keprt 11.12.2006

Úvod

- Obsahem této prezentace je představení nových konstrukcí, které do jazyka C# přináší verze 3.0. Mezi ně patří:
  - Klíčové slovo „var“
  - Nové možnosti inicializace objektů a kolekcí
  - Anonymní typy
  - Rozšiřující metody
  - Lambda výrazy
  - LINQ

Klíčové slovo ‘var’ (1/2)

- Slouží k deklaraci implicitně typovaných proměnných.
- Příklad:

```
// kód z C# 2.0
    int i = 10;
    string s = "Hello people!";

// kód stejné funkčnosti v C# 3.0
    var i = 10;
    var s = "Hello people!";
```

Klíčové slovo ‘var’ (2/2)

- Překladač vyhodnotí pravou stranu deklarace a najde typ, který daným datům odpovídá.
- Výsledná proměnná, stejně jako celý jazyk, je stále staticky typovaná

  (na rozdíl od podobně vypadajícího, ale zcela odlišného výrazu ‘variant’ známého například z VB 6)
- Typ je proměnné napevno stanoven při překladu její deklarace, tudíž například následující kód překladač zamítne:

```
var b;
b = 1337;
```

Inicializace objektů a kolekcí (1/2)

- Nová syntaxe umožňující kratší inicializaci objektů vypadá takto:

```
// kód z C# 2.0
Person p = new Person(); // Create object
p.Name = "Nick"; // Initialize property Name
p.Gender = "Male"; // Initialize property Gender
p.Active = true; // Initialize property Active

// kód stejné funkčnosti v C# 3.0
Person p = new Person { Name = "Nick", Gender = "Male", Active = true };
```

Inicializace objektů a kolekcí (2/2)

- Podobně lze takto inicializovat i kolekce:

```
// kód z C# 2.0
List<string> animals = new List<string>();
animals.Add("monkey");
animals.Add("cow");
animals.Add("dog");
animals.Add("cat");

// kód stejné funkčnosti v C# 3.0
List<string> animals = new List<string> { "monkey", "cow", "dog", "cat" } ;
```

Anonymní typy (1/3)

- Nová verze jazyka C# nabízí možnost vytvářet instance jednoduché datové třídy bez nutnosti obtěžovat se její deklarací.
- Chcete mít v paměti objekt s daty o vašem psu a nemáte deklarovanou žádnou vhodnou třídu? Nevadí! Stačí následující:

```
new {name=“Cicina”, hair="black", age=6}
```

- A máme vytvořenou instanci. Ale co s ní, když pro založení reference potřebujeme udat typ třídy na jejíž instanci ukazuje? Ano! Použijeme implicitně typovanou proměnnou:

```
var kocka = new {name=“Cicina”, hair="black", age=6};
```

Anonymní typy (2/3)

- Při překladu vytvoří stroj něco jako toto:

```
class __Anonymous1 {
    private string _name = "black";
    private string _hair = "black";
    private int _age = 6;
    public string name {get { return _name; } set { _name = value; }}
    public string hair {get { return _hair; } set { _hair = value; }}
    public int age {get { return _age; } set { _age = value; }}
}
```

Anonymní typy (3/3)

- Překladač navíc není žádný hlupák, a pokud dále v kódu budeme inicializovat instanci anonymní třídy, pak, pokud názvy datových proměnných a typy udaných hodnot budou stejné, místo vyvářeníní další anonymní třídy vytvoří instanci té, kterou již zná.
- Mimo jiné to způsobí i to, že implicitně typované reference pak mají stejný typ, takže je možné napsat například následující:

```
var kocka = new {name=“Cicina”, hair="black", age=6}
var psik = new {name=“Harik”, hair=„blue", age=8}
kocka = psik;
Console.WriteLine(“Cicina uz neni, {0} ji sezral!”, kocka.name);
```

Rošiřující metody (1/2)

- Rozšiřující metody jsou zvláštní nástroj umožňující rozšířit již existující třídu o další metody.
- Tímto nástrojem lze do třídy přidat jen statické metody.
  - Ty pak nemají přístup do privátních dat a nedědí se
  - Volají se však, jakoby statické nebyly ☺
- Navíc jak víme, je snadné původní třídu podědit a novou metodu přidat do potomka, tak k čemu je tento nástroj?
- Jeho využitelnost máme ocenit při práci s třídami, které nechceme nebo nemůžeme dědit.

Rošiřující metody (2/2)

- Pro ilustraci uvedeme příklad doplnění třídy String:

```
public static class StringExtensions {
    public static int ToInt(this string s) {
      return Int32.Parse(s);
    }
}
```

- Po zkompilování výše uvedeného budou mít všechny stringy v naší aplikaci novou metodu, která jejich obsah přeparsuje na integer.

```
string input = „12345“;
int value = input.ToInt();
```

Lambda výrazy (1/5)

- Již první verze jazyka C# nám umožnovala předávat metodám jako parametr jiné metody pomocí delegátů.

```
// kód z C# 1.0
class Program
{
        delegate void DemoDelegate();
        static void Main(string[] args)
        {
            DemoDelegate myDelegate = new DemoDelegate(SayHi);
            myDelegate();
         }
        void SayHi()
        {
            Console.Writeline("Hiya!!") ;
        }
}
```

Lambda výrazy (2/5)

- C# 2.0 nám dovolil tento zápis zjednodušit použitím anonymní metody.

```
// kód z C# 2.0
class Program
{
    delegate void DemoDelegate();
    static void Main(string[] args)
    {
        DemoDelegate myDelegate = delegate()
        {
            Console.Writeline("Hiya!!");
        };
        myDelegate();
    }
}
```

Lambda výrazy (3/5)

- Protože se často stává, že metody, které delegátem předáváme deklarujeme jen k jednomu jedinému použití, tudíž se nepotřebujeme dovolávat jejich jména, ušetří nám anonymní metody několik řádků kódu.
- V C# 3.0 máme jako další nadstavbu k dispozici lambda výrazy. Ty se zapisují jako seznam parametrů, symbol '=>' a výraz či kód určující návratovou hodnotu daného lambda výrazu.

```
// lambda výraz vracející bool který určuje, zda je číslo liché
i => (i % d) == 0
```

- Poznámka:

  Podobnost tohoto zápisu s funkcionálními jazyky jako Scheme nebo Lisp je více než zřejmá.

Lambda výrazy (4/5)

- S pomocí lambda výrazu přepíšeme předchozí kód takto:

```
// kód z C# 3.0
class Program {
    delegate void DemoDelegate();

    static void Main(string[] args) {
      DemoDelegate myDelegate = () => Console.WriteLine("Hiya!!");
      myDelegate();
    }
}
```

Lambda výrazy (5/5)

- Z předchozího příkladu by se mohlo zdát že lambda výrazy jsou jen jakýmsi syntakticky odlišným zápisem anonymních metod, ale není tomu tak. Lambda výrazy tak, jak budou v novém C# uvedeny, mají větší funkcionalitu a jsou významově nadřazeny anonymním metodám. Mezi jejich zajímavé vlastnosti patří:
  - lambda výrazy neudávají typy u vstupů ani výstupů
  - můžou obsahovat buď dotazový výraz, nebo C# kód
  - pomocí nového typu Expression lze lambda výraz zabalit a pracovat s ním jako s daty

Výrazové stromy – okrajová pozámka

- Lambda výraz z předchozího příkladu můžeme vložit do výrazového stromu touto deklarací:

```
Expression<DemoDelegate> filter = () => Console.WriteLine("Hiya!!") ;
```

- S tímto *výrazovým stromem* dále můžeme pracovat jako s datovým objektem a za chodu programu zkoumat a měnit jeho obsah (což bychom s klasickou funkcí pochopitelně nemohli).

LINQ

- LINQ = Language Integrated Query
- LINQ je převratné rozšíření jazyka C# (a VB 9.0), který má vést ke snadnému, přehlednému a objektovému (což je synonymum předešlých adjektiv ☺) propojení databáze a programu.
- Umožňuje přímo v jazyku C# psát konstrukce velice podobné jazyku SQL.
- Konstrukce z klíčových slov SELECT, WHERE, FROM a tak dále, psaných jen v trochu jiné syntaxi a ve zvláštně otočeném pořadí.

- Poznámka:
  - LINQ je natolik komplexní a složitý nástroj, že informace zde uvedené lze brát jedině jako obšírný úvod doplněný několika příklady.

LINQ – příklad

```
DataContext db = new DataContext("server=.;initial catalog=northwind");
Table<Orders> orders = db.GetTable<Orders>();
Table<Customers> customers = db.GetTable<Customers>();

var q =
    from o in orders, c in customers
    where o.ShipCity == "London" && (o.CustomerID == c.CustomerID)
    select new { o.OrderDate, c.CompanyName, c.ContactTitle, c.ContactName
    };

foreach (var item in q) {
    Console.WriteLine("Order for {0} {1} at {2} placed on {3}",
      item.ContactTitle, item.ContactName, item.CompanyName,
      item.OrderDate);
    }
```

Jak LINQ pracuje?

- Kód z předchozí strany je jakýmsi „syntaktickým cukrem“.
- Postupně se přes několik kroků přeloží do spleti volání funkcí, rozšiřujících funkcí a lambda výrazů, konkrétně předchozí kód se přeloží do (poněkud nepřehledného) tvaru:

```
var q = QueryExtensions.SelectMany(
    QueryExtensions.Where(orders, o => o.ShipCity == "London"),
    o => QueryExtensions.Select(
        QueryExtensions.Where(customers, c => o.CustomerID == c.CustomerID),
        c => new { o.OrderDate, c.CompanyName, c.ContactTitle, c.ContactName }));
```

Je LINQ efektivní?

- Z tohoto zápisu se zdá, že první zavolání *QueryExtensions.Where* vyústí v dotaz na databázi, který vyselektuje údaje z tabulky *orders,* s Londýnem jako místem dodání, a *QueryExtensions.SelectMany* vyústí v iterování skrze tyto hodnoty, užívající mnohonásobný databázový dotaz na tabulku customer, a že tímto dojde k mnohonásobnemu joinu těchto tabulek na klientské straně aplikace. Naštěstí, LINQ je mnohem chytřejší.
- Kompiler využívá všechny základní postupy pro optimalizaci SQL dotazování, a tak, k velkému překvapení všech skeptiků, vytvoří jediný komplexní dotaz na databázi, kterým získá jen přesně určený výsek potřebných dat.
- Na internetu lze nalézt mnoho nezávislých článků, jejichž autoři testovali efektivnost demoverze LINQ, a panuje mezi nimi shoda, že alespoň v případě relativně jednoduchých dotazů, jejichž efektivitu je pro člověka snadné ověřit, funguje překlad LINQu velice efektivně.

Překvapivé vlastnosti LINQ

- Důsledky použití lambda výrazů:
  - zpožděné vykonávání
    - Ikdyž by deklarativní zápis dotazu (var lowNumbers = …) mohl vést k opačnému pocitu, dotaz není v místě deklareace vykonán a jeho výsledek uložen do proměnné, nýbrž je do ní uložen sám dotaz.
    - To se projevuje tím, že dokud se někde dále v kódu nedomáháme výstupu z dotazu, není dotaz vůbec proveden. Což může vést k tomu, že mezi dvěma dotazováními na stejný dotaz se změní hodnoty se kterými pracuje, a tudíž i jeho výstup.
  - práce s kolekcemi
    - Zatím jsme o LINQ mluvili pouze jako o nástroji na práci daty z databází. Jak ale víme, lambda výrazy jsou netypové a mohou sestávat z dotazů i konstrukcí běžný pro jazyk C#.
    - Proto lze LINQ dotazy provádět nad jakýmikoliv kolekcemi s rozhraním IEnumerable.
    - Navíc lze divoce kombinovat C# kód a LINQ konstrukce.

Příklad (1/2) – práce s kolekcemi

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

var numberGroups =
    from n in numbers
    group n by n % 5 into g
    select new { Remainder = g.Key, Numbers = g };

foreach (var g in numberGroups) {
    Console.WriteLine(„Čísla se zbytkem {0} při dělění pěti:", g.Remainder);
    foreach (var n in g.Numbers) {
        Console.WriteLine(n);
    }
}
```

Příklad (2/2) - zpožděné vykonání

```
int[] numbers = new int[] { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

var lowNumbers =
    from n in numbers
    where n <= 3
    select n;

Console.WriteLine(„První volání <= 3:");
foreach (int n in lowNumbers) Console.WriteLine(n);

for (int i = 0; i < 10; i++) numbers[i] = -numbers[i];

// Druhé spuštění stejného dotazu bude iterovat nad změněnými daty
// a vrátí jiný výsledek
Console.WriteLine(“Druhé volání <= 3:");
foreach (int n in lowNumbers) Console.WriteLine(n);
```

Zdroje

- Microsoft .NET LINQ Preview (May 2006)
- http://www.codepost.org/
- http://www.developer.com/
- http://www.interact-sw.co.uk/