Nové prvky jazyka Visual C# 2.0 (2005)

Aleš Keprt

Katedra informatiky, Fakulta elektrotechniky a informatiky
VŠB – Technická Univerzita Ostrava
17. listopadu 15, 708 33, Ostrava-Poruba, Česká Republika
Ales@Keprt.cz Ales.Keprt@vsb.cz

Abstrakt. Zanedlouho nás čeká nová verze jazyka C#, která přinese několik poměrně zásadních novinek přímo na úrovni syntaxe a sémantiky jazyka. Standard příští verze jazyka je již schválen, proto se můžeme na všechny novinky podívat podrobně. Text detailně popisuje všechny důležité nové prvky jazyka C# verze 2.0, v závěru je připojen autorův komentář k nejdůležitějším změnám v jazyku a krátký souhrn novinek ve vývojovém prostředí Visual C# 2005.

Klíčová slova: jazyk C#, C# 2.0, C# 2005, generické typy, iterátory

1 Úvod

Microsoft používá pro svůj .NET Framework, Visual Studio a programovací jazyky dvojí systém číslování verzí – koncové produkty začínající slovem „Visual“ jsou číslovány dle roku vydání, zatímco .NET Framework a jazyky bez slůvka Visual jsou číslovány klasickým číselným označením 1.0 atd. Naštěstí v současnosti existují jen dvě oficiální verze: První z roku 2002 je verzí číslo 1.0, druhá je o rok novější a nese číslo 1.1. Také letos jsme se měli dočkat další verze všech zmíněných produktů, avšak Microsoft později odložil dokončení této další verze s číslem 2.0 až na rok 2005. Toto zpoždění bylo zapříčiněno několika důvody, nám ale především přinese větší množství novinek. Podívejme se nyní, jaké novinky můžeme očekávat v příští verzi jazyka C# (specifikace je již kompletně schválena, podoba jazyka by se tedy již neměla nijak změnit).

2 Generické typy a metody (generics)

Obecné neboli generické typy a metody (generics, také neformálně překládáno jako „generiky“) jsou v podobě šablon (templates) již dlouho součástí jazyka C++ a nedávno se objevily i v jazyku Java. Jelikož je to nejdůležitější, nejrozsáhlejší a nejsložitější nový prvek jazyka C# 2.0, je mu zde věnována celá samostatná kapitola. Generické mohou být třídy, struktury, rozhraní a metody. I přes jisté odlišnosti mezi těmito jazykovými konstrukty, z hlediska generického programování se od sebe navzájem příliš neliší.

2.1 Úvod do generických typů a metod

Cílem generických typů a metod je umožnit vyšší úroveň znovupoužití (reuse) kódu. Své uplatnění najdou všude tam, kde se opakuje stejný kód několikrát pro různé datové typy. Příkladem může být klasická šablona v C++, která definuje generickou funkci `swap` pro záměnu hodnot dvou proměnných libovolného (ale téhož) typu. V C# vypadá kód `swap` takto (je to metoda, umístíme ji do libovolné třídy):

```
void swap<T>(T a, T b) {
  T c = a;
  a = b;
  b = c;
}
```

`T` zde označuje parametr – datový typ, který dosadíme až při použití této generické metody.

```
int a,b;

swap<int>(a,b); //záměna a,b

swap(a,b);      //záměna s použitím typové inference
```

Na posledním řádku ukázky voláme generickou metodu bez uvedení typu jejího parametru. Tato konstrukce funguje díky typové inferenci - překladač „uhodne“ typ podle dosazených parametrů, podobně jako to umí v případě použití operátoru `?:`. V nejednoznačných situacích je inference řešena přesně dle specifikace jazyka, nejlepší však je se jakýmkoliv nejednoznačným kombinacím typů vyhnout.

2.2 Generické a silně typované kolekce

Generické typy jsou tedy jakési metatypy, jejichž instance jsou teprve klasickými třídami. Typickou oblastí využití tohoto nového prvku jazyka je deklarace silně typovaných kolekcí. Za příklad poslouží datová struktura zásobník. Nejprve ukázka kódu v C# 2.0:

```
class Stack<T> {

  T[] data;
  int pos;

  public Stack() {
    data = new T[10];
    pos = 0;
  }
```

```
  public void push(T value) {
    data[pos++] = value;
    if(pos >= data.Length) pos=0;
  }

  public T pop() {
    if(pos==0) pos = data.Length;
    return data[--pos];
  }
}
```

Tento kód definuje jednoduchou implementaci zásobníku pomocí pole pevné délky. `Stack<T>` je generická třída, `T` je parametrem této třídy, za který dosadíme konkrétní známý datový typ při vytváření instancí našeho zásobníku takto:

```
Stack<string> s = new Stack<string>();
```

Zde jsme vytvořili silně typovanou kolekci `Stack<string>` – zásobník nad typem `string`. Tento zásobník tedy odmítne jakýkoliv nekompatibilní typ. Díky silně typovaným kolekcím již také nemusíme používat boxování (zabalování hodnotových typů do objektu), neboť parametry generického typu mohou být jak referenční, tak hodnotové typy. Používání generických tříd pak vede jak k bezpečnějšímu, tak k rychlejšímu kódu (přetypování a zvláště boxování hodnotových typů je velmi *drahá* operace).

2.3 Kompilace a běh programu s generickými typy a metodami

Pro pochopení detailů fungování generických typů a metod je vhodné také uvést, jak jsou tyto programové konstrukty přeloženy.

U hodnotových typů překladač vytvoří novou konkrétní třídu pro každý hodnotový typ, který dosadíme za parametr. Používáme-li na více místech programu stejně parametrizovanou generickou třídu, např. `Stack<int>`, překladač použije vždy stejný kód. Když ale potom použijeme např. `Stack<long>`, je vytvořena nová třída.

V programech se mohou běžně vyskytovat stovky tříd a kdyby programátor chtěl pro každou z nich používat určitou specifickou generickou třídu, znamenalo by to vytvořit stovky instancí kódu této generické třídy, vždy znovu pro každý typ dosazený za parametr. Proto je překlad generických typů v případě referenčních parametrů ve skutečnosti prováděn jinak. Překladač využívá toho, že samotné reference na objekty jsou všechny stejně velké (v paměti), takže ke každé generické třídě je vytvořena jen jedna instance kódu pro všechny referenční typy. Čili např. `Stack<string>` a `Stack<MyClass>` budou sdílet jeden kód. Za běhu programu je samozřejmě možné používat reflexi ke zjištění konkrétních typů.

Co bylo zde uvedeno pro generické třídy, platí samozřejmě i pro ostatní formy generického programování (generické metody, rozhraní a delegáty).

2.4 Omezení typových parametrů, klauzule `where`

Zatímco C++ šablony můžeme používat způsobem hodně podobným textovým makrům, kde při překladu dojde k pouhému nahrazení textového názvu parametru skutečným typem, v C# to tak jednoduše nefunguje. Důvodem je především výše uvedený fakt, že v případě referenčních parametrů je ke každé generické třídě vytvořena (maximálně) jedna instance kódu. C# umožňuje na všech typových parametrech provádět jen obyčejné přiřazování, porovnávání s `null`[1] a volání metod, které jsou v `System.Object`. Ostatní věci lze realizovat jen s použitím klauzule `where T:...`, kde za dvojtečkou napíšeme seznam vyžadovaných vlastností typu `T`. Klauzule `where` umožňuje tato konkrétní použití:

Omezení na konkrétní typ je vyžadováno při volání konkrétních metod či operátorů. Toto omezení se deklaruje uvedením bázového typu nebo rozhraní, kde jsou deklarovány metod/operátory, které chcete použít.
Jako příklad uveďme generickou metodu, která volá na svém parametru metodu `test()`. Aby to bylo možné, musíme deklarovat rozhraní s touto metodou a v deklaraci generické metody pak uvést požadavek, aby typ parametru implementoval toto rozhraní.

```
interface ITestable {
  void test();
}

void test<T>(T obj) where T : ITestable {
  obj.test();
}
```

Zde je vidět zásadní rozdíl oproti C++, tam je totiž možno použít šablony i pro realizaci obecného polymorfizmu (tj. na dvou různých typech lze volat stejně pojmenovanou metodu, aniž by měla stejné parametry nebo pocházela z typu ve stejném stromu dědičnosti). V C# to možné není.

Vytváření nových objektů vyžaduje uvedení `new()` v seznamu vlastností.

```
T create<T>() where T:new() {
  return new T();
}
```

Tato metoda (umístíme ji do libovolné třídy) vytváří objekty typu daného parametrem `T`. Pro přehlednost také uveďme, jak se tato metoda zavolá:

```
MyClass a = create<MyClass>();
```

Tento způsob vytváření objektů lze použít jen u konstruktorů bez parametrů. Potřebujeme-li konstruktor s parametry, použijeme omezení na konkrétní typ, jak je uvedeno výše v případě volání metod.

[1] U hodnotových typů vrací test na `null` vždy `false`.

Omezení na pouze hodnotové nebo pouze referenční typy provedeme použitím omezení `class` nebo `struct`. Praktický dopad těchto omezení je minimální, mají sloužit zejména pro ochranu před lidskými omyly.

```
class RefOnly<T> where T : class {...}
```

Omezení `class` vynutí, aby na místě typového parametru byl referenční typ a umožní nám používat operátor `as`[2].

```
class ValueOnly<T> where T : struct {...}
```

Omezení `struct` vynutí, aby na místě typového parametru byl hodnotový typ a znemožní používat porovnání s `null` (které je však možné u hodnotových typů v generické třídě bez omezení!). Za `struct` se zde považují všechny hodnotové typy, tedy včetně `int`, `char`, apod.

Vícenásobná omezení lze uvádět oddělená čárkou. Jelikož typový parametr může sám být součástí deklarace omezení pro jiný typový parametr, lze vytvořit i deklaraci, jako je tato:

```
class MyClass<A, B, C>
  where A : B
  where B : C {
}
```

Deklarovali jsme generickou třídu se třemi parametry a jako podmínku jsme uvedli, že mezi typovými parametry je vyžadován vztah dědičnosti. Všimněte si, že jednotlivé klauzule `where` nemají mezi sebou žádný oddělovač (jen mezeru, ta ale není syntaktickým prvkem), editor Visual C# proto pro lepší přehlednost zdrojového kódu automaticky umisťuje vícenásobná omezení na samostatné řádky.

Odhalený (naked) parametr je ten, který je uveden jako omezení jiného parametru (viz příklad u předchozího bodu). Použití odhalených parametrů je dosti omezené, lze jimi definovat jen vztah dědičnosti mezi dvěma typovými parametry.

2.5 Dědičnost

Typový parametr nemůže být použit jako bázová třída. Následující kód tedy nefunguje.

```
class Outer<T> {
  class Inner : T {} //Chyba: Třída Inner nemůže dědit třídu T.
}
```

[2] Výrazem `var as Type` rozumíme změnu referenční proměnné `var` na referenci na typ `Type`. Není-li změna reference možná, výsledkem je `null`. Provádí se přitom jen změna reference, bez použití přetypovávacích metod.

Generické typy však mohou být předky ve stromu dědičnosti. V tom případě musíme rozlišit několik případů užití generické třídy.

```
//BaseType nám poslouží jako bázová třída
class BaseType {}

//Pozor! Tato generická třída je další úplně samostatnou třídou!
class BaseType<T> {}

//Type1 dědí konkrétní třídu BaseType
class Type1 : BaseType {}

//Chyba! Nelze zde zjistit typ parametru T
class Type2 : BaseType<T> {}

//Type3 je generická třída dědící z generické třídy  BaseType<T>
class Type3<T> : BaseType<T> {}

//Type4 je generická třída dědící z konkrétní třídy BaseType<int>
class Type4<T> : BaseType<int> {}
```

V ukázce je tedy vidět, že potomek může být buď konkrétní třídou `Type`, nebo generickou třídou `Type<T>`. Navíc může mít různé předky: Musíme rozlišovat mezi konkrétní třídou `BaseType`, generickou třídou `BaseType<T>`, a její instancí (konkrétní třídou) `BaseType<int>`.

Při dědění generických tříd obsahujících omezení (viz kap 2.4) musí generický potomek definovat omezení, která jsou nadmnožinou omezení předka nebo omezení předka je implikují.

2.6 Co všechno může být generické

Zatím byla řeč především o třídách, stejným způsobem lze však deklarovat i generická rozhraní a generické struktury. Samozřejmě platí obvyklá omezení těchto jazykových konstruktů – např. rozhraní nemají konstuktory, struktur se netýká dědičnost, atp.

Jak bylo ukázáno v úvodu kapitoly, generické mohou být i metody. Opět zde „přiměřeně“ platí totéž, co u tříd. Metody uvnitř generických tříd mohou používat typové parametry svých tříd (jinak by to ani nemělo smysl).

```
class MyClass<T> {
  //Chyba! V generické metodě nelze uvést stejný typový parametr.
  void MyMethod<T>() {}

  //Toto je v pořádku - použijeme typ T z mateřské třídy.
  bool IsNull(T obj) {
    return obj==null;
  }
}
```

2.7 Výchozí hodnoty – klauzule `default`

Chceme-li přiřadit do proměnné výchozí hodnotu generického typu, můžeme použít klíčové slovo `default`. Tento nový konstrukt se netýká jen generických typů, ale právě tady najde největší uplatnění.

```
int a = default(int);
MyClass b = default(MyClass);
MyStruct c = default(MyStruct);
```

Výchozí hodnota referenčního typu je vždy `null`. Výchozí hodnota číselného typu je binární nula a u struktur se toto pravidlo aplikuje rekurzivně na všechny vnitřní položky.

3 Další novinky v jazyku C# 2.0

3.1 Iterátory

Pokud bychom do výše uvedeného kódu zásobníku chtěli přidat podporu pro procházení všech prvků příkazem foreach, ve staré verzi C# by to vyžadovalo implementovat v naší třídě rozhraní `IEnumerable`, čili metodu vracející `IEnumerator`. Naše datová struktura je velmi jednoduchá, ale u složitějších datových struktur může být implementace enumerátorů dosti pracná. Proto nyní lze podporu `foreach` zajistit i novým jednodušším způsobem - pomocí iterátorů. Iterátory přinášejí do kódu určitý nový prvek náhledu, patří tak patří mezi složitější z novinek v C# 2.0.

Iterátor je programový blok, ve kterém se vyskytuje příkaz `yield`. Může to být tělo metody, operátoru nebo akcesoru (neboli přistupovače, anglicky accessor). Pro příklad uveďme doplnění iterátoru do výše uvedeného zásobníku:

```
class Stack<T> : IEnumerable<T> {
  public IEnumerator<T> GetEnumerator() {
    for(int i=0; i<pos; i++) {
      yield return data[i];
    }
  }
}
```

Do deklarace zásobníku jsme přidali informaci, že budeme implementovat generické rozhraní `IEnumerable<T>`, což je generická varianta klasického rozhraní `IEnumerable`. Rovněž deklarace metody `GetEnumerator` se od klasické implementace enumerátorů liší jen použitím generické verze rozhraní `IEnumerator<T>`. Tělo této metody je však iterátorem. Procházíme zde naše vnitřní pole, kterým implementujeme zásobník, a příkazem `yield` postupně jakoby „vracíme“ jednotlivé prvky uložené na zásobníku.

Překladač tuto konstrukci přeloží do podoby přepínání kontextů. Ukažme si tedy způsob provádění kódu, který používá náš iterátor.

```
foreach(string v in s) {
  Console.WriteLine(v);
}
```

Vlastní vykonávání kódu probíhá takto:

1. Na začátku je zavolána metoda `s.GetEnumerator()`.
2. V místě příkazu `yield` je provádění kódu pozastaveno, je uchován kontext a hodnota `data[i]` je dosazena do proměnné `v`.
3. Je provedeno tělo příkazu `foreach` (je vypsána hodnota `v`).
4. Kontext je přepnut zpět do metody `GetEnumerator` a vykonávání pokračuje za příkazem `yield` opět až po další příkaz `yield` (můžeme tedy také umístit několik `yield` příkazů na různá místa v metodě).
5. Body 2 až 4 se opakují, dokud neskončí provádění metody `GetEnumerator`. Tím je ukončeno i provádění `foreach`.

Použití iterátoru ve formě (pojmenované) metody může mít následující podobu: Vytvoříme iterátor vracející pro dané N mocniny dvojky od 2^1 až po 2^N.

```
IEnumerable Power(int N) {
  int counter = 0;
  int result = 1;
  while(counter++ < N) {
    result *= 2;
    yield return result;
  }
}
```

Iterátor použijeme takto (vypíšeme prvních 10 mocnin dvojky, kód umístíme do jiné metody téže třídy):

```
foreach(int i in Power(10)) {
  Console.WriteLine(i);
}
```

Tímto způsobem tedy můžeme iterátory použít i privátně (bez samostatné třídy) nebo naopak umístit více různých typů iterátorů do jedné třídy (základní iterátor se bude jmenovat `GetEnumerator`, další si pojmenujeme dle vlastního uvážení).

Zbývá dodat, že příkazem `yield break;` ukončíme vykonávání kódu iterátoru (nelze použít `return`, neboť metoda s iterátorem je deklarována jako vracející `IEnumerable`).

3.2 Částečně deklarované třídy

Jazyk C++ vždy umožňoval rozdělit definici třídy do několika souborů. To je vhodné zejména pro zpřehlednění kódu rozsáhlejších tříd nebo pro oddělení kódu ručně psaného od částí generovaných generátorem kódu (jako je Visual Studio apod.). Používání tohoto prvku je velmi jednoduché, pomocí klíčového slova `partial`.

```
partial class MyClass {...}
```

Takto můžeme sloučit z libovolného počtu souborů (nebo z opakovaných deklarací v témže souboru) deklarace tříd, rozhraní nebo struktur, včetně jejich vnitřních součástí, atributů i XML komentářů. Můžeme dokonce uvádět neúplné seznamy implementovaných rozhraní, viz následující příklad.

```
partial class MyClass : BaseClass, Interface1 {
  partial class Inner<T> {
    T a;
  }
}

partial class MyClass : Interface2 {
  partial class Inner<T> {
    T b;
  }
}
```

Používání částečných deklarací má jen několik málo omezení:

- Je-li něco deklarováno jako `partial`, musí to tak být deklarováno pokaždé.
- Všechny části deklarace musejí být umístěny v jednom modulu (potažmo seskupení).
- Jména částečných tříd i generických typů musí být ve všech částečných deklaracích stejná.
- Přístupová práva a další modifikátory musí souhlasit u všech částí deklarace.

Zbývá dodat, že Visual Studio 2005 opravdu masívně používá tento nový prvek jazyka C#, takže při vytváření „Windows.Forms“ okenních aplikací je námi psaný kód striktně oddělen od automaticky generovaného kódu zajišťujícího chod vizuálních částí programu. To je velmi příjemná změna.

3.3 Nulovatelné typy

Jak známo, hodnotové typy nepoužívají reference, nelze jim proto přiřadit `null`. Většinou to nemá žádný negativní vliv, někdy však může být taková nepřiřazená reference využita k jiným účelům. Například při práci s databázemi můžeme chtít nějak označovat „neuvedené“ hodnoty. Chceme-li mít takovou neuvedenou hodnotu například u typu `int`, pak nezbývá, než jí rezervovat některou z nepoužitých hodnot. To může být například 0 nebo -1, ale používání podobných konstrukcí komplikuje kód a často vede k celé řadě (lidských) chyb.

Nyní máme k dispozici tzv. nulovatelné typy (nullable types). Jsou to hodnotové typy, kterým také můžeme přiřadit hodnotu `null`. Nulovatelnou proměnnou deklarujeme uvedením příslušného (libovolného) hodnotového typu a přidáme otazník. Také základní použití nulovatelných proměnných je poměrně snadné.

```
int? x;

Console.WriteLine(x==null ? "null" : x.Value.ToString());

if(x.HasValue) Console.WriteLine("x má hodnotu");
```

Nulovatelné typy mají tyto vlastnosti:

- Syntaxe `T?` pouze zastupuje delší zápis `System.Nullable<T>`. Tyto dvě konstrukce jsou tedy ekvivalentní.
- Vlastnost `HasValue` slouží ke zjištění, zda má daná proměnná hodnotu (tehdy vrací `true`), nebo je `null` (tehdy vrací `false`).
- Vlastnost `Value` zpřístupňuje hodnotu (typu `T`).

Převod na bázový datový typ (T) je možný explicitním přetypováním, převod z bázového typu je dokonce implicitní.

```
int a;
int? x;

x = a;
a = (int)x; //vyvolá výjimku, když x bude null
```

Nulovatelné typy definují všechny operátory stejně jako bázový typ. Výsledkem operací, do kterých vstupuje null hodnota, je vždy null hodnota. Uvedeme jen několik příkladů.

```
double? a,b;
long? c;

a = b * 2.0;
c++;
if(a > c) {}
```

Pro pohodlnou práci můžeme použít také operátor `??` (dva otazníky), který umožňuje definovat výchozí hodnotu při přetypování na bázový typ.

```
int? x;

int y = x ?? -1;
```

V této ukázce do proměnné `y` přiřazujeme hodnotu proměnné `x`. Pokud by `x` bylo `null`, přiřadíme do `y` hodnotu -1. Operátor `??` můžeme použít i pro přiřazování mezi nulovatelnými proměnnými.

```
int? z = x ?? -1;
```

Přestože `z` je nulovatelná proměnná, tímto příkazem do ní vložíme vždy nenulovou (not null) hodnotu. (Operátor `??` totiž převede hodnotu na bázový typ `int`, při následném přiřazení do `int?` je pak použit implicitní konverzní operátor.)

Specifické postavení mezi nulovatelnými typy má `bool?`. Od ostatních typů se liší tím, že jeho operátory logického součtu `|` a logického součinu `&` dávají v některých případech nenulové výsledky, i když jeden z operandů je `null`. Toto chování přesně odpovídá klasické (matematické) trojstavové logice i chování trojstavové logiky v jazyku SQL, proto se u něj nebudeme dále podrobněji zastavovat.

3.4 Bezejmenné metody, vnější proměnné

Bezejmenné (anonymní) metody představují především přiblížení jazyka C# k funkcionálním jazykům. Nyní můžeme vzít „kus kódu“ a předat ho jako parametr. Tím vznikne metoda, která nemá jméno, ale můžeme ji používat díky referenci, kterou na ni máme. Tato konstrukce již existuje v jazyku Java, takže zřejmě nikoho nepřekvapí.

V C# se k odkazování na kód používají metody (ty deklarují onen kód) a delegáty (ty na tento kód odkazují). Nyní tedy můžeme používat anonymní metody všude tam, kde je očekáván delegát. Podobně jako iterátory zjednodušují jinak komplikovanou práci s enumerátory, bezejmenné metody mohou často zjednodušit práci s delegáty.

```
button.Click += delegate {MessageBox.Show("Click!");};
```

Tímto jednoduše přidáme kód uvedený ve složených závorkách (zobrazení zprávy „Click!“) na událost stisku tlačítka. Ukázka rovněž předvádí základní místo využití bezejmenných metod: Hodí se především tam, kde potřebujeme jen krátký kód (např. jediný příkaz, jako zde) a psaní a klasických metod a delegátů by bylo zbytečnou prací navíc.

Kód bezejmenné metody je vždy samostatným blokem odděleným od bloku, kde je bezejmenná metoda deklarována. Můžeme zde však používat lokální proměnné nadřazeného bloku (netýká se `ref` a `out` proměnných). Tyto proměnné nazýváme vnější (outer) a jejich platnost je narozdíl od běžných lokálních proměnných rozšířena na dobu platnosti bezejmenné metody (což je pochopitelné). Vnější proměnné jsou tedy plně sdílené mezi anonymní metodou a vnějším blokem, což nám umožňuje vytvářet poměrně krkolomné programové konstrukce.

```
int a = 0;
button.Click += delegate {MessageBox.Show((++a).ToString());};
...
```

Tento kód zobrazí při každém stisknutí tlačítka o jedna větší číslo než minule. Na místě tři teček můžeme proměnnou `a` používat také. Všechna místa použití této proměnné přitom pracují s jedinou fyzickou instancí této hodnotové proměnné.

3.5 Operátor ::

Pro minimalizaci nebezpečí záměny jmen je doporučováno používat plně kvalifikovaná jména (např. System.Console.WriteLine místo Console.WriteLine). Pokud se však v kódu vyskytuje například následující deklarace, použití plně kvalifikovaného jména nestačí.

```
int System,Console;
```

Tato deklarace efektivně znemožní vypisování čehokoliv na obrazovku. Nyní však můžete použít operátor `::` (dvě dvojtečky) pro přístup ke globálním jménům.

```
System.Console.WriteLine("Nefunguje!"); //toto fungovat nebude

global::System.Console.WriteLine("Funguje!"); //toto funguje
```

Nutno upozornit, že funkce operátoru `::` je zde odlišná od jazyka C++. Je to binární operátor, na levé straně uvádíme obvykle slovo `global` reprezentující globální prostor jmen, můžeme také uvést zástupné jméno prostoru jmen. V druhém případě má operátor `::` podobný význam jako operátor `.` (tečka), avšak omezuje hodnotu na levé straně pouze na zástupná jména (namespace aliases).

3.6 Statické třídy

Třídy, které obsahují pouze statické součásti, nazýváme statické. Vytvářet instance takové třídy nemá smysl. Nově můžeme třídy explicitně označit modifikátorem `static` jako statické, překladač pak vytváření instancí ani nedovolí.

Statické třídy mají všechny součásti statické, nemají konstruktor (statický mít mohou), nelze z nich vytvářet instance jsou zapečetěné (sealed).

V předchozích verzích jazyka C# bylo možno dosáhnout podobného chování také, použití modifikátoru `static` je však jednodušší a bezpečnější (jeho používání eliminuje lidské chyby).

3.7 Zástupná jména seskupení (assembly aliases)

Klíčové slovo `extern` doposud označovalo metody, které jsou v neřízeném kódu (obvykle používáno ve spojení s `DllImport`).

Nyní je možno použít `extern` také k rozlišení seskupení, která obsahují stejné prvky (stejně pojmenované prostory jmen i třídy, které jsou tím pádem navzájem nerozlišitelné).

Máme-li například dvě verze stejné komponenty, které chceme používat současně, použijeme právě `extern`. Místo použití příkazu `using` pro klasické reference, napíšeme toto:

```
extern alias Komponenta_verze1;
extern alias Komponenta_verze2;
```

Přiřazení těchto zástupných jmen konkrétním souborům se seskupeními provedeme pomocí parametrů příkazové řádky překladače:

```
/r:Komponenta_verze1=grid.dll
/r:Komponenta_verze2=grid20.dll
```

3.8 Odlišná přístupová práva pro `get` a `set`

Vlastnosti (properties) dosud musely mít stejnou úroveň přístupových práv pro akcesor `get` i `set`. Když jsme tedy chtěli, aby byla vlastnost z vnějšku jen pro čtení, nesměla mít operaci `set` definovanou vůbec. Nyní je možno přiřadit každému akcesoru jinou úroveň oprávnění.

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

V ukázkovém kódu deklarujeme vlastnost jako veřejnou (`public`), potom však uvedením modifikátoru `private` zamezujeme veřejný přístup k akcesoru `set`.

Při deklaraci odlišných přístupových práv pro `get`/`set` musíme dodržet některá omezující pravidla. Vnitřní změna práv může být uvedena jen u jedné z operací `get`/`set` a musí být vždy více restriktivní (v praxi tedy bude většinou využita pro `set`). Při předefinování vlastnosti předka (pomocí `override`) musíme vlastnost deklarovat vždy s identickými oprávněními.

Narozdíl od C++, implementace rozhraní se v C# nepovažuje za dědění. Pokud je vlastnost součástí rozhraní, které implementujeme, můžeme u akcesoru, který není součástí implementovaného rozhraní, modifikátor oprávnění použít. Viz následující příklad.

```
public interface MyInterface {
  int MyProperty {
    get;
  }
}

public class MyClass : MyInterface {
  public int MyProperty {
    get {...}
    protected set {...}
  }
}
```

V ukázce: Akcesor `set` nepatří do rozhraní `MyInterface`, ve třídě `MyClass` tedy může být deklarován (i na vlastnosti `MyProperty`, která sama o sobě je součástí rozhraní `MyInterface`) s omezením přístupu na `private`.

3.9 Kovariance a kontravariance

Již zmíněná typová inference u generických typů není jediným novým prvkem ulehčujícím nám od časté práce s explicitním přetypováním. Zvlášť u delegátů není změna typu parametrů snadná, právě proto do jazyka C# přibyla kovariance a kontravariance[3].

Kovariance umožňuje použití metody se zděděným návratovým typem jako delegátu. Kontravariance umožňuje použití metody se zděděnými parametry jako delegátu. Obojí tedy stojí na principu zastupitelnosti předka potomkem ve stromu dědičnosti. Pro větší srozumitelnost si uveďme příklad.

```
class A {}

class B : A {}

class C : B {}

class Test {

  delegate B Metoda(B param);

  B MetodaA(A param) {
    return null;
  }

  B MetodaB(B param) {
    return null;
  }

  C MetodaC(B param) {
    return null;
  }

  public Test() {
    Metoda ma = new Metoda(MetodaA);
    Metoda mb = new Metoda(MetodaB);
    Metoda mc = new Metoda(MetodaC);
  }
}
```

[3] S těmito neobvyklými výrazy jsme se doposud setkávali většinou jen v geometrii, nyní je budeme muset akceptovat i do terminologie programování.

Nejprve deklarujeme strom dědičnosti C–▷B–▷A. Delegát deklarujeme se vstupním i návratovým typem `B`. Dále deklarujeme tři metody, které ukazují všechny možnosti zastoupení.

1. `MetodaB` pasuje přímo, protože má odpovídající parametry.
2. (Kovariance) `MetodaA` má vstupní parametr typu `A`, což je předek od `B`. Delegát ma má vstupní parametr `B`. V okamžiku zavolání tohoto delegátu je předán parametr typu `B`. Metoda v delegátu očekává typ `A`. Předaný parametr typu `B` je tedy převeden na předka `A`.
3. (Kontravariance) `MetodaC` má návratovou hodnotu typu `C`, což je potomek od `B`. Delegát vrací hodnotu typu `B`. Při použití delegátu `mc` je zavolána `MetodaC`, která vrací objekt typu `C`. Ten může být přetypován na typ `B`, což je potom výsledná návratová hodnota delegátu `mc`.

3.10 Pole pevné délky

V neřízeném (unmanaged, unsafe) kódu můžete nyní deklarovat a používat pole pevné délky, která jsou fyzicky uložena na místě deklarace (hodnotový typ, bez reference) a neprovádí se pro ně kontrola přetečení bufferu apod.

```
unsafe public struct MyArray {
  public fixed char path[256];
  private int reserved;
}
```

Zde jsme vytvořili strukturu `MyArray` obsahující mj. proměnnou `path`, což je pole o 256 znacích. Toto skutečně fyzicky odpovídá poli např. v C++ a představuje tedy prvek, který dříve v C# nebylo možné nijak deklarovat. (Doposud bylo nutné pro pole používat reference.) Tato konstrukce platí jen v neřízeném kódu a je vhodná pro práci v COM apod.

3.11 Spřátelená seskupení

Chceme-li, aby seskupení (assembly) `B` mělo přístup do všech veřejných (public) i neveřejných součástí seskupení `A`, pak v seskupení `A` použijeme následující atribut:

```
[assembly:InternalsVisibleTo ("B", PublicKeyToken="...")]
```

Je to tedy obdoba `friend` známého již z C++, zde je však vyžadováno uvedení hodnoty public key token[4] seskupení, které chceme deklarovat jako spřátelené.

[4] Public key token je osmibajtová hodnota sloužící k rozlišení dvou stejně pojmenovaných seskupení různých autorů.

3.12 Nevypisování varovných hlášení

Vypisování varovných hlášení (warnings) můžete nyní částečně ovlivnit přímo v kódu, použitím nové direktivy (známé i z Visual C++):

```
#pragma warning disable 414,3021

#pragma warning restore 414,3021
```

První řádek v ukázce vypne vypisování hlášení číslo 414 a 3021, druhý řádek pak vypisování hlášení opět obnoví. Zajímavé je, že pokud vypnete vypisování varovných hlášení přímo v nastavení projektu, v kódu už jej nemůžete znovu zapnout (možnosti jsou jen `disable` = zakázat a `restore` = obnovit původní nastavení, možnost enable = povolit zde chybí).
Poznámka: Vypínání varovných hlášení samozřejmě není doporučeno.

4 Analýza, shrnutí

Představili jsme si tedy 13 nových prvků jazyka C# 2.0 (2005). Některé z nich již při jejich oznámení vyvolaly bouřlivé diskuze na internetu. Žádná z novinek naštěstí neovlivňuje fungování dosavadního kódu, takže kdo je používat nechce, tak většinou nemusí.

4.1 Generické typy

Nejvýznamnější novinkou je uvedení generických typů. Ty se také významně podepsaly na nové verzi knihovny základních tříd (BCL), samozřejmě především v oblasti kolekcí. Používání generických tříd samozřejmě ocení především programátoři zvyklí na C++. Pro ostatní to bude znamenat především zkrácení kódu (vyšší úroveň znovupoužití) a také silnější typovou kontrolu. To může být na jednu stranu prospěšné, na druhou stranu to bude řadu programátorů zvyklých na nepříliš čisté objektově orientované programování často vyloženě otravovat. Podobný pocit „otravných“ omezení mohou pociťovat i programátoři C++ zvyklí na neobvyklé konstrukce u šablon, které v C# povoleny nejsou. C++ například umožňuje používat šablony pro plný polymorfizmus (bez deklarace společného rozhraní, tedy de facto na úrovni textových maker), což v jazyku C# možné není.

Shrnutí: Osobně tento prvek považuji za jednoznačně pozitivní.

4.2 Nulovatelné typy

Nad nulovatelnými typy se strhla bouřlivá diskuze na internetu. Na jedné straně totiž zjednodušují kód, zejména práci s relačními databázemi, na druhé straně však přinášejí několik sporných bodů:

1. Deklarace ve tvaru `int?` mnoha lidem vadí. Především kvůli tomu, že jde jen o syntaktický cukr, se programátoři rozdělili na dva proti sobě stojící tábory. Zástupci prvního tábora milují C a odvozené jazyky pro jejich stručný zápis mnoha konstrukcí (viz operátor ?:), druhý tábor naopak považuje toto zbytečné zkracování deklarací za prvek znepřehledňující kód.
2. Nulovatelné typy jsou jen předkrokem k zavedení plné podpory proměnných ve formě regulárních výrazů. Výraz `int?` totiž můžeme chápat jako „žádný nebo jeden `int`“. Potom `int!` by odpovídalo `int` a znamenalo by „právě jeden `int`“. Dále `int+` by znamenalo „jeden nebo více `int`ů“ a `int*` by znamenalo „libovolný počet `int`ů“. Tyto konstrukce umožní pohodlnější práci s daty. Na druhé straně také přinášejí zesložitění jazyka C# a je možné, že řada programátorů je ani nepochopí. Navíc pak zejména konstrukce `int*` je pro svou zaměnitelnost se zcela odlišným prvkem jazyka C++ mnoha lidem vyloženě trnem v oku.
3. Zatímco zde byl vyřešen problém nulovatelných typů, ticho zůstalo v otázce „nenulovatelných“ typů, jak je známe v C++. Jazyk C# totiž dodnes neobsahuje konstrukci, jak zapsat kód ekvivalentní referencím jazyka C++, tedy referencím, které nemohou obsahovat `null`. Jak ukazují příkladové studie, přidáním tohoto prvku by se podstatným způsobem zvýšila bezpečnost i jednoduchost kódu, neboť na většině míst opravdu `null` hodnoty (místo skutečných objektů) vůbec nechceme.

4.3 Operátor `::`

Operátor `::` je novinkou, která rozšiřuje jazyk C# o prvek, který dřívějšími prostředky nebylo možné nijak nahradit. Ačkoliv jeho použití bude zřejmě velmi okrajové, jeho přítomnost je jistě potěšující.

4.4 Prvky zjednodušující práci

Některé nové prvky umožňují novým způsobem dělat věci, které již dříve byly možné; nyní je to však podstatně jednodušší. Do této skupiny patří

- Iterátory
- Bezejmenné metody
- Odlišná přístupová práva pro `get`/`set`
- Kovariance a kontravariance

4.5 Částečně deklarované typy

Částečně deklarované typy slouží ke zpřehlednění kódu. Využití najdou především v generátorech kódu (jako je Visual Studio), které se nyní již nemusejí starat o regiony a mohou kód psaný uživatelem umísťovat do samostatných souborů.

4.6 Ostatní novinky

Ostatní novinky můžeme považovat za méně důležité, kosmetické úpravy. Nelze říci, že by snad nebyly přínosem, ale jejich dopad či využití je v porovnání s těmi již zmíněnými mnohem menší.

5 Novinky ve vývojovém prostředí a knihovně tříd BCL

Kromě samotného jazyka je možno spatřit řadu velmi zajímavých novinek také ve vývojovém prostředí Visual C# (Visual Studio) 2005, stejně jako v knihovně základních tříd .NET Frameworku (BCL). Některé z nových součástí BCL jsou právě důsledkem nových možností jazyka (generické kolekce apod.), jiné pak čistě rozšiřují schopnosti vestavěných tříd pro snazší vytváření koncových aplikací (řada novinek v ASP.NET apod.). Uveďme alespoň některé z nich.

Generické kolekce: Generické typy s sebou také přinesly nové generické kolekce, najdeme je v prostoru jmen `System.Collections.Generic`. Je to částečně úprava původní sady kolekcí `System.Collections`, částečně jde však o zcela nové prvky. Původní kolekce jsou samozřejmě nadále k dispozici, ale jejich další používání se nedoporučuje (neboť nejsou silně typované).

Refactoring: Nyní můžeme velmi snadno provádět některé operace refactoringu jako je změna jmen, schování proměnné za vlastnost (property), rozdělení metody na dvě, vytvořit rozhraní podle třídy, odstranit parametry metody nebo změnit pořadí parametrů metody. Všechny tyto operace fungují tak, že stávající kód bude po jejich provedení i nadále použitelný, tj. například při změně pořadí parametrů metody se provede tato úprava nejen v deklaraci, ale i ve všech voláních této metody.

Expanze: Pro zrychlení psaní často se opakujících částí kódu můžeme použít funkci expanze podle vzoru kódu. Tato funkce se aktivuje velmi jednoduše: napsáním definované zkratky s následným stiskem klávesy Tab.

Debugger: Novinek se dočkal i debugger (součást pro ladění kódu). Nyní nabízí rozšířenou (tj. chytřejší) nápovědu vždy vzhledem k aktuálnímu kontextu, což je zvláště poznat u nově přidaných popisů všech systémových výjimek. V debuggeru je také řada dalších novinek usnadňujících ladění.

IntelliSense: Také funkcionalita IntelliSense byla rozšířena. Seznam nabízených slov pro dokončení se snaží chovat inteligentně a nabízet přednostně ta slova, se kterými často nebo obvykle pracujeme; seznamy jsou rovněž více kontextově závislé. Nově přibyla možnost automatického generování kódu dle kontextu, což velmi usnadňuje psaní delegátů pro obsluhu událostí.

Reference

1. MSDN – centrum nápovědy a dokumentace
   `http://msdn.microsoft.com/`
2. MSDN Lab. – laboratoř pro beta verze produktů apod.
   `http://lab.msdn.microsoft.com/`
3. Visual Studio 2005 Beta Documentation
   `http://msdn2.microsoft.com/`

Annotation:

The New Features of Programming Language C# 2.0 (2005)

The paper presents all the new features of C# language version 2.0. Presented are 13 topics, all including source code examples and also author's analysis.

Citační záznam: Keprt A. Nové prvky jazyka Visual C# 2.0 (2005). In proceedings of *Objekty 2004*, Praha. Ed. David Ježek, Vojtěch Merunka, VŠB Technical University, Ostrava, 2004, pp. 15-32, ISBN 80-248-0672-X.