LEKCE 2: DATOVÉ TYPY A ŽIVOTNÍ CYKLUS

Verze 1.1 – poslední změna 28.2.2012 22:22

SPOUSTA ZKRATEK

Při práci s .NET Frameworkem se nutně setkáte se spoustou zkratek a proto je tu malý slovník:

CLI *Common Language Infrastructure*
Otevřená a standardizovaná (ECMA/ISO) specifikace popisující běhové prostředí pro aplikace a jejich kódu. Microsoft .NET Framework je implementací CLI.

CLR *Common Language Runtime*
Běhové prostředí – součást .NET Frameworku, která se stará o spouštění programů, správu paměti, typovou bezpečnost nebo o výjimky.

CTS *Common Type System*
Část CLI specifikace popisující typy a typový systém.

BCL *Base Class Library*
Knihovna dodávaná s .NET Frameworkem. Obsahuje třídy pro práci se soubory, grafikou, databází nebo sítí. Typy z BCL se obvykle poznají podle umístění v jmenném prostoru `System` nebo jeho podprostorech.

CIL *Common Intermediate Language*
Nízkoúrovňový programovací jazyk, objektově orientovaný jazyk symbolických instrukcí. CIL je výsledkem kompilace programu na platformě .NET. Při spuštění programu pak dojde k JIT kompilaci CIL do instrukcí konkrétního procesoru.

JIT *Just-In-Time (kompilace)*
Při JIT kompilaci dochází k překladu CIL na instrukce procesoru před spuštěním programu nebo během jeho běhu. Lze tak provádět optimalizace pro konkrétní procesor nebo na základě statistik běhu programu.

Managed code (řízený kód)
Kód běžící jen ve virtuálním stroji .NET Frameworku.

TYPOVÝ SYSTÉM V .NET (CTS)

Common Types System (CTS) je část specifikace .NET Frameworku zahrnující datové typy a práci s nimi. Do toho patří i dědičnost, viditelnost a životní cyklus objektů. A také základní datové typy .NET Frameworku (build-in types):

Skupina	.NET Typ[1]	C# kličové slovo		přípona
celočíselné	`Byte`	`byte`	8 b, bez znaménka	
	`Int16`	`short`	16 b, se znaménkem	
	`Int32`	`int`	32 b, se znaménkem	
	`Int64`	`long`	64 b, se znaménkem	`L` nebo `l`
	`SByte`	`sbyte`	8 b, se znaménkem	
	`UInt16`	`ushort`	16 b, bez znaménka	
	`UInt32`	`uint`	32 b, bez znaménka	
	`UInt64`	`ulong`	64 b, bez znaménka	
plovoucí čárka	`Single`	`float`	32 b	`F` nebo `f`
	`Double`	`double`	64 b	`D` nebo `d`
logické	`Boolean`	`bool`	true | false	
ostatní	`Char`	`char`	16 b, Unicode znak	
	`Decimal`	`decimal`	128 b, pevná des. čárka - monetární typ	`M` nebo `m`
objekty	`Object`	`object`	kořen objektové hiearchie	
	`String`	`string`	řetězec Unicode znaků	

TŘÍDA SYSTEM.OBJECT

Ze třídy System.Object dědí všechny objekty v .NET Frameworku, včetně vestavěných typů jako `int` nebo `bool`. V této třídě jsou definovány metody `ToString` pro převod objektu na řetězec, `Equals` pro porovnání s jiným objektem a `GetHashCode` pro výpočet hashe objektu.

Pravidla pro implementaci vlastní verze metod Equals a GetHashCode najdete na MSDN.

REFERENČNÍ A HODNOTOVÉ TYPY

Typy .NET Frameworku se dělí na dvě základní skupiny: referenční a hodnotové typy. Rozdíl mezi nimi spočívá ve způsobu uložení dat v paměti.

Hodnotové typy mají svoji hodnotu uloženou přímo v proměnné (případně datové položce nebo parametru). Při přiřazení hodnotového typu dojde ke zkopírování obsahu proměnné (datové položky/parametru) a tedy i zkopírování celé hodnoty. Mezi hodnotové typy patří většina vestavěných typů (vše mimo Object a String), výčty a pro vytvoření vlastních hodnotových typů slouží struktury.

Objekty **referenčních typů** jsou vytvářeny v samostatné části paměti (tzv. haldě) a proměnné obsahují pouze odkaz na tento objekt. Pokud dojde k přiřazení proměnné, dojde ke zkopírování odkazu a nová proměnná tak bude obsahovat referenci na stejný objekt. Hlavním zástupcem referenčních typů jsou třídy, dále pak pole, rozhraní nebo delegáti (delegates).

[1] Všechny základní typy se nachází ve jmenném prostoru `System`.

Pokud máte problém s pochopením, zkuste si představit hodnotový typ jako textový soubor a referenční typ jako zástupce programu ve Windows. Zkopírováním textového souboru máte dva samostatné soubory, ale zkopírovaný zástupce pořád spouští stejný program nebo soubor.

TŘÍDY

Třídy v C# jsou zástupci referenčních datových typů, definují se pomocí klíčového slova `class`. Stejně jako v Javě a na rozdíl od C++ podporují jen jednoduchou dědičnost, mohou implementovat libovolný počet rozhraní. Pokud není uvedeno jinak dědí z třídy System.Object.

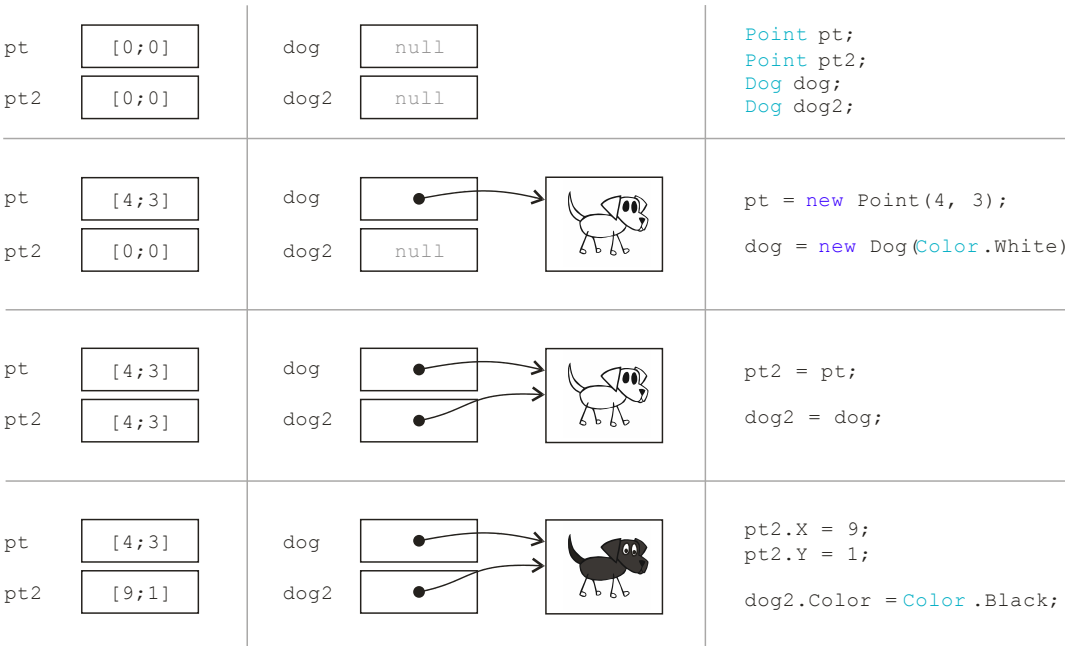
STRUKTURY

Struktury jsou hodnotové typy a deklarujete je klíčovým slovem `struct`. Jsou podobné třídám, ale mají několik omezení vyplývajících z podstaty hodnotových typů:

- Vždy mají výchozí konstruktor (bez parametrů) a nelze jej předefinovat.
- Libovolný jiný konstruktor musí vždy volat bezparametrický (pomocí `this()`).
- Nelze inicializovat členské proměnné struktury.
- Struktury nemůžou dědit, ale mohou implementovat rozhraní.

```
struct Point {
  public int X { get; set; }
  public int Y { get; set; }
}

class Dog {
  public Color Color { get; set; }
}
```



VÝČTOVÉ TYPY

Výčtové typy v .NET Frameworku jsou jen ohraničenou množinou celočíselných konstant. Každý výčet lze převést na jeho celočíselný podtyp a naopak (pokud není specifikováno jinak, je to `System.Int32`).

Deklarace se provádí klíčovým slovem `enum`, následovaným jménem typu a seznamem položek výčtu. Pokud položce není přiřazena hodnota explicitně, přiřadí ji kompilátor hodnotu předchozí položky + 1, první položce by pak přiřadil 0. Pokud vytvoříte proměnnou výčtového typu, je její hodnota inicializována na hodnotu 0 - a to i tehdy, pokud ve výčtu není položka pro 0.

```
enum Color {
  Red,          // ((int)Color.Red)    == 0
  Blue,         // ((int)Color.Blue)   == 1
  Black = 3,    // ((int)Color.Black)  == 3
  Yellow        // ((int)Color.Yellow) == 4
}
Console.WriteLine((int)Color.Yellow) // vypíše 4
```

NEEXKLUZIVNÍ VÝČTOVÉ TYPY – [FLAGS]

Běžné výčty mohou mít právě jednu hodnotu. Naopak neexkluzivní výčty se chovají jako pole bitů: jednotlivé položky lze dohromady kombinovat pomocí operací AND, OR, XOR (přesněji bitových operátorů &, |, ^). Nebo chcete-li jako odškrtávací seznam má-nemá.

Neexkluzivní výčet se deklaruje aplikováním atributu `FlagsAttribute` na výčet. Atribut lze zapsat jako `[FlagsAttribute]` nebo zkráceně jako `[Flags]` (díky jmenným konvencím lze vynechat koncovku Attribute a kompilátor i přesto uhodne celý název třídy atributu). Pro zjištění přítomnosti příznaku pak lze použít metodu `HasFlag`.

```
enum Permissions {
  Read    = 0x01,      // DEC(1) = BIN(001)
  Write   = 0x02,      // DEC(2) = BIN(010)
  Execute = 0x04,      // DEC(4) = BIN(100)
 }
Permissions readWrite = Permissions.Read | Permission.Write // DEC(3) == BIN(011)
bool canRead = readWrite.HasFlag(Permissions.Read);
```

TYPY S OTAZNÍKEM – NULLABLE<T>

Hodnotové datové typy v C# mají vždy nějakou hodnotu. Problém nastává, pokud je potřeba vyjádřit, že proměnná vlastně žádnou hodnotu nemá (například při komunikaci s relační databází může být NULL ve sloupci).
U referenčních typů stačí nastavit `null`, u hodnotých to ale udělat nejde. .NET Framework proto přichází s datovým typem `Nullable<T>`, což je generický typ rozšiřující typ `T` o hodnotu `null`. V jazyce C# lze takový typ zapsat pomocí otazníku za názvem typu, např `int?` nebo `DateTime?`.

Kromě vlastností jako `HasValue` a `Value` je typ `Nullable<T>` schopný konverze na typ `T` (explicitní) a naopak (implicitní).

```
int? age1 = 10;
int? age2 = null;
int age3 = (int)age1;
```

IMPLICITNĚ TYPOVANÉ PROMĚNNÉ

V metodách lze pro proměnné použít implicitní datový typ pomocí klíčového slova `var` místo typu. Proměnná typu `var` je stále silně typovaná proměnná, ale kompilátor si její typ určí sám. Typ se určuje z pravé strany inicializace proměnné – z toho co je za rovná se. Bez inicializace nejde implicitně typovanou proměnnou vytvořit.

```
var age = 10;           // 10 je Int32 => age je typu Int32
age = "Veverka";        // chyba při kompilaci: age je typu Int32
var name;               // chyba při kompilaci: nelze odhadnout typ
```

OBJEKTOVÁ INICIALIZACE A INICIALIZACE KOLEKCÍ

Při vytváření objektů můžete nastavit vlastnosti a datové položky v rámci jednoho příkazu, bez nutnosti mít specifický konstruktor. Tomuto se říká objektová inicializace.

```
Dog dog = new Dog { Name = "Alik", Age = 10 };
```

Podobně pak vypadá inicializace kolekcí, kdy se za volání konstruktoru kolekce vloží do složených závorek seznam objektů oddělených čárkou. Pro inicializací je potřeba, aby kolekce implementovala rozhraní IEnumerable a existovala v ním metoda Add, schopná do kolekce vložit patřičnou položku.

```
IList<Dog> dogs = new List<Dog>() { new Dog("Alik"), new Dog("Cvalik") };
```

ANONYMNÍ TYPY

Anonymní typy slouží pro zapouzdření proměnných bez nutnosti deklarovat samotný datový typ. Anonymní typy obsahují jednu nebo více vlastností, které se nastaví při vytvoření instance a dále jsou jen ke čtení. Protože nemají název, lze je deklarovat jen pomocí implicitně typované proměnné (`var`). Vlastnosti anonymního typu se pak určí z objektové inicializace.

```
var alik = new { Name = "Alik", Owner = "Bara" };   // vlastnosti Name a Owner
var zofinka = new { Name = "Zofinka", alik.Owner }; // druha vlastnost bude mit také nazev Owner
```

Anonymní typy mají překryté metody `Equals` a `GetHashCode` na rovnost všech vlastností. Metoda `ToString` je také přetížena a vypisuje seznam všech vlastností a jejich hodnost.

Platnost anonymního typu je omezena jen na metodu, ve které je vytvořen – nelze jej vracet jako návratovou hodnotu ani použít jako parametr metody.

ŽIVOTNÍ CYKLUS OBJEKTŮ

Životní cyklus objektů je přibližně následující.

1. Je deklarována proměnná pro daný typ objekt. Paměť v daném umístění se vynuluje.
2. V případě třídy se voláním konstruktoru třídy vytvoří její instance na haldě.
3. Objekt si jen tak existuje, případně je používán.
4. Provede se manuální úklid – např. dojde k zavření otevřeného souboru.
5. Ve chvíli, kdy se objekt stane nedosažitelným je vydán na milost garbage collectoru (GC) a ten jej odstraní z paměti.

.NET Framework disponuje automatickou správou paměti. Alokace a uvolňování paměti je úkolem garbage collectoru (GC). GC běží na pozadí programu a čas od času zastaví celý program a začne zjišťovat, které objekty jsou

nedostupné (neexistuje způsob jak se z kódu k objektu dostat). Na těchto objektech pak spustí finalizaci (z pohledu C# zavolá desktruktor - pokud existuje) a následně objekt odstraní z paměti.

KONSTRUKTORY

Konstruktory jsou zvláštní metody třídy nebo struktury, které se volají při jejím vytváření. Konstruktory nemají žádný návratový typ, jejich název je shodný s názvem třídy a mohou mít modifikátor viditelnosti (`public`, `private`, `protected`).

V rámci jedné třídy (nebo struktury) lze řetězit volání konstruktoru připojením klíčového slova `this` za hlavičku konstruktoru. Podobně lze volat i konstruktor bázové třídy, a to klíčovým slovem `base`.

```
class Dog : Animal
{
  public Dog(string name) : this(name, 0) { // kód konstruktoru }
  public Dog(string name, int age) : base(name, age) { // kód konstruktoru }
}
```

Pokud třída nemá žádný konstruktor, vytvoří C# výchozí bezparametrický konstruktor, který nastaví hodnoty členů na jejich výchozí hodnoty (`0`/`null`/`false`).

Struktury bezparametrický konstruktor mají vždy a nejde ho přepsat. Naopak všechny další konstruktory musí bezparametrický konstruktor zavolat.

DESTRUKTORY [PRO ZAJÍMAVOST]

Destruktor je metoda ve tvaru `~NázevTřídy()`, bez návratového typu a modifikátorů. Tuto metoda je volána jen automaticky ve chvíli, kdy se GC rozhodne třídu odstranit z paměti – proces se jinak nazývá finalizace (dokončení).

Toto tvrzení není úplně přesné, protože proces odstranění z paměti je složitější. Nicméně faktem je, že existence destruktoru zdržuje odstranění objektu z paměti a měla by se využívat jen pokud je opravdu potřeba. Volání desktruktoru navíc není deterministické, nevíte kdy přesně se GC rozhodne objekt odstranit.

```
class Dog : Animal
{
  ~Dog()
  {
    Console.WriteLine("Zazvonil zvonec a kde je psa konec…");
  }
}
```

ROZHRANÍ IDISPOSABLE [PROBEREME POZDĚJI]

Někdy je potřeba explicitního a deterministického úklidu po třídě. Například třída zapisující do souboru by soubor měla co nejdříve zavřít, aby jej mohl používat někdo jiný. Pro takový explicitní úklid je v .NET Frameworku určeno rozhraní `IDisposable`.

Typické použití třídy implementující rozhraní `IDisposable` společně s blokem `try-finally`, kde v části `finally` dojde k zavolání metody `Dispose`. Pro takové volání má C# syntaktickou zkratku v podobně bloku `using`. Tyto dva bloky kódu jsou pak ekvivalentní.

```
StreamWriter sw;
try
{
```

~

```
using(StreamWriter sw = new StreamWriter("data.txt"))
{
  sw.WriteLine("Ahoj");
```

```
  sw = new StreamWriter("data.txt");          }
  sw.WriteLine("Ahoj");
}
finally
{
  if(sw != null)
    sw.Dispose();
}
```