Programovací jazyk C#

Marek Běhálek

Programovací jazyk C#

Marek Běhálek

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Vážený studente,

následující dokument je určen jako doplňkový výukový materiál pro studium předmětu Programovací jazyk C#. Materiál byl poskládán z původních materiálů které jsem vytvořil spolu s Tomášem Turečkem a dalších výukových materiálů, které byly v minulých letech vytvořeny na VŠB-TU Ostrava v rámci různých bakalářských a diplomových prací. Při tvorbě jsem vycházel zejména z práce Jiřího Sovadiny a Milana Trávnička.

Materiál byl tvořen pro verzi jazyka 2.0. Novým vlastnostem jazyka, které přišly po verzi 1.0, je věnovaná větší pozornost. Zejména proto, že jde o konstrukce, které nejsou běžné v jiných jazycích. Určitě ale nejsou důležitější než jiné "základnější" vlastnosti jazyka.

Materiál nepředstavuje kompletní přehled vlastnostností jazyka C#. Některé vlastnosti jsou v tomto materiálu zmíněny pouze okrajově a některé vlastnosti jsou vypuštěny úplně. jde zejména o pomůcku pro studium v kurzu Programovací jazyk C#. Jiné využití bez vědomí autora není přípustné!

Marek Běhálek

Kapitola 1. .NET framework

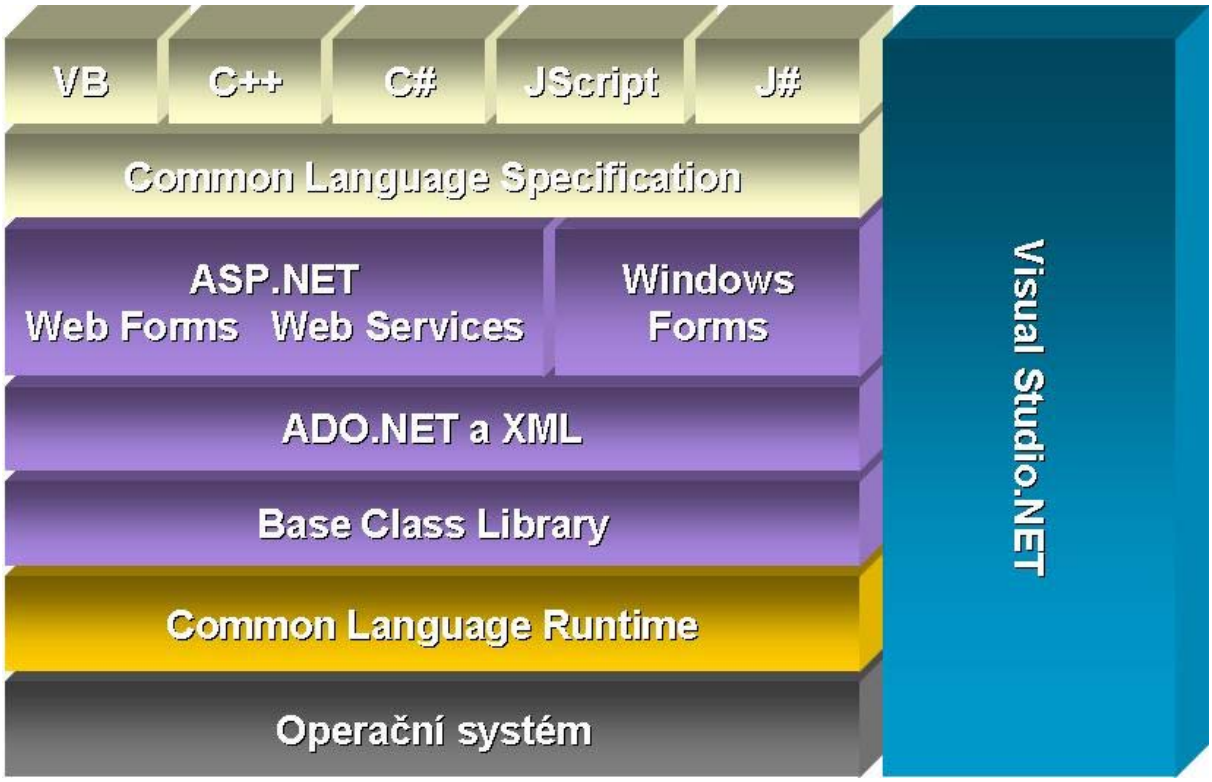
V dnešní době jsou vyvíjeny složité aplikace, které svým rozsahem převyšují schopnosti jednoho člověka. Na vývoji aplikací se většinou podílí více programátorů. Při své práci využívají (znovupoužívají) části (komponenty) svých aplikací nebo aplikace jiných firem. Programátor tedy již nemůže spoléhat jen sám na sebe, tak jak to bylo v dobách operačního systému MS-DOS. Prvním nástrojem pro modulární vývoj aplikací na platformě Microsoft Windows byly dynamické knihovny.

Počátkem 90. let byly většinou vytvářeny samostatné aplikace s velmi malou schopností vzájemné komunikace. Tento nedostatek byl odstraněn v polovině 90. let, kdy firma Microsoft uvedla technologii COM (Component Object Model). Obrovskou výhodou komponentové technologie je její jazyková neutralita v binární formě. Pro každou komponentu bylo definováno rozhraní, které zprostředkovává komunikaci mezi klientem a příslušnou komponentou. Časem se však ukázalo, že i tato technologie má svá omezení. V dnešní době je modulární architektura čím dál používanější. Využívané komponenty jsou většinou malé a jednoduché. Hlavní nevýhodou COM komponent je, že zakrývají svou vnitřní realizaci. Jediné co komponentu popisuje je příslušné rozhraní. Toto znemožňuje dědičnost na úrovni zdrojových kódů.

.NET Framework funguje doslova jako substrát, na kterém lze pěstovat software. Jeho jádro je založené na principech objektově orientovaného programování a všechny základní služby zpřístupňuje široké škále programovacím jazykům. .NET Framework automaticky podporuje třídy, metody, vlastnosti, konstruktory, události, polymorfismus atd. Ve výsledném efektu to znamená, že není podstatné, ve kterém programovacím jazyce komponenty vytváříme případně, jaké komponenty používáme. .NET Framework také řeší některé problémy související s bezpečností. Dalším problémem, který .NET Framework řeší, je nasazování a instalace aplikací (označovaný jako DLL Hell).

Architektura .NET Framework

Obrázek 1.1. Architektura .NET Framework



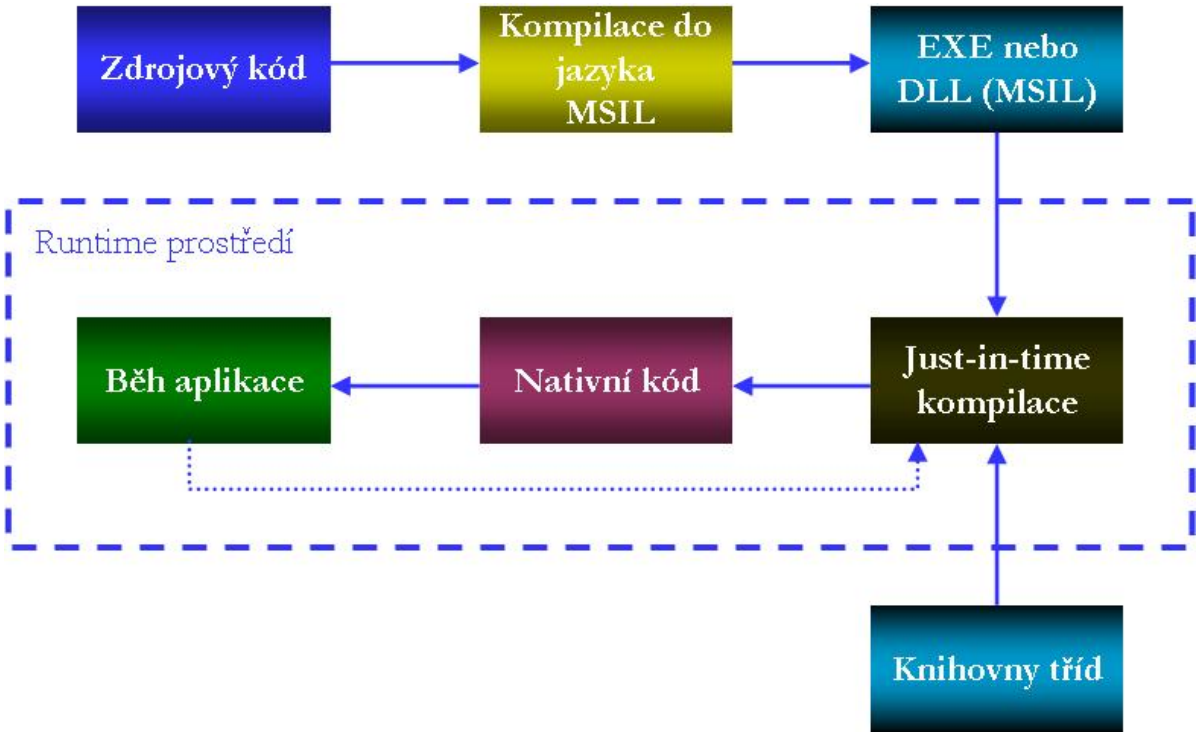
Na nejnižší úrovni se nachází *CLR- Common Language Runtime* realizující základní infrastrukturu, nad kterou je framework vybudován. Nad CLR se nachází několik hierarchicky umístěných knihoven. Ty jsou rozděleny do jmenných prostorů. Základem je knihovna nazvaná *Base Class Libary*. Nad ní je knihovna pro přístup k datům a

práci s XML soubory. Poslední vrstvou je sada knihoven usnadňující práci s uživatelským rozhraním. Je rozdělena do dvou skupin: *pro usnadnění vytváření webových aplikací* a *pro vytváření klasických aplikací*. Poslední vrstvu tvoří nelimitovaná množina programovacích jazyků. Jejich základní vlastnosti definuje *CLS – Common Language Specification*. V současné době jsou firmou Microsoft podporovány čtyři jazyky Visual Basic, C++, C# a Jscript. Tato množina ale není uzavřena a jakýkoliv výrobce ji může rozšířit. Celá tato architektura je podpořena novou verzí vývojového nástroje Visual Studio .NET/Visual Studio 2003.

CLR – Common Language Runtime

CLR si lze ztotožnit s pojmem virtuálního stroje při použití programovacího jazyka Java. Podobně jako u programovacího jazyka Java, nejsou zdrojové kódy kompilovány přímo do nativního kódu, který lze provádět, ale do *intertmediárního jazyka*. U programovacího jazyka Java je výsledkem soubor s příponou CLASS a tento je pak prováděn virtuálním strojem Javy. Podobně v prostředí .NET jsou zdrojové soubory libovolného programovacího jazyka zkompilovány do intermediárního jazyka (nazvaného *MSIL – Microsoft Intermediate Language*). V případě, že má být taková aplikace spuštěna, systém detekuje, že jde o aplikaci v MSIL a spustí *Just-In-Time kompilátor*. Ten vygeneruje skutečné instrukce cílové platformy. Vše je přehledně zobrazeno na následujícím obrázku.

Obrázek 1.2. Common Language Runtime - kompilace a spuštění



Jedním z hlavních cílů při vývoji .NETu je podpora různých programovacích jazyků. Důležitým prvkem CLR je podpora *společného typového systému* (*Common Type Systém – CTS*). Vedle CTS definovaném na systémové úrovni, CLR realizuje typovou bezpečnost a obecný objektově orientovaný model.

CTS

CTS je nezávislý na jazykové implementaci. Objektově orientované jazyky a přístup k tvorbě softwaru jsou již léta součástí vývojových nástrojů. Implementace se v různých prostředích ale výrazně liší. Některé programovací jazyky jsou čistě objektově orientované (například Smalltalk), jiné implementují vedle objektově orientovaných principů také některé neobjektové vlastnosti. V čistě objektově orientovaných programovacích jazycích jsou všechny datové typy představovány třídami a všechny proměnné jsou ve skutečnosti objekty. Objektovost dotažena do úplného konce je značně výhodná, ale přináší nemalou daň v podobě poklesu výkonu. Proto přidáváme čistě objektovým jazykům tzv. základní typy a množinu operací, které zlepšují výkonnost daného programovacího jazyka. Typový systém je rozdělen do dvou hlavních kategorií: *hodnotové typy* a *referenční typy*. V .NETu je vše (podobně jako v Javě) objektem. Základem každého typu je třída nazvaná

System.Object. Výjimku tvoří hodnotové typy. Jak již bylo uvedeno, pro zvýšení výkonnosti jsou i do .NETu přidány některé základní typy jako různé druhy celých nebo reálných čísel. Někdy je ovšem nutné, pracovat i s těmito typy jako s objekty. .NET nám umožňuje provádět automatickou konverzi hodnotového typu na referenční. Operaci, která převod zajistí se říká *Boxing*. Opačnému procesu - převedení hodnotového typu na referenční - se říká *Unboxing*. Všechny hodnotové typy mají definovány odpovídající třídu, na kterou jsou konvertovány. Každý základní typ je reprezentován implicitně svou hodnotou, ale v případě potřeby je možné jej převést na odpovídající objekt a pracovat s ním jako s každým jiným objektově orientovaným typem. Díky tomu se daří dosáhnout požadované plné objektovosti typového systému bez zaplacení výkonnostního penále.

Typová bezpečnost

Jedním z nejdůležitějších požadavků kladený na společný typový systém je typová bezpečnost. Můžeme jí také chápat jako garanci, že nad definovanými typy nemohou být provedeny nepovolené operace. Typová bezpečnost odstraňuje chyby pramenící z nekontrolované manipulace s poměnými nebo pamětí. Každý objekt vytvořený v programu je striktně typový a typová je i reference, která se na něj odkazuje. Každý typ je navíc sám zodpovědný za přístupová práva ke svým členům.

Management paměti

Obecným cílem managementu paměti je uvolnit systémové zdroje držené objektem a následně i paměti, kterou zabírá ve chvíli, kdy objekt již nikdo nevyužívá. .NET Framework má technologii *Garbage Collection*. Ta přináší automatickou správu paměti.

MSIL

Spustíme-li aplikaci o jejíž provádění se stará CLR, hovoříme o *řízeném kódu*. Pokud je spuštěna aplikace která není napsaná pro prostředí .NET, nebo se výhod řízeného kódu explicitně zřekneme, je kód *neřízený*. Ne všechny jazyky umí generovat řízený kód. Příkladem jazyka s absolutní volností je jazyk C++.

Výsledkem kompilátoru jazyka schopného generovat řízený kód je *MSIL - MicroSoft Intermediate Language*. MSIL je procesorově nezávislý jazyk podobný assembleru. Oproti asembleru je však mnohem vyspělejší. Umí pracovat s objekty, volat virtuální metody, pracovat s prvky pole nebo zpracovávat výjimky. Důvodem pro zavedení tohoto jazyka je snaha o jednoduché přenášení existujícího kódu mezi různými platformami. V současné době neexistuje procesor, na kterém by šlo provádět instrukce MSIL, proto CLR před vlastním spuštěním kompiluje (pomocí JIT kompilátoru) MSIL instrukce do nativního kódu na dané, konkrétní platformě. Hlavní výhodou použití intermediárního jazyka je platformní nezávislost.

JIT Kompilátor

Obecně se používají se tři druhy JIT kompilátorů.

1. Provádějící překlad v době instalace – pak se již podstatě nejedná o JIT kompilátor. Výhodou je odstranění zpoždění, které samotný překlad způsobuje.

2. Skutečný JIT překladač – před samotným spuštěním je aplikace přeložena do nativního kódu dané platformy. Výsledek je srovnatelný s produktem konvenčního překladače. Nevýhodou je zpoždění při zavádění aplikace do operační paměti a zpomalení započatí jejího vykonávání.

3. Ekonomický JIT překladač – provádí partikulární překlad programu. Jsou překládány jen ty části, které jsou zrovna potřeba. Hlavní výhodou jsou menší paměťové nároky.

Assembly

Velký problém je nasazování aplikací. Řešením tohoto problému v prostředí .NET je Assembly. Důvodů pro zavedení Assembly je několik:

- Aplikace musí být samostatné – snadná instalace nebo reinstalace.

- Aplikace musí obsahovat čísla verzí a musí být na ně vázána

- Čísla verzí komponent – vedle vlastní verze si musí aplikace pamatovat čísla verzí komponent, které interně používá.
- Musí podporovat Side-by-side komponenty – existence dvou a více verzí stejné komponenty na jednom počítači je realitou a jedním z velkých problémů současných instalací.
- Musí umožňovat izolaci aplikace – aplikace nesmí být náchylná k poruchám způsobených změnami provedenými na počítači.
- Musí zajistit bezpečný přístup ke kódu.
- Komponenty musí obsahovat informace o *veřejných typech*.

Assembly je logickou kolekcí, stávající se z jednoho nebo více *.exe*, *.dll* nebo *.module* souboru a zdrojů doplněných o *manifest*. Assembly je také často definována jako programová jednotka určená k nasazení a opakovanému použití, řízení verzí a bezpečnosti. Assembly je velice podobná dnešní dynamicky linkované knihovně, jsou však na ní kladeny vyšší požadavky. Tyto vyšší požadavky jsou splněny přidáním tzv. manifestu do Assembly. *Manifest* je blok metadat obsahující tyto informace:

- identita – jméno, verze a kultura;
- seznam souborů + kryptografické zabezpečení;
- odkaz na další použité assembly + jejich verze;
- exportované (veřejně viditelné) typy a zdroje;
- bezpečnostní požadavky.

Existují dva typy assembly.

- *Soukromé assembly* – jsou implicitním a doporučovaným typem aplikací v .NET. Jejich instalace (respektive odinstalování) je vlastně pouhé kopírování (respektive mazání). Manifest obsahuje všechny potřebné údaje pro korektní nasazení aplikace. Soukromé assembly si můžeme představit jako interní DLL knihovny pro aplikace vytvářené jednou firmou. Jde o programové jednotky, u nichž se neuvažuje s obecným nasazením.
- *Sdílené assembly* – představují komponenty vytvářené pro obecné použití více aplikacemi. Jako příklad může sloužit knihovna *Windows Forms*. Nejčastěji jsou instalovány do *GAC – Global Assembly Cache*. Sdílené assembly jsou strukturálně identické se soukromými. Liší se ve způsobu pojmenování (definují jednoznačné sdílené jméno) a ve způsobu řízení verzí. Složitější je také instalace a odinstalování aplikace.

Bezpečnost .NET Framework

Pojem bezpečnosti se dotýká několika oblastí:

- typová bezpečnost - typová bezpečnost v tomto kontextu se týká pouze bezpečného přístupu k paměti objektů. Během JIT kompilace probíhá verifikační proces zkoumající obsah metadat v manifestu a obsah MSIL kódu a ověřující zda-li je kód typově bezpečný;
- identita kódu - na základě informací o kódu definuje práva přidělená aplikacím;
- code access security (přístupová bezpečnost kódu) - tento typ bezpečnosti umožňuje nastavit důvěryhodnost kódu na požadovanou úroveň v závislosti na tom, odkud kód pochází a dalších aspektech daných identitou kódu;
- povolení (permissions) - CLR umožňuje aplikacím vykonávat jen ty operace, pro která má jejich kód povolení. Běhové prostředí používá speciální objekty zvané *permissons* pro implementaci omezení, která jsou kladena na řízený kód;
- bezpečnost založená na rolích - využívá identity asociované s daným exekučním kontextem;

- kryptografické služby - .NET Framework obsahuje celou sadu kryptografických objektů, které implementují známé algoritmy pro hashování, kryptování a digitální podpisy.

Kapitola 2. Základní charakteristika jazyka C#

Jazyk C# vyvinula firma Microsoft. Byl představen spolu s celým vývojovým prostředím .NET. Jak název napovídá, vychází tento jazyk v mnohém z programovacího jazyka C/C++, ale v mnoha ohledech je daleko bližší programovacímu jazyku Java. Základní charakteristiky jazyka jsou:

- Jazyk C# je čistě objektově orientovaný.
- Obsahuje nativní podporu komponentového programování.
- Podobně jako Java obsahuje pouze jednoduchou dědičnost s možností násobné implementace rozhraní.
- Vedle členských dat a metod přidává vlastnosti a události.
- Správa paměti je automatická. O korektní uvolňování zdrojů aplikace se stará *garbage collector*.
- Podporuje zpracování chyb pomocí vyjímek.
- Zajišťuje typovou bezpečnost a podporuje řízení verzí
- Podporuje atributové programování.
- Zajišťuje zpětnou kompatibilitu se stávajícím kódem jak na binární tak na zdrojové úrovni.

Většina uvedených vlastností vychází přímo s funkcionality vývojového rámce .NET. Jazyk C# je také integrován do vývojového prostředí Visual Studio.NET

Překladače jazyka C# jsou *case sensitive*. Rozlišují tedy velká a malá písmena. Podobně jako v jiných programovacích jazycích, i v jazyce C# bylo zavedeno několik konvencí. Jména balíků, tříd, rozhraní a většiny dalších položek začínají velkým písmenem. Malým začínají privátní a chráněné (protected) atributy, lokální proměnné a parametry. Více informací o používaných konvencích najdete v dokumentaci (Naming Guidelines). Následující příklad ukazuje jednoduchou kostru programu, kterou vygeneruje Visual Studio, vytvoříte-li konzolovou aplikaci.

Ukázka kódu

```
using System;

namespace Namespace
{
  /// <summary>
  /// Summary description for Class1.
  /// </summary>
  class Class1
  {
    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [STAThread]
    static void Main(string[] args)
    {
      //
      // TODO: Add code to start application here
      //
    }
  }
}
```

Předchozí příklad také ukazuje různé typy komentářů v jazyce C#. Podobně jako v C/C++ lze používat jak víceřádkové komentáře uvozené /* */ tak jednořádkové komentáře po znacích //. Specální význam má značka

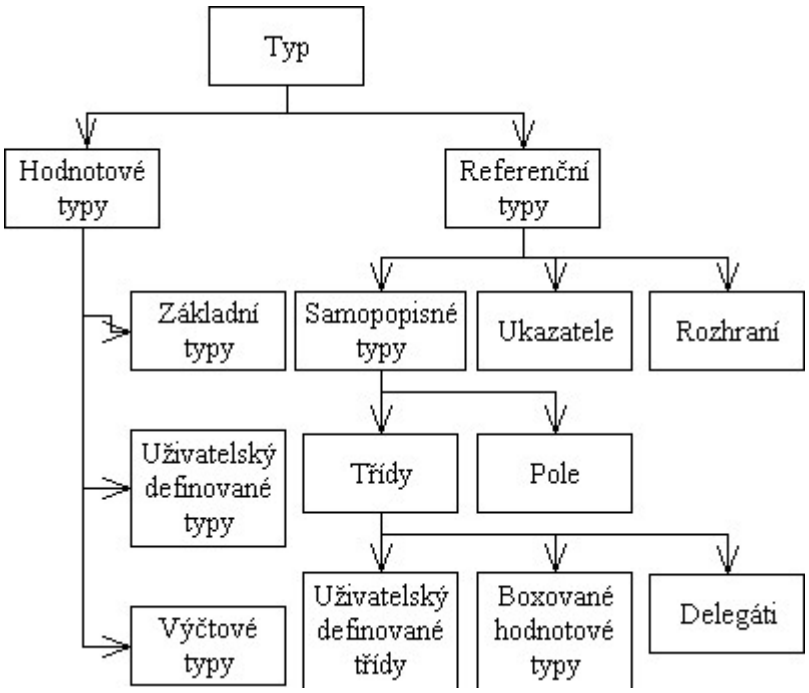
TODO. Komentář který po ní následuje se zobrazí v panelu aplikace Visual Studio s názvem *Task List*. Jednořádkové komentáře uvozené třemi lomítky budou obsaženy v dokumentaci, která je standardně generována ze zdrojového kódu (podobně jako v Javě /** */). Generovaná dokumentace využívá XML.

Kapitola 3. Typový systém jazyka

V první kapitole se seznámíme různými datovými typy, které lze v C# použít.

Na následujícím obrázku vidíte základní strukturu datových typů v jazyce C#.

Obrázek 3.1. Základní typy



Jazyk C# (podobně jako jiné programovací jazyky v prostředí .NET Framework) využívá společný typový systém CTS. Jazyk sám žádné speciální, s ostatními jazyky nekompatibilní, typy nepřináší. Typový systém je rozdělen do dvou hlavních kategorií: *hodnotové typy* a *referenční typy*.

Hodnotové typy

Instance hodnotových typů (proměnné) jsou ukládány na zásobník programu a program s nimi pracuje přímo. Hodnotové proměnné, jak napovídá jejich název, obsahují pouze hodnotu proměnné daného typu bez jakýchkoliv doplňujících informací. Hodnotové typy v jazyce C# lze dále rozdělit do tří kategorií: *základní typy*, *struktury* a *výčtové typy*.

Primitivní datové typy

Základní typy se příliš neliší od jiných programovacích jazyků. Základních typů je 14.

Celočíselné datové typy

Seznam primitivních celočíselných datových typů vyskytujících se v C# je uveden v tabulce primitivních datových typů celočíselných.

Tabulka 3.1. Primitivní datové typy celočíselné

Typ	Rozsah	Velikost	Typ v .NET Framework
sbyte	-128	znaménkové 8-bitové celé číslo	**System.SByte**

Typ	Rozsah	Velikost	Typ v .NET Framework
byte	0 až 255	neznaménkové 8-bitové celé číslo	**System.Byte**
char	U+0000 až U+ffff	Unicode 16-bitový znak	**System.Char**
short	-32 768 až 32 767	znaménkové 16-bitové celé číslo	**System.Int16**
ushort	0 až 65535	neznaménkové 16-bitové celé číslo	**System.UInt16**
int	-2 147 483 648 až 2 147 483 647	znaménkové 32-bitové celé číslo	**System.Int32**
uint	0 až 4 294 967 295	neznaménkové 32-bitové celé číslo	**System.UInt32**
long	-9 223 372 036 854 775 808 až 9 223 372 036 854 775 807	znaménkové 64-bitové celé číslo	**System.Int64**
ulong	0 až 18 446 744 073 709 551 615	neznaménkové 64-bitové celé číslo	**System.UInt64**

Při přiřazování hodnot proměnným nesmíme zapomenout na to, jak se v C# s čísly zachází. Pokud pracujeme s typem **byte**, nelze například jednotlivé proměnné typu **byte** sčítat bez explicitní konverze do **byte**.

Ukázka kódu

```
byte x = 1, y = 2, sum = 0;
sum = x + y; // nelze implicitne konvertovat int na byte (operator +)

byte x = 1, y = 2, sum = 0;
sum = (byte)(x + y); // explicitni konverze, toto lze
```

Pokud také používáme přetížené metody, které mají např. jako parametr typ **int** a následně typ **byte**, musíme explicitně uvést, s jakou metodou chceme pracovat, aby se program zachoval korektně.

Ukázka kódu

```
public void MaMetoda(int x) {}
public void MaMetoda(byte x) {}
...
MaMetoda(3); // zavola se metoda s parametrem typu int
MaMetoda((byte)3); // zavola se metoda s parametrem typu byte
```

Reálné datové typy a typ decimal

Jazyk C# obsahuje datové typy **float** a **double** pro počítání s desetinnou čárkou a navíc obsahuje typ **decimal**. Tento typ je 128-bitový a je vhodný pro měnové a peněžní operace. Má větší přesnost a menší rozsah než zbylé dva reálné datové typy.

Tabulka 3.2. Reálné datové typy a typ decimal

Typ	Přibližný rozsah	Přesnost	Typ .NET Framwork
float	$1{,}5 \times 10^{-45}$ až $3{,}4 \times 10^{38}$	7 desetinných míst	**System.Single**
double	5×10^{-324} až $1{,}7 \times 10^{308}$	15-16 desetinných míst	**System.Double**
decimal	$1, \times 10^{-28}$ až $7{,}9 \times 10^{28}$	28-29 podstatných des. míst	**System.Decimal**

Mezi typy s pohyblivou desetinnou čárkou a typem **decimal** neexistuje implicitní konverze. Proto vždy, když budeme provádět konverzi na (z) tento typ (tohoto typu), musíme explicitně uvést typ, do kterého se má převádět. Zároveň reálná konstanta se považuje za typ **double**, tedy i tehdy při přiřazení hodnoty do proměnné typu **decimal** musíme za konstantou uvést příponu (*m*, *M*).

Struktury

Struktury jsou vlastně uživatelsky definovaným hodnotovým typem. Je to jakási odlehčená třída. Od tříd se ale liší. Proměnné typu struktura jsou umístěny na zásobník, struktury nepodporují dědičnost a struktura sama se nemůže stát základem pro vytvoření nového typu. Struktury jsou logickým sdružením atributů, metod, vlastností, indexerů, operátorů, a případně dalších vnořených typů. Struktury mohou mít definovány dokonce i konstruktory.

Ukázka kódu

```
struct Zvirata_A_Nohy {
   int pocetNohou;
   string nazevZvirete;

   public Zvirata_A_Nohy(int novyPocetNohou, string novyNazevZvirete) {
      pocetNohou = novyPocetNohou;
      nazevZvirete = novyNazevZvirete;
   }

   public int vratPocetNohou() {
      return pocetNohou;
   }

   public string vratNazevZvirete() {
      return nazevZvirete;
   }
}
```

Když máme takto vytvořenou strukturu a chceme ji někde použít, je třeba vytvořit její instanci pomocí operátoru **new**. Pak už můžeme využívat všeho, co nám daná struktura nabízí.

Příklad ke stažení

Celý příklad se nachází na tomto místě [examples/2_2_struct.cs].

Výčtové typy

Výčtové typy se hodí pro celočíselné konstanty. Právě díky tomu, že každé položce ve výčtovém typu můžeme přiřadit i jejich hodnoty. Dokonce lze dvěma položkám přiřadit stejnou hodnotu a C# si potom vybere jednu hodnotu jako primární (z důvodů *reflexe* a řetězcových konverzí). Výčtové typy mají následující omezení:

- nemohou definovat své vlastní metody
- nemohou implementovat rozhraní
- nemohou definovat své indexery nebo vlastnosti

Ukázka kódu

```
enum DnyVTydnu {
   neděle, pondělí, úterý, středa, čtvrtek, pátek, sobota
};
```

Pokud není uvedeno explicitně jinak, první položka má hodnotu *0*. Pokud bychom chtěli získat hodnoty jednotlivých položek, musíme opět provést explicitní konverzi.

Ukázka kódu

```
int i = DnyVTydnu.neděle; // toto nelze
int i = (int)DnyVTydnu.neděle; // toto je spravne
```

Před klíčovým slovem **enum** mohou být rovněž modifikátory určující místo použití - **private**, **public**, **protected**, **internal**.

Referenční typy

Na rozdíl od hodnotových typů - referenční neuchovávají přímo hodnotu samotnou, nýbrž odkaz na místo v paměti, konkrétně na hromadě, kde je skutečná instance uložena. Tuto instanci nazveme objektem. V jazyce C# existují tyto referenční typy:

- typ *object* – jde o alias třídy *System.Object*. Všechny ostatní třídy rozšiřují tuto třídu. *Object* je tedy společným základem pro všechny ostatní typy. Do takové proměnné můžeme přiřadit jakýkoliv jiný typ;
- typ *string* – slouží k uložení textových řetězců. Jde opět o alias k třídě *System.String*. Práce s proměnnou typu String je podobná jako v Javě. Takovouto proměnnou lze přímo naplnit řetězcovou konstantou (bez nutnosti použití operátoru new). String se od ostatních tříd liší při porovnávání, jsou porovnávány hodnoty objektů (textové řetězce), ne pouze hodnoty odkazů;
- typ třída (class);
- typ rozhraní (interface);
- typ pole;
- typ delegát (delegate).

Ukázka kódu

```
string a = "acko";
string b = a;
```

Nullable typy

Běžným hodnotovým typům nelze přiřadit hodnotu *null*, tu lze přiřadit pouze referenčním typům. Nastává problém, pokud hodnotu *null* chápeme jako "bez hodnoty" místo "bez odkazu". Hodnota *null* představuje také jednu z typických hodnot v databázových sloupcích, kde i hodnotové typy mohou mít hodnotu *null*. C# 2.0 nabízí řešení v podobě typů s hodnotou *null* nazývané **nullable typy**.

Nullable typy deklarujeme tak, že využijeme předdefinovanou generickou třídu *Nullable<T>*, která může uchovat libovolnou hodnotu. Tudíž jak hodnotovou, tak referenční, ale zároveň vždy i hodnotu *null*.

Existuje také nová syntaxe s operátorem *?*, která mimo jiné nahrazuje použití názvu *Nullable* ve špičatých závorkách na konkrétní typ.

Nullable typy jsou zkonstruovány pomocí typového parametru *?*. Například **int?** je nullable forma předdefinovaného typu **int**. **Základní typ** nullable typů může být pouze hodnotový typ, který není typu nullable. Nullable typ je struktura, která kombinuje hodnotu základního typu s booleovským indikátorem *null*. Instance nullable typu má dvě neměnné, veřejné vlastnosti:

1. *HasValue*, typu **bool**. *HasValue* je **true** pro instance, které neposkytují hodnotu typu *null* a **false** pro *null* instance.
2. *Value*, má typ podle základního typu nullable typu. Vrací obsažený typ, když *HasValue* je true. V opačném případě, když je *HasValue* false, při pokusu o zpřístupnění jejího obsahu vyvolá výjimku.

Existuje také implicitní konverze z jakéhokoliv non-nullable hodnotového typu na nullable formu tohoto typu. Dále také existuje implicitní konverze z literálu *null*, na kterýkoliv nullable typ. Více v příkladu níže:

Ukázka kódu

```
int? x = 123;
int? y = null;
if (x.HasValue) Console.WriteLine(x.Value);
if (y.HasValue) Console.WriteLine(y.Value);
```

Hodnota 123 typu **int** a literál *null* jsou při deklaraci implicitně přetypovány na nullable typ **int?**. Výstup na obrazovku bude proveden pouze pro proměnnou *x*, jejíž booleovská hodnota metody *HasValue* je true, neboť *x* obsahuje non-nullable hodnotu. Proto se proměnná *y* nevypíše, neb neprojde přes podmínku příkazu *if*.

Další ukázka deklarací využívajících nullable konverze:

Ukázka kódu

```
int i = 123;
int? x = i;   // int --> int?
double? y = x;   // int? --> double?
int? z = (int?)y;  // double? --> int?
int j = (int)z;                // int? --> int
```

Uživatelem definovaný operátor konverze má větší prioritu, když nastane situace, že zdroj i cíl konverze jsou nullable hodnotové typy.

Kapitola 4. Základní prvky jazyka

V této kapitole budou rozebrány základní vlastnosti jazyka C#. Začneme jmennými prostory a budeme pokračovat dalšími vlastnostmi jazyka.

Direktivy preprocesoru

Podobně jako v jazyce C/C++ i jazyk C# umožňuje využít několik direktiv preprocesoru a tak řídit předzpracování zdrojového kódu. Direktivy preprocesoru začínají znakem: #. Existuje celá řada různých direktiv **#define**, **#undef**, **#if**, **#endif**, **#elif, #else**. Pomocí nich lze ovlivnit, které části kódu budou zpracovávány.

Další zajímavou direktivou preprocesoru je **#pragma**. Ta umožňuje zakázat či povolit vypisování některých varovných hlášení.

Velice užitečná je direktiva **#region** a **#endregion**. Ty jsou preprocesorem ignorovány, ale využívá jich například Visual Studio 2005. Umožňují definovat oblasti v kódu. Tyto pak umí VS formátovat (zabalit, rozbalit). Takto zpřehledňují vytvořený zdrojový program.

Jmenný prostor (namespace)

Co je to jmenný prostor (namespace)

Jmenné prostory slouží k tomu, aby logicky sdružovaly typy (jednoduché - např.: **int**, strukturované - např.: třídy), které spolu souvisejí. Mohou sdružovat i podřízené jmenné prostory.

Pokud chceme oznámit, který jmenný prostor chceme používat, jednoduše to provedeme pomocí klíčového slova **using** . Následuje jmenný prostor, který chceme používat.

Pokud například budeme chtít používat jmenný prostor **System**, nebudeme muset pro výpis na konsoli již psát **System.Console.WriteLine(...)**, ale postačí pouze **Console.WriteLine(...)**.

Užitečné jmenné prostory

1. Jmenný prostor **System** obsahuje mimo jiné tyto užitečné typy
    1. **Array** - umožňuje práci s poli
    2. **Console** - práce s konzolí
    3. **Math** - třída pro matematické funkce a operace
    4. **Random** - generování náhodných čísel
    5. **String** - práce s řetězci
    6. **DateTime** - struktura pro práci s datem a časem
2. **System.Collections** obsahuje třídy kolekcí a rozhraní pracující s kolekcemi. Podrobněji bude **System.Collection** probrána ve cvičení 5.
3. **System.IO** obsahuje třídy pro souborové operace
4. **System.Data** obsahuje třídy pro vytváření architektury datového přístupu ADO.NET
5. **System.Net** používá síťové operace (služby pro přístup k Internetu apod)
6. **System.Xml** slouží pro práci s XML daty

7. **System.Threading** pro práci s vlákny
8. **System.Security** obsahuje třídy zajišťující bezpečnost. Obsahuje jmenný prostor **Cryptography** (kódování), systémy pro zjištění totožnosti uživatele apod.

Tvorba jmenného prostoru

Jak bylo napsáno dříve, existující jmenné prostory používáme pomocí klíčového slova **using**. Např. chceme využívat typy jmenného prostoru **System**:

Ukázka kódu

```
using System;
```

Tato konstrukce nám umožní, abychom nemuseli třeba při výpisu na konsoli používat příkaz **System.Console.WriteLine(...)**, ale vystačíme si pouze s **Console.WriteLine(...)**

Nyní si ukážeme, jak lze vytvářet vlastní jmenné prostory. Nový jmenný prostor vytvoříme použitím klíčového slova **namespace**.

Ukázka kódu

```
namespace MyNamespace {
   class MyClass {
      public MyClass() { ... }
   }
}

class Test {
   public static void Main() {
      MyNamespace.MyClass mc = new MyNamespace.MyClass();
   }
}
```

V příkladu jsme si vytvořili jmenný prostor **MyNamespace**, ve kterém jsme implementovali třídu **MyClass**. přístup ke třídě **MyClass** lze poté provést buď s použitím názvu jmenného prostoru + názvu třídy nebo pomocí **using MyNamespace;**.

V C# lze vytvářet i *vnořené jmenné prostory*. Přístup do takového jmenného prostoru se poté provede použitím jména vnějšího (vnějších) a poté námi požadovaného (vnitřního). Např.:

Ukázka kódu

```
namespace Outer {
   namespace Inner {
      public class InnerClass {
         public InnerClass() { ... }
      }
   ...
}

class Test {
   public static void Main() {
      Outer.Inner.InnerClass ic = new Outer.Inner.InnerClass();
   }
}
```

Třídy

Implementace tříd a jejich členů v jazyce C# se příliš neliší od implementace v jazyce Java. Třída může obsahovat tyto členy:

1. položky (field) – členské proměnné, udržují stav objektu;

2. metody – jde o funkce, které implementují služby objektem poskytované. Každá metoda má návratovou hodnotu, pokud nic nevrací, je označena klíčovým slovem *void*. V jazyce C# lze přetěžovat metody. Přetížené metody se musí lišit v počtu parametrů, v typu parametrů nebo v obojím. Výjimečná metoda je statická metoda s názvem *Main*. Právě pomocí této metody je projekt spouštěn. Je-li v projektu definováno více metod Main, je nutné při kompilaci zadat jako parametr jméno jedné třídy z těchto tříd. Metoda Main této třídy je pak spuštěna. Každá třída může obsahovat maximálně jednu metodu Main;

3. vlastnost (property) – je také označována za chytrou položku. Navenek vypadají jako položky, ale umí kontrolovat přístup k jednotlivým datům;

4. indexer – u některých tříd je výhodné definovat operátor *[]*. Indexer je speciální metoda, která umožňuje aby se daný objekt choval jako pole;

5. operátory – v jazyce C# máme možnost definovat množinu operátorů sloužících pro manipulaci s jejími objekty;

6. událost (event) – jejím účelem je upozorňovat na změny, které nastaly např. v položkách tříd.

U jednotlivých členů třídy můžeme použít modifikátory přístupu. Modifikátor je nutné aplikovat na každého člena zvlášť. Jeho uvedení není povinné, implicitní hodnota je *private*. Možné modifikátory jsou:

1. public – člen označený tímto modifikátorem je dostupné bez omezení;

2. private – člen je přístupny pouze členům stejné třídy;

3. protected – přistupovat k takovému členu můžeme uvnitř vlastní třídy a ve všech třídách, pro které je třída základem;

4. internal – člen je přístupný všem v rámci jedné assembly.

Dalším možným modifikátorem je modifikátor static, pomocí něj lze deklarovat, že je daný člen třídy statický. Uvnitř nestatických metod třídy můžeme použít klíčové slovo *this*, to reprezentuje referenci objektu na sebe sama.

Ukázka kódu

```
class Point
{
  short x; //implicitně private
  private short y; //explicitně jsme uvedli, že jde o privátní položku

  public short GetX()
  {
    return x;
  }
  public short GetY()
  {
    return y;
  }
}
```

Metody a parametry

Parametry jsou obvykle předávány do metody hodnotou. Funkce získá kopii skutečných parametrů a jakékoliv modifikace uvnitř těla metody se nepromítnou zpět. V jazyce C# máme dvě řešení.

1. Předání parametru odkazem. Metoda si nevytváří vlastní kopii, nýbrž přímo modifikuje proměnnou, která ji byla předána. Takovéto parametry označíme klíčovým slovem *ref*. Předávané parametry musí být inicializovány.

2. Definovat parametry jako výstupní. Takové parametry označíme klíčovým slovem *out*. Parametry pak přenášejí své hodnoty směrem ven z metody. Hlavním rozdílem oproti předávání parametrů odkazem je, že předané proměnné nemusí být inicializovány před voláním metody. Uvnitř těla funkce je brán parametr jako neinicalizovaný.

Ukázka kódu

```
class Test
{
  public void Swap(ref int a, ref int b){
    int tmp;
    tmp=a;
    a=b;
    b=tmp;
  }

  public void GetXY(Point somePoint,out int x, out int y){
    x=somePoint.GetX();
    y=somePoint.GetY();
  }
  public static void Main()
  {
    int a=5,b=3;
    Swap(ref a,ref b);
    GetXY(new Point(1,2),out a,out b);
  }
}
```

Metoda s proměnným počtem parametrů

Chceme-li definovat metodu s proměnným počtem parametrů, využijeme klíčové slovo *params*. To musíme uvést před posledním parametrem. Typ tohoto parametru musí být pole.

Ukázka kódu

```
public void someMethod(params object[] p); //deklarace metody s proměnným počtem
parametrů
```

Konstanty

Chceme-li realizovat konstanty v jazyce C#, máme dvě možnosti. Definovat konstantu pomocí klíčového slova *const* . Žádná metoda pak nesmí takovou to hodnotu modifikovat. Jediné vhodné místo pro její inicializaci je definice. Druhou možností je implementovat položku jako read-only. Takovou to položku definujeme pomocí klíčového slova *readonly*. Rozdíl proti konstantě je, že takovouto položku můžeme inicializovat v konstruktoru. Konstanty jsou překladačem jazyka C# chápány jako statické položky.

Konstruktory a destruktory

Konstruktor nevrací žádnou hodnotu (ani void). Každá třída má definovaný implicitně konstruktor bez parametrů a s prázdným tělem. Každý objekt by pak měl být vytvořen pomocí operátoru *new*. Jiným typem konstruktoru je tzv. statický konstruktor. Pomocí něj lze inicializovat statické položky. Jiné položky pomocí tohoto typu konstruktoru inicializovat nelze. Tento konstruktor nesmí mít žádné parametry a je vyvolán ještě před vytvořením prvního objektu. Pro statické konstruktory dále platí, že jsou spouštěny v náhodném pořadí. Nelze se tedy v jednom spoléhat na druhý.

Ukázka kódu

```
class Point
{
  static short dimension;
  short x=0
  private short y=0;

  public Point()
  {
  }
  public Point(short nx, short ny)
  {
    x=nx;
    y=ny;
```

```
  }
  static Point()
  { //statický konstruktor
    dimension=2;
  }
  static void Main()
  {
    Point a = new Point();
    Point b = new Point(1,2);
  }
}
```

Chceme-li zavolat konstruktor stejné třídy, můžeme to udělat pomocí klíčového slova *this*. Syntaxi demonstruje následující příklad.

Ukázka kódu

```
public Point(Point p):this (p.GetX(), p.GetY())
{
}
```

Destruktor má název začínající tildou (~) následovaný jménem třídy. Od konstruktoru se liší tím, že nesmí mít žádné formální parametry a nemůže být přetížen. Pokud to není nutné, nemusí být definován.

Vlastnosti

Vlastnosti se navenek chovají jako veřejné položky třídy, avšak interně jsou tvořeny množinou přístupových metod, které realizují zápis a čtení vlastností.

Ukázka kódu

```
class Point
{
  private short x, y;
  public short X
  {
    get
    {
      return x;
    }
    set
    {
      x = value;
    }
  }
  public short Y
  {
    get
    {
      return y;
    }
    set
    {
      y = value;
    }
  }
  public static void Main()
  {
    Point p = new Point();
    p.X = 1;
    p.Y = 2;
  }
}
```

Metody set a *get* jsou spouštěny při operaci čtení a zápisu do vlastností. Vynecháním jedné z metod dostaneme *read only* respektive *write only* vlastnost. Předávaná hodnota do metody *set* je v jejím těle reprezentována klíčovým slovem *value*. V představeném příkladě jsou vlastnosti interně realizovány pomocí privátních položek. To ale nemusí být pravidlem. Vlastnosti mohou například zapisovat a číst přímo z nějakého portu, nebo

realizovat komunikaci prostřednictvím sítě.

Odlišná přístupová práva pro get a set

Vlastnosti (properties) musely mít ve verzi jazyka 1.0 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. Od verze 2.0 je možno přiřadit každému akcesoru jinou úroveň oprávnění.

Ukázka kódu

```
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.

Ukázka kódu

```
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`.

Indexer

Přidáním této členské položky třídě lze pracovat s třídou jako s polem a tedy indexovat ji. Obecná syntaxe indexeru používá klíčové slovo *this* s hranatými závorkami. Rozšiřme předcházející příklad o definici třídy reprezentující pole bodů. Tato třída bude obsahovat indexer. Pole bude mít omezený rozsah. Pokud se pokusíme vložit prvek mimo tento rozsah nestane se nic. Pokud čteme bod mimo tento rozsah vrátí objekt bod se souřadnicemi 0,0.

Ukázka kódu

```
class Point
{
  public int x,y;
  public Point(int x,int y)
```

```
  {
    this.x=x;this.y=y;
  }
}
class SetOfPoints
{
  Point[] points;
  public SetOfPoints(int size)
  {
    points=new Point[size];
  }
  public Point this[int index]
  {
    set
    {
      if (index<points.Length) points[index]=value;
    }
    get
    {
      if (index<=points.Length) return points[index];
      else return new Point(0,0);
    }
  }
}
class RunApp
{
  public static void Main()
  {
    SetOfPoints setOfPoints=new SetOfPoints(2);
    setOfPoints[3]=new Point(1,1); //pokud by indexer nefungoval, skončí výjimkou
                                   //IndexOutOfRangeException
    Point p=setOfPoints[0]; //bude null...
  }
}
```

Dědičnost a polymorfismus

Jazyk C# definuje pouze jednoduchou dědičnost. Dědičnost v definici třídy vyjádříme dvojtečkou uvedenou za jménem třídy a názvem základní třídy. Pro metody odvozené třídy pak platí:

- chceme-li předefinovat veřejnou metodu třídy kterou dědíme, musíme použít klíčové slovo *new*. V tomto případě záleží na typu reference, jaká metoda se zavolá;
- chceme-li realizovat polymorfismus, použijeme virtuální metody. Postup je následující: metodu základní třídy označíme jako virtuální (klíčové slovo *virtual*) a metodu v odvozené třídě označíme klíčovým slovem *override* (má se chovat polymorfně).

Ukázka kódu

```
class A
{
  public void SomeMethod()
  {
  }
  public virtual void AnotherMethod()
  {
  }
}
class B : A
{
  public new void SomeMethod()
  {
    //původní metoda je překryta
  }
  public override void AnotherMethod()
  {
  }
}
class Run
```

```
{
  static void Main()
  {
    A a=new B();
    a.SomeMethod(); //zpustí původní metodu třídy A
    a.AnotherMethod(); //zpustí metodu třídy B
  }
}
```

Konstruktory v odvozených třídách

Chceme-li volat konstruktor základní třídy v odvozené třídě, můžeme k tomu využít klíčové slovo *base*. Syntaxi demonstruje následující příklad:

Ukázka kódu

```
public SomeName(…):base(…){ … }
```

Další modifikátory

- Modifikátor *sealed* - označíme-li tímto modifikátorem třídu, nelze ji již dále rozšiřovat. Případný řetězec dědičností touto třídou končí. Aplikujeme-li tento modifikátor na metodu nebo vlastnost, určíme že takovýto člen třídy nemůže být předefinován. Metoda, kterou chceme označit jako sealed misí být virtuální a musí být předefinovaná! Modifikátor sealed v takovém případě musí být kombinován s modifikátorem override.
- Modifikátor *abstrakt* - tento modifikátor lze opět aplikovat na metody a třídy. Třída označená tímto modifikátorem je implicitně virtuální. Abstraktní třídy a třídy s abstraktními členy nelze instanciovat. Na abstraktní metody nelze aplikovat modifikátory sealed a private.

Statická třída

Statické třídy jsou třídy, u nichž neexistují členské metody ani členská data. Doplňují existující návrhový vzor *Sealed class* rozšířením o privátní konstruktor a statické členy. Statické třídy musí být odvozeny od třídy *object*.

Pokud třídu deklarujeme klíčovým slovem *static*, překladač si vynutí dodržení pravidel:

- zabrání založení instancí a dědičnosti
- zabrání použití členských metod a dat

Cvičení

K procvičení

Napište program, který vytvoří strukturu **Point**. Ta bude mít dvě veřejné proměnné **x**, **y** typu **int** reprezentující souřadnice bodu, konstruktor **Point**, který naplní tyto proměnné a metodu **int MeasureDistance(int x2, int y2)**, která vrátí vzdálenost bodu struktury od bodu zadaného v argumentu funkce souřadnicemi *x2, y2*.

[řešení [exercises/1_struct.cs]]

K procvičení

Předchozí program přepište tak, že vytvoříte jmenný prostor **Graphics** a v něm místo struktury vytvoříte třídu **Point**. Tato třída bude implementovat rozhraní **IDistance** obsahující metodu **double MeasureDistance(Point p)** pro zjištění vzdálenosti aktuálního bodu od bodu předaného jako argument funkce **MeasureDistance()**.

[řešení [exercises/1_idist.cs]]

Neúplné typy

Neúplné typy umožňují programátorovi rozdělit implementaci do několika samostatných souborů. Přesněji platí pro zdrojový kód jedné třídy, struktury a rozhraní. Neúplné typy jsou deklarovány pomocí typového modifikátoru *partial* a musí být umístěny ve stejném jmenném prostoru jako ostatní rozdělené části.

Výhody využití neúplných typů:

- několik vývojářů může ve stejnou chvíli pracovat nad zdrojovým kódem jednoho datového typu, který musí být stejný a je rozdělen do individuálních souborů.
- uživatelský kód může být oddělený od generovaného kódu, což zabraňuje nástrojům přepsat uživatelské změny. Tato vlastnost je využita u automaticky generovaného kódu *WinForms* a u silně typových *DataSetů*.
- možnost rozdělení větších implementací
- možné ulehčení údržby a řízení změn ve zdrojovém kódu
- umožňuje psaní kódu *code-beside* narozdíl od *code-behind*, což je využito u tvorby webových aplikací v ASP.NET 2.0

Při využití těchto technologií dochází k částečnému generování kódu, který je následně doplněn specifickým kódem, který dodává programátor. Aby nedocházelo ke kolizi mezi zapsaným kódem a automaticky vygenerovaným kódem je definice tříd, struktur nebo rozhraní rozdělena do individuálních souborů. Individuální soubory poté spojují jeden samostatný typ.

- Využitím neúplných typů nijak neovlivníme spustitelný kód.
- Soubory, které definují jeden typ jsou spojeny až při přeložení a nelze je později jakýmkoliv způsobem rozšířit nebo modifikovat.
- Neúplné typy nepovolují stejné deklarace proměnných v různých částech.
- Modifikátor *partial* není povolen pro deklarace delegátů a objektů typu **enum**.

Rozdělení neúplných typů podle kódu

Komulativní - takový kód, kde jsou části ze všech souborů spojeny v jednu výslednou. Mohou to být například metody, položky a rozhraní.

Nekomulativní - kód musí být ve všech souborech stejný. Jako například jednotlivé typy, viditelnost, či definice základní třídy konkrétního datového typu.

Pracujeme-li s neúplnými typy například v prostředí Visual Studio 2005, pak *Class View* nebo různé navigační lišty, vždy zobrazují typ jako celek, a to nezávisle na tom, do kolika souborů je rozdělen.

Neúplné typy v příkladech

Následující příklad prezentuje ukázku na založení a praktické využití neúplných typů, kdy je neúplná třída implementována do dvou částí. Tyto dvě části mohou být v různých zdrojových souborech. Například první část může být strojem generovaná databáze představující nástroj a druhá část je ručně napsaná uživatelem, jak tomu je u třídy *Customer*:

Ukázka kódu

```
public partial class Customer
{
    private int id;
    private string name;
    private string address;
    private List<Order> orders;
    public Customer() {
        ...
    }
```

```
}

public partial class Customer
{
    public void SubmitOrder(Order order) {
        orders.Add(order);
    }
    public bool HasOutstandingOrders() {
        return orders.Count > 0;
    }
}
```

Zde vidíme rozložení třídy *Customer* na dvě části, kde v první části jsou definovány vlastnosti a konstruktor neúplné třídy. Druhá část zahrnuje pouze uživatelem ručně psané metody.

Pole

Pole chápeme jako *množinu proměnných stejného datového typu*. S takovým polem pak zacházíme jako s celkem. Na jednotlivé prvky pole (proměnné) se můžeme odkazovat pomocí jejich *indexů* pole a poté s nimi pracovat jako s obyčejnými proměnnými. Každý počáteční prvek pole v C# má index *0*, pokud tedy máme pole délky *N*, index posledního prvku bude *N-1*.

Deklarace pole

V C# existují dva způsoby, jak deklarovat pole. První ze způsobů více naznačuje objektovost polí. Ten si ukážeme jen "ve zkratce", více se budeme zabývat druhým, klasickým a určitě i praktičtějším způsobem.

Jazyk C# umožňuje vytvořit instanci pole pomocí statické metody **System.Array.CreateInstance()**. Tento způsob více odpovídá objektové charakteristice tvorby objektů.

Ukázka kódu

```
using System;
...
Array pole = Array.CreateInstance(typeof(int), 4);
```

Vytvořili jsme instanci třídy **Array** typu **int** se čtyřmi prvky. Pokud budeme toto pole chtít naplnit, nemůžeme již k jednotlivým prvkům přistupovat pomocí indexů, jako jsme byli zvyklí u běžných polí. Je třeba k těmto prvkům přistupovat pomocí metod **SetValue()** a **GetValue()**.

Ukázka kódu

```
...
pole.SetValue(3, 0); // na nultou pozici pole vlozime cislo 3
Console.WriteLine(pole.GetValue(0)); // vytiskneme 0. prvek pole
...
```

Stejným způsobem, ale i s možností přistupovat k jednotlivým prvkům pole pomocí indexů, můžeme tehdy, pokud vytvoříme pole druhou (následující) konstrukcí. První způsob tedy opustíme.

Pro jednorozměrná pole v jazyce C# platí deklarace:

typ[] nazev_pole;

Vícerozměrná pole

Pokud chceme deklarovat vícerozměrné pravidelné pole, píšeme nejprve typ pole, následují hranaté závorky, do kterých napíšeme pouze čárky podle toho, kolikarozměrné pole chceme používat. Příklad: Deklarace 2-rozměrného pole s názvem **pole** typu **int**

Ukázka kódu

```
int[,] pole = new int[2,2] { {2, 1}, {4, 3} };
```

Vícerozměrná nepravidelná pole

V jazyce C# lze používat i nepravidelná pole. Nepravidelná pole se vyznačují různou délkou jednotlivých "řádků" pole. Pokud například chceme vytvořit pole, které má v prvním řádku 3 prvky a v druhém 2 prvky, provedeme to následovně:

Ukázka kódu

```
int[][] pole;
pole = new int[2];
pole[0] = new int[3];
pole[1] = new int[2];
```

Přístup k jednotlivým prvkům pole se pak provádí následovně: **pole[0][0]** nám vrátí hodnotu prvku v první řadě a prvním sloupci.

Práce s poli

Spousta užitečných operací, které s poli provádíme, se nachází ve třídě **System.Array**.

Kopírování polí

Místo obvykle používaného cyklu pro kopírování pole **x** do pole **y**:

Ukázka kódu

```
for(int i=0; i<=x.Length; i++) y[i] = x[i];
```

lze použít metodu **CopyTo()** dostupnou všem polím

Ukázka kódu

```
CopyTo()
x.CopyTo(y,0);
```

kde první parametr udává cílové pole, do kterého se má kopírovat druhý parametr udává počáteční index, na který se bude ukládat. Pokud cílové pole od daného počátečního indexu nebude dostatečně dlouhé, vyvolá se výjimka.

Třída **Array** nabízí také metodu pro vyhledávání hodnoty v poli **Array.BinarySearch()**, implementuje i třídění polí metodou **Array.Sort()**, převrácení pole **Array.Reverse()**. Pro vrácení indexu prvku na základě rovnosti hodnot (objektů) slouží metody **Array.IndexOf()** a **LastIndexOf()**.

Příklad ke stažení

Ukázkový příklad demonstrující některé metody třídy **Array** si můžete vyzkoušet zde [1/4_1_1.cs].

Rozhraní

Jazyk C# neumožňuje vícenásobnou dědičnost. Díky rozhraním jsme schopni třídám přidávat jisté charakteristiky nebo schopnosti nezávisle na hierarchii tříd. Definice rozhraní začíná klíčovým slovem *interface*, po které následuje jeho jméno. Ve složených závorkách je pak tělo rozhraní, které může obsahovat výčet metod (jen hlavička metody bez těla), vlastnosti, indexerů a událostí. U rozhraní jsou všechny členy automaticky veřejné (public).

Ukázka kódu

```
public interface IComparable
{
  int Compare(Object second);
  int SomeProperties
  {
    get;
  }
  object this[int index]
  {
    get;
    set;
  }
}

class SomeClass : AnotherClass, IComparable
{
  int Compare(Object second)
  {
    ...
  }
  ...
}
```

Jak již bylo řečeno, jazyk C# podporuje pouze jednoduchou dědičnost. Tím jsme se vyhnuli kolizi jmen. C# ale umožňuje, aby třída implementovala několik rozhraní. V takovém to případě se kolizi jmen nevyhneme. Nabízené řešení je kvalifikovat členské jméno jménem rozhraní. U jména metody (nebo jiného člena rozhraní) je vynechán modifikátor public a před toto jméno je přidáno jméno rozhraní. Takovýto člen pak bude přístupný pouze pomocí daného rozhraní (jako by ani nebyl implementován v dané třídě).

Ukázka kódu

```
public interface IA
{
  void Test();
}
public interface IB
{
  void Test();
}
public class SomeClass : IA, IB
{
  void IA.Test() {}
  void IA.Test() {}
  public static void Main()
  {
    SomeClass test = new SomeClass();
    test.Test(); //error
    ((IA)test).Test()
    IB b=new SomeClass();
    b.Test();
  }
}
```

Rozhraní lze rozšiřovat a kombinovat pomocí dědičnost. Takto lze přidávat další metody, slučovat několik rozhraní do jednoho nebo předefinovat existující metodu. Následující příklad demonstruje jak lze kombinovat rozhraní.

Ukázka kódu

```
public interface IA
{
  void Test();
}
public interface IB
{
  void Test();
}
```

```
public interface IC : IA, IB
{
  new void Test();
  void AnotherMethod();
}
```

Při vlastní implementaci takovéhoto rozhraní lze kvalifikovat jednotlivé členy. Při implementaci představeného příkladu máme v podstatě tři možnosti.

Třída může implementovat jednu veřejnou metodu Test.

Třída může implementovat tři kvalifikované metody Test, pro každé rozhraní zvlášť.

Třída může implementovat nějakou kombinaci předcházejících postupů.

Operátory

Množina operátorů v jazyce C# je až na pár výjimek identická s operátory v jiných programovacích jazycích, jako je třeba Java. Následující tabulka uvádí přehled operátorů seřazených dle priority.

Tabulka 4.1. tabulka priority operátorů

Kategorie	Operátory
primární	**(x) x.y f(x) a[x] x++ x-- new typeof sizeof checked unchecked**
unární	**+ - ! ~ ++x --x (T)x**
multiplikativní	*** / %**
aditivní	**+ -**
bitové posuny	**<< >>**
relační	**< > <= >= is as**
rovnost	**== !=**
logické AND (bitové)	**&**
logické XOR (bitové)	^
logické OR (bitové)	|
AND	**&&**
OR	||
podmíněný výraz	**? :**
přiřazení	**= *= /= %= += -= <<= >>= &= ^= |=**

typeof - slouží k *reflexi*. Vrátí instanci třídy *System.Type*.

Ukázka kódu

```
Type t = typeof(Console);
foreach(MemberInfo info in t.GetMembers()) Console.WriteLine(info);
```

sizeof - slouží k zjištění velikosti hodnotových typů. Tato operace je považována za nebezpečnou a proto musí být tento operátor umístěn v bloku *unsafe*.

Definování bloku unsafe se explicitně zříkáme bezpečnostních kontrol prostředí .NET

Blok je definován pomocí klíčového slova unsafe a složenými závorkami.

Je nutné překladači povolit unsafe bloky (Project-Properties-Configuration Properties-Build-Allow Unsafe Code Blocks).

Ukázka kódu

```
unsafe
{
  Console.WriteLine("Size of int: {0}",sizeof(int));
}
```

Při porovnávání stringů pomocí operátoru == je porovnávána hodnota. Pokud chceme porovnat reference, musíme alespoň jeden řetězec přetypovat.

Ukázka kódu

```
string a="hello";
string b="hello";
string c=String.Copy(a);
Console.WriteLine(a==c); //True
Console.WriteLine((object)a==(object)b); //True,same constant.
Console.WriteLine((object)a==(object)c); //Flase
```

Operátory is a as

Pomocí operátoru *is* jsem schopni zjistit, zda je daný prvek instancí dané třídy, případně implementuje-li dané rozhraní.

Ukázka kódu

```
if (someObject is SomeInterface) //vrátí true v případě, že someObject
implementuje dané rozhraní
{
  return someObject;
}
else
{
  return null;
}
```

Operátor *as* provádí přetypování na daný typ. Pokud přetypování není možné, vrací null. Předvedený příklad by tedy šlo implementovat pomocí operátoru as takto:

Ukázka kódu

```
return someObject as SomeInterface;
```

Operátor ??

C# 2.0 poskytuje nový operátor *??* nazvaný **null splývající operátor** (null coalescing operator) . Výsledek výrazu *A ?? B* je *A*, pokud je *A* non-nullable typu. Jinak je výsledek *B*. Pokud *A* obsahuje hodnotu *null*, tak *B* poskytne hodnotu k použití. Výraz *A ?? B* tedy vrací non-nullable hodnotu a poskytne vhodné implicitní přetypování:

Ukázka kódu

```
int? x = GetNullableInt();
int? y = GetNullableInt();
int? z = x ?? y;
int i = z ?? -1;
```

Typ výrazu *x ?? y* je **int?**, ale výraz *z ?? -1* je typu **int**. Operátor splývající null také funguje pro odkazové typy. Ukázka příkladu, kde program vypíše buď proměnnou *s* anebo, pokud je s*null*, vypíše *Unspecified*.

Ukázka kódu

```
string s = GetStringValue();
Console.WriteLine(s ?? "Unspecified");
```

Operátor : :

Ve verzi 2.0 v jazyce C# přibyl nový 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.

```
//this wont work
System.Console.WriteLine("doesnot work!");

//this will work
global::System.Console.WriteLine("works!");
```

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).

Přetěžování operátorů

Jazyc C# umožňuje přetěžování operátorů.

Ukázka kódu

```
public static returnType operator op (object1 [,object2]) {...}
```

Definice operátoru je vždy veřejná a statická. Následuje návratový typ. Obecně to může být libovolný typ. Klíčové slovo *operátor* určuje, že chceme přetížit operátor *op*. Přetěžujeme-li unární operátor definujeme jediný parametr. Pro operátory binární budou parametry dva. Na tyto parametry nejsou kladeny žádné omezení. Je vhodné, aby první parametr byl stejného typu jako třída, jejíž operátor přetěžujeme.

Ukázka kódu

```
class Test
{
  protected string text;
  public Test(string text)
  {
    this.text=text;
  }
  public string Text
  {
    get
    {
      return this.text;
    }
    set
    {
      this.text=value;
    }
  }
  public static Test operator + (Test t1,Test t2)
  {
    return new Test(t1.Text+","+t2.Text);
  }
```

```
}
class RunApp
{
  public static void Main()
  {
    Test a=new Test("A");
    Test b=new Test("B");
    Test c=a+b;
    Console.WriteLine(c.Text);
    Console.WriteLine((a+b).Text);
    a+=b;
    Console.WriteLine(a.Text);
  }
}
```

Následující tabulka ukazuje, které operátory lze přetížit.

Tabulka 4.2. Přetížitelné operátory

Kategorie	Operátory k přetížení
Unární operátory	**+ - ! ~ ++ --**
Binární operátory	**+ - * / % & | ^ << >> != == < > <= >=**
Logické konstanty	**true false**
Indexování	**[]**
konverzní funkce	**(typ)**

Uživatelsky definované konverze

Slouží ke konverzi třídy nebo struktury do jiné třídy, struktury nebo základního typu. Typ, pro kterou je konverze definovaná, se musí objevit buď v parametru nebo musí konverze tento typ vracet a opět (podobně jako u operátorů) musí být definice veřejná a statická.Jsou dva druhy uživatelsky definovaných konverzí:

implicitní - jsou prováděny aniž by uživatel musel cokoliv zadávat;

explicitní - uživatel misí zadat, že chce provést konverzi.

Ukázka kódu

```
public static implicit operator type(object)
public static explicit operator type(object)
```

Použití demonstruje následující příklad:

Ukázka kódu

```
class Test
{
  protected string text;
  public string Text
  {
    get
    {
      return this.text;
    }
    set
    {
      this.text=value;
    }
  }
  public static implicit operator Test(string a)
  {
    Test t=new Test();
    t.Text=a;
    return t;
```

```
  }
  public static explicit operator string(Test a)
  {
    return a.Text;
  }
}
class RunApp
{
  public static void Main()
  {
    Test a="hello";     //implicit conversion
    string b=(string)a; //explicit conversion
  }
}
```

Příkazy

Příkaz můžeme chápat jako krok v cestě algoritmu. Většinou se provádějí v pořadí, ve kterém jsou v programu zapsány a některé příkazy musejí být ukončeny středníkem. V C# se ovšem vyskytují příkazy, které se takto chovat nemusí. Říkáme jim *skokové příkazy*. Podrobněji vše probereme níže.

Základní příkazy

V této sekci si rozebereme základní příkazy jazyka.

Blok (složený příkaz)

Blok je typ příkazu, který může, ale nemusí, reprezentovat další množinu příkazů. Blok označíme stejně jako například v Javě symboly **{** a **}**. Dovnitř bloku můžeme vložit další příkazy, popřípadě další blok. Pokud blok neobsahuje žádné příkazy, nazývá se *prázdný příkaz*. Příklad:

Ukázka kódu

```
// priklad bloku
{
   Console.WriteLine("Outer block");

   //toto je vnoreny blok
   {
      string text = "Inner block";
      Console.WriteLine(text);
   }
}
```

Prázdný příkaz

Mohou nastat situace, kdy na někjakém místě programu jazyk C# vyžaduje přítomnost příkazu, ale my nechceme provádět žádný příkaz. Proto v C# existují i *prázdné příkazy*.

Jak jsme si uvedli v předchozím odstavci, prázdný příkaz můžeme vyjádřit pomocí dvou složených závorek, které nic neobsahují **{}**. Další možnost, jak vyjádřit prázdný příkaz, je použitím středníku **;**. To nám zaručí, že program na daném místě nebude nic provádět.

Příklad, kdy využijeme prázdných příkazů - chceme se dostat v řetězci na první znak, který není mezera (tedy vypustíme z úvodu řetězce mezery):

Ukázka kódu

```
int i=-1;
while((retezec[++i] == ' ') && (i < retezec.Length)) ; // nebo {}
```

Výrazový příkaz

Pokud k nějakému výrazu připojíme středník, stane se z něj tzv. *výrazový příkaz*. Výraz v tomto případě musí

nějakým způsobem měnit proměnnou, vlastnost apod. Například pokud za přiřazením čísla *1* do proměnné **i** použijeme středník, dostaneme výrazový příkaz:

Ukázka kódu

```
i = 1;
```

Deklarace lokální proměnné nebo lokální konstanty

I deklaraci lokální proměnné nebo lokální konstanty považujeme za příkaz. Potom v rámci lokálního bloku lze tyto proměnné či konstanty používat až do konce tohoto bloku.

Podmíněné příkazy

Příkaz if

Podmíněný příkaz **if** má stejnou syntaxi jako v ostatních programovacích jazycích. Pro ty z vás, kteří se s programováním přece jen nesetkali, uvedeme jeho popis.

Příkaz **if** má dvě podoby - úplné **if** a neúplné **if**.

Úplné **if** je tvořeno:

Ukázka kódu

```
if (podminka) { ... } // provede se telo prikazu {}, pokud podminka plati (ma
hodnotu true)
else { ... } // provede se telo prikazu {}, pokud podminka neplati (false)
```

Neúplné **if** je tvořeno podobně jako předchozí úplné **if**, s tím rozdílem, že neuvedeme situaci, kdy podmínka neplatí:

Ukázka kódu

```
if (podminka) { ... } // provede se telo prikazu {}, pokud podminka plati
```

Místo bloku (složeného příkazu) lze použít jednoduchý příkaz. V tom případě za příkazem musíme napsat i středník.

Příkaz **if** (jak úplný, tak i neúplný) můžeme také vnořit do předchozího příkazu **if** a vytvářet tak složité konstrukce na testování i různých podmínek. Objeví se zde ale problém v přehlednosti kódu. Proto je výhodné psát program co možná nejlépe strukturovaně, odsazovat jednotlivé části kódu, bloky, abychom se v nich vyznali buď my nebo ti, kteří po nás kód budou číst. V prostředí Microsoft Visual Studio .Net se o odsazování většinou starat nemusíme, nicméně někdy se stane, že vkládáme část kódu (např. pomocí *Copy-Paste*), která se nezformátuje správně.

Příkaz switch

Příkaz **if** je přehledný, pokud testujeme dvě podmínky. Pak se stává nepřehledným. Příkaz **switch** používáme pro testování více vstupních podmínek. Jeho zápis je ve tvaru:

Ukázka kódu

```
switch (vyraz) {
   case hodnota1 :
      prikazy;
   break;
   case hodnota2 :
      prikazy;
   break;
   ...
   default : prikazy;
}
```

kde **vyraz** značí výraz, který chceme testovat. Musí nabývat pouze hodnot celočíselných, **char**, výčtových nebo **string**. *case hodnota1*, *case hodnota2*... jsou návěští, pro které chceme, aby příkaz **switch** reagoval a **default** je návěští, kam se **switch** dostane, pokud se mezi konstantami *hodnota1*, *hodnota2*... nenajde ani jedna, která by odpovídala hodnotě výrazu. Návěští **default** není povinné, takže pokud **switch** neobsahuje návěští **default** a nenajde konstantu, která by odpovídala hodnotě výrazu, neprovede se nic.

Cykly

Cykly také nazýváme smyčkami nebo *iteračními příkazy*. Jedná se o příkazy, které provádějí cyklicky jeden nebo několik příkazů. Většinou počet opakování závisí na podmínce cyklu (opakování).

Příkaz while

Ukázka kódu

```
while (podminka) { ... }
```

Cyklus **while** se provádí tak dlouho, dokud podmínka **podminka** nabývá hodnoty *true*. Pokud podmínka nabude hodnoty *false*, cyklus skončí (tělo cyklu se přeskočí). U toho cyklu se může stát, že se neprovede ani jednou, neboť pomínka se vyhodnocuje před tělem cyklu.

Příkaz do-while

Cyklus **do-while** je podobný jako cyklus **while** s tím rozdílem, že podmínka se vyhodnocuje až po průchodu těla cyklu. Tedy tělo cyklu se provede pokaždé alespoň jednou.

Příkaz for

Příkaz **for** se používá asi nejčastěji z příkazů pro cykly. Je užitečný jak pro procházení indexů pole nebo jednoduchého opakování na základě změny hodnoty proměnné. Nepoužívá se ale nutně pouze v těchto případech.

Ukázka kódu

```
for(promenna;podminka;zmena_promenne) { ... }
```

Proměnná **promenna** určuje počáteční hodnotu proměnné v cyklu. Proměnná může být v tomto místě i deklarována a inicializována. Podmínka omezuje proměnnou, tedy do jaké hodnoty proměnné se má ještě tělo cyklu vykonávat. Jako poslední se definuje, na jakém základě se bude proměnná měnit (**zmena_promenne**). Následující příklad bude vypisovat na obrazovku hodnotu proměnné od *0* do *9* pokaždé na nový řádek:

Ukázka kódu

```
for(int i = 0; i < 10; i++) System.Console.WriteLine(i);
```

Cyklus **for** ale může fungovat i bez udání proměnné, podmínky a změny proměnné. Poté se tento cyklus bude provádět donekonečna. Aby se takto vytvořený cyklus dal nějak zastavit, pak se používají (kromě *CTRL+C* již za běhu programu) příkazy **break**, **return**, **goto** v jeho těle.

Příkaz foreach

Představme si nějaký kontejner (nyní nemám na mysli velkou nádobu, do které se vyhazuje odpad), který obsahuje různé objekty. My chceme pro každý z objektů zjistit typ tohoto objektu. Samozřejmě to můžeme provést několika způsoby, například příkazem **for**, ale zde se dobře hodí příkaz **foreach**, který prochází všechny prvky kontejneru:

Ukázka kódu

```
using System;
using System.Collections;
...
```

```
ArrayList a = new ArrayList();

a.Add(1);
a.Add("string");
a.Add(new object);

foreach(object obj in pole)
   Console.WriteLine(obj.GetType());
...
```

Program nám následně vypíše:

```
System.Int32
System.String
System.Object
```

Skokové příkazy

Příkazy, které způsobí, že program přeruší přirozenou posloupnost plnění příkazů (tedy za sebou, jak jsou napsány), se nazývají *skokové příkazy*. Tehdy program přejde na jiné místo v programu rozdílné od této přirozené posloupnosti.

Příkaz break

Příkaz **break** způsobí, že program vyskočí z nejbližšího cyklu, ve kterém se právě nachází. Kromě cyklů se **break** smí použít i v příkazu switch, jak jsme uvedli výše. V cyklu se používá následovně:

Ukázka kódu

```
using System.Collections;
...
ArrayList a = new ArrayList();
a.Add(3);
a.Add("string");
a.Add(new object());
foreach(object o in a)
   if(o.GetType().Name == "String") break;
...
```

v cyklu **foreach** postupně zjišťujeme, zda typ objektu je instancí třídy **String**. Pokud ano, příkaz **break** způsobí ukončení cyklu. Zde použitá metoda **GetType()** obvykle vrací kromě názvu typu také jmenný prostor (popřípadě prostory, pokud je vnořený), kam typ náleží. Vlastnost **Name** nám vrátí pouze název typu, což je výhodné. Jinak bychom museli zjišťovat, v kterém jmenném prostoru se daný typ nachází.

Příkaz continue

Lze použít pouze v cyklech. Pokud použijeme tento příkaz, způsobí to, že přeskočíme zbytek těla cyklu a začneme novou iteraci (další opakování cyklu).

Ukázka kódu

```
int sumaSude = 0;

for(int i=0; i<100; i++) {
   if(i % 2 != 0) continue;
   else sumaSude += i;
}
```

Pokud je **i** liché číslo, cyklus začne novou iteraci. Pokud je **i** sudé, přičte se k proměnné **sumaSude**.

Příkaz goto

Tento příkaz se většinou v novějších programovacích jazycích nesetkával s oblibou. Platilo, že čím více příkazů **goto**, tím horší je kvalita programu. Jazyk C# tento příkaz obsahuje, ale snaží se zabránit jeho zneužití, např. nelze skočit dovnitř blokového příkazu. Tento příkaz je doporučován pro použití v příkaze **switch** nebo pro předání řízení do vnějšího bloku, ale ostatní použití není nepovolené.

Ukázka kódu

```
...
// tato konstrukce neni povolena
goto UVNITR_CYKLU;

for(int i=0; i<100; i++) {
UVNITR_CYKLU :
   System.Console.WriteLine(i);
}

...
// takto prikaz goto pouzit lze
for(int i=0; i<100; i++) {
   if(i == 34) goto VEN;
   else ...
}

VEN : ;
...
```

Příkaz return

Tímto příkazem se vrací řízení volající funkci. Příkaz **return** může být spojen i s hodnotou, pokud metoda, která **return** použije, má návratovou hodnotu jinou než **void**.

Ukázka kódu

```
...
long mocnina(int zaklad, int mocnina) {
   long vysledek = 1;

   for(int i = 1; i <= mocnina; i++)
      vysledek *= zaklad;

   return vysledek;
}
...
```

Příkazy checked a unchecked

Pokud používáme konverzi typů, někdy se stane, že při konverzi z jednoho typu na druhý si chceme být jisti, že převod proběhl úspěšně. C# pro ten případ používá příkaz **checked**. Když potom chceme ověřit správnost konverze, uvedeme naši konverzi v kontrolovaném bloku příkazu **checked**.

Ukázka kódu

```
uint iSmall = 3;
byte bSmall;
uint i = 412;
byte b = 0;
...
// standardne je kontrola nastavena jako unchecked
bSmall = (byte)iSmall;
b = (byte)i;
System.Console.WriteLine(b+", "+bSmall);

// nic se nemeni
bSmall = unchecked((byte)iSmall);
b = unchecked((byte)i);
System.Console.WriteLine(b+", "+bSmall);

// zatim vse v poradku
bSmall = unchecked((byte)iSmall);
// zde pri konverzi vyskoci vyjimka
b = checked((byte)i);
System.Console.WriteLine(b+", "+bSmall);
```

Jelikož kontrola správnosti konverze má určitý vliv na výkonnost, nedoporučuje se ji používat u široce distribuovaného softwaru.

Je to ale užitečný pomocník pro kontrolu, zda dané přetypování proběhlo úspěšně. Proto se dá použít při kompilaci přepínač **/checked**, který způsobí, že všechny konverze typů se budou kontrolovat. Poté je ale třeba všechny konverze, které kontrolovat nechceme, označit pomocí příkazu **unchecked**.

Příkaz throw

Příkaz **throw** slouží k vyvolání výjimky. Uplatní se tehdy, pokud chceme ošetřit nějakou část kódu. V případě, kdy se vyskytnou jiné podmínky v programu, než jsme si představovali, můžeme pomocí **throw** vyvolat nějakou výjimku. Příklad:

Ukázka kódu

```
...
// deleni dvou realnych cisel
double vydel(double delenec, double delitel) {
   if(delitel == 0) throw new DivideByZeroException("Nulou nelze delit!");
   else return(delenec / delitel);
}
...
```

Příklad ke stažení

Další příklad na příkaz throw z prostředí dlužníků, pokud určitá třída neimplementuje rozhraní IDluzi [2/3_3_5_throw.cs].

Výjimky

Pokud program narazí na neočekávaný problém, například snažíme-li se zapsat hodnotu do pole na index, který pro toto pole není definován, objeví se *výjimka*. Výjimky můžeme chápat jednak jako takovýto problém a jednak jako objekty nesoucí informaci a vzniklém problému.

Výjimky jako objekty jsou odvozeny ze třídy **Object**. Z této třídy je odvozena základní třída výjimek **Exception**, ze které ostatní třídy výjimek dědí.

Výjimky může vyvolat buď prostředí .Net Framework. Jedná se například o dělení nulou, pokus o vstup mimo hranice pole apod. Můžeme je můžeme vyvolat také sami pomocí příkazu throw.

Hlídaný blok (try)

Pokud chceme ošetřit - *hlídat vznik výjimky*, provedeme příkazy, které by mohly výjimku vyvolat, v *hlídaném bloku*. Strukturu výjimky ukazuje následující příklad.

Ukázka kódu

```
try
{
  …
}
catch  (System.FileException e)
{
  …
}
catch (System.Exception e)
{
  …
}
finally
{
  …
}
```

Vezměme například metodu, kterou jsme použili v části 3.3.6 - dělení nulou.

Ukázka kódu

```
class MatematickeOperace {
...
   static double vydel(double delenec, double delitel) {
      if(delitel == 0) throw new DivideByZeroException("Nulou nelze delit!");
      else return(delenec / delitel);
   }

   public static void Main() {
      // zde vznikne vyjimka, kterou chceme osetrit
      Console.WriteLine(MatematickeOperace.vydel(3,9));
   }
}
```

Hlídaný blok začíná klíčovým slovem **try** a pokračuje samotným blokem. V našem případě budeme hlídat použití statické metody **MatematickeOperace.vydel(...)**

Ukázka kódu

```
...
public static void Main() {
   // deleni uvedeme do hlidaneho bloku
   try {
      Console.WriteLine(MatematickeOperace.vydel(3,0);
   }
   ...
}
```

Hlídaný blok nám sice ohlídá vznik výjimky, ale sám o sobě nemá valnou cenu. Musí být použit ve spojení s dalšími bloky - **catch** a (nebo) **finally**.

Blok obsluhy (catch)

Blok obsluhy se provádí ve spojení s blokem **try** tehdy, pokud v bloku **try** dojde ke vzniku výjimky. Předchozí ukázku rozšíříme o blok **catch**. Jelikož nám hrozí dělení nulou, budeme hlídat výskyt výjimky **DivideByZeroException**. Můžeme použít také nadřazenou obecnou výjimku **ArithmeticException** nebo **Exception**, ze které obě třídy výjimek dědí:

Ukázka kódu

```
...
public static void Main() {
   // deleni uvedeme do hlidaneho bloku
   try {
      Console.WriteLine(MatematickeOperace.vydel(3,0);
   }
   catch (DivideByZeroException e) {
     e.Message;
   }
}
...
```

Pokud víme o více výjimkách, které se mohou v hlídaném bloku objevit, jednoduše přidáme další blok obsluhy.

Výjimky obsahují mimo jiné užitečnou vlastnost **Message**. Ta umožňuje přístup k chybovému hlášení, které se dá zobrazit v případě výskytu výjimky.

Výjimku lze také pouze zachytit bez dalšího zpracování. Potom stačí v bloku catch uvést pouze **catch (Exception) {};**

Koncový blok (finally)

Někdy se nám může stát, že pracujeme v hlídaném bloku a zde se vyskytne výjimka. My budeme chtít, aby se v každém případě - tj. po vzniku výjimky i tehdy, když nevznikne, vykonala určitá část kódu související s hlídaným blokem.

Mějme například soubor, který otevřeme a poté do něj v hlídaném bloku zapisujeme data. Pokud se při zápisu dat vyskytne výjimka, přejde se na blok obsluhy a soubor by mohlzůstat nadále otevřen. Proto můžeme použít *koncový blok*, který způsobí zavření souboru v každém případě - tedy pokud se výjimka objeví i pokud ne.

Ukázka kódu

```
...
public stativ void Main(){
   try {
      Console.WriteLine(MatematickeOperace.vydel(3,0));
   }
   catch (DivideByZeroException e) { e.Message; }
   finally {
     Console.WriteLine("koncovy blok");
   }
...
```

Výjímky definované v .NET

Knihovna tříd .NET Framework definuje celou sadu výjimek. Některé z nich uvádí následující tabulka.

Tabulka 4.3. Tabulka některých výjimek z knihoven :NET Framework

Exception	Základní třída pro všechny výjimky.
SystemException	Základní třída pro všechny druhy výjimek generovaných za běhu aplikace.
IndexOutOfRangeException	Přístup k prvku mimi rozsah pole.
NullReferenceException	Generovaná při pokusu pracovat s neinicializovanou referencí.
ArgumentException	Základní třída pro všechny výjimky způsobené chybnými argumenty.
ArgumentNullException	Parametr s null hodnotou nebyl očekáván.
InteropException	Výjimka generovaná mimi CLR .NET Framework.

V prostředí .NET můžeme definovat vlastní typy vyjímek. Všechny vycházejí ze třídy System.Exception. Možnosti, které při vytváření vlastního typu výjimky máme, demonstrují konstruktory třídy Exception.

```
public Exception();
```

Implicitní konstruktor inicializuje členská data na předdefinované hodnoty.

```
public Exception(string)
```

Nastavuje popis výjimky jako textový řetězec.

```
public Exception(SerializationInfo, StreamingContext)
```

Inicializuje člena třídy tzv. serializovanými daty.

```
public Exception(string, Exception)
```

Dává nám možnost vytvořit událost obsahující data jiné události. Takto můžeme tvořit celou hierarchii do sebe zanořených událostí.

Delegáti

V jazyce C existuje ukazatel na metodu. Obdobný mechanismus poskytuje i jazyk C#. Jde o delegáty. Oproti ukazatelům na funkce jsou však delegáti typově bezpeční. Delegáta deklarujeme pomocí klíčového slova *delegate*.

Ukázka kódu

```
class SomeClass
{
  public delegate void DelegatsName(int a);
}
```

Uvedený příklad představuje deklaraci delegáta. Nejde o členskou položku třídy, ale pouze o deklaraci typu. Chceme-li delegáta použít, musíme nejprve vytvořit metodu se stejnou signaturou (návratovým typem a stejnými parametry) a delegáta pak instanciovat. Delegáti ale nejsou jen ekvivalentem ukazatelů na metody z jazyka C. Oproti nim mají jednu další vlastnost: kompozici (composition). Delegát nemusí být interně spojen pouze s jedinou metodou, ale s celou množinou metod. K instanci delegáta lze připojit dalšího delegáta a to pomocí operátoru += nebo +. Vrací-li delegát neprázdný návratový typ, pak je vrácena hodnota posledního delegáta. Delegáta lze odebrat pomocí -= nebo -.

Ukázka kódu

```
class SomeClass
{
  public delegate void DelegatsName(int a);
  public void Method(DelegatsName d)
  {
    d(5); // delegát je ukazatel na metodu, kterou můžeme normálně volat
  }
}

class AnotherClass
{
  public static void StaticRealMethod(int a)
  {
  }
  public void RealMethod(int a)
  {
  }
  static void Main(string[] args)
  {
    SomeClass sc = new SomeClass();
    //je vytvořena instance delegáta
    SomeClass.DelegatsName del=new SomeClass.DelegatsName(RealMethod)
    del += SomeClass.DelegatsName(StaticRealMethod)
    // delegát je předán jako argument při volání metody
    sc.Method(del);
  }
}
```

Anonymní metody

V kapitole si ukážeme jak lze efektivně využít **anonymních metod** ve spojení s návratový typem a seznamem parametrů delegáta. Tato vlastnost přibyla ve verzi jazyka 2.0.

Anonymní metody

V programu se občas může objevit kód, který je volán pouze prostřednictvím delegáta. Následný kód se umísťuje do pojmenovaných metod jako součást tříd nebo struktur, což se jeví jako zbytečné. Výhodnější je využít tzv. **anonymních metod**, které představují elegantnější řešení.

Anonymní metody umožňují psát kód delegátů přímo **"in-line"** . Využít je lze v místech kódu, kde jsou očekáváni delegáti, případně delegáti s argumenty. Vytváří se klíčovým slovem **delegate** . Připomínají lambda funkce, které známe z jazyků Python a Lisp. **Lisp** je jazyk pro funkcionální programování s výrazně regulární syntaxí. Syntaxe jazyka Lisp je nazývána prefixová, neboť operátor se nachází na začátku výrazu a až po něm následují operandy. Anonymní metody jsou založeny na implicitní konverzi na typ **delegate**. Kde překladač vyžaduje kompatibilní seznam parametrů a návratovou hodnotu.

Dvě hlavní vlastnosti anonymních metod:

První vlastností je možnost vypuštění seznamu parametrů při definici delegáta, pokud tento seznam

nevyužíváme. Avšak jde o rozdílnou definici než u prázdného seznamu typových parametrů, který je deklarován dvěmi prázdnými kulatými závorkami *'()'*.

Seznam typových parametrů je považován za kompatibilní se všemi delegáty kromě těch, kteří mají *out* parametry. Parametry definující typ delegáta je nutné vždy dodat, a to ve chvíli, kdy je delegát volán.

Druhá možnost je využití lokálních proměnných, které jsou definovány v těle metody, v níž je anonymní metoda umístěna. Z anonymní metody můžeme přistupovat k lokálním proměnným a parametrům. Takovým proměnným a parametrům říkáme **vnější (outer) proměnné** anonymních metod. Navíc při odkazování se na tyto vnější proměnné je nazýváme **zachycené** (captured). Životnost zachycených vnějších proměnných, případně parametrů, je následně prodloužena minimálně na takovou dobu, jaká je doba životnosti náležící anonymní metody.

Pravidla určující anonymní metodu

Seznam parametrů delegáta je kompatibilní s anonymní metodou pokud:

Anonymní metoda nedefinuje seznam parametrů a delegát nemá žádné *out* parametry.

Pro anonymní metodu definovanou společně se seznamem parametrů platí, že seznam parametrů musí přesně odpovídat parametrům delegáta. Seznam parametrů tedy musí mít stejný *počet, typ a modifikátory parametrů* jako parametry delegáta.

Návratový typ delegáta je kompatibilní s anonymní metodou pokud platí alespoň jedno z těchto tvrzení:

Návratový typ delegáta je **void** a anonymní metoda neobsahuje žádné příkazy *return* anebo obsahuje tento příkaz bez výrazu, takže poskytuje pouze *return;*.

Návratový typ delegáta není **void** a všechny výrazy v anonymní metodě vázané s příkazem *return* mohou být implicitně přetypovány na návratový typ delegáta.

Návratový typ a seznam parametrů delegáta musí být kompatibilní s anonymní metodou ještě předtím, než dojde k implicitní konverzi na delegátský typ delegáta.

Anonymní metody v příkladech

Následující příklad využívá anonymních funkcí napsaných jako **"in-line"**. Anonymní metody jsou zde vloženy jako parametry delegátského typu *Function*.

Nejprve deklarace delegáta vně příslušné třídy:

Ukázka kódu

```
delegate double Function(double x);
```

Dále si vytvoříme, již uvnitř vlastní třídy, statickou metodu *Apply()* s jedním parametrem pro pole čísel **double** a druhým parametrem bude náš vytvořený delegát. Tato metoda vrátí pole reálných čísel, jejichž přepočet je dán pozdějším předpisem anonymní funkce při volání metody.

Ukázka kódu

```
static double[] Apply(double[] a, Function f)
{
    double[] result = new double[a.Length];
    for (int i = 0; i < a.Length; i++)
    {
        result[i] = f(a[i]);
    }
    return result;
}
```

Volání metody *Apply()* v metodě *Main()* by vypadalo následovně:

Ukázka kódu

```
double[] a = {0.0, 0.5, 1.0};  //fill the field
double[] squares = Apply(a, delegate(double x) { return x * x; });
```

Jak můžete vidět, druhým parametrem v metodě *Apply()* je ukázka **"in-line"** psané anonymní metody, která je kompatibilní s delegátským typem *Function*. Tato anonymní metoda vrací mocninu jejího argumentu. Proto výsledek volání metody *Apply()* je typu **double[]** a obsahuje mocniny hodnot v poli *a*.

Vytvořením další metody *MultiplyAllBy()* můžeme rozšířit funkčnost stávajícího programu:

Ukázka kódu

```
static double[] MultiplyAllBy(double[] a, double factor)
{
    return Apply(a, delegate(double x) { return x * factor; });
}
```

Metoda *MultiplyAllBy()* vrací také pole čísel typu **double[]**, kde je každá z hodnot v poli daná argumentem *factor*. Tato metoda k výpočtu využívá volání předchozí anonymní metody *Apply()*. V metodě *Apply()* anonymní metoda násobí argumenty *x* hodnotou argumentu *factor*. Volání funkce *MultipleAllBy()*, která přenásobí prvky hodnotou 2 typu **double** by vypadalo takto:

Ukázka kódu

```
double[] a = {0.0, 0.5, 1.0};  //naplnění pole
double[] doubles = MultiplyAllBy(a, 2.0);
```

Konverze skupinových metod

V předchozí sekci jsme si říkali, že anonymní metoda může být implicitně přetypována na kompatibilní delegátský typ. C# 2.0 dovoluje stejný způsob konverze i pro **skupinu metod**. Dovoluje totiž explicitním konkretizacím delegátů, aby byly ve všech případech vynechány. Využitím předešlé metody *Apply()*

Ukázka kódu

```
static double[] Apply(double[] a, Function f)
{
    double[] result = new double[a.Length];
    for (int i = 0; i < a.Length; i++)
    {
        result[i] = f(a[i]);
    }
    return result;
}
```

mohou být tyto výrazy

Ukázka kódu

```
addButton.Click += new EventHandler(AddClick);
Apply(a, new Function(Math.Sin));
```

zapsány následovně s vynecháním zápisu instance události a delegáta

Ukázka kódu

```
addButton.Click += AddClick;
Apply(a, Math.Sin);
```

Při využití druhé kratší formy příkazu překladač sám odvodí konkretní delegátský typ. Efekt je samozřejmě u obou forem zápisu stejný. Jedná se o novou vlastnost, která je využívána i dalšího rysu jazyka C# 2.0, a to u **dedukce delegátů** .

Události

Zpracování událostí je v zásadě proces, kdy jeden objekt dokáže upozornit další objekty na to, že došlo k nějaké změně (události). Systém událostí v jazyce C# interně využívá delegátů. Na tyto delegáty jsou kladeny tyto podmínky:

delegát musí mít dva parametry a oba jsou objekty;

první udává kdo je zdrojem události;

druhý obsahuje informace spojené s konkrétní událostí. Tento objekt musí rozšiřovat třídu *EventArgs*.

Využití delegátu a událostí demonstruje následující příklad. Jde o implementaci jednoduchého příkladu:

Výrobce vytváří zboží. Tento proces je by ve skutečnosti mohl být asynchronní. Aplikace by pak měla více vláken...

V případě, že je zboží hotové, jsou upozorněni zaregistrovaní zákaznici.

Ukázka kódu

```
class InfoEventArgs : EventArgs
{
  private string info;
  public InfoEventArgs(string info)
  {
    this.info=info;
  }
  public string Info
  {
    get {return info;}
  }
}
class Producer
{
  string name;
  public Producer(string name)
  {
    this.name=name;
  }
  public string Name
  {
    get {return name;}
  }
  public  delegate void WantToKnow(Producer source,InfoEventArgs args);
  public event WantToKnow ItemProduced;
  public void Produce(string productName)
  {
    Console.WriteLine("Production of "+productName+" started.");
    InfoEventArgs info=new InfoEventArgs(productName);
    Console.WriteLine("Production of "+productName+" ended.");
    if (ItemProduced!=null) ItemProduced(this,info); //vyvolání události
  }
}
class Customer
{
```

```
  string name;
  public Customer(string name,Producer producer)
  {
    this.name=name;
    producer.ItemProduced+=new Producer.WantToKnow(NewItemProduced);//registrace
  }
  public void NewItemProduced(Producer producer,InfoEventArgs info) //skutečná
obsluha události
  {
    Console.WriteLine(this.name+": "+producer.Name+" produce item:"+info.Info);
  }
}
class RunApp
{
  public static void Main()
  {
    Producer producer=new Producer("Haven inc.");
    Customer marek=new Customer("Marek",producer);
    Customer tom=new Customer("Tom",producer);

    producer.Produce("Ferrari");
    producer.Produce("pencil");
    producer.Produce("cake");
  }
}
```

Události fungují jako jakýsi sběrný bod pro konkrétní delegáty. V případě, že někdo tuto událost vyvolá, jsou spuštění všichni zaregistrovaní delegáti.

Výstupem předcházejícího programu bude:

Ukázka kódu

```
Production of Ferrari started.
Production of Ferrari ended.
Marek: Haven inc. produce item:Ferrari
Tom: Haven inc. produce item:Ferrari
Production of pencil started.
Production of pencil ended.
Marek: Haven inc. produce item:pencil
Tom: Haven inc. produce item:pencil
Production of cake started.
Production of cake ended.
Marek: Haven inc. produce item:cake
Tom: Haven inc. produce item:cake
```

Generické datové typy

V této kapitole se budeme zabývat generickými datovými typy. Nejprve si uvedeme motivaci, proč používat generické datové typy a na jakých základech jsou postaveny. V další části pak bude ukázáno jak tyto konstrukce implementovat.

Teorie *generik*

Základní myšlenkou rozšíření jazyka C# 2.0 byla podpora elegantního a rychlého typově bezpečného kódu. Také zjednodušení některých základních úkolů programátora, které se velmi často opakovali. Tyto důvody, pro rozšíření jazyka, dále navrhnout a zrealizovat s velkým důrazem na zpětnou kompatibilitu s předchozí verzí jazyka (C# 1.0). Než pochopíme využití *generik* musíme navázat na předešlou problematiku psaní kódu.

Univerzální kód

Univerzální kód využívá programátor pro všestrannou práci nad libovolnými daty. Z toho důvodu knihovny často využívají základní a hierarchicky nejvyšší typ **object**, který umožňuje uchovávat libovolná data. Tudíž jak hodnotového, tak i referenčního typu. Lze jej také využít pro univerzální položky, parametry a návratové hodnoty funkcí. S použitím univerzálního kódu však vzniká řada základních problémů:

Nejzásadnějším problémem je chybějící typová kontrola při překladu, kdy nemůžeme zjistit, zda přiřazení konkrétního typu do typu **object** je správné, anebo zdali není tak závadné, aby způsobilo během provádění aplikace chybu. Tuto chybu lze způsobit použitím nevhodných nebo nekompatibilních typů.

Další problém nastává u použití hodnotových typů, a to operací pro zaobalení hodnot do objektové reprezentace (boxing) a následující rozbalení zpět na jejich konkrétní hodnotu (unboxing). Během těchto operací se musí neustále provádět typová kontrola za běhu aplikace, což má za důsledek časové a výkonnostní zpomalení aplikace.

Také použití referenčních typů není bezproblémové. Nedochází u nich k předešlým zbytečným operacím, ale při převodu zpět z proměnné typu **object** na konkrétní typ (unboxing), musí dojít k explicitně zadané typové konverzi. Zde právě může dojít k chybě, pokud konverze neodpovídá uloženému typu.

Poslední problém může nastat v nepřehlednosti, kdy častým využitím univerzálního kódu získáme obtěžující a nevzhledné typové konverze.

Popis *generik*

Mluvíme-li o generickém kódu, máme namysli kód, který využívá **parametrizované typy**. Parametrizované typy jsou nahrazeny konkrétními typy v době použití kódu. Generický typ T se uvádí za název definovaného objektu (třídy, rozhraní, metody, atd.) a je ohraničen špičatými závorkami **<T>**. Princip generik je znám také z jiných jazyků jako Eiffel, Ada (Java) a především z jazyku C++. Generika mají podobnou syntaxi jako šablony v programovacím jazyku C++, avšak realizují hodně odlišnou implementaci. V jazyce C# 2.0 lze pomocí generik definovat *třídy, struktury, rozhraní, metody a delegáty*. Použitím generik získáme několik výhod:

V prvé řadě dosáhneme silné typové kontroly v době překladu zdrojového kódu, a tedy možného nalezení všech případů, které by mohly způsobit havárii aplikace způsobenou nesprávným využitím použitých typů.

Odstraníme také časté a z časového hlediska zbytečné operace *boxing a unboxing*.

Omezení explicitních typových konverzí, které jsou nutné v době překladu, kdy neznáme typ objektu uloženého v univerzální kolekci.

Díky těmto výhodám dosáhneme nejen čistějšího, rychlejšího výsledného kódu, ale především náš kód bude hlavně bezpečnější.

Omezení

C# 2.0 dále poskytuje tzv. **omezení**, které můžeme aplikovat na zmíněná generika. Specifikace požadavků na typové parametry:

Zabezpečují, že typy poskytují požadovanou funkcionalitu.

Umožňují silnější typové kontroly při překladu.

Omezují potřeby explicitních typových konverzí.

Poskytují možnost omezení použitelnosti výsledného generika.

V případě psaní obecného kódu nad obecným typem předpokládáme jeho jistou funkcionalitu. U konkrétního typu budeme proto požadovat také jeho určitou funkcionalitu. Pro zajištění této funkcionality existují tři typy omezení:

Omezení třídy (Class constraint)

Konkrétní typ, který v generickém kódu použijeme, musí být odvozen od specifické základní třídy. Tím zajistíme, že máme definovanou jistou funkcionalitu.

Omezení rozhraní (Interface constraint)

Zajišťuje, že rozhraní, které použijeme, je implementováno od konkrétního specifického rozhraní.

Omezení konstruktoru (Constructor constraint)

Požadavek, aby třída nebo struktura, kterou používáme, obsahovala veřejný implicitní konstruktor.

Generika ve srovnání s kolekcemi

Kolekce jsou nevýhodné z časových důvodů a možného výskytu programových chyb

O základních prvcích kolekcí typů **object** víme:

jsou dynamicky napsané => programové chyby jsou odhaleny pouze až v době běhu aplikace

jsou potřebná přetypování za běhu programu => což ovšem má za důsledek snížení celkové rychlosti programu

hodnoty jednoduchých datových typů (např. int) musí být zabalené (jako Integer) => to zabere volné místo a potřebný čas

Příklad deklarace kolekce typu **Map**:

Ukázka kódu

```
Map  map = new HashMap();
```

Z takto nadefinované proměnné nejsme schopni určit s jakými typy bude kolekce pracovat. Možným řešením pro dokumenty s netriviálním využitím kolekcí je vhodné vložení komentáře k deklaraci funkce příslušné kolekce, jak je uvedeno níže. Komentáře jsou samozřejmě ignorovány překladačem.

Ukázka kódu

```
Map /* from Integer to Map from String to Integer */ map = new HashMap();
```

Generiky mohou udělat výsledný kód **typově bezpečným**. S generickými kolekcemi využívajícími **parametrický polymorfismus** (neboli generika), lze předešlou deklaraci zapsat takto:

Ukázka kódu

```
IMap<int,IMap<string,int>> map = new HashMap<int,IMap<string,int>>();
```

Takto nadefinovanou proměnnou získáme nejen typové bezpečí, ale také lépe pochopíme smyl použití dané proměnné.

Kolekce je nutné přetypovávat

Společnost Microsoft přidala k verzi frameworku .NET 2.0 nový prostor jmen pro kolekce nacházející se v *System.Collections*. Tento prostor jmen obsahuje několik odlišných rozhraní a tříd, které definují různé typy pomocných kontejnerových seznamů, slovníků a hašovacích tabulek. Každý zmíněný typ má jinou implementaci a slouží ke konkrétnímu účelu. Každá položka musí být přetypována během procesu zpřístupnění z kolekce. Toto povinné přetypování může způsobit chybu, která by vyplývala z dalšího jiného typu objektu, který byl někdy předtím vložen do kolekce.

Pomocí vytvoření nové kolekce zajistíme typovou bezpečnost, avšak kolekce není znovu použitelná pro další typy. Navíc pokud máme více specifických tříd, pak se údržba na nich stává tak těžkopádnou, že ani nemá cenu ji realizovat za účelem poskytnutí typového bezpečí. Řešením jsou tedy **generika**.

Generika byli vytvořeny proto, aby nabízely spojení typového bezpečí, výkonu a všeobecných zásad v definovaném typu. Generika je tudíž typově bezpečná třída, která je deklarovaná bez konkrétního typu a aplikovaná v definici. Spíše lze říci, že typ je konkretizovaný až v době, kdy je objekt užívaný.

System.Collections.Generics je nový prostor jmen (namespace) určený pro generické kolekce, který

obsahuje několik předběžně sestavených tříd, které jsou navrhnuty pro reprezentaci běžných typů, včetně těchto: *Zřetězený seznam, Seznam, Fronta, Zásobník*

Příklad ke stažení

Ukázka využití existujícího generického vzorového kódu, namísto vytváření vlastního generického kódu zde [examples/generika/pr_1.cs].

Generika v příkladech

Proč potřebujeme *generika*

Prozatímní účel datových struktur je ve využití typu **object** k uložení dat různého typu. Například, zde vidíme objektově založenou třídu *Stack* reprezentující zásobník:

Ukázka kódu

```
public class Stack
{
    object[] items;
    int count;
    public void Push(object item) {...}
    public object Pop() {...}
}
```

Využitím typu **object** je tato třída velmi flexibilní, avšak má i stinné stránky. Například, pokud vložíme na zásobník hodnotu nějakého hodnotového typu metodou *Push*, třeba typu **int**, dojde k automatickému zabalení hodnoty do objektové reprezentace (boxingu). Při pozdějším použití metody *Pop*, pro získání této hodnoty ze zásobníku, musí dojít k explicitnímu přetypování do správného konkrétního typu (unboxing).

Ukázka kódu

```
Stack stack = new Stack();
stack.Push(3);
int i = (int)stack.Pop();
```

Explicitní přetypování je pro programátora únavné a společně s operací zabalení (boxing) přidává výkonovou režii aplikace, neboť potřebují dynamickou alokaci paměti a typové kontroly za běhu aplikace. Také je nevýhodou, že nelze zjistit jaký je typ objektu vloženého do zásobníku. Dále pak může dojít k tomu, že objekt uložený v zásobníku může být po vyjmutí explicitně přetypován do jiného typu. Jako zde na ukázce:

Ukázka kódu

```
Stack stack = new Stack();
stack.Push(new Customer()); // class for customer
string s = (string)stack.Pop(); // bad explicit cast, but not error
```

Zatímco výše uvedený kód je založen na nevhodně použité třídě *Stack*, je kód po technické stránce správný a proto není oznámena chyba během provádění kompilace. Problém není zřejmý do té doby, než je kód spuštěný. Poté se už chyba projeví vyvoláním výjimky pro špatné přetypování *InvalidCastException*.

Konstrukce a využití *generik*

Generika dovedou vytvořeným typům přiřadit **typové parametry** . Typový parametr je specifikován v lomených závorkách za jménem definovaného objektu (<T>). Genericky vytvořené objekty přijímají pouze typ, pro který byly vytvořeny a ukládají data tohoto typu bez zbytečných konverzí, které využívají objektově založené struktury. Všimněte si rozdílů v implementaci předešlého zásobníku a generického zásobníku s

typovým parametrem, jehož implementace je zde naznačena:

Ukázka kódu

```
public class Stack<T>
{
    T[] items;
    int count;
    public void Push(T item) {...}
    public T Pop() {...}
}
```

Při použití generické třídy *Stack<T>* se aktuální typ nahradí za již specifikovaný typ T. V následující ukázce je typ **int** předán jako **typový argument** pro T:

Ukázka kódu

```
Stack<int> stack = new Stack<int>();
stack.Push(3);
int x = stack.Pop();
```

Typ *Stack<T>* je nazýván **zkonstruovaným typem** (constructed type). V objektu typu *Stack<int>* je každý výskyt typového parametru T nahrazen typem v argumentu, zde tedy typem **int**. Když je vytvořena instance objektu generického typu *Stack<int>*, její položky jsou uloženy jako pole typu **int[]**, což je rozhodně lepší než, kdyby byly tyto položky uloženy v typu **object[]**. Důvodem je značné využití paměti u negenerického zásobníku *Stack*.

Generika poskytují přísnou kontrolu typů, což například znamená, že je chybou vložení proměnné typu **int** do generického zásobníku typu *Customer*. *Stack<Customer>* je omezený pouze na objekty typu *Customer*, proto překladač zahlásí chybu v následujícím příkladu u posledních dvou řádků:

Ukázka kódu

```
Stack<Customer> stack = new Stack<Customer>();
stack.Push(new Customer());
Customer c = stack.Pop();
stack.Push(3);              // Type mismatch error
int x = stack.Pop();  // Type mismatch error
```

Generické typy

Generické typy mohou mít více než jeden parametrický typ, jak tomu bylo v předchozím příkladu. Na dalším příkladu je možné vidět využití generické třídy *Dictionary*, která obsahuje jeden typový parametr pro klíče a druhý pro typ vkládaných hodnot slovníku:

Ukázka kódu

```
public class Dictionary<K, V>
{
    public void Add(K key, V value) {...}
    public V this[K key] {...}
}
```

V případě, že použijeme instanci objektu *Dictionary<K, V>*, pak mu musíme dodat příslušné argumenty:

Ukázka kódu

```
Dictionary<string,Customer> dict = new Dictionary<string,Customer>();
dict.Add("Peter", new Customer());
Customer c = dict["Peter"];
```

Generické typy programátorům dovolí vytvořit a testovat kód pouze jednou a znovu jej použít pro jakékoliv typy. Navíc v případě hodnotových typů je použití generického uložení dat mnohem výkonnější, protože se vyhneme operacím boxing, unboxingu a také přetypování.

Budeme chtít srovnat rychlost provádění genericky vytvořené datové struktury s objektově založenou datovou strukturou v závislosti na použitém typu.

Zjistíme, že generika jsou na tom podstatně lépe, samozřejmě při implementaci stejného problému. Jak blíže určuje dolní tabulka.

Tabulka 4.4. Tabulka porovnání rychlostí generických objektů s negenerickými

Typ objektu	Čas provedení kódu [s]		Časový rozdíl
	negenerického	generického	[s]
Seznam typu **string**	0,562	0,438	0,125
Seznam typu **int**	0,984	0,343	0,641
Zásobník typu **string**	0,406	0,406	0,000
Zásobník typu **int**	0,781	0,265	0,516

Tabulka poskytuje srovnání generického a negenerického objektu, který je vždy naplněn 10 000 prvky a každý prvek je přiřazen do proměnné příslušného typu. V tabulce můžeme vidět snížení času provedení kódu při využití generického objektu. Tabulka dále ukazuje na větší časový rozdíl při práci s typem **int**. Dokonce v případě typu **string** vkládaného do zásobníku je čas u jeho negenerické i generické implementace stejný.

Generické typy a IL

Podobně jako u negenerického typu je zkompilovaná reprezentace generického typu složená z instrukcí IL a metadat. Samozřejmě také zakóduje existenci a použití typových parametrů.

Nejprve dojde k vytvoření instance zkonstruovaného generického typu jako *Stack<int>*. Následně pak překladač JIT (just-in-time) běhového prostředí .NET CLR přemění generický IL a metadata do *nativního kódu*. Mezitím dochází také k dosazení aktuálních typů za typové parametry. Pozdější odkazy na zkonstruovaný generický typ pak využívají stejný nativní kód. Proces vytvoření konkrétního zkonstruovaného typu z generického typu se nazývá **konkretizace generického typu**.

Běhové prostředí **CLR (Common Language Runtime)** platformy .NET vytváří speciální kopii nativního kódu pro každou konkretizaci generického typu s hodnotovým typem. Avšak pro všechny odkazové typy sdílí unikátní kopii nativního kódu. Je tomu tak, protože v nativní úrovni kódu jsou odkazy (reference) jen ukazateli stejné reprezentace.

Důvody a ukázky *omezení*

Generická třída toho obvykle udělá více než jen ukládání dat založených na typových parametrech. Často budeme chtít vyvolat metody na objektech, jejichž typ je dán typovým parametrem. Blíže na ukázce příkladu, kdy metoda *Add* v generickém slovníku třídy *Dictionary<K,V>* bude potřebovat porovnat užívané klíče K metodou *CompareTo*:

Ukázka kódu

```
public class Dictionary<K, V>
{
```

```
    public void Add(K key, V value)
    {
        ...
        if (key.CompareTo(x) < 0) {...}    // Error, no CompareTo method
        ...
    }
}
```

U metody *CompareTo* ovšem nastane chyba během provádění kompilace, neboť typový argument K je předepsán pro jakýkoliv typ. Proto jediní členové, kteří můžou být součástí parametru **key**, jsou deklarovány typem **object**. Tak jako metody *Equals, GetHashCode a ToString*. Řešením je přetypování parametru **key** na takový typ, který obsahuje metodu *CompareTo*, například *IComparable*:

Ukázka kódu

```
public class Dictionary<K, V>
{
    public void Add(K key, V value)
    {
        ...
        if (((IComparable)key).CompareTo(x) < 0) {...}
        ...
    }
}
```

Toto řešení je už funkční, avšak vyžaduje dynamickou typovou kontrolu za běhu programu, což přidává na režii. Dále může vyvolat chybové hlášení *InvalidCastExeption*, pokud typový parametr **key** neimplementuje *IComparable*.

Z důvodů silnější typové kontroly za běhu aplikace a snížení přetypování, nabízí jazyk C# jakýsi nepovinný seznam **omezení** dodán ke každému typovému parametru. Dále určuje, že typ má být používaný jako argument pro typový parametr. Omezení jsou deklarována slovem **where** následované typovým parametrem, za ním dvojtečkou, pak čárkami odděleným seznamem třídních typů, typů rozhraních a typových parametrů (nebo také speciálním odkazovým typem, hodnotovým typem, ale i omezeným konstruktorem).

Aby třída *Dictionary<K,V>* zabezpečila to, že klíče budou vždy implementovány jako *IComparable*, musí deklarace třídy obsahovat **omezení pro typový parametr**:

Ukázka kódu

```
public class Dictionary<K, V> where K: IComparable
{
    public void Add(K key, V value)
    {
        ...
        if (key.CompareTo(x) < 0) {...}
        ...
    }
}
```

Touto deklarací zajistíme, že překladač doplní za typový argument K pouze takový typ, který implementuje rozhraní *IComparable*. Nemusíme tudíž ani explicitně přetypovávat, neb všichni členové typu, daného omezením typového parametru, jsou přímo dosažitelní z objektu.

Omezení typovým parametrem

Pro daný typový parametr je možné specifikovat jako omezení několik rozhraní a typových parametrů, ale pouze jednu třídu. Každý omezený typový parametr má oddělovač **where**. V dolním příkladu má typový parametr *K* dvě omezení rozhraní, zatímco typový parametr *E* má omezení třídním typem a konstruktorem:

Ukázka kódu

```
public class EntityTable<K, E>
    where K: IComparable<K>, IPersistable
    where E: Entity, new()
{
    public void Add(K key, E entity)
    {
        ...
        if (key.CompareTo(x) < 0) {...}
        ...
    }
}
```

Omezení konstruktorem *new()* zajistí, že typ použitý jako typový argument pro *E* má veřejný bezparametrový konstruktor. Tím povolí generické třídě použití *new E()* k vytvoření instance jejího typu.

Omezení typových parametrů je nutné používat obezřetně, neboť také můžeme omezit možné vhodné využití generického typu.

Generické metody

V některých případech není typový parametr potřebný pro celou třídu, ale pouze uvnitř konkrétní metody. Příkladem je metoda, která vezme generický typ jako parametr. Například, budeme chtít vložit několik hodnot v řádku jedním zavoláním metody s využitím dříve popsané generické třídy *Stack<T>*. Metoda pro specificky zkonstruovaný typ by vypadala takto:

Ukázka kódu

```
void PushMultiple(Stack<int> stack, params int[] values) {
    foreach (int value in values) stack.Push(value);
}
```

Vícenásobné vložení dat

Tuto metodu můžeme použít k vícenásobnému vložení hodnot typu **int** do *Stack<int>*:

Ukázka kódu

```
Stack<int> stack = new Stack<int>();
PushMultiple(stack, 1, 2, 3, 4);
```

Metoda ovšem pracuje pouze pro konkrétní zkonstruovaný typ *Stack<int>*. Pro práci se všemi typy musí být napsána jako **generická metoda**.

Generická metoda má jeden nebo více typových parametrů zapsaných v hranatých závorkách < a > za jménem metody. Typové parametry mohou být použity uvnitř seznamu parametrů jako návratový typ a v těle metody. Generická metoda *PushMultiple()* by poté vypadala:

Ukázka kódu

```
void PushMultiple<T>(Stack<T> stack, params T[] values) {
     foreach (T value in values) stack.Push(value);
}
```

Při následném volání metody jsou typové argumenty zadány v hranatých závorkách v místě, kde dochází k vyvolání metody:

Ukázka kódu

```
Stack<int> stack = new Stack<int>();
PushMultiple<int>(stack, 1, 2, 3, 4);
```

V mnoha případech může překladač odvodit správný typový parametr z argumentů příslušné metody. Tento proces se nazývá **typové odvozování** (type inferencing). Z výše uvedeného příkladu, přesněji z prvního argumentu typu *Stack<int>* a následujících argumentů metody typu **int**, může překladač rozpoznat typový parametr. Ten musí tedy být typu **int**. Proto může být generická metoda *PushMultiple()* volána bez deklarujícího typového parametru:

Ukázka kódu

```
Stack<int> stack = new Stack<int>();
PushMultiple(stack, 1, 2, 3, 4);
```

Volání generických metod

Volání generické metody může explicitně určit programátor seznamem typových argumentů. Nebo může seznam typových argumentů při volání metody opomenout, nevypsat a spolehnout se na **typové odvozování**, které zajistí určení správných typových argumentů. V následujícím příkladu si ukážeme, jak dojde k rozhodnutí přetížení po typovém odvozování a po tom, co jsou typové argumenty nahrazeny v parametrovém seznamu:

Ukázka kódu

```
class Test
{
   static void F<T>(int x, T y) {
       Console.WriteLine("one");
   }
   static void F<T>(T x, long y) {
       Console.WriteLine("two");
   }
   static void Main() {
       F<int>(5, 324);            // Ok, prints "one"
       F<byte>(5, 324);           // Ok, prints "two"
       F(5, 324);                       // Ok, prints "one"
       F<double>(5, 324);         // Error, ambiguous
       F(5, 324L);                      // Error, ambiguous
   }
}
```

U prvních tří volání generických metod ve funkci *Main()* nedojde k chybě, neboť se u prvních dvou funkcí explicitně určí seznam typových argumentů. U třetí metody je určen typový argument pomocí typového odvozování. Avšak u posledních dvou generických metod dojde k chybě způsobené nejednoznačným výběrem generické metody podle typových argumentů.

Signatura generických metod

Omezení jsou u signatur generických metod ignorována. Významný je počet generických typových parametrů jako i seřazení pozic typových parametrů. Následné srovnání signatur jednotlivých metod v příkladu ukáže více:

Ukázka kódu

```
class A {}
class B {}

interface IX
{
```

```
    T F1<T>(T[] a, int i);              // Error
    void F1<U>(U[] a, int i);            // Error

    void F2<T>(int x);                   // Ok
    void F2(int x);                           // Ok

    void F3<T>(T t) where T: A;          // Error
    void F3<T>(T t) where T: B;          // Error
}
```

U obou funkcí *F1* dojde k chybě, neboť obě deklarace mají stejnou signaturu. A také z důvodu, že návratový typ a jméno typového parametru není u druhé funkce *F1* stejný. U funkcí *F2* je vše v pořádku, neboť počet typových parametrů je součástí signatury. Chyba u funkcí *F3* je ukázkou výše zmíněného pravidla, že omezení jsou v signaturách ignorována.

Generická třída

Deklarace generické třídy je stejná jako deklarace třídy, akorát vyžaduje seznam typových parametrů k vytvoření skutečných typů. Deklarace třídy může libovolně definovat typové parametry:

Ukázka kódu

Deklarace třídy:

atributy modifikátory **class** *identifikátor <seznam typových parametrů> bázi_třídy omezení_typových_parametrů* **tělo_třídy** *;*

Všimněte si v předpisu částí deklarace generické třídy napsaných *kurzívou*. Tyto části nemusí být definovány při deklaraci, neboť jsou *nepovinné*.**Povinné** části deklarace třídy jsou zvýrazněny **tučně**.

Tento předpis je podobný i u ostatních generických struktur. Základní rozdíl je v použití klíčového slova, které specifikuje vytváření příslušného objektu (jako interface, struct, atd.).

Deklarace třídy nemůže poskytovat *omezení_typových_parametrů*, pokud rovněž nedodává *seznam_typových_parametrů*. Deklarace generické třídy navíc mohou být vložené uvnitř deklarace negenerické třídy.

Zkonstruovaný typ

Generická třída je odkázaná na používání zkonstruovaných typů. Deklarace generické třídy:

Ukázka kódu

```
class List<T> {}
```

Příklady zkonstruovaných typů mohou být *List<T>, List<int> a List<List<string>>*. Zkonstruovaný typ, který využívá jeden nebo více typových parametrů, jako *List<T>*, se nazývá **otevřený zkonstruovaný typ**. Naopak pokud nevyužívá typové parametry, jako *List<int>*, tak se nazývá **uzavřený zkonstruovaný typ**.

Generické typy mohou být **přetěžovány** v závislosti na počtu typových parametrů. Například, pokud jsou dvě deklarace ve stejném jmenném prostoru nebo vnější deklaraci, mohou používat stejný identifikátor v deklaraci tak dlouho, dokud mají různý počet typových parametrů.

Ukázka kódu

```
class C {}
class C<V> {}
struct C<U,V> {}   // Error, C with two type parameters defined twice
class C<A,B> {}     // Error, C with two type parameters defined twice
```

U posledních dvou deklarací nastane chyba porušením výše zmíněné podmínky o počtu typových parametrů.

Typové parametry

Typové parametry mohou být dodávány do deklarace třídy. Každý typový parametr je identifikátor sloužící k udržení místa pro později použitý **typový argument**. Při vytvoření zkonstruovaného typu se za typový parametr dosadí aktuální typ, představovaný typovým argumentem.

Typový parametr nemůže mít stejné jméno jako jiný typový parametr obsažený v té samé deklaraci anebo jméno člena deklarovaného ve třídě. Nemůže mít také stejné jméno jako je jméno jeho typu.

Rozsah typového parametru určuje *základní třída, omezení typových parametrů a tělo třídy*. Rozsah se nerozšiřuje k odvozeným třídám narozdíl od členů třídy. Uvnitř rozsahu může být typový parametr užíván jako: **hodnotový typ, referenční typ, typový parametr**.

Typové parametry mohou být konkretizovány mnoha různými aktuálními typovými argumenty, avšak mají lehce odlišné operace a omezení než ostatní typy. Typové parametry jsou navíc záležitostí pouze kompilační úrovně. Během provádění aplikace jsou nahrazeny typem, jenž je specifikován typovým argumentem.

Vložené typy

Deklarace generické třídy může obsahovat také deklarace **vložených typů** (nested types). Uvnitř vloženého typu mohou být použity typové parametry z příslušné třídy. Deklarace vloženého typu může obsahovat další typové parametry, které platí pouze vzhledem k vloženému typu.

Pro každou deklaraci uvnitř deklarace generického typu platí, že je implicitně deklarací generického typu. Pokud se budeme odkazovat na typ vložený uvnitř generického typu, pak současně zkonstruovaný typ musí být pojmenovaný, včetně jeho typových argumentů. **Vložený typ** může být použit bez kvalifikace, tedy bez pojmenování. Instance typu vnější třídy může být implicitně použita při sestavení vloženého typu. V příkladu níže si ukážeme tři různé, avšak správné, způsoby jak se odkázat na zkonstruovaný typ vytvořený z vnitřní třídy *Inner*:

Ukázka kódu

```
class Outer<T>
{
    class Inner<U>
    {
        public static void F(T t, U u) {...}
    }
    static void F(T t) {
        Outer<T>.Inner<string>.F(t, "abc");      // These two statements have
        Inner<string>.F(t, "abc");                   // the same effect
        Outer<int>.Inner<string>.F(3, "abc");    // This type is different
        Outer.Inner<string>.F(t, "abc");             // Error, Outer needs type
arg
    }
}
```

První dvě inicializace jsou ekvivalentní, třetí je odlišná. V posledním čtvrtém příkladě nastane chyba, neboť třída *Outer* potřebuje typový argument. Ačkoliv to není zrovna správný programovací styl, typový parametr ve vloženém typu může ukrýt člena typového parametru deklarovaného ve vnějším typu:

Ukázka kódu

```
class Outer<T>
{
    class Inner<T>     // Valid, hides Outer's T
    {
        public T t;     // Refers to Inner's T
    }
}
```

Statická pole

U deklarace generické třídy je statická proměnná sdílena mezi všemi instancemi **stejného uzavřeného zkonstruovaného typu**. Ovšem mezi různými instancemi uzavřeného zkonstruovaného typu není statická proměnná sdílena. Zmíněné pravidlo platí bez ohledu na to, zda-li typ statické proměnné zahrnuje či nezahrnuje typové parametry.

Příklad deklarace generické třídy využívající statické pole:

Ukázka kódu

```
class C<V>
{
    static int count = 0;
    public C() {
        count++;
    }
    public static int Count {
        get { return count; }
    }
}
```

Příklad ke stažení

Celý příklad lze nalézt zde [examples/generika/pr_2.cs]

Nový **uzavřený zkonstruovaný třídní typ** je inicializovaný poprvé, když buď:

je vytvořena instance uzavřeného zkonstruovaného typu

jsou zmínění někteří ze statických členů uzavřeného zkonstruovaného typu

Statický konstruktor je vykonán přesně jednou pro každý uzavřený zkonstruovaný třídní typ. To je vhodné místo k vynucení si ověření typového parametru za běhu programu, neboť nemůže být kontrolován během kompilace skrze omezení. Například, následující typ užívá statického konstruktoru k vynucení si toho, že typový argument je typu enum:

Ukázka kódu

```
class Gen<T> where T: struct
{
    static Gen() {
        if (!typeof(T).IsEnum) {
            throw new ArgumentException("T must be an enum");
        }
    }
}
```

Statický konstruktor

Generická třída využívá statický konstruktor k inicializaci dříve zmíněného statického pole. Dále jej můžeme využívat k dalším inicializacím pro jakýkoliv **různý uzavřený zkonstruovaný typ**, který je vytvořen z této deklarace generické třídy. Typové parametry deklarace generického typu mohou být používány uvnitř těla statického konstruktoru.

Přetěžování

Uvnitř deklarace generické třídy můžeme přetěžovat:

metody

konstruktory

indexery

operátory

Deklarované signatury musí být jedinečné. Není ovšem vyloučeno, že náhrada typových argumentů může mít za výsledek totožné signatury. V takových případech budou pravidla o rozlišení přetížení vybrána pro nejvíce specifického člena. Následující příklad ukáže přetížení, která jsou platná či neplatná dle tohoto pravidla:

Ukázka kódu

```
interface I1<T> {...}
interface I2<T> {...}

class G1<U>
{
    int F1(U u);           // Overload resulotion for G<int>.F1
    int F1(int i);         // will pick non-generic
    void F2(I1<U> a);  // Valid overload
    void F2(I2<U> a);
}

class G2<U, V>
{
    void F3(U u, V v);             // Valid, but overload resolution for
    void F3(V v, U u);            // G2<int,int>.F3 will fail
    void F4(U u, I1<V> v);   // Valid, but overload resolution for
    void F4(I1<V> v, U u);   // G2<I1<int>,int>.F4 will fail
    void F5(U u1, I1<V> v2); // Valid overload
    void F5(V v1, U u2);
    void F6(ref U u);              // valid overload
    void F6(out V v);
}
```

Implicitní hodnoty

Implicitní hodnoty využívají klíčového slova **default** . Vrací implicitní hodnotu konkrétního typového parametru.

null pro odkazové typy

0 pro číselné typy

false pro booleovské typy

'\0' znakové typy

strukturu inicializovanou implicitní hodnotou

Ukázka použití klíčového slova **default** pro typové parametry:

Ukázka kódu

```
public class C<T>
{
    private T value;
    public T M() {
        return (condition) ? value : default(T);
    }
}
```

Metoda *M()* na základě vyhodnocení regulárního výrazu vrací, buď proměnnou *value* anebo implicitní hodnotu typového parametru přes výraz *default(T)*.

Využití implicitních hodnot není omezeno jenom na generika. Lze je využít v jakémkoliv kódu. Implicitní hodnoty patří k dalším rozšířením jazyka C# 2.0

Výhody *generik*

program se stává staticky napsaným => takže chyby jsou odhaleny v době provádění kompilace a nikoliv přímo před uživatelem za běhu aplikace

běhová přetypování nejsou potřebná => čímž je program rychlejší

hodnoty jednoduchých datových typů (jako např. int) nemusí být zabalené => program je rychlejší a zabírá méně místa

Negenerické objekty

Vlastnosti, události, indexery, operátory, konstruktory a destruktory nemohou sami mít typové parametry. Mohou se ovšem vyskytovat v generických typech a využívat typového parametru. Ukázka kódu z již dříve probraného příkladu, kde se vyskytuje použití indexeru, který využívá předaný typový parametr. Na příkladu vidíme využití generické třídy *Dictionary* pro práci se slovníkem, která obsahuje jeden typový parametr pro klíče a druhý pro typ vkládaných hodnot slovníku:

Ukázka kódu

```
public class Dictionary<K, V>
{
    public void Add(K key, V value) {...}
    public V this[K key] {...}
}
```

Atributy a práce s metadaty

Platforma .NET definuje možnost asociovat libovolné informace (metadata) se zdrojovým kódem aplikace. Tyto metadata jsou reprezentovaná atribut. Po kompilaci se atributy stanou součástí assembly. V programu jsem schopni tyto metadata získat pomocí mechanismu reflexe.

Příklad: pokud chceme, aby nějaká třída (její instance) byla serializována (uložena na diskový prostor), vložíme před ni atribut **Serializable**:

Ukázka kódu

```
[Serializable]
class MyClass() ...
```

Vytvoříme-li projekt pomocí Visual Studia.NET, vytvoří tento nástroj soubor AssemblyInfo.cs. Tento soubor obsahuje atributy vztahující se k celé assembly. Vygenerovaný soubor vypadá takto:

Ukázka kódu

```
using System.Reflection;
using System.Runtime.CompilerServices;

[assembly: AssemblyTitle("")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("")]
[assembly: AssemblyProduct("")]
[assembly: AssemblyCopyright("")]
```

```
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]
[assembly: AssemblyVersion("1.0.*")]
[assembly: AssemblyDelaySign(false)]
[assembly: AssemblyKeyFile("")]
[assembly: AssemblyKeyName("")]
```

Dalším používaným atributem je atribut Obsolete. Tento atribut signalizuje, že daná metoda je již zastaralá a nemá být používaná.

```
[Obsolete(message,bool)
```

message - vysvětlující text, který bude vypisovat překladač.

bool je true - použití metody je chyba při překladu.

bool je false - použití metody způsobí warning při překladu.

Představené příklady ukazují některé předdefinované atributy. Platforma .NET jich celou řadu. Interně jsou atributy realizovány pomocí tříd. Třída reprezentující atribut musí rozšiřovat System.Attribute. Tato třída je základní pro všechny atributy. V .NET Framework nejsou kladeny žádné omezení na množinu atributů. Uživatel může tuto množinu libovolně rozšiřovat. Atribut se skládá z parametrů. Existují dva typy parametrů.

Poziční parametry se zadávají jako formální parametry konstruktoru a jsou tudíž vždy povinné.

Pojmenované parametry se zadávají pomocí jména a operátoru přiřazení.

Atributy mohou také mít určité vlastnsoti. Tyto vlastnosti můžeme nastavit pomocí specifických atributů, které přidáme k definici našeho nového atributu.

```
[AttributeUsage(destination, AllowMultiple=bool, Inherited=bool)]
```

destination - typ cíle, pro který je atribut použitelný.

Lze povolit násobnou aplikaci.

Povolit dědičnost atributu.

Cíl atributu je dán výčtovým typem *AttributeTargets*.

Assembly, Class, Constructor, Delegate, Enum, Event, Field, Interface, Method, Parameter, Property, RetrurnValue, Struct, All.

Cíle lze kombinovat pomocí logického operátoru OR.

Následující příklad demonstruje použití atributu. Nejprve definujeme atribut MyAttribute. Tento atribut má poziční parametr Message a pojmenovaný paramet Number. Atribut je použitelný pouze u tříd a metod. Může být použit násobně a nelze jej dále rozšiřovat. Tyto vlastnosti jsem atributu dodali pomocí dalších specifických atributů. Použití je demonstrováno na jednoduché třídě. V metodě Main pak demonstrujeme získání zapsaných informací pomocí mechanismu reflexe.

Ukázka kódu

```
using System;
using System.Reflection;

[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method,
AllowMultiple=true, Inherited=false)]
class MyAttribute : Attribute
{
  private string message;
  public MyAttribute(string message)
  {
    this.message=message;
  }
  int number;
```

```
  public int Number
  {
    get {return number;}
    set {number=value;}
  }
  public string Message
  {
    get {return this.message;}
  }
}
[MyAttribute("Simple class",Number=1)]
[MyAttribute("but very nice!")]
class Example
{
  [MyAttribute("Stupid method",Number=10)]
  public void Test() {}
}

class RunApp
{
  static void Main(string[] args)
  {
    Type example=typeof(Example);
    foreach(MyAttribute myAttribute in example.GetCustomAttributes(false))
      Console.WriteLine("Class:"+myAttribute.Message+"("+myAttribute.Number+")");

    foreach(MemberInfo member in example.GetMembers())
    foreach(MyAttribute myAttribute in member.GetCustomAttributes(false))

Console.WriteLine("Member:"+myAttribute.Message+"("+myAttribute.Number+")");
  }
}
```

Výstup programu bude:

Ukázka kódu

```
Class:but very nice!(0)
Class:Simple class(1)
Member:Stupid method(10)
```

Spolupráce s existujícím kódem

Autoři platformy .NET a jazyka C# umožnili programátorům použít stávající programy. Používáme-li takovéto programy, zříkáme se výhod, které poskytuje CLR. Hovoříme pak o neřízeném kódu. Existují tři hlavní oblasti použití.

Spolupráce s komponentami modelu COM - schopnost prostředí .NET používat komponenty modelu COM a naopak komponentám modelu COM používat prvky prostředí .NET.

Použití existující COM komponenty v .NET aplikaci.

Nejprve je nutné COM komponentu registrovat v registry (nástroj regsvr32.exe).

Vytvoříme nový projekt v jazyce C#. V okně Solution Explorer aplikace Visual Studio vybereme položku Add Reference.

Najdeme příslušnou COM komponentu.

Reference na tuto komponentu byla přidána a lze ji normálně používat (tedy instanciovat a pak volat její metody,....).

Použití .NET komponenty na místo COM komponenty.

Vytvoříme nový C# projekt typu Class Library.

Komponentu musíme zaregistrovat v Registry. K tomu složí nástroj regasm.exe.

Pro programy, které pracují s typovanými knihovnami lze vygenerovat potřebný .tlb soubor nástrojem tlbexp.exe

Pak lze tuto komponentu využívat stejně jako COM komponentu.

Více o spolupráci s COM komponentami bude v následujících kapitolách.

Spolupráce s běžnými knihovnami DLL - tyto služby umožňují programátorům v prostředí .NET používat knihovny DLL. Následující příklad ukazuje volání metody MessageBox z knihovny User32.dll.

```
using System;
using System.Runtime.InteropServices;

class RunApp
{
  [DllImport("user32.dll", EntryPoint="MessageBox")]
  static extern int NiceWindow(int hWnd, string msg,string caption,int type);

  static void Main(string[] args)
  {
    NiceWindow(0,"Hello world!","C# is calling!",0);
  }
}
```

Platform Invocation Services (PInvoke) - tyto služby umožňují řízenému kódu pracovat s knihovnami a funkcemi exportovanými z dynamických knihoven.

Knihovna DLL je importovaná pomocí atributu DllImport.

Importované funkce musí být označeny jako externí (klíčové slovo extern).

Importovanou funkci lze pro použití v jazyce C# přejmenovat. Slouží k tomu pojmenovaný parametr EntryPoint atributu DllImport.

Nezabezpečený kód - umožňuje programátorům v jazyce C# používat například ukazatele. Takto vytvořený program není spravován CLR systému .NET.

Blok je definován pomocí klíčového slova unsafe a složenými závorkami.

Klíčové slovo unsafe lze použít také jako modifikátor u metody případně tříd.

Je nutné překladači povolit unsafe bloky (Project-Properties-Configuration Properties-Build-Allow Unsafe Code Blocks).

Používáme-li přímý přístup do paměti, mohlo by dojít ke srážce s algoritmem garbage collection.

Klíčové slovo fixed je nástrojem určeným pro znehybnění kusu paměti.

```
fixed (type* pointer = expression) statement
```

Kapitola 5. Bázové třídy

Tato kapitola se zaměřuje na popis bázových tříd. Ty dávají jazyku C# mnoho zásadních funkcí. Jsou základem pro tvorbu kódu.

Třída `System.Object`

Univerzální bázová třída od níž je odvozeno vše ostatní.

Metody třídy `System.Object`:

`string ToString()` - vrací textové vyjádření objektu;

`int GetHashCode()` - používá se tehdy, když je objekt umístěn ve struktuře map. Metoda je využívají instance tříd, které manipulují s uvedenými strukturami. Určují pomocí ní pozici, kam má být instance třídy umístěna;

`bool Equals(Object obj)` - porovnává odkazy dvou objektů. Ve vlastní třídě ji lze implementovat tak, aby prováděla porovnání hodnot dvou objektů. V případě vlastní implementace této metody by programátor neměl vyvolávat výjimky, může to způsobit ve třídách slovníků a v dalších bázových třídách, které interně tuto metodu volají. Statická metoda Equals(object objA, object objB) navíc umí porovnat i hodnoty null;

`bool ReferenceEquals(object objA, object objB)` - statická metoda, rozhoduje, zda se dva odkazy odkazují na stejnou instanci třídy.

operátor porovnání == - ve většině případů znamená porovnání odkazů. V případě určitých tříd je však vhodnější porovnávat obsahy tříd. Například porovnání řetězců. Operátor lze přetížit. Jde o mezistupeň mezi metodami Equals a ReferenceEquals;

`void Finalize()` - destruktor, který je volán v okamžiku, kdy je z dynamické paměti mazán referent;

`Type GetType()` - vrací instanci třídy System.Type, tato třída poskytuje spoustu informací o třídě, jejímž členem dotyčný objekt je. Např. bázový typ, metody, vlastnosti..;

`object MemberWiseClone()` - vytváří mělkou kopii objektu, tzn. kopíruje všechny hodnotové typy v instanci třídy.

Práce s konzolou

V předchozích případech jsme poměrně bohatě využívali výstup na konzolu. Třída **Console** obsahuje množství statických metod, které umožňují vytvářet konzolové aplikace. Ve *Windows Forms* aplikaci jsou ovšem tyto metody ignorovány.

Vstup je získán ze standardního proudového vstupu (**System.IO.TextReader**) , výstup se posílá na standardní proudový výstup (**System.IO.TextWriter**) a chybový výstup se posílá na standardní chybový výstupní proud. Tyto proudy lze změnit pomocí metod konzoly **SetIn()**, **SetOut()**, **SetError()**. Existují tu statické vlastnosti zjišťující současné proudy: **Console.Out**, **Console.In**, **Console.Error**. Ty obsahují pouze funkci **get**, tedy nedají se přímo měnit.

Nejdůležitější pro nás ale budou metody přímo pracující s konzolou, tedy čtení a zápis na standardní datový proud.

Funkce Write, WriteLine

Console.Write(), **Console.WriteLine()** slouží pro zápis hodnot. **Console.WriteLine()** kromě toho, že stejně jako **Console.Write()** vypíše jakýkoliv standardní datový typ, navíc vloží na konec řádku znak nového řádku *\n*.

Způsob použití funkcí **Write** a **WriteLine** může těmato způsoby. Můžeme pracovat s celým argumentem jako s jedním řetězcem. To znamená, že skládáme vše, co se objeví na obrazovce, do jednoho řetězce.

Ukázka kódu

```
int i=3;
Console.WriteLine("Moje cislo "+i+ "neni zrovna stastne.");
```

Nebo lze využít zápis ve tvaru

Ukázka kódu

```
int i=3;
Console.WriteLine("Moje cislo {0} neni zrovna stastne.", i);
```

Kde výraz ve složené závorce odpovídá pořadovému číslu parametru, který vložíme za zobrazovaným řetězcem. Tato možnost umožňuje různý způsob formátování parametru.

Formátovací znaky

Tabulka 5.1. Formátovací znaky pro čísla

Formátovací znak	Význam
C, c	měna
D, d	desítkový zápis
E, e	exponenciální zápis
F, f	zápis s pevnou desetinnou čárkou
G, g	obecný zápis
N, n	číslo
X, x	hexadecimální zápis

Uvedeme si příklad formátování čísla typu **decimal** na tři desetinná místa. Pokud ve formátovacím řetězci neuvedeme za znakem pro zobrazovaný typ nic, u typu **decimal** by se standardně zobrazila dvě desetinná místa.

Ukázka kódu

```
decimal d = 12860m;
Console.WriteLine("{0:c3}", d);
```

Můžeme si ale také definovat vlastní formátovací masku.

Ukázka kódu

```
double d = 4543.909;
Console.WriteLine("{0:00000.##}", d);
```

V tomto příkladu nuly za dvojtečkou ukazují, kolik se musí zobrazit míst před desetinnou čárkou, znak # za desetinnou čárkou, kolik se jich má zobrazit maximálně (tedy nemusí žádné, pokud by číslo d obsahovalo hodnotu *4543*).

Kromě čísel ovšem můžeme při výpisu formátovat i datum.

Pro lepší názornost si ukážeme, jak se vypíše tato hodnota struktury **DateTime**: *10.9.2004 15:34:20.250* v českém prostředí, tzn. používají se tečky jako oddělovače dnů a měsíců.

Tabulka 5.2. Formátovací znaky pro datum

Formátovací znak	Formátovací vzor	Výsledek

Formátovací znak	Formátovací vzor	Výsledek
d	dd.MM.rrrr	10.9.2004
D	dd. MMMM.rrr	10. září 2004
f	dd. MMMM.rrrr HH:mm	10. září 2004 15:34
F	dd. MMMM.rrrr HH:mm:ss	10. září 2004 15:34:20
g	dd.MM.rrrr HH:mm	10.9.2004 15:34
G	dd.MM.rrrr HH:mm:ss	10.9.2004 15:34:20
m, M	dd MMMM	10 září
r, R	ddd, dd MMM rrrr HH':'mm':'ss 'GMT'	Fri, 10 Sep 2004 15:34:20 GMT (RFC 1123)
s	rrrr-MM-ddTHH:mm:ss	2004-09-10T15:34:20 (ISO 8601)
t	HH:mm	15:34
T	HH:mm:ss	15:34:20
u	rrrr-MM-dd HH:mm:ssZ	2004-09-10 15:34:20Z
U	dd. MMMM rrrr HH:mm:ss	9. září 2004 15:34:20
y, Y	MMMM rrrr	září 2004

Tabulka 5.3. Uživatelské formátování data a času

Vzor	Význam
dd	den měsíce jako číslo, pokud je menší než 10, pak s úvodní 0 (10)
ddd	den měsíce jako zkratka (pá)
dddd	název dne v měsíci (pátek)
MM	měsíc jako číslo, pokud je menší než 10, pak s úvodní 0 (09)
MMM	číslo měsíce v římských číslicích (IX)
MMMM	název měsíce (září)
yy	poslední dvě číslice roku (04)
yyyy	všechny čtyři číslice roku (2004)

Speciální znak @

Při výpisu na konzolu lze použít před řetězec, který chceme vytisknout, speciální znak @. Ten překladači říká, aby řetězec bral až do té doby, dokud nenarazí na ukončení řetězce, tedy koncových uvozovek. V takto uvozeném řetězci navíc můžeme uvést znaky běžně nutně uvozované zpětným lomítkem (\), aby byly zobrazeny (například ono samotné lomítko). To umožňuje rozepsat řetězec na více řádků a používat při tom běžně klávesy enter a tyto "problémové" znaky. Řetězec je pro uživatele v programu lépe čitelný.

Ukázka kódu

```
using System;
...
Console.WriteLine(@"Tento řetězec se
na obrazovce objeví tak, jak se v programu
zadá.

Navíc může obsahovat i zpětná lomítka, tedy tato: \");
...
```

Tento znak se ale dá použít i před název proměnné. To způsobí, že překladač bere jakýkoliv text, třeba i klíčové slovo, vyskytující se za tímto znakem, jako identifikátor.

Ukázka kódu

```
int @if = -1; // lze v C# pouzit
```

```
int if = -1; // nelze
```

Takovýmito zápisy ale dochází k nepřehlednosti kódu, proto tento postup není doporučován.

Funkce Read, ReadLine

Console.Read() vrací jeden znak z konzoly, **Console.ReadLine()** celý řetězec ukončený znakem konce řádku. Pokud chceme z konzoly načíst číslo určitého typu, je zapotřebí provést konverzi vstupu na daný typ:

Ukázka kódu

```
double d;
try {
   d = System.Convert.ToDouble(Console.Readline());
   Console.WriteLine("Nactene cislo: "+d);
} catch (FormatException e) {
  Console.WriteLine(e.Message);
}
```

Práce se soubory

Jako soubor se berou data, která jsou uložena na nějakém pamětovém médiu a datový proud (stream) se dá chápat jako prostředník mezi zdrojem (soubor) a cílem (náš program) v případě získávání dat a naopak v případě posílání dat.

My pro práci se soubory budeme využívat především třídy **System.IO.StreamReader** a **System.IO.StreamWriter**. Tyto třídy implementují třídy **System.IO.TextReader** a **System.IO.TextWriter** pro práci s bytovými proudy v patřičném kódování.

K tomu, abychom s těmito třídami mohli pracovat, budeme potřebovat i nějaký proud, odkud (kam) budeme číst (zapisovat). Pro práci se soubory se používá proud **System.IO.FileStream**, pro pamět jako úložiště dat se používá **System.IO.MemoryStream**.

C# kromě práce se soubory (třída **File**), obsahuje také třídu **System.IO.Directory**, která implementuje práci s adresáři. My se budeme více věnovat práci se soubory.

Textový soubor

Textový soubor pracuje se znaky tak, aby byly čitelné uživateli. Všechna zapisovaná data se tedy převedou na typ **string**.

Zápis do textového souboru

Ukázka kódu

```
using System.IO;
...
FileStream fs = new FileStream("soubor.txt", FileMode.Create);
StreamWriter sw = new StreamWriter(fs);

for(int i=0; i<text.GetLength(0); i++)
sw.WriteLine(i);
...
sw.Close();
```

Vytvoříme soubor s názvem *soubor.txt* a mód souboru nastavíme na *Create*, takže pokud soubor s názvem *soubor.txt* už existuje, bude přepsán. Pro tento soubor pak vytvoříme **StreamWriter**, do kterého už jednoduše vkládáme hodnoty, které se nám zlíbí.

V našem příkladě do souboru zapisujeme položky pole typu **string**. Abychom to mohli provést, je třeba vždy volat metodu **ToString()**.

Čtení z textového souboru

Čtení se provádí analogicky, tedy vytvoříme **StreamReader** a proud **FileStream** a poté už můžeme načítat a zpracovávat naše data.

Ukázka kódu

```
...
FileStream fs = new FileStream("soubor.txt", FileMode.Create);
StreamReader sr = new StreamReader(fs);

while(sr != null) {
   Console.WriteLine(sr.ReadLine());
}
sr.Close();
...
```

Binární soubor

Když něco zapíšeme do textového souboru, a pak to chceme zpětně číst, musíme počítat s tím, že budeme načítat řetězce. Proto pro naše textové soubory budeme muset vymyslet různé konvertory do datových reprezentací pro nás vhodných.

O něco jednodušší je to s binárními soubory, jelikož do nich můžeme ukládat binární data. Tedy při čtení můžeme načíst celou hodnotu typu **decimal** apod. Nevýhodou ale zůstává, že do něj můžeme s použitím třídy **BinaryWriter** ukládat pouze atomické typy.

Zápis do binárního souboru

Pro zápis do binárního souboru budeme potřebovat opět proud pracující se souborem. Ten ale můžeme vytvořit i voláním metody **System.IO.File.Create()** a předat ho jako parametr konstruktoru pro binární zápis dat - **BinaryWriter**.

Ukázka kódu

```
using System.IO;
...
BinaryWriter bw = new BinaryWriter(File.Create("soubor.bin", FileMode.Create));

for(int i=0; i<text.GetLength(0); i++)
   bw.Write(i);
bw.Close();
...
```

Čtení z binárního souboru

Při čtení z binárního souboru musíme mít na paměti, který datový typ se právě má číst. Pokud jsme do našeho binárního souboru ukládali například informace o zaměstnancích včetně dat narození, platu apod., kde se vyskytuje více typů (**string**, **decimal** a jiné), musíme si na pořadí typů dát pozor.

Ukázka kódu

```
using System.IO;
...
BinaryReader br = new BinaryReader(File.Open("soubor.bin", FileMode.Open));
try {
   while(true)
      Console.WriteLine(br.ReadInt32());
   }
catch (EndOfStreamException) {}
finally {
   br.Close();
}
```

V příkladu čtení binárních dat načítáme celá čísla tak, jak jsme je do souboru zapsali. Pro přečtení celého souboru jsme použili nekonečný cyklus a hlídáme výjimku na konec vstupu.

Serializace

Jak jsem se zmínil u binárních souborů, lze pomocí třídy **BinaryWriter** ukládat pouze jednoduché typy. Určitě se nám ale bude hodit, pokud budeme chtít uložit celou instanci objektu. To je samozřejmě možné. Tento proces se nazývá *serializace* a lze jej použít pomocí třídy **System.Runtime.Serialization.Formatters.Binary.BinaryFormatter**.

Znalosti

K tomu, abychom mohli instanci nějakého objektu uložit, umístíme před deklaraci jeho třídy atribut **Serializable**. Mějme například třídu `Osoba`, jejíž instance budeme chtít ukládat.

Ukázka kódu

```
[Serializable]
class Osoba {
   private string jmeno;
   private string prijmeni;
   ...
   public Osoba(sring jmeno, string prijmeni) { ... }
}
```

V momentě, kdy budeme chtít nějakou instanci třídy **Osoba** uložit, provedeme následující:

Ukázka kódu

```
...
Osoba karel = new Osoba("Karel", "Polacek")

FileStream fs = new FileStream("soubor.srd", FileMode.Create);
System.Runtime.Serialization.Formatters.Binary.BinaryFormatter output =
   new System.Runtime.Serialization.Formatters.Binary.BinaryFormatter();
output.Serialize(fs, karel);
...
```

V tomto příkladu vytváříme nový soubor. Podobně je možno ukládat i celé kolekce, se kterými jsme předtím pracovali.

Práce s řetězci

Klíčové slovo string je synonymem bázové třídy `System.String`. Tato třída není jedinou třídou pro práci s řetězci. Např. `System.Text` a `System.Text.RegularExpressions`. Při opakované úpravě řetězce je třída `System.String` velmi neefektivní. Je lepší pro takový případ použít `System.Text.StringBuilder`. Formátování textu při volání `Console.WriteLine()` lze ovlivňovat implementací rozhraní `IFormatProvider` a `IFormattable`. Pro vyhledávání a úpravu řetězců lze využít regulárních výrazů za pomocí `System.Text.RegularExpressions`.

Třída `System.String`

Třída pro ukládání řetězců, obsahuje mnoho řetězcových operací. Díky přetíženému operátoru „+“ lze řetězce spojovat. Lze za pomocí indexeru extrahovat znaky z řetězce.

Některé metody třídy System.String:

`Compare()` - porovná znak po znaku obsah řetězců, zohledňuje místní jazykové nastavení;

`CompareOriginal()` - totéž co Compare() ale nezohledňuje místní jazyková nastavení;

`Format()` - formátuje řetězec;

`IndexOf()` - najde první výskyt podřetězce nebo znaku v řetězci;

`IndexOfAny()` - najde první výskyt libovolné množiny znaků v řetězci;

`LastIndexOf()` - najde poslední výskyt podřetězce nebo znaku v řetězci;

`LastIndexOfAny()` - najde poslední výskyt libovolné množiny znaků v řetězci;

`PadLeft()` - Doplní zleva řetězec opakováním daného znaku;

`PadRight()` - Doplní zprava řetězec opakováním daného znaku;

`Replace()` - nahradí výskyty daného řetězce zadaným řetězcem nebo znakem;

`Split()` - rozdělí řetězec na podřetězce, zlomení dojde v místech nalezení zadaného znaku;

`Substring()` - vrátí podřetězec požínající zadanou pozicí v řetězci;

`ToLower()` - převede řetězec na malá písmena;

`ToUpper()` - převede řetězec na velká písmena;

`Trim()` - odstraní ze začátku a konce řetězce mezery.

Řetězec ve třídě `String` je neměnný. Metody manipulující s řetězcem vytvářejí vždy novou instanci. Aby nedocházelo k problémům s paměťovou náročností při úpravě řetězců, máme k dispozici třídu `System.Text.StringBuilder`. Třída není tak výkonná jako `System.String`, ale její použití při konkatenaci, náhradě, atd. řetězců je mnohem efektivnější.

Porovnávání řetězců

System.String je odkazovým typem. Lze jej však porovnat operátorem ==, neboť je přetížený.

Třída `System.Text.StringBuilder`

má vlastnosti `Length` (aktuální délka řetězce) a `Capacity` (objem rezervované paměti). Konstruktor očekává řetězec, kapacitu, nebo řetězec a kapacitu. Zde se při úpravě řetězce nevytváří nová instance, ale upravuje se obsah stávající. Nová paměť bude rezervována tehdy, když po vykonané operaci řetězec přesahuje kapacitu.

Některé metody třídy `System.Text.StringBuilder`:

`Append()` - připojí řetězec;

`Insert()` - vloží řetězec;

`Remove()` - odstraní znaky z řetězce;

`Replace()` - nahradí výskyty znaku nebo podřetězce jiným znakem nebo podřetězcem;

`ToString()` - Vrací obsah řetězce přetypovaný na `System.String`.

Formátovací řetězce

jsou řetězce, které jsou doplněny o další informace říkající, jak má výsledný řetězec nakonec vypadat. Používá se zejména u specifických formátů příslušejících různým místním nastavením. Například formát data a podobně. Do složených závorek se uvádí číselné hodnoty, které určují jak má být zobrazovaná informace formátována, resp. kolik znaků bude k zobrazení informace potřeba. První číslo však vždy uvádí, který z následujících argumentů bude použit při formátování. Například `Console.Writeln(„a = {0,10:E}; b = {1};”,`

`a, b);` Kladné číslo zarovnává doprava, záporné číslo doleva. Za dvojtečkou je uveden typ zobrazované informace, viz tabulka několika typů:

C – číselný typ – lokální formát měny ($435,25)

D – celočíselný typ – obecný formát

E – číselný typ – vědecký formát s exponentem (4,85300E+003)

F – číselný typ – pro pevnou desetinnou čárku (458,235)

G – číselný typ – obecný formát (458,235)

N – číselný typ – odděluje tisíce na základě národního nastavení

P – číselný typ – formát procent

X – Pouze celočíselný typ – hexadecimální formát

Metoda `Console.Writeln` předá argumenty metodě `String.Format()`. Ten rozloží řetězec na několik částí a předá jej ke zpracování několika metodám.

Metoda AppendFormat musí vědět jakým způsobem má objekt formátovat. Zjistí, zda daný objekt implementuje rozhraní `System.IFormattable`. Pokud ne, zavolá metodu `ToString()`. Pokud ano zavolá dvouargumentovou metodu `ToString (viz rozhrani System.IFormattable)`. Použití metody `Writeln` s jedním argumentem vytiskne řetězec bez formátování.

Rozhraní `System.IFormattable` předpokládá definici dvouargumentové metody `ToString(string format, IFormatProvider formatProvider);` Prvním argumentem je řetězec určující požadovaný formát a druhým argumentem je odkaz na objekt implementující rozhraní `IFormatProvider`. Toto rozhraní poskytuje další informace, jak formátovat řetězec.

Regulární výrazy

Slouží pro práci s řetězci. Na základě zadaného vzoru lze v řetězci vyhledat (a třeba následně upravit) podřetězce. Regulární výrazy v jazyce C# jsou navrženy podle Perl 5 s dalšími rozšířeními. Operace s regulárními výrazy poskytuje třída `System.Text.RegularExpressions`. Pro použití se vytvoří instance třídy `System.Text.RegularExpressions.Regex`, nebo se zavolá statická metoda `RegEx`, které se předá v parametru řetězec a regulární výraz. Následuje příklad použití regulárního výrazu k vyhledání textu.

Ukázka kódu

```
string text = "hello world";
string pattern = "o";

MatchCollection Matches(text, pattern, RegexOptions.IgnoreCase |
RegexOptions.ExplicitCapture);

foreach (Match NextMatch in Matches)
{
  Console.WriteLine(NextMatch.Index);
}
```

Několik metaznaků, které může regulární výraz obsahovat:

^ - počátek vstupního textu;

$ - konec vstupního textu;

. – jakýkoliv znak kromě konce řádku;

* - libovolný počet výskytů znaku uvedeného před hvězdičkou;

+ - jeden nebo více výskytů znaku uvedeného před plus;

? – 0 nebo 1 výskyt znaku uvedeného před hvězdičkou;

\s – bílý znak;

\S – vše kromě bílého znaku;

\b – hranice slova;

\B – jakákoli pozice, která není hranicí slova;

kulaté závorky – ohraničují skupiny znaků (metazaků).

Kolekce

Kolekce jsou standardní datové struktury doplňující pole, což je jediná vestavěná datová struktura v jazyce C#. Jazyk obsahuje sadu typů poskytujících datové struktury a podporu vytváření vlastních typů. Ty se dělí do dvou kategorií: rozhraní, jež definují standardizovanou sadu vzorů návrhu pro kolekce obecně, a konkrétní třídy implementující tato rozhraní.

Iterování kolekcemi

Existuje mnoho různých druhů kolekcí. Interní implementace se značně liší, procházení kolekcemi je téměř univerzální. Tato funkčnost je zajištěna dvěma rozhraními `IEnumerable` a `IEnumerator`.

Rozhraní `IEnumerable` a `IEnumerator`

Potřebuje-li kolekce vystavit enumerátor, implementuje rozhraní `IEnumerable`. Enumerátor umožňuje procházet iterovanými prvky směrem vpřed.

Ukázka kódu

```
public interface IEnumerable
{
  IEnumerator GetEnumerator();
}
public interface IEnumerator
{
  bool MoveNext();
  object Current {get;}
  void Reset();
}
```

Rozhraní `IEnumerable` má metodu `GetEnumerator()`, která vrací odkaz na rozhraní `IEnumerator`.

Rozhraní `IEnumerator` nabízí standardní mechanismus pro iterování kolekcí (metody `MoveNext()` a `Reset()`) a převzetí aktuálního prvku (vlastnost `Current()`) jako obecný odkaz na `object`.

Následuje příklad procházení kolekce:

Ukázka kódu

```
MyCollection myCollection = new MyCollection()
...
//IEnumerator usage, insert your type instead of XXX
IEnumerator ie = myCollection.GetEnumerator();
while(ie.MoveNext())
{
  XXX  item = (XXX) ie.Current;
  Console.WriteLine(item);
  ...
}
```

Příklad procházení kolekce pomocí příkazu `foreach`:

Ukázka kódu

```
MyCollection myCollection = new MyCollection()
...
//usage of foreach, insert your type instead of XXX
foreach(XXX item in myCollection)
{
  Console.WriteLine(item);
  ...
}
```

Implementování `IEnumerable` a `IEnumerator`

Při implementaci zmíněných rozhraní je třeba věnovat pozornost sémantickým kontraktům rozhraní. Rozhraní IEnumerator je často implementováno jako vnořený typ, který se inicializuje předáním kolekce konstruktoru IEnumerator:

Ukázka kódu

```
public class MyCollection : IEnumerable
{
  int[] data;

  public virtual IEnumerator GetEnumerator()
  {
    return new MyCollection.Enumerator(this);
  }

  private class Enumerator:IEnumerator
  {
    MyCollection outside;
    int actualIndex = -1;
    internal Enumerator(MyCollection outside)
    {
      this.outside = outside;
    }
    public object Current
    {
      get
      {
        if (actualIndex == outside.data.Length) throw new
InvalidOperationException();
        return outside.data[actualIndex];
      }
    }
    public bool MoveNext()
    {
      if (actualIndex > outside.data.Length) throw new
InvalidOperationException();
      return ++actualIndex < outside.data.Length;
    }
    public void Reset()
    {
      actualIndex = -1;
    }
  }
}
```

Rozhraní `IDictionaryEnumerator`

Používá se u slovníkových datových struktur. Je to standardizované rozhraní používané k postupnému procházení obsahu slovníku, kde má každý element klíč a hodnotu. Vlastnost `Entry` je obdobou `Current` a `Key` a `Value` zajišťují přímý přístup ke klíčům a hodnotám elementů.

Ukázka kódu

```
public interface IDictionary : IEnumerator
{
  DictionaryEntry Entry {get;}
  object Key {get;}
  object Value {get;}
}
```

Iterátor implementovaný pomocí *yield*

Pokud bychom do kódunapříklad 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:

Ukázka kódu

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

Standardní rozhraní kolekcí

.NET definuje sadu tří standardních rozhraní, které implementují v zájmu zajištění operací jako určení velikosti, prohledávání kolekce a podobně.

Rozhraní `ICollection`

Standardní rozhraní pro počitatelné kolekce. Umožňuje určit velikost , možnost změny, synchronizaci kolekce a podobně. `ICollection` rozšiřuje `IEnumerable`, takže je možné typy implementující `ICollection` procházet podobně jako v předchozích případech.

Ukázka kódu

```
public interface ICollection : IEnumerable
{
  void copyTo(Array array, int index);
  int Count {get;}
  bool IsReadOnly {get;}
  bool IsSynchronized {get;}
  object SyncRoot {get;}
}
```

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.

Ukázka kódu

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

Vlastní vykonávání kódu probíhá takto:

Na začátku je zavolána metoda `s.GetEnumerator()`.

V místě příkazu `yield` je provádění kódu pozastaveno, je uchován kontext a hodnota `data[i]` je dosazena do proměnné `v`.

Je provedeno tělo příkazu `foreach` (je vypsána hodnota `v`).

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).

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`).

Rozhraní `IList`

Standardní rozhraní pro indexované kolekce. Kromě ICollection umožňuje indexovat prvky pomocí pozice. Lze odstraňovat, přidávat a měnit prvky kolekce.

Ukázka kódu

```
public interface IList : ICollection, IEnumerable
{
  object this[int index] {get; set;}
  int Add(object o);
  void Clear();
  bool Contains(object value);
  int IndexOf(object value);
  void Insert(int index, object value);
  void Remove(object value);
  void RemoveAt(int index);
}
```

Rozhraní `IDictionary`

Standardní rozhraní pro kolekce používající dvojice klíč/hodnota jako hešovací tabulky a mapy vlastností. podobá se IList, ale umožňuje přistupovat k prvkům na základě klíče.

Ukázka kódu

```
public interface IDictionary : ICollection, IEnumerable
{
  object this[object key] {get; set;}
  ICollection Keys {get;}
  ICollection Values {get;}
  void Clear();
  bool Contains(object value);
  IDictionaryEnumerator GetEnumerator();
  void Remove(object key);
}
```

Předdefinované třídy kolekcí

K dispozici je rozsáhlá sada předem sestavených datových struktur.

Třída `Array`

Datová struktura představující pole pevné velikosti odkazů na objekty jednotného typu. Implementuje rozhraní `ICollection`, `IEnumerable` a `IList`, takže s poli lze pracovat jako se seznamy, s kolekcemi nebo množinami elementů, jimiž lze procházet. `Array` dále nabízí možnost řazení a prohledávání pole. Přiklad použití:

Ukázka kódu

```
string[] strs1 = { "now", "time", "right", "is" };
Array.Reverse(strs1);
Array strs2 = Array.CreateInstance(typeof(string), 4);
strs2.SetValue("for", 0);
strs2.SetValue("all", 1);
strs2.SetValue("good", 2);
strs2.SetValue("people", 3);
Array strings = Array.CreateInstance(typeof(string), 8);
Array.Copy(strs1, strings, 4);
strs2.CopyTo(strings, 4);
foreach(string s in strings)
  Console.WriteLine(s);
```

Třída `ArrayList`

Dynamické pole objektů implementující rozhraní `IList`. Udržuje si interní pole objektů, které je nahrazeno větším polem, jakmile je zcela naplněno elementy. Třída je efektivní při vkládání objektů, protože na konci obvykle bývá volné místo. Příklad použití:

Ukázka kódu

```
ArrayList a = new ArrayList();
a.Add("Vernon");
a.Add("Corey");
a.Add("William");
a.Add("Muzz");
a.Sort();
for(int i = 0; i< a.Count; i++)
  Console.WriteLine(a[i]);
```

Třída `Hashtable`

standardní slovníková klíč/hodnota datová struktura. Používá hashovací algoritmus k ukládání a indexování

hodnot. Příklad použití:

Ukázka kódu

```
Hashtable ht = new Hashtable();
ht["one"] = 1;
ht["two"] = 2;
ht["three"] = 3;
Console.WriteLine(ht["two"]);
```

Třída `Queue`

Datová struktura reprezentující frontu (FIFO - First In First Out). Poskytuje jednoduché operace pro přidání do fronty (`Enqueue`), odebrání z fronty (`Dequeue`), prohlédnutí elementu na začátku fronty a podobně.

Třída `Stack`

Datová struktura reprezentující zásobník (LIFO - Last In First Out). Poskytuje jednoduché operace pro přidání do zásobníku (`Push`), odebrání ze zásobníku (`Pop`).

Třída `BitArray`

Dynamické pole hodnot `bool`. Z hlediska paměti je vhodnější než pole hodnot `bool`, protože používá pro každou položku právě jeden bit, zatímco pole `bool` používá celý bajt.

Třída `SortedList`

Slovníková datová struktura implementující rozhraní `IDictionary`. K indexování prvků používá prohledávání binárním odseknutím.

Třída `StringCollection`

Datová struktura kolekce pro ukládání řetězců. Implementuje rozhraní `ICollection`.

Třída `StringDictionary`

Slovníková datová struktura pro ukládání dvojic klíč/hodnota v nichž je typem klíče řetězec. Nabízí podobné metody jako třída `Hashtable`. Implementuje standardní rozhraní `IEnumerable`.

Třídění instancí

Implementace schopnosti řazení a prohledávání kolekcí závisí na prvcích obsažených v samotných objektech kolekce. K porovnávání se využívá vygenerované číslo, tzv. hešovací kód (hashcode).

Rozhraní `IComparable`

Umožňuje jednomu objektu indikovat své pořadí vzhledem k jiné instanci téhož typu.

Ukázka kódu

```
public interface IComparable
{
  int CompareTo(object rhs);
}
```

Sémantická pravidla:

a patří před b, pak `a.CompareTo(b) < 0;`

a patří za b, pak `a.CompareTo(b) > 0;`

a je rovno b, pak `a.CompareTo(b) = 0;`

`null` je první: `a.CompareTo(null) > 0;`

když `a.CompareTo(b)`, pak `a.GetType() == b.GetType()` .

Příklad implementace rozhraní:

Ukázka kódu

```
public sealed class Person : IComparable
{
  public string Name;
  public int Age;
  public int CompareTo(object o)
  {
    if (o == null) return 1;
    if (o.GetType() != this.GetType()) throw new ArgumentException();
    Person rhs = o as Person;
    if (Age < rhs.Age) return 1;
    if (Age < rhs.Age) return -1;
    return 0;
  }
}
```

Rozhraní `IComparer`

Implementace tohoto rozhraní provádí porovnávání (či řazení). Porovnává dva nezávislé objekty. Obsahuje jedinou metodu `int Compare(object x, object y)`.

Generování hešovacího kódu

Všechny instance objektů mohou poskytnout 32bitový celočíselný hešovací kód svého obsahu s využitím metody `GetHashCode()`. Metoda se používá u rychlého (méně důvěryhodného) porovnání dvou objektů. Tzn. v případě, že chceme porovnávat objekty, lze nejprve provést porovnání pomocí `GetHashCode()` a až po úspěchu použít `Equals(..)`.

U použití GetHashCode() u řazení je dobré si uvědomit, jak generování heše probíhá. Záleží na implementaci, na tom jaký zvolí programátor algoritmus. Dobré algoritmy využívají všech 32 bitů a poskytují dobrou rovnoměrnou distribuci a v ideálním případě zachovávají pořadí proměnných. To aby bylo zajištěno, že například bod(10,20) bude hešován jinak než bod(20,10). Zachování pořadí se obvykle provádí vynásobením každé proměnné nějakým konstantním prvočíslem (obvykle jde o číslo 37).

Některé datové struktury (jako Array) nemusejí svůj obsah hešovat správně, proto může být zapotřebí hešovat je ručně.

Příklad implementace hešování tak aby došlo ke vhodné distribuci:

Ukázka kódu

```
public sealed class Data
{
  public readonly short x, y;
  public readonly Color c;
  ArrayList al = new ArrayList();

  public override int GetHashCode()
  {
    int hc = 1; //base.GetHashCode when base!=object
    hc = 37*hc + x<<16|(ushort)x;
    hc = 37*hc + y.GetHashCode();
    hc = 37*hc + (c==null ? 0 : c.getHashCode());

    foreach (object o in al)
      hc = 37*hc + o.GetHashCode()
```

```
        return hc;
      }
    }
```

Reflexe

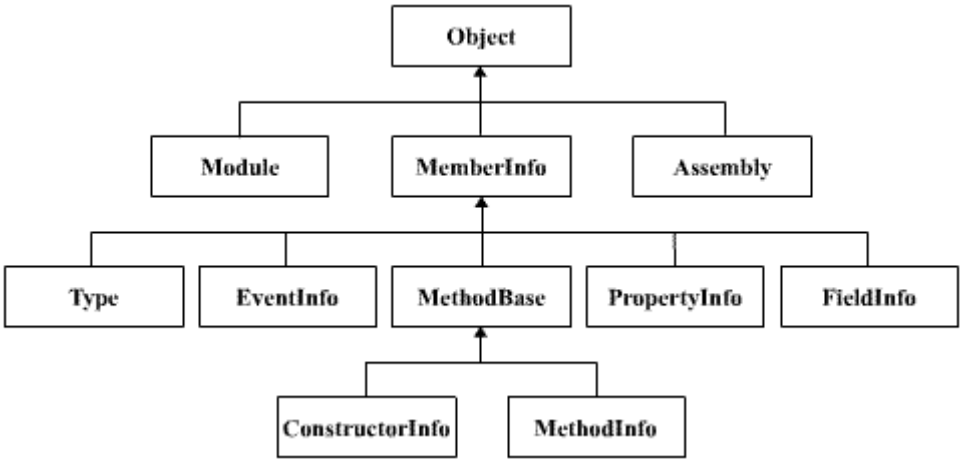
Mnohé ze služeb platformy .NET (jako pozdní vazba, serializace, vzdálené řízení, atributy a podobně) závisejí na přítomnosti metadat. Vytvářené programy mohou využívat tato metadata a rozšiřovat je o nové informace. *Reflexí* je označováno prozkoumávání existujících typů prostřednictvím metadat. Uskutečňuje se prostřednictvím sady typů v oboru názvů `System.Reflection`. Je rovněž možné dynamicky vytvářet nové typy pomocí tříd v oboru názvů `System.Reflection.Emit`. Reflexe představuje procházení a manipulování s objektovým modelem, který představuje nějakou aplikaci včetně všech jejích elementů kompilace a běhu.

Hierarchie typů

Základní jednotkou aplikace jsou její typy obsahující členy a vnořené typy. Typy jsou v modulech a ty obvykle v sestavách. Všechny tyto elementy jsou popsány metadaty. Za běhu jsou tyto elementy obsaženy uvnitř domény AppDomain.

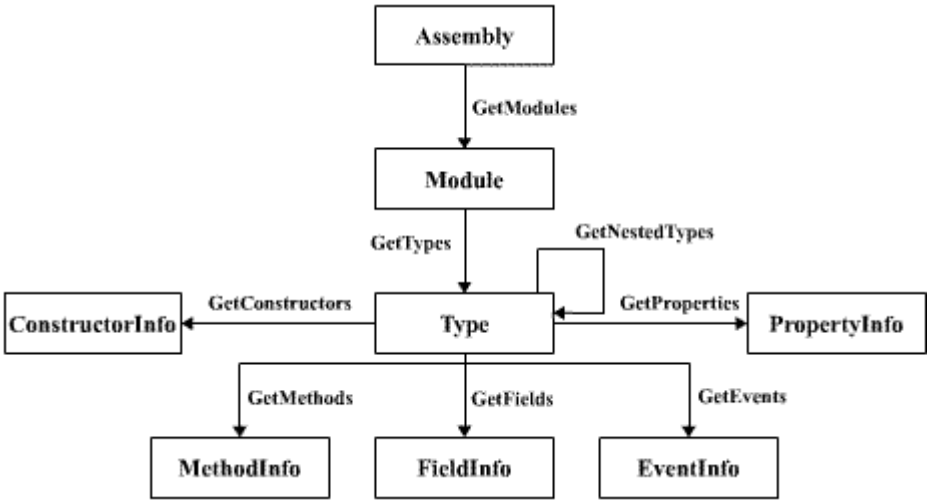
Každý z prvků aplikace je vystaven prostřednictvím odpovídajícího typu z oboru názvů `System` nebo `System.Reflection`.

Obrázek 5.1. Vztahy mezi dědičnosti mezi reflexními typy .NET



V okamžiku, kdy máte odkaz na některý z těchto elementů, můžete se pohybovat vztahy mezi daným elementem a souvisejícími prvky, jak je uvedeno na následujícím obrázku.

Obrázek 5.2. Pohyb hierarchií reflexe .NET



Typy, členy a vnořené typy

Třída `Type` je nejzákladnějším typem reflexe. Reprezentuje metadata pro jednotlivé deklarace typů v aplikaci. Typy obsahují členy. Ty zahrnují konstruktory, proměnné, vlastnosti, události a metody. Typy mohou navíc obsahovat vnořené typy, které se typicky používají jako pomocné třídy. Typy jsou seskupené do modulů a moduly jsou obsažené v sestavách.

Sestavy a moduly

Sestava je logickým ekvivalentem knihovny DLL v systému Win32 a je základní jednotkou zavádění, správy verzí a opakovaného používání typů

Modul je fyzický soubor (.exe, .dll), nebo nějaký prostředek (.gif, .jpg). Sestava může být tvořena několika moduly.

`AppDomain`

`AppDomain` je kořenem hierarchie typů a slouží jako kontejner sestav a typů. `AppDomain` je logickým ekvivalentem procesu v aplikaci Win32. Chyby aplikace způsobí zhroucení pouze té `AppDomain`, kde došlo k chybě.

Zjištění typu instance

V jádru systému reflexe je `System.Type`, což je abstraktní bázová třída poskytující přístup k metadatům nějakého typu.

K instanci třídy Type lze přistoupit pomocí `GetType()`. Tato metoda je implementovaná v `System.Object`. Po zavolání vrátí metoda konkrétní typ `System.Type`, který dokáže typ reflektovat a manipulovat s ním.

Přímé zjištění typu

Určitou třídu `Type` lze převzít pomocí názvu prostřednictvím statické metody `GetType()` ve třídě `Type`.

Ukázka kódu

```
Type t = Type.GetType("System.Int32");
Type t2 = Type.GetType("MyNamespace.MyType", MyAssembly);
```

C# nabízí operátor `typeof`, který vrací třídu `Type` libovolného typu známého během kompilace.

Ukázka kódu

```
Type t = typeof(System.Int32);
```

Rozdíl mezi těmito dvěma přístupy je, že přístup pomocí `Type.GetType` se vyhodnocuje za běhu, `typeof` se vyhodnocuje v době kompilace.

Reflektování hierarchie typů

Máme-li instanci `Type`, můžeme přistupovat k metadatům prostřednictvím typů, jež představují členy, moduly, sestavy, třídy AppDomain a vnořené typy. lze zkoumat metadata, vlastní atributy, vytvářet nové instance typů a volat členy.

Příklad používá reflexi k zobrazení členů ve třech různých typech.

Ukázka kódu

```
using System;
using System.Reflection;

class Test
{
  static void Main()
  {
    object o = new Object();
    Information(o.GetType());
    Information(typeof(int));
    Information(Type.GetType("System.String"));
  }
  static void Information(Type t)
  {
    Console.WriteLine("Type: {0}", t);
    MemberInfo[] miarr = t.GetMembers();
    foreach(MemberInfo mi in miarr)
      Console.WriteLine(" {0}={1}", mi.MemberType, mi);
  }
}
```

Pozdní vazba

Je dynamické vytváření instancí a používání typů za běhu.

Ukázka kódu

```
//Hello.cs

public abstract  class Hello {
   public abstrakt void SayHello();
}
```

Ukázka kódu

```
//EnglishHello.cs

using System;
public class AmericanHello : Hello
{
  private string message = "Hey Dude!";
  public override void SayHello()
  {
    Console.WriteLine(message);
  }
}
public class BritishHello : Pozdravy
{
  private string message = "Hello!";
  public override void SayHello()
```

```
  {
    Console.WriteLine(message);
  }
}
```

Ukázka kódu

```
//SayHello.cs

using System;
using System.Reflection;

class Test
{
  static void Main()
  {
    string s = "EnglishHello.dll";
    Assembly a = Assembly.LoadFrom(s);
    foreach (Type t in a.getTypes())
    {
      if (t.IsSubClasOf(typeof(Hello))
      {
        object o = Activator.CreateInstance(t);
        MethodInfo mi = t.GetMethod("SayHello");
            mi.Invoke(o, null);
      }
    }
  }
}
```

Podle řetězce *s* se nahraje *EnglishHello.dll*. Z jejich typů se vyhledá ten objekt, který má za předka třídu `Hello` a zajistí se nalezení metody `SayHello`. Metoda se vyvolá.

Vytváření nových typů za běhu

Obor názvů `System.Reflection.Emit` obsahuje třídy, které dokáží vytvořit za běhu úplně nové typy. Třídy umožňují:

definovat dynamickou sestavu v paměti;

definovat dynamický modul v této sestavě;

definovat nový typ v tomto modulu, včetně všech jeho členů;

vytvořit kódy MSIL potřebné k implementování aplikační logiky ve členech.

Vlákna

Aplikace může běžet v jednom nebo více vláknech (tocích), která se vykonávají "paralelně" v téže aplikaci.

Jednoduchá vícevláknová aplikace

Ukázka kódu

```
using System;
using System.Threading;

class ThreadTest
{
  static void Main(){
    Thread t = new Thread(new ThreadStart(Run));
    t.Start();
    Run();
   }
  static void Run()
  {
```

```
    for(char c='a'; c<'z'; c++)
      Console.Write(c);
  }
}
```

V příkladu se zkonstruuje nový objekt vlákna předáním delegáta `ThreadStart`, který obaluje metodu, specifikující, kde má začít vykonávání daného vlákna. Pak se vlákno spustí a zavolá metodu `Run()`. Obě vlákna začnou vypisovat abecedu do konzole. Nevýhodou je, že sdílejí konzoli společně, takže se může (a pravděpodobně i stane), že budou obě abecedy zapletené do sebe.

Synchronizace vláken

Technika zajišťující koordinovaný přístup ke sdíleným prostředkům.

Příkaz `lock`

Příkaz `lock` zajišťuje, že k nějakému bloku kódu může přistoupit pouze jedno vlákno.

Ukázka kódu

```
using System;
using System.Threading;

class LockTest
{
  static void Main()
  {
    LockTest lt = new LockTest();
    Thread t = new Thread(new ThreadStart(lt.Run));
    t.Start();
    lt.Run();
  }
  public void Run()
  {
    lock(this) {
      for(char c='a'; c<'z'; c++)
        Console.Write(c);
    }
  }
}
```

Výsledkem bude v konzoli dvakrát za sebou napsaná abeceda.

Operace `Pulse` a `Wait`

Operace umožňující vláknům navzájem komunikovat prostřednictvím monitoru, který udržuje seznam vláken čekajících na obdržení zámku nějakého objektu.

Ukázka kódu

```
using System;
using System.Threading;

class MonitorTest
{
  static void Main()
  {
    MonitorTest mt = new MonitorTest();
    Thread t = new Thread(new ThreadStart(mt.Run));
    t.Start();
    mt.Run();
  }
  static void Run(){
    for(char c='a'; c<'z'; c++)
    lock(this)
    {
      Console.Write(c);
```

```
            Monitor.Pulse(this);
            Monitor.Wait(this);
          }
        }
    }
```

Metoda `Pulse` říká monitoru, aby vzbudil další vlákno, které čeká na zámek daného objektu. Volání metody `Wait` vlákno uspí. Výsledkem programu bude v konzoli napsaná zdvojená abeceda *aabbccdd....*

Atomické operace

Atomická operace je operace, která nemůže být přerušena jiným vláknem a tudíž není třeba používat zámku. Například aktualizování proměnné je atomická operace, protože je zaručeno dokončení operace bez předání řízení jinému vláknu.

Obvyklé typy vláken

Třída `Monitor`

`System.Threading.Monitor` poskytuje implementaci Hoareho monitoru. Jde o nejzákladnější třídu vláken.

Metody `Enter` a `Exit`

Získávají (resp. uvolňují) zámek nějakého objektu. Pokud nějaký objekt drží nějaké vlákno, pak metoda `Enter()` čeká, až bude zámek uvolněn nebo dokud není vlákno přerušeno výjimkou ThreadInterruptedException. Každému volání `Enter()` by mělo odpovídat volání `Exit()` pro tentýž objekt v tomtéž vlákně.

Metoda `PulseAll`

Pokud čeká ve frontě monitoru na zámek více vláken, tak metoda `Pulse()` probudí jen to první ve frontě. Metoda `PulseAll()` postupně probudí všechny.

Asynchronní delegáti

Je možné zavolat nějakou metodu asynchronně. Runtime nabízí standardní způsob asynchronního volání metod.

Invoke

```
návratový-typ Invoke(seznam-parametrů);
```

`Invoke()` volá metodu asynchronně a volající musí čekat až delegát skončí vykonávání (standardní volání delegáta v C# volá `Invoke()`)

`BeginInvoke` a `EndInvoke`

```
IAsyncResult BeginInvoke(seznam-parametrů, IAsyncCallback ac, object stav);

návratový-typ EndInvoke(ref/out seznam-parametrů, IAsyncCallback ac);
```

`BeginInvoke` volá delegáta se zadaným seznamem parametrů a pak se okamžitě vrací. Toto volání se provede, je-li v *ThreadPool* k dispozici volné vlákno. `EndInvoke` přebírá návratovou hodnotu volané metody se všemi odkazovanými parametry a výstupními parametry. Mohlo totiž dojít k jejich změně.

Ukázka kódu

```
using System;
using System.Threading;
using System.Runtime.Remoting.Messaging;
```

```
delegate int Compute(string s);

class Class1
{

  static int TimeConsumingFunction(string s)
  {
    return s.Length;
  }
  static void ViewResultFunction(IAsyncResult ar)
  {
    Compute c = (Compute)(((AsyncResult)ar).AsyncDelegate);
    int result = c.EndInvoke(ar);
    string s = (string)ar.AsyncState;
    Console.WriteLine("{0} contains {1} chars", s, result);
  }
  static void Main()
  {
    Compute c = new Compute(TimeConsumingFunction);
    AsyncCallback ac = new AsyncCallback(ViewResultFunction);
    string s1 = "Christopher";
    string s2 = "Nolan";
    IAsyncResult ar1 = c.BeginInvoke(s1, ac, s1);
    IAsyncResult ar2 = c.BeginInvoke(s2, ac, s2);
    Console.WriteLine("Ready");
    Console.Read();
  }
}
```

Výstupem je:

Ukázka kódu

```
Ready
Christopher contains 11 chars
Nolan contains 5 chars
```

Kapitola 6. Práce v síti, webové služby a ASP.NET

Kdo by ve dnešní době nepoužíval Internet nebo alespoň neměl nějakou tu zkušenost s počítačovými sítěmi? V této kapitole si nastíníme některé body, z nichž síťová komunikace v .NET Framework sestává. Sestrojíme si jednoduchý webový klient, řekneme si něco o síťové komunikaci. Podíváme se na webové služby a řekneme si něco málo o ASP.NET.

V .NET Framework jsou pro programování síťových aplikací k dispozici zejména dva jmenné prostory: **System.Net** a **System.Net.Sockets**. Třídy a metody v nich obsažené nám pomohou v mnohém při komunikaci po síti. V .NET Framework lze komunikovat prostřednictvím proudů (streams) i datagramů. Pro komunikaci založenou na proudech je používán protokol TCP, zatímco pro komunikaci založenou na datagramech protokol UDP.

Důležitá třída ze jmenného prostoru **System.Net.Sockets** je **System.Net.Sockets.Socket**. Instance **Socket**u má k sobě připojen místní a vzdálený koncový bod. Místní koncový bod v sobě obsahuje informace o spojení pro současnou instanci třídy **Socket**.

Další třídy užitečné pro programování se sítěmi jsou **IPEndPoint**, **IPAddress**, **SocketException** apod. Prostředí .Net Framework podporuje jak synchronní, tak asynchronní komunikaci mezi klientem a serverem.

Získání informací o hostu

Pro práci v síti existuje řada užitečných tříd a metod. Nyní si ukážeme příklad na jednoduché aplikaci zjišťující informace o zadaném hostu (serveru).

Abychom mohli pracovat s informacemi o serveru, budeme potřebovat třídu **System.Net.HostEntry**. Ta udržuje informace o adrese hosta. Adresu je možno získat s použitím statické metody **Dns.Resolve()**. Tato metoda vrací typ **HostEntry** a jako argument lze použít buď doménu nebo IP adresu. Existují zde samozřejmě jiné metody, jako například **Dns.GetHostByName()** nebo **Dns.GetHostByAddress()** provádějících prakticky totéž.

Ukázka kódu

```
using System.Net;
...
IPHostEntry he = Dns.Resolve("www.cs.vsb.cz");
```

Z instance třídy **IPHostEntry** lze následně získat informace o hostu - aliasy hosta, všechny IP adresy přiřazené dané doméně nebo celé jméno hosta.

Ukázka kódu

```
...
foreach (string alias in he.Aliases)
  Console.WriteLine(alias);
foreach (IPAdress ip in he.AddressList)
  Console.WriteLine(ip);
Console.WriteLine(he.HostName);
```

Příklad ke stažení

Celý příklad je k dispozici ke stažení zde [examples/HostInfo.cs].

Webový klient

Pracovat v sítovém prostředí lze na nižší úrovni, k tomu nám slouží například třídy jmenného prostoru **System.Net.Sockets**. Zde musíme dbát na to, zda je napojen socket a posléze tento socket zpracovat. Dá se ale

pracovat i bez potřeby práce na nižší úrovni. Můžeme využívat třídy, které se postarají o navázání spojení a vyřízení požadavku.

Třídy WebRequest a WebResponse

Na následujícím příkladu si vytvoříme jednoduchého webového klienta s pomocí tříd **WebRequest** a **WebResponse**.

Ukázka kódu

```
using System.Net;
using System.IO;
...
// vytvoreni get pozadavku na cilovou uri
WebRequest request = WebRequest.Create("www.cs.vsb.cz");
// ziskani odezvy na pozadavek
WebResponse response = request.GetResponse();
// vytvoreni proudu z odezvy
Stream receieveStream = response.GetResponseStream();
// vytvoreni ctenare proudu
StreamReader sr = new StreamReader(receiveStream, Encode.UTF-8)

// vytvoreni a nacteni bufferu pro nacitana data
Char[] read = new Char[256];
int count = sr.Read(read, 0, 256);

// zpracovani ziskanych dat
while(count > 0) {
  string str = new string(read, 0, 256);
  System.Console.WriteLine(str);
  count = sr.Read(read, 0, 256);
}
```

Jak je vidět z ukázky kódu, stačí vytvořit instance tříd **WebRequest**, **WebResponse**, **Stream** a **StreamReader** a můžeme zpracovávat data získaná z webového požadavku.

Příklad ke stažení

Kompletní projekt na demonstraci tříd **WebRequest** a **WebResponse** s pomocí grafického rozhraní je k dispozici zde [examples/HTTP_Client.zip].

Třída WebClient

Předcházející příklad se dá vytvořit jak složitěji, tak o něco jednodušeji. A to s pomocí třídy **WebClient**, která pracuje na vyšší úrovni než **WebRequest** a **WebResponse**.

Ukázka kódu

```
using System;
using System.Net;
using System.Text;

class MyWebClient {
  public static void Main(string[] args) {
    WebClient wc = new WebClient();
    byte[] buffer = wc.DownloadData(args[0]);
    string content = Encoding.ASCII.GetString(buffer);
    Console.WriteLine(content);
  }
}
```

Vidíme, že není třeba vytvářet vlastní webové klienty. S pomocí typu **WebClient** můžeme provádět operace jako download a upload dat či souborů, získávat informace o hlavičkách apod.

Příklad ke stažení

Tento přklad je možno k vyzkoušení stáhnout zde [examples/EasyWebClient.cs].

Třída TcpClient, TcpListener

Jak jsme se zmínili v odstavci 2.2, lze webového klienta (a síťovou práci celkově) sestrojit pomocí "delšího kódu" ručně. To si ukážeme nyní. Pro podporu protokolu TCP nám dobře poslouží třídy (**System.Net.**) **TcpClient** a **TcpListener**.

TcpListener je vytvořen pro naslouchání příchozích konekcí a odesílání odpovědí instancí typu **Socket** odpovídající požadavku připojení. Používá se tedy na straně serveru. **TcpClient** slouží k připojení ke vzdálenému počítači (hostu). Po vytvoření připojení lze využít proudu (**NetworkStream**) pro posílání/příjem dat.

Nyní si pro změnu ukážeme náznak jednoduchého HTTP serveru vyřizujícího požadavek GET.

Pokud chceme realizovat HTTP server, je třeba znát IP adresu a port, na který se budou posílat data. Pro naše účely použijeme místní počítač (adresa *127.0.0.1*) a port si zvolíme jiný než *80*, např. *5050*. Vytvoříme tedy instanci třídy **TcpListener**, která bude na dané adrese naslouchat.

Ukázka kódu

```
using System;
using System.Net;
using System.Net.Sockets;
...
TcpListener myListener = new TcpListener(IPAddress.Parse("127.0.0.1"), 5050);
Socket socket;
// zacina se naslouchat na dane IP adrese a portu
myListener.Start();

// navazani spojeni a vyrizovani prichozich socketu
while(true) {
  socket = myListener.AcceptSocket();

  if(socket.Connected) {
    // vytvoreni bufferu pro ziskana data
    Byte[] received = new Byte[1024];
    // prijimani dat - pozadavku
    int i = socket.Receive(received, received.Length, 0);

    // prevod bytu na retezec
    string request = System.Text.Encoding.ASCII.GetString(received);

    /// vytvoreni bufferu pro naplneni daty k odeslani
    Byte[] dataToSend = new Byte[...]
    // .... zpracovani prichoziho pozadavku ....
    // .... ziskani souboru k odeslani ....

    // odeslani dat klientu
    socket.Send(dataToSend, dataToSend.Length, 0);
  }
}
...
```

Po vytvoření "naslouchače požadavků" se čeká (v nekonečném cyklu) na příchozí požadavek. V momentě, kdy nejaký klient zažádá na adrese *127.0.0.1* a portu *5050* o spojení, to se vytvoří. Pak se vytvoří přijímací buffer a získaná data se načtou pomocí metody **Socket.Receive()**. Následně se získaná data převedou na řetězec a ten se může zpracovat jako text obsahující podmínky požadavku.

Jakmile získáme z požadavku název souboru, který se má zobrazit (poslat), můžeme provést jeho načtení (binární) a naplnění bufferu připraveného pro odeslání. Odeslání provedeme metodou **Socket.Send()**.

Samozřejmě takto vytvořený server by nebyl velice užitečný. V jednu chvíli by mohl zpracovat a vyřídit pouze jeden požadavek, v té chvíli by byl nedostupný pro ostatní klienty. Toto se dá vyřešit s pomocí vláken, kdy

bychom s každým příchozím požadavkem spouštěli metodu tento požadavek vyřizující.

Příklad ke stažení

Příklad s HTTP serverem je k dispozici k nahlédnutí a vyzkoušení zde [examples/HTTP_Server.zip].

Webové služby

Znalosti

Webová služba umožňuje webové stránce rozšířit její funkcionalitu a dynamičnost díky možnosti programování. Webové služby mohou přijímat zprávy a na tyto zprávy odpovídat.

Dnes je možno přes internet provádět dotazy nad databázemi, objednat si letenku, zjistit stav naší zakázky apod.

Webové služby mohou být volány prostřednictvím *POST* nebo *GET* metod protokolu *HTTP* ve spojení s protokolem *SOAP*.

Znalosti

SOAP (Simple Object Access Protocol) slouží pro vzdálené volání procedur. Jedná se o specifikaci vytvořenou skupinou společností, mezi které se řadí i Microsoft a DevelopMentor. SOAP je založen na všeobecně přijímaných internetových standardech jako je XML.

Webové služby se ukládají do souborů s příponou *.asmx*. Podmínkou pro správnou funkci webové služby je přítomnost ASP.NET na webovém serveru. Pokud prostředí ASP.NET zjistí požadavek na soubor s touto příponou, zprostředkuje volání handleru webové služby.

Tyto handlery jsou reprezentovány instancemi tříd implementujících rozhraní **System.Web.IHTTPHandler**. Rozhraní **IHTTPHandler** definuje dvě metody - **IsReusable** a **ProcessRequest**. **IsReusable** umožňuje zjistit instanci **IHTTPHandleru**, zda může být použita vícekrát. Metoda **ProcessRequest** má objekt **HttpContext** jako svůj parametr. Právě zde se začíná implementovat práce s HTTP.

Tyto handlery přijímají požadavky a vracejí odpovědi. Pokud se v HTTP prostředí zjistí přítomnost na požadavek souboru s příponou .aspx, zavolá se handler registrovaný pro soubory s touto příponou. V prostředí ASP.NET je tímto handlerem **System.Web.UI.PageHandlerFactory**. V případě .asmx je zavolán handler **System.Web.Services.Protocols.WebServiceHandlerFactory**.

Díky tomuto handleru je ASP.NET schopno s použitím reflexe vytvořit dynamicky HTML stránku. Stránka bude obsahovat možnosti a metody volané služby. Na této straně se rovněž objeví *možnost otestovat metody této služby*.

WSDL a SDL

Znalosti

WSDL (Web Service Definition Language) je jazyk založený na XML sloužící pro popis možností webových služeb. Popisuje rozhraní služby, typy argumentů a návratových hodnot. Tento jazyk umožňuje zapsat webovou službu jako XML data a odeslat klientovi. Díky této možnosti lze tuto webovou službu přenést na lokální počítač.

Jednoduchá webová služba

Vytvoříme jednoduchou webovou službu, kterou uložíme jako *GreetingService.asmx*:

Ukázka kódu

```
<%@ WebService Language="C#" class="GreetingService" %>
using System;
using System.Web.Services;
```

```
public class GreetingService {
  [WebMethod]
  public string Hello(string who) {
    return ("Hello, world!!! Special greetings for " + who);
  }
}
```

Pro označení metody jako použitelné pro webovou službu se nad tuto metodu přidává atribut **WebMethod**. Abychom mohli tuto metodu vyzkoušet na našem počítači, je zapotřebí mít nainstalovaný webový server podporující ASP.NET (IIS apod.). Tento soubor zkopírujeme do virtuálního adresáře serveru a zavoláme jej z prohlížeče, např.: *http://localhost/sluzba/GreetingService.asmx*.

Webové služby nejsou omezeny používáním pouze jednoduchých datových typů, mezi které patří řetězce a čísla. Lze používat i složitější typy, například struktury apod.

Způsob využití webové služby

Součástí Microsoft Visual Studia je konzolová aplikace WSDL (umístění v adresáři *Microsoft Visual Studio .NET\Sdk\v1.1\Bin*). Tato aplikace vytvoří zástupce webové služby tak, aby ji bylo možno používat lokálně. Předtím je třeba udělat následující kroky.

Nejprve zjistíme umístění webové služby.

Například víme, že se služba nachází na adrese *http://localhost/sluzba\GreetingTest.asmx*.

Následně vytvoříme místního zástupce (proxy) pro vzdálenou službu. Tento zástupce umožní programátorovi s webovou službou pracovat, jako by se nacházela na lokálním počítači.

wsdl "http://localhost/sluzba/GreetingService.asmx" /out:GreetingServiceProxy.cs

Vytvoříme knihovnu ze zástupce webové služby

csc /out:bin\GreetingServiceProxy.dll /t:library /r:system.web.services.dll /r:system.xml.serialization.dll GreetingServiceProxy.cs

vytvořili jsme *GreetingServiceProxy.dll* v adresáři *Bin* aktuálního adresáře

Následující aplikace používá službu **GreetingService**.

Ukázka kódu

```
using System;
class MyClass {
  public static void Main(string[] args) {
    // vytvoreni instance objektu puvodne definovaneho v GreetingService.asmx
    // a pripojeneho diky WSDL kontraktu
    GreetingService gs = new GreetingService();
    // nasleduje bezna prace s instanci vzdaleneho objektu
    Console.WriteLine(gt.Hello("Jan"));
  }
}
```

Takto vytvořený kód uložíme do souboru *GreetingTest.cs* a budeme kompilovat následujícím způsobem (a s těmito knihovnami):

csc /r:system.web.services.dll /r:GreetingServiceProxy.dll GreetingTest.cs

Ukázka ASP.NET

Dnes stalé víc a víc výrobců programuje webové aplikace i když jsou určeny pro použití pouze na osobních počítačích. Jejich výhodou je, že jsou postaveny na standardních protokolech jako je HTTP, HTML, XML a podobně. Další velkou výhodou je uživatelské rozhraní. Programátor může využít standardní internetový prohlížeč. V prostředí .NET je podpora webových aplikací realizovaná technologií ASP.NET. Oblastí, kde se tato technologie protíná s jazykem C# jsou webové formuláře.Jde o programový model, ve kterém jsou stránky

(obsahující webové formuláře) generovány na straně serveru a v podobě čistého HTML dodávány klientovi(prohlížeči). Proces vytváření takovéto stránky lze takto shrnout.

Nejprve vytvoříme HTML stránku, ta bude obsahovat všechny statické prvky.

Pak doplníme program v jazyce C#. Tento program bude generovat dynamický obsah.

Program se spustí na straně serveru a výsledek jeho činnosti bude integrován do statického HTLM.

Výsledná HTML stránka je poslána prohlížeči klienta.

Webové formuláře rozdělují uživatelské rozhraní na dvě části: vizuální komponenty,také nazývané uživatelské rozhraní(user interface-UI) a logiku, která je skryta za těmito vizuálními prvky. Podpora webových formulářů je integrovana ve vývojovém prostředí Visual Studio.NET. Autoři tohoto vývojového prostředí implementovali technologii Rapid Application Development (RAD). Tato technologie si klade za cíl zjednodušit vývoj webové aplikace. Programátor jednoduše "nakliká" stránku a pak už jen implementuje logiku této stránky a právě tuto logiku budeme vytvářet v jazyce C#.

Architektura webových formulářů je postavena na událostech. Události jsou generovány ve chvíli kdy uživatel stiskne tlačítko, vybere položku v menu nebo jinak využije možnosti, které mu poskytuje uživatelské rozhraní. Další události také může generovat systém. Právě obsluha těchto událostí tvoří logiku webového formuláře. ASP.NET používá dva typy událostí:

Postback události jsou ty, které způsobí, že se daný formulář ihned odešle zpět na server;

Non-postback události jsou naproti tomu kontrolní jednotku uloženy a zpracovány až ve chvíli, kdy je provedena nějaká postback událost.

Prostředí Internetu a webových aplikací je bezestavové. ASP.NET ale poskytuje jistý mechanismus, který definuje stav uživatele. Tento stav je uložen ve vnitřní proměnné *ViewState* a předáván ve skryté položce formuláře.

Vytvoření webového formuláře

Podpora webových formulářů je integrovaná ve Visual Studiu. Jako v jiných případech lze aplikace vytvářet i jinak, ale tvorba webových formulářů je takto mnohem jednodušší.Vytvoříme-li projekt ASP.NET Web From s názvem HelloWeb je vygenerovaná následující kostra.

HelloWeb.aspx - soubor s uživatelským rozhraním.

Ukázka kódu

```
<%@ Page language="c#"
  Codebehind="HelloWeb.aspx.cs"
  AutoEventWireup="false"
  Inherits="ProgrammingCSharpWeb.HelloWeb" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<html>
  <head>
    <title>HelloWeb</title>
    <meta name="GENERATOR"
      Content="Microsoft Visual Studio 7.0">
    <meta name="CODE_LANGUAGE" Content="C#">
    <meta name="vs_defaultClientScript" content="JavaScript">
    <meta name="vs_targetSchema"
      content="http://schemas.microsoft.com/intellisense/ie5">
  </head>
  <body MS_POSITIONING="GridLayout">
    <form id="Form1" method="post" runat="server">
    </form>
  </body>
</html>
```

HelloWeb.aspx.cs - soubor se zdrojovými kódy. Ty implementují logiku, která je skryta za danou stránkou.

Ukázka kódu

```
public class HelloWeb : System.Web.UI.Page
{
  ...
}
```

Tuto kostru lze jednoduše rozšiřovat. Nejjednodušší variantou je vpisovat kód do souboru HelloWeb.aspx. Do tohoto souboru lze umisťovat buď HTML kód, nebo vpisovat kód v jazyce C#. Tyto možnosti demonstruje následující příklad:

Ukázka kódu

```
Hello world! It is now <%= DateTime.Now.ToString() %>
```

Nejzajímavější možností jak rozšiřovat webové formuláře je přidávání ovládacích prvků. To mohu být tlačítka, textová pole a podobně. V prostředí ASP.NET existují dva typy ovládacích prvků.

HTML Web Controls (server-side HTML controls) - Toto jsou standardní ovládací prvky HTML fromulářů doplněné o atribut *runat="Server"*.

```
<input type="button">
```

ASP.NET Web Controls

Byly navrženy, aby v prostředí ASP.NET nahradily stávající ovládací prvky jazyka HTML.

Ovládací prvky jazyka ASP poskytují mnohem konzistnější objektový model a jména atribut.

Tyto ovládací prvky jsou pak ve skutečnosti realizovány pomocí standardních HTML značek.

```
<asp:Button>
```

Použití tchto prostředků demonstruje následující příklad. Vygenerovanou kostru příkladu HelloWeb jsme rozšířili o tlačítko a textové políčko.

HelloWeb.aspx - soubor s užvatelským rozhraním.

Ukázka kódu

```
<%@ Page language="c#"
  Codebehind="HelloWeb.aspx.cs"
  AutoEventWireup="false"
  Inherits="ProgrammingCSharpWeb.HelloWeb" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<html>
  <head>
    <title>HelloWeb</title>
    <meta name="GENERATOR"
      Content="Microsoft Visual Studio 7.0">
    <meta name="CODE_LANGUAGE" Content="C#">
    <meta name="vs_defaultClientScript" content="JavaScript">
    <meta name="vs_targetSchema"
      content="http://schemas.microsoft.com/intellisense/ie5">
  </head>
  <body MS_POSITIONING="GridLayout">
    <form id="Form1" method="post" runat="server">
      <asp:TextBox id="SomeTextBox"
        style="Z-INDEX: 107; LEFT: 128px; POSITION: absolute; TOP: 64px"
        runat="server"></asp:TextBox>
      <asp:Button id="SomeButton"
        style="Z-INDEX: 105; LEFT: 20px; POSITION: absolute; TOP: 197px"
        runat="server" Text="Button"></asp:Button>
    </form>
  </body>
</html>
```

HelloWeb.aspx.cs

Ukázka kódu

```
public class HelloWeb : System.Web.UI.Page
{
  protected System.Web.UI.WebControls.Button SomeButton;
  protected System.Web.UI.WebControls.TextBox SomeTextBox;
  #region Web Form Designer generated code
  override protected void OnInit(EventArgs e)
  {
    //
    // CODEGEN: This call is required by
    // the ASP.NET Web Form Designer.
    //
    InitializeComponent( );
    base.OnInit(e);
  }
  /// <summary>
  /// Required method for Designer support - do not modify
  /// the contents of this method with the code editor.
  /// </summary>
  private void InitializeComponent( )
  {
    this.SomeButton.Click += new System.EventHandler(this.SomeButton_Click);
  }
  #endregion
  private void Order_Click(object sender, System.EventArgs e)
  {
    // create the message by getting
    // the values from the controls
  }
}
```

Kapitola 7. ADO.NET

Tato kapitola je zasvěcena oblasti ADO.NET. Po prostudování kapitoly bychom měli být schopni říci, co ADO.NET vlastně je. Měli bychom také umět vytvořit jednoduché aplikace vycházející z této technologie.

ADO.NET

Znalosti

ADO.NET (Microsoft ActiveX Data Objects .NET) představuje množinu tříd nabízejících služby pro přístup k datům a tvorbu databázových aplikací. Daty máme nyní na mysli převážně informace uložené v databázích. Ať již se jedná o data v databázích například na Microsoft SQL Serveru či data zpřístupněná přes OLE DB nebo XML. Mezi jeho přednosti patří především jednoduchý způsob použití, rychlost při zpracování a další. Stačí vytvořit spojení se serverem, s kterým budeme chtít pracovat, pomocí zvoleného adaptéru a zadaného dotazu získat z databáze data a ty pak načíst do některé z připravených konstrukcí pro práci s daty z tabulek.

ADO.NET ale nemusí pracovat pouze s databázemi na nějakém serveru. Bylo navrhováno současně s XML třídami v prostředí .NET Framework. Také díky tomu je možno data načítat i ve formátu XML nebo data zapisovat jako XML soubory spolu s definičním souborem XSD definujícím schéma dané databáze.

Nástroje ADO.NET byly navrženy tak, aby se oddělil způsob přístupu k datům od manipulace s daty. K první skupině patří .NET Framework data provider obsahující množinu komponent zahrnujících podmnožiny *Connection* (připojení), *Command* (množinu příkazů pro vybrání dat), *DataReader* (načítání dat) a *DataAdapter* (adaptér pro připojení k databázi). K druhé skupině řadíme mimo jiné objekt **DataSet** (skládající se z objektů **DataTable**, **DataRow**...). Jedná se o objekty uchovávající data načtená z databází. Tyto objekty mohou s daty pracovat stejně jako s daty v databázi. V dalším textu budeme převážně používat objekt typu **DataSet**.

Prostředí Microsoft Visual Studio .NET 2003 nabízí poměrně jednoduchý způsob, jak vytvořit připojení k databázi a jak vybrat potřebná data.

Přesto oblast ADO.NET je velice obsáhlá a vydala by na samostatný kurz. Proto si osvětlíme hlavně její základy.

Connection - přístup k databázi

Pro to, abychom mohli s databází pracovat, je třeba nejprve vytvořit připojení. V závislosti na používaném serveru vybereme typ připojení. V našem příkladu budeme předpokládat, že používáme Microsoft SQL Server. Proto použijeme typ **SqlConnection** z jmenného prostoru **System.Data.SqlClient**.

Pro připojení je nutné znát tzv. *ConnectionString*, tedy řetězec používaný pro připojení k databázovému serveru, ve kterém se uvádí název databáze, login, heslo a další parametry. Naštěstí ve Visual Studiu toto nemusíme nutně znát. Stačí, když, při zobrazení designer (View -> Designer) z Toolboxu (View -> Toolbox - Data) vybereme prostředek s názvem *SqlConnection* (viz obrázek) a vložíme jej do našeho formuláře.

Tím se vytvoří nový objekt **SqlConnection**. Zdrojový kód každého vkládaného prvku si můžeme zobrazit poklepáním pravým tlačítkem myši na příslušný prvek a vybráním možnosti *View Code*. Po vytvoření objektu se vytvoří část kódu

Ukázka kódu

```
this.sqlConnection = new System.Data.SqlClient.SqlConnection();
```

kde **sqlConnection** je jméno instance tak, jak jsme si ji vytvořili.

Můžeme vše psát bez pomocí vizuálních prvků. S nimi je ale manipulace daleko jednodušší a přehlednější. Například výše zmíněný *ConnectionString* nemusíme vůbec vymýšlet. Pravým tlačítkem myši si vyvoláme vlastnosti prvku **SqlConnection**, kde se v sekci *Data* nachází rolldown menu pro *ConnectionString*. Pokud

klikneme na nové spojení, objeví se dialogové okno, ve kterém si nastavíme vše, co potřebujeme (viz obrázek).

Samozřejmě na našem serveru (nebo na tom, na který se připojujeme) musí existovat nějaká databáze, ze které budeme data čerpat.

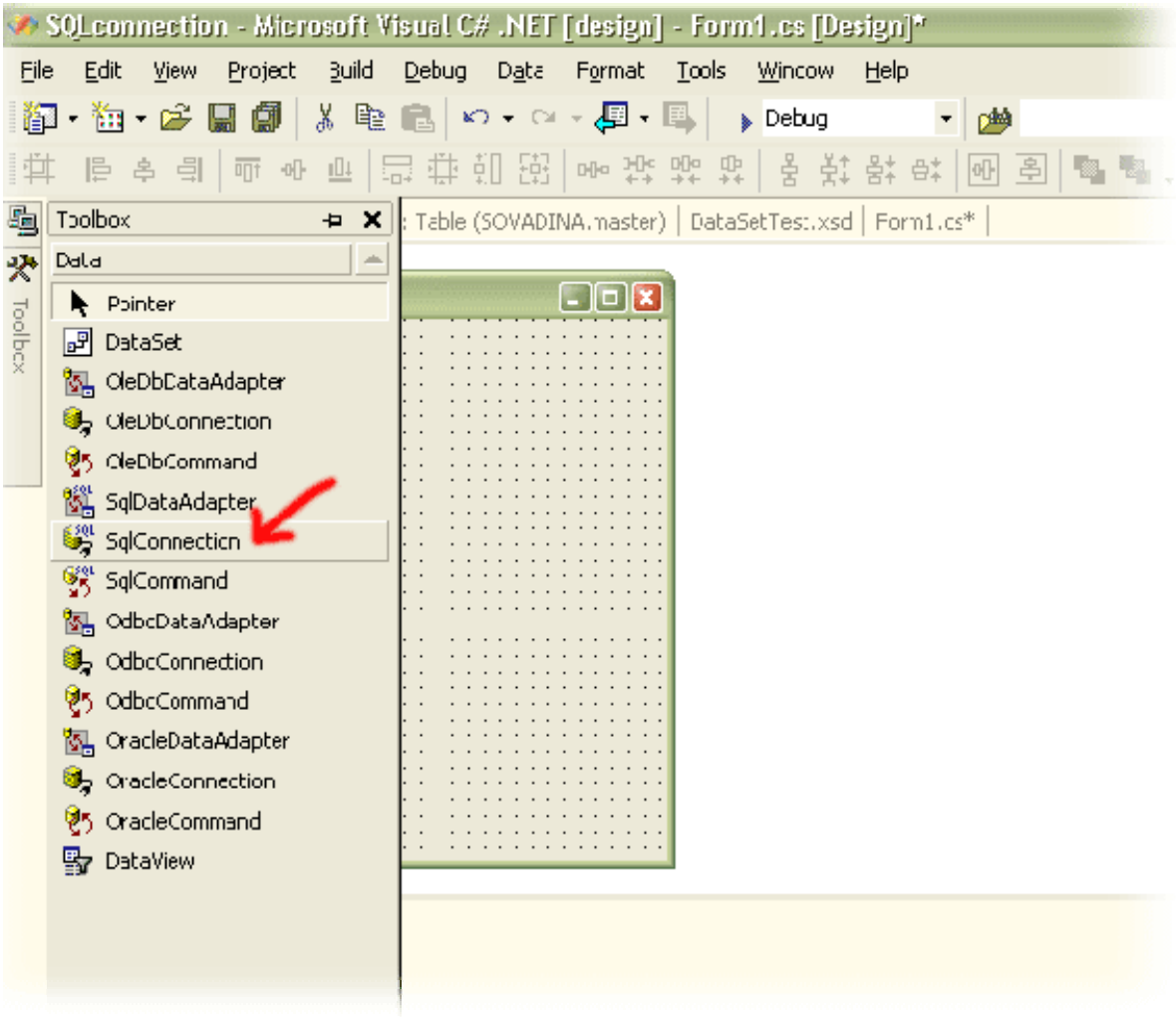
Konkrétně na našem obrázku jsme vybrali databázový server *SOVADINA*, jméno databáze *master*. Následně se v metodě **InitializeComponent()** vytvořil tento *ConnectionString*:

Ukázka kódu

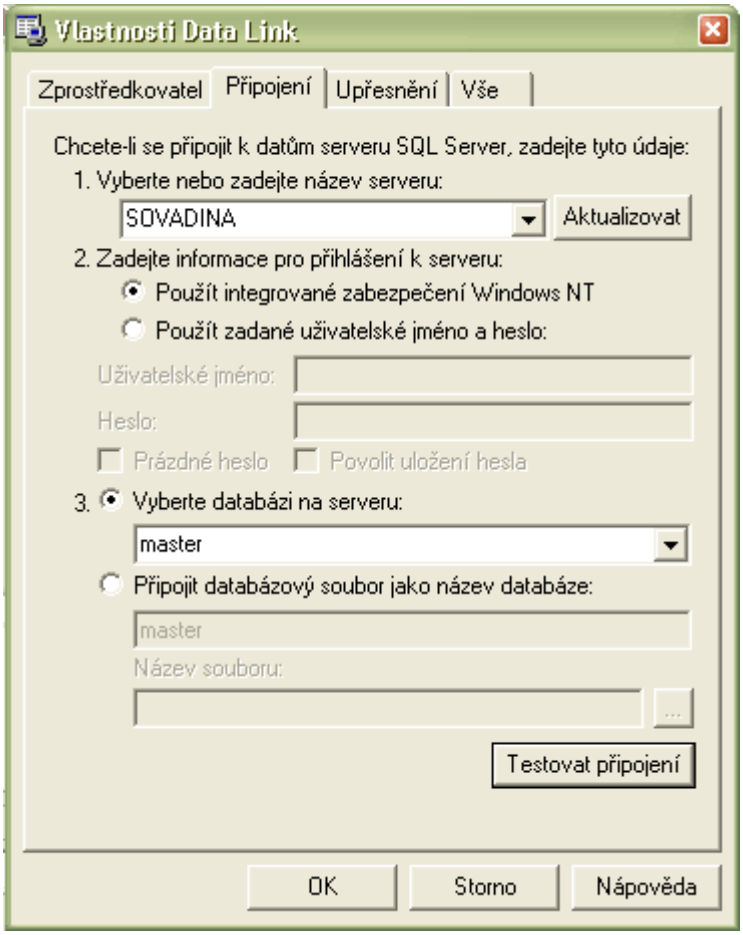
```
this.sqlConnection.ConnectionString = "workstation id=SOVADINA;packet
size=4096;integrated " +
   "security=SSPI;data source=SOVADINA;persist security info=False;initial
catalog=master";
```

V ConnectionStringu výraz *integrated security=SSPI* znamená, že se bude jako login brát přihlašovací jméno do systému (Windows).

Obrázek 7.1. Toolbox

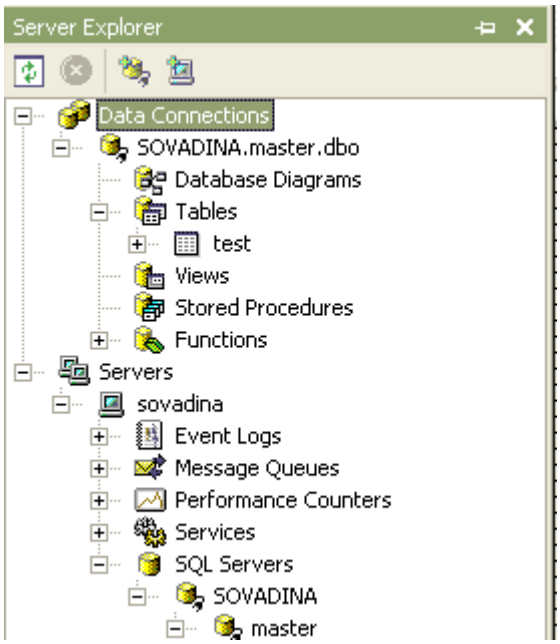


Obrázek 7.2. Dialogové okno nového spojení



Poté, co jsme vytvořili připojení (**SqlConnection**), se v nástrojovém okně *Server Explorer* vytvoří naše připojení jako nová položka *Data Connections*. Díky této možnosti nebudeme muset prohledávat celý server a pokaždé stačí si patřičná data najít právě zde.

Obrázek 7.3. Server Explorer s vytvořeným datovým spojením



Data Adapter

DataAdapter je užitečný především pro spolupráci s prvky dočasné databázové paměti typu **DataSet**. Pokud se tedy rozhodneme nepoužít **DataSet**, například u jednoduchých operací, můžeme s databází spolupracovat přímo bez DataAdapteru.

Ve druhém případě je dalším krokem přidání prvku **DataAdapter** do našeho projektu.

V našem případě použijeme **SqlDataAdapter**. Opět jej vložíme z *Toolboxu*. V momentě vkládání se objeví průvodce konfigurací **DataAdapter**u. Také jej později můžeme vyvolat poklepáním na prvek pravým tlačítkem myši, volba *Configure Data Adapter*.

V průvodci Data Adaptéru si opět můžeme vybrat různá nastavení, např. zda dotazy na databázi budeme vytvářet sami nebo chceme opět použít průvodce s předdefinovanými procedurami.

Pokud se po ukončení průvodce podíváme na vlastnosti prvku **SqlDataAdapter**, zjistíme, že jsou nastaveny vazby na **SqlConnection**, které jsme vytvořili dříve. Můžeme opět jednotlivé vlastnosti měnit podle našich přání a požadavků. Například měnit dotazy typu *SELECT*, *INSERT*, *UPDATE*, *DELETE*.

Samozřejmě u zdrojových kódů přibyla spousta nových řádků, které vygenerovalo vývojové prostředí.

Je na místě oznámit, že je třeba vytvořit tolik prvků **SqlDataAdapter**, do kolika tabulek databáze se budeme chtít připojovat.

Nyní máme vytvořené potřebné připojení k databázi a připravena data vybraná pomocí příkazů pro práci s databázemi. Zbývá jen těmito daty naplnit objekty nabízející se v ADO.NET.

Přístup k datům databáze bez DataAdapteru

Pokud pracujeme s databázemi, lze využívat jejich dat i bez objektu **DataSet**. Po vytvoření spojení vytvoříme příkaz, který chceme nad naší databází provést a spustíme jej. Pokud příkaz vrací data z databáze (*SELECT*...), vytvoříme navíc konstrukce pracující s takto získanými daty.

Ukázka kódu

```
using System;
using System.Data.SqlClient;
...
SqlConnection myConn = new SqlConnection("..."); // spravny connection string
SqlCommand selectCommand = new SqlCommand("SELECT ... WHERE ... ", myConn); //
spravna formulace dotazu SELECT
SqlDataReader myDataReader = new SqlDataReader();
myDataReader = selectCommand.ExecuteReader();
try {
  myConn.Open();
  while(myDataReader.Read()) {
    ... // zpracovani dotazu
  }
} catch (Exception e) {
  Console.WriteLine("Zase se to stalo :-( \n" + e.Message);
} finally {
  myConn.Close();
}
```

Pro ty z vás, kteří se s programováním s databázemi setkali, nebude ničím zvláštní (až na jazykové odlišnosti) výše uvedený kód.

Nejprve se vytvoří spojení s databází. Následuje příkaz *SELECT*. Ten vybere z databáze data na základě určité podmínky (*WHERE*). **DataReader** načítá vždy po jednom záznamu vyhovujícímu podmínce. Data takto získaná lze zpřístupnit pomocí indexerů. Pokud například chceme získat všechna data z tabulky test obsahující sloupce *IDzakazky*, *nazev*, *hodnota*, pak **myDataReader** bude vždy (pokud byl správně vygenerován na počátku) obsahovat indexery **myDataReader["IDzakazky"]**, **myDataReader["nazev"]**, **myDataReader["hodnota"]**.

DataSet

DataSet představuje *reprezentaci vázaných dat "offline" v paměti*. Jeho výhoda tedy je, že všechna data jednou načte, a pak už nic nenačítá z databáze.

DataSet může obsahovat i více tabulek v něm reprezentovaných jako objekty **DataTable**. Může navíc definovat i vazby mezi těmito tabulkami a spoustu dalších možností. Celkově se tedy bere jako "úschovna dat", s kterou můžeme provádět velkou množinu operací.

Datový typ **DataSet** může být naplněn dvěma způsoby - z klasické databáze nebo ze souboru XML reprezentujícího databázi.

Typovaný vs. netypovaný DataSet

Existují dva způsoby, jak přistupovat k prvku **DataSet**.

Jedním z nich je vytvořit prvek **DataSet** odvozený od původní **DataSet** třídy, následně využít informace z XML schématu týkajícího se databáze, kde má být náš **DataSet** použit a vygenerovat novou třídu. Takto se tabulky, sloupce, relace apod. se generují do této třídy jako množina podtříd a vlastností. Tento způsob se nazývá *typovaný dataset*.

Znalosti

Netypovaný DataSet oproti typovanému nepotřebuje žádné schéma. Obsahuje tabulky, sloupce a ostatní, ale tyto jsou přístupny pouze jako kolekce.

Následují příklady typovaného a netypovaného prvku **DataSet**.

Ukázka kódu

```
// chceme z tabulky Zamestnanci ziskat prvni hodnotu sloupce s nazvem IDzam
// DataSet se nazyva dsZam
// typovany DataSet
string s = dsZam.Zamestnanci[0].IDzam;

// netypovany DataSet
string s = (string)dsZam.Tables("Zamestnanci").Rows[0]["IDzam"];
```

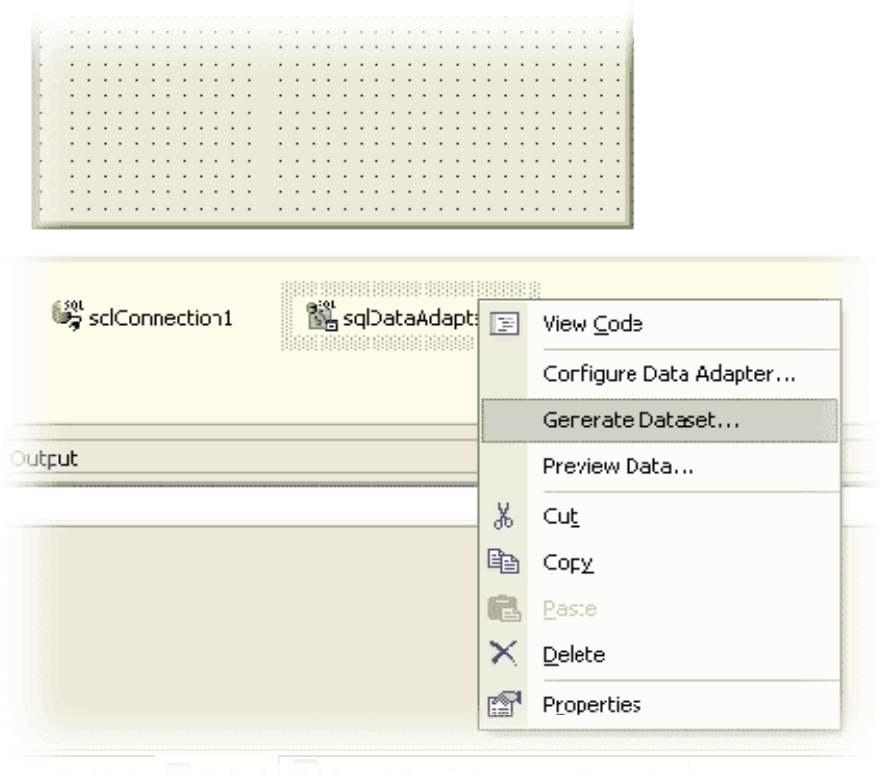
Z příkladu je patrné, že je daleko přehlednější použít typovaný **DataSet**. Visual Studio má pro typovaný **DataSet** podporu, jako je například *IntelliSense* (zobrazování možností při psaní kódu). Je také méně náchylný na chyby. Ovšem ne vždy budeme moci typovaný **DataSet** použít. Jedná se například o obecné použití prvku DataSet pro práci s různými daty.

Občas mohou nastat situace, kdy se neubráníme používání netypovaného **DataSet**, ale pokud je ta možnost, je typovaný **DataSet** doporučován. My v dalším budeme pracovat s typovanými prvky **DataSet**.

DataSet při získání dat z databázového serveru

U databázového serveru máme informace o tom, v jakém tvaru budou všechny tabulky a potřebná data. Proto pokud budeme chtít vytvořit nový **DataSet** tak, aby obsahoval přístup k datům z naší tabulky (tabulek), jednoduše opět poklepeme pravým tlačitkem myši na prvek dříve vložený prvek **SqlDataAdapter**. Ten se již nachází pod naším formulářem (Form) v návrhovém režimu (Design). Vybereme možnost *Generate Dataset*.

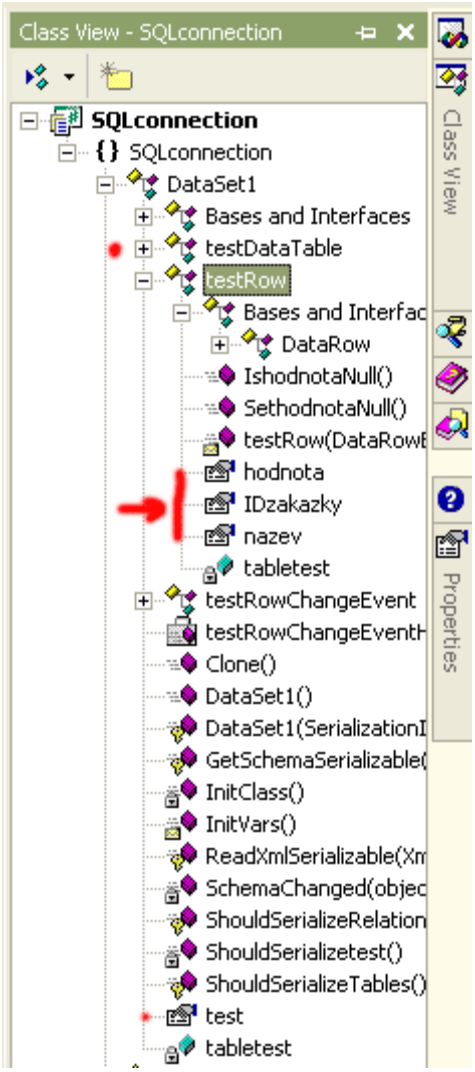
Obrázek 7.4. SqlDataAdapter - Generate DataSet



Pokud se po tomto kroku podíváme do seznamu tříd v našem projektu, všimneme si, že přibyla nová třída nesoucí název právě vloženého prvku **DataSet**. Obsahuje zároveň i potřebné metody a vlastnosti pro přístup k položkám tabulky *test*.

Na obrázku vidíme strukturu třídy **DataSet1** vegenerovanou z **SqlDataAdapteru** na tabulku *test*. Mimo jiné se zde nachází třídy **testDataTable** a **testRow** (třídy jsou obvykle nazývámy velkým písmenem na začátku, nyní byl název ovlivněn názvem tabulky "*test*" s malým písmenem na počátku). **testDataTable** popisuje celou tabulku *test*, **testRow** popisuje jeden řádek tabulky. Názvy tříd jsou samy vysvětlující.

Obrázek 7.5. Automaticky vygenerovaná třída DataSet1



DataSet je z databáze možno vygenerovat i druhým způsobem, a to pomocí XML schématu.

Pokud budeme chtít používat **DataSet**, musíme jej ještě před tím přes **DataAdapter** naplnit daty. Při napojení na databázový server zavoláme metodu **SqlDataAdapter.Fill(DataSet)**.

Ukázka kódu

```
...
myDataAdapter.Fill(myDataSet);
...
```

DataSet myDataSet můžeme naplnit i z více **DataAdapter**ů. Záleží na tom, jak (a zda) máme vytvořeno schéma databáze.

DataSet při získání dat z XML souboru

Předpokládejme nyní, že databázi máme uloženou v XML souboru. Abychom ji mohli načíst do prvku **DataSet**, je třeba pro ni nejprve vytvořit (či načíst) XML schéma umožňující při automatické generaci vytvořit podtřídy a metody potřebné pro práci s tabulkami.

Vytvoření XML schématu

Pro vytvoření nového schématu provedeme volbu z menu: *Project -> Add New Item* (popř. *Existing Item*, pokud schéma již máme někde vytvořeno).

Objeví se prázdná plocha s textem oznamujícím způsob vytvoření nového schématu. Zde můžeme postupným vkládáním elementů vytvořit strukturu podobně jako v databázi. Můžeme také vytvořit stejné schéma, jaké se nachází v naší databázi. Jednoduše přetáhneme ze *Server Explorer*u - volby *Data Connections* naše tabulky na plochu. Doplníme vazby mezi tabulkami, pokud je to nutné.

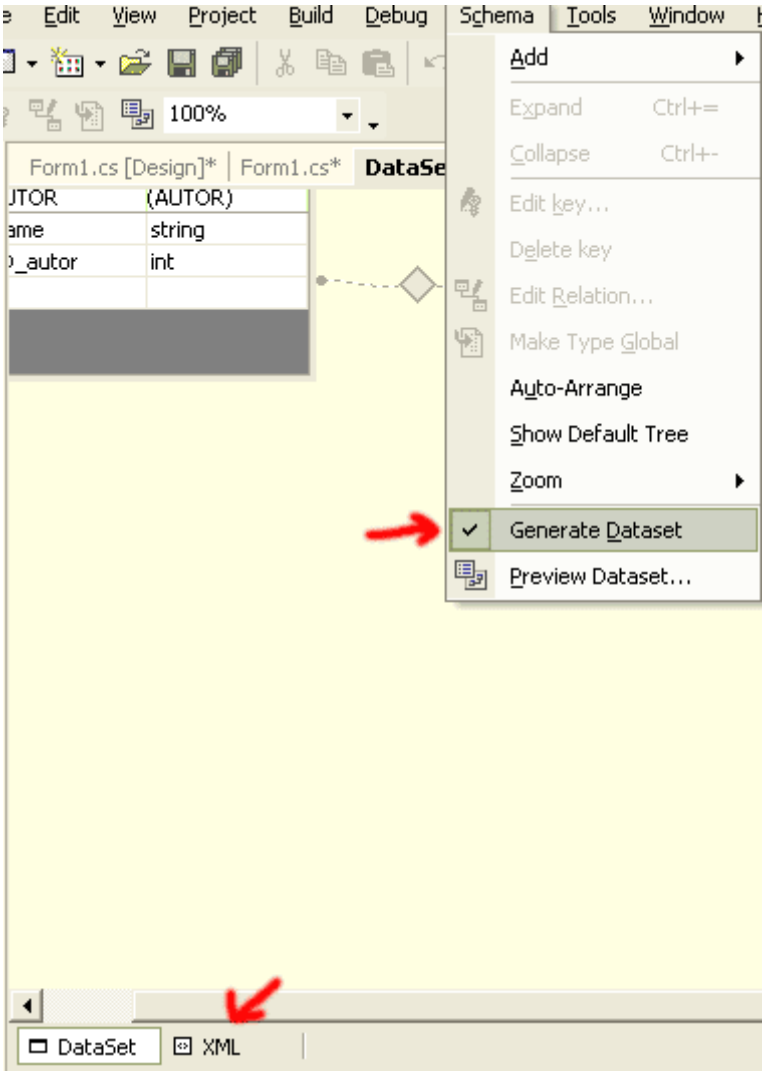
Obrázek 7.6. Databázové schéma

Pokud se nacházíme v okně s návrhem XML schématu, podobně jako u **SqlDataAdapteru** můžeme vygenerovat **DataSet**. Zde se to provádí zaškrtnutím volby z menu: *Schema -> Generate DataSet*. Tato volba nám při každém uložení schématu generuje **DataSet**.

Pokud jsme vytvořili XML schéma přetáhnutím tabulky ze *Server Explorer*u, vytvoříme shodný prvek **DataSet** jako v předchozím případě (pouze s jiným názvem).

Můžeme si zobrazit zdrojový XML kód poklepáním na záložku XML nacházející se pod oknem se schématem.

Obrázek 7.7. Možnost generace DataSet, záložka XML



Naplnění prvku DataSet z XML souboru

Nyní, když máme vytvořeno databázové schéma, můžeme bez problému načíst data. Protože při práci s XML daty reprezentujícími databáze nepotřebujeme **DataAdapter**, volá se metoda prvku **DataSet.ReadXml()**. Jako parametr této metody uvedeme název XML souboru s našimi daty.

Práce s daty v prvku DataSet

Nyní máme data načtena do prvku **DataSet**. V této chvíli s nimi můžeme zacházet, jak se nám zlíbí. Můžeme i použít přístup k datům jako u netypovaného prvku **DataSet**. Jak už ale bylo zmíněno výše, typovaný **DataSet** nachází ve Visual Studiu lepší podporu.

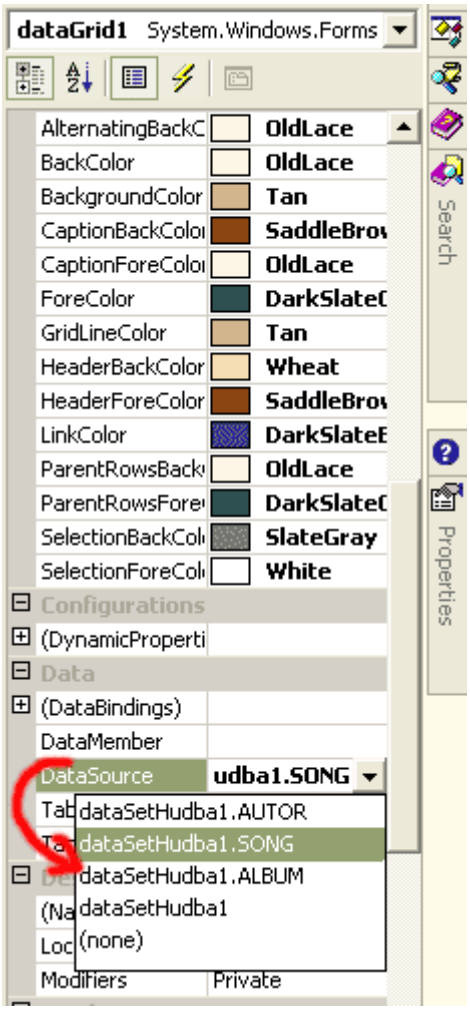
Vybírání (hledání) dat v DataSet

Je spousta prostředků, jak data z prvku **DataSet** vybrat. Můžeme vyhledat jednotlivou položku v celém **DataSet** nebo jej celý můžeme jednoduše zobrazit pomocí prvku **Windows.Forms.DataGrid**. Samozřejmě nyní hovoříme o grafickém režimu. V klasickém konzolovém výstupu bychom museli pravděpodobně většinu dat pracně vypisovat.

DataGrid

Pro hromadný výpis dat nám mimo jiné velice dobře poslouží nástroj **DataGrid**. Nalezneme jej opět v *Toolboxu* pod oddílem *Windows Forms*. Po přetáhnutí **DataGrid** do formuláře si zobrazíme jeho vlastnosti a zaměříme se v nich na oddíl *Data*, možnost *DataSource*.

Obrázek 7.8. DataSource u nástroje DataGrid



Pokud máme vytvořený **DataSet**, můžeme právě zde přiřadit, která data si **DataGrid** načte. Do nástroje **DataGrid** můžeme vložit samostatnou tabulku z prvku **DataSet** nebo celý **DataSet**. V druhém případě se nám při spuštění programu objeví v nástroji **DataGrid** nejprve políčko plus. Po rozbalení se objeví seznam tabulek, ve kterých se už naviguje pomocí odkazů. Pokud v jednotlivých tabulkách máme definovány vazby na jiné tabulky, objeví se tyto vazby u každého řádku tabulky (opět políčko plus).

Data uložená v **DataGrid**u můžeme různě upravovat i přidávat nová a mazat existující. Jelikož je náš **DataGrid** napojen přímo na náš **DataSet**, mohli bychom přímo změny takto vzniklé uložit do prvku **DataSet** a ty by se následně po uložení **DataSet**u projevily ve výsledné databázi.

Přístupy k jednotlivým datům

Určitě je zapotřebí získat i jednotlivá data vztahující se k určité položce. Pro tento případ slouží metoda **Select()** prvku **DataSet**. Jestliže máme typovaný **DataSet**, můžeme tuto metodu reprezentující SQL dotaz provést přímo na určitou tabulkou v prvku **DataSet**.

V příkladu je prvek **dataSetHudba1** naplněn třemi tabulkami *SONG*, *AUTOR*, *ALBUM*. Tabulky jsou spojeny vazbami *AUTORSONG*, *ALBUMSONG*. V názvech vazeb je na prvním místě tabulka rodičovská (v ní se nachází primární klíč) a na druhém potomek (pracuje se s cizím klíčem). Budeme například chtít vybrat autora písně s názvem *Chorale*.

Ukázka kódu

```
DataRow[] autor;
autor = dataSetHudba1.SONG.Select("name = 'Chorale' AND
Parent.(AUTORSONG).ID_autor=ID_autor");
```

```
// na konzole se zobrazi jmeno autora z databaze - Adiemus
System.Console.WriteLine(autor[0].GetParentRow("AUTORSONG")["name"].ToString());
```

S příkazem *SELECT* se jistě většina setkala, nebudeme se jím tedy vice zabývat. Více o něm v jazyce C# je možno se dočíst na stránkách http://msdn.microsoft.com [http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemdatadatatableclassselecttopic.asp].

Úprava dat v DataSet

Změna dat

Jak bylo ukázáno v porovnání mezi typovaným a netypovaným prvkem DataSet, lze přistupovat k jednotlivým položkám tabulek pomocí názvu tabulek, indexu řádky a názvu sloupce. Stejným způsobem lze data na tomto umístění i měnit. Existuje celá řada dalších způsobů, jejichž objevování nechám na každém zvlášť.

Smazání dat

Smazání dat lze provést podobně jako v předchozím případě při změně údajů pouze s tím rozdílem, že nyní mažeme celý řádek, tedy pokud budeme chtít smazat první řádek tabulky *SONG*, provedeme to pomocí metody **Delete()**.

Ukázka kódu

```
...
dataSetHudba1.SONG[0].Delete();
...
```

Uložení dat z prvku DataSet

Všechna data jsou nyní zpracována a aby zůstaly nové změny zachovány, v současné chvíli musíte uložit data v prvku **DataSet**. Ať již se jedná o uložení na databázový server či přímo do XML souboru, provádí se vše velice jednoduše.

Uložení na databázový server

Pokud používáme data z databázového serveru, mějme na paměti, že **DataSet** zde spolupracuje s **DataAdapter**em. **DataAdapter** naplnil náš **DataSet**. Také **DataAdapter** bude aktualizovat data v databázi. Mějme například **SqlDataAdapter SqlDataAdapterSong**, který se stará o práci s tabulkou *SONG*. My pracujeme s prvkem **DataSet** s názvem **dataSetHudba1**.

Ukázka kódu

```
SqlDataAdapterSong.Update(dataSetHudba1);
```

Uloženo.

Uložení do XML souboru

Pokud jsme data načítali z XML souboru a chceme je opět uložit do souboru (což není nezbytně nutné, tato data lze uložit v kterékoliv vhodné databázi), prvek **DataSet** nám poslouží i nyní. Stejně, jako jsme provedli načtení XML dat, provedeme i zápis. Opět předpokládejme **DataSet** s názvem **dataSetHudba1**.

Ukázka kódu

```
string filename = "music_collection.xml";
...
dataSetHudba1.WriteXml(filename);
```

Příklad ke stažení

Příklad na práci s prvkem DataSet s využitím XML [examples/XMLDataSet.zip].

Opakování

Příklady

K procvičení

Vytvořte databázové schéma podle příkladu z kapitoly 4 - 4 tabulky: *ZANR*, *KNIHA*, *VYPUJCKA*, *UZIVATEL*. Následně vytvořte vazby mezi tabulkami a vygenerujte **DataSet**. Podle návodu v této kapitole naplňte komponenty typu **DataGrid** a uložte **DataSet** do XML souboru. Takto získanou evidenci vypůjčených knih upravte podle vašich představ. Pokud budete mít příležitost, vyzkoušejte si i připojení a úpravu dat v databázi na serveru.

[řešení [exercises/6_ds.zip]]

Kapitola 8. XML

XML už na poli IT nějakou dobu působí. Ne každému je ale jasné, o co vlastně jde a k čemu je to dobré. Pokud i vy patříte k této skupině, můžete se v této kapitole dozvědět něco nového o tomto jazyce a různých způsobech jeho využití. V neposlední řadě se s XML naučíme pracovat prostřednictvím jazyka C#.

Stručný úvod do XML

Znalosti

Jazyk XML (eXtensible Markup Language) je *značkovací jazyk* odvozený od SGML (Standard Generalized Markup Language). U jazyka SGML se zjistitlo, že se v praxi používá jen část jeho možností. Proto se vybrala ta nejdůležitější množina jazyka SGML a dala vzniknout novému jazyku - XML.

O co se tedy jedná, když se vysloví termín XML? XML je jazyk navrhnutý tak, aby mohl uchovávat strukturovaná data. Vemme si například nějakou HTML stránku nějaké firmy, která zde prezentuje své výrobky či služby. Pro člověka jsou všechna data jednoduše čitelná. Pokud například uvidíme adresu firmy, poznáme, že se jedná o její adresu.

Ukázka kódu

```
...
<!-- zdrojovy kod HTML stranky nejake firmy -->
<p>
<b>Firma F</b><br>
Františka Nejedlého 12<br>
Kralupy nad Vltavou
</p>
...
```

Když si nějaký podobný kód zobrazíme v prohlížeči, název firmy "Firma F" se zobrazí tučně. Z toho můžeme usoudit, že se jedná o název firmy. Jak to ale pozná počítač? Značka <*b*> slouží pouze pro výstupní zvýraznění textu. Pro počítač je zde velice málo informací o tom, kde se nachází název firmy a kde adresa.

V jazyce XML bychom mohli adresu firmy popsat pomocí následujících značek.

Ukázka kódu

```
<?xml version="1.0" ?>
<address subject="company">
  <name>F</name>
  <street>Františka Nejedlého 12</street>
  <city>Kralupy nad Vltavou</city>
</address>
```

Takto strukturovaný dokument je pro počítač mnohem snáze čitelný - ví přesně, kde co hledat. Díky tomu je možno v XML umístit velké množství informací.

V minulé kapitole jsme si také ukázali, že pomocí XML je možno strukturovat data dat tak, aby byla použitelná při databázovém zpracování. Jazyk XML má samozřejmě další možnosti. Ať už se jedná o výměnu dat mezi pobočkami firem (*business-to-business applications*), webové stránky, elektronické publikování (tyto výukové materiály jsou psány v XML s DTD DocBook) či úložiště dat pro programy.

DTD a kontrola struktury dokumentu

V XML dokumentu můžeme použít jakékoliv značky. Samozřejmě můžeme jejich používání omezit na předem definované značky. To se provádí pomocí *DTD* - Document Type Definition (Definice Typu Dokumentu). Pak můžeme kontrolovat, zda dokument vyhovuje naší definici a zda ho tedy můžeme zpracovat podle našich požadavků. Kontrola správnosti struktury dokumentu se provádí pomocí tzv. *parseru*.

Samozřejmě, pokud si každý programátor na světě vytvoří svůj vlastní DTD, pak by asi výměna a zpracování XML dat neměly velký smysl. Proto existují různé skupiny a sdružení vydávající DTD, které jsou doporučeny pro užívání v dané oblasti.

Tvorba XML dokumentu

V odstavci 1 této kapitoly jsme si naznačili, jak může vypadat část XML dokumentu. Nyní si osvětlíme některé detaily. Každý dokument by měl být uvozen hlavičkou oznamující, o jaký typ dokumentu se jedná. V případě XML by tato hlavička měla mít tvar

Ukázka kódu

```
<?xml version="1.0" encoding="typ_pouziteho_kodovani"?>
```

Následuje samotné tělo dokumentu.

Dokument se skládá z elementů, jejichž názvy jsou ohraničeny ostrými závorkami - <*element*>. Obecně platí, že se zde musí nacházet kořenový element, do kterého se vkládají další elementy. Každý element musí být označen začátkem a koncem, tedy <*element*> na začátku elementu a </*element*> na konci. Elementy se rovněž nesmějí křížit.

Ukázka kódu

```
...
<narozeni>
<den>21</den>
<mesic>8</mesic>
<rok>1968</rok>
</narozeni>
...

...
takto by neměl vypadat <b>bold <i>a italic</b></i> text.
...
```

Jednoduchý dokument obsahující recepty z kuchařky (v našem případě pouze na černý čaj) může být takto vytvořen:

Ukázka kódu

```
<?xml version="1.0" encoding="UTF-8"?>
<kucharka>
  <recept nazev="Černý čaj s citronem" priprava="5">
    <ingredience nazev="čaj" mnozstvi="1" jednotka="sáček"/>
    <ingredience nazev="citron" mnozstvi="1" jednotka="plátek"/>
    <ingredience nazev="cukr" mnozstvi="2" jednotka="čajová lžička"/>
    <ingredience nazev="voda" mnozstvi="0.4" jednotka="l"/>
    <postup>Přivedeme vodu k varu. Mezitím si nachystáme hrnek, do něj vložíme
sáček čaje.
    Jakmile voda zavaří, vlijeme ji do hrnku. Necháme 5 minut louhovat a poté
sáček vyjmeme.
    Přidáme cukr a citron.</postup>
  </recept>
</kucharka>
```

Jmenný prostor System.Xml

Jmenný prostor **System.Xml** nabízí mimo jiné podporu pro

XML 1.0

XSD schémata

výrazy XPath

transformace XSLT

System.Xml obsahuje velké množství tříd a rozhraní pracujících s těmito prvky. Budeme popisovat jen ty, které použijeme pro práci s XML souborem na příkladu. Informace o ostatních třídách jsou k dispozici na http://msdn.microsoft.com.

Práce s XML souborem

Načítání z XML souboru

Vezměme si příklad, ve kterém budeme chtít vygenerovat HTML stránku s recepty uloženými v XML souboru. Data mají strukturu jako v kuchařce ze sekce Tvorba dokumentu. Data z XML souboru lze načíst opět různými způsoby. my použijeme třídu **XmlTextReader**, která umožňuje jednosměrné načítání dat. Aby tato data byla uchována v rozumné odpovídající struktuře, nabízí C# třídu **XmlDocument**.

Ukázka kódu

```
using System.Xml;
...
XmlDocument myXml;

// xmlReader bude nacitat data ze souboru kucharka.xml
XmlTextReader xmlReader = new XmlTextReader("kucharka.xml");
myXml.Load(xmlReader);
...
```

Zpracování načtených dat

Nyní, když máme data načtená, můžeme s nimi pracovat.

Nejprve se ale podívejme na některé metody přístupu k datům v **XmlDocument**.

Metody přístupu k datům objektu XmlDocument

Tak, jak je dokument strukturován v XML souboru, můžeme přistupovat k jednotlivým elementům či atributům i my. Budeme chtít získat hodnotu elementu uvedeného v následujícím výpise XML souboru.

Ukázka kódu

```
<?xml version="1.0" encoding="UTF-8"?>
<kucharka>
  <recept nazev="testovaci krme">
    <postup>---TENTO TEXT CHCEME ZISKAT---</postup>
  </recept>
</kucharka>
```

Vidíme v něm, že se zde nachází deklarace, následuje první - kořenový - element *kucharka*, následuje element-potomek *recept* a ten má element-potomka *postup*. Pokud chceme získat hodnotu elementu *postup* právě na této úrovni, v C# to provedeme následovně. XML soubor již máme načtený v instanci třídy **XmlDocument** s názvem **myXml**.

Ukázka kódu

```
myXml.ChildNodes[1].ChildNodes[0].ChildNodes[0].InnerText;
```

Vlastnost **ChildNodes** vrací uzly-potomky předchozího uzlu. Operace **myXml.ChildNodes[1]** nám tedy vrátí uzel *kucharka*. Postupným zanořováním do stromu se dostaneme až k hledané hodnotě.

V praxi se ovšem stane, že nevíme, zda uzel, na který právě ukazujeme, je ten náš - tedy uzel *postup*. Naštěstí stačí ověřit, zda souhlasí jméno uzlu.

Ukázka kódu

```
if(myXml.ChildNodes[1].ChildNodes[0].ChildNodes[0].Name == "postup")

System.Console.WriteLine(myXml.ChildNodes[1].ChildNodes[0].ChildNodes[0].InnerTex
t);
```

Díky podobným konstrukcím můžeme vytvořit libovolně hluboké zanoření do stromu XML dokumentu.

Generování HTML kódu ze souboru kucharka.xml

V předchozím odstavci jsme si objasnili základní metody přístupu k uzlům objektu **XmlDocument**. Nyní si vygenerujeme kuchařku tak, aby byla zobrazitelná a snadno čitelná v internetovém prohlížeči. Postupně si budeme vypisovat jednotlivé úseky metody **Main()** ve třídě **XMLCtecka**. Tato třída obsahuje kód pro generování HTML kódu.

Pro objasnění - metoda **ZapisText(string text)** do výsledného HTML souboru zapisuje jednotlivé řádky.

Ukázka kódu

```
XMLCtecka ctecka = new XMLCtecka();
...
foreach(XmlNode receptNode in mainNode) {
  if(receptNode.Name == "recept") {
    // zapise nazev receptu
    ctecka.ZapisText("<h2>" + receptNode.Attributes["nazev"].Value + "</h2>");
    ctecka.ZapisText("\n<b>Potřebné přísady:</b>\n<ul>");

    // zacne vypisovat jednotlive prisady
    foreach(XmlNode node in receptNode) {
        // muze zde byt i uzel postup
        if(node.Name == "ingredience") {
            ctecka.ZapisText("\n<li>");
            // vysledek bude ve tvaru napr.: 2 x lzice - olej
            ctecka.ZapisText(node.Attributes["mnozstvi"].Value + " x ");
            ctecka.ZapisText(node.Attributes["jednotka"].Value + " - ");
            ctecka.ZapisText(node.Attributes["nazev"].Value + "</li>");
        }
    }
    ctecka.ZapisText("</ul><br>");

    // nasleduje vypsani postupu vareni
    foreach(XmlNode node in receptNode) {
        if(node.Name == "postup") {
            ctecka.ZapisText("\n<div class='postup'><b>Postup:</b><br>");
            ctecka.ZapisText(node.InnerText + "</div>");
        }
    }
    ...
}
```

Příklad ke stažení

Zde [examples/4.cs] se nachází příklad na výše uvedené získávání elementů z XML souboru a generování kuchařky. Zde [examples/kucharka.xml] se nachází XML dokument kucharka.xml

Zapsání dat do XML souboru

Podobně, jako jsme data z XML souboru načítali pomocí **XmlTextReader**u, data určená pro zápis do XML souboru budeme zapisovat s pomocí třídy **XmlTextWriter**. V této třídě budou pro základní práci s XML souborem užitečné zejména metody

XmlTextWriter.WriteStartDocument()

Zapíše na daný výstup deklaraci XML dokumentu

XmlTextWriter.WriteStartElement()

Zapíše na daný výstup element

XmlTextWriter.WriteAttributeString()

Zapíše na daný výstup k elementu naposled připojenému určený atribut

XmlTextWriter.WriteEndElement()

Zapíše na daný výstup koncovou značku pro tag naposledy vypsaný pomocí **WriteStartElement()**

Následující kód zapíše na konzolu data jako XML dokument.

Ukázka kódu

```
using System.Xml;
...
  XmlTextWriter tw = new XmlTextWriter(System.Console.Out);

  tw.WriteStartDocument();
    tw.WriteStartElement("birthday");
    tw.WriteStartElement("day");
      tw.WriteString("22");
    tw.WriteEndElement();
    tw.WriteStartElement("month");
      tw.WriteString("12");
    tw.WriteEndElement();
    tw.WriteStartElement("year");
      tw.WriteString("1979");
    tw.WriteEndElement();
  tw.WriteEndElement();
...
```

XPath

Jmenný prostor **System.Xml.XPath** obsahuje *XPath* parser a výpočetní engine. Podporuje W3C XML Path Language (XPath).

Co je to XPath

Znalosti

XPath je jazyk používaný pro identifikaci uzlů v XML dokumentu. Pravděpodobně nejdůležitějším rysem jazyka XPath je možnost vyjádření relativní cesty od uzlu k jinému uzlu či atributu. Připomíná dotazovací jazyk SQL, zejména dík tomu, že na základě podmínky vrátí jeden nebo množinu výrazů (nebo žádný) odpovídající vstupní podmínce.

Podobně jako při procházení adresářové struktury se ve struktuře XML dokumentu dá procházet pomocí lomítek. Vemme například naši kuchařku. Pomocí tříd jmenného prostoru **System.Xml** jsme generovali HTML kód. Používali jsme přitom metody a vlastnosti třídy **XmlDocument**. Pro každý element, do kterého jsme se dostali, bylo nutno testovat jeho název, abychom měli jistotu, že pracujeme právě s ním. Díky jazyku XPath se tyto podmínky stávají jednoduššími. Pokud se totiž pomocí XPath budeme chtít dostat například do uzlu postup, stačí nám k tomu výraz

Ukázka kódu

```
/kucharka/recept/postup
```

Stručně si popíšeme význam některých symbolů a způsob získávání elementů, pro podrobnější popis doporučuji stránky Jiřího Koska (viz odkaz na Výrazy XPath [http://www.kosek.cz/xml/xslt/vyrazy.html]).

Tabulka 8.1. Výběr z XPath

rodič/potomek	Pokud lomítko umístíme na začátek výrazu, jako počáteční uzel se bere kořen dokumentu. Jinak se nalezne uzel potomek v uzlu rodič aktuálního uzlu
rodič//potomek	Potomek nemusí být přímým následníkem rodiče - může se vyskytovat hlouběji ve struktuře. Pokud se nachází dvě lomítka na začátku výrazu (//rodič), naleznou se všechny výskyty elementu rodič v celém dokumentu
rodič/*	vybere všechny uzly uzlu rodič v aktuálním uzlu
.	vybere aktuální uzel
..	pohyb na element o úroveň výše
@postup	vybere atribut (zde atribut postup) aktuálního uzlu
[ ]	podmínka
rodič/potomek[@atribut="3"]	vybere všechny elementy potomek z elementu rodič aktuálního uzlu, jejichž atributy atribut odpovídají podmínce (zde jsou tedy rovny 3)

XPath umožňuje rovněž použití funkcí. Zkráceně si některé vypíšeme:

position(), **last()**, **count()**

vrací pozici aktuálního uzlu, posledního uzlu, počet uzlů v daném kontextu

name(), **local-name()**, **namespace-uri()**,

úplné jméno, název aktuálního uzlu, název jmenného prostoru

string(), **number()**, **boolean()**

převod objektu na typ **string**, **number**, **boolean**

contains(), **substring-before()**, **substring-after()**, **string-length()**...

práce s řetězci

Využití XPath v C# a jmenném prostoru **System.Xml.XPath** si ukážeme na příkladech.

Způsob získání dat pomocí XPath

Jak jsme si řekli, bude o něco jednodušší při hledání informací v XML dokumentu používat XPath. V podstatě budeme pro hledání výrazů používat tři základní třídy - **XPathDocument**, **XPathNavigator** a **XPathNodeIterator**.

XPathDocument slouží jako cache pro načítání dat z dokumentu, **XPathNavigator** v sobě zahrnuje metody důležité pro implementaci dotazů XPath. **XPathNavigator** lze vytvořit ze tříd **XPathDocument** a **XmlNode** voláním metody **CreateNavigator()**. Obě třídy totiž implementují rozhraní **IXPathNavigable**.

XPathIterator slouží k procházení uzlů vyhovujících vstupní podmínce zadané v instanci třídy **XPathNavigator** nebo z jiné instance **XPathIterator**.

Nyní již k samotným příkladům.

Zůstaneme stále u kuchařky. Nejprve vytvoříme **XPathDocument**, následně mu přiřadíme **XPathNavigator** a pokusíme se nalézt jména všech přísad, které jsou zapotřebí pro všechny recepty s použitím XPath. Stačí provést dotaz "*/kucharka/recept/ingredience/@nazev*". Pokud budeme chtít zjistit, jaké ingredience to jsou, napíšeme následující kód.

Ukázka kódu

```
using System.Xml.Xpath;
...
XPathDocument myXpathDoc = new XPathDocument("kucharka.xml");
XPathNavigator myNav = myXPath.Doc.CreateNavigator();
```

```
XPathNodeIterator xi = myNav.Select("/kucharka/recept/ingredience/@nazev");
while (xi.MoveNext()) {
  Console.WriteLine(xi.Current.Value);
}
...
```

Metoda **MoveNext()** objektu **XPathNodeIterator** se musí volat vždy, abychom měli jistotu, že pracujeme skutečně s uzly, které jsme zadali v dotazu. Jinak bychom mohli dojít k jiným výsledkům, než jsme očekávali.

Stejně jako jsme položili dotaz na instanci třídy **XPathNavigator**, můžeme vytvářet dotazy i na jednotlivé instance **XPathNodeIterator**. Lze je vytvořit nad aktuálním uzlem získaným s pomocí vlastnosti **Current**.

Příklad ke stažení

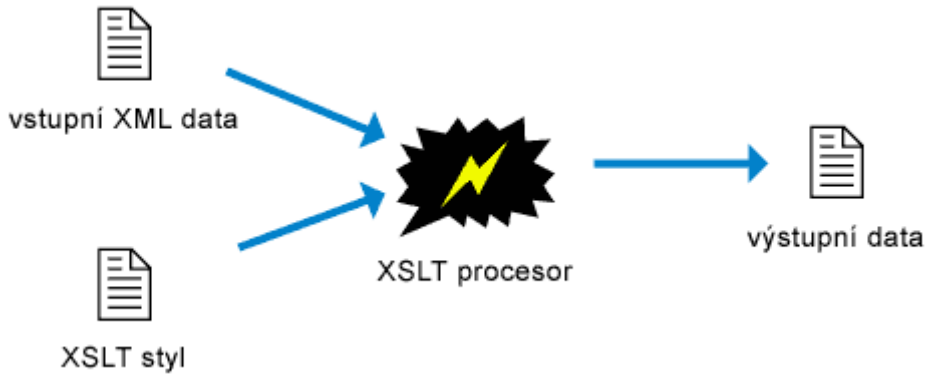
Kompletní řešení generování HTML kódu kuchařky pomocí výrazů XPath je ke shlédnutí zde [examples/5_2.cs].

XSLT

Po seznámení s XML dokumenty a jazykem XPath se podíváme na další způsob, jak lze transformovat XML dokument jiným způsobem než ono generování programovat přímo v nějakém obecném programovacím jazyce.

Speciálním jazykem pro transformaci XML dokumentů na jinou formu (např. HTML, PDF, PS apod) je právě jazyk *XSLT* (eXtensible Stylesheet Language Transformations). XSLT lze vyjádřit následujícím obrázkem

Obrázek 8.1. Transformace XML na uživatelský formát



Tedy k tomu, aby bylo možno XML dokument transformovat, je zapotřebí k XML dokumentu vytvořit navíc *XSLT styl*. Ten XSLT procesoru říká, jakým způsobem má jednotlivé značky v XML dokumentu transformovat na výstupní značky. Existují různé implementace XSLT procesorů (Saxon, Xalan, XT...), jelikož ale procházíme kurzem C#, zaměříme se na procesor dostupný v .NET Framework.

Tvorba XSLT Stylesheets

Styly pro generování z XML dokumentů se opět píší s použitím XML. Pokud chceme použít pro zpracování nějaký styl, obecně se do XML dokumentu zařazuje odpovídající značka hned za deklaraci dokumentu.

Ukázka kódu

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet href="styl.xsl" type="text/xsl"?>
...
```

K jednomu XML dokumentu lze připojit i více stylů.

V prostředí .NET Framework toto můžeme ipmlementovat také, nicméně zpracování stylu funguje na principu načtení XML dokumentu, načtení XSLT stylu a následně provedení transformace.

Pojďme se ale nyní podívat, jak se v praxi vytváří takový XSLT styl. Budeme stále věrní kuchařce. Kód, který transformuje vstupní XML data na HTML kód nyní vypadá takto:

Ukázka kódu

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <html>
      <head>
        <title>Kuchařka</title>
      </head>
    <body>
    <xsl:for-each select="/kucharka/recept">
      <h2><xsl:value-of select="./@nazev"/></h2>
      <ul>
        <xsl:for-each select="./ingredience">
          <li>
            <xsl:value-of select="./@nazev"/> : <xsl:value-of
select="./@mnozstvi"/> x
            <xsl:value-of select="./@jednotka"/>
          </li>
        </xsl:for-each>
      </ul>
      <p><xsl:value-of select="./postup"/></p>
    </xsl:for-each>
    </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

Vidíme zde, že je v XSLT stylech možno používat jakékoliv značky. Ty, které jsou určeny pro zpracování XSLT parserem, mají předponu *xsl*. Pomocí výrazů jazyka XPath si navíc jednoduše zvolíme, jaký element budeme právě potřebovat a zpracujeme jej.

Nebudu se zde rozsáhle rozepisovat o možnostech XSLT, kdo bude mít zájem o větší detaily doporučuji odkazy www.w3c.org/style/XSL [http://www.w3c.org/Style/XSL/] a stránky Jiřího Koska XSLT v příkladech [http://www.kosek.cz/xml/xslt/]. My si osvětlíme základy používání XSLT.

Základy XSLT

Jako každý XML dokument, i XSLT soubor by měl kromě deklarace obsahovat i kořenový element. V případě XSLT je kořenovým elementem *xsl:stylesheet*. V tomto elementu se uvádí jmenný prostor, který používá XSLT, případně verze XSLT. My u XSLT budeme používat jmenný prostor *http://www.w3c.org/1999/XSL/Transform*.

Šablony (templates)

Základ transformace XSL tvoří *šablony* (templates). Ty popisují, jak se který element má měnit. Následující ukázka šablony vybere kořenový element (tedy celý dokument). Pro výběr elementů se tedy používá atribut **match** a jako výraz se zadá cesta k uzlu.

Ukázka kódu

```
...
<xsl:template match="/">
...
</xsl:template>
...
```

Do šablon je možno vkládat jiné šablony pomocí **xsl:apply-templates**. Pokud do tohoto příkazu jako atribut vložíme **select** a hodnotu elementu, který chceme zpracovat, provede se právě tento. Pokud atribut **select** nepoužijeme, provedou se všechny šablony, které vyhovují nalezeným elementům.

Někdy se může stát, že budeme potřebovat pro jednu šablonu více způsobů zpracování. Provádíme to pomocí tzv. *módů šablony*.

Ukázka kódu

```
...
<xsl:apply-templates select="ingredience" mode="LI"/>
...
<xsl:template match="ingredience" mode="LI">
  <li><xsl:value-of select="@nazev"/></li>
</xsl:template>
```

XSL transformace se dá zapsat i bez většího použití šablon, podobně jako v úvodním příkladu. Pokud ale budeme chtít transformovat rozsáhlejší dokument, pravděpobně se bez šablon neobejdeme. S šablonami je navíc výsledný kód přehlednější.

Výběr hodnot

Příkazem pro zobrazení dat je **xsl:value-of**. Slouží pro získání hodnoty konkrétního elementu. V jeho atributu **select** se pak pomocí výrazů XPath zadá cesta k elementu. Následující příklad zjistí hodnotu atributu **nazev** aktuálního uzlu. Výraz atributu **select** lze také zapsat pouze ve tvaru **@nazev**.

Ukázka kódu

```
...
<xsl:value-of select="./@nazev"/>
...
```

Tvorba elementů, atributů

Budeme chtít vytvořit nový element. Například chceme vytvořit odkaz na nějaké místo a jako hodnotu vzít hodnotu existujícího atributu. Dá se to provést následujícím způsobem - pomocí příkazu **xsl:element** vytvoříme nový element a k němu vytvoříme příkazem **xsl:attribute** potřebné atributy.

Ukázka kódu

```
...
<xsl:element name="a">
  <xsl:attribute name="href">www.w3c.org</xsl:attribute>
  World Wide Web Consortium
</xsl:element>
...
```

Kód výše vytvoří následující výstup:

```
<a href="www.w3c.org">World Wide Web Consortium</a>
```

Iterativní zpracování

Uvnitř šablon lze použít příkaz **xsl:for-each**. U tohoto příkazu se v atributu select udává uzel, který má být zpracováván. Viz úvodní příklad - tam jsme pomocí tohoto příkazu zpracovávali jak jednotlivé recepty, tak i přísady vztahující se ke každému receptu.

Zpracování XSLT v C#

Pokud máme vytvořen XML dokument a k němu styl, který k němu chceme připojit, zpracování v C# už je velmi jednoduché. Vytvoříme si instanci třídy **System.Xml.Xsl.XslTransform**. Vzápětí načteme styl, kterým chceme XML dokument zpracovat a zavoláme metodu **System.Xml.Xsl.XslTransform.Transform()**. Jejími argumenty jsou vstupní soubor XML a výstupní soubor.

Ukázka kódu

```
...
System.Xml.Xsl.XslTransform xt = new XslTransform();
xt.Load("style.xsl");

xt.Transform("kucharka.xml", "kucharka.html");
```

. . .

Příklad ke stažení

Příklad s XSLT kompletní příklad se zdrojovým kódem, XML dokumentem *kucharka.xml* a dvěma styly s různým způsobem formátování naleznete zde [examples/XSL.ZIP].

Kapitola 9. Bezpečnost a zabezpečení

V souvislosti se zapojením počítačů do počítačové sítě je na místě být velmi opatrný, ať se týká brouzdání po Internetu nebo otevírání souborů stažených ze sítě celkem. Je také důležité mít svá data zabezpečená. Kapitola Bezpečnost a zabezpečení nastíní některé techniky kódování, bezpečné komunikace a probere způsoby ochrany před škodlivým kódem a před neoprávněným přístupem cizí osoby.

Při práci s počítačovými daty a aplikacemi hrozí celá škála útoků na choulostivá data. Je známa řada způsobů, jak je možno zjistit obsah komunikace mezi vzdálenými počítači, nebo jak získat například přístupová hesla k osobním účtům u bank přes internet. Více o bezpečnosti a kryptografii pojednává předmět vyučovaný (mimo jiné) na VŠB-TU, katedře informatiky s názvem Kryptografie a počítačová bezpečnost (KPB).

My se zaměříme na způsoby, jak lze v praxi realizovat zabezpečení našich dat. Ukážeme si postupy kódování choulostivých dat (známý, ale dnes již uznaný jako slabý - 128 bitový hashovací algoritmus *MD5*, často používaný 160 bitový *SHA-1* apod.) sloužící pro *digitální podepisování*. Podíváme se také na třídy pracující s *certifikáty*.

Ukážeme si také, jak lze zamezit přístup k souborům či omezit k nim přístupová práva.

Kódování a hashování

Microsoft .NET obsahuje třídy rozšiřující kryptografické služby nabízené Windows CryptoAPI. Tyto třídy se nachází v jmenném prostoru **System.Security.Cryptography** a zahrnují

Symetrické kódování (kódování se symetrickým klíčem)

Asymetrické kódování (s asymetrickým klíčem)

Hashování (hashovací funkce)

Digitální certifikáty

XML podpisy

Symetrické kódování

Znalosti

Algoritmy využívající symetrické kódování jsou velice rychlé a hodí se pro kódování velkého množství dat. Tyto algoritmy mohou data jak kódovat, tak i dekódovat. I když jsou vcelku bezpečné, pokud je k dispozici dost času, může být jejich šifra prolomena.

Symetrické kódování je v .NET implementováno pomocí tříúrovňového dědění. Na první úrovni se nachází abstraktní třída **SymmetricAlgorithm** a specifikuje, že se jedná o symetrický algoritmus. Na následující (druhé) úrovni se nacházejí třídy pojmenované po patřičném typu symetrického algoritmu. Třídy na druhé úrovni jsou rovněž abstraktní. Na poslední (třetí) úrovni se nachází kompletní implementace symetrických algoritmů. Je realizována buď s použitím tzv. poskytovatelů kryptografických služeb ve Windows CryptoAPI (*Cryptographic Service Providers*) nebo vlastní implementací v .NET.

Symetrické algoritmy v .NET jsou pojmenovány podle toho, zda jsou realizovány s pomocí Windows a jejich poskytovatelem kryptografických služeb nebo mají vlastní implementaci v .NET. Přípona algoritmu označuje jejich způsob realizace.

RC2CryptoServiceProvider

DESCryptoServiceProvider

TrippleDESCryptoServiceProvider

RijndaelManaged

Když dojde k vytovření instance konkrétního algoritmu, její konstruktor vytvoří silný tajný klíč a nastaví všechny hodnoty potřebné pro kódování.

Následující příklad načte ze souboru text, který s pomocí symetrického algoritmu *DES* zakóduje do výstupního souboru *DESencrypted.txt*.

Ukázka kódu

```
using System.Security.Cryptography;
...
   string filename;
   // nactu pole bytu z vytvorenych funkci
   byte[] byteArrayInput = StringToByteArray(GetStringFromFile(filename));

   // DES algoritmus
   DESCryptoServiceProvider des = new DESCryptoServiceProvider();

   // rozhrani pro zakladni operace s kryptografickymi transformacemi
   // vytvoreni koderu
   ICryptoTransform desEncrypt = des.CreateEncryptor();

   // vytvori se kryptovaci stream transformujici soubor s danou krypt. metodou
   CryptoStream cryptostream = new CryptoStream(
   new FileStream("DESencrypted.txt", FileMode.Create, FileAccess.Write),
      desEncrypt,
      CryptoStreamMode.Write);

   // zapise se zakodovany soubor
   cryptostream.Write(byteArrayInput, 0, byteArrayInput.Length);
   cryptostream.Close();
...
```

Asymetrické kódování

Asymetrické algoritmy nejsou tak rychlé jako symetrické, ale jsou o mnoho bezpečnější. Tyto algoritmy spoléhají na dva klíče - veřejný a soukromý. Veřejný klíč se používá ke kódování zprávy, zatímco soukromý je použit pro dekódování zprávy. Jelikož nejsou rychlé, nejsou asymetrické algoritmy příliš vhodné pro přenos většího množství dat. Jeden klasický příklad použití asymetrického kódování je zakódovat a přenést druhé straně symetrický klíč a inicializační vektor (IV). Pak se přenos provádí s použitím symetrického kódování.

Asymetrické má podobně jako symetrické kódování také tříúrovňovou strukturu dědičnosti. Pokud se vytvoří instance konkrétního asymetrického algoritmu, konstruktor vždy vygeneruje náhodné páry klíčů (silné hodnoty klíčů). Asymetrické algoritmy mohou být také vytvořeny tak, aby mohly získat páry klíčů z kontejneru klíčů *Crypto Service Provideru*.

Asymetrické algoritmy (názvy tříd) implementované v .NET:

RSACryptoServiceProvider

RSA

DSACryptoServiceProvider

DSA

Následujcí příklad zakóduje vstupní text s použitím algoritmu *RSA*

Ukázka kódu

```
using System.Security.Cryptography;
...
   byte[] rsaByteArray;

   RSACryptoServiceProvider rsa = new RSACryptoServiceProvider();
   // do rsaByteArray se ulozi kodovany obsah vstupiho (nekodovaneho) pole typu
byte
   rsaByteArray = rsa.Encrypt(byteArrayInput, false);
```

...

Hashování

Znalosti

Hash hodnoty (nazývané také výtažky zpráv, otisky zpráv, obsahy zpráv) se používají všude tam, kde si nepřejeme získat původní hodnotu zprávy. Tedy i nechceme po nikom jiném, aby ji získal.

Hashovací funkce vemou řetězec libovolné délky a transformují jej na bytový řetězec s pevnou délkou. Jelikož je tento způsob kódování jednocestný, používá se například ke kódování hesel, tedy malých množství dat. Heslo, které napíše uživatel, se zpracuje pomocí některé hash funkce a uloží do databáze. Pokud se chce uživatel přihlásit, napíše své heslo, to se opět zpracuje pomocí hash funkce a porovná s hodnotou uloženou v databázi. Tento způsob je výhodný zejména proto, když dojde k neoprávněnému přístupu do databáze - všechna hesla jsou již pomocí hash funkce kódována, tedy je velice obtížné (i když ne úplně nemožné) zjistit původní heslo.

Hashovací funkce rozlišujeme buď s privátním klíčem nebo bez klíče. Pokud používáme hashovací funkci s klíčem, je třeba tento klíč znát i pro ověření hodnoty vzniklé hashovací funkcí s klíčem.

Typy hashovacích funkci (názvy tříd) bez klíče:

MD5CodeServiceProvider

SHA1CodeServiceProvider

SHA1Managed

SHA256Managed

SHA384Managed

SHA512Managed

Typy hashovacích funkcí (názvy tříd) s klíčem:

HMACSHA1

HMAC s použitím algoritmu *SHA-1*

MACTripleDES

MAC s použitím algoritmu *TripleDES*

Následující příklad zakóduje řetězec '*krokodýl*'

Ukázka kódu

```
using System;
using System.Text;
using System.Security.Cryptography;
...
   byte[] byteOutput;
   // hashovaci funkce SHA-1 je ziskana ze tridy SHA1CryptoServiceProvider
   SHA1CryptoServiceProvider sha1 = new SHA1CryptoServiceProvider();
   // prevedeme retezec na pole bytu a vratime ho hashovaci funkci
   byteOutput = sha1.ComputeHash(new ASCIIEncoding.GetBytes("krokodyl"));

   // cyklus vytiskne na obrazovku tuto hash hodnotu:
   // 551832f80e3c53545581aa1e833e52bdb177ef
   foreach(byte b in byteOutput) {
      Console.Write("{0:x}", b);
   }
...
```

Digitální obálky, podpisy a certifikáty

Digitální obálky (Digital Envelopes)

Znalosti

Digitální obálka je aplikace, ve které odesílatel zapečetí zprávu a nikdo jiný než zamýšlený příjemce zprávy ji nemůže otevřít. Toto se děje s pomocí symetrických (obsah zprávy) i asymetrických kryptovacích algoritmů (kódování tajného klíče).

Odesílatel (označován jako A - Alice) zakóduje čistý text příjemcovým veřejným klíčem s použitím nějakého asymetrického algoritmu a pošle zprávu příjemci (označován jako B - Bob). Tato zpráva může být přijemcem otevřena pouze tehdy, pokud přijemce zná svůj tajný klíč (náležející k veřejnému klíči příjemce.

Jak bylo řečeno výše, asymetrické algoritmy jsou pomalejší (uvádí se dokonce, že až 1000x) a nejsou tedy vhodné pro přenos velkého množství dat. Asymetrickým algoritmem se proto zašifruje tajný klíč, a tento tajný klíč je poslán příjemci. Jelikož obě strany znají tajný klíč (odesílatel jej znal ze začátku, příjemce jej získal dekódováním první zprávy svým tajným klíčem), může se nyní komunikovat s pomocí symetrického algoritmu.

Pro vytvoření digitální obálky se obvykle používá algoritmus *RSA* (třída **RSACryptoServiceProvider**).

Digitální podepisování (Digital Signing)

Znalosti

Digitální podpis je aplikace, ve které odesílatel podepíše zprávu tak, že každý může ověřit, zda je od daného odesílatele a že nikdo jiný ji nezměnil poté, co byla odesílatelem podepsána. K digitálním podpisům se používají hashovací funkce a algoritmy s asymetrickým kódováním.

Pro digitální podepisování lze použít algoritmy *RSA*, *DSA*.

Digitální certifikáty

Předchozí aplikace mají několik nedostatků

Jak se příjemce zprávy dozví odesílatelův veřejný klíč a jak si může být jist jeho pravostí?

Jak se příjemce dozví, jaké použít hashovací funkce a jaké jsou jejich parametry?

Jak se příjemce dozví, které kódovací algoritmy použít a jaké jsou jejich parametry?

Tyto nedostatky řeší právě digitální certifikáty. Digitální certifikace je aplikace, ve které certifikační autorita podepíše zvláštní zprávu obsahující jméno uživatele a uživatelův veřejný klíč. Tato zpráva obsahuje také použité algoritmy a jejich parametry. Navíc si každý může zjistit, že tato zpráva byla podepsána právě danou certifikační autoritou. Tato zpráva se nazývá digitální certifikát.

Ukázka kódu

```
using System;
using System.Security.Cryptography.X509Certficates;
...
   string certFile;
   ...
   try {
      // nacteme certifikat ze souboru
      X509Certificate cert = X509Certificate.CreateFromCertFile(certFile);

      // vypiseme informace ziskane z certifikatu
      Console.WriteLine("Nazev: "
        + cert.GetName().Substring(cert.GetName().IndexOf("CN=")+3));
      Console.WriteLine("Vystavil: "
        + cert.GetIssuerName().Substring(cert.GetIssuerName().IndexOf("OU=")+3));
      Console.WriteLine("Platnost: "
        + cert.GetEffectiveDateString() + " - "
        + cert.GetExpirationDateString());
   } catch (System.Security.Cryptography.CryptographicException e) {
```

```
        Console.WriteLine(e.Message);
    }
...
```

Příklad ke stažení

Testovací program na výpis obsahu certifkátu společně s certifkátem je možno možno ve fromátu *zip* stáhnout zde [examples/2_3.ZIP].

Způsoby ochrany prostředků a kódů

Prostředí .NET nabízí dva způsoby ochrany prostředků a kódů před neautorizovanými kódy a uživateli.

Chráněný přístup ke zdrojovým kódům

Přístup mezi kódy a síťovými zdroji a operacemi je ovládán pomocí povolení a zákazů (permissions)

Bezpečnost založená na rolích (*Role-based security*)

Přináší informace potřebné k rozhodnutí, zda uživatel bude mít přístup k datům či prostředkům nebo ne. Zakládá se na uživatelově *identitě, roli nebo obojím*. Příklad role může být *administrátor*, *uživatel* apod.

Chráněný přístup ke kódům (code access security)

První způsob ochrany kodů chrání počítačové systémy před záludnými mobilními kódy a nechává legitimní mobilní kódy bezpečně fungovat.

Mobilní kód přichází z jakéhokoliv zdroje a může být vsazen a použit v mnoha prostředích. Běžné zdroje zahrnují e-mailové přílohy, dokumenty a downloadované soubory z Internetu. Mnoho uživatelů se někdy setkalo s takovým záludným kódem, at již se jedná o počítačové viry nebo červy poškozující nebo ničící data a tím i čas a peníze.

V prostředí .NET Framework dovoluje mechanisums chráněného přístupu různé stupně důvěryhodnosti kódů. Důvěryhodnost kódů závisí na jejich původu a jiných aspektech identity kódu. Mechanismus chráněného přístupu ke kódům používá funkce typu definování povolení, umožnění určitému kódu žádat o povolení k přístupu, která potřebuje k běhu, a následně přidělování povolení k přístupu.

Chráněný přístup ke (zdrojovým) kódům může omezit možnost útoku spouštěného kódu a tím minimalizovat škodu vyplývající z bezpečnostních slabin.

Klasický příklad chráněného přístupu budeme demostrovat na souborových operacích. Omezme se pouze na povolení/zamezení čtení ze souboru pro určitý kód.

Ukázka kódu

```
using System;
using System.Security.Permissions;
...

  string user;
  user = ...
  // vytvoreni noveho permission a nastaveni pouze pro cteni souboru "file.txt"
  new FileIOPermission perm = new FileIOPermission(FileIOPermissionAccess.Read,
"file.txt");

  // povoli se pristup k danemu prostredku, zde cteni souboru
  if(user == "safeuser") perm.Assert();
  // zakaze se pristup
  else perm.Deny();
...
```

Lze vytvářet celé kolekce povolení. Pokud chceme používat takovéto kolekce, vkládáme jednotlivé instance tříd **Permission** (např. **FileIOPermission**) do instance typu **PermissionSet**.

Bezpečnost založená na rolích (Role-based security)

Znalosti

Druhý způsob ochrany dat je založen na tzv. *rolích*. Prostředí .NET Framework uvádí sjednocený model pro správu uživatelovy identity a rolí pro autorizaci. Model je založen na tzv. *principalovi* jako uživateli, jehož kód je prováděn. *Autentifikace* je proces ověřování pověření a zjištění identity principala. Principal může mít žádnou nebo i více rolí reprezentující stupeň jeho pověření.

Role jsou pojmenované skupiny uživatelů seskupené podle stejných práv. Může se zde vyskytovat i *principal*, což se dá volně přeložit jako "představitel hlavní role". Principal může být členem jedné nebo více skupin. Běžící aplikace pak může zjistit identitu současného principala a případně se dotázat, zda vyhovuje roli nezbytné pro provedení určité operace.

U podniků je důležitá identita uživatelů založená na přihlašování do systému Windows. V tom případě lze použít třídu **WindowsPrincipal**, aby zjistila roli současného principala.

Ukázka kódu

```
using System;
using System.Security.Principal;
...

   // zjisteni informaci o aktualnim uzivateli pocitace
   WindowsPrincipal wp = new WindowsPrincipal(WindowsIdentity.GetCurrent());

   // uzivatelske jmeno
   Console.WriteLine("User name: " + wp.Identity.Name);
   // vypise true, pokud aktualni uzivatel ma administratorska prava
   Console.WriteLine("Authentication Type: "
+wp.IsInRole(WindowsBuiltInRole.Administrator);
```

Někdy se ovšem může hodit vytvořit si vlastní role a oprávnění. Pak se použije třída **GenericPrincipal**.

Ukázka kódu

```
...
   // vytvoreni pole s pouzivanymi rolemi pro uzivatele jmenem jason
   string[] jasonRoles = new string[5];
  jasonRoles[0] = "admin"; // ....

   // vytvori se nova vseobecna identita se jmenem jason
   GenericIdentity gi = new GenericIdentity("jason");
   // vytvoreni principala a prirazeni mu roli
   GenericPrincipal gp = new GenericPrincipal(gi, roles);
   // vypis na obrazovku jmena uzivatelskeho principala
   Console.WriteLine("name    : " + gp.Identity.Name);     // vypise: "name    :
jason"
   Console.WriteLine("is admin: " + gp.IsInRole("admin")); // vypise: "is admin:
True"
...
```

Příklad ke stažení

Příklad na vytvoření instance třídy **GenericPrincipal** a zjištění informací o instanci třídy **WindowsPrincipal** je k dispozici ke stažení zde [9/3_2.cs].

Kapitola 10. Komponenty COM a distribuované aplikace

V dnešní době poměrně hojně využívaná COM technologie a její aspekty - to je téma této kapitoly. Povíme si něco o způsobu spouštění COM komponent, dozvíme se něco o .NET komponentách a ukážeme si tvorbu .NET komponent. Ukážeme si možnosti využití prostředí .NET Framework a jazyka C# pro programování distribuovaných aplikací.

Objekt COM

Znalosti

Component Object Model je způsob, kterým mohou komunikovat softwarové komponenty. Jedná se o binární a sítový standard umožňující kterýmkoliv dvěma komponentám komunikovat bez starostí, na kterém stroji právě běží (pokud jsou stroje propojeny), pod jakým OS stroje pracují (pokud podporují COM) a v jakém jazyce jsou komponenty napsány.

COM je založena na objektech. Tyto objekty ale nejsou úplně to stejné jako v objektových jazycích (i když si jsou velice podobné). COM objekty jsou velmi dobře zapozdřeny, není tedy možnost získat přístup k vnitřní implementaci objektu. Není možnost zjistit, jaká data, datové struktury a podobné může daný objekt obsahovat. Obvykle se COM objekty kreslí pouze jako prázdné obdélníky.

Znalosti

Jediný způsob, jak komunikovat s COM objekty je skrze rozhraní. Rozhraní pak zpřístupňují sadu metod použitelnou pro práci s objektem. Rozhraní tvoří *dohodu mezi komponentou a klientem*. Jinými slovy, rozhraní pouze nedefinuje, které funkce a metody jsou použitelné, ale definuje také, co objekt dělá, když jsou dané metody volány. Tato dohoda ale nevymezuje způsob implementace - COM objekt není nijak omezen ve způsobu implementace metod rozhraní. Jediné, co musí dodržet, je že už jednou dohodnuté rozhraní nelze změnit.

Toto se děje z důvodu, že na dohodnutém způsobu komunikace mohou záviset další aplikace. Pokud by se dohoda rozhraní porušila, mohlo by dojít k selhání.

Existují však úskalí této technologie. Nelze například zcela využít kód nacházející se v těchto komponentách (právě kvůli "dokonalému" zapouzdření). Nelze tedy z komponent dědit a následně upravit některé metody. U COM komponent se hovoří o tzv. *DLL hell*, tedy "peklu DLL". Potíže se vyskytují zejména v souvislosti s verzemi. Často se stává, že novější verze komponent přepíší jejich starší verze a tím způsobí nekorektní chování aplikací vycházejících z předchozích verzí.

Každý COM objekt implemetuje rozhraní **IUnknown** obsahující následující tři funkce. Z něj dědí všechna ostatní rozhraní použitá v COM objektu.

AddRef()

Slouží ke zvýšení počtu odkazů na použité rozhraní.

QueryInterface()

Slouží pro přepínání mezi rozhraními

Release()

Sníží počet odkazů na použité rozhraní. Pokud počet klesne na nulu, objekt je uvolněn.

GUID

GUID (Globally Unique Identifier) je klíčová část modelu COM. Jedná se o 128 bitovou strukturu, u níž je zajištěno, aby žádné dva GUID nebyly stejné. GUID se v COM modelu používá ze dvou důvodů:

jednoznačně rozlišit určitý COM objekt. GUID přiřazené ke COM objektu se nazývá class ID (CLSID). To se používá k vytvoření instance COM objektu.

jednoznačně rozlišit patřičné rozhraní COM. GUID používané s určitým COM rozhraním se pak nazývá interface ID (IID). Tohoto se používá tehdy, když se žádá o náležité rozhraní od objektu.

I když GUID se bere jako struktura, většinou se jedná o řetězec ve tvaru *{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}*, kde x představují hexadecimální čísla.

HRESULT hodnoty

Všechny COM objekty vracejí 32-bitovou hodnotu nazývanou **HRESULT**. Nejčastěji se **HRESULT** bere jako struktura dvou druhů informací

Zda daná metoda uspěla či nikoliv

Detailnější informace o výstupu operace podporované metodou

COM v .NET Framework

V této kapitole se budeme zabývat problémem, jak v prostředí .Net načteme do naší aplikace COM objekt. Klient žádající o zpřístupnění patřičného COM objektu jej získá prostřednictvím *RCW (Runtime-Callable Wrapper)*. RCW zabalí COM objekt a zprostředkuje mezi ním a .NET CLR (Common Language Runtime) prostředí tak, že se COM objekt bude .NET klientu jevit jako běžný .NET objekt a naopak COM objektu se bude .NET klient jevit jako běžný COM klient.

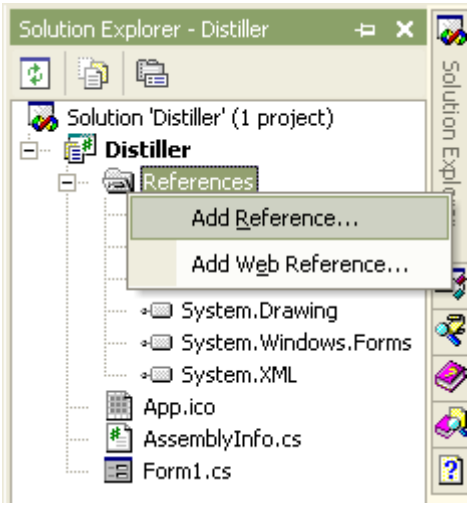
Obrázek 10.1. Přístup ke COM objektu přes RCW

Generování RCW ve Visual Studiu pro komponentu Adobe Distiller

RCW je nutno generovat. Lze to provést (minimálně) dvěma způsoby. Prostřednictvím .NET Visual Studia nebo z příkazové řádky nástrojem nazvaným *TlbImp.exe*. Ukážeme si, jak se generuje RCW s použitím Visual Studia. Budeme volat modul **Adobe Distiller** pro konverzi formátu *PostScript* na *PDF*.

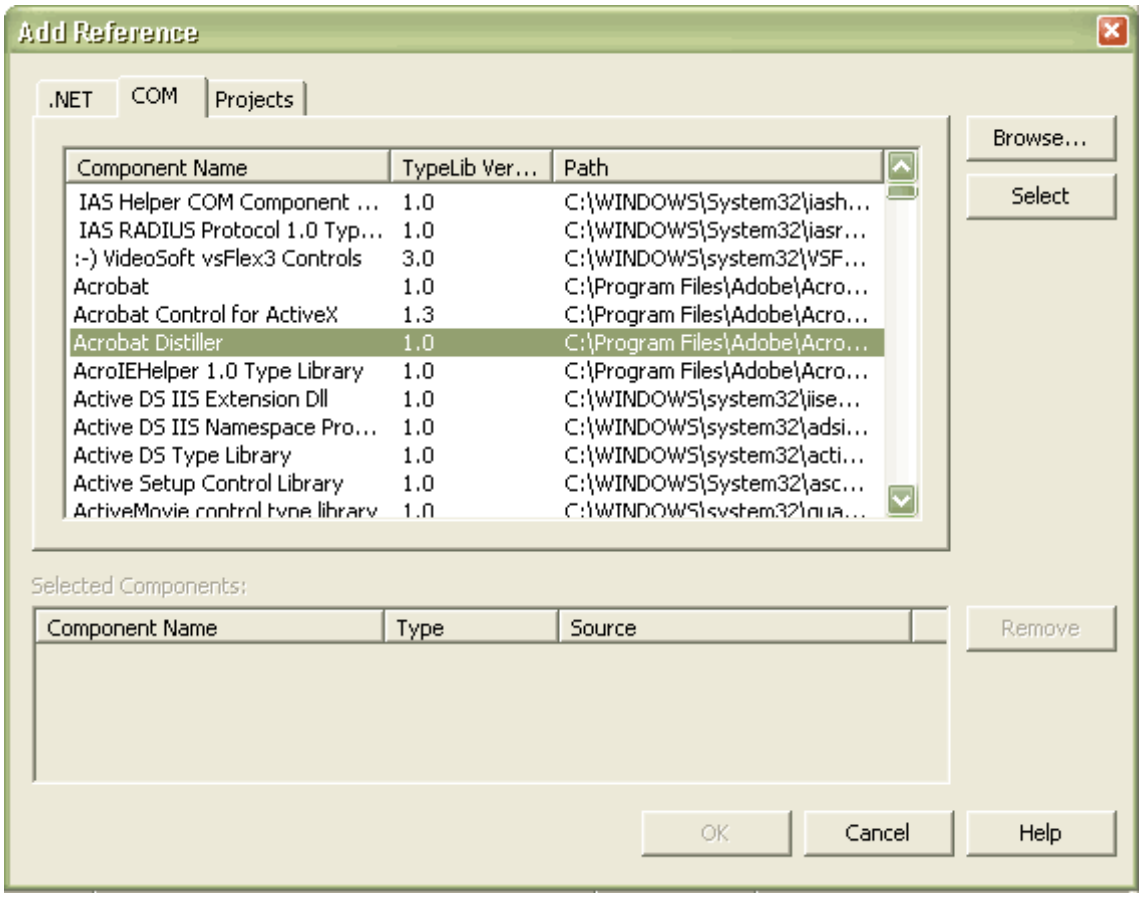
V sekci *Sollution Explorer* našeho projektu poklepeme pravým tlačítkem myši na *References*.

Obrázek 10.2. Solution Explorer - Add Reference



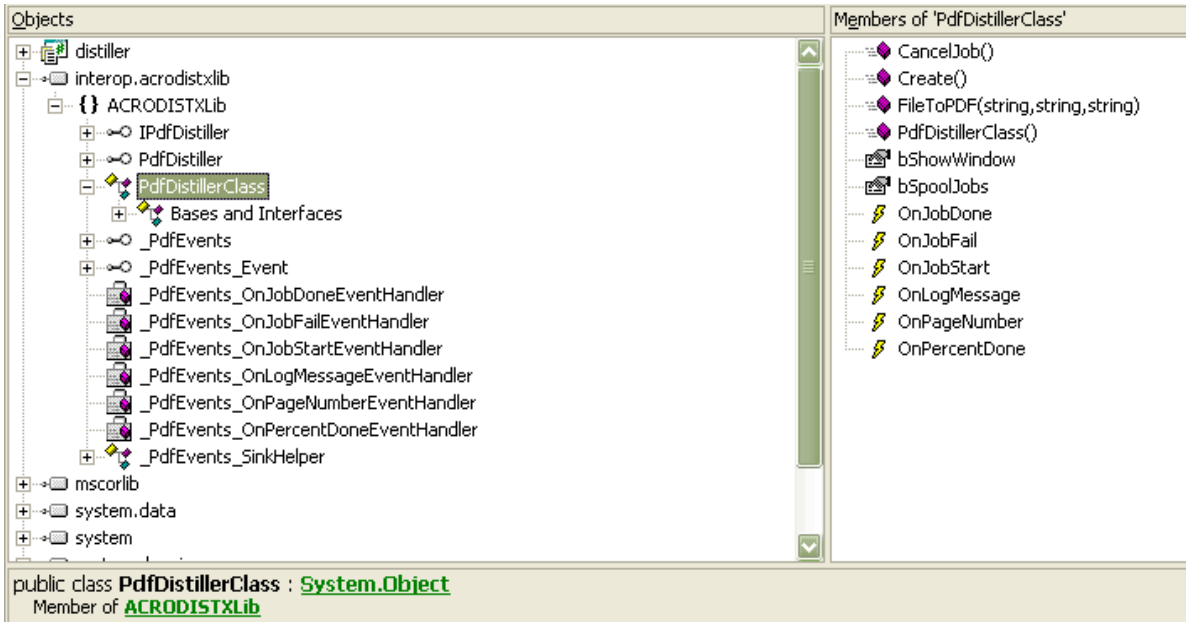
Vybereme záložku *COM*. Samozřejmě, pokud ji chceme použít, komponenta **Adobe Distiller** musí existovat v našem systému. Vybereme ji příkazem *Select*. Po stisknutí *OK* se v seznamu *References* objeví nová komponenta **ACRODISTXLib**.

Obrázek 10.3. Vybrání komponenty Adobe Distiller



Nyní, když máme komponentu zařazenu do našeho projektu, můžeme s ní zacházet jako se jmenným prostorem a přistupovat k jejím metodám. Prostředí Visual Studia umožňuje i pro komponenty funkci *IntelliSense* pro zobrazení dostupných metod a rozhraní. Pokud bychom se chtěli podívat, co všechno komponenta obsahuje, ve Visual Studiu na záložce *Solution Explorer - References* poklepeme pravým tlačítkem myši na komponentu a vybereme *View in Object Browser*. Naše komponenta **ACRODISTXLib** bude zařazena do assembly **interop.acrodistxlib**.

Obrázek 10.4. Object Browser - ACRODISTXLib



Na obrázku vidíme, co všechno *Object Browser* zobrazí - pokud budeme chtít zobrazit obsah **PdfDistillerClass**, poklepeme na ni myší a v pravém okně se zobrazí seznam všech metod, vlastností apod. Pod dvěma okny se zobrazuje popis aktuálně vybraného. Nyní se zobrazují informace o třídě **PdfDistillerClass**.

V této chvíli už nám nic nebrání vytvořit instanci této třídy a zacházet s ní jako s jinou běžnou instancí.

Použití komponenty Adobe Distiller

Jak jsme si řekli výše, stačí jen vytvořit instanci třídy **PdfDistillerClass** a zavolat soubor, který chceme převést.

Ukázka kódu

```
...
string myPSfile = "mypsfile.ps";
string myPDFfile = "mypdffile.pdf";
ACRODISTXLib.PdfDistillerClass distiller = new ACRODISTXLib.PdfDistillerClass();
distiller.FileToPDF(myPSfile, myPDFfile);
...
```

Příklad ke stažení

Příklad na komponenty s konverzí PS na PDF i se spustitelným souborem je ve formátu zip stažení zde [examples/Distiller_COM.zip].

COM vs .NET komponenty

V předchozích dvou kapitolách jsme psali o použití COM komponenty a její začlenění do našeho .NET projektu. V prostředí .NET existuje nový typ komponent rozšiřující vlastnosti COM a odstraňující jejich nedostatky.

V COM technologii se používá metoda GUID - nutnost generování jedinečného kódu pro každou komponentu, její metody. .NET komponenty oproti tomu samy obsahují svůj vlastní popis obsažený v segmentech označených jako manifest. Metadata oproti tomu se starají o volání patřičných metod, řeší problém s verzováním DLL, umísťují do paměti instance tříd. Také se nemusejí starat o dobu života objektu. Většinu potřebných operací zajistí CLR.

Následující tabulka poukazuje na rozdíly mezi COM komponentami a .NET komponentami

Tabulka 10.1. COM Vs. .NET komponenty

COM	.NET komponenty
globální přístup ke komponentám na počítači	žádný globální přístup
nutná registrace DLL	žádná potřeba registrace
informace registru a DLL jsou na dvou různých místech (DLL peklo)	Kompletní informace nezbytné pro rozpoznání verze a ostatních jsou na jednom místě (DLL ráj)
spouštěno neřízeném prostředí	spouštěno v řízeném prostředí
pro ASP aplikace je nezbytný pro začlenění COM restart serveru	není třeba restart

Technologie .NET komponent určitě přináší zlepšení pro práci s komponentami, nicméně mnoho firem vložilo nemalé finanční prostředky do technologie COM. Navíc tato technologie se vyskytuje ve světě informačních technologií více než sedm let. Je tedy přinejmenším zdlouhavé přepsat všechny COM komponenty na komponenty .NET.

Tvorba .NET komponenty

Vytváření .NET komponenty je triviální.

Ukázka kódu

```
class MaPrvniKomponenta {
  public int ReturnTen() {
    return 2*5;
  }
}
```

- prakticky v každém případě, když jsme vytvořili nějakou třídu, jsme mohli vytvořit i komponentu. Stačí pouze ve třídě, ze které chceme komponentu vytvořit, nevložit metodu Main a uložit ji jako knihovnu (DLL), pomocí příkazové řádky takto:

csc /t:library /out:nazevCSsouboru.dll nazevCSsouboru.cs

V prostředí Visual Studia vytvoříme nový projekt s názvem *Class Library*, vložíme do ní kód komponenty a běžným způsobem kompilujeme.

Pokud budeme tuto komponentu někdy v budoucnu chtít používat, postupujeme podobně jako při výběru COM komponenty, pouze s tím rozdílem, že vybereme klikneme na tlačítko *Procházet* (*Browse*) a umístíme ji.

.NET Remoting

Postupem času se mění způsob tvorby aplikací z lokálních na distribuované aplikace. Zvláště se jeví praktické vytvářet aplikace jako sadu komponent distribuovaných po síti počítačů a pracovat s nimi, jako by celá sada byla umístěná na jediném počítači.

Architektura .NET Remoting

.NET Remoting přináší infrastrukturu pro distribuované objekty. Prostředí .NET Remoting nabízí komplexní funkcionalitu včetně podpory pro předávání objektů hodnotou nebo odkazem, ovládání podmínek životního cyklu apod.

Jednoduše řečeno základ *Remoting* tvoří používání referencí objektů ke komunikaci mezi serverem a klienty.

V zásadě je nutno dodržet následující podmínku, abychom mohli používat .NET Remoting:

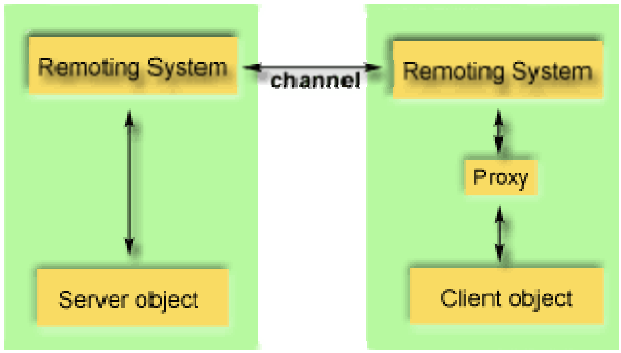
> Všechny vzdálené objekty musí být registrovány Remoting systémem (prostředím), než k nim může přistoupit klient.

Tato registrace se obvykle provádí před startem hostující aplikace a je ji možno provést buď s pomocí konfiguračního souboru nebo voláním registrační metody **RegisterWellKnownServiceType()** s patřičnými parametry. Když je objekt zaregistrován, vytvoří prostředí referenci na tento vzdálený objekt a extrahuje z assembly metadata potřebná pro lokalizaci vzdáleného objektu kdekoliv na síti.

Pokud se podaří nakonfigurovat klienta a server správně, stačí nám vytvořit novou instanci vzdáleného objektu pomocí klíčového slova **new**. Klient posléze získá referenci na objekt serveru a zbytek už se provádí stejně, jako by objekt byl lokální.

Na obrázku vidíme způsob, jak se pracuje se vzdálenými objekty. Pokud se vytvoří instance vzdáleného objektu, vytvoří se lokální proxy objekt jevící se klientovi stejně jako vzdálený objekt. Klient volá metodu na proxy objekt. Volání se předá *Remoting Systemu*, ten jej předá řídícímu procesu, vyvolá se server objekt a jeho odpověď se vrací zpět ke klientovi. *Channel* na obrázku reprezentuje typ, který bere proud dat, zaobalí jej v závislosti na konkrétním sítovém protokolu a odesílá jej k druhému počítači.

Obrázek 10.5. Princip Remoting



Pokud bychom psali *Remoting System* vlastnoručně, potřebovali bychom se naučit (pokud to ještě neumíme) širokou oblast protokolů a specifikací formátů pro komunikaci na síti. U .NET Remoting se o toto nemusíme starat.

Objekty .NET Remoting

Můžeme použít tyto způsoby aktivace vzdálených objektů (a s tím související způsoby vytváření instancí objektů)

Aktivace na straně serveru

Objekty na straně serveru nejsou vytvářeny při volání new na straně klienta, ale pouze tehdy, když jsou skutečně potřeba (volání metod apod.). Při volání new se pouze vytvoří proxy pro daný typ.

Jednoduché volání (*Single Call*)

Tyto objekty obslouží pouze jediný požadavek. Nejsou přizpůsobeny pro uchovávání stavových informací. Vždy se vytváří nová instance objektu pro každý požadavek klienta.

Jediný objekt (*Singleton Object*)

Tyto objekty obsluhují více klientů. Používají se v případech, kdy je třeba explicitně sdílet data mezi klienty. Nikdy se nevytváří více instancí. Pokud existuje instance objektu, všechny požadavky se směrují směrem k ní.

Aktivace klientem

Chování jako u klasických objektů. Klient vydá požadavek pomocí operátoru **new** a server vytvoří instanci dané třídy. Poté vyšle referenci na objekt, který byl vytvořen. Každé volání **new** vrací proxy k nové instanci daného typu na serveru. Pokaždé, když klient vytvoří novou instanci, se vytvoří skutečná instance na serveru a je platná do doby platnosti instance.

Tvorba a použití Remote objektů

Použití vzdálených objektů si ukážeme na vzorovém příkladu. Jedná se o jednoduchou třídu **MathLibrary** s metodami pro sčítání, odčítání celých čísel a "uvítací" metody. Pokud bychom tuto třídu umístili na server (například IIS), můžeme hovořit o používání webové služby, pokud tuto třídu náležitě zpřístupníme.

Se vzdálenými třídami je možno pracovat přímo vytvořením patřičných vazeb v jazyce C# (nebo v jiném podporovaném .NET Framework) - aktivace vzdáleného objektu s pomocí třídy **System.Activator** na straně klienta a registrováním vzdáleného objektu pomocí metody **RemotingConfiguration.RegisterWellKnownServiceType()**. Lze s nimi pracovat i pomocí konfiguračních souborů, pak stačí tyto soubory načíst a může se rovnou pracovat se vzdálenými objekty. Ukážeme si druhý způsob.

Vytvoření vzdáleného typu (remotable type)

Aby bylo možno pracovat s instancemi vzdálených tříd, tyto třídy musí dědit z **MarshalByRefObject**. Následující kód ukáže jednoduchou třídu, která může být vytvořena a volána v jiné aplikační doméně.

Ukázka kódu

```
using System;
public class MathLibrary : MarshalByRefObject {
  public string Invitation() {
    return "Hi there, this is a remote object!";
  }

  public int Add(int num1, int num2) {
    return (num1+num2);
  }

  public int Sub(int num1, int num2) {
    return (num1-num2);
  }
}
```

Tuto třídu uložíme jako *MathLibrary.cs* a zkompilujeme v příkazovém řádku pomocí:

csc /noconfig /t:library MathLibrary.cs

Pokud budete mít problém se spuštěním kompilátoru a využíváte svůj vlastní počítač, csc.exe se nachází v adresáři *WINDOWS\Microsoft.NET\Framework\(co nejnovější verze)\csc.exe*

Vytvoření naslouchací aplikace

K tomu, abychom mohli vytvářet instance objektů na dálku, je třeba vytvořit hostitelskou aplikaci nebo naslouchací aplikaci. Tyto dělají následující dvě věci

Vybere a registruje komunikační kanál (reprezentovaný objektem, pracujícím se sítovými protokoly a formáty serializací) .

Registruje náš typ pomocí .NET Remoting system tak, aby .NET Remoting mohl využívat vytvořený komunikační kanál k naslouchání požadavků pro náš typ.

Vytvoříme tedy kód naslouchající aplikace.

Ukázka kódu

```
using System;
using System.Runtime.Remoting;
public class Listener {
  public static void Main() {
    RemotingConfiguration.Configure("Listener.exe.config");
    Console.WriteLine ("Listening for requests. Press Enter to exit...");
    Console.ReadLine();
```

```
  }
}
```

Předchozí kód uložíme do souboru *Listener.cs* a stejného adresáře jako *MathLibrary.cs* a opět skompilujeme, tentokrát spolu s naší knihovnou.

csc /noconfig /r:MathLibrary.dll Listener.cs

Předchozí kód ale používá k naslouchání vzdálených typů soubor *Listener.exe.config*, je zapotřebí vytvořit i tento konfigurační soubor. V něm nastavíme, jak se bude s naším typem zacházet, způsob komunikace apod. Uložíme jej opět do stejného adresáře.

Ukázka kódu

```
<configuration>
  <system.runtime.remoting>
    <application>
      <service>
        <wellknown mode="Singleton" type="MathLibrary, MathLibrary"
objectUri="MathLibrary.rem" />
      </service>
      <channels>
        <channel ref="http" port="8989"/>
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>
```

Vytvoření aplikace na straně klienta

V této chvíli máme vytvořenou knihovnu s matematickými funkcemi a hostitelskou aplikaci pro vzdálené používání (Remoting).

Nyní vytvoříme klientskou aplikaci. Ta se musí zaregistrovat jako klient pro vzdálený objekt. Pak při volání instance vzdálené třídy .NET Remoting zachytí zprávy klienta a předá je vzdálenému objektu. Následně vrátí klientu výsledek.

Ukázka kódu

```
using System;
using System.Runtime.Remoting;

public class Client {
  public static void Main() {

    string strNum1, strNum2;
    int num1, num2;

    RemotingConfiguration.Configure("Client.exe.config");
    MathLibrary lib = new MathLibrary();
    Console.WriteLine("Enter number 1:");
    strNum1 = Console.ReadLine();
    Console.WriteLine("Enter number 2:");
    strNum2 = Console.ReadLine();

    num1 = Convert.ToInt16(strNum1);
    num2 = Convert.ToInt16(strNum2);
    Console.WriteLine("Addition {0} + {1} = {2}", num1, num2, lib.Add(num1,
num2));
  }
}
```

Výše uvedený kód uložíme jako *Client.cs* do stejného adresáře jako předchozí a opět skompilujeme s knihovnou *MathLibrary*

csc /noconfig /r:MathLibrary.dll Client.cs

Jelikož ale klient opět používá konfigurační soubor *Client.exe.config* k naslouchání vzdáleného typu, je třeba ještě vytvořit tento soubor. Umístíme jej opět do stejného adresáře jako všechny předchozí.

Ukázka kódu

```
<configuration>
  <system.runtime.remoting>
    <application>
      <client>
        <wellknown type="MathLibrary, MathLibrary"
url="http://localhost:8989/MathLibrary.rem" />
      </client>
    </application>
  </system.runtime.remoting>
</configuration>
```

Spuštění aplikace

Nyní nám už nic nebrání spuštění námi vytvořené aplikace. Je třeba mít na paměti, že po celou dobu volání klientů musí být spuštěno naslouchání - *Listener.exe*.

Spustíme tedy *Listener.exe*, následně spouštíme *Client.exe* a vidíme výsledek.

Příklad ke stažení

Tento příklad je k dispozici ke stažení ve formátu zip zde [examples/remoting.zip].

Kapitola 11. Windows Forms

Až do této chvíle jsme většinou (až na výjimky, viz např. kapitola ADO.NET) používali textový výstup na konzolu počítače. Tento způsob zobrazování dat se hodí ale spíše pro technické aplikace, které nemají příliš obsáhlý výstup. Horší to je s aplikacemi, které vykreslují grafy, chtějí reagovat na uživatelovy požadavky interaktivně, tedy ne formou konzolového výstupu apod. V těchto případech se samozřejmě více uplatní grafický výstup. Navíc tento způsob zobrazení vypočtených a získaných dat je pro uživatele mnohem více srozumitelný a přehledný. Prostředí .NET nabízí pro vytváření a práci s aplikacemi typu Windows (tedy "okenní") jmenný prostor Windows.Forms obsahující rozsáhlé množství prvků, kterými je možno vytvořit, zpřehlednit a zjednodušit pro uživatele naši aplikaci.

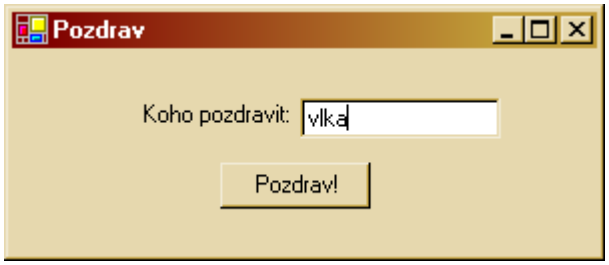
Jednoduchá aplikace - Pozdrav

Na ukázku, jak lze vytvořit jednoduchou aplikaci uvádím tuto, nenáročnou na programování: vytvoříme formulář s textovým polem, do kterého načteme jméno člověka, kterého chceme pozdravit. Po zmáčknutí tlačítka "*Pozdrav!*" se otevře druhý formulář s pozdravem.

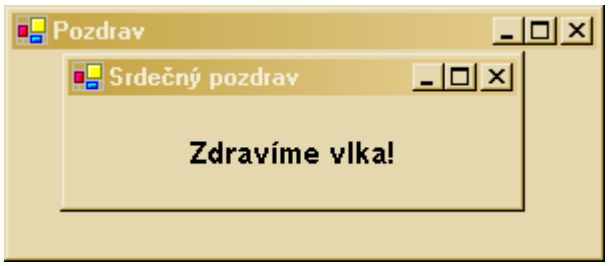
Příklad ke stažení

Ukázka v zipu je k dispozici zde [11/pozdrav.zip].

Obrázek 11.1. Windows Forms Aplikace "Pozdrav", okno 1



Obrázek 11.2. Windows Forms Aplikace "Pozdrav", okno 2



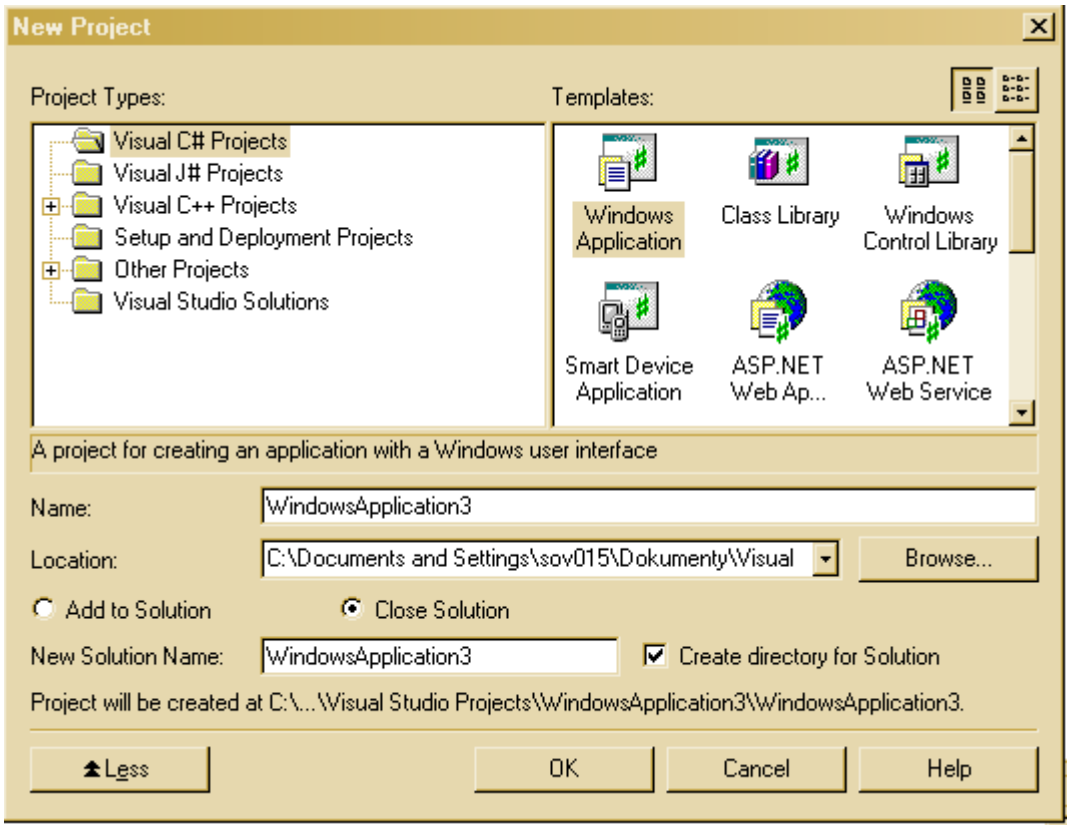
Naše ukázková aplikace využívá dva ovládací prvky (textové pole a tlačítko). Tlačítko na základě vzniklé události (**Click**) provede nějakou akci. Tímto způsobem - tedy na základě událostí - funguje ovládání všech ovládacích prvků.

Nyní si popíšeme celý průběh vytvoření formuláře s ovládacími prvky.

Vytvoření aplikace s využitím Windows Forms v .NET Visual Studiu

Vytvoříme nový projekt, vybereme možnost *Visual C# Projects -> Windows Application*. Po zadání názvu projektu se vygeneruje úvodní formulář a k němu inicializační zdrojový kód. Je na místě si všimnout, že pokud používáte Visual Studio, v konstruktoru našeho formuláře se nachází privátní metoda **InitializeComponent()**. Tato metoda nastaví všechny prvky formuláře tak, jak jsme si je rozvrhli v módu návrhu formuláře. Zároveň je doporučováno zasahovat do této metody pokud možno co nejméně a radši vše ostatní provádět mimo ni.

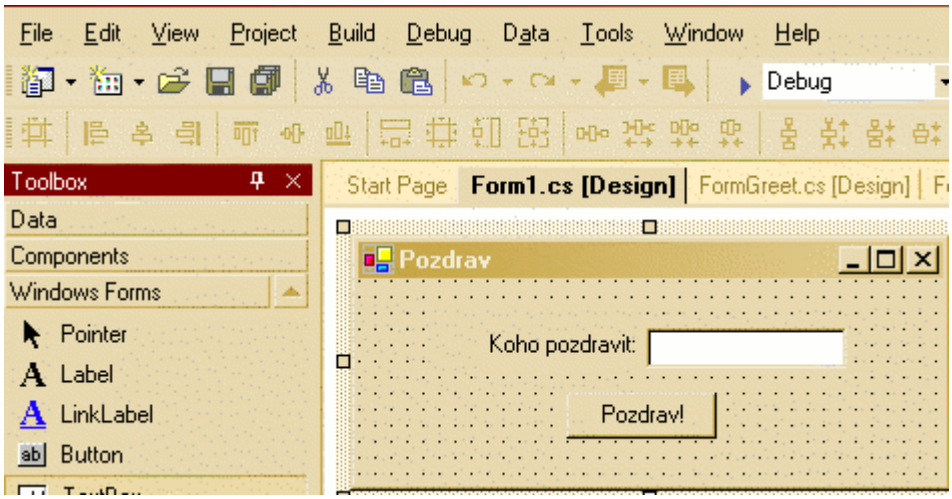
Obrázek 11.3. Nová aplikace typu Windows Application



Vložení ovládacích prvků do formuláře

Máme vytvořený prázdný formulář. Nyní do něj vložíme ovládací prvky textové pole (**TextBox**) a tlačítko (**Button**). K textovému poli také vložíme popisek (**Label**), aby bylo jasno, čím se toto textové pole má plnit. Všechny komponenty pro používání ve formuláři se nacházejí na liště *Toolboxu*. Postupně z něj tedy vložíme (buď přetažením myší nebo kliknutím na komponentu a poté jejím vytvořením myší na požadovaném místě) do našeho formuláře tlačítko a textové pole. Do projektu poté vložíme ještě jeden formulář, který bude obsahovat text pozdravu.

Obrázek 11.4. Vytvoření ovládacích prvků



"Oživení" ovládacích prvků

V této chvíli máme vytvořené ovládací prvky, které samy o sobě ještě nic neudělají. Je třeba provést nějakou

akci po stisku tlačítka "*Pozdrav!*". Ta se vytvoří přidáním reakce na událost tlačítka **Click**.

Ukázka kódu

```
...
System.Windows.Forms.Button button1 = new System.Windows.Forms.Button;
...
// eventFunction je funkce reagujici na udalost tlacitka Click
button1.Click += new System.EventHandler(ButtonEventFunction);
...
// proved akci
void ButtonEventFunction(object sender, System.EventArgs e) {
   ...
}
```

Totéž, ale mnohem jednodušeji, provedeme dvojím poklepáním myší na tlačítko "*Pozdrav!*" v návrhu formuláře (na obrázku s názvem Form1 - *Form1.cs [Design]*). Microsoft Visual Studio za nás vytvoří samo funkci pro zpracování události (se jménem **button1_Click**) a samo přidá k události **Click** tlačítka s názvem *button1* **EventHandler**. Poté, co se vytvoří obsluha události tlačítka, Visual Studio nás nasměruje do této vytvořené funkce. My do této funkce pouze přidáme, že chceme vytvořit nový formulář s textem pozdravu.

Ukázka kódu

```
...
private void button1_Click(object sender, System.EventArgs e) {
  // formular s textem pozdravu, predam jmeno toho, koho chci pozdravit
  new FormGreet(this.textBoxWho2Greet.Text);
}
...
```

K tomu, aby se formulář po stisku tlačítka objevil, a my ho takto mohli vytvořit, musíme přetížit konstruktor. V konstruktoru předáme jméno typu **string**, koho chceme pozdravit. V tom samém konstruktoru zároveň nastavíme, aby se formulář s pozdravem zobrazil.

Ukázka kódu

```
this.Show() = true;
```

Za zmínku ještě stojí obsluha události **Exit_Click()**. Ta se provádí tehdy, pokud chceme aplikaci zavřít klasicky kliknutím na křížek vpravo nahoře. Musíme ji přidat do projektu, jinak by totiž kliknutí na křížek nedělalo nic.

Ukázka kódu

```
private void Exit_Click(object sender,EventArgs e) {
  // uvolneni zdroju
  this.Dispose();
  Application.Exit();
}
```

A máme zajištěno, že aplikaci budeme moci uzavřít.

MDI (Multiple Document Interface)

Multiple Document Interface používáme běžně, i když si toho možná nejsme vědomi. Jedná se o možnost mít otevřených více oken v rámci jedné aplikace. Tato podřízená okna jsou přímo závislá na rodičovském okně a nemohou se přesunout z jeho plochy. Rodičovská aplikace rovněž může provádět s potomky různé operace v závislosti na jejich změnách (zavření potomka, aktivace potomka apod.).

Vytvoření rodičovského okna

Pokud chceme vytvořit takovouto aplikaci, je třeba určit formulář, který bude rodičovský. To provedeme s pomocí vlastnosti **Form.IsMdiContainer**.

Ukázka kódu

```
// konstruktor rodicovskeho formulare
public FormParent() {
...
  // nastavim aktualni formular jako rodic
  this.IsMdiParent = true;

  // vytvoreni a zobrazeni formularu - potomku
  Form pdr = new FormChild();
  // nastaveni rodicovskeho formulare
  pdr.MdiParent = this;
  pdr.Show();

  Form pdr1 = new FormChild2();
  pdr1.MdiParent = this;
  pdr1.Show();
}
```

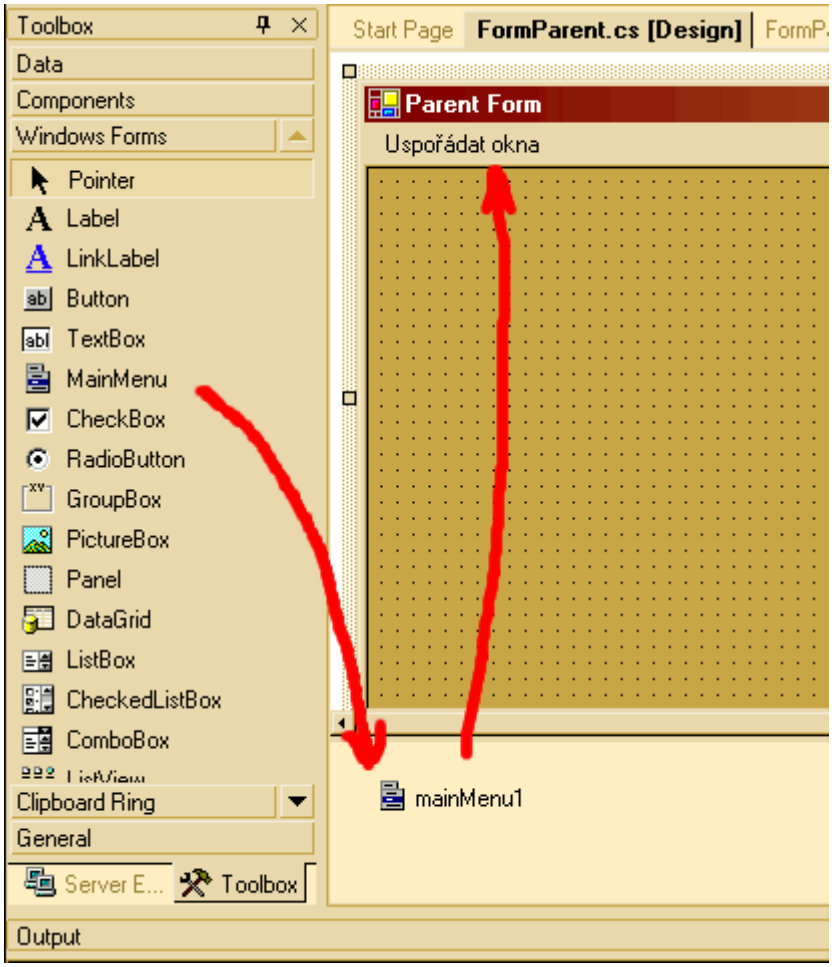
V kódu výše uvedeném jsme zároveň určili, která okna budou potomky aktuálního okna.

Práce s potomky rodičovského okna

Nyní můžeme s okny - potomky provádět různé operace. Ukážeme si, jak lze použít prvek **Menu**. Z menu budeme ovládat, jak seřadit potomky aktuálního okna (horizontální, vertikální, kaskádové řazení).

Nejprve do rodičovského okna vložíme prvek **Menu** a do něj odkazy na styl uspořádání oken - potomků (*TileHorizontal*, *TileVertica*l, *Cascade*). Pokud bychom toto chtěli dělat ručně, čeká nás dost zdlouhavé a nezáživné práce s vytvářením menu. Proto je výhodnější použít vizuální nástroje, které vygenerují menu tak, jak si jej vytvoříme v náhledu.

Obrázek 11.5. Vložení menu do formuláře



Poté, co vložíme menu do našeho projektu, se objeví na stadnardním místě. Když jednou klikneme na první pozici (většinou bývá prázdná), budeme moci tuto pozici přejmenovat a po přejmenování se objeví nová pozice. Pokud budeme chtít nad touto pozicí vyvolat nějakou akci, poklikáme na ni dvakrát a dostaneme se přímo do obsluhy události pro tuto pozici.

Ve vlastnostech konkrétní pozice si můžeme nastavit, jakou tato pozice bude mít klávesovou zkratku a také horkou klávesu.

Horká klávesa se označí přidáním znaku "*&*" před písmeno vlastnosti **Text**, které chceme označit jako "horké".

Klávesová zkratka se nastavuje prostřednictvím vlastnosti **Shortcut**.

Takto tedy vytvoříme hlavní položku menu "*Uspořádat okna*" a pod ni umístíme podmenu s položkami *Horizontálně*, *Vertikálně*, *Kaskádovat*. Následně vytvoříme akce pro tyto položky. Pro uspořádání oken se zavolá metoda předka, která své potomky uspořádá podle potřeby.

Ukázka kódu

```
// usporada okna - potomky v rodicovskem okne horizontalne
this.LayoutMdi(MdiLayout.TileHorizontal);
...
```

Příklad ke stažení

Ukázka na práci s okny rodič - potomek je k dispozici zde [examples/Forms.zip].

Dialogy

Vývojové prostředí .NET nabízí celou řadu dialogů. Od klasických dialogů načítání a ukládání souboru přes dialog výběru barvy, fontu nebo dialog pro tisk.

Práce s dialogy není nijak složitá, příkládám proto pouze projekt objasňující to, jak lze získat hodnoty z dialogu pro vybrání složky, otevření souboru (pokud je vybrána složka, nastaví se aktuální cesta jako ta vybraná z dialogu vybrat složku), a kterou barvu jsme získali z dialogu pro vybrání barvy.

Znalosti

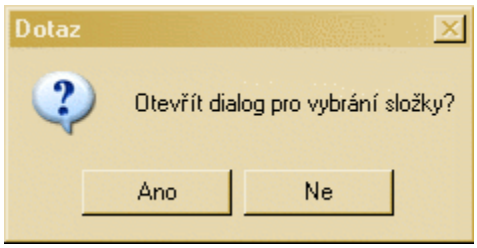
Za zmínku stojí ještě dialogová okna oznamující plánované provedení nějaké akce nebo třeba chybovou zprávu zobrazenou uživateli. Tato okna se zobrazují přes objekt **System.Windows.Forms.MessageBox** voláním metody **Show()**.

Ukázka kódu

```
using System.Windows.Forms;
...
// reakce uzivatele na dotaz
DialogResult dr;
// zobrazeni dialogu
dr = MessageBox.Show("Otevřít dialog pro vybrání složky?",
       "Dotaz", MessageBoxButtons.YesNo, MessageBoxIcon.Question);
// v dialogu stisknuto "Ano"
if(dr == DialogResult.Yes) { ... }
else { ... }
```

V předchozím dialogu se uživateli zobrazí tykovýto výsledek:

Obrázek 11.6. Výsledek vytvoření dialogu na srovnání oken



Příklad ke stažení

Projekt je ke stažení zde [examples/Dialogy.zip].

Drag And Drop

V dnešní době hojně používaná metoda pro přesouvání nebo kopírování položek by neměla v našem kurzu aplikací pro Windows chybět.

V zásadě se pro realizaci operace Drag And Drop mezi ovládacími prvky používají tři události. V případě prvků **TreeView** a **ListView** se používají události **ItemDrag**, **DragEnter** a **DragDrop**. V případě jiných komponent, jako například **ListBox**, se místo události **ItemDrag** používá událost **MouseDown**. Kromě toho musejí mít všechny prvky, do kterých chceme položky přetahovat, nastavenu vlastnost **AllowDrop** na *true*. Následně se v události **ItemDrag** zavolá metoda **DoDragDrop**().

Mějme příklad, ve kterém budeme chtít přetahovat položky z komponenty **ListView** do komponenty **TreeView**.

Vytvoříme tedy patřičný formulář. Do něj přidáme obsluhu události pro moment, kdy uživatel uchopí položku v prvku **ListView** (**ItemDrag**).

Ukázka kódu

```
this.listView1.ItemDrag += new
```

```
System.Windows.Forms.ItemDragEventHandler(this.listView1_ItemDrag);
```

Následující kód ukáže volání metody pro prvek **TreeView** (do něj se zde přetahují prvky) v obsluze události **ItemDrag**. Ta má jako druhý argument instanci typu **ItemDragEventArgs**, tedy můžeme z ní získat data typu **object**, která jsou přenášena (vlastnost **Item**). Jelikož víme, že budeme přetahovat pouze jméno položky v prvku **ListView**, můžeme provést explicitní přetypování na prvek **ListViewItem** a přímo vybrat text prvku.

Ukázka kódu

```
private void listView1_ItemDrag(object sender,
System.Windows.Forms.ItemDragEventArgs e) {
  string sItem = ((ListViewItem)e.Item).Text;
  // start operace drag & drop
  DoDragDrop(sItem, DragDropEffects.Copy | DragDropEffects.Move);
}
```

Teď víme, jaká data se tedy budou přenášet. Nyní je třeba nastavit, co se stane, když se dokončí přetahování objektu myší. To se nastaví v obsluze události **DragDrop**. V ní nejprve získáme data přenášená událostí a posléze je zpracujeme. V těchto typech událostí se jako argumenty přenášejí i souřadnice, na kterých skončilo přetahování. Tyto souřadnice vybereme a zjistíme, na kterou položku v našem prvku (**TreeView**) ukazovaly.

Ukázka kódu

```
private void treeView1_DragDrop(object sender, DragEventArgs e) {
  Point position = new Point(0, 0);
  // ziskam data prenasena z argumentu udalosti
  string s = (string)e.Data.GetData(typeof(string));

  // zjistim pozici, u ktere se prestalo pretahovat
  Position.X = e.X;
  Position.Y = e.Y;

  // zjistim pozici uzlu a nasledne uzel, u ktereho se prestalo pretahovat
  // (pustilo se tlacitko mysi)
  position = treeView1.PointToClient(position);
```

Pokud daná pozice vyhovuje našim podmínkám, můžeme přetahovaná data vložit.

Příklad ke stažení

Tento příklad na Drag&Drop je k dispozici ke stažení v souboru zip zde [examples/dragdrop.zip].

Další prvky

Není nic lepšího, než si sami vyzkoušet, co všechno lze s ovládacími prvky vyrobit. V tomto, posledním, odstavci se zaměřím pouze na stručný popis obvykle používaných ovládacích prvků.

U většiny následujících ovládacích prvků se dají použít klasické události **GotFocus** (bylo vstoupeno do prvku), **LostFocus** (vystoupeno z prvku)

Label, Link Label

O prvku **Label** jsem se zmiňoval v aplikaci *Pozdrav*. Jedná se o vložení popisku do formuláře. Většinou informativní charakter.

Prvek **LinkLabel** se chová podobně jako odkaz. Využívá události **LinkClicked**, podle které se provede náležitá operace.

Button

Klasický formulářový prvek tlačítko. Pomocí události **Click** se hlídá, zda bylo stisknuto.

TextBox, RichTextBox

TextBox je textové editovatelné pole. Standardně je nastaveno jako jednořádkové, pomocí vlastnosti **Multiline** na true lze nastavit na více řádků.

RichTextBox slouží pro zobrazení dlouhých řetězců - až *2147483647* znaků.

Událost **TextChanged** hlídá změnu obsahu textového pole z obou prvků.

MainMenu

Vložení ovládacího menu do aplikace. **MainMenu** obsahuje položky typu **MenuItem**, u nichž se nastavují konkrétní akce při jejich spuštění. O možnostech jednotlivých prvků menu (horké klávesy a zkratky) jsem psal v odstavci MDI. U jednotlivých položek **MenuItem** se používá při jejich stisknutí rovněž událost **Click**.

CheckBox, RadioButton

Klasická formulářová tlačítka.

CheckBox i **RadioButton** obsahují vlastnost **Checked** označující jejich stav. Při změně stavu se vyvolává událost **CheckedChanged**.

GroupBox, Panel

Tyto formulářové prvky slouží pro logické sdružování prvků. Prvek **GroupBox** navíc obsahuje název množiny, kterou obsahuje. Nastavuje se vlastností **Text**.

DataGrid

Velice užitečný prvek sloužící pro zobrazení i úpravu tabulkových dat, například z databáze. Může se na data přímo napojit pomocí vlastnosti **DataSource**. Jako zdroj dat lze využít **DataSet**.

ListBoxy

ListBox, **CheckedListBox** uchovávají objekty nacházející se v nich v kolekcích. Objekt se do těchto prvků vloží metodou **Add()** a standardně se v těchto prvcích zobrazí s použitím zděděné metody **ToString()**. Pro přehlednější zobrazení proto je vhodné tuto metodu přepsat.

U obou prvků se nabízí událost **SelectedIndexChanged** vracející prvky, které se u vyvolání této události změnily.

ProgressBar, StatusBar

Oba prvky slouží k informování uživatele. **ProgressBar** se používá pro zobrazení pokroku prováděné aplikace zatímco **StatusBar** se hodí téměř ke všem typům krátkých informačních zpráv, jak jsme na ně zvyklí.

U těchto prvků nebude ani tak důležité znát některé jejich události. Přece jen spíše ony reagují na jiné události. Uvedu tedy vlastnosti měnící jejich stav.

StatusBar mění svůj obsah vlastností **Text**.

ProgressBar mění svou hodnotu, tedy počet "vybarvených obdélníků" chápaných jako postup operace, několika způsoby. Může se nastavit buď hodnota na pevno pomocí vlastnosti **Value** nebo ji lze zvýšit pomocí metody **ProgressBar.Increment()** o množství uvedené jako argument metody.

Bibliografie

C# pro zelenáče. Neocortex. Miroslav Virius.

C# Concisely. ADDISON-WESLEY. Judith, M. Bishop. Nigel, R. Horspool.

C# in a Nutshell. O'REILLY. Peter Drayton. Ben Albahari. Ted Neward.

HiTMilL. *C# Programming*. http://www.hitmill.com/programming/dotNET/csharp.html [???].

C# Help. http://www.csharphelp.com.

.NET Framework Community Website. *GotDotNet*. http://samples.gotdotnet.com.

IC#Code. *SharpDevelop - The Open Source Development Environment for .NET*. http://www.icsharpcode.net/OpenSource/SD/Default.aspx.

Microsoft. *MSDN*. http://msdn.microsoft.com.

interval.cz. *Vývoj .NET aplikací*. http://interval.cz/?idcategory=15&idsubcategory=157.

Principy webových služeb. http://webovesluzby.wz.cz/.

[i3] HEJLSBERG, A., GOLDE, P., KATZENBERGER, S.. *C# Version 2.0 Specification*. [online]. Microsoft Corporation. September 2005. <http://msdn2.microsoft.com/en-us/vcsharp/aa336809.aspx> [http://msdn2.microsoft.com/en-us/vcsharp/aa336809.aspx].

[i4] KAČMÁŘ, Dalibor. *Visual Studio 2005: C# 2.0*. [CD-ROM]. Microsoft s.r.o.. Září 2005. resources/VS2005-CSHARP/VS2005-CSHARP.ppt.

MATOUŠEK, Tomáš - PROŠEK, Ladislav. *Advanced C# and .NET programming: CLI 2.0 & C# 2.0*. <http://tmd.havit.cz/teaching/CSharp/Lecture12/CSharp2.pdf> [http://tmd.havit.cz/teaching/CSharp/Lecture12/CSharp2.pdf].

SESOFT, Peter. *Experience with Generic C#*. Royal Veterinary and Agricultural University and IT University of Copenhagen. Denmark: Copenhagen. .

STRAWMYER, Mark. *Generics in .NET: Type Safety, Performance, and Generality*. <http://www.developer.com/net/net/article.php/3499301> [http://www.developer.com/net/net/article.php/3499301].

WANG Annie. *Deploying Microsoft .NET Framework Version 3.0*. Microsoft Corporation. June 2006. <http://msdn2.microsoft.com/en-us/library/aa480198.aspx#netfx30_topic2> [http://msdn2.microsoft.com/en-us/library/aa480198.aspx#netfx30_topic2].

Microsoft .NET Framework 3.0 Community (NetFx3). Microsoft Corporation. <http://www.netfx3.com> [http://www.netfx3.com].

Windows Communication Foundation (WCF). Microsoft Corporation. 2006. <http://wcf.netfx3.com> [http://wcf.netfx3.com].

Windows Presentation Foundation (WPF). Microsoft Corporation. 2006. <http://wpf.netfx3.com> [http://wpf.netfx3.com].

Windows Workflow Foundation (WF). Microsoft Corporation. 2006. <http://wf.netfx3.com> [http://wf.netfx3.com].

Windows CardSpace. Microsoft Corporation. 2006. <http://cardspace.netfx3.com> [http://cardspace.netfx3.com].

[i1] SOVADINA, Jiří. *Multimediální doplňky a cvičení ke studijní opoře C#*. Ostrava. 2005. Diplomová práce v

rámci Fakulty elektotechniky a informatiky Vysoké školy báňské - Technické univerzity katedry informatiky.

[i1] TATARA, Milan. *Rozšíření výukových opor k jazyku C# o verzi 2.0*. Ostrava. 2007. Diplomová práce v rámci Fakulty elektotechniky a informatiky Vysoké školy báňské - Technické univerzity katedry informatiky.