Malý úvod do programování v .NET C#

JAROSLAV PEŇAŠKA

Obsah

Úvod

Kdysi dávno, když ještě nebyl Unix, MS-DOS ani MS-Windows, v dobách, kdy vznikaly první operační systémy a počítače byly příliš pomalé a jednoduché, se první programy psaly přímo ve strojovém kódu. Sekvenci číselných instrukcí vykonatelnou počítačem jako takzvaný spustitelný (executable) počítačový kód, vždy umí interpretovat jen konkrétní typ aritmeticko-logické jednotky neboli procesoru. Programování bylo tehdy opravdu náročnou záležitostí i pro zasvěcené.

Potom přišel Assembler (od slova assemble = sestavit), jazyk symbolických adres, který proces vytváření počítačového programu podstatně zjednodušil a poněkud zlidštil. Čísla vyjadřující konkrétní instrukce procesoru (nebo třeba pozici dat v počítačové paměti) se nahradila zástupnými symboly (slovy a zkratkami, nahrazujícími opravdovou číselnou hodnotu stejně jako třeba při počítání s proměnnými v matematice). Programy se začaly vytvářet překladem, jinak řečeno, symboly programovacího jazyka (Assembleru) se před prvním spuštěním programu nahradily skutečnými číselnými hodnotami instrukcí, dat a adres a ze zdrojového kódu (člověkem napsané, lépe čitelné, textové podoby programu) se pomocí překladače (kompileru) vygeneroval konkrétní strojový kód.

Ani tento způsob programování však ještě nebyl dostatečně jednoduchý a vhodný pro vytváření rozsáhlejších projektů. Vznikly proto další, vyšší programovací jazyky, vhodnější k výuce i práci, lépe pochopitelné a snáze použitelné k vyjádření algoritmů (řešení složitějších logických problémů v jednotlivých krocích a smyčkách programu). Zároveň ale nastala potřeba lepší definice logických celků uvnitř programu a nového způsobu organizace dat v paměti.

Vznikl objektový programovací model. Jeho nástup vyřešil spoustu problémů i nové přinesl. Bylo třeba stávající programovací nástroje přizpůsobit. Vznikly nové, čistě objektové (a často i multiplatformní) programovací jazyky, jako je JAVA, jiné byly rozšířeny o objektovou syntaxi (způsob zápisu definice objektů). Vznikly i grafické systémy pro návrh objektových struktur jako je rozšíření jazyka UML. Zrodily se také proprietární objektové knihovny jako je MFC nebo VCL. Stále ale chyběla koncepce objektové knihovny, pokud možno multiplatformní a přitom

společné pro všechny základní jazyky a operační systémy, která by programátorům přinesla potřebnou kompatibilitu zdrojových kódů a zároveň poskytla standardizované bezpečnostní prvky, nejmodernější technologie a univerzální správu aplikace.

Na tuto pozici nyní aspiruje .NET framework. Nejen, že přinesl moderní jednotnou objektovou knihovnu společnou pro více programovacích jazyků, ale navíc i multiplatformní kompatibilitu založenou na MSIL (Microsoft Intermediate Language). MSIL je jazyk podobný Assembleru a je to vlastně vyšší universální strojový kód, který je možno pružně přeložit do konkrétního strojového kódu platformy, na které má příslušný program pracovat, a to až těsně před jeho spuštěním. Myšlenka to není úplně nová, ale poprvé použitá s tak velkými ambicemi. Počítá se dokonce s nástupem moderních procesorů, schopných interpretovat přímo MSIL. Technologie .NET opravdu do jisté míry způsobila revoluci v programování. Některé programovací jazyky se ve svých objektových modifikacích a implementacích sice stále aktivně používají (Visual Basic, Delphi Pascal, Visual C++), ale současně se již zrodili nové, čistě objektově orientované jazyky, jako jsou Visual Basic.NET nebo C#.

Právě poslednímu jmenovanému se budeme v našemu úvodu do .NET programování věnovat, spojuje v sobě totiž většinu výhod nejpoužívanějších programovacích jazyků současnosti, jako je úspornost syntaxe jazyka C++ nebo přehlednost zdrojového kódu v jazyce Pascal. Tento moderní jazyk navíc umožňuje například generování dokumentace z přímo ze strukturovaných poznámek v kódu (zapisovaných v podobě XML tagů) a má i mnoho dalších příjemných vlastností.

Tato příručka zdaleka nepatří mezi vyčerpávající referenční manuály k jazyku C#. Je úmyslně koncipovaná tak, aby vás od programování neodradila, ale stala se průvodcem při prvních krocích ve vašem studiu a zároveň vám i v budoucnu poskytla rychlý přehled všech syntaktických pravidel jazyka C#. Nyní už tedy nezbývá, než vám popřát dostatek trpělivosti při studiu pravidel vytváření kódu v jazyce C# a co nejméně syntaktických chyb ve vašich prvních programech.

Základní terminologie tvorby kódu

algorithm – algoritmus (řešení problému v několika krocích)

syntax – syntaxe (způsob zápisu elementů a struktur konkrétního programovacího jazyka)

case sensitive – citlivý na velikost (C# používá case sensitivní syntaxi)

implementation – implementace (provedení, realizace funkcionality konkrétním kódem)

default – úvodní nebo také bezchybné nastavení

implicitní – automatické, předem předpokládané, standardní

explicitní – zadané (dodatečně), výslovné, určené

project – projekt, plán, úkol - v terminologii Visual Studia soubor obsahující zdrojový seznam všech souborů a pomocných dat (konfigurace projektu atd.), patřících do jednoho přeložitelného celku.

solution – řešení - v terminologii Visual Studia soubor obsahující zdrojový seznam projektů a pomocných dat (konfigurace řešení atd.), které spolu souvisí a společně se překládají.

build – přeložení (proces překladu nebo výsledek procesu překladu)

assembly – komplet (nebo také sestavení) jeden celistvý kus kódu samostatně přeložitelný jako aplikace nebo knihovna

library – knihovna (ve smyslu knihovna programových funkcí využitelná ostatními programy)

resources – zdroje (data programu specifických vlastností - texty, obrázky atd. - která jsou zpravidla umístěna a modifikovatelná samostatně) nebo prostředky operačního systému (paměť, místo na disku, vyhrazený čas procesoru atd.)

code element – základní stavební kámen programovacího jazyka, zpravidla jeden příkaz nebo jedna direktiva.

interrupt – přerušení (činnosti)

local – lokální, místní (proměnná existující jen v daném místním bloku, pro který definice platí)

global – obecné, celkové (proměnná platná všude v programu nebo alespoň jedné jeho samostatné části)

heap – halda, neboli volný paměťový prostor pro dynamickou alokaci paměti objekty a daty.

dynamic – dynamický (ve smyslu programování měnitelný za běhu programu, volně vytvořitelný ve volném místě v paměti)

static – statický (ve smyslu programování neměnný za běhu programu, pevně zasazený do určitého místa paměti)

garbage collector – automatický odstraňovač dynamicky alokovaného prostoru dále již nepoužívaných objektů

register – paměť o malém rozsahu k uchovávání malého množství informací (jednoho čísla, znaku a podobně)

registry – registr informací ve smyslu databáze operačního systému uchovávají různá nastavení programů a podobně.

command – příkaz jazyka (element jazyka, který plní nějakou funkci, která se přímo projeví jako činnost programu)

directive – direktiva jazyka nebo vývojového prostředí (element jazyka, který plní nějakou funkci, která se přímo neprojeví jako činnost programu, ale ovlivňuje kompilaci kódu, jeho napojení na jiné části aplikace, jako jsou knihovny, či ovlivňuje jeho interpretaci během kontroly správnosti syntaxe apod.)

identifier - identifikátor (slovní označení proměnné, vlastnosti, typu nebo metody v programu)

function – funkce (podprogram vracející hodnotu)

procedure – procedura (podprogram bez návratové hodnoty)

if, else – *jestliže* platí podmínka, potom proveď něco *jinak* proveď něco jiného (druh větvení)

label – návěští (místo na které se lze skokem v programu přesunout)

goto – jdi na (jdi na návěští) instrukce pro přesun místa vykonávání kódu jinam (skok na jiný řádek kódu). Používá se zejména ve spojení s dynamicky generovaným návěštím uvnitř větvení pomocí **switch**.

switch, case – *přepni* podle podmínky na *případ* (druh větvení)

for – *pro* každé x z rozsahu prováděj cyklus

while – *dokud* platí podmínka, opakuj (prováděj cyklus)

do - **while** – *dělej, dokud* platí podmínka (prováděj cyklus)

loop – smyčka, cyklus programu

branch – větev programu

break – zlomit, přerušit (násilné ukončení cyklu)

continue – pokračovat (v běhu programu tam, kde skončil)

return – vrátit se (ukončit podprogram a vrátit hodnotu)

error – chyba (zpravidla hlášení překladače, že v programu je něco nesprávně a tato chyba způsobuje neplatnost programu a nelze jej tak ani přeložit)

exception – výjimka (objekt obsahující informace o výjimečné situaci v programu způsobené obvykle chybou)

warning – varování (zpravidla hlášení překladače, že v programu je něco nesprávně, ale tato chyba nezpůsobuje nefunkčnost programu a lze jej i tak přeložit a použít)

trace – vystopovat (proces spouštění jednotlivých kroků programu (krokování) zpravidla za účelem kontroly funkčnosti a odstraňování runtime chyb a skrytých problémů)

debug – odstraňovat chyby (proces odstraňování runtime chyb a skrytých problémů v programu pomocí krokování a dalších ladících technik)

debugger – program pro odstraňování chyb (pomáhá s procesem odstraňování runtime chyb a skrytých problémů v programu pomocí krokování, trasování, vkládání break-pointů a dalších ladících technik)

break-point – bod přerušení (místo, kam byla uměle vložena instrukce přerušení běhu programu za účelem kontroly činnosti programu v tomto místě nebo krokování od tohoto místa)

layer – vrstva (databázová vrstva nebo jiná komplexní aplikační logika sloužící jako podklad pro vyšší funkční objekty)

watch – hlídat, sledovat (sledovat stav proměnné během ladění programu za účelem kontroly správnosti činností s proměnnou prováděných)

replace – nahradit (nahradit data jinými daty)

exchange – prohodit (data mezi sebou)

assignment – přiřazení (hodnoty proměnné)

initialization - inicializace (nastavení na úvodní hodnoty)

Struktura tříd a objektů

object – objekt (data (vlastnosti) a kód (metody) sdružená pospolu do paměťové struktury za účelem lepšího uspořádání logické struktury programu)

class – třída (nehmatatelný formální popis objektu, jeho vlastností a metod)

instance - instance (hmatatelná okamžitá podoba objektu vytvořeného v paměti na základě jeho popisu třídou)

method – metoda (funkce nebo procedura patřící k objektu)

constructor – konstruktor (metoda sloužící k vytvoření instance objektu)

destructor – destruktor (metoda sloužící ke zrušení instance objektu)

property – vlastnost (přístup k datům objektu skrze accessor)

accessor – metoda umožňující přístup k datům vlastnosti objektu

field – pole ve smyslu proměnná držící v objektu data vlastnosti, nepřístupná zpravidla vně objektu jinak než přes accessor.

Modifikátory přístupu

Modifikátory přístupu **(access modificators)** slouží k zabránění nežádoucího přístupu zvenčí k datům a kódu objektu. Každá třída, vlastnost, pole, typ nebo metoda může být definována včetně definice typu přístupu nebo ponechána **s** výchozím nastavením private (soukromé).

public – veřejný (veřejně přístupný, je viditelný i vně objektu – jinak řečeno tvoří rozhraní objektu)

protected – chráněné (přístupné z potomků třídy)

internal –vnitřní (pro vnitřní potřebu, přístupné uvnitř jednoho assembly)

private – soukromé (přístupné v rámci jedné třídy)

Modifikátory přístupu se uvádí na začátku programového elementu typu třída, typ, metoda, vlastnost nebo pole (proměnná pro uchování dat vlastnosti).

př.:

public int MyNumber;

protected int myNumber;

internal int myNumber;

private int myNumber;

Pravidla pro pojmenovávání identifikátorů

1. **Pravidlo pro použití velkého písmene na začátku pojmenování identifikátoru:**

Pokud je jazykový element definován s přístupem ***public*** *bude* ***jako první písmeno jeho identifikátoru použito písmeno velké abecedy****, ve všech ostatních případech bude jako první písmeno použito písmeno abecedy malé.*

2. Pravidlo pro pojmenování identifikátoru názvu pole(field):

Pokud je identifikátor názvem proměnné, která je použita jako pole vlastnosti, použije se jako ***první znak tohoto identifikátoru znak _*** *(podtržítko neboli anglicky* ***underscore****) a následující písmeno je vždy malé.*

3. Pravidlo pro psaní velkých písmen uvnitř identifikátoru:

Pokud se název identifikátoru skládá z více slov, budou všechna slova a zkratky v názvu identifikátoru spojena dohromady, napsána písmeny malé abecedy a mezery budou nahrazeny změnou velikosti po vynechané mezeře následujícího písmene na písmeno abecedy velké.

př.:

```
private int thisIsMyIdentificator; //This is my identificator
public int ThisIsMyIdentificator; //This is my identificator
private string rtfText; //RTF text
public string RtfText; //RTF text
```

Proměnná použita jako pole vlastnosti:

```
private string _rtfText; //RTF text
```

(pole vlastnosti by nemělo být nikdy public, proto je zde jen jedna varianta)

Dědičnost objektů

V obecném objektovém programovacím modelu lze objekty od sebe navzájem odvozovat, čili takzvaně dědit. Tento mechanizmus umožňuje měnit a přidávat funkcionality objektu vytvořením objektu nového typu odvozením jeho třídy od třídy původní, při zachování objektu původního v jeho původní podobě pro původní účel.

inheritance – dědičnost (vlastnost tříd umožňující dědění složitějších tříd z jednodušších)

ancestor – předek, předchůdce (třída objektu, z které byla odvozena jiná třída)

descendant – potomek, následník (třída objektu odvozená od jiné třídy)

př.: *Z obecné třídy „Útvar“ zdědíme konkrétní třídy „Kruh“ a „Čtverec“a doplníme je o funkcionalitu a vlastnosti, kterou má jen útvar kruh nebo útvar čtverec*

Přetěžování metod

Pokud je metoda v objektu definovaná jako virtuální nebo deklarovaná jako abstraktní, lze ji v potomkovi této třídy přetížit (override)

virtual – virtuální (téměř skutečná – taková metoda může být v odvozené třídě nahrazena novou metodou, rozšiřující nebo nahrazující její funkčnost jinou funkčností)

override – přetížit, překrýt původní novým (přetížená metoda nahrazuje svou funkčností funkčnost původní virtuální metody v předkovi třídy) *Takovémuto způsobu nahrazení nebo implementace metody říkáme přetížení.*

př.: *Ve třídě „Útvar“(míněno geometrický útvar) bude existovat metoda „Kresli“, kterou bude možné na objektu předka i potomka volat bez ohledu na konkrétní objekt útvar, který bude reprezentovat. Kreslení bude zajištěno již v základní třídě útvar a bude možné vykreslit některé základní tvary podle nastavení původního objektu. Ale zděděná třída ji bude využívat a poskytovat doplňkové informace použitelné při kreslení svého vlastního útvaru ve svém přetížení metody. Abychom mohli z obecné třídy „Útvar“ zdědit třídu „Kruh“ a třídu „Čtverec“, nemusíme sice metodu „Kresli“ přetížit povinně, ale můžeme pokud chceme a pokud lze metodu využít ke kreslení našeho objektu.*

Abstraktní třídy a metody

Abstract class - Abstraktní třída je taková třída, ve které je deklarována alespoň jedna abstraktní metoda. Takovou třídu nelze přímo instanciovat (vytvořit z ní v paměti instanci objektu), protože v ní chybí implementace funkcionality té abstraktní metody a kdyby se někdo tu metodu pokusil zavolat, došlo by k chybě).

Abstract method – abstraktní metoda je metoda, jejíž funkcionalita je vyžadována již v obecné třídě, ve které je deklarována, ale ještě není možné přesně tuto funkcionalitu specifikovat, protože konkrétní podoba implementace se pro jednotlivé potomky třídy příliš liší a může být tedy provedena výhradně až v potomkovi této třídy.

př.: *Ve třídě „Útvar“ bude existovat abstraktní metoda „Kresli“, kterou bude možné na objektu potomka volat bez ohledu na konkrétní objekt útvar, který bude reprezentovat. Kreslení ale zajistí až třída z útvaru odvozená. Abychom mohli z obecné třídy „Útvar“ zdědit třídu „Kruh“ a třídu „Čtverec“, musíme v obou třídách přetížit metodu „Kresli“, ale tak, že v prvním případě bude kreslit kruh a ve druhém čtverec.*

abstract – abstraktní (metoda, jejíž definice bude dodána až v potomkovi, definována je pouze hlavička s parametry a třída, ve které je abstraktní metoda použita se sama stává abstraktní třídou, čili z ní nelze zkonstruovat objekt, pouze je z ní možno zdědit jinou třídu, ve které je abstraktní metoda již nahrazena konkrétní metodou).

Struktura zdrojového kódu v C#

Každý programovací jazyk používá k zápisu svých struktur a příkazů nějakou syntaxi. Zdrojový kód jazyka C# používá v podstatě převzatou syntaxi jazyka C++ s některými elementy pozměněnými, jinými modifikovanými nebo vynechanými úplně. Také lze říci, že syntaxe jazyka C# se velmi podobá jazyku JAVA.

Dále **C#** stejně jako C++ **používá case sensitivní syntaxi**, rozlišuje tedy v kódu malá a velká písmena a identifikátory které se od sebe liší i pouze použitím malých a velkých písmen považuje za dva různé identifikátory. ***př.:*** kresli a Kresli budou pro C# dvě různé funkce.

Zdrojový soubor jazyka C# obvykle sestává z těchto částí:

__Import jmenných prostorů__ častěji použitých v elementech kódu definovaného v tomto souboru.

Definice __jmenných prostorů__ do kterých spadají elementy definované v tomto souboru (obvykle jen jeden jmenný prostor)

Definice __regionů__ v tomto souboru

Definice __typů__

Definice __tříd__ objektů

Definice __polí, vlastností a metod__ tříd definovaných v tomto namespace

Tuto strukturu si dále ukážeme na příkladu souboru zdrojového kódu. A protože se při praktickém programování neobejdeme bez znalosti anglického jazyka a je dobrým zvykem, že do zdrojového kódu se píše jen anglicky, je-li to možné, začneme hned od začátku psát komentáře i názvy identifikátorů v angličtině, abychom se později ve skutečném zdrojovém kódu lépe zorientovali. Následuje tedy příklad souboru zdrojového kódu jazyka C#, který si dále podrobně rozebereme.

```
//Prefixes of the class library namespaces
using System;

//Namespace in which this class name is unique
namespace DemoClassLibrary
{
    /// <summary>
    /// Description for the DemoClass object
    /// </summary>
    public class DemoClass
    {
        //property field definition
        private string _demoData = "";

        /// <summary>
        /// Demo data property definition
        /// </summary>
        public string DemoData
        {
            //Accessor for the getting of data
            //returns the value of the property
            //stored in field by the keyword
            //return
            get
            {
                //returning value
                //of the field _demoData
                return _demoData;
            }

            //Accessor for the setting
            //of the data contains the variable
            //containing data and named
            //by the predefined keyword value
            set
            {
              //setting of the field _demoData
               _demoData = value;
            }
```

```
}

/// <summary>
/// This method displays demodata value
/// </summary>
public void DisplayDemoData()
{
    DisplayDemoData(_demoData);
}

/// <summary>
/// This method Displays demodata value
/// as a message form
/// </summary>
private void DisplayDemoData(string msg)
{
   Forms.MessageBox.Show(msg,"Info");
}
```

```
        /// <summary>
        /// Parameterless constructor
        /// </summary>
        public DemoClass()
        {
            //
            // TODO: Add constructor logic here
            //
        }

        /// <summary>
        /// Parametric constructor of the class
        /// </summary>
        public DemoClass(string demoData)
        {
            //Field initialization
         _demoData = demoData;
        }
    }
}
```

Import jmenných prostorů

Prvním elementem, který vidíme na začátku souboru je direktiva: ***using*** *System*;

System je zde název takzvaného jmenného prostoru. Následuje jednoduchá definice pojmu:

namespace – *jmenný prostor (pojmenovaný blok ohraničující a zahrnující prostor, ve kterém je identifikátor unikátní)*

Každý soubor zdrojového kódu C# začíná zpravidla deklarací těch jmenných prostorů, které chceme dále v kódu používat a zároveň se nám je nechce později stále dokola vypisovat v dlouhé formě zápisu, tedy uvozovat každý použitý identifikátor prefixem namespace, ve kterém se vyskytuje.

V našem případě jsme importovali konkrétně dva jmenné prostory pomocí následujících direktiv:

```
using System;
using System.Windows;
```

Zároveň tím říkáme, že z těchto namespace chceme používat objekty (třídy) a typy. Existuje-li tedy objekt, který chceme v programu použít v některém z uvedených jmenných prostorů za direktivou **using**, bude použit v případě holého zápisu identifikátoru dále v kódu ten objekt, který je znám, tedy vyskytuje se v některém z importovaných jmenných prostorů (**namespace**).

př.: Uvedeme na začátku kódu direktivu:

using *System.Windows;*

...a dále v kódu pak již nemusíme psát například:

System.Windows.Forms.MessageBox.Show(message,"Test");

…ale stačí napsat jen:

Forms.MessageBox.Show(message,"Test");

Definice jmenného prostoru

Dále každý soubor zdrojového kódu C# zpravidla obsahuje definici bloku jmenného prostoru (namespace), do kterého spadají třídy v tomto souboru deklarované. Pokud spadají tyto třídy do různých jmenných prostorů, může být v souboru těchto definic bloku jmenného prostoru rovněž několik.

V našem případě obsahuje soubor jeden namespace pod názvem **DemoClassLibrary**.

```
namespace DemoClassLibrary
{
}
```

Vidíme, že tato direktiva není ukončena středníkem, ale znaky **{ }**. Tyto znaky znamenají definici začátku a konce bloku. Namespace je tedy platný pro všechny definice a deklarace uvedené uvnitř tohoto bloku. Povšimněme si také, že za definicí bloku ani identifikátorem jména namespace se v tomto případě neobjevil středník. Pojďme si tedy pravidla pro definici bloku a ukončení příkazu objasnit.

Definice bloku a standardní ukončení elementu jazyka C#

Jak jsme si již řekli u popisu definice namespace, v C# se používají k určení rozsahu platnosti části kódu znaky pro definici bloku: **{** (začátek bloku) a **}** (konec bloku).

Pokud je za příkazem nebo direktivou jazyka C# uvedena definice bloku, pak je příkaz nebo definice platný/á pro celý tento blok nebo v celém jeho rozsahu. V takovém případě se příkaz jazyka C# neukončuje znakem **;**

Ve všech ostatních případech se element jazyka C# (příkaz, definice, deklarace) ukončuje středníkem!

Definice poznámky ve zdrojovém kódu C#

Kód dále pokračuje poznámkou. Poznámky ve zdrojovém kódu při programování slouží obecně k lepší orientaci v kódu. V jazyce C# však byl zaveden systém strukturovaných poznámek, které dokonce umožňují přímé generování dokumentace kódu a její zobrazování jako nápovědy v intelisense systému prostředí Visual Studio .NET i IDE jiných výrobců. Před každý element kódu s přístupem typu **public** je dokonce vložení poznámky při zapnutí funkce generování dokumentace **povinné** a při překladu kódu, ve kterém poznámka před public elementem není nás kompiler upozorní hláškou typu **warning**.

př.: Definice strukturované poznámky elementu kódu:

```
/// <summary>
/// Parametric constructor
/// </summary>
/// <param name="demoData"></param>
```

Jak vidíme, strukturovaná poznámka obsahuje XML tagy, jež označují typ dat v poznámce zapsaných. Tag *summary* označuje celkový popis elementu, tag *param name* uvozuje popis parametrů funkce atd.

(Ještě na závěr doplním, že **po napsání třech lomítek v prostředí Visual Studio .NET se automaticky vytvoří příslušná kostra strukturované poznámky do které je možno dopsat text.**

Překlad napovědy se ve Visual Studiu **aktivuje zadáním názvu cílového XML souboru ve vlastnostech projektu** v položce **Configuration Properties** v podpoložce **Build** do editačního pole **XML Documentation File**).

Definice třídy v jazyce C#

Třída je základním elementem každého objektově orientovaného jazyka. Je to vlastně popis struktury objektu a definice jeho vlastností a metod. Podívejme se tedy na konkrétní syntaxi definice třídy v jazyce C#.

př.: V našem ukázkovém kódu máme definovánu třídu DemoClass a to následujícím způsobem:

```
public class DemoClass
{

}
```

Budeme-li vyhodnocovat výraz z levé strany, jedná se tedy o element veřejný (public), viditelný i vně této assembly. Dále je zde řečeno, že se jedná o element typu class (třída) a že bude nazýván, tedy odlišen od ostatních, identifikátorem **DemoClass**.

Dále následuje definice bloku třídy **{}**, který zde zároveň vyjadřuje rozsah objektu. Uvnitř tohoto bloku pak budou definovány všechny nové vlastnosti a metody této třídy. Proč zdůrazňuji, že nové? Protože pokud bude tato třída zděděna z jiné třídy, pak bude automaticky obsahovat i vlastnosti a metody nedefinované přímo v jejím bloku, ale i ty definované v jejím předkovi, čili třídě, ze které je odvozena, aniž by bylo nutné je znovu deklarovat, či dokonce znovu definovat. Definice takové zděděné třídy by potom vypadala takto:

```
public class DemoClass : OriginalDemoClass
{

}
```

Takováto třída by potom měla všechny vlastnosti a metody své, plus ty zděděné z jejího předka, např. třídy OriginalDemoClass.

Definice interface

Pokud chceme mít možnost použít ve stejné situaci několik různých objektů a ty spolu sdílí jen některé vlastnosti a metody, ale nemají společného předka, použijeme jako proměnnou držící objekt interface neboli rozhraní objektu. Definice interface vypadá stejně jako definice třídy, s tím rozdílem, že se zde pouze deklarují již existující vlastnosti a metody objektů, ke kterým lze přes tento interface přistupovat. Třídy, které interface implementují jej pak v jazyce C# musí mít ve své definici za jménem předka, ze kterého jsou odvozeny, ještě v seznamu interface, které implementují, uveden tento interface. Každá metoda jež implementuje některou metodu interface se musí jmenovat stejně, jako metoda v interface, kterou implementuje.

př.: *Definice interface*

```
/// <summary>
/// Common interface for some classes
/// sharing the same methods and properties
/// </summary>
public interface IThisIsInterface
{
    /// <summary>
     /// Some property of the classes sharing this
interface
    /// </summary>
    int MyProperty
{
    //Signalization, which accessors will be used
     get;
     set;
}

/// <summary>
     /// Some method of the classes sharing this
interface
    /// </summary>
void Draw();
}
```

***př.:** Implementace interface ve třídě*

```
public class DemoClass : OriginalDemoClass,
IThisIsInterface
{
    public void Draw()
    {
        DrawSomething();
     }
}
```

Datové a objektové typy

Následuje výčet některých klíčových slov a symbolů pro práci s typy v jazyce C#:

type – typ dat(programový element, popisující formát dat proměnné nebo jiného datového objektu použitého v programu)

enum (enumerable) – výčtový typ(typ, jehož hodnoty jsou vyjmenovány a přímo pojmenovány místo určení prostého rozsahu hodnot)

object – všechny třídy a většina typů jsou od něj v C# (respective v celém .NET framework) odvozené.

struct (structure) – struktura je datový typ sestávající z pojmenovaných proměnných různého typu

[ ] (array) – datový typ pole sestávající z indexovaného seznamu proměnných stejného typu

Přiřazení hodnoty proměnné

Při programování se pracuje stejně jako v matematice s proměnnými. Každá proměnná je určitého datového typu – může jí být přiřazena hodnota jen z určitého definičního oboru.

Nejužívanějším datovým typem v C# je pravděpodobně **integer** čili celé záporné nebo kladné číslo v 32-ti bitovém rozsahu

(-2147483648 až +2147483647).

V programovacím jazyce C# lze proměnnou vytvořit následující syntaxí: **typ_proměnné identifikátor_proměnné**

př.:

int *myVariable;*

Proměnné lze přiřadit hodnotu následujícím způsobem

př.:

myVariable = 2;

Proměnné lze přiřadit hodnotu již během její deklarace, neboli ji takzvaně inicializovat na konkrétní hodnotu

př.:

int *myVariable = 2;*

Základní datové typy v C#

byte, int (integer), unsigned short int(**word**)**, long int**

celočíselné datové typy

float, double

datové typy s plovoucí desetinnou čárkou (pro vyjádření racionálních čísel)

DateTime

Datum a čas

char, string

znakové datové typy pro uchování znaků a textů

bool (boolean)

výčtový binární datový typ sloužící k uchování logické hodnoty **true**/**false** (pravda/lež)

void - prázdný typ (návratový typ pro procedury v C#, C++ a podobných jazycích, kde jsou procedury implementované jako funkce nevracející hodnotu – tedy jejich návratový typ je definován jako prázdný)

Konverze mezi jednotlivými datovými typy

Pokud se datový typ proměnných jen mírně liší a chceme jim navzájem přiřazovat jejich hodnoty, použijeme takzvaný **type-casting** čili přetypování proměnné. V C# se přetypování provádí uvedením typu, na který proměnnou přetypováváme do kulatých závorek před její jméno.

př.:

int smallNumber = 1;

long bigNumber = **(long)** smallNumber;

Pole hodnot (array)

Pole hodnot je posloupnost hodnot stejného typu, kde je možno se na jednotlivé prvky pole odkazovat přes jejich pořadové číslo v této posloupnosti neboli **index**.

syntaxe: ***název_typu*** **[ ]**

př.: byte[ ] *myBytes =* ***new*** *byte[3];*

myBytes[0] = 255;

myBytes[1] = myBytes[0];

Pole hodnot lze inicializovat i zkráceným zápisem, kde hodnoty, kterými chceme pole na začátku na plnit uvedeme do bloku hned za příkaz vytvoření instance pole.

př.:

string[ ] myString = **new string** [2] {“A”,”B”};

Typ struktura

Kromě pole existuje ještě další datový typ schopný držet více prvků a tím je struktura. Jeho hodnoty nejsou očíslované a přístupné pomocí indexů, ale pojmenované a mohou být různého typu.

Struktura se v jazyce C# definuje klíčovým slovem **struct**.

př.:

```
public struct Man
{
   string FirstName;
   string Surname;
   DateTime BirthDay;
}
```

Definice výčtového typu v C#

enum – výčtový typ je zvláštní druh datového typu, kde si jednotlivé hodnoty můžeme pojmenovat identifikátory.

př.:

```
public enum FourSeasons
{
        spring = 1,
        summer = 2,
        autumn = 3,
        winter = 4
}
```

Tento datový typ pak můžeme dále v programu používat a přetypovávat na něj ordinální datové typy (například integer)

Definice indexeru

Indexer se používá k indexování položek vlastností. Indexer pak umožňuje přistupovat k vlastnosti jako by sama byla typu pole.

př.:

```
private byte[ ] _workBytes;

public byte WorkBytes[int index]
{
	get
	{
		return _workBytes[index];
	}
}
```

Kolekce

Collection - kolekce je druh objektu do kterého lze lehce dynamicky přidávat další uživatelsky definované prvky a provádět s nimi i jiné operace. Dá se většinou také indexovat (podobně jako proměnné typu pole - array) nebo iterovat pomocí příkazu **foreach** (v případě, že má kolekce definován iterátor). Na rozdíl od pole je kolekce složitější objekt, který může v sobě skrývat řadu uživatelský užitečných funkcí pro práci s jeho prvky.

Operátory v C#

Většina operátorů je v C# převzata z jazyka C++ a variant jeho syntaxe.
V C# jsou tedy definovány následující operátory:

Aritmetické: **+ - * / %**
Logické (booleovské a bitové): **& | ^ ! ~ && || true false**
Pro slučování řetězců: **+**
Zvyšování a snižování hodnoty proměnné: **++ --**
Operátory bitového posunu: **<< >>**
Operátory relační: **== != < > <= >=**
Přiřazení: **= += -= *= /= %= &= |= ^= <<= >>=**
Přístupu ke členům: **.**
Indexování: **[ ]**
Přetypování: **()**
Operátor podmíněné hodnoty: **? :**
Operátor pro slučování a odstraňování delegátů: **+ -**
Operátor pro vytváření instancí objektů: **new**
Operátor pro získání informace o typu: **as is sizeof typeof**
Operátor pro aktivaci kontroly přetečení: **checked unchecked**
Operátor indirekce a získání adresy: *** -> [ ] &**

Definice metody v jazyce C#

Většina programovacích jazyků definuje svou funkcionalitu ve funkcích a procedurách kterým se v objektové terminologii říká metody.

Podíváme-li se znovu na náš příklad kódu, vidíme, že metody se v jazyce C# definují například v následující syntaxi:

```
public void DisplayDemoData()
{
   DisplayDemoData(_demoData);
}
```

Definice metody začíná modifikátorem přístupu (zde **public**).

Může následovat údaj o dědičnosti (zde chybí, protože metoda je v C# ve výchozím stavu (defaultně) virtual, ale na této pozici se může vyskytovat například klíčové slovo **virtual, abstract, override**)

Následuje deklarace návratového typu (zde **void**, protože se jedná o proceduru, čili podprogram nevracející žádnou, respektive prázdnou hodnotu)

Poté definujeme identifikátor, tedy jméno metody (zde **DisplayDemoData**), který by měl vždy vystihovat co nejstručněji funkcionalitu v metodě implementovaného kódu.

Následuje blok definice seznamu parametrů (znak začátku **(** a znak konce bloku **)** definice parametrů) (zde prázdný, neboť tato metoda nemá parametry)

Poté následuje definice bloku kódu (**{ }**) nebo také jinak řečeno rozsahu metody.

V bloku samotném je pak zapsán kód metody, v našem případě volání jiné metody **DisplayDemoData(_demoData)**, která na rozdíl od naší, očekává nějaké vstupní parametry, proto, i když se jmenuje stejně, jedná se o jinou metodu (bude vysvětleno dále). Každý programový element v bloku je ukončen středníkem, pokud sám nedefinuje svůj vlastní blok.

Návratová hodnota metody

Metody jazyka C#, které vracejí výsledek (tedy jsou to funkce) mají před jménem deklarován návratový typ (např. int) a uvnitř funkce se musí alespoň jednou vyskytovat klíčové slovo **return** (vrať výsledek) následované hodnotou, která má být vrácena kódu, jenž naši metodu volá.

př.:

```
private void startDisplayData()
{
    bool result = DisplayDemoData();
}

public bool DisplayDemoData()
{
    DisplayDemoData(_demoData);
    return true;
}
```

Definice přístupových metod vlastností

Přístupové metodě vlastnosti objektu říkáme **accessor**

Rozlišujeme dva typy aksesorů:

get – accessor typu získej (vrací data z pole vlastnosti)

set - accessor typu nastav (zapisuje data do pole vlastnosti)

Pro účely definice aksesoru typu set je v rozsahu jeho bloku definována proměnná **value** (hodnota) držící data zapisovaná do pole vlastnosti nebo jinak manipulovaná uvnitř aksesoru.

př:

```
public string Name
{
    get
    {
        return _name;
    }

    set
    {
        _name = value;
    }
}
```

Definice konstruktoru objektu

Constructor - konstruktor objektu je metoda, která ze třídy objektu vytvoří v paměti instanci objektu, čili z abstraktního popisu zkonstruuje skutečný, hmatatelný objekt, kus paměti obsahující data a strojový kód.

př.:

Bezparametrický konstruktor (parametter-less) naší třídy DemoClass

```
/// <summary>
/// Parameterless constructor of the class
/// </summary>
public DemoClass()
{
    //
    // TODO: Add constructor logic here
    //
}
```

př.:

Parametrický konstruktor (parametric) naší třídy DemoClass

```
/// <summary>
/// Parametric constructor of the class
/// </summary>
public DemoClass(string demoData)
{
    //Initialization of the field
    _demoData = demoData;
}
```

Definice destruktoru objektu

Destructor - destruktor objektu je metoda, která objekt z paměti odstraní a dealokuje jeho prostředky. V C# se automaticky volá implicitní destruktor, který existuje v každém objektu a explicitně se destruktory vytváří jen ve výjimečných případech (nutnost rychlé dealokace prostředků objektu bez čekání na garbage collector). Destruktor se označuje znakem **~** (tilda) na začátku jeho identifikátoru.

př.:

Bezparametrický destruktor (parameterless) naší třídy DemoClass

```
/// <summary>
/// Parameterless constructor of the class
/// </summary>
public ~DemoClass()
{
   //
   // TODO: Add destructor logic here
   //
}
```

Vytvoření instance objektu

Vytvoření instance objektu je proces, kdy konstruktor objektu vytvoří v paměti skutečný, hmatatelný objekt, jinými slovy kus paměti obsahující data a strojový kód z jeho abstraktního popisu čili třídy. Jakmile je instance objektu vytvořena, můžeme volat jeho metody.
př.:

```
public void CreateDemoObject()
{
    DemoClass myDemoClass = new DemoClass();
    myDemoClass.DisplayDemoData();
}
```

Větvení programu pomocí příkazu if

Pokud se v programu máme rozhodnout na základě nějaké podmínky, použijeme příkaz programového větvení. V C# je to příkaz **if/else**.

př.:

```
public bool AreOperandsEqual(int a, int b)
{
    if (a == b)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

Zcela nově oproti ostatním jazykům existuje v C# příkaz podmíněné hodnoty vyjádřený znakem **?** (**question mark** - otazník) a **:**(**colon** - dvojtečka). Pokud je podmínka platná, vrátí výraz jako hodnotu operand před dvojtečkou jinak vrátí jako hodnotu operand za dvojtečkou.

př.:

```
public int ReturnBiggerValue(int a, int b)
{
    return (a > b)? a:b;
}
```

Větvení programu pomocí příkazu switch

Dalším způsobem větvení programu jsou dynamické skoky na návěští **case** (případ) vytvářená z hodnoty řídící proměnné bloku **switch** (přepínač). Do závorek za příkaz **switch** vložíme řídící proměnnou a do jednotlivých bloků **case** píšeme před dvojtečku jako návěští hodnotu řídící proměnné, pro kterou je případ platný a za dvojtečkou pak pokračujeme blokem příkazů, který se má v tomto konkrétním případě provést. Toto je jedna z výjimek, kdy se v jazyce C# nepoužije pro definici bloku znaků **{}**, ale blok příkazů začíná pouze návěštím **case**, pak následují příkazy ukončené středníky a celý podprogram je buď ukončen pomocí **break** či **return** nebo pokračuje na jiné návěští příkazem **goto**. Tato syntaxe je dána nepřímo tím, že switch lze v podstatě chápat jako podprogram.

př.:

```
public bool IsItCharAB(char unknownChar)
{
    switch(unknownChar)
    {
        case 'a':
        goto 'A';
        case 'A':
        break;
        case 'b':
        break;
        case 'B':
        return true;
        default:
        return false;
    }
    return true;
}
```

Cyklus v programu

V C# rozeznáváme tři druhy cyklu: **FOR, WHILE, FOREACH**

FOR - *pro* definovaný rozsah hodnot opakuj cyklus

```
for (int i = 0;i<10;i++)
{
	x+=(i);
}
```

WHILE - *dokud* je splněna podmínka, prováděj cyklus

```
while (i<10)
{
	x+=(i);
	i++;
}
```

FOREACH - *pro každou* položku pole nebo kolekce opakuj cyklus

```
foreach (string myString in myStrings)
{
	if (myString == hledanyRetez) break;
}
```

Statické metody a vlastnosti

Metody a vlastnosti označené v C# klíčovým slovem **static** lze použít bez nutnosti vytvořit instanci třídy, ve které jsou definovány. Statická vlastnost je navíc sdílena všemi instancemi třídy, ve které je deklarována. Statická metoda nebo vlastnost však nemá přístup k dynamickým funkcím a vlastnostem třídy na které je definována, pokud sama v sobě nevytváří instanci této třídy. To je jeden z obvyklých způsobů využití statických metod a vlastností. Pokud má objekt nějakou zajímavou metodu, ale nechceme jej kvůli jejímu využití pokaždé ručně instanciovat, můžeme v něm nadefinovat statickou metodu, která nám objekt vytvoří a zprostředkuje výsledek funkce, kterou chceme využít.

př.:

```
public static void DrawDemoData()
{
    DemoClass myDemoClass = new DemoClass("Hello");
    myDemoClass.DisplayDemoData();
}
```

Statickou metodu pak lze z jiných objektů volat bez nutnosti vytvářet instanci jejího objektu.

```
public void SomeOtherObjectMethod()
{
    DemoClass.DrawDemoData();
}
```

Statický konstruktor

Specifickým typem statické metody je statický konstruktor. Nevolá se ručně v programu, ale automaticky při prvním vytvoření instance dané třídy. Umožňuje tak inicializovat (nastavit na úvodní hodnoty) statické vlastnosti této třídy. Vhodným využitím je například načtení konfigurace objektu z databáze apod. – obecně činnosti, které se v programu mají provést jen jednou, společně pro všechny budoucí instance této třídy, nikoliv při každém jednotlivém vytvoření instance třídy.

př.:

```
private DbSetting _demoClassSetting;

public static DbSetting DemoClassSetting
{
    get
    {
        return _demoClassSetting;
     }
}

public static DemoClass()
{
 _demoClassSetting = new DbReader()).ReadDbSetting();
}
```

př.:

Máme dánu třídu, ze které se budou tvořit instance. Ale tyto instance budou všechny používat nějaké nastavení, například z databáze, které chceme načíst jen jednou, a to při vytvoření první instance, nikoliv při každém vytvoření jednotlivé instance. Nadefinujeme si tedy v této třídě statický konstruktor a nebudeme ho nikde volat, protože se volá sám při prvním vytvoření instance z této třídy. Načteme v něm například nějaké nastavení a zapíšeme je do statické vlastnosti této třídy. No a všechny instance této třídy pak používají toto jedno společné nastavení, protože všechny přistupují ke společné statické vlastnosti na jejich třídě, nikoliv ke svým specifickým datům.

Procedurální typ delegát

Obecně existuje v různých programovacích jazycích typ, kterému se říká procedurální. Je to typ schopný držet v sobě odkaz na proceduru (podprogram, který nevrací hodnotu), tedy v podstatě adresu vykonatelného kódu. Proměnnou takového typu lze potom vykonat jako kód. Změnou obsahu takové proměnné tak, aby odkazovala na jinou proceduru je pak možno dynamicky měnit běh kódu a volat na stejném místě pokaždé jiný podprogram. Definice procedurálního typu v podstatě definuje prototyp hlavičky procedury, pro jejíž volání má být proměnná tohoto typu určena a na kterou je pak schopna v sobě držet odkaz. V jazyce C# a obecně .NET (který je striktně typový a objektový) je procedurální typ, (jež je ve většině starších jazyků definován v podstatě jako **pointer** - ukazatel na proceduru) nahrazen typem **delegate** – delegát. Název tedy vystihuje, že jde o odkaz na proceduru, u níž očekáváme, že přijde jen „na návštěvu“. V .NET a C# je hlavním způsobem využití delegátů definice událostí, o nichž se zmíníme dále.

př.:
*V C# se delegát definuje pomocí klíčového slova **delegate** třeba takto:*

```
public delegate void MyDelegate(string someParameter);
```

Hlavička metody (tedy typy a pořadí parametrů) pak musí odpovídat hlavičkám metod, které budeme chtít do proměnné tohoto typu zapisovat.

Události v programu

Kromě klasických vlastností, obsahujících běžná data, existuje v objektovém modelu programování také zvláštní druh vlastnosti, která se nazývá **event** – událost. Je to v podstatě vlastnost, jejíž data drží proměnná procedurálního typu (v .NET a C# typu **delegate** - delegát). Událost tedy uchovává v proměnné typu delegát (respektive v poli delegátů) ukazatel na metody cizího objektu, která se má vykonat, pokud dojde v našem objektu, k určité události, například stisku klávesy. Volané metody navíc lze dynamicky měnit a propojovat tak objekty událostmi i za běhu programu. Stejně jako vlastnost, používá i událost k uložení svých dat většinou datové pole (field), tedy nějakou obvykle privátní proměnnou stejného typu a přistupuje k ní přes přístupové metody. Ty se však v dynamicky definované události nenazývají get a set, ale **add** (přidej) a **remove** (odstraň) a vnitřně většinou využívají metody **Combine** (zkombinuj) a **Remove** (odstraň) nebo operátory **+=** a **- =** pro možnost volání několika metod současně při jedné události. *V C# se události definují pomocí klíčového slova **event**.*

př.:

```
private MyDelegate _myEvent;

/// <summary>
/// This is the event definition
/// </summary>
public event MyDelegate MyEvent
{
 add
 {
                          _myEvent                              =
(MyDelegate)System.Delegate.Combine(_myEvent, value);
 }
 remove
 {
                          _myEvent                              =
(MyDelegate)System.Delegate.Remove(_myEvent, value);
 }
}
```

Ošetření výjimek

Pokud v programu nastane situace, na kterou není program přímo připraven reagovat normálními mechanismy, dojde k vytvoření **exception** – objektu výjimky, který nese o této situaci informace. Tento objekt je možno na různých úrovních aplikace zachytit a informace v něm uložené zpracovat. Výjimka se pak neprojeví jako tolik známé chybové hlášení uživateli, které všichni uživatelé z duše nenávidí, protože je většinou následováno pádem aplikace. Objekt výjimky se vyskytuje v několika variantách a může být odvozen od základní třídy výjimky, stejně jako od některého jejího potomka (applicationException atd.) Pro zachycení objektu výjimky na jeho cestě ven z aplikace lze v jazyce C# použít příkazu **try** (zkus) pro ohraničení nebezpečného bloku, ve kterém možnou výjimku očekáváme v kombinaci s příkazem **catch** (chyť) následovaným blokem pro její zpracování nebo příkazu **finally** (na závěr) následovaného blokem, který nezbytně nutnou činnost v něm definovanou vykoná bezpečně vždy, ať už k výjimce došlo nebo ne (například uzavře otevřené soubory atd.).

př.:

```
try
{
    myFile.Read();
}
catch (e exception)
{
    SaveToLog(e.Message);
}
finally
{
    myFile.Close();
}
```

Výjimku lze také úmyslně vyvolat a použít jako kontejner pro svoji vlastní chybovou informaci, kterou chci předat do vyšší vrstvy aplikace a to příkazem **throw**(hodit).

př.:

throw new exception(“Hi!“);

Komponenty a ovládací prvky

Při tvorbě aplikací s uživatelským rozhraním se často využívají programové stavební kameny, kterým se říká **component** (komponenta). Jsou to samostatné funkční celky s vlastním rozhraním tvořeným vlastnostmi označenými atributy před definicí vlastnosti, které umožnují jejich editaci již během designování aplikace a jsou díky tomuto označení například viditelné ve Visual Studiu v okně vlastností (**properties**) se dosáhne použitém atributu **[Browsable(true)]** před její definicí.

Dále je možné definovat kategorii vlastnosti (**[Category(“Appearance”)]**) nebo třeba popis (**[Description(“This is the property”)]**). Specifickým typem komponenty je **control** (ovládací prvek). Ovládací prvky je možné umísťovat na formulář a mají své vlastní grafické uživatelské rozhraní (zobrazují sami sebe na formuláři a reagují například na kliknutí myší (ovládací prvek tlačítko) nebo přímo obsahují jiné komponenty jako jsou tlačítka, obrázky a podobně. Zatímco komponenty se dědí z jejich základní třídy **Component** a jejich potomků, ovládací prvky se dědí z třídy **Control** jež **je také jedním z nepřímých potomků třídy Component**. Komponenty i ovládací prvky lze do projektu ve Visual studiu lehce přidat kliknutím na **Add Component...** nebo **Add User Control...** v menu **Project**

př.:

Upravíme naši UserControl tak, že ji zdědíme z ovládacího prvku Button

```
public class MyTestControl : System.Windows.Forms.Button
```

Přidáme vlastnost Counter viditelnou v Properties window

```
private int _counter;
[Description("Counter of the clicking")]
[Category("Appearance")]
[Browsable(true)]
public int Counter
{
    get
    {
        return _counter;
    }
    set
    {
        _counter = value;
    }
}
```

Použití více vláken v programu

Pro některé úkoly je třeba více než jeden programový **thread** (vlákno nebo jinými slovy posloupnost provádění instrukcí programu procesorem). Pokud chceme například v jedné aplikaci provádět nějaké zpracování dat a souběžně průběžně zpracovávat jiná data, mohli bychom toho dosáhnout právě vytvořením dalšího vlákna programu. Oboje zpracování dat pak poběží současně a procesor bude mezi vlákny velmi rychle přepínat a přidělovat jim svůj čas podle jejich priority a dalších okolností. V ideálním případě na víceprocesorovém systému by dokonce mohlo běžet každé vlákno na svém vlastním procesoru. Oba procesy zpracování budou tedy probíhat víceméně paralelně a nebudou na sobě navzájem přímo závislá. Takových to vláken lze spustit tolik, kolik nám prostředky systému dovolí. Konkrétní implementace podpory vláken v C# je velmi jednoduchá. Prostě vytvoříme objekt typu Thread a v jeho konstruktoru mu předáme metodu našeho objektu, u které požadujeme, aby běžela v separátním vlákně. Poté spustíme vlákno metodou Start na objektu typu Thread. Vlákno lze také uspat metodou **sleep** na dobu potřebnou pro vykonání něčeho jiného.

př.:

```
private Thread t;

private void threadMethod()
{
    //TODO: Add thread logic here
}

private void runThread()
{
    t = new Thread(new System.Threading.ThreadStart(threadMethod));
    t.Start();
}
```

Synchronizace více vláken

V případě, že hrozí kolize dvou vláken na jednom místě programu (třeba uvnitř metody nebo vlastnosti), můžeme vlákna synchronizovat příkazem **lock** (zámek) následovaným v kulatých závorkách názvem objektu, který má být uzamknut a definicí bloku příkazů, při jejichž vykonávání má být objekt uzamčen.

př.:

```
private void synchronizedMethod()
{
    lock(mySynchronizedObject)
    {
        //TODO: Add synchronized logic here
    }
}
```

Zpracování programu počítačem

Počítač zdrojový kód programovacího jazyka přeloží na kód strojový a teprve potom je schopen jej vykonat. Skutečná interpretace strojového kódu je složitý proces a plně jí porozumět znamená věnovat se studiu této problematiky více do hloubky, ale i mlhavá představa o interpretaci programu počítačem je vždy užitečná a to i pro programátora, programujícího ve vysoko-úrovňovém (high-level) jazyku, jakým je C#. Proto se na závěr budu ještě chvíli věnovat rozboru zpracování programu počítačem.

ALU - arithmetic logic unit neboli CPU

BUS - adresní sběrnice, datová sběrnice

IRQ - Interrupt Request – požadavek na přerušení právě probíhající činnosti procesoru

DMA – Direct Memory Access

port - brána vstupní/výstupní

memory - paměť

I/O - Input/Output - vstup/výstup

call - instrukce volání kódu

LS - least significant – s nižší hodnotou - méně významný byte

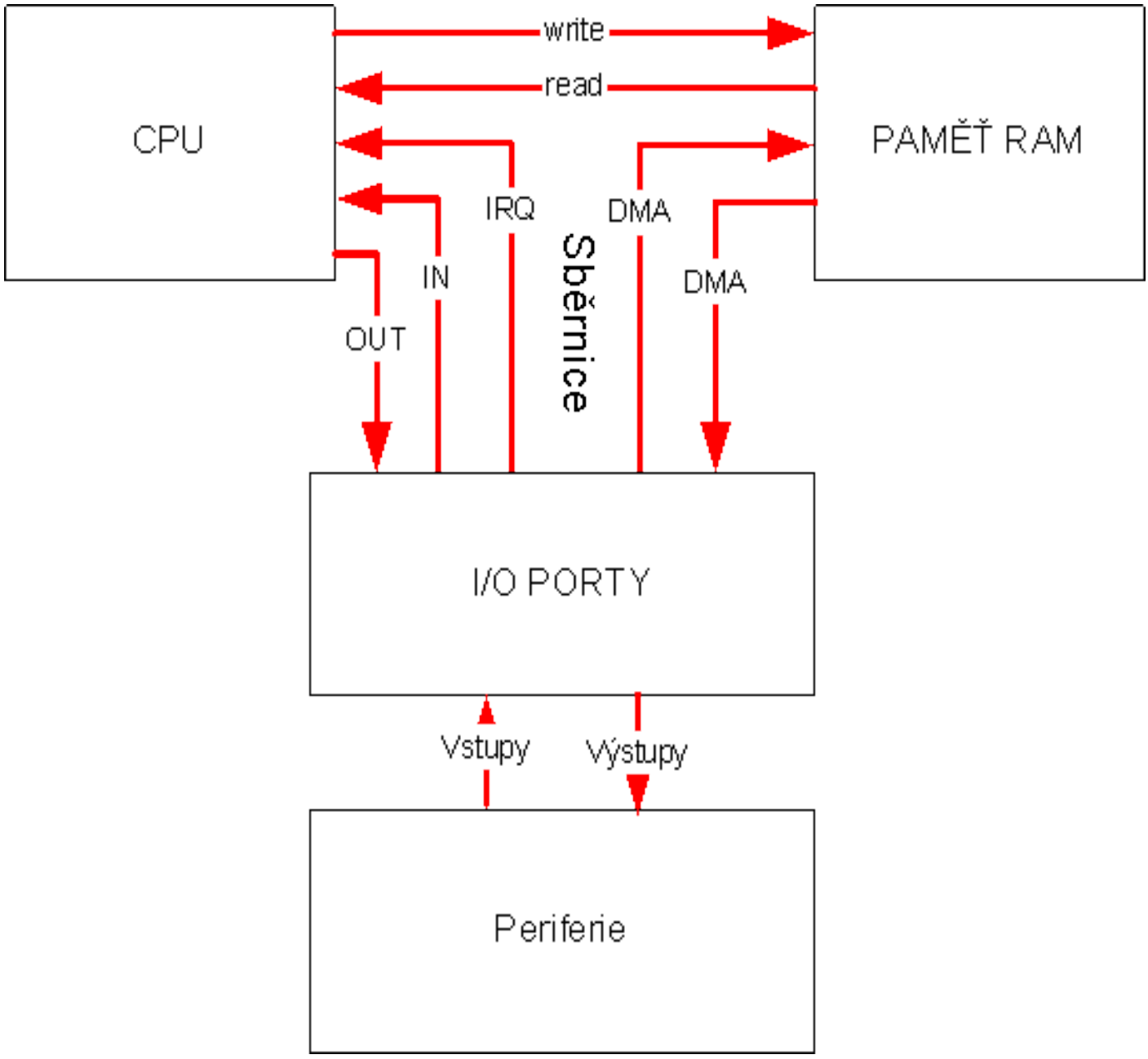
MS - most significant - s vyšší hodnotou - významnější byte

stack – zásobník

Datový model počítače

Počítačový program je ve skutečnosti sled čísel, která vyjadřují instrukce procesoru a data, se kterými instrukce provádějí operace. Data se k procesoru dostávají po datové sběrnici a to z paměti nebo portů. Adresní sběrnice pak umožňuje procesoru a vstupně výstupním zařízením se v paměti a prostoru portů orientovat. Při volání podprogramů se adresy ukládají na zásobník a při návratu se zase vyzvedávají. Rovněž parametry funkcí se v interpretaci některých programovacích jazyku ukládají na zásobník nebo častěji do volných registrů procesoru.

Obecné schéma počítače



CPU neboli procesor

Procesor (CPU = central processing unit) zpracovává informace podle předem vytvořeného počítačového programu. Obsahuje registry a sadu instrukcí pro práci s nimi. Registry obsahují data ke zpracování a výsledky operací.

REGISTR = paměť o malém rozsahu k uchování a zpracování zpravidla jedné hodnoty

(E) = enhanced – rozšířený (rozuměj v tomto smyslu 32-ti bitový oproti původnímu 16-ti bitovému)

(E)AX – Akumulátor (ukládají se do něj nejčastěji výsledky výpočtů)

(E)BX – Base register (používá se nejčastěji jako ukazatel na data)

(E)CX – Counter register (používá se nejčastěji jako počítadlo)

(E)DX – Data register (používá se nejčastěji jako ukazatel na data)

Existuje ještě čárkovaná (náhradní sada) registrů AX' až DX'

F – Flags register (registr příznaků, drží informace o stavech operací)

BP - Base Pointer(ukazatel na začátek dat)

SI - Source Index (Index Zdroje – místo, odkud se budou číst data)

DI - Destination Index (Ukazatel Cíle – místo, kam se uloží data)

SP - Stack pointer (ukazatel zásobníku)

IP – Instruction pointer (ukazatel na právě prováděnou instrukci v CS)

Segmentové registry drží adresy segmentů paměti, ve kterých jsou uložena data a kód, se kterými se právě pracuje

CS – Code Segment – segment, ve kterém právě běží kód

DS – Data Segment – segment, ve kterém jsou uložena data (lokální proměnné a podobně)

ES – Extra Segment – náhradní datový segment

SS – Stack Segment – segment, ve kterém je umístěn zásobník

Registr příznaků

F (FLAGS)– tento registr uchovává různé systémové binární hodnoty ve formě nastavení svých jednotlivých bitů na 1 nebo 0. Například jeho indikátor **zero flag** je nastaven na jedna, jestliže výsledek poslední aritmetické operace byl **nula**. Indikátor **carry flag** zase signalizuje přenos hodnoty přes hranici bytu. Indikátor **Overflow flag** signalizuje přetečení rozsahu typu atd.

Příznaky se využívají zejména při hardwarovém ladění kódu, pro řízení cyklů v programu a pro detekci chyb. Lze je však ve strojovém kódu testovat uživatelsky a některé z nich i nastavovat, ať už přímo instrukcemi nebo nepřímo výsledky instrukcí.

Závěr

Programovat se ve skutečnosti může naučit téměř každý, kdo je schopen logického abstraktního myšlení. Stačí jen nastudovat patřičné množství informací, které přestože to tak nevypadá, není v případě programování zdaleka tak rozsáhlé jako v jiných oborech. Počítačový program má svá pravidla, ale tvoříte ho vy. Doufám, že tato má publikace pomůže všem začínajícím programátorům jako rychlý úvod do problematiky a že se k ní budou občas vracet jako k přehlednému shrnutí základních pravidel. Tak je také tato příručka míněna a samozřejmě nemůže nahradit všechny ty tlusté knihy, jenž byly o programování v C# napsány, ale jako úvod do problematiky a první seznámení s jazykem C# vám určitě dobře poslouží. Přeji vám mnoho úspěchů ve vaší případné kariéře programátora a velkou fantazii při vytváření nových zajímavých aplikací - každý váš program je přece originál a svým způsobem umělecké dílo, ale také se při tvorbě nenechte příliš unést ideou a pamatujte, že přehledný a profesionálně napsaný zdrojový kód je vaše nejlepší vizitka.

Jaroslav Peňaška

O autorovi

Programováním se zabývám od roku 1992 a věnuji se širokému rozsahu technologií včetně programování databází, Direct3D, O3D a XNA grafických 3D aplikací, tvorby komponent, tvorby aplikací pro mobilní zařízení a PDA, programování monolitických počítačů PIC a řízení periferií počítačem. Moje znalosti vychází z praxe. Jsem autorem několika zajímavých aplikací (používaných celosvětově v řádech tisíců instalovaných kopií) pro 3D editaci objektů, pro převod textu na řeč ve virtuální realitě (s tímto software jsem reprezentoval Českou republiku na virtuálním veletrhu ITE (SL EXPO) 2007) nebo například pro generování VR skriptů.

Pracuji obvykle jako programátor a grafik na volné noze, vedl jsem několikrát menší tým vývojářů a programoval i v prestižních zahraničních společnostech, jako je londýnská Casewise Ltd nebo holandský ISAH. Jsem také autorem série odborných článků, návodů a přednášek o využití virtuální reality ve vzdělávání, internetovém podnikání a pro pomoc tělesně postiženým vést s využitím VR (jako je například Second Life, kde je můj řečový interface jádrem navigačního systému pro zrakově postižené) aktivnější sociální život. Programuji především v RAD jazycích, jako je C#, VB.NET, Java, Delphi Pascal, ale i v C++ nebo Assembleru.