KATEDRA INFORMATIKY
PŘÍRODOVĚDECKÁ FAKULTA
UNIVERZITA PALACKÉHO

SOFTWAROVÁ LABORATOŘ V JAZYCE C#

ALEŠ KEPRT

VÝVOJ TOHOTO UČEBNÍHO TEXTU JE SPOLUFINANCOVÁN
EVROPSKÝM SOCIÁLNÍM FONDEM A STÁTNÍM ROZPOČTEM ČESKÉ REPUBLIKY

Olomouc, 30.4.2008

Abstrakt

Softwarová laboratoř je místo, kde mají studenti informatických oborů možnost vyzkoušet si pod vedením vyučujícího programování pomocí v praxi používaného programovacího jazyka a vývojových nástrojů. Studium v rámci Softwarové laboratoře probíhá převážně formou cvičení, kde i studenti kombinovaného studia mají možnost tvorby programů přímo ve škole pod vedením vyučujícího. V rámci těchto cvičení se studenti naučí používat některý z v praxi běžně používaných programovacích jazyků a také mají možnost si vyzkoušet implementaci úloh, na které narazili při studiu dalších předmětů v rámci svého vysokoškolského studia. Tento studijní text si klade za cíl poskytnout studentům teoretický základ k těmto cvičením a to ve formě materiálu vhodného pro domácí samostudium, aby se studenti na cvičeních ve škole již teoretickými otázkami nemuseli zdržovat a mohli plně využít svůj čas u počítačů s vyučujícím jako rádcem a pomocníkem při praktické práci.

Cílová skupina

Cílovou skupinou jsou studenti informatických oborů na Přírodovědecké fakultě Univerzity Palackého v Olomouci, speciálně pak studenti studující obor Aplikovaná informatika v kombinované formě a studenti učitelství výpočetní techniky pro střední školy. Těmto studentům učební text pomůže zvládnout především obtížné začátky jejich programovacích praktik a pomůže jim tak snáze zvládnout praktickou část studia, která je zvláště pro studenty v kombinované formě studia velmi obtížná. Text může být užitečný také pro studenty příbuzných oborů, jmenovitě bakalářského studia oborů Informatika a Aplikovaná informatika v prezenční formě, kde opět studentům poslouží jako pomůcka při úvodních pokusech programování v jazyce C#.

Obsah

1 Úvodní seznámení 6
1.1 Vývojové prostředí 6
1.1.1 Instalace a první spuštění 6
1.1.2 Řešení a projekty 7
1.1.3 Hello World – konzolová verze 9
1.1.4 Jmenný systém jazyka C# 10
1.1.5 Hello World – okenní verze 11
1.2 Platforma .NET 13
1.3 Jazyk C# 15
1.3.1 Charakteristika jazyka 15
1.3.2 C# za pět minut 16
1.3.3 Příklad – součet dvou čísel 16
1.3.4 Příklad – dynamická fronta 17
1.3.5 Cvičení: Zásobník 17
1.3.6 Řešení chybových stavů 18
1.3.7 Vlastní výjimky 19
1.3.8 Generické typy 20
1.4 Kolekce a obecný typ object 21
2 Základní programové konstrukty 24
2.1 Datové typy a proměnné 24
2.1.1 Proměnné 24
2.1.2 Čísla 24
2.1.3 Znaky a řetězce 25
2.1.4 Další základní typy 26
2.1.5 Referenční typy 26
2.1.6 Převod ze stringu a na string 27
2.1.7 Reálná čísla a výpočty 27
2.2 Základní příkazy 28
2.2.1 Středníky a závorky 28
2.2.2 Podmínky – if, switch–case 28
2.2.3 Opakování – while, do–while, for 28
2.3 Pole 29
2.3.1 Obyčejné jednorozměrné pole 29
2.3.2 Operace s poli 30
2.3.3 Vícerozměrné pole - zubaté 30
2.3.4 Vícerozměrné pole – obyčejné 30

2.4 Operátory 30
2.5 Výčtové typy 31
2.6 Volání metod 31
2.7 Kolekce a příkaz foreach 32
2.7.1 Přehled 32
2.7.2 Dynamické pole 33
2.7.3 Asociativní pole 33
2.7.4 Fronta, zásobník a spojový seznam 33
2.8 Property (vlastnosti) 34
2.9 Textový vstup a výstup 34
2.10 Čtení a zápis souborů 36
3 Objektově orientované prvky 38
3.1 Základní pojmy 38
3.1.1 Třída 40
3.1.2 Dědičnost 41
3.1.3 Anonymita klienta 41
3.2 Objektově orientované prvky v C# 42
3.2.1 Viditelnost 42
3.2.2 Třídy 42
3.2.3 Rozhraní (interface) 43
3.2.4 Dědění tříd a implementace rozhraní 43
3.2.5 Metody a překrývání jmen 44
3.2.6 Modifikátory tříd 45
3.2.7 Proměnné a předávání parametrů 45
4 Znaky a text 48
4.1 Přehled 48
4.2 Třída Char 48
4.3 Třída String 49
4.3.1 Metody 50
4.3.2 Statické metody 51
4.3.3 String jako kolekce. 51
4.3.4 Internace řetězců 51
4.4 Další operace s řetězci 51
4.4.1 Stavění stringů – třída StringBuilder 52
4.4.2 Konverze hodnoty z a na řetězec 52
4.4.3 Parsování stringů – třída Regex 52
5 Disky, soubory a adresáře 54

5.1 Přehled 54
5.2 Výjimka IOException 54
5.3 Třída Path – práce s cestami 55
5.4 Třída DriveInfo – informace o disku 55
5.5 Soubory a adresáře 56
5.5.1 Třída FileSystemInfo 56
5.5.2 Třída DirectoryInfo 57
5.5.3 Třída FileInfo 58
5.5.4 Třídy File a Directory 58
5.6 Proudy, čtenáři a písaři 59
5.6.1 Třída Stream 59
5.6.2 Třída FileStream 60
5.6.3 Třída MemoryStream 60
5.6.4 Třída GZipStream 60
5.7 Čtenáři a písaři souborů 61
5.7.1 Třída BinaryReader – binární čtenář 61
5.7.2 Třída BinaryWriter – binární písař 61
5.7.3 Třída StreamReader – textový čtenář 61
5.7.4 Třída StreamWriter – textový písař 62
5.8 Zabezpečení na bázi ACL 62
5.9 Izolované uložení (isolated storage) 63
Seznam obrázků 65
Seznam tabulek 65
Reference 66

1 Úvodní seznámení

Studijní cíle: Na začátku každého studia je student zavalen velkou spoustou nových pojmů, ve kterých se těžko orientuje. Z praktického hlediska je ale vhodnější ukázat si hned v úvodu „od každého něco“, a teprve potom přejít k podrobnějšímu studiu jednotlivých témat. V této kapitole se tedy letmo seznámíme s vývojovým prostředím Visual Studio, které budeme při práci používat, a pak si stejně stručně představíme i programovací jazyk C# a platformu .NET, která tomuto jazyku tvoří pracovní rámec.

Klíčová slova: Visual Studio, C#, .NET Framework, třída, Main, projekt, řešení (solution)

Potřebný čas: 150 minut

1.1 Vývojové prostředí

Poznámka: Informace uvedené v této sekci jsou platné k datu psaní tohoto textu, tj. únor 2008.

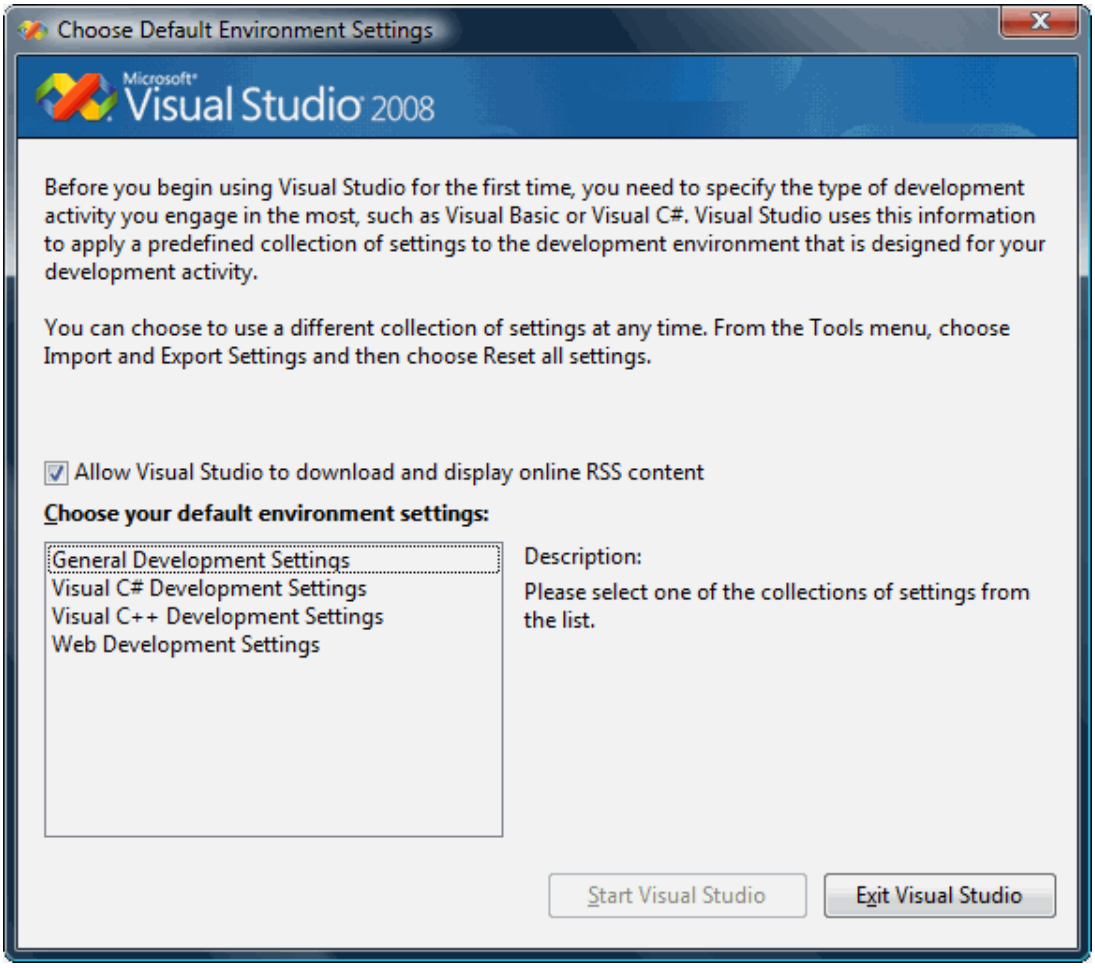
1.1.1 Instalace a první spuštění

Pro práci v Softwarové laboratoři mají studenti k dispozici vývojové prostředí Visual Studio 2008, ve kterém je možno programovat v jazyce C# a také v jazycích C++ a Visual Basic. Toto vývojové prostředí je nejčastěji používaným prostředím vývojáři ve Windows a jeho základní verze pro jednotlivé programovací jazyky je možno získat zdarma na stránce společnosti Microsoft. Konkrétně edici Visual C# 2008 Express, která je vhodná pro práci v rámci softwarové laboratoře, najdete na adrese http://www.microsoft.com/express/vcsharp/.

Průvodce studiem

Katedra informatiky nabízí v rámci programu MSDN Academic Alliance možnost používat vyšší verze Visual Studia studentům a to i na domácích počítačích. Ve škole si (u správce) zjistěte podrobnosti, zda se tato možnost týká i konkrétně vás a kde přesně můžete Visual Studio získat.

Při prvním spuštění Visual Studia se objeví dialog pro přednastavení prostředí pro některý z jazyků (Obrázek 1). Visual Studio umožňuje pracovat vždy se všemi (nainstalovanými) programovacími jazyky, ale jeden z nich je možno nastavit jako primární a prostředí tak více přizpůsobit jemu. Je tedy vhodné zde přímo zvolit Jazyk C#.



Obrázek 1. Dialog přednastavení Visual Studia 2008

Zvolené přednastavení má vliv na výchozí vzhled prostředí, klávesové zkratky apod. Všechny tyto věci je pak možné kdykoliv detailně nastavit dle vlastních potřeb v menu Tools–Options, na tamní kartě Keyboard lze změnit i přednastavení (ztratí se tím však všechny uživatelské změny).

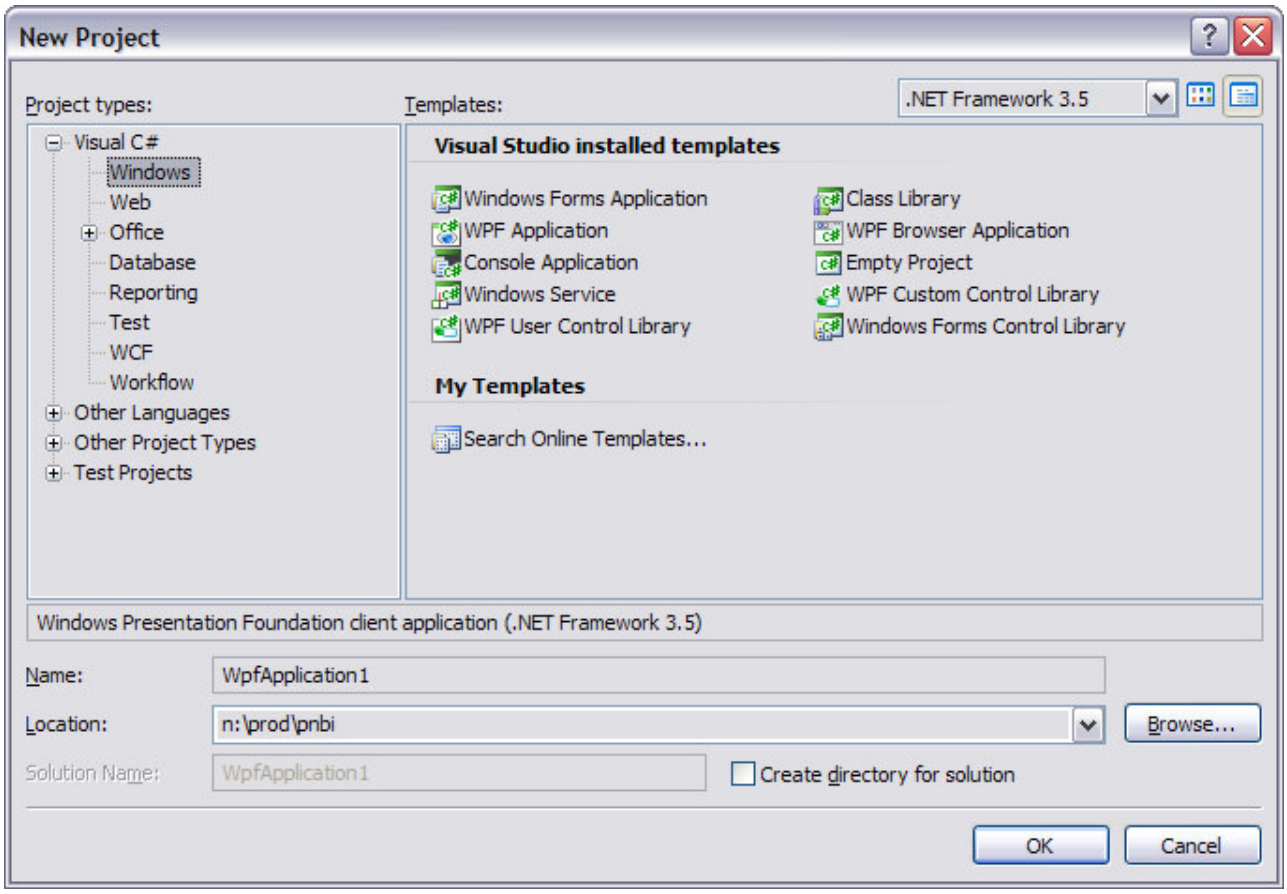
Poznámka: Celé vývojové prostředí je pouze anglicky, v české verzi se nedodává.

1.1.2 Řešení a projekty

Programům, na kterých pracujeme, říkáme projekty. Každý projekt se může skládat z jednoho nebo více vstupních souborů, ve kterých máme kód našeho programu, a výstupem (výsledkem) projektu je obvykle spustitelný „exe“ soubor.

Řešení (solution) je pak skupina projektů, na kterých pracujeme a které dohromady tvoří řešení našeho úkolu. U menších programů obvykle pracujeme pouze s jedním projektem, přesto však musíme mít i založené i nějaké řešení, protože Visual Studio to jednoduše vyžaduje.

Pro založení nového projektu zvolíme v menu položku File–New–Project…, otevře se nám okno výběru typu projektu (Obrázek 2).



Obrázek 2. Založení nového projektu

Visual Studio rozlišuje velké množství různých typů projektů a pro přehlednost nám je zde nabízí rozdělené do různých kategorií. V levé části okna vidíme seznam těchto kategorií. Používáme-li přednastavení pro jazyk C#, tak při otevření dialogu se standardně nabízí v první záložce projekty pro jazyk C# a z nich podkategorie Windows. V pravé části okna vidíme nabídku konkrétních typů projektu – zde tedy 10 různých typů projektů pro Windows.

Průvodce studiem

Není třeba lámat si hlavu z velkého množství různých typů projektů, které Visual Studio nabízí. My totiž budeme používat jen dva typy a oba si hned na úvod vyzkoušíme.

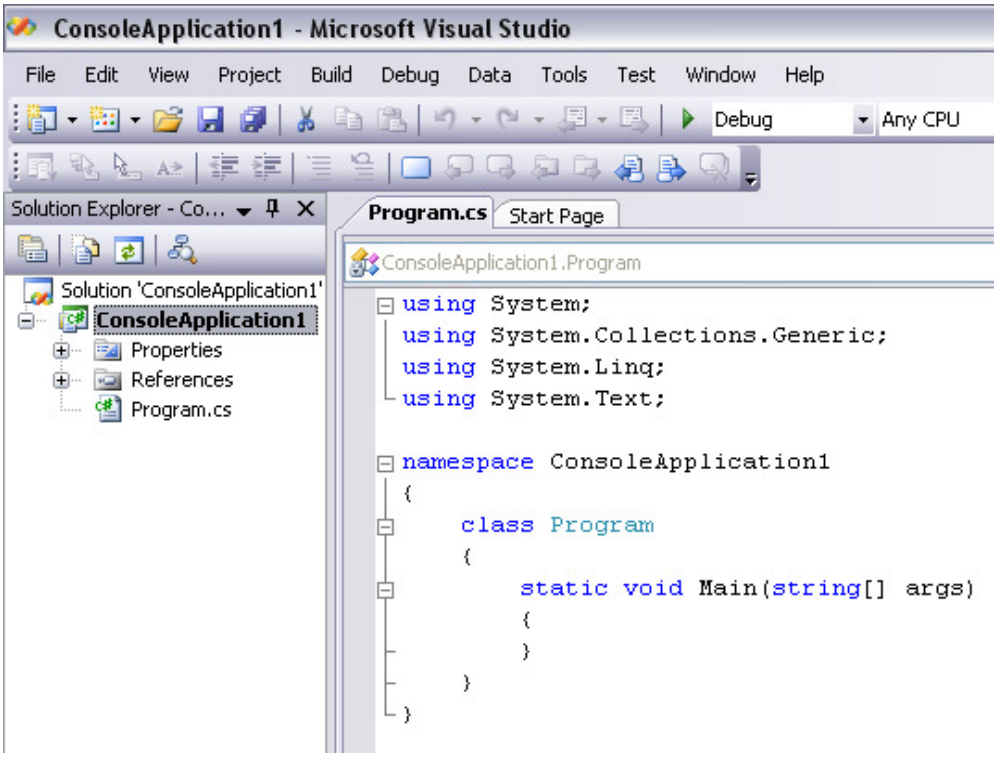
Z nabídky vyberte „Console Application“ a v dolní části okna zadejte název projektu (řádek Name) a místo, kam jej chcete uložit na disk (řádek Location). V místě uvedeném jako Location se vytvoří adresář jménem Name a do něj se budou ukládat všechny soubory vašeho projektu. (Name tedy bude názvem projektu i adresáře, kde je uložen. Location je nadřazený adresář.)

Průvodce studiem

Pozor! Z bezpečnostních důvodů je nutno projekty vytvářet na lokálním disku. Pracujete-li ve škole, můžete projekty ukládat například do své složky Dokumenty nebo do c:\public, nesmíte je však vytvářet na síťovém disku N:, kde máte svá data ze serveru.

Při odhlášení z počítače se složka Dokumenty překopíruje na server při vašem příštím přihlášení ji opět budete mít k dispozici. (Abyste náhodou nepřišli o své práce, raději si při prvním použití Visual Studia ověřte, že tato archivace na serveru opravdu funguje.)

Vytváření projektu může chviličku trvat, po dokončení operace se vám objeví kostra aplikace (Obrázek 3).



Obrázek 3. Kostra konzolové aplikace

1.1.3 Hello World – konzolová verze

Jakmile máme vytvořený projekt a kostru aplikace, můžeme si rovnou vytvořit klasický úvodní program „Hello World“, který vypíše na obrazovku pozdrav.

Na výchozí kostře programu (Obrázek 3) je možno pozorovat obvyklou strukturu souboru se zdrojovým kódem C#. Na obrázku vidíme v levé části ještě „Solution Explorer“, ve kterém se zobrazuje seznam souborů obsažených našem projektu. Tento panel je při práci velmi důležitý, například dvojklikem na kterýkoliv soubor se nám tento otevře v textovém editoru. Ve výchozím nastavení Visual Studia je tento panel obvykle na pravé straně obrazovky, řada programátorů je však z historických důvodů zvyklá mít jej vlevo. Chcete-li jej přesunout doleva, uchopte ho myší a tažením jej můžete snadno přesunout na jiném místo pracovní plochy. Všimněte si, že takto přetáhnout lze i další panely či lišty prostředí Visual Studia.

V projektu může být a obvykle také bývá více zdrojových souborů, každý má příponu .cs označující, že jde o soubor v jazyce C# (z anglického C–sharp). Na začátku souboru vždy uvádíme seznam „using“ direktiv, ty si vysvětlíme později. Zbytek souboru je samotný program a ten je obvykle rozdělený do různých bloků, které se obvykle vnořují do sebe (ovšem jen dle určitých pravidel). Každý blok je ohraničen složenými závorkami; při programování v C# skutečně velmi často používáme složené závorky a jak vidíme na obrázku, i v prázdném

programu, který nic nedělá, máme tři vnořené bloky. Před každým blokem je pak uvedeno, co to je za blok (a tímto se nyní nemusíme trápit).

Jak tedy vytvořit Hello World: Každý program po spuštění začíná vykonávat příkazy ve statické metodě Main. Znáte-li z jiných jazyků pojem funkce (či procedura), tak Main můžete považovat za zvláštní funkci (či proceduru). Pojem „statická metoda“ de facto označuje funkci nacházející se uvnitř třídy (anglicky class – viz Obrázek 3). Dodejme ještě, že funkce to je bez ohledu na to, zda vrací, nebo nevrací nějakou hodnotu. Například Main nikdy nic nevrací. Dovnitř naší metody/funkce Main tedy přidáme příkaz pro vypsání pozdravu na obrazovku:

```
Console.WriteLine("Hello World");
```

Program nyní můžeme spustit klávesou Ctrl+F5 (případně v menu Debug–Start without Debugging).

Průvodce studiem

Visual Studio nabízí dva spouštěcí příkazy: Pro ladění programu se používá F5 (Start Debugging), který umožňuje také krokovat program, tj. vykonávat příkazy pomalu jeden po druhém a sledovat, co se v programu děje. Nevýhodou tohoto typu spuštění je, že po skončení programu se okamžitě zavře jeho okno a nevidíme tak výsledek. Proto pro náš úvodní program používáme Ctrl+F5 (Start without Debugging), který sice neumožňuje program ladit, ale po jeho skončení ještě před zavřením okna čeká na stisk klávesy. Můžeme tak vidět náš pozdrav.

1.1.4 Jmenný systém jazyka C#

Nyní když máme hotový první jednoduchý program, zastavme se u trochu složitější, ale velmi důležité věci. A tou je systém jmen v C#.

Funkce WriteLine (anglicky „Write Line“ = napiš řádek) vypíše daný text na obrazovku a odřádkuje. Tato funkce se nachází ve třídě Console, proto abychom ji našli, vždy se na ni odkazujeme jménem `Console.WriteLine`. Jak budete brzy vidět i v dalších programech, v C# se velmi často takto oddělují části jména tečkou a tento styl skládání jmen je jistě každému známý například z internetových adres zadávaných do webového prohlížeče. Je zde pouze ten rozdíl, že v prohlížeči se konkrétní název zadává nalevo a za něj se přiřazují obecnější (příklad: ulice.město.stát.světadíl), zatímco v C# je konkrétní název napravo a obecnější se připojují zleva (příklad: světadíl.stát.město.ulice).

Ačkoliv se zatím tváříme, že pojem „třída“ nás teď nezajímá, z ukázek to vypadá, že třída je jakési místo, které sdružuje skupinu metod či funkcí. A skutečně je tomu tak: Třídu mimo jiné sdružuje metody příbuzného určení. Například třída Console obsahuje také metodu Write, která vypíše text, ale neodřádkuje, nebo také metodu ReadLine, která naopak přečte text z klávesnice.

Podíváme-li se na náš program, zbývá nám pochopil smysl úvodních řádků using a namespace. Namespace je doslova „prostor jmen“ či „jmenný prostor“ a slouží ke sdružení více tříd dohromady. Takže naše metoda Main se ve skutečnosti celým jménem nazývá ConsoleApplication1.Program.Main. Přitom výchozí namespace má stejné jméno jako projekt, takže místo ConsoleApplication1 se vám zřejmě zobrazuje jiný název.

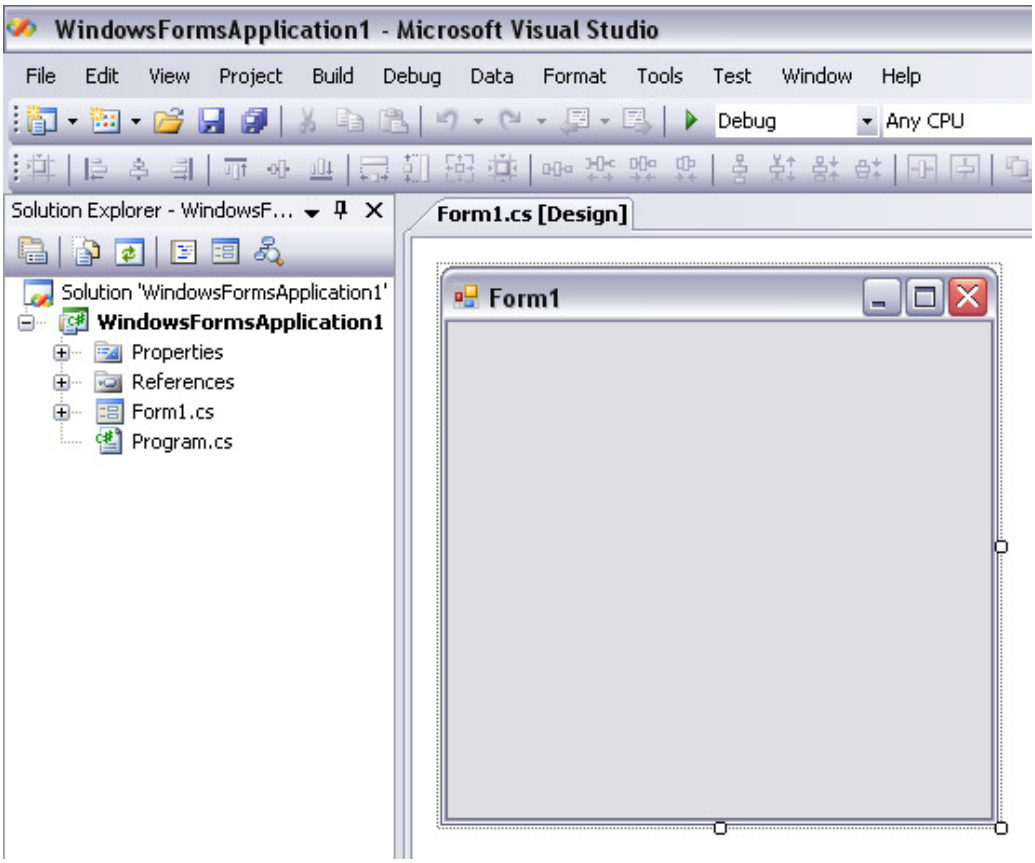
Jak vidíme, skutečná jména metod jsou často velmi dlouhá (a to to v našem případě ještě není nejhorší ☺) a právě proto se v C# používají úvodní řádky using, ty totiž určují, které předpony názvů se mají automaticky používat. Například výpis na obrazovku, který jsme použili, se celým jménem nazývá `System.Console.WriteLine` a první řádek v souboru uvádějící using System; říká, že slovo System se má použít jako předpona při hledání třídy Console.

Nyní tedy už chápeme a umíme se vyznat v systému jmen C#. Závěrem dodejme, že v rámci jedné třídy lze metody volat jen uvedením jejich jména bez jména třídy. A stejně tak v rámci jednoho namespace lze používat jména tříd bez znovuopakování namespace. A pokud se náhodou vyskytuje více stejných jmen na různých úrovních, přednost při hledání má to, co je naše místní, před jménem připojeným přes using.

1.1.5 Hello World – okenní verze

Opusťme nyní teorii a vytvořme si další Hello World program, nyní jako „opravdový“ program s oknem. (Všechny programy jsou opravdové, ale na řadu uživatelů ve Windows nepůsobí programy bez oken jako „opravdové“, takže si jeden takový hned vytvoříme.)

Založíme nový projekt, tentokrát typu Windows Forms Application (ve Visual Studiu 2005 pod názvem Windows Application). Tento typ projektu najdeme v nabídce opět v kategorii Visual C#–Windows. Po vytvoření projektu se nám objeví editor formulářů (anglicky Form designer) – místo, kde vytváříme grafickou podobu okna. V programu můžeme mít libovolný počet takových oken, my si pro začátek samozřejmě vystačíme s tím jedním, které máme již vytvořené jako součást kostry programu (Obrázek 4).

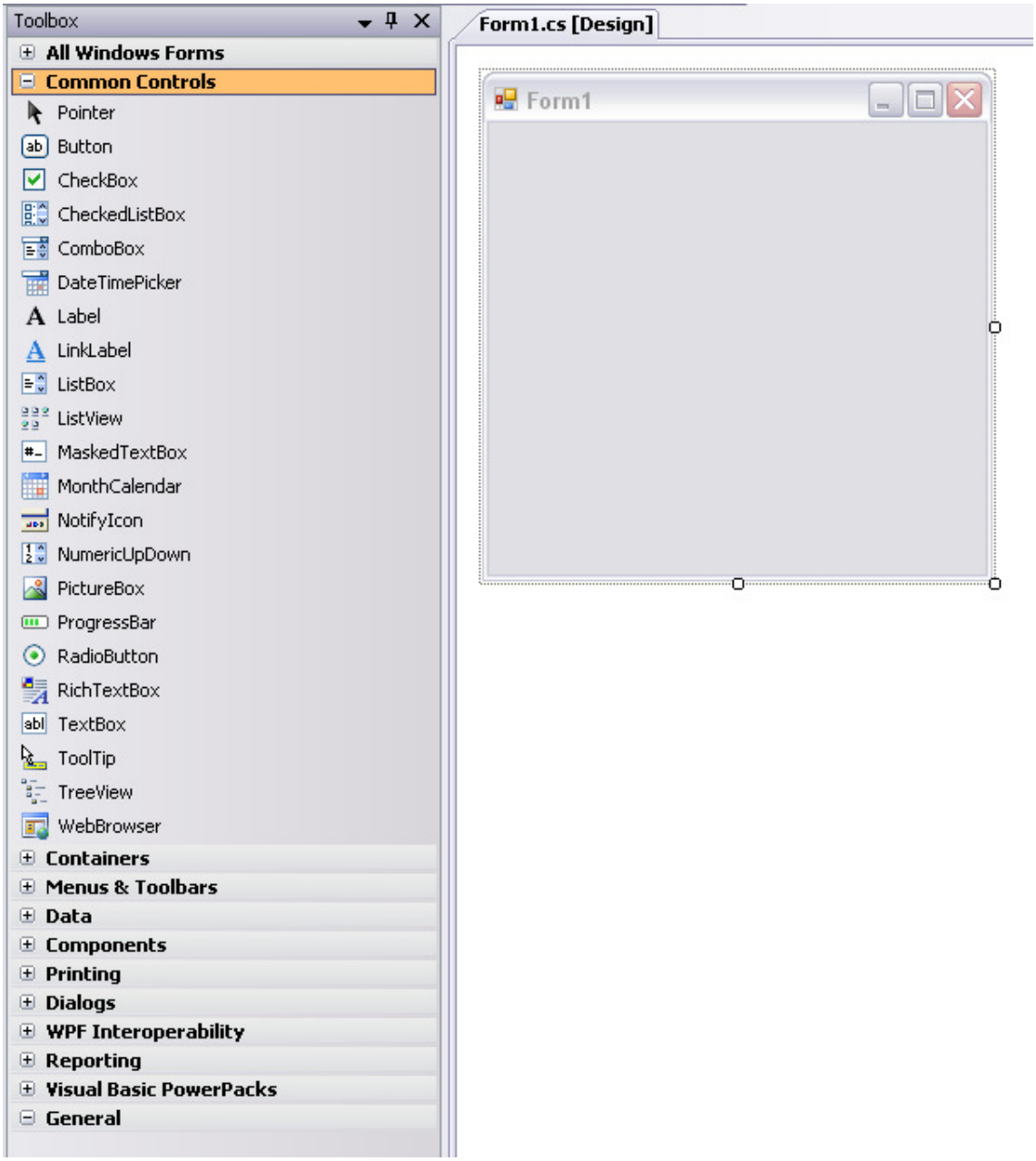


Obrázek 4. Kostra okenní aplikace a otevřený editor formulářů

Vlevo v Solution exploreru vidíme, že v projektu máme dva soubory se zdrojovým kódem. Soubor Form1.cs obsahuje okno, druhý soubor Program.cs pak obsahuje spouštěcí metodu Main. Toto ale zatím nemusíme prozkoumávat, vrhneme se rovnou na vytvoření Hello World programu.

Co budeme dělat: Do prázdného okna přidáme nápis velkým písmem: Hello World. Pod něj přidáme tlačítko s nápisem OK, po jehož zmáčknutí se program zavře.

Abychom mohli přidávat prvky do okna, otevřeme si nástrojovou lištu neboli panel Toolbox (menu View–Toolbox, Obrázek 5). Na tomto panelu najdeme řadu okenních ovládacích prvků, které jistě znáte z jiných programů. My použijeme dva: Prvek Label pro nápis Hello World a prvek Button pro tlačítko OK. Zvolený prvek do okna přidáte tak, že jej myší přetáhnete z toolboxu na plochu okna aplikace. Můžete pak ještě doladit jeho polohu opětovným uchopením a přetažením po ploše okna.



Obrázek 5. Toolbox

Nejprve tedy přetažením myší vložíme do okna prvek Label. V tom okamžiku se ve Visual Studiu automaticky otevře nový panel Properties – je to místo, kde můžeme upravit mnoho vlastností každého ovládacího prvku (i celého okna). My potřebujeme napsat text Hello World, najděte si proto v panelu Properties položku Text a do ní napište požadovaný text.

Průvodce studiem

Program můžete kdykoliv spustit klávesou F5. Nyní je na to vhodný okamžik, abyste se přesvědčili, že nápis Hello World je v pořádku. Program můžete zavřít křížkem v horním pravém rohu okna.

Než se pustíme do tvorby tlačítka, ještě náš nápis zvětšíme. Najděte si v Properties položku Font a klikněte na tlačítko se třemi tečkami. Ve zvláštním okně si pak nastavte velikost či tvar písma.

Nyní přetáhněte na plochu okna prvek Button. V panelu Properties opět najděte položku Text a tentokrát vložte OK. Je to nápis, který se objeví na tlačítku.

Průvodce studiem

To, že Label i Button mají vlastnost Text, není náhodou. Ve Windows všechny ovládací prvky mají vlastnost Text. (I když u některých z nich z logiky věci nemá žádný smysl, je to dobré pro snadné zapamatování.)

Nyní už zbývá jen zadat, že po stisku našeho tlačítka se má program zavřít. Jednoduše dvojklikněte na tlačítko v editoru formulářů a Visual Studio automaticky vloží do programu novou metodu a nastaví ji tak, aby se vykonala při stisku tlačítka. Příkaz pro ukončení programu je `Application.Exit();`, takže jej napište do připravené prázdné metody. (Vysvětlení: `Application` je třída obsahující metody týkající se aplikace/programu jako celku. `Exit()` je statická metoda, která ukončí běžící aplikaci/program. Prázdné kulaté závorky se v C# musejí povinně psát všude tam, kde nechceme předávat žádné parametry při volání. A konečně středník se v C# musí psát na konec každého příkazu (s výjimkou složených závorek, jak si vysvětlíme později).

Náš okenní Hello World je nyní hotový, takže si jej můžete spustit klávesou F5.

Průvodce studiem

Soubor s hotovým programem najdete v adresáři s projektem, přesněji v jeho podadresáři bin\Debug. Soubor se jmenuje stejně jako projekt a má příponu exe. (Umístění a jméno tohoto souboru samozřejmě můžete změnit, toto je výchozí případ.)

1.2 Platforma .NET

Microsoft .NET Framework, krátce obvykle nazýván jako „dotnet“, je běhové prostředí společnosti Microsoft určené pro psaní programů určených k běhu v systému Windows. .NET tvoří jakousi nástavbu nad operační systém Windows a poskytuje aplikacím úplné (kompletní) prostředí, takže ve většině případů jednotlivé aplikace nemusí přímo operační systém používat a vystačí si s tím, co pro ně umí udělat .NET a jeho knihovna funkcí. Je to tedy celá samostatná platforma, z hlediska aplikací se tvářící přímo jako operační systém. Představíme si nyní jen velmi stručně základní vlastnosti a principy .NETu.

Průvodce studiem

Podrobnější povídání o platformě .NET by jistě také bylo zajímavé, ale naším cílem je především věnovat se programování v C# a k tomu žádné hluboké znalosti platformy .NET potřebovat nebudeme, proto absolvujeme jen opravdu stručný úvod.

C# (někdy Microsoftem označován také jako Visual C# pro zdůraznění, že k dispozici je i editor formulářů, který jsme před chvílí zkoušeli) je primární programovací jazyk v .NETu. Tento jazyk vznikl přímo při tvorbě .NETu a je mu doslova šitý na míru. Microsoft jej navíc velmi podporuje a je v současnosti také nejpoužívanějším jazykem v rámci .NETu (mezi širokou veřejností). Visual Studio 2008 přímo nabízí ještě podporu jazyků Visual Basic a C++ (v několika variantách), my však budeme používat právě C#. Tento jazyk je odvozený od C++ a Javy a je těmto jazykům velmi podobný. Během posledních let do něj však Microsoft přidal řadu nových prvků, které již nejsou odvozeny od C++ či Javy (a mimochodem také komplikují začátečníkům jeho učení). Tyto moderní prvky se budeme snažit pro začátek nepoužívat. Visual Studio 2008 nabízí verzi C# 2.0 a 3.0 a podobně několik verzí platformy .NET. Ve výchozím nastavení se projekty vytvářejí pro nejvyšší (nejnovější) verze .NETu a C#, což je pro nás vyhovující a nebudeme to měnit.

Průvodce studiem

Visual Studio .NET (2002) podporovalo jen .NET 1.0 a C# 1.0. Visual Studio .NET 2003 podporovalo jen .NET 1.1 a C# 1.1. Visual Studio 2005 podporovalo jen .NET 2.0 a C# 2.0. Současné Visual Studio 2008 podporuje jednak .NET 2.0 a 3.0 s C# 2.0, a potom také .NET 3.5 s C# 3.0.

Jak už víme, .NET je především prostředí pro běh programů. Samotné běhové prostředí .NETu se nazývá CLR (Common Language Runtime – společné běhové prostředí) a mohou v něm běžet programy napsané v jakémkoliv programovacím jazyce. CLR zajišťuje především bezpečný běh programů, kde bezpečností je rozuměno, že systém dokáže kontrolovat práci s proměnnými a paměti a také oprávnění provádět různé systémové operace. Systém má automatickou správu paměti, což také přispívá k zajištění bezpečnosti při běhu. Další důležitou vlastností CLR je, že programy se nikdy neinterpretují, ale vždy překládají. Jazyk C# je navržen tak, aby byl vhodný pro překlad a běh programu byl rychlý.

Průvodce studiem

Prakticky každý jazyk lze interpretovat i překládat, máme-li pro něj příslušný interpret či překladač. Interpret je jednodušší, překlad však vede k rychlejšímu vykonávání kódu. Programy v C# jsou kvůli automatické správě paměti sice obvykle pomalejší než například C++, ale na druhé straně jsou díky překladu rychlejší než jiné interpretované jazyky jako Java či PHP.

.NET definuje CTS (Common Type System – společný typový systém), je to sada datových typů, které povinně musejí používat všechny programovací jazyky. Díky tomu je možno kusy programů psát v různých jazycích a libovolně je kombinovat. (Ve skutečnosti můžeme kód kombinovat na úrovni tříd – každou třídu lze napsat v jiném jazyce a pak z nich nechat vytvořit

například jeden výsledný exe soubor.) V zájmu bezproblémové spolupráce mezi různými jazyky .NET také definuje CLS (Common Language Specification – společná specifikace jazyků), což je sada pravidel, které je nutno dodržet, aby třídy napsané v různých jazycích spolu skutečně mohly spolupracovat. Příkladem CLS může je pravidlo, že všechny veřejně přístupné součásti třídy se musejí jmenovat tak, aby byly mezi sebou rozlišitelné bez ohledu na malá a velká písmena. Není tedy možné mít veřejnou Funkci a FUNKCI, protože až na velikost písmen se jmenují stejně. Vnitřní součásti tříd, které nejsou navenek vidět, se však v C# takto podobně jmenovat mohou, neboť C# vždy velikost písmen rozlišuje. (Toto je nová důležitá znalost!)

.NET také obsahuje velmi rozsáhlou knihovnu tříd BCL (Base Class Library – základní knihovna tříd). Díky ní si v většině případů vystačíme s .NETem a nepotřebujeme žádné další knihovny či funkce operačního systému. Výhoda této knihovny je, že všechny její součásti mají automatickou správu paměti.

Průvodce studiem

Knihovna BCL je opravdu velmi rozsáhlá a Microsoft stále přidává další součásti. Proto je nemyslitelné, že by se ji někdo někdy naučil celou. Každý programátor zná jen základní součásti BCL a pak ty věci, které je zvyklý používat. Ostatní si až v okamžiku potřeby najde někde v literatuře či [MSDN].

1.3 Jazyk C#

1.3.1 Charakteristika jazyka

Jazyk C# je převážně statický objektově orientovaný jazyk a imperativní jazyk. Statický (či přesněji: staticky typovaný) znamená, že při zakládání proměnných (obvykle) musíme uvést datový typ a tento typ pak proměnná má po celou dobu své existence. Objektově orientovaný znamená, že programy jako celky se skládají ze tříd a objektů, které můžeme tvořit i používat. Imperativní znamená, že jednotlivé dílčí příkazy se vykonávají postupně jeden po druhém tak, jak jsou napsány za sebou.

Průvodce studiem

Pojmy objektově–orientovaný a imperativní jdou principiálně proti sobě, ale nevylučují se, protože imperativním způsobem skládáme dílčí jednotlivé příkazy a z bloků takových příkazů pak skládáme metody a třídy a používáme je objektově orientovaným způsobem. Při studiu C# se tedy budeme učit tyto dva přístupy k programu jako dílčím příkazům a programu jako celku, které jsou velmi rozdílné a pouze při zvládnutí jich obou se naučíme vytvářet kvalitní programy.

Základní syntaxe (tj. jak se písmenka skládají za sebe, kam psát závorky či středníky apod.) se velmi podobá C++, sémantika (tj. co program dělá) se pro změnu velmi podobá Javě. C# lze provozovat jen s automatickou správou paměti. C# má i řadu prvků, které v C++ či/ani Javě nenajdeme, jsou to vesměs funkcionální rozšíření (jako uzávěry, funkce vyššího řádu či lambda výrazy).

1.3.2 C# za pět minut

Pro programátory zvyklé na C++, Javu či podobné jazyky je zvládnutí základů C# otázka dvou minut. Ani pro programátory zvyklé na PHP či Delphi to není o mnoho těžší.

- Základním rozdílem oproti většině starších jazyků je automatická správa paměti.
- Nové objekty zakládáme pomocí new.
  ```
  Typ x = new Typ();
  ```
- O rušení objektů se nestaráme.
- Všechny objektové proměnné se chovají jako pointery v C++, v proměnné tedy máme vždycky jen odkaz na objekt, ne přímo objekt.
- Výjimkou v tomto jsou základní datové typy `char`, `int`, `bool` apod., u kterých je v proměnné přímo uložena hodnota, a ne jen odkaz na ni.
- Nikam nepíšeme hvězdičky *, ani ampersandy &.
- Místo operátorů šipka -> a dvě dvojtečky :: se používá obyčejná tečka.
  ```
  odkaz.metoda();
  třída.metoda();
  ```
- Znaky (`char`) jsou dvoubajtové, čili stejné jako `short`. Typ pro jeden bajt se jmenuje byte.
- Nelze dělat globální proměnné. (Ale můžeme mít statické proměnné ve třídách. Potom tedy proměnná je umístěná ve nějaké třídě, ale je globálně přístupná.)

1.3.3 Příklad – součet dvou čísel

První dva programy už máme sice za sebou, podívejme se ale na další ukázku:

```
int Součet(int a, int b) {
  return a + b;
}
```

Toto je funkce pro sečtení dvou čísel. Příkaz `return` tedy vrací hodnotu, před název funkce navíc místo slova `void` musíme uvést typ, který bude funkce vracet. (Srovnejte s funkcí Main, kde nic nevrací.)

Průvodce studiem

Levou složenou závorku v tomto textu většinou uvádíme na konci předchozího řádku. Je to čistě pro úsporu místa, odřádkování má totiž v C# stejný význam jako jednoduchá mezera a nemá tedy vliv na funkčnost programu (toto platí skutečně ve všech částech zdrojového kódu). Visual Studio standardně dává levou složenou závorku na samostatný řádek, je však možno si zvolit i jiný vlastní styl formátování (v menu Tools–Options–Text Editor–C#–Formatting–New lines).

Tuto funkci musíme umístit do nějaké třídy. Jinam než do třídy se v C# metody dát nedají. Před definici funkce také musíme přidat slovo `static`, jinak nepůjde volat přímo na třídě bez objektů. (Toto je stejné jako u funkce Main. Má-li to být funkce použitelná třeba v mainu, musíme každou takovou také označit slovem `static`.)

Naši funkci můžeme nyní vyzkoušet zavolat z mainu a rovnou vypíšeme výsledek:

```
Console.WriteLine(Součet(7, 15));
```

Tento program vypíše číslo 22. Pro začátečníky může být srozumitelnější rozepsat tento příkaz na dva řádky. Výsledek bude úplně stejný.

```
int výsledek = Součet(7, 15);
Console.WriteLine("Výsledek je " + výsledek);
```

1.3.4 Příklad – dynamická fronta

Nyní si uvedeme složitější příklad, uvidíme na něm zejména rozdíl oproti C++. Jak jsme si uvedli ve stručném přehledu výše, každá objektová proměnná v C# obsahuje jen odkaz na objekt, a ne přímo objekt. Představme si nyní klasickou dynamickou datovou strukturu frontu jako řetěz prvků, kde každý má nějakou hodnotu a odkaz na sousední prvek. Takový prvek se v C# vytvoří jako nová samostatná třída takto:

```
class Prvek {
  public int hodnota;
  public Prvek soused;
}
```

Co jsme tedy vytvořili: Nový je pro nás samotný fakt, že máme celou novou třídu. V této třídě ale nemáme žádné metody či funkce, ale jen dvě proměnné. První je číselná hodnota prvku (typ int je 32bitové celé číslo), druhá je odkaz na souseda.

Průvodce studiem

V jazyce C++ by taková deklarace byla nesmysl, protože bychom vytvořili prvek, který obsahuje další prvek a vznikl by nekonečný řetěz vnoření. V C# to ale je jen odkaz, ne přímo prvek, takže je vše v pořádku.

Nově jsme také přidali označení `public` ke každé proměnné. Součásti tříd, ať už jde o metody, proměnné či něco jiného, musejí být vždy označeny `public`, aby byly vidět z jiných tříd. Výjimkou je náš známý Main, který si jako startovní bod programu systém najde, i když `public` nebude. (Tím se nám to celé trochu komplikuje. Ale pamatujme si: Vše musíme označit `public` (veřejné), aby to bylo z ostatních tříd vidět. A naopak, co vidět nepotřebujeme, to `public` neoznačujeme.)

1.3.5 Cvičení: Zásobník

V předchozím příkladě jsme vytvořili třídu Prvek pro dynamickou frontu. Stejnou třídu je možno použít i pro implementaci zásobníku, a protože zásobník je nejjednodušší dynamická datová struktura, zkusíte si v rámci cvičení naprogramovat sami.

Vytvoříte tedy třídu `Zásobník` a v ní tři metody:

- `void Push(int)` – přidá na vrchol zásobníku nový prvek s danou hodnotou
- `int Pop()` – odebere prvek z vrcholu zásobníku a vrací jeho hodnotu
- `int Top()` – vrací hodnotu prvku z vrcholu zásobníku, ale nechá jej tam

Správnou funkčnost vaší třídy si vyzkoušejte například vložením čísel 1–5 a následným vypsáním obsahu zásobníku (mělo by být pozpátku, tj. 5–1):

```
Zásobník s = new Zásobník();
for(int i=1; i<=5; i++) s.Push(i);
for(int i=1; i<=5; i++) Console.Write(s.Pop() + " ");
Console.WriteLine();
```

Na tomto ukázkovém testovacím programu je také vidět základní smysl tříd: Na prvním řádku založíme novou proměnnou, kde typ této proměnné je náš zásobník. Třída je tedy datovým typem. Dále pak používáme metody naší třídy odkazem přes jméno této proměnné. Takto můžeme v programu současně založit libovolné množství proměnných typu Zásobník a nezávisle na sobě je používat. Speciální statické střídy a statické metody, se kterými jsme se také setkali (metoda Main a celá třída Console) mají tu zvláštnost, že se od nich proměnné nevytvářejí a existují tedy jen v jediném exempláři v programu. (Nyní jsme poznali z nejdůležitějších principů objektově orientovaného programování – program a data definujeme jen jednou v každé třídě, ale pak od této třídy můžeme vytvářet libovolně mnoho objektů a proměnných.)

Průvodce studiem

Zásobník (anglicky stack) patří mezi základní dynamické datové struktury. Operace push, pop a top jsou standardní operace nad zásobníkem. Trošku matoucí je kombinace českých a anglických termínů, dále v textu se proto raději budeme držet anglických názvů. C# sice umožňuje i české názvy a to včetně diakritiky, ale v angličtině budeme mít názvy jednotné.

Princip zásobníku si můžete představit takto: Je to jako komínek na sebe postavených kostek, na který můžeme přidávat nové kostky (hodnoty) a zvyšovat jej tak, nebo naopak z jeho vrcholu odebírat kostky (hodnoty). Vždy můžeme přidat či odebrat jen kostku na vrcholu. Výsledným efektem této struktury je, že to, co poslední přidáme, potom jako první odebereme, proto jsou čísla vypsaná testovacím programem pozpátku. Zásobník je díky své jednoduchosti a této vlastnosti obracet pořadí velmi užitečnou datovou strukturou.

1.3.6 Řešení chybových stavů

Další téma, kterému se budeme věnovat, jsou chybové stavy. Váš zásobník zatím chybové stavy neřeší (jednoduše proto, že to neumíte). Ošetřování chybových stavů je však velmi důležitá věc a tak se ji naučíme hned teď.

Ve starších programovacích jazycích se chyby řešily obvykle tak, že funkce vracely tzv. chybové kódy. Podle návratové hodnoty funkce se tedy poznalo, zda proběhla bez chyby, či s chybou. Nevýhodou takového řešení bylo, že po každém volání funkce jsme museli přidat příkaz if (či nějaký jiný příkaz pro testování hodnoty), abychom chybu ihned zjistili a nějak mohli ošetřit. Další nevýhodu takového řešení vidíme například na funkci pop – ta má vrátit hodnotu ze zásobníku, nemůže tedy vracet chybový kód. Mohli bychom se domluvit, že třeba číslo –1 bude označovat chybu, pak ale právě toto číslo nelze do zásobníku nikdy uložit, protože bychom nedokázali rozlišit, zda funkce pop vrátila řádně číslo –1 ze zásobníku, nebo –1 jako chybový kód. V C# (a dalších moderních jazycích) naštěstí máme pro řešení chyb jiný nástroj.

V C# se chyby nikdy neřeší přes návratové hodnoty. Program píšeme tak, aby fungoval v bezchybném případě a chyby řešíme zvláštním konstruktem zvaným výjimky. Výjimky existují i v některých starších jazycích, ale například v C++ jsou omezené a nedají se v praxi rozumně (k užitku) používat.

Systém výjimek si stručně představíme přímo na příkladu zásobníku. Nejprve si rozmysleme, jaké chyby tam vlastně mohou nastat: V metodě push žádná, v metodách pop a top shodně při prázdném zásobníku není co vrátit. Takže při volání pop nebo top při prázdném zásobníku oznámíme chybu tímto příkazem:

```
if(vrchol == null) throw new Exception("Zásobník je prázdný");
```

V tomto řádku se nám sešlo hned několik nových věcí: Příkaz if (doslova „pokud“) slouží k otestování nějaké podmínky a vykonání následného příkazu jen při platnosti této podmínky. My testujeme podmínku `vrchol == null`, kterou testujeme, zda je vrchol zásobníku prázdná/neexistující hodnota. Používáme k tomu dvě rovnítka za sebou, což je nutné. Jedním rovnítkem přiřazujeme hodnotu, dvěma rovnítky testujeme rovnost. Slovo null pak označuje speciální hodnotu, kterou můžeme přiřadit či testovat u objektů – znamená „žádná hodnota“.

Zbytek řádku (za první kulatou závorkou) je již příkaz, který se vykoná jen při prázdném zásobníku. Je to příkaz `throw`, který takzvaně „vyhodí“ výjimku. Výjimka je objekt popisující chybu, která vznikla. Nejjednodušší způsob jak takový objekt vytvořit je pomocí `new Exception` vytvořit nový objekt a jako parametr zadat text popisující chybu. Systém do tohoto objektu automaticky přidá informaci o tom, kde přesně tato chyba nastala.

První krok tedy máme hotový, umíme vyhodit výjimku. Druhým krokem bude takovou výjimku zachytit. K tomu slouží konstrukce `try–catch`, která se používá takto: Část programu, kde chceme sledovat výjimky, uzavřeme do bloku `try` a přidáme blok `catch`, který se vykoná jen při výskytu výjimky. Tuto konstrukci můžeme dát kamkoliv, kde chceme výjimky odchytávat a nějak zpracovávat. My toto přidáme do mainu, protože tam náš zásobník používáme.

```
try {
  …původní program…
}
catch(Exception e) {
  …ošetříme výjimku…
}
```

Význam tohoto kódu je intuitivní. V definici bloku `catch` vždy uvedeme, jaký typ výjimky chceme zachytávat (zde máme uvedeno `Exception` jako „jakoukoliv výjimku“) a jméno proměnné (zde máme `e`).

Máme-li odchytávání výjimek hotové, můžeme třeba pro zkoušku změnit kód v mainu tak, aby se snažil ze zásobníku odebrat více hodnot, než tam předtím vložil. Program tak jistě skončí s výjimkou a vykonávání kódu se dostane do našeho bloku `catch`. Tam je vhodné umístit WriteLine s nějakým výpisem informace o chybě uživateli programu.

1.3.7 Vlastní výjimky

Náš program se díky odchytávání výjimek dokáže vypořádat s chybovými stavy. Co když ale někde v programu (kdekoliv, na pro nás nečekaném či neznámém místě) vznikne jiná chyba? Systém se zachová tak, že vyhodí výjimku a my ji opět zachytíme v našem `catch` bloku. Problém této situace je, že si myslíme, že jsme za zachytili naši chybu prázdného zásobníku, ale ve skutečnosti jde o úplně jinou chybu. Co s tím?

Abychom mohli jednoznačně identifikovat, které chyby jsou „ty naše“ a nezachytávali v `catch` bloku všechny chyby, odvodíme si vlastní typ výjimky. Proces odvození je velmi snadný: Vytvoříme novou třídu odvozením od třídy `Exception`. Tím dáme na jeho, že naše třída je taky výjimka, ale nějaká jiná, než přímo původní třída `Exception`.

```
class VýjimkaZásobníku : Exception {
  VýjimkaZásobníku(string zpráva) : base(zpráva) { }
}
```

Jednoduchým použitím dvojtečky v záhlaví definice třídy jsme specifikovali, že naše třída VýjimkaZásobníku je odvozená od třídy `Exception`. Ve třídě jsme nemuseli nic programovat, protože naše třída zdědila všechny součásti od třídy, ze které jsme ji odvodili. Napsat jsme však museli jeden řádek – zvláštní metodu pojmenovanou stejně jako třídu, která nic nevrací. Takové zvláštní metodě se říká konstruktor a je to metoda volaná při použití `new` (tedy při vytváření nového objektu). My jsme zde doslova napsali, že při vytváření nového objektu typu `StackException` je třeba zadat textovou zprávu, která se předá jako parametr vytváření objektu třídy `Exception`. Tato jednoduchá definice (konstruktor je pro začátečníky ne příliš

srozumitelný, ale definice třídy na jednom řádku je vskutku stručná) je funkční a nic dalšího nemusíme přidávat.

Průvodce studiem

Možnost odvodit třídu od jiné je jedním ze základních nástrojů objektově orientovaného programování. Na příkladu výjimky jsme si jej nenásilně představili. Místo odvození se často hovoří o dědičnosti, říkáme, že naše nová třída dědí třídu `Exception`*, nebo, že naše třída je potomkem třídy* `Exception`*.*

Na tomto příkladě jsme také poprvé potkali konstruktor – další důležitý prvek objektově orientovaného programování. Zatím nám sice spíše komplikoval pochopení kódu, než aby v něčem pomohl, ale vrátíme se k němu ještě později a naučíme se jej efektivně používat.

1.3.8 Generické typy

V předchozích odstavcích jsme nechodili jen tak okolo, ale bez okolků jsme se vrhli do programování reálného kódu. Ani v tato sekci nebude výjimkou, podíváme se na generické typy – další zajímavou součást jazyka C#.

Průvodce studiem

Pochopení této sekce vyžaduje jistou dávku abstraktního myšlení, nebo předchozí zkušenosti se stejnými věcmi z jiného jazyka. Budete-li mít problém látku pochopit, zkuste ji přeskočit a vrátit se k ní po prostudování další kapitoly.

Náš zásobník je již robustní (čili správně funguje i při chybových stavech), ale má tu nevýhodu, že do něj lze ukládat jen celá čísla typu `int`. V kapitole 1.3.1 jsme si jazyk C# charakterizovali jako převážně statický jazyk, tj. každá proměnná má nějaký předem daný typ, který později nelze měnit. A to je samozřejmě věc, která nám komplikuje situaci.

Obecně každá datová struktura sloužící k ukládání většího množství jiných objektů či hodnot (pole, fronta, zásobník, strom, atd.) může být užitečná pouze tehdy, když nám umožní vkládat různé typy objektů. Takovým strukturám říkáme kolekce a obvykle v celé kolekci máme prvky stejného typu, ale nemusí to být zrovna čísla `int`. C# toto naštěstí umožňuje velmi efektivně a přitom pro programátora jednoduše řešit pomocí tzv. generických typů. Náš zásobník změníme na generický (či jinak řečeno parametrizovatelný či parametrický) typ takto:

- Do definice třídy přidáme za název jméno parametru do lomených závorek. Je zvykem tento parametr pojmenovávat T, takže napíšeme např. `class Zásobník<T>`.
- Uvnitř třídy změníme všechny výskyty typu `int` v souvislosti s hodnotami prvků na `T`.
- Při zakládání objektů (např. v Mainu) vždy budeme uvádět, jaký typ dosadíme za `T`. Dělá se to opět pomocí lomených závorek,
  např. `Zásobník<int> s = new Zásobník<int>()`.

Nyní si program upravte, aby zásobník byl generický. Potom do Mainu přidejte další testovací kód, kde zásobník bude typu `<string>`. To je typ pro textové řetězce, vložte tedy do zásobníku několik řetězců (zadávají se stejně jako v našem Hello World programu, čili v uvozovkách) a pak je opět zkuste vypsat a měly by být pozpátku.

Na závěr ještě upřesnění, jak je to s těmi typy: **`Zásobník<T>`** je generický typ (generická třída). Není to klasická třída a nelze z ní tedy vytvářet objekty. Teprve dosazením nějakého skutečného typu za **`T`** vznikne skutečný typ (třída), takže **`Zásobník<int>`** a **`Zásobník<string>`** jsou dvě různé třídy, které náhodou dělají totéž, ale s jinými typy prvků (dat).

Průvodce studiem

Jako vedlejší efekt jsme se zde naučili používat nový datový typ: string. Je to jeden ze základních datových typů, budeme s ním pracovat i nadále. String se od ostatních základních typů C# v mnoha věcech liší, ale nyní nás to nemusí trápit, protože jeho použití je naprosto intuitivní. Později se ke stringu ještě vrátíme a probereme si jej podrobněji.

1.4 Kolekce a obecný typ object

Jak bylo řečeno hned v úvodu, .NET nabízí rozsáhlou knihovnu BCL, asi tedy nepřekvapí, že v ní najdeme i již připravené třídy pro frontu či zásobník, se kterými jsme v předchozím textu operovali. Jak už víme, tyto dvě a řada dalších tříd sloužících pro ukládání jiných objektů či hodnot nazýváme kolekce. Konkrétně generický zásobník je ve třídě `System.Collections.Generic.Stack<T>`. Kromě toho však máme k dispozici i obecný zásobník ve třídě **`System.Collections.Stack`** a ten má tu zajímavou vlastnost, že do něj můžeme vkládat prvky i tak, že každý z nich má jiný typ. (Do generického zásobníku také lze ukládat cokoliv, ale do jednoho konkrétního vždy jen prvky stejného typu.) Jak je to možné?

U výjimek jsme si ukázali princip odvození nového typu. Přitom.NETu je každý typ od něčeho odvozený a na začátku celé hierarchie stojí třída jménem **`object`**. (Ano, jmenuje se object, ale není to objekt nýbrž třída.) Pokud při definici nové třídy neuvedeme, od čeho ji chceme odvodit, automaticky je odvozena přímo od typu **`object`**. Přidáme-li k tomu jedno další základní pravidlo objektově orientovaného programování: „Potomek může zastoupit předka.“, máme nástroj, jak vytvořit obecný zásobník. Je to vlastně zásobník typu **`<object>`**. Lze do něj pak vložit jakoukoliv hodnotu, protože cokoliv může zastoupit svého předka a každý typ je přímo či nepřímo odvozen od typu **`object`**.

Vyzkoušejte si upravit Main s vaším zásobníkem tak, aby vytvořil zásobník typu **`<object>`** a potom do něj vložil několik čísel a stringů. Pak obsah zásobníku vypište.

Všechny zabudované kolekce mají ještě jednu důležitou vlastnost: Je možno je procházet, čili prozkoumat jejich obsah bez ohledu na to, jaká je jejich vnitřní struktura. Tuto operaci provádíme obvykle příkazem **`foreach`** (doslova „pro každý“), který umožňuje provést nějaký příkaz pro každý prvek kolekce. Náš vlastní zásobník toto neumožňuje, takže nejprve si změňte main tak, aby používal zásobník z BCL a potom si foreach vyzkoušejte. Vložte do zásobníku několik hodnot a pak je vypište bez toho, abyste je ze zásobníku odebrali. Příkaz foreach se píše takto:

`foreach(`typ hodnota **`in`** kolekce**`)`** příkaz

Za typ zde dosadíte typ prvků v kolekci. Místo jednoho příkazu můžete napsat i více, v C# platí pravidlo obdobné jiným jazykům: Všude, kde lze napsat jeden příkaz, lze také vložit blok příkazů uzavřený do složených závorek. To znamená, že vlastně kdekoliv v kódu můžete zahájit další vnořený blok pomocí složených závorek (ačkoliv to takto „kdekoliv“ má pramalý význam).

Průvodce studiem

Podpora příkazu foreach je dobrým důvodem pro to, abychom dávali přednost používání zabudovaných kolekcí před vlastními. Zabudované kolekce jsou však především optimalizované, napsali je špičkoví programátoři. Vlastní zásobník jsme zde vytvářeli jen proto, abychom si vyzkoušeli různé konstrukce C# v praxi. Mohli bychom nyní také přidat podporu procházení do našeho zásobníku, na první kapitolu učebnice pro začátečníky je to však téma příliš složité.

Shrnutí

Náplní této úvodní kapitoly bylo seznámení s vývojovým prostředím, platformou a jazykem. Naučili jsme se spustit Visual Studio a dělat v něm základní operace. Stručně jsme si představili .NET Framework a vlastnosti jazyka C#, ve kterém budeme dále programovat.

Tento kurz je určen pro studenty, kteří již programovat umějí, i když v jiném jazyku či jiných jazycích, takže jsme neotáleli a ihned se vrhli do tvorby většího programu, než jen ukázkového Hello Wordu. Naše dílo má sice stále jen několik málo desítek řádků kódu, ale poznali jsme díky němu hne několik velmi důležitých pojmů a konstruktů, které programátoři při práci v C# používají. V dalších kapitolách budeme ve studiu pokračovat a i na zde vyzkoušená témata se podíváme znovu a více systematicky.

Pojmy k zapamatování

- Vývojové prostředí
- Řešení (solution) a projekt
- Konzolová a okenní aplikace
- Třída (class)
- Jmenný prostor (namespace)
- Editor formulářů (form designer)
- Nástrojová lišta (toolbox)
- Platforma .NET
- Společné běhové prostředí (CLR – Common Langure Runtime)
- Společný typový systém (CTS – Common Type System)
- Společná specifikace jazyků (CLS – Common Langure Specification)
- Základní knihovna tříd (BCL – Base Class Library)
- Statické typování
- Automatická správa paměti
- Výjimka (exception)
- Blok try – catch – finally.
- Generický typ
- Obecný typ object
- Příkaz foreach

Kontrolní otázky

1. *Vysvětlete, co je to „řešení“ a „projekt“ ve Visual Studiu.*
2. *Čím se vyznačují konzolové aplikace? Jak se liší on „nekonzolových“?*
3. *Co je to CLR (Common Langure Runtime)?*
4. *K čemu je v .NETu společný typový systém CTS?*
5. *Vysvětlete, co je to zásobník.*

6. *Jak se programech v .NETu řeší chybové stavy? Jaké má toto řešení výhody či nevýhody oproti jiným běžně používaným?*
7. *Co to jsou generické typy, jaký mají přínos? Uveďte nějaký další příklad jejich použití (kromě toho, na kterém jsme si je představovali).*
8. *Vysvětlete význam třídy object.*
9. *Jakým jednoduchým způsobem lze procházet jednotlivé prvky kolekce?*

2 Základní programové konstrukty

Studijní cíle: V této kapitole si stručně projdeme základní prvky jazyka C#. Programátoři zvyklí na některý příbuzný jazyk budou po nastudování této kapitoly moci sami programovat. Úplní začátečníci budou mít po absolvování a pochopení této kapitoly znalosti potřebné k tvorbě jednoduchých programů v textovém (konzolovém) režimu. Kapitola je přitom koncipována tak, aby poskytla informace z širokého spektra oblastí a vždy jen s minimální hloubkou. Ty prvky jazyka, které jsou složité, ale v praxi důležité, se naučíme jen používat. Jednoduché a často používané prvky se naučíme i sami tvořit.

Klíčová slova: C#, datový typ, řetězec, příkaz, pole, operátor, metoda, výčtový typ, kolekce

Potřebný čas: 140 minut

2.1 Datové typy a proměnné

Průvodce studiem

Čísla fungují velmi podobně jako v C++. U znaků, stringů a objektů už je to jiné.

Všechny jazyky v .NETu rozlišují dvě velmi rozdílné skupiny datových typů.

- Hodnotové typy – proměnné těchto typů nesou přímo hodnotu.
  Příklad: Všechny číselné typy, znaky, datum, bool.
- Referenční typy – proměnné těchto typů nesou odkaz čili referenci na hodnotu. Samotná hodnota (objekt) je uložena samostatně v paměti. Více proměnných může odkazovat na stejný objekt.

2.1.1 Proměnné

Proměnnou založíme, jak už umíme, uvedením typu, pak za mezeru jméno a středník.

```
int a;
```

Tomuto říkám definice. V definici můžeme také proměnné přiřadit výchozí hodnotu.

```
int a = 7;
```

Proměnné můžeme umístit jedině do tříd, pak jsou společné pro všechny součásti třídy, nebo přímo do kódu, pak jsou tzv. lokální v rámci metody. Lokální proměnné můžeme zakládat i jinde než na začátku metody, platí vždy od místa založení do konce bloku (nejbližší pravé složené závorky).

2.1.2 Čísla

Všechny číselné typy jsou hodnotové. Obvykle vystačíme s několika základními, viz následující tabulka:

typ	popis
int	Běžný typ pro celá čísla (32 bitů)
uint	Neznaménkový int
short	Menší celá čísla (16 bitů)
ushort	Neznaménkový short
long	Velká celá čísla (64 bitů)
ulong	Neznaménkový long
double	Běžný typ pro čísla s řádovou čárkou (64 bitů)
char	Znak (16 bitů)
byte	Bajt bez znaménka (8 bitů)
sbyte	Bajt se znaménkem

Tabulka 1. Základní číselné typy.

U čísel se uplatňují konverze takto: Z menšího typu na větší se číslo převádí (konvertuje) automaticky (říkáme implicitně), můžeme také sami vynutit převod (pak říkáme: explicitní konverze). Explicitní konverzi provedeme uvedením požadovaného typu do závorky před hodnotu.

```
double b = (double)37;
```

Uvedený příklad převátí celé číslo `37` na `double`. Číselné literály (číslo přímo uvedené v kódu programu) jsou obvykle typu int, desetinná čísla pak typu double. Toto má mnohem složitější pravidla, pro zjednodušení si pamatujme: Každé číslo má typ, když chceme do double přiřadit konstantu, tak musíme napsat např. `37.0` a tím už je to desetinné číslo. `37` je ale `int` a jen tak do `double` přiřadit nelze.

```
double b = 37.0;
```

Průvodce studiem

Jména většiny základních typů můžeme psát buď malými písmeny, jak tak to děláme obvykle, nebo také s prvním či více velkými, protože tak to definuje .NET. Některé mají i jiné podobné názvy, např. int je stejné jako `System.Int32`*. Důvodem je, že všechny tyto typy jsou součástí CLS a jmenují se tam* `System.Něco`*, zatímco v C# je lze psát také malým písmenem (a bez předpony System.). Jak vidíme na příkladu* `int`/`Int32`*, dva názvy navíc nemusejí být úplně stejné.*

2.1.3 Znaky a řetězce

Znakový typ je char. Je to současně i 2bajtové neznaménkové číslo. Všechny znaky jsou v kódování unicode (UTF-16). Znakové literály se zapisují v apostrofech.

```
char c = 'a';
```

Jelikož je to číselný typ, můžeme napsat například také 'a'–'A' (výsledkem je 32, číslo typu int). Pro práci s malými/velkými písmeny apod. je však k dispozici řada statických metod na typu char. Uveďme několik ukázek, na každém řádku je za značkou // uveden popis, co to dělá.

```
char.ToLower('A') //změní písmeno na malé
char.ToUpper('a') //změní písmeno na velké
char.IsLetter('a') //dotaz, zda je znak písmenem
char.IsDigit('a') //dotaz, zda je znak číslicí
```

Průvodce studiem

Když ve Visual Studiu napíšete do kódu char. (čili jméno typu a tečku), objeví se vám nabídka všech statických metod, které typ nabízí. Tato funkcionalita se nazývá Intellisense.

Pro znakové řetězce máme typ string. Řetězcové literály zadáváme do uvozovek.

```
string s = "Ahoj světe";
```

Řetězce můžeme konkatenovat (spojovat) operátorem `+`.

```
string s = "Ahoj" + " " + "světe";
```

Další operace s řetězcem lze provádět pomocí různých metod, voláme je přímo na objektech. Zde několik příkladů:

```
s.Substring(začátek, délka) //vrací podřetězec
s.IndexOf("vzor") //hledá prvního výskyt vzoru v řetězci a vrací jeho index
s.Length //délka stringu (používá se bez závorek)
s.Contains("vzor") //zjistí, zda řetězec obsahuje daný vzor
```

Průvodce studiem

Celou nabídku metod opět zjistíte pomocí Intellisense. Všimněte si, že string nabízí řadu metod jak přímo ve třídě (získáte je napsáním string.), tak i pro objekty (nabídku získáte až po založení proměnné typu string a napsání tečky za její jméno). Jde o různé metody.

Do řetězcových proměnných nelze přiřadit číselnou nulu. Chcete-li řetězec vymazat, přiřaďte do něj prázdné uvozovky `""`, nebo `null`. (Není to totéž – to první je řetězec nulové délky, to druhé je speciální značka, že proměnná neobsahuje nic, tedy ani prázdný řetězec. Co z toho je v praxi vhodnější, závisí na konkrétní situaci. K null například nelze konkatenovat další řetězec.)

2.1.4 Další základní typy

Uveďme ještě tři další základní typy. Typ `bool` uchovává pravdivostní hodnotu `true`/`false`. Typ `DateTime` uchovává datum a čas (nebo i jen jedno z toho). Pomocí `DateTime.Now` zjistíte aktuální datum a čas. Typ `TimeSpan` je časový rozdíl, vznikne odečtením dvou `DateTime` od sebe. (Používá se například při měření času.)

2.1.5 Referenční typy

String a většina ostatních typů, které jsou v BCL či našich programech, jsou referenční. U stringu to nemá prakticky na nic vliv, protože platí, že jednou vytvořený string už nelze změnit. (U neměnných objektů se smazává rozdíl mezi hodnotovými a referenčními typy. Proto je nám u stringu jedno, že je referenční. Nemá to v praxi žádný podstatný vliv.)

Proměnné všech typů se vytvářejí stejně, například pro int a string:

```
int a;
string s;
```

Odlišnost je jen u vytváření objektů. Proměnná referenčního typu totiž je něco jiného, než objekt. Je to jen prázdná reference. Objekty vytváříme vždy pomocí operátoru `new`, za který uvedeme jméno typu a do kulatých závorek můžeme uvést parametry oddělené čárkami. Toto

jsme používali už při práci s výjimkami v první kapitole. Výjimka je objekt, takže ji kromě vyhození můžeme i vytvořit jako každý jiný objekt a přiřadit do proměnné (i když to nemá žádný praktický smysl).

```
Exception e = new Exception("popis výjimky");
```

Také typ object je referenční. Jak víme, je to předek všech ostatních typů a potomek může zastoupit předka, takže do proměnné typu objet můžeme přiřadit cokoliv.

```
object o1 = 23;
object o2 = "Ahoj";
```

Důležitá poznámka: Každý objekt si „pamatuje“ svůj typ. Přiřazením objektu do proměnné typu `object` se neprovádí žádná konverze, takže objekt stále má svůj typ. Při další práci s takovými proměnnými je třeba vždy proměnnou přetypovat na správný typ. Přetypování se provádí stejně jako u konverze čísel. Jako příklad poslouží zásobník:

```
object o = new Stack<int>();
((Stack<int>)o).Push(10);
```

Totéž lze udělat také operátorem `as` (anglický termín pro „jako“). Tento zápis může být srozumitelnější a přehlednější:

```
(o as Stack<int>).Push(10);
```

Jak se to liší: Pokusíte-li se přetypovat proměnnou na nesprávný typ, tak první způsob vyhodí výjimku a program dál nepokračuje, zatímco druhý způsob dá jako výsledek takového přetypování null. Operátor `is` (anglicky „je“) může posloužit k otestování typu:

`if(o is int)` …je to číslo…

2.1.6 Převod ze stringu a na string

Zajímavou vlastností třídy `object` je metoda `ToString()`, která převede objekt na `string`. Jelikož všechny typy jsou odvozeny od `object`, všechny povinně poskytují tuto metodu. Tato metoda tedy vrací hodnotu objektu zapsanou jako string a tam kde to nemá smysl, vrací obvykle jméno třídy. (Můžete vyzkoušet, co udělá zavolání `ToString()` na vašem zásobníku.)

Opačná konverze, tj. ze stringu na jiný typ, není standardní či automaticky poskytovaná funkcionalita. U celé řady typů, hlavně těch základních, najdeme statickou metodu Parse, která bere string jako parametr a vrací jeho hodnotu typu dané třídy. Převod se nezdaří, když je stringu kromě samotné hodnoty ještě něco dalšího.

```
int.Parse("17");    //OK – vrací číslo 17
int.Parse("17abc"); //chyba
```

2.1.7 Reálná čísla a výpočty

Typ double používaný pro výpočty je stejný jako v jiných jazycích. V statické třídě System.Math najdeme celou řadu matematických funkcí, většinou pracujících právě s typem double. Některé však fungují i s celými čísly. Například:

```
Math.Sin(20) //sínus
Math.Abs(-5) //absolutní hodnota (i pro celá čísla)
```

Poznámka: System.Math je třída, takže nelze napsat using System.Math. Lze napsat jen usány System, a proto při používání jednotlivých funkcí musíte vždy psát Math.něco(…).

Generátor pseudonáhodných čísel je ve třídě `System.Random`. Používá se tak, že nejprve vytvoříme objekt typu `Random`, a pak na něm můžeme volat metodu `Next()`, která vždycky vrátí jiné náhodné číslo. Nepovinným parametrem je omezení rozsahu a vrací se pak číslo od 0 do max–1, neuvedeme-li maximum, tak se vrací jakékoliv číslo int.

2.2 Základní příkazy

Průvodce studiem

Základní příkazy jsou stejné jako v C++ (až na výjimečné drobné rozdíly, např. nutnost psát break u každého case).

2.2.1 Středníky a závorky

Ukončovací středník je součástí příkazu. (Na rozdíl od Pascalu to tedy není oddělovač, ale přímo součást a píše se jednoduše ke každému příkazu.)

Speciální význam mají složené závorky. Složené závorky jsou samy příkazem, nepíše se za ně středník. Dovnitř lze napsat libovolné množství příkazů (třeba i další složených závorek). Říkáme tomu blok (příkazů).

2.2.2 Podmínky – if, switch–case

Příkaz if je klasickým nástrojem pro podmíněné vykonávání kódu. Použije se takto:

`if(`podmínka`)` příkaz

`if(`podmínka`)` příkaz `else` příkaz

Podmínka se vždy musí uvést do kulatých závorek, pak hned následuje příkaz. Varianta s `else` přidává ještě další příkaz, který se vykoná při nesplnění podmínky.

Poznámka: Chceme-li při splnění či nesplnění vykonat více příkazů, použijeme blok ve složených závorkách (viz sekce 2.2.1).

Příkaz `switch` použijeme pro otestování více případů současně. Rozdíl oproti `if` je, že kód switch může být přehlednější a kratší (pokud se pro daný případ hodí použít). Příklad (vypíše den v týdnu podle jeho zkratky):

```
switch(zkratka) {
case "po":
  Console.WriteLine("pondělí");
  break;
case "út":
  Console.WriteLine("úterý");
  break;
default:
  Console.WriteLine("neznámá zkratka");
  break;
}
```

Jednotlivé případy tedy uvádíme za slovo `case` a přidáme dvojtečku. Pak následuje libovolný počet příkazů a blok musí končit `break;`. Na konci můžeme nepovinně přidat blok označený default:, jejíž kód se provede, když neplatí žádá jiná varianta.

2.2.3 Opakování – while, do–while, for

Příkaz while je základním konstruktem pro opakování kódu.

`while(`podmínka`)` příkaz

Funguje podobně jako if, ale dokud podmínka platí, tak se opakuje. Pokud podmínka neplatí hned na začátku, příkaz se neprovede ani jednou. Naproti tomu příkaz do–while provede příkaz

alespoň jednou, i když podmínka nikdy neplatí, protože ji testuje poprvé až po provedení příkazu. Z příkladu je to patrné (imperativně: nejprve je příkaz, pak teprve podmínka).

`do` příkaz `while(`podmínka`);`

U opakovacích příkazů opět často používáme bloky, abychom mohli opakovat víc než jeden příkaz. Pokud blok nepotřebujeme, celou opakovací konstrukci uvedeme na jeden řádek.

Složitější konstrukce nám umožní provádět příkaz `for`. Používá se stejně jako v C++ a příbuzných jazycích, uvedeme si příklad vložení čísel 1–5 do zásobníku:

```
for(int i=1; i<=5; i++) s.Push(i);
```

Příkaz `for` je podobný jako while, v kulaté závorce jsou tři sekce oddělené středníkem. První sekce se provede jen jednou na začátku a je určena k inicializaci proměnné, které budeme používat jako počítadlo opakování. Je možno použít existující nebo založit novou proměnnou (založení vidíme na příkladu). Ve druhé sekci je podmínka, stejně jako u while se testuje před každým vykonáním příkazu. Ve třetí sekci je příkaz, který se provede na konci každého vykonání příkazu. Můžeme si vše ještě uvést formálně. Příkaz `for` vypadá takto:

```
for(A;B;C) D
```

a dělá de facto toto:

```
{
  A
  while(B) {
    D
    C
  }
}
```

2.3 Pole

2.3.1 Obyčejné jednorozměrné pole

Pole je skupina objektů či hodnot stejného typu, která má pevnou velikost. Je to referenční typ.

```
int[] pole = new int[20];
Zásobník<int>[] zásobníky = new Zásobník<int>[10];
```

Jak vidíme na ukázkách, typ pole se označuje jako typ prvku a prázdné hranaté závorky. Vytvoříme-li takovou proměnnou, žádné pole v ní není (samozřejmě, je to přece reference), proto musíme volat new a tam uvedeme do hranatých závorek i velikost pole. Druhá ukázka vytváří pole zásobníků. Pozor na to, že vytvořením pole o 10 zásobnících, jak je v ukázce, nevytváříme žádné zásobníky, ale jen pole, ve kterém je 10× hodnota `null`.

K prvkům pole přistupujeme pomocí hranatých závorek a prvky jsou číslovány od nuly, např. `pole[0]` je první prvek `pole`. Pole můžeme předat jako parametr funkci či vrátit jako návratovou hodnotu (je to reference, takže s ním lze dělat cokoliv jako s běžnými proměnnými). Definujeme-li funkci tak, že přijímá jako parametr pole, tak ji lze zavolat s polem libovolného počtu prvků.

Máme-li pole nějakého typu, tak nezáleží na tom, od čeho je typ prvků odvozen. Pole je totiž vždy odvozeno od třídy `Array`. Například máme-li pole čísel `int[]`, tak není odvozeno od pole `object[]`, přestože typ `int` je odvozen od typu `object`. Toto je důležité vědět, protože chceme-li přijímat pole bez omezení typů, tak musíme parametr definovat jako `Array`, a ne `object[]`. (A aby to nebylo tak jednoduché, u polí referenčních typů je převod na pole bázového typu podporován. Proto to, co nefunguje pro pole čísel int[], funguje např. po pole řetězců string[] – to totiž lze na object[] převést. Použití `Array` je však univerzální.)

2.3.2 Operace s poli

Na polích lze provádět řadu operací, např. pomocí `Length` lze zjistit velikost pole (používá se bez závorek). Například metoda `Sort()` setřídí prvky pole, třídí od nejmenšího po největší, řetězce podle abecedy a je možno použít i jiné způsoby (pro začátečníky to není důležité). Třída `Array` nabízí i několik statických metod (viz [MSDN]).

2.3.3 Vícerozměrné pole - zubaté

Vícerozměrné pole můžeme vytvořit buď jako pole polí, což je stejné jako v C++, ale pro většinu použití zbytečně složité. Takové pole intuitivně definujeme pomocí přidání `[]` za typ.

```
int[][] dvojité = new int[20][];
```

Poznámka: Toto je často neprakticky složité, neboť jsme vytvořili jen pole `null` hodnot typu pole a je třeba v dalším kroku vytvořit ručně jednotlivá pole. Tento způsob se používá jen tam, kde chceme vnitřní pole vytvářet o různých velikostech, proto se mu taky říká zubaté.

2.3.4 Vícerozměrné pole – obyčejné

Obyčejné jednorozměrné pole vytvoříme uvedením čárek v jedné hranaté závorce.

```
int[,,] trojité = new int[10][10][10];
```

V ukázce jsme vytvořili trojrozměrné pole o velikosti 10×10×10 prvků.

2.4 Operátory

C# nabízí poměrně hodně operátorů, projdeme si je však jen velmi stručně. Binární aritmetické operátory jsou vesměs intuitivní: `+` (součet), `-` (rozdíl), `*` (součin), `/` (podíl), `%` (zbytek po dělení). Binární operátory `>>` a `<<` slouží pro bitové posuny o daný počet bitů.

Dále máme porovnávací operátory `>`, `>=`, `<`, `<=`, `==` (rovná se) a `!=` (nerovná se). Jejich výsledkem je hodnota typu bool, stejně jako u operátorů logických: `&` (and), `|` (or), `^` (xor). Operátory `&` a `|` je možno zdvojit do podoby `&&` a `||`, kdy pravá strana se vyhodnocuje jen tehdy, když to je potřeba pro vyhodnocení výrazu jako celku. (Například ve výrazu „`true || cokoliv`“ se `cokoliv` už nevyhodnocuje, protože je jasné, že výsledek bude `true`.)

Speciální ternární operátor `?:` se používá podobně jako příkaz if, ale pracuje s výrazy. Uveďme pro srovnání příklad:

```
if(výrazA) příkazB; else příkazC;

výrazA ? výrazB : výrazC
```

Rozdíl je tedy v tom, že výsledkem konstrukce `if` není nikdy výraz, zatímco konstrukci `?:` lze umístit dovnitř většího či složitějšího výrazu. (Tuto konstrukci nemusíte umět vytvářet, ale když na ni narazíte v cizím programu, měli byste ji dokázat pochopit.)

C# má také celou řadu přiřazovacích operátorů. Kromě klasického = můžeme vyrobit přiřazovací verzi od dalších binárních operátorů doplněním rovnítka z operátor. Například následující dva řádku dělají totéž:

A = A + B;

A += B;

U složitějších výrazů lze pro upřesnění pořadí použít závorky. Jinak se postupuje dle priorit operátorů. Nejvyšší prioritu mají unární a speciální tzv. primární operátory (jako `new`), potom */%, potom +–, potom >> <<, potom > >= < <=, potom == !=, potom &, potom ^, potom |, potom &&, potom ||, potom ?: a nakonec přiřazovací operátory.

Průvodce studiem

V této sekci jsme nezmínili všechny operátory C#, ale jen ty obvykle používané. Kompletní seznam včetně priorit najdete v [MSDN].

2.5 Výčtové typy

Výčtové typy (tzv. enumy) jsou zvláštní datové typy, které definují přesný seznam pojmenovaných hodnot. Proměnné těchto typů jsou hodnotové a nabývají jen hodnot z tohoto seznamu. Jako příklad si uveďme výčtový typ (enum) pro čtyři roční období:

```
enum RočníObdobí { Jaro, Léto, Podzim, Zima };
```

Jména hodnot se používají vždy dohromady se jménem typu, takže proměnnou založíme takto:

```
RočníObdobí období = RočníObdobí.Podzim;
```

Dodejme ještě, že výčtové typy jsou ve skutečnosti interně pouze čísla a seznam pojmenovaných hodnot jen určuje, jak se které číslo jmenuje. Technicky vzato lze proměnné výčtového typu přiřadit jakékoliv číslo.

Zajímavějším případem jsou však výčtové typy, které umožňují proměnným přiřadit i více svých hodnot současně. V knihovně BCL je poznáte podle toho, že je u nich uvedeno `[Flags]`, sami je však zatím vytvářet nebudeme.

Průvodce studiem

Pro zvídavé je zde jeden příklad, jak vypadá definice příznakového výčtového typu:

```
[Flags]
enum Médium { CD=1, Floppy=2, HD=4, USB=8 };
```

Tímto způsobem jsme definovali typ `Médium`*, jehož hodnoty mohou mít libovolnou kombinaci hodnot uvedených v seznamu. Tyto jsou číslovány vždy mocninami dvojky a spojují se operátorem* `|` *(svislá čára neboli „or“).*

2.6 Volání metod

Z našich úvodních příkladů v kapitole 1.3 už víme, že kód programu se píše do metod a taky je vlastně intuitivně již umíme definovat i volat. Metodu můžeme definovat pouze ve třídě. Uveďme několik příkladů metod:

```
public static void M1();
public void M2();
static void M3();
string M4(int p1, int p2);
```

Metoda M1 je veřejná statická (podobně jako main), lze ji tedy volat přímo na třídě. Metoda M2 je veřejná instanční, takže ji lze volat jen na objektech. Metoda M3 je naopak statická, ale neveřejná, takže ji lze volat na třídě, ale jen z jiné metody téže třídy. Metoda M4 je soukromá instanční a má dva parametry typu int. Tyto parametry můžeme v těle metody používat jako proměnné. Tato metoda navíc vrací hodnotu typu string. Hodnotu v metodě vracíme příkazem `return`, například tedy:

```
return "Vracíme string";
```

K příkazu `return` se nepíší kulaté závorky (samozřejmě: není to přece volání funkce). U metod, které nic vracet nemají, uvádíme místo typu návratové hodnoty slovo `void`.

Volání metod také již známe: Uvedeme jméno metody a do kulatých závorek uvedeme hodnoty k dosazení za parametry (těmto hodnotám říkáme argumenty). Metody bez parametrů musíme volat s prázdnými kulatými závorkami. Volání je výraz, takže lze vkládat jedno volání do druhého jako např.

```
Math.Abs(Math.Cos(x)).
```

Na tomto příkladě také vidíme poslední důležitou věc: Při volání musíme před jméno instanční metody uvést jméno objektu a před jméno statické metody jméno její třídy. Tato jména uvádíme buď celá, nebo zkráceně (za pomoci `using` direktiv, viz kap. 1.1.4).

Průvodce studiem

Pojem "výraz" znamená „cokoliv, co má nějakou hodnotu". Výraz je tedy něco, co lze použít na pravé straně přiřazovacího příkazu (máme-li třeba příkaz A=B; tak B je výraz). Pojem "příkaz" popisuje cokoliv, co lze uvést samostatně a něco to dělá. Například zmíněná konstrukce A=B; je příkaz, stejně tak if(...) {...} je příkaz, volání metody je příkaz, atp. Volání metod, které nejsou void, je příkazem i výrazem. Každý výraz však není příkazem, např. 2+3 je výraz, ale není to příkaz – nemá to přece smysl uvádět samostatně bez dalšího zpracování výsledku 5. V tomto ohledu se C# liší od C/C++, kde každý výraz je současně i příkazem.

2.7 Kolekce a příkaz foreach

2.7.1 Přehled

BCL nabízí řadu datových kolekcí. Kromě polí, které už známe, tedy máme k dispozici další způsoby, jak uchovávat více objektů či hodnot pohromadě. Seznam nejběžněji používaných kolekcí uvádí následující tabulka.

Jméno typu	Popis
ArrayList, List<T>	Dynamické pole
Hashtable, Dictionary<K,V>	Asociativní pole
SortedList, SortedDictionary<K,V>	Setříděné asociativní pole
Queue, Queue<T>	Fronta
Stack, Stack<T>	Zásobník
LinkedList<T>	Obousměrný spojový seznam

Tabulka 2. Základní datové kolekce BCL

U většiny kolekcí máme k dispozici obecnou i generickou verzi, přičemž ne vždy se tyto třídy jmenují stejně. Všechny obecné kolekce jsou v prostoru jmen `System.Collections` a všechny generické pak v prostoru `System.Collections.Generic`.

Průvodce studiem

Důvodem dvou sad kolekcí není ani tak snaha o obecnost, ale fakt, že generické prvky se do C# a .NETu dostaly až ve verzi 2.0 a původní kolekce v BCL pro zpětnou kompatibilitu zůstaly také. Mnohé nové generické kolekce mají také upravená jména tak,

aby vyjadřovala smysl/využití kolekce (např. dictionary – slovník), a ne způsob implementace (hash table – hešovací tabulka).

2.7.2 Dynamické pole

Dynamické pole je struktura, která se chová jako pole, ale umožňuje přitom měnit počet prvků. Nejčastěji se používá tak, že se vytvoří prázdná a přidávají se do ní prvky metodou `Add()`. (Nové prvky se tak řadí vždy na konec.) Na závěr se volá metoda `ToArray()`, která vrací obsah kolekce v podobě pravého pole (je to kopie, u které už pak nelze počet prvků měnit). Prvky jakéhokoliv pole lze také setřídit metodou `Sort()`.

Pozor na chování jakýchkoliv kolekcí při práci s referenčními prvky. Pokud okopírujeme kolekci, tak kopírujeme jen kolekci, ale nekopírujeme její prvky. Máme-li v kolekci referenční objekty, pak obě kolekce obsahují tytéž prvky. Když potom v jedné z kolekcí například odebereme jeden z prvků, ve druhé tento zůstává. Když ale změníme něco na prvku, tak se to mění v obou kolekcích.

2.7.3 Asociativní pole

Asociativní pole je zvláštní typ pole, kde prvky nejsou na pozicích indexů 0, 1, 2…, ale můžeme jim dát jakékoliv indexy a dokonce to ani nemusejí být čísla. Můžeme si například vytvořit pole přiřazující názvům měsíců roční období, kam patří.

```
Dictionary<string, RočníObdobí> a = new Dictionary<string, RočníObdobí>() {
  { "Leden", RočníObdobí.Zima },
  { "Únor", RočníObdobí.Zima },
  { "Srpen", RočníObdobí.Léto },
};
```

Na tomto příkladě jsme si zároveň předvedli, jak lze elegantně inicializovat pole přímo v definici – jednoduše uvedeme seznam hodnot do složených závorek, u asociativního pole uvádíme seznam čárkami oddělených párů { klíč, hodnota}. Za posledním prvkem v seznamu je možno uvést ještě jednu čárku (tak, jak je to v ukázce), ale není to povinné.

Asociativní pole se používají k evidenci a rychlému nalezení hodnoty podle klíče. Metody `ContainsKey()` a `ContainsValue()` slouží ke zjištění, zda daný klíč či hodnota v poli již je. Pro vkládání nových hodnot se obvykle používá tento scénář:

```
if(!kolekce.ContainsKey(klíč)) kolekce.Add(klíč, hodnota);
```

Setříděná asociativní pole uchovávají prvky seřazené podle klíčů, což se projeví jen při procházení celé kolekce (příkazem `foreach`).

Průvodce studiem

Jen pro zajímavost: Inicializace asociativního pole pomocí párů klíč+hodnota ve vnořených složených závorkách je možná až od C# verze 3.0.

2.7.4 Fronta, zásobník a spojový seznam

Na frontu a zásobník jsme již narazili v první kapitole. Pro vkládání a odebírání prvků má zásobník metody `Push()` a `Pop()`, fronta pak `Equeue()` a `Dequeue()`. Obě třídy mají stejně pojmenovanou metodu `Peek()`, která vrací další prvek na řadě bez toho, aby jej odebrala z kolekce.

Obě tyto třídy také podporují procházení pomocí `foreach` (což je možná z hlediska teorie datových struktur zvláštní, ale v .NETu skutečně všechny kolekce podporují procházení).

Obousměrný spojový seznam `LinkedList<T>` je zobecněním fronty a zásobníku, kde lze prvky přidávat i odebírat z libovolného místa (tedy z obou konců, i z vnitřních uzlů). Na koncích to zajistí metody `AddFirst()`/`AddLast()` a `RemoveFirst()`/`RemoveLast()`, popis dalších metod můžete najít v [MSDN].

2.8 Property (vlastnosti)

Kromě metod a proměnných, což jsou základní součásti tříd a umožňují nám naprogramovat řadu programů bez potřeby dalších složitých konstrukcí, mohou třídy obsahovat také další prvky. Jeden z nich se nazývá property (česky vlastnosti) a používá se tak často, že si jej také musíme vysvětlit alespoň z hlediska uživatelského.

Property je dvojice spřažených metod, která se tváří a používá jako proměnná. Jedna metoda slouží k přečtení hodnoty, druhá k nastavení hodnoty této pseudo–proměnné. Property ve skutečnosti žádnou proměnnou nemá, jsou to skutečně jen dvě metody. S property jsme se v předchozím textu již setkali a nenápadně jsme je přešli poznámkou, že se používají bez závorky. Například property ke zjištění délky řetězce se jmenuje Length. Používá se bez závorky, jako by to byla proměnná, je to ale property. (V BCL nejsou proměnné ve třídách nikdy veřejné, vždycky to jsou pouze property.) Property Length je také příkladem toho, že je možné mít i property jen pro čtení či jen pro zápis.

Průvodce studiem

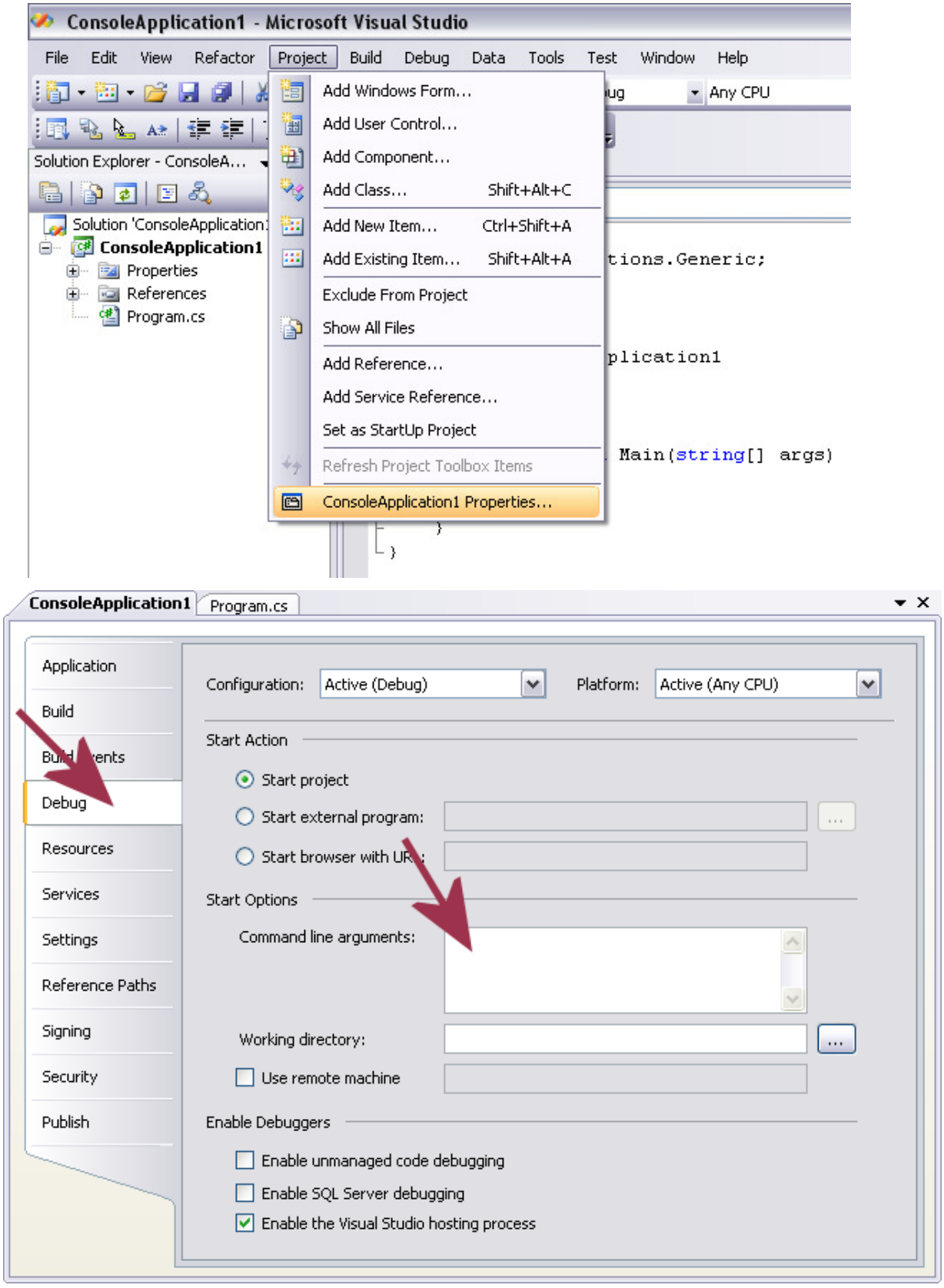
Property jen pro čtení je vlastně jako metoda bez parametrů, která vrací hodnotu, ale používá se bez prázdných kulatých závorek. Fakt, že C# má dva tolik podobné konstrukty, vede často ke zmatkům. Visual Studio vám při špatném použití prázdných kulatých závorek vypíše chybu při překladu.

2.9 Textový vstup a výstup

Podívejme se nyní na standardní (textový) vstup a výstup, což pro začátečníky můžeme přeložit jako „vstup z klávesnice a výpis na obrazovku“.

Pro psaní na obrazovku již známe statickou třídu `Console`, v ní jsou statické metody `WriteLine()` a `Write()`, které vypíší text na obrazovku a první z nich také odřádkuje.

Pro čtení vstupu z klávesnice můžeme použít `Console.ReadLine()`. Často se hodí také použití startovacích parametrů programu, které nastavíme jako řetězec ve vlastnostech projektu a pak jej dostaneme rozdělený na jednotlivá slova jako pole stringů ve spouštěcí metodě Main. (Používání startovacích parametrů je vhodné si hned vyzkoušet v praxi!) Nastavení spouštěcích parametrů ukazuje Obrázek 6: Konfigurační okno otevřeme pomocí posledního řádku v menu Project. Parametry pak na kartě Debug vepíšeme do Command line arguments.



Obrázek 6. Nastavení spouštěcích parametrů programu.

2.10 Čtení a zápis souborů

BCL nabízí velké množství tříd pro práci se soubory a adresáři, pro tento okamžik se naučíme dělat některé základní operace, se kterými si pro řadu úloh bohatě vystačíme.

Třída `File` nabízí několik užitečných statických metod, kterými je možno přečíst či zapsat soubor bez nutnosti učit se celý systém práce se soubory. A to je pro začátečníky ideální.

`File.ReadAllBytes("cesta");` //načte binární soubor jako pole bajtů
`File.ReadAllLines("cesta");` //načte textový soubor jako pole řetězců, každý je jeden řádek
`File.ReadAllText("cesta");` //načte celý textový soubor jako jeden string

U metod pracujících s textovým souborem je možno jako druhý parametr uvést použité kódování češtiny. Nezadáte-li nic, předpokládá se kódování unicode s hlavičkou. Obvykle používané kódování Windows–1250 se zadá takto: `Encoding.GetEncoding("windows-1250")`.

Průvodce studiem

Na české verzi Windows lze české kódování 1250 vybrat pomocí Encoding.Default. Můžete to však použít jen pro své vlastní testování, neboť nikdy nevíte, jakou verzi Windows mají uživatelé vašich programů. Pozor taky na to, že toto „default" je jiné, než nezadat žádné kódování (pak se totiž použije unicode, jak je uvedeno výše).

```
File.WriteAllBytes("cesta", pole_bajtů);  //zapíše byte[] jako soubor
File.WriteAllLines("cesta", pole_řetězců);//zapíše string[] jako řádky souboru
File.WriteAllText("cesta", řetězec);      //zapíše string jako celý soubor
```

Shrnutí

Cílem této kapitoly bylo poskytnout stručný přehled základních programových konstrukcí jazyka C#. Navázali jsme tak na předchozí kapitolu, kde jsme si již některé prvky ukázali na příkladech.

Pojmy k zapamatování

- Hodnotový typ, referenční typ
- Číselné typy (int, uint, short, ushort, long, ulong, double, char, byte, sbyte)
- Znakový literál
- Znak (char) a znakový řetězec (string)
- Pravdivostní hodnota (bool)
- Datum a čas (DateTime) a časový rozdíl (TimeSpan)
- Podmínka, opakování
- Pole
- Obyčejné a zubaté vícerozměrné pole
- Operátor
- Výčtový typ
- Datová kolekce, foreach
- Dynamické pole (ArrayList, List<T>)
- Asociativní pole (Hashtable, Dictionary<K,V>)
- Setříděné asociativní pole (SortedList, SortedDictionary<T,K>)
- Fronta (Queue, Queue<T>)

- Zásobník (Stack, Stack<T>)
- Obousměrný spojový seznam (LinkedList<T>)
- Property (vlastnost)
- Textový vstup a výstup (třída Console)
- Spouštěcí parametr
- Soubor (třída File)
- Kódování češtiny (třída Encoding)

Kontrolní otázky

1. *Vysvětlete rozdíl mezi hodnotovými a referenčními typy v .NETu.*
2. *Vyjmenujte základní číselné datové typy a jejich přesnost. (Ano, toto je skutečně třeba znát nazpaměť.)*
3. *Co jsou to „znaménkové“ a „neznaménkové“ typy? Jmenujte několik příkladů každého typu.*
4. *Které datové typy má .NET pro práci s datem a časem? Proč jsou dva? Vysvětlete jejich použité na nějakém vhodném příkladu.*
5. *Jakým způsobem převádíme string obsahující zápis čísla na skutečné číslo a obráceně číslo na string obsahující jeho zápis?*
6. *Příkaz v C# je ukončení středníkem, zatímco v jiném programovacím jazyce Pascal je středík oddělovačem příkazů. Vysvětlete rozdíl mezi těmito velmi podobnými definicemi.*
7. *Jaký je rozdíl mezi cykly while a do–while?*
8. *Vysvětlete syntaxi příkazu for. (Je to jeden z nejpoužívanějších příkazů, proto byste jej měli dobře znát a umět v případě potřeby okamžitě použít.)*
9. *Na příkladu ukažte, jak se v C# vytváří pole.*
10. *Víme, že pole objektů obsahuje jiné objekty. Je také pole jako celek objekt? A pokud ano, je odvozen od typu object jako ostatní objekty? Vysvětlete, proč.*
11. *Vysvětlete rozdíl mezi obyčejným a zubatým vícerozměrným polem.*
12. *Co je to výčtový typ?*
13. *Co je to property (vlastnost)?*
14. *Jmenujte některé třídy kolekcí, které nabízí knihovna BCL. Vysvětlete jejich význam (neboli rozdíly mezi nimi).*

3 Objektově orientované prvky

Studijní cíle: Tato kapitola se věnuje vysvětlení pojmů objektově orientovaného programování (dále jen OOP) a studenti se především naučí používat je v jazyce C#.

Klíčová slova: objekt, třída, rozhraní, viditelnost, dědičnost, metoda, proměnná, volání

Potřebný čas: 100 minut

3.1 Základní pojmy

Objektově orientovaný přístup je při programování klíčem k úspěchu. Ačkoliv existují i jiné způsoby, jak efektivně programovat, objektově orientovaný způsob je nejvhodnější pro většinu situací, se kterými se člověk setká. Při správném způsobu výuky a volbě vhodných nástrojů je tento způsob nakonec i intuitivní. V Softwarové laboratoři není dostatek prostoru, abychom se zabývali všemi detaily, musíme se naopak zaměřit jen na zvládnutí základních pojmů.

Průvodce studiem

V zájmu objektivity nutno dodat, že není pravda, že by se celá programátorská obec nějak „shodla" na tom, že právě objektově orientovaný přístup je jediný správný. V praxi však jednoznačně převažuje procedurální a objektový styl programování (na Google lze najít i konkrétní výsledky průzkumů, dva jmenované tvoří cca 99%). Poměr používání nějaké techniky v praxi sám o sobě ještě není důkazem o její nadřazenosti, ale drtivá převaha již o něčem svědčí. Samotný jazyk C# je nativně právě jakousi kombinací imperativního, procedurálního a objektově orientovaného programování, kde dílčí příkazy, které jsme je probírali v předchozí kapitole, jsou v imperativním stylu, a celková struktura programů se řeší prostředky objektově orientovanými, které zde více méně zobecňují procedurální programování a jsou tématem této kapitoly.

Z předchozích kapitol už víme, že v programech se vyskytují třídy a objekty. Víme také, že objekt je jednou konkrétní instancí třídy. Můžeme také říkat, že třída je typem objektu. Programování s třídami a objekty jsme zatím byli víceméně přinuceni tím, jak funguje prostředí .NET a BCL – zdálo by se skoro, že nic než třídy a objekty tam není…

Proč vlastně takto máme programovat? Hned na začátku studia jsme pracovali se zásobníkem – kolekcí, která je navržena zcela dynamicky tak, aby dokázala pojmout libovolné množství prvků. Každý zásobník byl objektem, takže jsme mohli vytvořit jen jeden kód (metody) a ty pak znovu a znovu používat u dalších zásobníků. U zásobníku jsme také používali další třídu pro vyjádření jednoho prvku řetězu, ve kterém si zásobník pamatuje prvky, které jsou do něj vloženy. Tento celý příklad poměrně jasně ukázal možnosti OOP u dynamických struktur (tj. tam, kde předem nevíme, kolik čeho budeme potřebovat – vytvoříme si až při běhu programu tolik objektů, kolik bude třeba).

Objektově orientovaný přístup je ale přínosem i u méně dynamických struktur. Tím, že uzavřeme skupinu metod do třídy společně se skupinou proměnných, se kterými pracují, máme program přehlednější a bezpečnější. Proměnné objektu nikdy nejsou veřejné (public) a tak jediným způsobem, jak změnit něco v objektu je použitím některé jeho metody. Metody pracující se stejnými proměnnými jsou ve stejné třídě, takže související části kódu máme pěkně

pohromadě. Pokud při programování většího projektu zjistíme, že něco nefunguje, hned víme, kde hledat chybu – bude-li se třeba chovat divně zásobník, tak chyba může být jedině v souboru obsahujícím metody třídy Zásobník. Také v okamžiku, kdy chceme změnit nějakou již hotovou část kódu, nemusíme složitě hledat, kde všude se používá a co všechny by potenciálně mohlo přestat fungovat. Místo toho stačí sledovat jednu dotčenou třídu. Třídy nám také rozdělují program do samostatných částí, které je možno vzít a přenést do jiného programu. V praxi například při tvorbě informačních systémů se často opakují stejné podúlohy, například skoro každá aplikace má nějaký adresář zákazníků či jiných osob – je-li toto ve třídě, pak můžeme při práci na projektu vzít třídu ze staršího již hotového projektu a nemusíme složitě hledat, kde přesně které části kódu je potřeba okopírovat. (Případně můžeme vzít více spolupracujících tříd, jde-li o složitější úlohu.)

Opusťme nyní konkrétní příklady a přenesme se do obecné roviny. Popíšeme si základní vlastnosti každého objektu [Kra99]:

1. Objekt obsahuje vnitřní paměť, tj. může si něco pamatovat. (Tedy jsou v něm nějaké proměnné, obvykle jim odborně říkáme *atributy objektu*.) Vnitřní paměť je soukromou záležitostí objektu, okolní svět se nestará a nemůže starat o to, co a jak přesně si objekt pamatuje.
2. Objekt obsahuje metody. Tyto metody jsou také vnitřní součástí objektu a pracují s atributy objektu a jenom ony to mohou dělat (neboť k vnitřním atributům není niko jiný oprávněn přistupovat).
3. Objekt dokáže nějakým způsobem přijmout zprávu z vnějšku (od jiných objektů) a zpracovat ji. Jednotlivé programovací jazyky se v detailech liší, každý ale nějakým způsobem musí umožnit definovat, která metoda má odpovídat které zprávě. Objekt tedy nejspíše bude mít nějakou tabulku, která pro každou konkrétní zprávu určí, která metoda ji má zpracovat. Poslání zprávy objektu je jediným způsobem, jak může okolní svět s objektem komunikovat.
4. Objekt může obsahovat jiné objekty či odkazy na ně. Tak je zajištěn samotný život programu spočívající v tom, že jednotlivé objekty si postupně posílají zprávy.

Všimněte si, že poslední bod definuje samotný život programu, tj. jak se program rozhýbe k tomu, aby vůbec něco dělal. Pro běh programu v objektově orientovaném prostředí tedy nejsou podstatné příkazy, které jsme probírali v předchozí kapitole, ale posílání zpráv mezi objekty.

Důležité je také porozumět bodům 2 a 3. Doposud jsme vždy ke třídám přistupovali tak, že obsahují nějaké metody a ty můžeme volat. Ve skutečnosti jsme však jen používali zjednodušený princip volání, kdy jazyk C# jako výchozí činnost automaticky překládá zprávu na stejnojmennou metodu. Nadefinujeme-li metodu, která není public, je to jen metoda bez zprávy. Proto ji nelze zavolat. Nadefinujeme-li metodu jako veřejnou, překladač automaticky přidá stejnojmennou zprávu a přiřadí ji k této metodě. Později se i naučíme, jak vytvářet pouze zprávy bez metod apod. Přitom po chvíli přemýšlení jistě každého studenta napadne i řada dalších způsobů jak provádět přiřazování zpráv v objektu. Z praxe už například známe metodu `ToString()`, kterou definuje třída `object` a můžeme díky ní převést kterýkoliv objekt na řetězec. Jak je toto ve skutečnosti implementováno? Třída `object` ve skutečnosti definuje jen nejhloupější možnou metodu `ToString()`, která vrací jen jméno třídy, ze které je daný objekt, a zároveň tuto zprávu označuje příznakem `virtual`, který povoluje, aby potomci třídy `object` přesměrovali zprávu na jinou metodu. Jednotlivé třídy pak definují vlastní metody `ToString()` a přiřazují je k této zprávě. Tento způsob změny chování zpráv při dědění je velmi silným nástrojem (a dalším příkladem výhod objektově orientovaného přístupu, i když není tak snadno pochopitelný na první pohled).

Průvodce studiem

Budeme-li důslední v terminologii, tak metodu objektu nelze nikdy zavolat. Pojem „volání metody“ ve skutečnosti vždy znamená, že objektu posíláme zprávu. Při „volání statické metody“ pak posíláme zprávu přímo třídě. Dokonce i když se volají metody uvnitř jednoho objektu mezi sebou, tak také jen posílají zprávu svému objektu.

Přestože tedy nyní už víme, že naše dosud používané názvosloví není úplně korektní, budeme se jej i nadále držet. Každé poslání zprávy tedy budeme označovat jako „volání“.

Zajímavé je také srovnání objektů a tříd na jedné straně a datových struktur na straně druhé. Datové struktury (či strukturované datové typy) je možno vytvářet v řadě programovacích jazyků a na první pohled jsou dosti podobné třídám. Rozdíl je především v tom, že datové struktury z principu slouží k uchování většího množství dat pohromadě, ale nemají vlastní metody a hlavně nepoužívají systém zasílání zpráv. V důsledku toho tam tedy není možné ani nějak speciálně upravovat mapování zpráv na metody, jak to lze v třídách a objektech. (To, že dva různé objekty mohou reagovat na zaslání téže zprávy vykonáním různých metod, nazýváme *polymorfizmus*. Tento termín teď pro nás není podstatný, uvádíme si ho jen proto, že v jiné literatuře se často používá. Při našich znalostech jde již o zcela intuitivní součást programování – má-li každý objekt zvlášť metody a zvlášť tabulku určující, co při které zprávě udělat, pak pochopitelně na stejnou zprávu mohou různé objekty reagovat různě.) Hovoříme-li o stejné zprávě, pak v C# nestačí mít dvě zprávy stejně pojmenované, ale musí jít opravdu u jednu a tutéž zprávu. Sílu polymorfizmu lze předvést na jednoduchém příkladě: Vytvoříme obecnou funkci, která vypíše na obrazovku hodnotu objektu. (Říkáme „obecnou“, protože bude akceptovat jakýkoliv typ, podobně jako obecné kolekce.)

```
static void VypišHodnotuObjektu(object o) {
  Console.WriteLine(o.ToString());
}
```

Správnost můžeme hned otestovat:

```
int a = 23;
DateTime b = DateTime.Now;
VypišHodnotuObjektu(a);
VypišHodnotuObjektu(b);
```

Tento program vypíše číslo 23 a pod něj aktuální datum a čas.

3.1.1 Třída

Třída je jakousi pomůckou k vytváření objektů. Třída vlastně definuje nějakou základní podobu objektu daného typu. Operátor `new`, který už známe, můžeme chápat jako nástroj, který vytvoří duplikát tohoto základního objektu. Jelikož už jsme se třídami hodně pracovali, není třeba zacházet nyní do podrobností na úrovni teorie. Musíme si však vysvětlit jednu důležitou vlastnost C#: Každá třída je objektem typu Type. Co to znamená? Platforma .NET obsahuje jedná základní třídu jménem Type. Každá třída počínaje třídou `object` a konče všemi našimi třídami je objektem tohoto typu Type. Tento princip nám umožňuje pracovat s třídami jako objekty a už jsme jej dokonce používali, i když jsme si to neřekli: Statické metody jsou vlastně metodami třídy, a ne metodami objektu. Voláme-li statickou metodu, pak ve skutečnosti posíláme zprávu objektu třídy. Ačkoliv toto může na začátečníky v objektovém světě působit děsivě, umožňuje to další zjednodušení programování a souvislostí uvnitř systému. Jazyk C# je v tomto směru ještě docela omezený, některé jiné objektově orientované jazyky jdou až tak daleko, že v nich platí: „Všechno je objekt.“ (Jistě si dovedete představit, že třeba blok kódu ve složených závorkách by mohl být objektem, jazyk C# však až tak daleko nejde. Všechny typy ale objekty skutečně jsou, tím je C# více objektový než třeba podobný jazyk Java, kde například základní číselné typy objekty nejsou.)

3.1.2 Dědičnost

Dědičnost jsme si předváděli už v první kapitole na třídách výjimek. Nyní při znalosti principu posílání zpráv je jasné, že dědičnost je hlavně způsob, jak tvořit třídy tak, že obsahují jiné třídy. Skutečně, tak jako může objekt obsahovat jiný objekt, může i třída obsahovat jinou třídu. Aby to bylo snáze představitelné, hovoříme o dědění či odvození typu. Když třída B zdědí třídu A, znamená to, že zdědí všechny její metody a proměnné a také její tabulku zpráv, přičemž ji může libovolně upravit. Prostředky, kterými můžeme konkrétně v jazyce C# měnit tabulku zpráv při dědění nejsou sice příliš bohaté, ale zato jednoduché: Můžeme přidat nové zprávy a změnit metody u existujících zpráv. Třída v C# navíc může také přikázat, že ji nikdo nesmí zdědit, nebo také u konkrétních zpráv zakázat jejich změnu v potomkovi. (Pro zajímavost: Jazyk C++ umožňuje také dědit třídu tak, že její zprávy nebudou v potomkovi dále k dispozici.)

Smyslem dědičnosti je snížit množství opakovaného kódu. V praxi se běžně stává, že máme velmi podobné třídy (často i více než dvě) lišící se jen v několika málo metodách či proměnných. Jejich společnou část pak můžeme umístit do základní třídy (předka) a lišící se části pak do jednotlivých potomků. Takový systém tříd se pak používá způsobem, který jsme již potkali dříve: Potomek může vždy zastoupit předka, takže stačí napsat kód pro použití s předkem a lze jej pak používat i s každým potomkem (viz ukázku metody `VypišHodnotuObjektu()` výše). Podobnou konstrukcí jsou pak rozhraní (anglicky interface). Rozhraní je z hlediska objektově orientované teorie zvláštní třídou, která obsahuje jen předpis zpráv, ale nic dalšího. Rozhraní vlastně tedy jen definuje, jaké zprávy by měl objekt přijímat. Takovou definici pak použijeme tím způsobem, že toto rozhraní de facto zdědíme a doplníme ke všem zprávám metody. V jazyce C# se místo „dědění“ u rozhraní hovoří o „implementaci“ – má to čistě jazykový důvod. Rozhraní tedy vlastně předepisuje, co má objekt umět a třída jej implementuje, čímž o sobě deklaruje, že její objekty toto umějí.

Průvodce studiem

V praxi se rozhraní používá třeba k tomu, aby kolekce mohla být použita v příkazu foreach. Náš zásobník by musel implementovat rozhraní IEnumerable a tím by o sobě řekl, lze procházet jeho obsahem. I jiná použití rozhraní v BCL mají často tento význam: Rozhraní definuje určitou funkcionalitu. Třída pak implementuje toto rozhraní proto, aby dala najevo, že přesně tuto funkcionalitu (klientům) dokáže poskytnout.

Poznámka: Každé rozhraní je typem, čili už známe dva druhy typů: třídy a rozhraní. Můžeme tedy vytvářet proměnné takového typu, předávat jej jako parametr volání metody apod. Rozhraní však není vzorem pro vytváření objektů (pochopitelně), operátor `new` tedy na něj použít nelze. Implementované rozhraní se samozřejmě v mnoha věcech chová jako klasický předek při dědění, například každý objekt může zastoupit nejen svého předka–třídu ale i rozhraní, které implementuje.

3.1.3 Anonymita klienta

Programy v OOP musejí dodržovat anonymitu klienta. Znamená to, že třída při zpracování zprávy nesmí brát zřetel na to, kdo ji zprávu poslal. Anonymita klienta je pravidlem hygieny v OOP, tzn. není to pravidlo vynucené jazykem, překladačem či systémem. V jazyce C# sice z principu nevíme, kdo nám metodu zavolal, ale můžeme to obejít tak, že jako jeden z parametrů metody budeme požadovat odkaz na odesílatele zprávy. Uděláme-li to, technicky bude program v pořádku, ale dopustíme se prohřešku vůči pravidlu anonymity klienta.

Průvodce studiem

Základním nástrojem k řešení těch složitějších situací, které svou povahou k poručení tohoto pravidla svádějí, je dědičnost.

3.2 Objektově orientované prvky v C#

3.2.1 Viditelnost

S viditelností jsme se setkali již na začátku studia, když jsme používali slovo public. C# definuje ještě několik dalších klíčových slov pro nastavení viditelnosti entity, říkáme jim také modifikátory viditelnosti či přístupu. (Entita nebo také součást je obecný termín pro nějaký prvek v jazyce.)

Průvodce studiem

Viditelnost se týká přitom jak součástí tříd, tak také součástí seskupení (assembly, někdy také překládáno jako sestavení, někdy se v češtině i přímo používá anglický pojem assembly). Pojem seskupení označuje v .NETu jednotku výsledného programu, obvykle je to tedy „exe" či „dll" soubor, který je výsledkem našeho programování. Viditelnost tedy popisuje jednak viditelnost entity z jiné třídy, ale také viditelnost entity z jiného seskupení. Vytváříme-li program jako „exe", pak jej již nelze vložit do jiného programu a nastavování viditelnosti na úrovni seskupení je tedy zbytečné. Překladač ale přesto může odmítat program přeložit, pokud se budete snažit používat méně viditelné prvky ve více viditelné třídě na úrovni seskupení. Není proto od věci vytvářet entity na úrovni seskupení buď vždy veřejné, nebo vždy bez uvedení viditelnosti (tedy soukromé).

Vysvětlíme si čtyři základní modifikátory viditelnosti:

- `private` – Entita je viditelná jen v rámci své třídy.
- `protected` – Entita je viditelná v rámci své třídy a také z odvozené třídy.
- `public` – Entita nemá omezení viditelnosti.
- `internal` – Entita není vidět z jiných seskupení, ale v rámci svého nemá viditelnost omezenou.

3.2.2 Třídy

Třídy už vytvářet umíme. Víme také, že prostor jmen sám o sobě nic netvoří a slouží jen ke zkrácení dlouhých jmen. Třídu lze definovat buď přímo na globální úrovni, v prostoru jmen, nebo uvnitř jiné třídy. Definovat třídu uvnitř jiné je vhodné v kombinaci s viditelností private – je to pak vnitřní součást třídy, která není zvenku vidět. Tento způsob je vhodný například u prvku zásobníku – třída pro prvek má význam jen uvnitř zásobníku, protože ve zprávách podporovaných zásobníkem se nevyskytuje. Takže je lepší ji do zásobníku rovnou „ukrýt" a nekomplikovat tak život zbytku programu tím, že by viděl něco, co nepotřebuje.

Průvodce studiem

Každá třída má v programu nějaký smysl, má nějaký úkol a je za jeho plnění odpovědná. Existuje proto pravidlo: Třída má dělat jen to, co má za úkol, a nic navíc.

Toto pravidlo říká, že do třídy nedáváme zbytečné veřejné zprávy, které nejsou nutné pro to, aby tato třída splnila svůj účel. Smyslem tohoto pravidla je přehlednost kódu a v důsledku toho také menší chybovost.

3.2.3 Rozhraní (interface)

Rozhraní definujeme klíčovým slovem interface. V rámci .NETu je zvykem pojmenovávat je s velkým písmenem `I` na začátku. Je dovoleno i mít stejně pojmenovanou třídu a rozhraní s tím, že rozhraní má ono `I` na začátku názvu navíc a předepisuje, jaké operace má třída nabízet, a třída toto rozhraní implementuje a předepsané operace nabízí.

```
interface Něco {
  void DělejNěco();
}
```

Všimněte si, že součásti rozhraní nemají danou viditelnost. Rozhraní totiž předepisuje zprávy, ty jsou vždy veřejné, tudíž jakoby vše v rozhraní bylo automaticky public.

3.2.4 Dědění tříd a implementace rozhraní

Uveďme nejprve příklad:

```
class Potomek : Předek, IRozhraní {
  …
}
```

Kolikrát lze dědit: Třída může mít jen jednoho předka, může však implementovat libovolný počet rozhraní. Někdy se tomuto omezení také říká: „Třída může dědit jen jednu implementaci, ale neomezený počet rozhraní.“ (Některé jazyky toto omezení nemají, pak hovoříme o „vícenásobné dědičnosti“, nebo dokonce „opakované dědičnosti“, když třída dědí víckrát stejného předka. Takové konstrukce .NET vůbec nepodporuje, naštěstí je známo, že tyto konstrukce jsou obvykle známkou chybného návrhu programu, takže stávající omezení je nakonec spíše k dobru.)

C# a také přímo .NET definují poměrně přísná pravidla pro komunikaci mezi potomkem a jeho předkem, opět je to hlavně z důvodu snahy o přehlednost a bezchybnost programů. Dědic je navíc klientem předka a platí anonymita klienta, takže předek o potomkovi neví. Potomek naopak nezná předky svého předka, ale jen toho jednoho předka, ze kterého je přímo odvozen. (Říkáme „přímý“ a „nepřímý“ předek/potomek.)

První kontakt s předkem máme už v konstruktoru. Konstruktor musí inicializovat celý objekt, tj. včetně předka. A k inicializaci předka slouží jeho konstruktor, který samozřejmě potomek použije. S použitím konstruktoru předka jsme se již setkali v kapitole 1.3.7, připomeňme si znovu ten příklad:

```
class VýjimkaZásobníku : Exception {
  VýjimkaZásobníku(string zpráva) : base(zpráva) { }
}
```

Jak vidíme, konstruktor předka voláme pomocí dvojtečky a slova **`base`** (anglický výraz pro bázi, základ), které obecně označuje předka. Nepíšeme sem tedy přímo jméno třídy předka (nemá to smysl, protože volat lze jen konstruktor přímého předka a ten je vždy jen jeden).

Proč vlastně takto explicitně voláme konstruktor předka? Obvykle proto, že předkovi do konstruktoru potřebujeme předat parametry. Ty nemusejí být vždy stejné, jako parametry

našeho konstruktoru, ale libovolné dle potřeby. Předek také může mít více konstruktorů a pak musíme určit, který z nich má být použit. Pokud však předek má konstruktor bez parametrů, nemusíme jej uvádět jako `: base()`, bude totiž zavolán automaticky.

Slovo `base` lze používat i v metodách třídy, odkazuje nás na objekt předka. Podobně slovo `this` nás odkazuje přímo na náš objekt. Tyto pomůcky se hodí nejčastěji v případech, kdy máme více entit se stejným jménem. Máme-li třeba lokální proměnnou `prom` a zároveň proměnnou `prom` i v objektu, pak pomocí `this.prom` pracujeme s proměnnou objektu a pomocí `prom` pracujeme s lokální proměnnou (lokální entita má přednost, protože „košile bližší než kabát"). Slovem `base` si podobně pomůžeme, když se nám překrývají stejně pojmenované entity v naší třídě a třídě předka. Více o překrývání jmen v další sekci.

Závěrem ještě dodejme, že také `this` lze použít za dvojtečkou u konstruktoru. Zavoláme tím jiný konstruktor téže třídy a používá se to tehdy, když by se kód několika konstruktorů podobal. Místo opakování stejného kódu nadefinujeme jeden konstruktor celý, ze druhého jej zavoláme a pak jen změníme to, co se má lišit. (Přínosem je tedy kratší kód, neopakování/znovupoužití a v důsledku i menší chybovost.) Nemůžeme ale použít `this` i `base` současně.

3.2.5 Metody a překrývání jmen

Překladač C# vynucuje určitá omezení co do překrývání jmen, například proměnná v bloku nesmí zakrýt stejně pojmenovanou proměnnou nadřazeného bloku. Ukázka chybného kódu je nasnadě:

```
int a;
{
  int a; // chyba – nelze zakrýt proměnnou stejného jména v nadřazeném bloku
}
```

U metod však máme lepší možnosti. Předně je možno definovat libovolný počet stejně pojmenovaných metod. Jméno totiž není jediným prvkem identifikace, k rozlišení dvou metod totiž stačí, aby měly odlišné parametry. (Naopak rozdílnost pouze v typu návratové hodnoty nestačí.) Tato praktika se nazývá *přetížení* (anglicky *overloading*, každá varianta metody je pak nazývána *overload*). Přetížení se od překrytí liší tím, že jsou vidět všechny varianty současně, zatímco při překrytí jedna varianta zakrývá jinou.

C# nedovoluje definovat tzv. *nepovinné parametry*, takže při každém volání musíme zadat hodnoty všech parametrů a přetížení je tedy způsob, jak lze nepovinné parametry napodobit (a proto se třeba v BCL používá často).

Průvodce studiem

C# nezná ani funkce s proměnlivým počtem parametrů. Lze to však napodobit pomocí parametru typu pole, který na začátku označíme slovem `params`*. Zde je příklad:*

```
void MojeMetoda(params object[] pole);
```

Uvnitř této metody máme k dispozici všechny předané argumenty v jednom poli. Zavolat tu metodu lze buď skutečně s polem, nebo s libovolným počtem libovolných argumentů – je to jen syntaktická pomůcka, protože překladač ke každému volání přidá vytvoření onoho pole, aby funkci mohl s polem zavolat.

Další možnosti přináší dědičnost. Zde máme poměrně hodně možností, protože kromě překrývání metod, kdy potomek zakryje metodu stejného jména v předkovi, do hry přicházejí i zprávy. K dispozici máme několik užitečných klíčových slov, kterým říkáme modifikátory a lze je přidat na začátek deklarace metody (před nebo za modifikátor viditelnosti) a upravit tak její chování.

- `static` – Statická metoda, patří přímo třídě a ne objektům. (To už známe.)
- `virtual` – Virtuální metoda, bude vždy volána přes zprávu poslanou na potomka. Potomek má tedy šanci tuto metodu „vyměnit“ za jinou a svou novou verzi „vnutit“ i předkovi. (Probereme dále v textu.)
- `abstract` – Abstraktní metoda. Nemá tělo, takže je to vlastně jen zpráva a uvádí se stejně jako v rozhraní (se středníkem místo kódu). Je to podobné jako virtual, ale potomek tentokrát musí metodu dodat, jinak nelze vytvářet objekty. Má-li třída abstraktní metody (svoje nebo zděděné), musí být i celá třída stejným způsobem označena jako abstraktní.
- `sealed` – zapečetěná metoda, potomek ji nemůže změnit. Zapečetit lze jen virtuální/abstraktní metodu.

Existují ještě modifikátory `extern` a `unsafe`, ty ale potřebovat nebudeme.

Je-li metoda abstraktní, pak je automaticky i virtuální. Potomek může takovou metodu předefinovat pomocí modifikátoru `override`. Další modifikátor `new` zruší označení `virtual` a nadefinuje novou nevirtuální metodu stejného jména, aniž by nahradil virtuální metodu zděděnou z předka.

Průvodce studiem

Modifikátory metod používané v souvislosti dědičností se mohou zdát jako značně matoucí, a to nejen pro začátečníky. Je-li to i váš problém, nezbývá, než „nechat věci uležet“. Všechno chce svůj čas a po dostatečném množství praxe v této oblasti vám to jistě přejde do krve.

3.2.6 Modifikátory tříd

Podobně jako u metod, i u tříd lze použít několik modifikátorů.

- `abstract` – Abstraktní třída, nelze od ní vytvářet objekty. Definuje-li třída nějaké nepřipojené zprávy, pak musí být povinně abstraktní.
- `static` – Statická třída, všechny její součásti musejí být statické a opět od ní nelze vytvářet objekty (pochopitelně, nemělo by to přece smysl).
- `sealed` – Zapečetěná třída, nelze ji dědit. Řada základních typů BCL je zapečetěná.

3.2.7 Proměnné a předávání parametrů

V třídách jsou samozřejmě i proměnné. Technicky vzato C# umožňuje nastavit jim viditelnost stejným způsobem jako u metod, pravidla OOP však zakazují, aby proměnná byla veřejná. Toto pravidlo se v praxi často porušuje, vede to obvykle ke kratšímu kódu, ale také k větší chybovosti. Můžeme použít tyto modifikátory:

- `readonly` – Nastavit hodnotu proměnné lze jen v konstruktoru, potom už bude jen ke čtení.
- `new` – Toto je nutné použít, chceme-li nadefinovat proměnnou, která se jmenuje stejně jako proměnná zděděná (je-li tamta viditelná, tj. protected či public).
- `static` – Statická proměnná, stejný význam jako u metod.

- `const` – Konstanta, není to tedy pravá proměnná, ale jen hodnota nějakého daného typu. (Zde je velký rozdíl od C++, protože tamní const se chová jako zdejší `readonly`.)

Při předávání parametrů při volání se předané parametry chovají jako lokální proměnné. Parametry lze předat buď hodnotou, nebo odkazem. Při volání hodnotou se změny provedené na proměnné nepromítnou zpět do volajícího kontextu. Hodnotou lze předa jakýkoliv výraz stejného typu jako má definovaný parametr. Toto je výchozí chování.

Předání parametru odkazem definujeme pomocí klíčového slova `ref` přidaného na začátek definice parametru (před typ). V tomto případě se změny proměnné promítnou zpět do kontextu volajícího. Odkazem nelze předat výraz, ale jen proměnnou. Speciálním případem volání odkazem je výstupní parametr, ten se definuje slovem `out` a má ten rozdíl, že můžeme předat i neinicializovanou proměnnou a platná bude až po skončení volání. Podívejme se na čtyři ukázkové metody:

```
void M1(int a) { a++; }

void M2(ref int a) { a++; }

void M3(List<int> p) { p.Add(4); p = null; }

void M4(ref List<int> p) { p.Add(5); p = null; }
```

Tyto metody vyzkoušíme takto:

```
int a = 0;
M1(a); Console.WriteLine(a); // vypíše 0, protože volání bylo hodnotou
M2(ref a); Console.WriteLine(a); // vypíše 1, protože volání bylo odkazem

int[] p = new int[] {1, 2, 3};
M3(p);
Console.WriteLine(p.Length); //vypíše 4, protože jsme do pole vložili prvek 1
M4(ref p);
Console.WriteLine(p.Length); //v proměnné p je null, takže nastane výjimka
```

Na příkladě metod `M3()` a `M4()` je vidět, že předávání odkazem se týká objektu, který předáváme, ale netyká se jeho vnitřních součástí. Proto v `M3()` máme hodnotou předanou proměnnou `p`, takže máme hodnotu odkazu na pole, zatímco v `M4()` máme tuto proměnnou předanou odkazem, takže máme odkaz na odkaz na pole.

Průvodce studiem

To, co jsme právě viděli na příkladě, je velmi důležité! Není to žádný chyták či zvláštnost C#, jen je třeba důsledně rozlišovat hodnoty proměnných,které předáváme hodnotou či odkazem, a objekty. Objekt jako parametr nikdy nepředáme, jen odkaz na objekt. (S výjimkou hodnotových struktur, které jsme se zatím neučili.)

Shrnutí

V této kapitole jsme dokončili seznámení s jazykem C#. Na rozdíl od témat předchozí kapitoly, která se týkají základních kamenů, ze kterých v C# skládáme program, zde probraná témata slouží ke skládání menších kousků kódu do větších celků. Teprve u větších programů se projeví přínos objektově orientovaného přístupu – programy budou přehlednější, méně chybové a snáze udržovatelné v reálném prostředí, které neustále přináší požadavky na dodatečné úpravy a změny stávajícího kódu. U malých programů nejsou objektově orientované prostředky obvykle přímo nutné ke zdárnému zvládnutí či vyřešení úlohy, avšak jazyk C# a celá platforma .NET jsou z principu objektové, takže programátor v prostředí .NETu pracující se bez znalosti základních objektových principů neobejde. I kdybyste totiž nechtěli sami objekty aktivně ve

svých programech používat a vytvářet si vlastní třídy či rozhraní, budete vždycky potřebovat alespoň umět použít funkcionalitu vestavěné knihovny tříd BCL.

Pojmy k zapamatování

- Objektově orientovaný přístup
- Objekt
- Třída
- Zpráva
- Metoda
- Dědičnost
- Anonymita klienta
- Viditelnost (součástí)
- Rozhraní (interface)
- Dědění tříd a implementace rozhraní
- Překrytí (či zakrytí) jména
- Přetížení metody (overloading)
- Modifikátor třídy
- Předávání parametrů – hodnotou a odkazem

Kontrolní otázky

1. *Jmenujte čtyři základní vlastnosti každého objektu.*
2. *Co je to polymorfizmus?*
3. *Co je to třída? Vysvětlete význam tohoto pojmu v souvislosti s objekty.*
4. *Vysvětlete, co znamená výrok: „Všechno je objekt.“*
5. *Popište, co je to dědičnost. Pokuste se uvést alespoň dva příklady z praxe, které by při implementaci vedly na dědičnost.*
6. *Vysvětlete pojem „anonymita klienta“.*
7. *Jaké typy viditelnosti v jazyce C# rozlišujeme? Jakých entit se viditelnost týká?*
8. *Vysvětlete rozdíl mezi pojmy „implementace rozhraní“ a „dědění třídy“. Snažte se vystihnout hlavně ten zásadní rozdíl.*
9. *Co je to přetížení (overloading)? K čemu se může hodit v praxi?*
10. *C# neumožňuje definovat a používat nepovinné parametry u metod. Jaké náhradní řešení je k dispozici?*
11. *Vysvětlete význam modifikátorů tříd (abstrakt, static, sealed).*
12. *Vysvětlete význam modifikátorů proměnných (readonly, new, static, const).*
13. *Datové položky tříd označené readonly lze jen číst, stejně jako const. Jaký je mezi nimi rozdíl?*
14. *Jakým způsobem lze z metody vracet více než jednu hodnotu?*

4 Znaky a text

Studijní cíle: Tato kapitola se věnuje práci se znaky a textem. Jsou to témata jednoduchá a nenáročná, pro řadu studentů tedy půjde jen o velmi rychlé zopakování již dříve nabytých znalostí.

Klíčová slova: znak, char, text, string

Potřebný čas: 45 minut

V předchozí kapitole jsme uzavřeli základní studium jazyka C# a nyní se už budeme věnovat jen knihovně BCL, která je součástí každé instalace .NETu a nabízí programům širokou funkcionalitu. Jde de facto o systémovou knihovnu, která chce plně nahradit operační systém a přidává i řadu dalších funkcí, které se v praxi jednoduše „mohou hodit“. Právě z důvodu obrovské rozsáhlosti této knihovny není možné se ji „naučit celou“. Každá kniha, i ty nejsilnější vícesvazkové tisíce stránek obsahující publikace vždy upozorňují, že knihovnu BCL prostě nikdy nikdo nebude umět celou. Proto pro úspěch ve vašem programování budete potřebovat jinou schopnost: Umět potřebnou věc najít, nejlépe přímo v MSDN. Na druhou stranu je pravda, že naučíte-li se vhodnou podmnožinu BCL, bude se vám daleko snáze řešit různé často se opakující situace. Na následujících stránkách se budeme věnovat několika základním prvkům, které se v programování často používají, a proto je tedy vhodné je „umět“.

4.1 Přehled

Kódování čísel je v .NETu poměrně přímočaré, protože jde o věc související přímo s architekturou mikroprocesoru a je tedy ve všech jazycích víceméně stejné. Kódování znaků a potažmo textů se však v jednotlivých prostředích liší. Základní rozdíl je v tom, zda kódování znaků vychází z historického jednobajtového přístupu, nebo novějšího dvojbajtového.

Platforma .NET používá výhradně kód unicode UTF-16, kde každý znak v paměti má 2 nebo 4 bajty, kde 4bajtové lze poznat podle zvláštního identifikátoru v prvních 2 bajtech. Začněme však od začátku…

4.2 Třída Char

Znak je reprezentován datovým typem `System.Char` nebo stručně `char` (obojí je v C# totéž, většinou se u všech typů používají ty kratší názvy psané samými malými písmeny). Je to číselný typ, hodnota je v rozsahu 0 – 65535. Char tedy reprezentuje jen 65536 znaků unicode, většinou to bohatě stačí, ale je dobré vědět, že ve standardu unicode existují i jiné znaky. Potřebujeme-li pracovat s ostatními unicode znaky, je třeba použít typ int nebo dvojici charů.

Průvodce studiem

Dvojice UTF-16 hodnot, která dohromady definuje znak s kódem nad 65535, se nazývá surrogate pár. Surrogate lze uložit kamkoliv do stringu, ale pochopitelně se nevleze do jednoho charu. Máme-li string, tak surrogate páry v něm dokážeme přesně identifikovat, protože první i druhý char z páru mají zvláštní identifikátor v několika

horních bitech. Toto však aplikační programátor nepotřebuje znát, neboť vše zařídí systém. Pro více informací viz např. [MSDN] či [Wiki].

Char je hodnotový typ, znaky tedy mají pořadí (stejně jako čísla). Jelikož jde o číselný typ, operátory se chovají jako při práci s 2bajtovými čísly a implicitní i explicitní konverze mezi charem a dalšími základními typy fungují zcela intuitivně. Stejně tak jsou k dispozici konstanty MaxValue a MinValue s hodnotami 0 a 65535.

Třída Char nabízí také řadu statických metod (funkcí) pro práci se znaky.

- `ConvertToUtf32()` – převádí surrogate pár do kódu UTF-32
- `ConvertFromUtf32()` – převádí UTF-32 znak na surrogate pár
- `GetNumericValue()` – převede číslo ve stringu na číslo (např. '1' → 1)
- `GetUnicodeCategory()` – vrací kategorii znaku (malé písmeno, číslo, atp.)
- `IsDigit()`, `IsLetter()`, `IsSymbol()`… – skupina funkcí pro zjištění, zda znak je určitého konkrétního typu
- `ToLower()`/`ToUpper()` – převede písmeno na malé/velké, zohledňuje kulturu
- `ToLowerInvariant()`/`ToUpperInvariant()` – totéž bez ohledu na kulturu

Průvodce studiem

Všimněte si, že všechny užitečné metody jsou statické. Je to proto, že char je jedním ze základních hodnotových typů a statická implementace nezatěžuje jednotlivé objektové instance obtížným smetím. Stejnou věc můžeme najít i u dalších základních datových tříd .NETu.

Přidejme ještě několik zásad pro práci se znaky:

- Znaky na čísla nepřevádíme starobylým `číslo = znak - 48`, ale funkcí `GetNumericValue()`.
- Zjišťování, zda je písmeno malé neděláme starobylým `if(znak>='a' && znak<='z')` ale pomocí příslušných funkcí. Totéž pro velká písmena a platí to i pro konverzi.
- Rovněž zjišťování typu znaku děláme pomocí příslušných funkcí.

Průvodce studiem

Nedodržování zde uvedených jednoduchých zásad je u začátečníků dost častým jevem. Stejné prohřešky však často dělají i zkušení programátoři, je to zjevně důsledek historické situace, kdy žádné užitečné funkce pro práci se znaky v systémech nebyly.

4.3 Třída String

Znakové řetězce reprezentuje v .NETu třída `System.String`, v C# má také ekvivalentní krátký název `string`. String má zvláštní postavení mezi datovými typy, není to však z technických

důvodů – jde totiž o zcela běžnou třídu. Tím důvodem je, že lidé tak často pracují s texty, že jejich podpora v systému je velmi užitečná. Funkcionalita .NETu je zde velmi bohatá, podívejme se alespoň na některé základní užitečné věci.

String je posloupnost znaků typu char. Není zde žádná ukončovací nula jako v C nebo délka v prvním bajtu jako v Pascalu. String je referenční třída, ale je immutable (neměnitelná), tzn. po založení objektu typu string již nelze jeho hodnotu měnit. (Toto omezení sice lze obejít, což ve speciálních situacích může velmi zrychlit běh programu, tímto se však nebudeme zabývat.)

K dispozici máme operátory pro porovnání (`==` a `!=`) a konkatenaci (`+`) dvou stringů, všechny fungují intuitivně. K jednotlivým znakům v C# přistupujeme pomocí indexeru (hranaté závorky). Property **Length** udává délku stringu, property `Empty` vrací prázdný string. (Vytvoříme-li string deklarativně (string s = ""), pak má referenci stejnou jako `String.Empty`.)

Jelikož je string immutable, každá operace vedoucí ke změně nemění původní hodnotu, ale vytváří nový objekt. Pomocí konstruktoru lze string vytvořit také z pole znaků nebo bajtů či pointerů na znak nebo bajt.

4.3.1 Metody

Třída String nabízí řadu metod, zde je stručný přehled:

- `Clone()` – implementuje rozhraní `ICloneable`, vrací přitom přímo this (tedy ne kopii)
- `CompareTo()` – porovnává dva řetězce ve stylu menší/větší/rovno.
- `Contains()` – testuje, zda string obsahuje daný podřetězec.
- `CopyTo()` – kopíruje řetězec nebo jeho část na dané místo do pole znaků.
- `StartsWith()`/`EndsWith()` – testuje, zda řetězec začíná/končí daným podřetězcem. Lze nastavit styl porovnávání jako v `Equals` a také libovolnou kulturu.
- `IndexOf()`/`LastIndexOf()` – hledá první/poslední výskyt daného podřetězce v řetězci.
- `IndexOfAny()`/`LastIndexOfAny()` – hledá první/poslední výskyt kteréhokoliv znaku z daného pole.
- `Insert()` – vloží daný řetězec na určené místo do tohoto řetězce.
- `Normalize()` – normalizuje řetězec, tj. převede všechny znaky na jednotnou podobu. (Normalizace je nutná k tomu, aby porovnání správně poznalo rovnost dvou řetězců, protože stejné některé znaky mají v unicode víc možný číselných kódů.)
- `IsNormalized()` – zjistí, zda je řetězec normalizován.
- `Join()` – spojí řetězce v poli dohromady a vloží mezi ně daný oddělovač.
- `PadLeft()`/`PadRight()` – upraví řetězec na danou délku doplněním o potřebný počet daných výplňových znaků.
- `Remove()` – vyjme kus řetězce (toto je opak od `Insert()`).
- `Replace()` – zamění všechny výskyty daného podřetězce jiným podřetězcem.
- `Split()` – rozdělí řetězec na části, které pozná podle daného oddělovače. (Toto je opak od `Join`.)
- `Substring()` – vrací podřetězec (od dané pozice, dané délky).
- `ToCharArray()` – převede řetězec na pole charů.
- `ToLower()`/`ToUpper()` – převede text na malá/velká písmena. Možno určit kulturu.

- `ToLowerInvariant()`/`ToUpperInvariant()` – převede text na malá/velká bez ohledu na kulturu.
- `Trim()` – odřízne všechny bílé znaky ze začátku a konce řetězce. (Bílý znak je mezera, tab a další podobné znaky.) Místo bílých znaků je možno určit vlastní seznam.
- `TrimStart()`/`TrimEnd()` – totéž jako `Trim()`, ale odřízne jen ze začátku/konce.

4.3.2 Statické metody

Třída String také nabízí několik užitečných statických metod:

- `Concat()` – konkatenuje až 4 řetězce (obdoba operátoru `+`)
- `Compare()` – porovnává dva řetězce ve stylu menší/větší/rovno. Lze nastavit porovnání s kulturou, bez kultury, číselné a to vše s nebo bez rozlišení malých/velkých písmen.
- `Copy()` – vrací pravou hlubokou kopii řetězce (na rozdíl od `Clone()`).
- `Equals()` – porovnává dva řetězce na rovnost/nerovnost. Lze nastavit s kulturou, bez kultury, číselné a to vše s nebo bez rozlišení malých/velkých písmen.
- `Format()` – formátuje více hodnot do řetězce. V prvním parametru uvedeme formát, v dalších parametrech předáme libovolný počet hodnot, které chceme společně naformátovat. Výsledkem je naformátovaný řetězec. (Toto je obdobou C funkce `printf`.)
- `IsNullOrEmpty()` – zjistí, zda je reference null nebo prázdný řetězec.

4.3.3 String jako kolekce.

Třída String je také kolekcí, takže na ni lze použít `foreach` a další operace s kolekcemi, včetně celé řady rozšiřujících metod definovaných v .NETu verze 3.5. Jedná se o opravdu velké množství rozšiřujících metod, které tu nebudeme jednotlivě zmiňovat.

4.3.4 Internace řetězců

Prostředí .NETu si vede tzv. internát – globální tabulku často používaných řetězců. Internace je tedy proces umístění řetězce do této tabulky, tj. řekneme systému, že právě tento řetězec chceme do internátu umístit. Automaticky jsou internovány všechny literály, pomocí metody Intern() lze internovat kterýkoliv další řetězec.

Smyslem a přínosem internace má být, že se v aplikacích nebudou opakovat stejné stringy, čímž se může ušetřit někdy i velké množství paměti. V praxi se však tato technologie neosvědčila, neboť internát je globální v CLR a je tedy společný všem procesům. I když váš proces skončí, internované řetězce zůstávají v systému, takže místo aby se paměti šetřilo, plýtvá se jí. Proto se nedoporučuje internaci používat a i automatické internovaní literálů lze vypnout pomocí atributu `[assembly:CompilationRelaxationsAttribute (CompilationRelaxations.NoStringInterning)]`.

4.4 Další operace s řetězci

V této sekci se stručně zastavíme u několik dalších témat, jejichž podrobný popis je však již nad rámec tohoto textu.

4.4.1 Stavění stringů – třída StringBuilder

Třída `System.Text.StringBuilder` je obdobou třídy `String`, umožňuje však měnit či připojovat další text bez vytváření nového objektu. Hlavní rozdíl mezi těmito dvěma třídami je tedy v tom, že `StringBuilder` není immutable. `StringBuilder` je určen ke „stavění řetězců“ (anglicky string builder = stavěč řetězců).

Nejčastěji se `StringBuilder` používá tak, že opakovaným voláním metody `Append()` postupně připojujeme další text na konec, až sestavíme celý text. Pak jej pomocí metody `ToString()` zkonvertujeme na klasický string. Tento způsob skládání řetězců z menších částí je mnohem rychlejší než pomocí operátoru `+` ve třídě `String`.

4.4.2 Konverze hodnoty z a na řetězec

Všechny třídy v .NETu mají povinně metodu `ToString()`, která vrací obsah objektu jako text. Je to virtuální metoda definovaná ve třídě `object` tak, že vrací jméno třídy. U tříd, kde to dává smysl, tato metoda předefinovaná tak, aby provedla konverzi hodnoty objektu na typ `String`. Pro zajímavost dodejme, že i literály jsou v .NETu objekty, takže lze napsat např. `23.ToString()` a výsledkem je text "23".

K opačné konverzi nabízí řada typů statickou metodu `Parse()`, která převede string obsahující právě jednu hodnotu na hodnotu daného typu. Například pro převod stringu na číslo použijeme statickou metodu `int.Parse("47")`. Tato metoda už není povinná pro všechny třídy.

4.4.3 Parsování stringů – třída Regex

Parsování neboli rozbor či analýzu složitějších řetězců můžeme provést pomocí třídy `System.Text.RegularExpressions.Regex`. Použití této třídy je trochu složitější, protože formát stringu musíme zadat ve formě regulárního výrazu, kde vyznačíme, které části chceme načíst. K dispozici je zde řada detailních nastavení, podrobnosti můžete nastudovat v [MSDN].

Shrnutí

V této kapitole jsme se věnovali znakům a textu, což je jeden ze základních typů informace, které ukládáme v paměti počítače. Zatímco pro popis uložení čísel v paměti stačí několik vět, popsat uložení textů nám zabralo několik stran. Nejdůležitějším faktem je, že .NET pracuje výhradně s unicode znaky, kde každý znak nemá obvyklý jeden bajt, ale dva nebo čtyři. Podpora kódování unicode je daná přímo operačním systémem Windows NT včetně řady pokročilých funkcí podporující i práci s národními abecedami. Třída String reprezentující řetězce je pak „jen“ výhodný doplněk či obalení existující funkcionality do třídy .NETu.

V záběru kapitoly jsme zmínili ještě StringBuilder a Regex, další dvě velmi užitečné třídy, i když jen velmi stručně, neboť naším cílem není zacházet do podrobností běžného programování s textem.

Pojmy k zapamatování

- Char
- Unicode
- UTF-16
- Surrogate pár
- String
- Internace [řetězce]
- Normalizace [řetězce]

- StringBuilder
- Regex

Kontrolní otázky

1. *Zjišťování , že znak je malým písmenem neděláme starobylým* `if(znak>='a' && znak<='z')`. *Vysvětlete, proč ne a jak se to má správně dělat.*
2. *Vysvětlete rozdíl mezi metodami Clone() a CopyTo() u stringu (nebo také pole).*
3. *Co je to internace? Diskutujte výhody a nevýhody této techniky.*
4. *String nelze měnit, takže při potřebě jakékoliv úpravy musíme vytvořit nový objekt. Jakou alternativu nabízí .NET pro situace, kdy potřebujeme stringy často měnit?*

5 Disky, soubory a adresáře

Studijní cíle: V této kapitole se budeme věnovat jmennému prostoru `System.IO`, který zpřístupňuje informace o discích a také se zabývá operacemi se soubory a adresáři. Naučíme se pracovat s adresáři na disku i daty v jednotlivých souborech.

Klíčová slova: disk, adresář, soubor

Potřebný čas: 110 minut

5.1 Přehled

Práce se soubory a související témata patří k nejdůležitějším věcem, které by každý programátor měl ovládat. Látka, která nás v této kapitole čeká, není složitá, ale je poměrně rozsáhlá. Na úvod si proto uveďme stručný přehled.

Potřebujeme-li zpracovat, analyzovat či nějak upravit řetězec obsahující cestu k souboru či adresáři, použijeme třídu `Path`. Je to statická třída nabízející metody pracující se syringem ve stylu „vrať jen jméno bez přípony“ či „změň příponu“. Používání této třídy může i výrazně zkrátit kód ve srovnání třeba s jazykem C, kde se se soubory pracuje velmi často, ale podobná funkcionalita v jeho základní knihovně zabudována není.

Třídy ze skupiny `FileSystemInfo` slouží k procházení či prohledávání adresářů a zjišťování informací o souborech bez toho, abychom četli či zapisovali jejich obsah.

Pro čtení či zápis dat slouží v .NETu proudy. Proud je de facto otevřený soubor. Při práci se souborem si vždy vybereme jednu z mnoha nabízených tříd, rozhodneme se podle umístění souboru (např. „soubor na disku“), formy přístupu (čtení, zápis, nebo obojí). Pro pohodlnou práci s textovými soubory, se kterými se pracuje nejčastěji, jsou zvláštní třídy `StreamReader` a `StreamWriter`.

Všechny třídy pracující s obsahem souborů implementují rozhraní `IDisposable` a po skončení práce s nimi proto musíme volat metodu `Dispose()`. (Třída zastupuje předka, proto říkáme stručně: „Tyto třídy jsou IDisposable.“) Připomeňme, že v jazyce C# je k tomu určen speciální konstrukt using, který se používá takto:

```
Using(Typ proměnná = new Typ(…)) {
  ...práce s proměnnou...
}
```

5.2 Výjimka IOException

Třídy z prostoru jmen `System.IO` při chybě vyvolávají nejčastěji výjimku `System.IO.IOException`, případně některého jejího potomka. Kód pracující se soubory byste měli vždy uzavřít do bloku `try–catch` a tuto výjimku ošetřit. Pro seznam potomků této třídy viz popis třídy v [MSDN].

Průvodce studiem

Vyvarujte se klasické začátečnické chybě a sledujte skutečně jen výjimky IOException. Ostatní výjimky mají jiný význam a pokud by se některá taková jiná vyskytla v kódu práce s diskem, mohli byste ji omylem považovat za diskovou chybu a zahodit ji bez ošetření. Diskové chyby se totiž obvykle skutečně „ošetřují" vypsáním informační hlášky uživateli a zahozením výjimky.

5.3 Třída Path – práce s cestami

Statická třída `System.IO.Path` slouží k práci s řetězci obsahujícími cesty (tj. názvy adresářů či souborů.) Je statická, protože de facto pracuje s řetězci, takže string reprezentující cestu dáváme vždy jako první parametr volané statické metody. Uveďme některé součásti:

- `DirectorySeparatorChar` – oddělovač adresářů (obvykle zpětné lomítko)
- `GetInvalidFileNameChars()` – vrací seznam znaků nepovolených pro jména
- `GetExtension()`/`ChangeExtension()` – vrací/změní příponu
- `GetFileName()` – vrací jméno bez adresáře
- `GetFileNameWithoutExtension()` – vrací jméno bez adresáře a přípony
- `GetFullPath()` – vrací celou absolutní cestu
- `GetTempPath()` – vrací temp adresář
- `IsPathRooted()` – dotaz, jestli je cesta absolutní
- `HasExtension()` – dotaz, zda má cesta příponu
- `GetRandomFileName()` – vrací náhodné jméno pro adresář nebo soubor
- `GetTempFileName()` – vytvoří temp soubor a vrací jeho jméno
- `Combine()` – spojí dvě cesty (první může být absolutní, druhá je relativní k první)

5.4 Třída DriveInfo – informace o disku

Informace o disku reprezentuje třída `System.IO.DriveInfo`. Za „disk" se zde přitom považuje písmeno ve stylu MS-DOSu, čili jde o svazek připojený jako tzv. „jednotka" s písmenem. Statická metoda `GetDrives()` vrací všechny disky ve formě pole `DriveInfo[]`. K objektům jednotlivých disků se lze dostat také přes konstruktor, kde jako parametr uvedeme písmeno disku (např. "c"). Ve třídě nejsou žádné další užitečné metody, pouze několik hodnot ve formě property. Uveďme alespoň některé:

- `AvailableFreeSpace` – volný prostor v rámci uživatelské kvóty
- `TotalFreeSpace` – celkový volný prostor na disku
- `IsReady` – dotaz, zda je disk připraven (CD v mechanice atp.)
- `TotalSize` – celková kapacita disku
- `RootDirectory` – cesta ke kořenovému adresáři (např. "c:\")

Další informace je možno nalézt v [MSDN].

5.5 Soubory a adresáře

V této sekci se budeme věnovat souborům a adresářům, neřešíme přitom obsah souborů, ale jejich jména, délku apod. Čtení a zápis souborů bude diskutován v samostatné sekci dále v textu.

Soubor je reprezentován objektem typu `System.IO.FileInfo`, adresář objektem typu `System.IO.DirectoryInfo`. Tyto dvě třídy mají společného předka `FileSystemInfo`, což jistě nikoho nepřekvapí, jelikož soubory a adresáře mají mnoho společných vlastností (jméno, datum, atributy atp.). Uvedené třídy používáme podobně jako u DriveInfo ke zjišťování informací o souborech a adresářích. Potřebujeme-li přitom jen jednu informaci, můžeme namísto vytváření objektu pro soubor/adresář využít statickou třídu System.IO.File či System.IO.Directory a dosáhnout tak stejné funkcionality bez vytváření objektů.

Průvodce studiem

Každý objekt FileSystemInfo je identifikován svou cestou a tato nesmí obsahovat mezery na začátku či konci. Proto necháváte-li zadávat cesty uživatelem, volejte vždy po zadání jakékoliv cesty String.Trim() pro odstranění přebytečných mezer.

5.5.1 Třída FileSystemInfo

Třída `System.IO.FileSystemInfo` zahrnuje společnou funkcionalitu pro adresáře a soubory. Vychází se zde z toho, že každý adresář je vlastně jistým speciálním souborem, proto má řadu vlastností se soubory společných. Objekty FileSystemInfo mohou být vytvořeny ke každé cestě bez ohledu na to, zda takový soubor existuje či nikoliv. Vybrané property této třídy:

- `Attributes` – atributy souboru ve formě flags enum (příznakový výčtový typ)
- …`Time` – sada datumů/časů vytvoření, změny, přístupu v aktuálním časovém pásmu
- …`TimeUtc` – totéž v časovém pásmu GMT
- `Exists` – true, když soubor existuje
- `Name` – jméno souboru s příponou (bez cesty k adresáři)
- `Extension` – přípona souboru
- `FullName` – celá absolutní cesta k souboru

Průvodce studiem

Časová pásma mohou mít různý vliv na různé typy disků. NTFS například ukládá všechny časy v GMT, takže při posunu na letní čas se soubory zdánlivě posunou o jednu hodinu (soubory na disku se nemění, ale vypadají jako by se jim čas změnil, protože se posunulo časové pásmo počítače). Tento zdánlivě puntičkářský detail má pozitivní vliv na náročné či citlivé aplikace, protože zaručuje že čas plyne stále kupředu.

Důležité: Podobně jako u mnoha dalších objektů, FileSystemInfo je jen obrazem skutečného stavu souboru v okamžiku vytvoření tohoto objektu. Pro aktualizaci dat v objektu je nutno volat metodu `Refresh()`.

Dále máme k dispozici metodu `Delete()`, která soubor/adresář smaže.

Soubory a adresáře mají i další společně pojmenované metody, ty však nejsou zděděny z `FileSystemInfo`. (Je to proto, že mají různé parametry a dělají trošku odlišné věci.) Metoda `Create()` založí nový soubor/adresář, metody `Get/SetAccessControl()` slouží ke správě přístupových oprávnění ACL na discích NTFS. Práce se systémem ACL je poměrně složitá a nebudeme se jí podrobněji věnovat. Metoda `MoveTo()` přesune soubor/adresář do jiného adresáře na tomtéž svazku.

5.5.2 Třída DirectoryInfo

Tato třída reprezentuje adresář. Má navíc property `Parent` a `Root` vracející nadřazený a kořenový adresář.

Metoda `CreateSubdirectory()` vytvoří podadresář, umí vytvořit i více vnořených adresářů jedním voláním. Další metody pak slouží k procházení adresářů: `GetDirectories()` vrací pole adresářů, `GetFiles()` vrací posle souborů a `GetFileSystemInfos()` vrací pole obsahující adresáře i soubory dohromady.

Průvodce studiem

NTFS disk udržuje všechny položky adresáře vždy abecedně seřazené, takže procházecí metody třídy DirectoryInfo vracejí seznamy také vždy abecedně seřazené.

Úloha: Procházení adresářů

Vyzkoušíme si procházení adresářů pomocí třídy `DirectoryInfo` či statické třídy `Directory`. Úkolem je napsat program, který zjistí počet adresářů a souborů ve stromu adresáře C:\Windows. Čili musíte rekurzivně projít celý adresářový strom počínaje tímto adresářem a průběžně počítat adresáře a soubory ve dvou proměnných.

Úlohu lze řešit několika způsoby, rozhodující se přitom jen správný výsledek. Při práci můžete narazit na dvě výjimky: `SecurityException` se vám bude objevovat, když budete program spouštět ze síťového disku. Síťový disk má standardně roli local network a programy z něj spouštěné mají v .NETu z bezpečnostních důvodů omezené některé funkce. A například procházení adresářů patří mezi ně. Program tedy musíte spouštět vždy z lokálního disku. (Toto se bude týkat i dalších programů, které budete tvořit, tak na to pamatujte!)

Druhá možná výjimka je `UnauthorizedAccessException`, tu vám budou vyhazovat adresáře, ke kterým nemáte přístup z důvodu nastavení ACL ve Windows. Je dobré si toto vyzkoušet například ve škole na učebně, protože doma se vám to zřejmě nestane. Tuto výjimku musíte vhodně ošetřit tak, aby proces procházení adresáře pokračoval dál a vynechával jen ta místa, kam právě nemáte přístup.

Průvodce studiem

S výjimkou `UnauthorizesAccessException` musíte ve všech programech počítat, nikdy totiž nevíte, jaká přístupová oprávnění k danému souboru či adresáři, se kterým chcete pracovat, máte. Nesnažte se ani předem zjišťovat, zda máte nějaké oprávnění, protože pomocí ACL lze zakázat oprávnění zjišťovat oprávnění a všechny ostatní oprávnění povolit, takže v tom případě budete moci dělat vše jen ne zjistit, jaká oprávnění máte. Správný postup je tedy skutečně oprávnění neřešit a tuto výjimku zachytávat a

vhodně ošetřovat. (Co přesně bude znamenat „vhodně ošetřovat“, samozřejmě závisí na konkrétní situaci.)

5.5.3 Třída FileInfo

Tato třída reprezentuje soubor. Má navíc několik property:

- `Directory` – vrací objekt adresáře, ve kterém je soubor
- `DirectoryName` – vrací jméno adresáře
- `FileName` – vrací jméno souboru bez adresáře
- `IsReadOnly` – vrací/nastavuje příznak read only („jen ke čtení“)
- `Length` – vrací délku souboru (je typu `long` – 64bitové číslo)

Třída má také několik užitečných metod:

- `AppendText()` – vrací písaře k zápisu dalších dat na konec souboru
- `Create()` – vytvoří soubor a vrací zapisovací proud
- `CreateText()` – vytvoří textový soubor a vrací objekt písaře
- `Encrypt()`/`Decrypt()` – zašifruje/odšifruje soubor na NTFS disku
- `Open()` – otevře soubor a vrací proud, možno nastavit řadu parametrů
- `OpenRead()` – otevře soubor a vrací čtecí proud
- `OpenText()` – otevře soubor a vrací objekt čtenáře
- `OpenWrite()` – otevře soubor a vrací objekt písaře
- `Replace()` – nahradí daný soubor tímto a vytvoří zálohu původního souboru

Zde zmíněné proudy, čtenáře a písaře si vysvětlíme v dalších sekcích této kapitoly.

Úloha: Procházení adresářů podruhé

Vytvořte ještě jednu verzi programu na procházení adresářů, která vypíše nejen počet, ale i součet délek všech souborů. Využijete k tomu program z předchozí úlohy a property `Length`.

Při procházení adresářů použijte metodu `GetFileSystemInfos()`, která vrací adresáře a soubory dohromady v jednom poli. K rozlišení adresářů od souborů použijte operátory `is` a `as`.

5.5.4 Třídy File a Directory

Jak už bylo vysvětleno výše, tyto třídy jsou statickými alternativami ke třídám odvozeným od `FileSystemInfo`. Všechny jejich metody mají tedy navíc jeden parametr typu string udávající jméno souboru/adresáře, se kterým chceme pracovat. Namísto property jsou zde metody s přidaným Get na začátek názvu. Popíšeme si zde pouze metody, ve kterých se od svých instančních protějšků odlišují.

Třída `File` má metodu `AppendAllText()`, která otevře souboru, připíše mu na konec daný string a ihned jej zavře. Obdobně metody `ReadAllBytes()`, `ReadAllLines()` a `ReadAllText()` soubor otevřou, přečtou ve formě `byte[]`, `string[]` respektive `string` a zavřou. Jedná se tedy o velmi pohodlný způsob načtení celého souboru do paměti, a to jak binárního, tak textového. Podobně zde jsou také metody `WriteAllBytes()`, `WriteAllLines()` a `WriteAllText()`, které provádějí stejným způsobem zápis.

Třída `Directory` má také metodu `CreateDirectory()`, která vytvoří nový adresář (i více adresářů do sebe vnořených). Všechny procházecí metody jsou zde upraveny tak, že namísto objektů tříd odvozených od `FileSystemInfo` vracejí jen názvy.

5.6 Proudy, čtenáři a písaři

Proudy jsou nástrojem ke čtení a zápisu dat do souborů i jiných úložišť, například stringů, polí nebo síťových soketů. Proudy pracují přímo s bajty uloženými v souborech a nijak je neinterpretují. (Můžeme však napsat odvozenou třídu, která k proudu přidá nějakou vyšší logiku.) Základem proudů je třída `System.IO.Stream`.

Samotný pojem či princip proudu je běžně používaný prvek ve většině běžných operačních systémů. V .NETu i dalších objektově orientovaných prostředích mají proudy podobu objektů. Proud je jakýsi zvláštní druh roury, kudy tečou data. Mohou téct jedním nebo i oběma směry a jeden konec této roury je obvykle v našem procesu, zatímco druhý je obvykle v souboru na disku. (Existují i jiné typy rour.) Podle toho, o jaký konkrétní typ proudu jde, tento může nebo nemusí podporovat různé operace. Například zjištění délky pomocí Length je možné u souborů na disku, ale nemusí být podporováno u jiných zdrojů dat, kterým také říkáme soubory.

Dalším typem objektů jsou čtenáři a písaři sloužící pro práci s textovými soubory. Tyto objekty nabízejí práci se soubory na vyšší úrovni (anglicky high level). Jejich třídy třídu Stream nedědí, ale agregují (obsahují) což jim dává možnost větší změny veřejného rozhraní.

5.6.1 Třída Stream

Systém proudů těží z objektového principu dědičnosti. Základní třída `Systém.IO.Stream` je na nejnižší úrovni a obsahuje operace jako „načti bajt“. Další třídy ji pak dědí a přidávají další operace jako „zjisti délku souboru“. Knihovna .NET přímo definuje 13 potomků třídy Stream plus čtenáře a písaře, kteří nedědí, ale agregují, a další třídy do tohoto systému samozřejmě můžete dle libosti přidávat i vy sami.

Průvodce studiem

Je vcelku pochopitelné, že třída Stream je `IDisposable`*, neboť pracuje se systémovými zdroji (objekty souborů v operačním systému). Je proto nezbytné volat na konci práce s proudem* `Dispose()`*; v C# je doporučeno používat blok* `using`*, který toto volání zajistí automaticky.*

Popišme si stručně nejdůležitější metody třídy `Stream`:

- `Read()` – načte několik bajtů a uloží do pole
- `ReadByte()` – načte jeden bajt
- `Write()` – zapíše několik bajtů z pole
- `WriteByte()` – zapíše jeden bajt
- `Position`/`Seek()` – zjistí/změní pozici v souboru
- `Lenght`/`SetLength()` – zjistí/změní délku souboru
- `Flush()` – vyprázdní buffery
- `Close()` – uzavře proud

- `CanRead`, `CanWrite`, `CanSeek` – pro zjištění schopností proudu

Jak vidíme, třeba metoda `Seek()` tu je definována pro všechny proudy, ale ne každá implementace ji podporuje a proto je zde `CanSeek` ke zjištění schopnosti daného objektu.

Třída Stream podporuje i asynchronní operace čtení a zápisu, tyto jsou podrobněji diskutovány například v publikaci [Kep08].

5.6.2 Třída FileStream

Třída `System.IO.FileStream` reprezentuje souborový proud. Konstruktor otevře soubor, máme k dispozici úctyhodných 15 variant, kde lze nastavit různé typy přístupu a sdílení souboru. Navíc máme k dispozici property Name vracející námi zadané jméno souboru, metody `Get/SetAccessControl()` pro správu oprávnění ACL a metody `Lock()`/`Unlock()` pro zamykání souborů před přístupem jinými procesy (i po částech).

5.6.3 Třída MemoryStream

Třída `System.IO.MemoryStream` používá jako úložiště pole bajtů. Konstruktor vytvoří buffer, který je implicitně nafukovací, lze jej však nasměrovat i na existující pole (pevné délky). K dispozici máme property `Capacity`, které vrací či nastaví velikost bufferu. Dále máme k dispozici několik metod:

- `GetBuffer()` – vrací vnitřní buffer
- `ToArray()` – vrací kopii dat v novém poli
- `WriteTo()` – zapíše obsah svého vnitřního bufferu do jiného proudu

5.6.4 Třída GZipStream

Třída `System.IO.GZipStream` umožňuje provádět kompresi a dekompresi algoritmem GZip, kterou provádí nad jiným proudem. (Tato třída tedy dědí/specializuje třídu Stream a zároveň jiný Stream agreguje.) Konstruktor vytvoří kompresní či dekompresní proud nad jiným proudem. Property `BaseStream` vrací vnitřní proud, nic dalšího zde k dispozici není.

Tato třída navíc neumí `Length`, `Position`, `Seek()`, ani `SetLength()`.

Úloha: Hledání stejných souborů

Vytvoříme ještě jednu verzi procházení adresářů, tentokrát budeme hledat soubory se stejným obsahem. Program tedy upravte tak, aby nejprve prohledal celý adresářový strom a zapamatoval si seznamy souborů se stejnou délkou. Potom soubory se stejnými délkami porovnejte, tj. načtěte je do paměti a zjistěte, jestli jsou stejné.

Při řešení této úlohy budete jako vedlejší problém muset algoritmizovat dvě věci:

1. Evidence souborů stejné délky. Potřebujete evidovat seznam souborů pro každou hodnotu délky, která se vyskytuje. Přitom souborů pro každou délku může být mnoho, nebo také žádný. Vhodné je proto použít například `SortedDictionary<Queue<string>>`, ale můžete zkusit i jiné vlastní řešení.
2. Porovnání obsahu dvou souborů. Soubory mohou být tak velké, že se nevejdou do paměti. Navíc načítat celý soubor by bylo zbytečné zdržení, neboť dva stejně dlouhé soubory obvykle stejné nejsou a stačí k tomu porovnat několik prvních bajtů. Vhodnější je proto nejprve načíst začátek souboru, porovnat jej se začátkem druhého souboru, a teprve při shodě načítat zbytek souborů po větších blocích. Nemá však smysl načítat méně než alespoň 4KB na začátku a potom alespoň řádově stovky KB v jednotlivých

blocích. Existují i jiná řešení, můžete zkusit nějaké vymyslet.
Nezapomeňte, že v rámci jedné délky je třeba porovnávat každý s každým souborem.

5.7 Čtenáři a písaři souborů

Jak už víme, čtenáři a písaři jsou speciální třídy (či přesněji objekty – instance těchto tříd), které nabízejí rozhraní na vyšší úrovni abstrakce. Jsou to třídy, které agregují Stream, takže je lze použít s libovolnými typy proudů (čili nejen se soubory na disku).

Pozor! Všechny třídy čtenářů a písařů jsou `IDisposable`.

5.7.1 Třída BinaryReader – binární čtenář

Třída `BinaryReader` slouží ke čtení binárních souborů. Proud, ze kterého má čtenář číst, uvedeme jako parametr konstruktoru. Získat ho zpět můžeme pomocí property `BaseStream`.

K dispozici máme několik desítek variant čtecích metod, všechny se jmenují `Read`... + jméno datového typu, který metoda čte, včetně variant pro čtení různých polí. K zavření souboru použijeme metodu `Close()`.

Průvodce studiem

Jak je vidět, třída `BinaryReader` *nemá zrovna mnoho užitečných metod. Například chybí možnost podívat se dopředu, jaká data následují v proudu; lze to jedině metodou* `PeekChar()`*, která však čte znak v nastaveném kódování a paradoxně tedy neumí přečíst skutečně binární bajt tak, jak je uložen v binárním souboru. Mnohé další operace lze provádět přímo na vnitřním proudu (zrovna „peek" však proud neumí).*

5.7.2 Třída BinaryWriter – binární písař

Třída `System.IO.BinaryWriter` je opakem předchozí – slouží ke zápisu binárních souborů. Objekt vytvoříme konstruktorem s uvedením proudu, kam má písař psát. Tento proud lze později zjistit pomocí property `BaseStream`.

K dispozici je opět jen několik málo metod: Metoda `Write()` nabízí 18 variant zápisů různých datových typů, její použití je naprosto intuitivní. Dále můžeme použít metody `Flush()`, `Seek()` a `Close()`. Jejich význam známe již z proudů.

5.7.3 Třída StreamReader – textový čtenář

Třída `System.IO.StreamReader` slouží jako čtenář textových proudů. Dědí abstraktní třídu `TextReader`, která má ještě druhého potomka `StringReader` sloužícího ke čtení dat ze stringu. Podrobnosti k StringReaderu si můžete najít v [MSDN], nyní se soustředíme jen na práci s proudy.

Konstruktor má mnoho variant, prvním parametrem je proud či jméno souboru, další pak umožňují nastavit kódování. Výchozí kódování je `Encoding.UTF8`, což není běžně používaný kód ve Windows. Chceme-li načítat soubory v českém Windows, uvedeme `Encoding.Default`, nebo lépe `Encoding.GetEncoding(1250)`, protože výchozím nastavení Windows nemusí být na všech počítačích právě CP1250. Nastavené kódování lze později zjistit pomocí property `CurrentEncoding`.

Průvodce studiem

Problematice kódování je třeba věnovat patřičnou pozornost. Jako ideální se jeví používat výchozí UTF8, ale většina programů z historických důvodů toto buď nepoužívá jako výchozí, nebo dokonce vůbec nepodporuje.

Řada problémů s češtinou v .NETu je spojena s mylnou domněnkou, že výchozí = Encoding.Default. Ve skutečnosti je výchozí vždy UTF8, zatímco Encoding.Default bere kódovou stránku operačního systému.

K dispozici máme několik čtecích metod: `Read()` čte jeden znak a vrací jej jako typ int, aby bylo možno načítat i surrogate páry. `Peek()` vrací příští znak bez toho, aby se posunula pozice v souboru. `ReadBlock()` čte více znaků do pole. `ReadToEnd()` čte celý soubor najednou. `ReadLine()` čte celý řádek a vrací jej bez koncových značek. Property `EndOfStream` signalizuje konec proudu.

Průvodce studiem

Textový čtenář v .NETu podporuje všechny používané kombinace značek konců řádek \n a \r, takže spolehlivě přečte i soubory z jiných operačních systémů.

5.7.4 Třída StreamWriter – textový písař

Třída `System.IO.StreamWriter` slouží jako písař textových proudů. Dědí abstraktní třídu `TextWriter`, která má ještě druhého potomka `StringWriter` sloužícího ke čtení dat ze stringu. Podrobnosti k StringWriteru si opět můžete najít v [MSDN].

Písař funguje podobně jako čtenář. Stejný je způsob použití konstruktoru, navíc zde můžeme nastavit velikost používaného vnitřního bufferu. K psaní slouží metody `Write()` a `WriteLine()`, které nabízejí mnoho variant parametrů a fungují stejně jako u známé třídy `Console`. Property `Encoding` vrací používané kódování, `BaseStream` vrací používaný proud, `NewLine` umožňuje zjistit nebo i změnit značku používanou pro ukončování řádků (standardně "\r\n") a `AutoFlush` umožňuje zjistit nebo nastavit, zda jsou data zapisována přímo do proudu, nebo skrze cache. Použití cache zrychluje zápis, ale pokud nedojde k řádnému ukončení zápisu (např. při chybě v programu), obsah cache není do proudu zapsán. Obsah cache je zapsán při volání `Flush()` a `Close()` zápis ukončí a proud zavře.

Průvodce studiem

Třídy `Stream`, `TextReader` *a* `TextWriter` *nabízejí také metodu* `Synchronized()`*, která vrací vláknově bezpečné (thread–safe) varianty těchto instancí. K tomuto tématu se vrátíme později při probírání vláken a synchronizace.*

5.8 Zabezpečení na bázi ACL

Systém zabezpečení (použitelný u souborů na discích NTFS, ale ne FAT) používá systém oprávnění v podobě ACL (access control list), ke je možno nastavit každému uživateli či skupině povolení či zakázání různých operací s objektem a také pravidla auditu. ACL je

funkcionalita operačního systému Windows NT. U souborů je ACL dobře známé, používá se však i u dalších systémových objektů. Ačkoliv práce s ACL je v .NETu od verze 2.0 díky podpoře jeho knihovny daleko jednodušší, než přímo ve Windows, jsou to věci, které programátor v praxi velmi málo využije a nebudeme se jimi proto zabývat.

5.9 Izolované uložení (isolated storage)

Jedním z důležitých rozdílů .NETu oproti systému Windows je daleko pečlivější starostlivost o bezpečnost. Podobně jak se nastavují zóny zabezpečení ve webovém prohlížeči, i .NET rozlišuje několik zón, ke kterým jsou v systému nastaveny výchozí pravidla bezpečnosti. Každý spouštěný program pak běží v některé z těchto zón (a pokud mu nastavení nezměníme), pracuje v rámci nastavených oprávnění pro tuto zónu.

Programy spouštěné v méně bezpečných zónách například nemusejí mít vůbec přístup k lokálnímu disku. Týká se to například .NETových aplikací spouštěných přímo ve webovém prohlížeči a přináší to na jedné straně vyšší bezpečnost, na druhé ale problém, že tyto programy nemají kam ukládat svá data. Platforma .NET umožňuje těmto programům používat tzv. izolované uložení (anglicky isolated storage), což je vymezená oblast disku, kde program může zapisovat (a číst) svá data. Tato oblast je fyzicky jako běžný adresář někde na pevném disku, ale nedůvěryhodný program přímo celý disk nevidí, jen svůj adresář.

Používání izolovaných uložení je pokročilé téma, podíváme se však alespoň na jeho principy. Základním prostředkem je třída `IsolatedStorageFileStream`, která dědí `FileStream`. Poskytuje tedy souborový proud, který lze používat například ve čtenářích a písařích apod. Soubor přímo vytvořený konstruktorem této třídy je v úložném prostoru platném pro jedno spuštění programu. Pomocí třídy `IsolatedStorageFile` lze získat přístup k trvalému úložnému prostoru aplikace, seskupení či aplikační domény a to buď pro konkrétního uživatele, nebo sdíleně pro všechny uživatele počítače.

Shrnutí

V této kapitole jsme se seznámili s prací se soubory v prostředí .NETu. Zmíněno bylo poměrně hodně detailů, protože tato témata patří k nejdůležitějších věcem a měla by být každému programátorovi blízká.

Statická třída Path nabízí sadu metod pro analýzu či úpravu řetězců obsahujících cestu k souboru či adresáři. Její metody řeší obvyklé scénáře jako změnu přípony souboru, zjištění adresáře apod.

K procházení či prohledávání adresářů slouží třída `DirectoryInfo` nebo také statická třída `Directory`. Příbuzné třídy `FileInfo` a `File` poskytují informace o souborech bez toho, abychom je otevírali pro čtení či zápis.

Pro práci s obsahem souborů máme k dispozici velké množství tříd kolem základní třídy Stream. Soubory před prací musíme otevřít (či založit) a otevřený soubor nazýváme proud. Platforma .NET zná i jiné než souborové proudy, my jsme se zde však věnovali jen těm souborovým. Nejčastěji používané jsou přitom třídy nazývané „čtenáři“ a „písaři“, které základní třídu Stream nedědí, ale agregují (obsahují). Rozlišujeme binární a textové čtenáře, ty textové pak mohou pracovat s proudy nebo řetězci. Obvykle používáme třídy `StreamReader` pro pohodlné čtení a `StreamWriter` pro pohodlný zápis textového souboru. Prodiskutovali jsme také problematiku kódování, což je u textových souborů velmi důležité, neboť zatímco v paměti se používá univerzální kód UTF-16, v souborech můžeme potkat řadu jiných kódů a obvykle to nelze automaticky detekovat.

V závěru kapitoly jsme se zastavili ještě u témat týkajících se zabezpečení, programování s těmito věcmi je však již nad rámec našeho studia.

Pojmy k zapamatování

- IOException
- Path
- DriveInfo
- Directory a DirectoryInfo
- File a FileInfo
- Stream, FileStream, MemoryStream, GZipStream
- Čtenáři a písaři souborů (BinaryReader, BinaryWriter, StreamReader, StreamWriter)
- ACL
- Izolované uložení

Kontrolní otázky

1. *Jaké výjimky je třeba sledovat při práci s adresáři a soubory?*
2. *Proč práci se souborem typicky uzavíráme do bloku using?*
3. *Která třída v .NETu nabízí podporu pro práci se stringy obsahujícími názvy adresářů a souborů?*
4. *Popište postup otevření souboru pro čtení. Vysvětlete rozdíl mezi objektem souboru, objektem proudu a objektem čtenáře proudu. (Snažte se vystihnout to hlavní: Proč máme tolik tříd? V čem je konkrétní výhoda oproti řešení s menším počtem tříd?)*
5. *Jmenujte jiné typy proudů (streamů), než souborové. Jak se s nimi pracuje?*
6. *Co je to izolované uložení (isolated storage)?*

Seznam obrázků

Obrázek 1. Dialog přednastavení Visual Studia 2008 7
Obrázek 2. Založení nového projektu 8
Obrázek 3. Kostra konzolové aplikace 9
Obrázek 4. Kostra okenní aplikace a otevřený editor formulářů 11
Obrázek 5. Toolbox 12
Obrázek 6. Nastavení spouštěcích parametrů programu. 35

Seznam tabulek

Tabulka 1. Základní číselné typy 25
Tabulka 2. Základní datové kolekce BCL 32

Reference

[Kep08] Aleš Keprt. *Systémové programování v jazyce C#.* Studijní text pro distanční vzdělávání, Univerzita Palackého v Olomouci, 2008.

[Kra99] Ilja Kraval. *Úvodní pojmy objektového programování.* Object Consulting, 1999.

[MSDN] Visual Studio 2005/2008 – nápověda MSDN. Totéž je i na webu na adrese http://msdn.microsoft.com/.

[Wiki] *Wikipedia, the free encyclopedia.* http://en.wikipedia.org/