

© Ing. Miroslav Virius, CSc., 2002

Vydal: Neocortex spol. s r. o.
Na Rovnosti 2245/3, 130 00 Praha 3
Tel.: 284 860 942, 284 860 882
<http://www.neo.cz/>
Tisk: EKON, družstvo, Jihlava

Windows je registrovaná obchodní známka *Microsoft Corporation*.
Obrázky pavouka a mouchy © Task Force Clip Art
Jiné případné názvy mohou být ochrannými známkami nebo registrovanými ochrannými známkami příslušných vlastníků.

ISBN 80-86330-11-7

Obsah

Úvod	8
1 Než opravdu začneme	12
1.1 Počítač	12
Operační paměť	13
1.2 Datové typy a proměnné	13
1.3 Programy a programovací jazyky	14
Operační systém	16
1.4 Program a algoritmus	17
1.5 Objekty a třídy	20
Zapouzdření	20
Dědičnost	23
Polymorfismus	25
Dědičnost versus skládání	26
1.6 Jazyk C#	27
Prostředí .NET Framework	27
Kde získat C#	28
Třídy v C#	29
C#, Java a C++	29
2 První programy	31
2.1 Nápis na obrazovce	31
Zdrojový text	31
Překlad	32
Běh programu	33
2.2 Co jsme naprogramovali	34
Komentář	34
Deklarace třídy	35
Metoda Main()	35
Doplňující poznámky	36
2.3 Ódy žlutého koně: čeština v programu	37
Druhý program	37
Programátoři, čeština a my	39
2.4 Seskupení (assembly)	39
3 Jednoduché příklady	41
3.1 Drobné úpravy programu	41
3.2 Počítač ať počítá	44
Metoda, která vypočte hodnotu	46
3.3 Základní vstup	48
Vytváříme dynamickou knihovnu	49
Použití dynamické knihovny	50
3.4 Ještě trochu počítání	51
Cyklus	51
Užitečné operátory	53

Podmínka	55
Indikace chyby	56
Výjimky	56
Magická čísla	58
3.5 Univerzální nápis	60
Třída Text	60
Vytvoření instance	62
Automatická správa paměti	65
4 Složitější příklady	67
4.1 Řazení slov	67
Třídy v programu	67
4.2 Seznam	76
Jednosměrně zřetěžený seznam	76
Prvek seznamu	76
Implementace seznamu	77
4.3 Vylepšujeme vstup	80
5 Začínáme naostro	83
5.1 Základní pojmy	84
Komentář	84
Klíčová slova	85
Identifikátor	86
Zápis programu	87
5.2 Prostory jmen	87
Deklarace prostoru jmen	87
Spojování prostorů jmen	89
Globální prostor jmen	89
using	90
5.3 Atributy	91
Některé atributy	92
6 Proměnné a datové typy	93
6.1 Proměnné	93
Typy a proměnné	93
6.2 Hodnotové typy poprvé: atomické typy	94
Celá čísla	94
Příklad: ještě jednou faktoriál	98
Znaky	98
Reálná čísla	100
Logické hodnoty	103
Desítková čísla	105
Prázdný „typ“ void	105
Příklad: Převod do jiné číselné soustavy	105
6.3 Pole: skupina proměnných stejného typu	108
Deklarace a vytvoření pole	108
Inicializace pole	109
Vícerozměrná pole	110
Nepravidelná pole	110
Práce s poli	111
6.4 Instance tříd	113

Automatická správa paměti	113
6.5 Hodnotové typy podruhé	114
Výčtové typy	114
Struktury	115
6.6 Balení a vybalování	116
6.7 Delegáty	117
Vícenásobné delegáty	119
7 Výrazy a operátory	120
7.1 Vlastnosti operátorů	120
7.2 Aritmetické výrazy	121
Smíšené výrazy	121
7.3 Relační výrazy	123
7.4 Logické výrazy	123
7.5 Přehled operátorů	123
Podmíněný výraz	124
Konverze a přetypování	124
Přířazovací operátor	125
8 Příkazy	128
8.1 Blok (složený příkaz)	128
Vnořené bloky	128
8.2 Elementární příkazy	129
Prázdný příkaz	129
Výrazový příkaz	129
Deklarace	130
8.3 Podmíněné příkazy	130
Příkaz if	130
Příkaz switch (přepínač)	131
8.4 Cykly	133
Příkaz while	134
Příkaz do-while	134
Příkaz for	136
Příkaz foreach	137
8.5 Skokové příkazy	138
Příkaz break	138
Příkaz continue	139
Příkaz goto	140
Příkaz return	141
Příkazy throw, checked a unchecked	141
9 Třídy a objekty	142
9.1 Deklarace třídy	142
Modifikátory v deklaraci třídy	142
9.2 Tělo třídy	144
9.3 Datové složky	145
Nestatické datové složky	145
Neměnitelné složky	146
Statické datové složky	147
9.4 Metody	148
Deklarace metody	148

Parametry metod	149
Nestatické metody	151
Statické metody	152
Lokální proměnné	154
Rekurze	154
Metoda Main()	155
this	157
9.5 Konstruktory a destruktory	157
Statický konstruktor	159
Destruktor	159
9.6 Vlastnosti	160
Deklarace vlastností	160
Třída Bod	161
9.7 Dědění	162
Konstruktor potomka	162
base	163
Předefinované metody a vlastnosti	163
Nepolymorfní chování předefinovaných metod	164
Polymorfní chování metod	166
Příklad: grafické objekty	166
Abstraktní metody, abstraktní třídy	172
Zapečetěné třídy, zapečetěné metody	173
9.8 Struktury a třídy	173
10 Další vlastnosti tříd	174
10.1 Rozhraní	174
Deklarace a implementace rozhraní	175
Použití rozhraní	176
Klonování objektů	177
Seznam jako (téměř) standardní kontejner	179
10.2 Přetěžování operátorů	182
Základní pravidla	182
Deklarace přetíženého operátoru	183
Nepřímé přetěžované operátory	183
Některé operátory tvoří dvojice	183
Komplexní čísla	183
Konverzní operátor	186
Indexování	188
10.3 Události	190
Delegát pro událost	190
Deklarace události	190
Vyvolání události	191
Příklad	191
11 Výjimky	193
Proč výjimky?	193
11.2 Výjimka v C#	195
Třídy pro přenos informací o výjimkách	195
Vznik výjimky	196
11.3 Ošetřování výjimek	197
Handler	197

Pošli to dál	201
Koncovka	201
11.4 Složitější příklady	203
Ještě jednou vstupy	203
Rovnost komplexních čísel	206
12 Soubory, vstupy a výstupy	207
12.1 Soubor a adresář	208
Způsob práce se souborem	209
Metody třídy File	209
Třída Directory	209
12.2 Čtení a zápis binárních dat	210
Čtení a zápis atomických typů	211
Ukládání objektů (serializace)	213
12.3 Čtení a zápis znakových dat	215
Zápis do textového souboru	216
Čtení z textového souboru	217
Čeština a kódování	218
Formátování	220
13 Grafické uživatelské rozhraní	222
13.1 První okenní program	222
Knihovna Windows Forms	223
Vlastností komponent	225
Třída okna	226
Prázdné okno ještě jednou	226
13.2 Používáme komponenty	227
Přidáváme komponentu do okna	228
Předefinované události	229
Učíme tlačítko fungovat	231
13.3 Příklad: problém N dam	232
Jak najít řešení	232
Program	233
Třída Řešitel	234
Třída okna	237
Dialogová okna se zprávami	241
Handlery	242
Dialogové okno O programu	246
Literatura	249
Rejstřík	250

Úvod

V této knize se seznámíte se základy programovacího jazyka C#, který je žhavou novinkou počátku třetího tisíciletí. Kniha je určena naprosto všem začátečníkům, a proto nepředpokládám, že byste měli jakékoli předběžné znalosti o programování.

Jediné, co očekávám, je, že máte chuť se to naučit – a samozřejmě, že počítač není vaším nepřítelem. Abych byl přesnější: předpokládám, že víte, co je soubor, a že umíte spouštět programy z grafického rozhraní Windows a z příkazové řádky.

Proč právě C#?

Proč byste se měli učit právě C#? Jde o nový jazyk, právě uvedený na trh, a kdo vám zaručí, že během krátké doby nezapadne?

Upřímně řečeno, zaručit vám to nemůže nikdo. Přesto si myslím, že tento jazyk má před sebou slibnou budoucnost, a mám pro to dobré důvody. Posuďte sami:

- Tento jazyk vytvořila a šíří velká počítačová firma – Microsoft. To samo sice není zárukou úspěšnosti, ale je to dobrý předpoklad k ní.
- Jde o jazyk určený pro tvorbu programů pro MS Windows, nejrozšířenější prostředí na PC.
- Tým, který jazyk C# navrhl, vedl Andres Hejsberg, autor Turbo Pascalu a vedoucí týmu, který vyvinul Borland Delphi – dva velice úspěšné programovací prostředky.
- Návrh jazyka C# vychází ze zkušeností s jazykem C++, který byl nejpopulárnějším programovacím jazykem v devadesátých letech minulého století, a s jazykem Java, který se těší velké popularitě v současnosti.



Ostatně k podobnému závěru dospěli i jiní, a proto (mimo firmu Microsoft) vznikl projekt jazyka C# a celého prostředí .NET, který je open source; v současné době je k dispozici verze označená 0.9. Podrobnější informace o tomto projektu lze najít na webové adrese <http://www.go-mono.com>.

Co v této knize najdete

Začneme tím, že se seznámíme s několika základními pojmy, které by měl každý programátor znát. Povíme si něco o počítači a jeho součástech, o algoritmech a samozřejmě také o třídách a objektech. C# je totiž tzv. objektově orientovaný jazyk, a proto se objektům prostě nelze vyhnout. Nakonec si řekneme i několik slov o historii C#.

V následujících dvou kapitolách si na několika jednoduchých programech ukážeme, jak vypadá program v C#, jak vypadají nejběžnější programové konstrukce, jak se program překládá a spouští. Tím si vytvoříme rámec, který nám umožní v následujících kapitolách ukazovat smysluplné příklady.

Pak začne výklad jazyka „naostro“: seznámíme se postupně s datovými typy, příkazy, vytvářením a používáním objektových typů atd.

Co v této knize nenajdete

Především mějte prosím na paměti, že jde o knihu pro zelenáče. To znamená, že nemůže obsahovat vše, co lze o jazyku C# napsat. O řadě témat jsem se nezmínil, protože na ně nezbylo místo, o jiných proto, že mi připadají pro začátečníka zbytečně komplikovaná (a přitom bez jejich znalosti lze také dobře programovat).

Jde např. o programování vícevláknových aplikací, o práci s atributy pomocí mechanismu reflexe, o programování webových služeb a jejich klientů atd.

Příkazová řádka

Ve většině knihy budu předpokládat, že máte k dispozici nástroje spouštěné z příkazové řádky. Možná vám to bude připadat podivné – C# je přece součástí Visual Studio .NET firmy Microsoft, které obsahuje řadu vizuálních nástrojů pro automatické generování kódu. Já se jím ale nebudu v této knize příliš zabývat, a to z několika důvodů:

- Popis Visual Studio .NET vydá na samostatnou knihu, a nijak tenkou. To ale není cílem této učebnice; ostatně většinu potřebných informací o tomto nástroji lze zjistit z dokumentace, ať už tištěné nebo elektronické.
- Visual Studio .NET v určitých situacích automaticky generuje zdrojový kód programů. To je skvělé při rutinní práci, pro výuku to ale není to nejlepší. Budete-li na počátku vše psát sami znak po znaku, pochopíte daleko snáze, co váš program dělá – a to je hlavní cíl této knihy.

Navíc základní nástroje pro vývoj programů v C# jsou k dispozici zdarma – a pokud to nepřekazí nějaké soudní procesy, tak zdarma zůstanou; v první kapitole se dočtete, kde a jak je můžete získat.

Pokud znáte mou knihu Java pro zelenáče, jež vyšla v nakladatelství Neocortex v roce 2001, zjistíte, že některé části této knihy se jí nápadně podobají. I já jsem si toho všiml. Důvodem je především skutečnost, že některé rysy C# jsou nápadně podobné odpovídajícím rysům Javy, a proto i výklad o nich musí být podobný.

Typografické konvence

identifikátor

Kurziva znamená zdůraznění. Píšeme tak také nově zaváděné termíny. V popisech syntaxe jazyka C# označuje pojmy, které je třeba ještě dále vyjasnit (tzv. neterminální symboly).

while

Tučně píšeme v popisu syntaxe symboly, které lze přímo opsat do programu (tzv. terminální symboly).

ArrayList

Neproporcionálním písmem píšeme ukázky programů, výstupy z počítače a také klíčová slova, identifikátory ap.

✧

Tento symbol označuje konec příkladu.



Ručička upozorňuje na místa, která si zaslouží zvýšenou pozornost.



Pavouk upozorňuje na informace, které lze najít na internetu. Zpravidla jde o zdrojové soubory příkladů z této knihy.



Tato ikona upozorňuje na poznámky určené především čtenářům, kteří znají programovací jazyk C nebo C++. Pokud tento jazyk neznáte, můžete je ignorovat.



Tato ikona upozorňuje na poznámky určené především čtenářům, kteří znají programovací jazyk Java. Pokud tento jazyk neznáte, můžete je ignorovat.



Moucha upozorňuje na možné programátorské chyby a na obtížná a problematická místa.

Zdrojové texty příkladů

Úplné zdrojové texty všech příkladů z této knihy si můžete stáhnout z internetové adresy www.neo.cz ze stránky věnované této knize. Jsou komprimovány programem PKZIP, a to včetně adresářové struktury. V textu knihy se na ně odolávám slovy „na WWW“; pokud v tomto odkazu uvádím adresář, jde o adresář obsažený v komprimovaném souboru.

Na závěr

Je mou milou povinností poděkovat všem, kteří svými radami a připomínkami přispěli ke zdárnému dokončení této knihy, a především lektorovi Ing. Pavlu Heroutovi, PhD., ze Západočeské univerzity v Plzni za řadu cenných připomínek.

Jsem si jist, že přes veškerou péči, kterou jsem této knize věnoval, v ní budou chyby. Za ně samozřejmě nesu veškerou odpovědnost já sám. Pokud na nějakou přijdete, pošlete mi prosím upozornění na níže uvedenou adresu; na internetových stránkách

<http://kmdc.fjfi.cvut.cz/~virijs/errata.htm>

<http://tjn.fjfi.cvut.cz/~virijs/errata.htm>

uveřejním opravu.

Miroslav Virius

katedra softwarového inženýrství
FJFI ČVUT

virijs@km1.fjfi.cvut.cz

1 Než opravdu začneme

Než začneme programovat, měli bychom se seznámit s pojmy, které budeme později potřebovat. Povíme si krátce něco o nejen o počítačích a programovacích jazycích, ale také o algoritmech a objektově orientovaném programování. Mnohé z toho jistě znáte; přesto vám doporučuji tuto kapitolu alespoň zběžně přečíst, abychom si sjednotili terminologii.

1.1 Počítač

O počítačích se často hovoří jako o „strojích na zpracování informací“. S tím vystačí lidé, které počítače vlastně nezajímají, ale chtějí vypadat informovaně. Ovšem programátor, a to i ten nejzelenější zelenáč, musí o počítačích vědět přece jen o trochu víc.

S trochou zjednodušení můžeme říci, že běžný počítač se skládá ze čtyř základních částí:

- Součást, která opravdu „počítá“, tedy zpracovává informace, a také řídí činnost všech ostatních částí počítače, se nazývá *procesor*.
- *Operační paměť* slouží k přechodnému ukládání dat (informací), která počítač zpracovává, a programu, tedy příkazů, které určují, co má dělat. Používá se pro ni také anglická zkratka RAM (Random Access Memory, doslova „paměť s náhodným přístupem“). Vše, co je v této paměti, se při vypnutí počítače ztratí, „zapomene“.
- *Vstupní a výstupní zařízení* slouží počítači k výměně informací s okolím. Nejběžnější vstupní zařízení jsou klávesnice, myš, skener atd. Nejobvyklejší výstupní zařízení osobních počítačů je obrazovka monitoru nebo tiskárna. Pro vstupní a výstupní zařízení se používá zkratka V/V nebo I/O (z anglického input/output).
- *Vnější (trvalá) paměť* slouží k trvalému ukládání dat a programů; data v ní se při vypnutí počítače neztrácejí. Má zpravidla mnohonásobně větší kapacitu než operační paměť, ale práce s ní je také mnohonásobně pomalejší než práce s operační pamětí (přibližně tisíckrát). Jako vnější paměť se používají převážně magnetické disky (pevný disk, disketa). Dnes se často používá i kompaktní disk (CD, DVD).

Operační paměť

Bity Operační paměti je tvořena elektronickými obvody, které mohou mít dva dobře rozlišitelné stavy – např. vypnuto nebo zapnuto. Jeden z těchto stavů obvykle odpovídá číslici 0, druhý číslici 1. Jakékoli údaje do paměti proto můžeme zapisovat jen pomocí nul a jedniček, v tzv. dvojkové soustavě. Každé místo, na které můžeme zapsat jednu číslici 0 nebo 1, označujeme jako *bit*. Operační paměť je tedy dlouhá řada bitů.

Bajty S jednotlivými bity však obvykle nepracujeme, neboť je to nepohodlné. Proto se bity v operační paměti sdružují do větších celků. V dnešních počítačích se zpravidla používají skupiny velikosti 8 bitů, které se nazývají *bajty* (anglicky byte, tj. slabika).

Adresa Aby mohl počítač s pamětí snadno pracovat, jsou jednotlivé bajty, které paměť tvoří, očíslovány. Počáteční bajt má číslo 0, následující má číslo 1 atd. Toto pořadové číslo se nazývá *adresa* bajtu. (Na některých počítačích, včetně PC, je záležitost s adresami trochu složitější, ale to můžeme v souvislosti s jazykem C# ponechat stranou.)

1.2 Datové typy a proměnné

Není těžké zjistit, že nejmenší číslo, které může jeden bajt obsahovat, se skládá z osmi nul a představuje i v desítkové soustavě nulu. Největší takové číslo se bude skládat z osmi jedniček a v desítkové soustavě představuje 255. To je žalostně málo; v běžných programech se obvykle objevují daleko větší čísla. Proto se pro ukládání dat obvykle používají různě velké skupiny za sebou následujících bajtů.

Ani to ovšem nestačí. Vezmeme-li např. dva za sebou následující bajty, můžeme do nich uložit celá čísla v rozmezí od 0 do 65 535 (od 0 do 1111 1111 1111 1111 ve dvojkové soustavě). Pokud nám ani to nevystačí, můžeme vzít skupinu 4 nebo třeba 8 bajtů. Jenže ani to neřeší problém, co dělat, když budeme potřebovat záporná čísla, reálná čísla, znaky nebo třeba logické hodnoty (jež vyjadřují, že nějaké tvrzení platí nebo neplatí).

Musíme tedy najít nějaký způsob, který nám umožní reprezentovat data různých „druhů“ v paměti počítače. Jinými slovy, musíme najít způsob, jakým určité hodnotě jednoznačně přiřadíme skupinu bitů, která bude tuto hodnotu představovat – jak tuto hodnotu v počítači *zakódovat*.

Datový typ Můžeme se např. dohodnout, že bajt s hodnotou 01000001 bude představovat znak 'A'. Táž skupina bitů může za jiných okolností také představovat celé číslo, které má v desítkové soustavě hodnotu 65.

Bajt se stejnou hodnotou ale může být i součástí většího celku s na-
prsto jiným významem.

Odtud je vidět, že počítači nestačí znát adresu bajtu nebo bajtů, s ni-
miž pracuje. Vedle adresy musí vědět, jak velký úsek – kolik bajtů –
má vzít a jak má jeho obsah interpretovat. Jinými slovy, musí znát
datový typ hodnoty, která je tam uložena – musí vědět, zda jde o celé
číslo, znak, logickou hodnotu atd.

Datový typ také určuje operace, které lze s danou hodnotou provádět.
Celá čísla lze například sčítat a odečítat, znaky lze spojovat do řetěz-
ců, tedy do souvislého textu.

Proměnná Až dosud jsme se hodnotami, uloženými v paměti, zabývali z hlediska
počítače; nyní se na ně podívejme z hlediska programátora. Hodnotu,
s níž budeme chtít ve svém programu pracovat, potřebujeme uložit do
paměti: Musíme si vyhradit místo a říci, jakého typu budou údaje,
které bude obsahovat. Takovéto místo pro ukládání hodnoty budeme
nazývat *proměnná*.

Počítač bude s proměnnou zacházet pomocí její adresy (pořadového
čísla jejího počátečního bajtu). Pro programátora by ale práce s adre-
sou byla nepohodlná, a proto ji pojmenujeme, dáme jí *identifikátor*.
Tomu se v programování říká *deklarace* proměnné.¹

1.3 Programy a programovací jazyky

Budeme-li od počítače chtít, aby zpracovával data, která mu předlo-
žíme, musíme mu také říci, co s nimi má vlastně dělat – musíme mu
dát *program*. Program, podobně jako data, uložíme do operační pa-
měti.

Procesor umí s daty provádět různé jednoduché operace. Umí např.
sečíst, odečíst, vynásobit nebo vydělit dvě čísla, zadáme-li mu jejich
adresy, umí vzít znak a zobrazit ho na displeji atd. Protože však do
jeho paměti nelze uložit nic jiného než čísla, musí být tyto příkazy
vyjádřeny – zakódovány – také čísly. Číselné vyjádření instrukcí
(příkazů) pro procesor se nazývá *strojový kód* a je to jediná věc, které
procesor rozumí.

Aby nebyl život příliš jednoduchý, používají různé procesory různé
strojové kódy, takže programy ve strojovém kódu nejsou obvykle
přenositelné mezi počítači s různými procesory.

¹ Později uvidíme, že identifikátor – tedy jméno – mohou mít i jiné části
programu, nejen proměnné.

Vyšší
programovací
jazyky

To ale není na programování ve strojovém kódu to nejhorší. Je asi
jasné, že programování ve strojovém kódu je velice namáhavé a ne-
přehledné. Také to téměř nikdo nedělá; místo toho se používají tzv.
vyšší programovací jazyky, jako je Ada, Basic, C, C++, ... , a také C#.

Program ve vyšším programovacím jazyku obsahuje popis řešení
úlohy vyjádřený pomocí vybraných anglických slov a případně pomo-
cí matematických výrazů zapsaných podobně jako v matematice.
Napsat program ve vyšším programovacím jazyce je pochopitelně
daleko snazší než napsat odpovídající program ve strojovém kódu.

Ovšem nic není zadarmo: Program ve vyšším programovacím jazyku
nelze na počítači přímo spustit, neboť počítač mu nerozumí. Takovýto
program se proto musí buď *přeložit* do strojového kódu nebo *inter-
pretovat*. V obou případech k tomu potřebujeme další program (nebo
skupinu programů), které to za nás udělají.

Překlad Textový soubor, který obsahuje zápis programu ve vyšším programo-
vacím jazyku, se zpravidla nazývá *zdrojový kód* nebo *zdrojový pro-
gram*, v programátorské hantýrce „zdroják“. K překladu do strojového
kódu (*kompilaci*) slouží program zvaný *překladač* neboli *kompilátor*.
Často s ním spolupracuje ještě *sestavovací program* neboli *linker*,
který dokáže spojit několik nezávisle přeložených částí programu do
jednoho celku. Linker také připojí knihovny – části programu, které
už někdo naprogramoval předem a které můžeme už jen používat.

Překladem (a případně následným sestavením) programu vznikne sou-
bor obsahující strojový kód, který již lze na cílovém počítači spustit.
Mezi typické překládané programovací jazyky patří např. C, C++
nebo Pascal.

Interpretace Místo překladu však můžeme v některých případech použít speciální
program, který bude číst zdrojový text a interpretovat ho, tj. provádět
příkazy, které v něm najde. Typickým interpretovaným jazykem je
klasický Basic².

Interpretované programy obvykle běží podstatně pomaleji než překlá-
dané programy. Vedle toho musíme na cílový počítač spolu s naším
programem instalovat také interpretační program.

C# C# je tak trochu zvláštní případ. Zdrojový kód programu napsaného
v tomto jazyce se přeloží, ovšem nikoli do strojového kódu počítače,
ale do univerzálního pomocného jazyka označovaného *Microsoft*

² To se netýká Visual Basicu; jeho nedávné verze se překládaly do strojového
kódu a s poslední verzí, Visual Basicem .NET, se zachází stejně jako s C#, o
kterém si povíme dále.

*Intermediate Language*³ (MSIL nebo jen IL). Ten se pomocí dalšího překladače převede do strojového kódu cílového počítače – ale zpravidla až v okamžiku, kdy program spustíme.

To znamená, že na cílovém počítači musí být překladač, který překládá z IL do strojového kódu. Tento počítač se nazývá *JIT* (z anglického *Just In Time*, neboť překládá *právě v době*, kdy je to potřeba). Věc je ale ještě složitější: Na cílovém počítači musí instalováno prostředí *.NET Framework*, které kromě překladačů JIT obsahuje i další součásti potřebné pro běh programů vytvořených v jazyce C#.

Na závěr ještě doplníme, že překladač libovolného programovacího jazyka (i C#) zároveň kontroluje syntaktickou správnost programu – tedy zda je program napsán podle jistých formálních pravidel, která zaručují, že mu počítač porozumí. (Syntaktická správnost programu bohužel nezaručuje věcnou správnost programu, tj. nezaručuje, že program bude dělat to, co si přejeme.)

Operační systém

Počítač bez programového vybavení by nám nebyl mnoho platný. Proto se s ním zpravidla dodává alespoň jeden základní program, který se rozeběhne automaticky hned po spuštění počítače a běží po celou dobu jeho provozu. Tento program se nazývá *operační systém* a „oživuje“ počítač, tj. přijímá pokyny uživatele, stará se o jejich provedení a informuje uživatele o výsledcích. Pokyny mohou být vyjádřeny příkazem zapsaným v příkazové řádce, kliknutím myši na ikoně nebo jiným způsobem.

Operační systém má ovšem ještě řadu dalších úloh, z nichž pro nás nejdůležitější je, že poskytuje služby dalším programům. Stará se o jejich spouštění, o přidělování paměti, poskytuje nástroje pro práci se soubory atd. Když např. chceme v programu otevřít soubor, program předá odpovídající požadavek operačnímu systému a ten se postará o vše potřebné.

Mezi nejznámější operační systémy na osobních počítačích patří různé verze Windows a Linuxu.

Programy, vytvořené v jazyce C#, mohou (zatím) běžet pouze na počítačích vybavených operačním systémem Windows NT verze 4, Windows 2000 nebo Windows XP, na nichž je instalován *.NET Framework*.

³ V poslední době se také používá označení *Common IL* (CIL, společný mezijazyk), neboť tento jazyk a některé další součásti prostředí *.NET Framework* byly přijaty mezinárodní organizací ECMA jako standard.

1.4 Program a algoritmus

Jazyk C#, podobně jako ostatní programovací jazyky, slouží k zápisu programu. Už víme, že program je nějaký soubor instrukcí (příkazů), které počítači říkájí, co má dělat.

Každý program představuje návod k řešení nějakého problému. Při programování si ale musíme uvědomit, že počítač neumí najít způsob, jak problém vyřešit: To musíme udělat my. Když už známe cestu, jak najít řešení, můžeme to naučit také počítač. *Počítač umí jen to, co ho my – nebo někdo jiný – naučí, na co má program.*

Algoritmus

Návod, který chceme předložit počítači, musí mít určité vlastnosti; některé z nich mohou vypadat jako samozřejmé, ale přesto je zde uvedeme.

- Návod musí vést k požadovanému výsledku. I když to možná vypadá směšně, není to vždy jednoduchá záležitost. Musíme např. uvážit, *za jakých předpokladů* určitý postup vede ke správnému výsledku a zda budou tyto předpoklady splněny pro data, s nimiž bude náš návod pracovat.
- Musí se skládat z kroků, kterým počítač rozumí – tzv. elementárních kroků. (To pro nás znamená, že ho musíme umět zapsat v některém programovacím jazyce, nejlépe v C#.)
- Těchto kroků nesmí být nekonečně mnoho. Nejde o to, že bychom mohli napsat program, který je nekonečně velký; to se vám asi nepodaří. Není ale nic těžkého napsat krátký program, který nikdy neskončí, protože se v něm bude do nekonečna opakovat určitá skupina instrukcí.

Návod, který tyto podmínky splňuje, se obvykle označuje jako *algoritmus*. Na program se můžeme dívat jako na zápis algoritmu v programovacím jazyce.

Metoda shora dolů

Než začneme programovat, musíme tedy znát způsob, jak daný problém vyřešit. Pak musíme postup řešení zapsat jako algoritmus⁴, tedy jako posloupnost elementárních kroků. Přitom se používá obvykle tzv. *metoda shora dolů*: Návod se rozkládá na menší a menší úseky, až dospějeme k elementárním krokům. Přitom se samozřejmě může stát, že budeme muset předchozí rozdělení upravit, některé kroky spojit, některé kroky přidat ap.

⁴ Algoritmům a metodám jejich návrhu je věnována řada knih; většina z nich ale předpokládá znalost určitého konkrétního programovacího jazyka, nejčastěji Pascalu (např. [5]). Asi nejznámější jsou knihy D. Knutha [4], které znalost žádného konkrétního programovacího jazyka nepředpokládají.

Příklad Metodu shora dolů si ukážeme na následujícím příkladu: Máme skupinu N číselných proměnných, které označíme $A[0], A[1], \dots, A[N-1]$. Takovou skupinu proměnných stejného typu, se kterými můžeme zacházet jako s jedním celkem, nazýváme *pole*. Zde tedy máme pole A s prvky $A[0], A[1], \dots, A[N-1]$. Čísla, označující jednotlivé prvky pole, se nazývají indexy.

Naším úkolem je uspořádat pole A tak, aby platila podmínka

$$A[0] \leq A[1] \leq \dots \leq A[N-1]. \quad (1)$$

Třídění Úloha „zpréházet“ hodnoty, uložené v prvcích pole, aby splňovaly podmínku (1), se nazývá *třídění*⁵ pole. Naším úkolem tedy je seřadit pole A .

Náš první pokus o napsání algoritmu třídění⁶ by mohl vypadat takto:

1. Najdi v poli A prvek, který obsahuje nejmenší hodnotu, a vyměň jeho obsah s prvkem, který je na prvním místě (na místě s indexem 0).
2. Najdi v poli A prvek, který obsahuje druhou nejmenší hodnotu, a vyměň ho s prvkem, který je na druhém místě.
3. ... atd., dokud není pole seříděné.

Je zřejmé, že tento postup povede k cíli; pro naše účely se ale příliš nehodí, neboť program nemůže obsahovat nějaké „... atd.“. Musíme proto najít lepší formulaci.

Klíčem k úspěchu bude následující úvaha: Poté, co v prvním kroku vyhledáme nejmenší prvek z celého pole a dáme ho na první místo, je nejmenší hodnota již na svém místě a nemusíme se o ni starat. Stačí tedy seřadit zbytek pole. (Známý princip *rozděl a panuj* se neuplatňuje jen v politice.)

Ve druhém kroku nám postačí najít nejmenší prvek ze zbylého úseku pole $A[1], \dots, A[N-1]$ a ten dát na druhé místo, ve třetím kroku najít nejmenší hodnotu mezi $A[2], \dots, A[N-1]$ atd. V posledním kroku budeme hledat nejmenší z prvků $A[N-2], A[N-1]$. Pak bude pole seříděné, tj. bude vyhovovat podmínce (1).

⁵ Ve skutečnosti jde o řazení prvků podle velikosti, nikoli o rozdělování do tříd. V české programátorské hantýrce se ale zpravidla mluví o *třídění*, a proto u tohoto termínu zůstaneme. Ostatně anglický termín *sorting* znamená také třídění, nikoli řazení.

⁶ Popsaná metoda se nazývá třídění výběrem (selection sort) a je vhodná pro menší pole; pro rozsáhlá pole se zpravidla používá tzv. rychlé třídění (quicksort). To je velmi efektivní postup, jeho výklad ale přesahuje rámec naší knihy. Podrobnější informace viz např. ve třetím dílu knihy [4] nebo v [5].

Všimněte si, že vlastně stále opakujeme podobnou akci: Vyhledáme nejmenší prvek v úseku pole a dáme ho na první místo v tomto úseku. Přitom v každém následujícím kroku pracujeme se stále menším úsekem pole. Zkoumaný úsek pole začíná postupně prvkem s indexem 0, 1, ..., $N-2$.

Zapišme výsledek předešlých úvah jako posloupnost kroků, které bychom mohli považovat za elementární. Při výpočtu budeme používat pomocnou proměnnou i , která nám poslouží jako „počítadlo“.

Lepší formulace

1. Do proměnné i vlož hodnotu 0.
2. Mezi prvky $A[i], \dots, A[N-1]$ najdi ten, který obsahuje nejmenší hodnotu.
3. Vyměň tuto hodnotu s hodnotou prvku, který je na i -tém místě.
4. Zvětši hodnotu proměnné i o 1.
5. Je-li $i < N-1$, pokračuj krokem 2, jinak skonči.

Kroky 1, 4 a 5 můžeme považovat za elementární, kroky 2 a 3 je třeba zpřesnit.⁷ Podívejme se nejprve na krok 2. Budeme v něm potřebovat dvě pomocné proměnné. V jedné, kterou pojmenujeme min , si budeme pamatovat index nejmenšího zatím nalezeného prvku, a druhá, k , nám poslouží jako „počítadlo“ při procházení zkoumaného úseku pole. Rozepsaný krok 2 bude vypadat takto:

2. Mezi prvky $A[i], \dots, A[N-1]$ najdi nejmenší.
 - 2a. Do obou pomocných proměnných, k a min , ulož hodnotu i . (To je index počátečního prvku zkoumaného úseku. Zatím jsme prozkoumali jen jeden prvek a tak ho pokládáme za nejmenší.)
 - 2b. Zvětši k o jedničku. (Přejdi na další prvek v poli.)
 - 2c. Je-li $k = N$, přejdi na krok 3. (Vyčerpali jsme všechny prvky v úseku.)
 - 2d. Je-li $A[k] \leq A[min]$, ulož do min hodnotu k . (Našli jsme menší prvek, zapamatuj si jeho index.)
 - 2e. Přejdi na bod 2b.

Po skončení kroku 2 bude proměnná min obsahovat index nejmenšího prvku v prohledávaném úseku.

Podívejme se ještě, jak bude vypadat rozepsaný bod 3:

3. Vyměň prvek $A[min]$ s prvkem $A[i]$.
 - 3a. Do pomocné proměnné x ulož hodnotu $A[min]$.
 - 3b. Do $A[min]$ přesuň hodnotu z $A[i]$.
 - 3c. Do $A[i]$ přesuň hodnotu x .

⁷ To prozatím berte jako fakt; z vašich dosavadních znalostí to nevyplývá, protože ještě nevíte, jak vypadají příkazy jazyka C#.

Až se naučíme C#, budeme vědět, že tyto kroky již můžeme pokládat za elementární. ✧

Metoda zdola nahoru Vedle metody shora dolů se občas se také hovoří o tzv. *metodě zdola nahoru*. To znamená, že naučíme svůj počítač nové elementární kroky, které poskládáme z kroků, které počítač už umí. Jestliže například často třídíme pole, naprogramujeme si výše uvedený algoritmus jednou pro vždy a později ho budeme už jen používat.

Knihovna Součástí každého programovacího jazyka jsou *knihovny*, které obsahují řadu běžných algoritmů již naprogramovaných tak, že je můžeme ve svých programech používat jako elementární kroky.

1.5 Objekty a třídy

Jazyk C# je čistě objektový, a proto než se do něj pustíme, musíme si říci, co se za tím skrývá. V tomto oddílu si vysvětlíme základní principy objektově orientovaného programování. Až se trochu seznámíme s jazykem C#, budeme si moci povědět více.

Zapouzdření

Zatím jsme uvažovali o programování a programech převážně z hlediska počítače, hardwaru. Od zobrazování dat v paměti jsme došli k datovým typům, od strojového kódu k programovacím jazykům. Pokusme se nyní na programování podívat z hlediska programátora, který stojí před úkolem něco vyřešit – ať už jde o program pro vedení účetnictví malé či velké firmy, počítačovou hru nebo systém řídicí vstřikování paliva v automobilovém motoru. Program musí v každém případě odrážet vybrané stránky řešené úlohy, musí *být jejím počítačovým modelem*.

Proč objekty? Je asi jasné, že programátorovi se bude lépe pracovat, bude-li moci uvažovat v termínech řešeného problému, než když bude muset uvažovat v termínech datových typů, které jsou jednou pro vždy dány jako součást programovacího jazyka. Jinými slovy, bylo by vhodné, aby si programátor mohl definovat své vlastní datové typy, které by mohl použít k programovému popisu řešené úlohy. V OOP se jim říká *objektové typy* neboli *třídy*, neboť popisují třídy objektů, které se vyskytují v řešeném problému.

Jako příklad si představme, že programujeme grafický editor. Jeho uživatel si v něm bude chtít kreslit obrázky složené ze základních geometrických tvarů – bodů, úseček, kružnic ap.

Programátor bude muset ve svém programu tyto grafické objekty nějak reprezentovat. Bude si muset vytvořit programový popis pro

Bod jako objekt

bod, úsečku atd. a bude muset naprogramovat také operace, které lze s grafickými objekty dělat.

Co to znamená? Abychom si to ujasnili, zamysleme se např. nad bodem. Z geometrie si nepochybně pamatujete, že bod v rovině (v našem případě na obrazovce monitoru) je určen souřadnicemi – dvojicí reálných čísel. Vedle toho ovšem potřebujeme určit také jeho barvu; tu můžeme v počítači vyjádřit jedním celým číslem. Z toho plyne, že k reprezentaci bodu stačí skupina tří čísel; s touto skupinou budeme chtít často zacházet jako s celkem.

Abychom mohli s bodem snadno pracovat, musíme popsat (naprogramovat) také operace, které s ním chceme dělat. Musíme ho umět vytvořit, zrušit, zobrazit, přemístit, zjistit jeho barvu, změnit jeho barvu, zjistit jeho polohu, změnit jeho polohu atd.

Třída

V programu budeme určitě potřebovat větší množství bodů. Abychom nemuseli popisovat každý zvlášť, definujeme *Bod* jako nový datový typ – jako *třidu*. Jednotlivé body, které nakreslí uživatel našeho grafického editoru, budou v programu představovány proměnnými (budeme také říkat *objekty* nebo *instancemi*) třídy *Bod*. Každá instance třídy *Bod* bude obsahovat *datové složky*⁸, které popisují jeho polohu a barvu.

Metoda

K zacházení s jednotlivými body budeme používat operace, které jsme k tomu účelu naprogramovali. Také tyto operace budou součástí definice třídy; v OOP se nazývají *metody*.

Při definici nového typu jsme udělali (zatím jen naznačili) dvě věci: Určili jsme

- datovou reprezentaci nového typu,
- operace s ním (metody).

Zapouzdření

V objektovém programování se tomu říká *zapouzdření* (anglicky *encapsulation*). Poznamenejme, že pro některé složky můžeme specifikovat tzv. přístupová práva – můžeme např. předepsat, že je smějí používat pouze metody dané třídy (tzv. soukromé složky) nebo že je smí používat kdokoli (složky veřejně přístupné). O veřejně přístupných metodách hovoříme také jako o *rozhraní třídy*.

Ukrytí implementace

Jedno z pravidel objektového programování říká, že s datovými složkami dané třídy bychom měli manipulovat pouze prostřednictvím jejích metod. Později si vysvětlíme, proč je to tak důležité.

⁸ Datovým složkám se v objektovém programování často říká také *atributy*. V souvislosti s C# se ale tomuto termínu vyhneme, neboť jej budeme používat pro něco podstatně jiného.

Objektový program se skládá pouze z objektů, tedy z instancí objektových typů, a tyto objekty si navzájem posílají zprávy. To zní asi velice nesrozumitelně, ale v podání jazyka C# to prostě znamená, že volají (spouštějí) své metody.

Vezměme opět grafický editor a podívejme se na něj jako na celek. Celá aplikace bude nejspíš představována instancí třídy Editor, jednotlivé součásti obrázku budou představovány instancemi tříd Bod, Úsečka atd. Když uživatel stiskne tlačítko, kterým určí, že chce někde vytvořit nový bod, tedy instanci třídy Bod, dozví se to editor. Ten požádá o vytvoření instance třídy Bod (pošle třídě Bod zprávu, která bude představovat žádost o vytvoření instance určitých vlastností). Bude-li třeba vybraný bod přemístit, pošle editor této instanci zprávu „přemísti se“, tj. zavolá odpovídající metodu.

Bod
souřadnice x souřadnice y barva
zjistix() zjistiy() nastavx() nastavy() zjistibarvu() nastavbarvu()

Při návrhu a popisu objektového programu se dnes často využívá tzv. unifikovaný modelovací jazyk (Unified Modeling Language, UML). To je soubor pravidel pro vytváření diagramů, které popisují různé aspekty řešené úlohy. Mezi jiným popisuje také třídy a jejich instance, které se v úloze vyskytují, a jejich vzájemné vztahy.

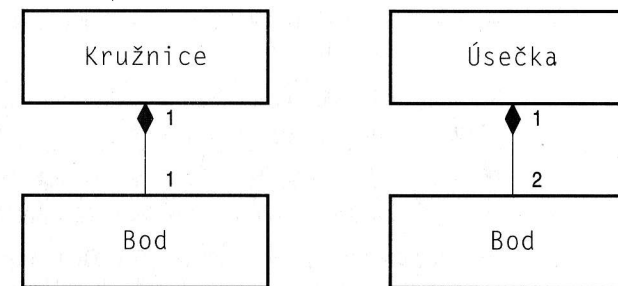
Třídy a jejich vztahy popisuje tzv. diagram tříd (class diagram). Pro znázornění jednotlivých tříd se používá ikona ve tvaru obdélníka. V horní části je jméno třídy, pod ní – oddělen vodorovnou čarou – je seznam datových složek a pod ním, opět pod vodorovnou čarou, seznam metod. (Pokud seznamy datových složek nebo metod nepotřebujeme, můžeme je samozřejmě vynechat.) Možné znázornění třídy Bod ukazuje obrázek po levé straně.

Pojďme dál. Vedle bodů budeme v programu potřebovat také kružnice. Jak víme, je každá kružnice určena svým středem a poloměrem. Vedle toho ovšem musíme ještě určit tloušťku čáry a její barvu. Mohli bychom tedy tvrdit, že k popisu kružnice nám stačí pětice čísel; ovšem podíváme-li se na to pozorněji, uvědomíme si, že střed kružnice je bod a k jeho popisu můžeme využít už hotový typ Bod.

To znamená, že nový typ – třída – Kružnice bude popsán jednou instancí třídy Bod a třemi čísly (poloměrem, barvou a tloušťkou čáry).

Podobně třída Úsečka bude popsána dvěma body, barvou a tloušťkou čáry. Při definici nového typu nám nic nebrání použít jako datovou složku typ, který už existuje. Tomu říkáme *skládání objektů*.

Skládání objektů se v UML vyznačuje čarou zakončenou vyplněným kosočtvercem. Tato „šipka“ směřuje od třídy představující komponentu ke třídě, která ji používá. Vztah mezi třídami Bod, Úsečka a Kružnice ukazuje obr. 1.1.



Obr. 1.1. Znázornění vztahu mezi třídami Bod, Úsečka a Kružnice. Čísla, připojená ke spojnicím, vyjadřují, že Kružnice obsahuje jeden Bod, zatímco úsečka obsahuje dva Body

Zastavme se na ještě u identifikátorů našich tříd, jako je Kružnice nebo Úsečka. V programech se k pojmenování objektů, metod atd. obvykle písmena s háčky a čárkami nepoužívají – většina starších programovacích jazyků to nedovoluje, neboť pocházejí převážně z anglického prostředí. Nicméně při analýze úlohy, např. v diagramech v UML, takovéto identifikátory nejen ničemu nevadí, ale navíc usnadňují přemýšlení o řešeném problému.

Dědičnost

Samotné zapouzdření je sice dobré, zlepšuje přehlednost programu, ale je to málo. Při psaní grafického editoru nejspíš brzy zjistíme, že velice často píšeme podobné úseky programu.

V našem editoru budeme např. chtít jako jednu z možností nabídnout otáčející se úsečku, třídu ÚsečkaRot. Ta bude mít stejné vlastnosti jako obyčejná úsečka (bude popsána stejnými datovými složkami), navíc ale bude obsahovat údaj o rychlosti otáčení a metody, které nám umožní tuto rychlost zjišťovat a měnit.

Bylo by jistě nesmyslné při programování otáčivé úsečky opisovat znovu vše, co jsme naprogramovali pro obyčejnou úsečku. OOP proto nabízí mechanismus nazvaný (nepříliš šťastně) *dědičnost*, který umožňuje od existující třídy odvodit třídu novou.⁹

Odvozená třída bude obsahovat všechny datové složky a všechny metody předka; k nim můžeme přidat nové datové složky a nové metody. Můžeme také předefinovat (změnit, překrýt) některou z metod



⁹ Pro třídu, od které odvozujeme, se obvykle používá označení *předek*, *rodčovská* nebo *bázová* třída. Pro třídu, kterou od ní odvodíme pomocí dědičnosti, se používá označení *potomek*, *odvozená* nebo *dceřinná* třída.

předka. Při dědění nemůžeme žádnou datovou složku ani žádnou metodu odstranit.

Vraťme se k našemu příkladu s otáčející se úsečkou. Třída `ÚsečkaRot`, kterou odvodíme jako potomka třídy `Úsečka`, bude obsahovat:

- Stejně datové složky jako rodičovská třída. My k nim přidáme novou datovou složku vyjadřující rychlost otáčení.
- Stejně metody jako rodičovská třída. My k nim přidáme metody pro zjištění a nastavení rychlosti otáčení a změníme metodu pro kreslení, neboť otáčející se úsečka se bude kreslit jinak než obyčejná úsečka.

Ovšem dědičnost se při práci na grafickém editoru uplatní ještě jinak. Při programování tříd vyjadřujících různé grafické objekty brzy zjistíme, že mnohé části programu jsou stejné nebo hodně podobné:

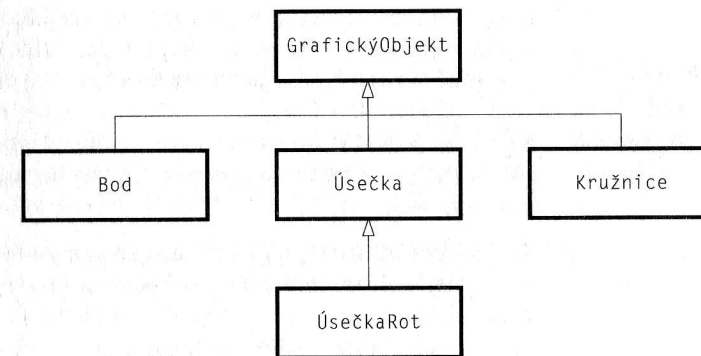
Všechny třídy vyjadřující grafické objekty obsahují datovou složku popisující barvu a metody pro její zjištění a nastavení, všechny tyto třídy obsahují metody pro nakreslení a smazání objektu a další...

Programovat jednu věc několikrát, to nikdy není dobré; jednak je to zbytečná práce navíc, jednak rozhodneme-li se něco změnit, musíme to měnit na několika místech a snadno se stane, že na některé místo zapomeneme.

Proto bude rozumné vzít společné vlastnosti všech tříd grafických objektů z našeho programu a přenést je do společného předka. Tuto třídu nazveme výstižně `GrafickýObjekt` a ostatní třídy `Bod`, `Úsečka` a další pak odvodíme jako její potomky.

Jestliže bude třída `GrafickýObjekt` obsahovat např. metodu `nakresli`, která objekt nakreslí, bude tuto metodu obsahovat i každá z odvozených tříd. Je ovšem pravděpodobné, že ji budeme muset v každém z potomků předefinovat; přece jen úsečka se kreslí jinak než kružnice a ta se kreslí jinak než obecný grafický objekt. (Jak dále uvidíme, ten vlastně vůbec nakreslit nejde.)

V UML znázorňujeme dědičnost pomocí nevyplněné šipky, která směřuje od potomka k předkovi. Třída, která je sama potomkem, může sloužit jako předek jiné třídy. Tak mohou vzniknout *dědické hierarchie*, posloupnosti tříd, navzájem odvozených děděním. Obrázek 1.2 ukazuje dědickou hierarchii tříd, o kterých jsme hovořili v příkladu o grafickém editoru.



Obr. 1.2 Dědická hierarchie tříd z příkladu o grafickém editoru

Polymorfizmus

Zamysleme se nyní nad během našeho grafického editoru. Uživatel si bude kreslit: To znamená, že bude vytvářet úsečky, rotující úsečky, kružnice a další instance jednotlivých tříd představujících grafické objekty. Se všemi se bude v programu zacházet velmi podobně: Bude třeba je vytvořit, určit jejich polohu, barvu atd., bude potřeba je nakreslit, možná smazat atd. Z našich předchozích úvah vyplynulo, že budou obsahovat řadu stejných metod. Bylo by tedy velice pohodlné, kdybychom s nimi mohli také v programu jednotným způsobem zacházet – kdybychom se nemuseli starat o konkrétní typ té které instance, kdyby stačilo vědět, že jde o instanci třídy, která je odvozena od společného předka, třídy `GrafickýObjekt`.

To opravdu v objektově orientovaném programování jde. Platí totiž velice důležité pravidlo, které lze zjednodušeně formulovat slovy *Potomek může vždy zastoupit předka*. To znamená, že na místě, kde v programu očekáváme instanci předka, můžeme vždy použít instanci potomka. (Občas se o tom hovoří jako o pozdní vazbě.)

V našem editoru tedy použijeme pro reprezentaci nakresleného obrázku např. pole instancí třídy `GrafickýObjekt` a budeme do něj ukládat úsečky, kružnice, body – podle toho, co uživatel zrovna vytvoří.

Když budeme chtít celý obrázek nakreslit, vezmeme prostě jednu instanci po druhé a pošleme jí zprávu, že se má nakreslit, tj. zavoláme její metodu `nakresli`. O skutečný typ instance se nebudeme starat: Ať je jakéhokoli typu, metodu `nakresli` určitě obsahuje, neboť ji obsahuje společný předek. (V každém z odvozených typů je samozřejmě definována jinak, neboť úsečka se kreslí jinak než kružnice.)

Abstraktní třída

Této vlastnosti se říká *polymorfizmus* neboli mnohotvarost. Jeden objekt může mít mnoho tvarů: Např. jedna instance třídy GrafickýObjekt může představovat podle okolností bod, kružnici, ...

Při úvahách o třídě GrafickýObjekt, která představuje obecný grafický objekt, narazíme na malý problém: Jak se vlastně takový obecný grafický objekt kreslí?

Nijak – kreslit obecný grafický objekt přece nemá smysl, stejně jako nemá smysl třeba chtít kupovat obecnou potravinu nebo obouvat si obecnou obuv. GrafickýObjekt je tzv. *abstraktní třída*, tj. třída, která představuje pojem natolik abstraktní, že určitá operace, typická pro všechny její podtřídy, pro něj nemá smysl. V našem případě nemá smysl uvažovat o kreslení obecného grafického objektu. Přesto je jasné, že třída GrafickýObjekt bude muset mít metodu nakresli, abychom ji mohli použít pro instance odvozené třídy – abychom mohli říci „nakresli tento grafický objekt“. Ve skutečnosti to bude instance některého z odvozených typů, ale v době psaní programu nevíme kterého.

Metodu, která nemá v dané třídě konkrétní smysl, ale kterou musíme definovat, abychom ji mohli volat v odvozených třídách, označujeme také jako *abstraktní*.

Dědičnost versus skládání

Při vytváření tříd pro grafické objekty bychom se mohli nechat svést následující úvahou: Kružnice je popsána středem a poloměrem. Střed je bod. Odvodíme tedy třídu Kružnice jako potomka třídy Bod a ušetříme tak jednu zdánlivě zbytečnou třídu, totiž GrafickýObjekt.

To by mohlo na první pohled fungovat. Jenže ono by to také znamenalo, že můžeme kružnici použít všude, kde program očekává bod – a to může vést k problémům, neboť kružnice *není* bod.

Odvozená třída totiž vždy znamená zvláštní případ bazové třídy. Úsečka je zvláštní případ grafického objektu; rotující úsečka je zvláštní případ úsečky. Proto má smysl odvodit třídu Úsečka jako potomka třídy GrafickýObjekt a třídu ÚsečkaRot jako potomka třídy Úsečka. Na druhé straně kružnice *není* bod, takže odvodit třídu Kružnice jako potomka třídy Bod nemá smysl.

Ovšem kružnice *má* bod (střed). Proto má smysl použít skládání.

Předchozí úvahy ukazují něco, čemu se občas říká *test je – má* (anglicky *is a – has a*). Jestliže můžeme říci, že třída B je zvláštní případ třídy A, má smysl odvodit třídu B jako potomka třídy A. Jinak je vhodnější použít skládání.

1.6 Jazyk C#

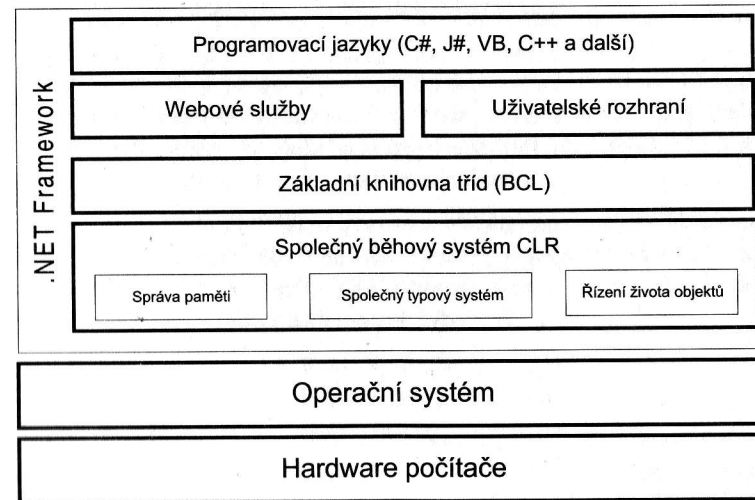
C# je nový programovací jazyk; první zmínky o něm pronikly na veřejnost někdy v roce 2000 a oficiálně byl uveden na trh počátkem roku 2002 jako součást Visual Studia .NET. Můžeme se na něj dívat jako na zjednodušenou, vylepšenou a čistě objektovou verzi programovacího jazyka C++.

Označení C# – mříž se někdy také píše jako horní index, C[#] – se přepisuje *C sharp*. Vedle anglické výslovnosti *sí šárp* se u nás setkáme i s českou verzí *cé mříž*.

Původ tohoto označení, pokud vím, nikdo nepublikoval, lze si pouze domýšlet. Jedna oblíbená teorie říká, že # zde představuje hudební značku, která znamená *o půl tónu výš*. Takže nejde o C, ale o *Cis*. Jako název to rozhodně není špatné.

Prostředí .NET Framework

Nezbytnou podmínkou pro programování v jazyce C# a pro spouštění programů v něm vytvořených je instalace prostředí .NET Framework. Toto prostředí se skládá z několika součástí, o nichž se zde alespoň krátce zmíníme (viz též obr. 1.3).



Obr. 1.3 Struktura prostředí .NET Framework

Jádrem prostředí .NET Framework je *společné běhové prostředí*, anglicky označované *Common Language Runtime (CLR)*. Toto prostředí zajišťuje běh programů přeložených z různých programovacích

jazyků – nejen z C# – do mezijazyka MSIL. CLR umožňuje jejich vzájemnou spolupráci, takže různé součásti programu mohou být napsány v různých programovacích jazycích. Stará se mj. o automatickou správu paměti, o řízení doby života objektů a o další věci nezbytné pro běh programů v tomto prostředí.

Jedním z důležitých pojmů, které se v souvislosti s CLR používají, je *řízený kód (managed code)*. To je kód, který běží pod dozorem CLR a využívá jeho služeb – např. kód, který vznikne překladem programu napsaného v C#.

Další důležitou součástí prostředí .NET Framework je knihovna tříd, nazývaná souhrnně *Základní knihovna tříd (Basic Class Library, BCL)*. Pokrývá téměř vše, nač si lze vzpomenout – od vstupních a výstupních operací po kontejnery (třídy sloužící pro ukládání různých druhů dat). Vzhledem k tomu jazyk C# prakticky nepotřebuje své vlastní knihovny.

Nad knihovnou BCL jsou ještě knihovny pro tvorbu grafického uživatelského rozhraní programů a knihovny pro webové služby.

Nejvyšší vrstvu prostředí .NET Framework tvoří překladače z různých programovacích jazyků a překladače JIT. (Překladače JIT jsou celkem tři a liší se tím, zda se program přeloží celý v době instalace, v době spuštění nebo zda se budou překládat jeho jednotlivé části až v okamžiku, kdy jsou za běhu opravdu potřeba.)

Prostředí .NET Framework podporuje práci s tzv. metadaty. To jsou dodatečné informace, které se ukládají do MSIL v době překladu a které umožňují získávat z přeloženého programu velmi podrobné informace o třídách, které obsahuje, o jejich metodách atd. Tomuto mechanismu se říká *reflexe (reflection)*.

Jazyk C# – a prostředí .NET Framework – ovšem umožňuje spolupracovat i s tzv. *neřízeným kódem (unmanaged code)*. To je kód, který neběží v rámci CLR; může jít o starší programy. Neřízený kód lze psát i v C#, i když to není standardní postup.

Součástí prostředí .NET Framework jsou také mechanismy, které se starají o zabezpečení distribuovaných aplikací. Tím se ale zabývat nebudeme, neboť programování distribuovaných aplikací přesahuje rámec naší knihy.

Kde získat C#

Zdarma Už jsme si řekli, že překladač jazyka C# je součástí všech provedení produktu Microsoft Visual Studio .NET; lze ho také koupit jako samostatný produkt pod názvem Visual C# .NET. Ovšem překladač



spouštěný z příkazové řádky a všechny nástroje potřebné pro vývoj aplikací pro .NET Framework lze získat zdarma. V současné době (březen 2002) si ho lze stáhnout z webových stránek firmy Microsoft na webové adrese <http://msdn.microsoft.com/net/framework/>. Jde ale o poměrně rozsáhlý balík (131 MB), a proto si ho lze stáhnout také rozdělený do 10 dílů po 13 MB.

Všechny tyto části si stáhneme do jednoho pomocného adresáře a spustíme program `setup.exe`, který se nás provede instalací.

Připomeňme, že k instalaci tohoto prostředí potřebujeme operační systém Windows NT 4 (požadován je Service Pack 6a), Windows 2000 (doporučuje se mít Service Pack 2) nebo Windows XP.

Třídy v C#

V C# se rozlišují dvě základní skupiny datových typů: Do první patří tzv. hodnotové typy, což jsou např. čísla, znaky, logické hodnoty, struktury, a do druhé patří objektové typy (třídy). Jak později uvidíme, zacházení s proměnnými těchto dvou skupin se v některých ohledech významně liší.

Třída object

V C#, podobně jako v mnoha jiných čistě objektových jazycích, je řada předdefinovaných tříd, které všechny tvoří jednu hierarchii, tj. jsou odvozeny od společného předka – třídy `object`. V této hierarchii leží i třídy, které ve svém programu definujeme my. Jestliže v deklaraci nové třídy neuvedeme předka, doplní si překladač jako předka třídu `object`. To znamená, že všechny třídy obsahují určité základní metody, a tedy u všech instancí všech tříd lze očekávat jisté společné chování. Tyto metody lze v odvozených třídách samozřejmě předefinovat.

Jako potomci třídy `object` se chovají i hodnotové typy. To znamená, že např. i pro proměnné celočíselných typů můžeme volat metody zděděné od třídy `object`. Hodnotové typy ovšem nepodporují dědičnost.

C#, Java a C++



Čtenáři, kteří se dosud nesetkali s jazyky C, C++ nebo Java, mohou tento oddíl klidně přeskočit. Pokud některý z nich alespoň povrchně znáte, najdete v tomto odstavci upozornění na vlastnosti C#, které vám budou připadat známé, ale mohou způsobit problémy.

Syntax jazyka C#, tedy způsob zápisu programu, se totiž – alespoň na první pohled – velmi podobá syntaxi C++ nebo Javy. To znamená, že programátoři, kteří znají některý z těchto jazyků, se velice brzy naučí

používat příkazy a některé další konstrukce C#. *Ovšem pozor: Podobnost C# s těmito jazyky je povrchní a brzy zjistíte, že C# se v mnoha ohledech podstatně liší.*

- C# je na rozdíl od C++ tzv. čistě objektový jazyk. To znamená, že vše v tomto jazyce jsou objekty.
- Na rozdíl od C++ (a podobně jako v Javě) lze pracovat s C# pouze s dynamickými objekty (vytváříme je pomocí operátoru `new`). O úklid vytvořených objektů se nemusíme starat, .NET Framework obsahuje automatickou správu paměti (tzv. garbage collector). To je mechanismus, který se postará o automatické odstranění nepoužívaných objektů.
- C# má zcela jiné knihovny než C++ i než Java. Tyto knihovny jsou společné pro všechny programovací jazyky pro .NET Framework. Obsahují mj. třídy pro vytváření grafického uživatelského rozhraní, pro programování vstupních a výstupních operací atd. Některé ze tříd této knihovny jsou podobné třídám z knihovny jazyka Java, ale práce s nimi se může odlišovat.
- Zatímco v Javě se prakticky všechny metody volají pomocí pozdní vazby, v C# můžeme – podobně jako v C++ – používat i časnou vazbu. Metody, které chceme volat pomocí pozdní vazby, označujeme klíčovými slovy `virtual` a `override`.
- C#, podobně jako Java a na rozdíl od C++, obsahuje vestavěnou podporu vícevláknového programování (multithreadingu).
- Na rozdíl od Javy podporuje C# přetěžování operátorů.
- Součástí C# jsou i nástroje, které nemají v C++ ani v Javě obdobu – např. používání tzv. vlastností a atributů.

2 První programy

Chceme-li se naučit nějaký programovací jazyk, musíme v něm psát programy – jinak to nejde. Proto budu předpokládat, že si vše, o čem zde budeme hovořit, hned vyzkoušíte. V této kapitole si napíšeme jednoduchý program a ukážeme si, jak se překládá a spouští. Postupně se seznámíme s nejzákladnějšími prvky jazyka, abychom si později, až se pustíme do výkladu „naostro“, mohli ukazovat alespoň trochu smysluplné příklady.

2.1 Nápis na obrazovce

Většina dnešních učebnic programování začíná jednoduchým programem, který vytiskne nějaký vtipný text – nejlépe něco jako „hello, world“ – a skončí. Je to dobrá tradice, a proto se jí přidržíme.

Zdrojový text

Editor

Nejprve vytvoříme zdrojový kód programu. K tomu potřebujeme libovolný ASCII editor, tj. editor, který vytvoří soubor obsahující pouze znaky, rozčleněné na řádky. Pokud máte k dispozici editor, který ukazuje čísla řádků, dejte mu přednost, bude se vám to hodit při hledání chyb. Pokud takový editor nemáte, můžete použít třeba známý notepad (poznámkový blok) z Windows. Máte-li k dispozici Visual Studio .NET nebo jiný nástroj pro vývoj programů, můžete samozřejmě použít editor, který je jeho součástí. Jako editor pro vývoj programů se nehodí například Word nebo WinText 602, které text formátují, tj. vkládají do něj i dodatečné informace o velikosti a typu písma apod.

Nyní tedy spusťte editor a vytvořte v něm soubor `Prvni.cs`, který bude obsahovat následující text. Vytvořený soubor uložte do samostatného adresáře:

```
/* náš první program v C# - soubor Prvni.cs */  
  
class Banka  
{  
    static void Main()  
    {  
        System.Console.WriteLine("Pozor, tunel!");  
    }  
}
```



Je důležité, abyste při opisování dodrželi velikost písmen, neboť v C# se velká a malá písmena důsledně rozlišují. Soubor také musí mít příponu .cs.¹⁰



Tento zdrojový text najdete také na WWW¹¹ v adresáři Kap01\01.

Překlad

Jak víme, musíme program nejprve přeložit do podoby srozumitelné pro počítač, teprve pak jej budeme moci spustit. V našem případě to znamená překlad do jazyka MSIL. Postará se o něj překladač csc.exe, který je součástí prostředí .NET Framework; najdeme ho v adresáři, který se jmenuje \Winnt\Microsoft.NET\Framework\v1.0.3705 nebo podobně (může se lišit číslo verze ve jménu posledního podadresáře).

V konzolovém okně nejprve přejdeme do adresáře se zdrojovým souborem a pak spustíme překladač příkazem

```
csc Prvni.cs
```

Příponu cs ve jménu souboru nelze vynechat. Pokud překlad proběhne bez problémů, vypíše překladač na konzolu (do příkazové řádky) zprávu, že se rozeběhl, a nic víc (viz obr. 2.1).

Jak to zkazit



Může se stát, že se náš první program nepodaří přeložit. V tom případě nás buď překladač csc nebo operační systém informuje o vzniklých problémech.

- Objeví-li se na obrazovce hlášení Bad command or file name, případně scs není názvem vnitřního ani vnějšího příkazu, proveditelného programu nebo dávkového souboru nebo podobná zpráva, znamená to, že operační systém nenašel program csc.exe. Pokud máme prostředí .NET Framework správně nainstalované, znamená to, že operační systém nezná cestu do podadresáře s jeho instalací. Buď ji tedy zadáme při spouštění, např.

```
C:\Winnt\Microsoft.NET\Framework\v1.0.3705\csc Prvni.cs
```

nebo (raději) uložíme tuto cestu do systémové proměnné PATH.

- Objeví-li se hlášení

```
error CS2001: Source file 'Prvni' could not be found
fatal error CS2008: No inputs specified
```

¹⁰ Pokud použijete notepad, nezapomeňte při ukládání v dialogovém okně *Uložit jako* nastavit typ souboru *Všechny soubory*, resp. *All files*. Jinak se ke jménu připojí ještě přípona .txt, takže dostanete soubor Prvni.cs.txt.

¹¹ Internetová adresa stránky, z níž si můžete stáhnout zdrojové texty příkladů, je uvedena v úvodu knihy na str. 9.

znamená to nejspíš, že jsme ve jménu souboru zapomněli uvést příponu cs.

Syntaktické chyby

Ne vždy napíšeme do zdrojového textu programu přesně to, co potřebujeme. Pokud bude náš program obsahovat prohřešek proti pravidlům jazyka C# (syntaktickou chybu), překladač to zjistí a nebude schopen přeložit. Smažme např. středník na konci šestého řádku, tak náš program bude mít tvar

```
/* Náš první program v C# - obsahuje syntaktickou chybu */
class Banka
{
    static void Main()
    {
        System.Console.WriteLine("Pozor, tunel!")
    }
}
```

a výsledek zkuste přeložit. Překladač vypíše zprávu

```
Prvni.cs(6,44): error CS1002: ; expected
```

jež nám říká, že v souboru Prvni.cs našel překladač na šestém řádku syntaktickou chybu, kterou lze v dokumentaci vyhledat pod označením CS1002. Následuje stručná charakteristika chyby v angličtině: *očekávaný středník*. (Zdaleka ne vždy jsou ovšem chybová hlášení tak přehledná a snadno srozumitelná jako v tomto případě.)

Soubory

Jestliže se nyní podíváte do adresáře, kde je uložen zdrojový program, najdete v něm kromě souboru Prvni.cs také soubor Prvni.exe. To je *spustitelný soubor*, tedy hotový program. Je ve formátu PE (*portable executable*, standardní formát spustitelných souborů pro 32bitové Windows), neobsahuje ovšem téměř žádný strojový kód, ale instrukce v jazyce MSIL.

Budeme-li chtít hotový program přenést na jiný počítač, budeme potřebovat nášet právě tento soubor.

Běh programu

Program je hotov, nezbyvá, než ho vyzkoušet. Jsme-li v adresáři, kde je tento program uložen, spustíme ho příkazem

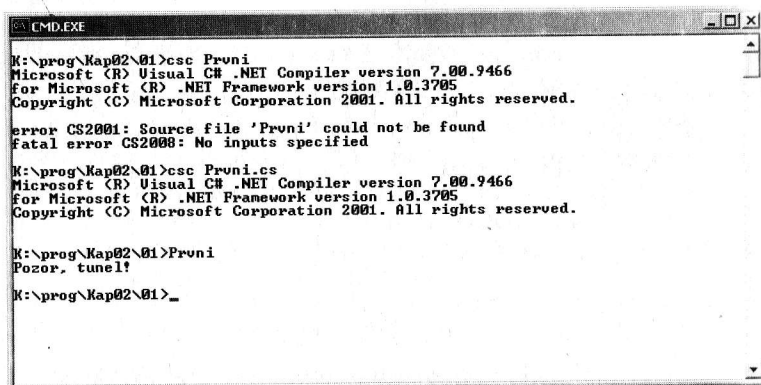
```
Prvni
```

Program vypíše do následujícího řádku překvapující sdělení

```
Pozor, tunel!
```


O překlad programu z MSIL do strojového kódu se postaralo prostředí .NET Framework samo, bez našeho zásahu.

Překlad, spuštění a běh našeho prvního programu ukazuje obr. 2.1.



```
K:\prog\Kap02\01>csc Prvni
Microsoft (R) Visual C# .NET Compiler version 7.00.9466
for Microsoft (R) .NET Framework version 1.0.3705
Copyright (C) Microsoft Corporation 2001. All rights reserved.
error CS2001: Source file 'Prvni' could not be found
fatal error CS2008: No inputs specified

K:\prog\Kap02\01>csc Prvni.cs
Microsoft (R) Visual C# .NET Compiler version 7.00.9466
for Microsoft (R) .NET Framework version 1.0.3705
Copyright (C) Microsoft Corporation 2001. All rights reserved.

K:\prog\Kap02\01>Prvni
Pozor, tunel!

K:\prog\Kap02\01>_
```

Obr. 2.1 Překlad, spuštění a běh našeho prvního programu

2.2 Co jsme naprogramovali

Nyní umíme napsat, přeložit a spustit jednoduchý program; je tedy nejvyšší čas, abychom se také podívali, co vlastně znamená. Náš výklad ovšem nebude úplný, v mnoha případech si řekneme jen nejdůležitější a zjednodušené informace. Upřesníme je pak v dalších kapitolách.

Komentář

V prvním řádku najdete text

```
/* Náš první program v C# - soubor Prvni.cs */
```

To je *komentář* – část programu, která nemá pro překladač vůbec žádný význam, nijak neovlivní chod přeloženého programu. Komentáře slouží programátorům, kteří si do nich zapisují poznámky o významu různých částí programu.

Komentář začíná dvojicí znaků `/*` zapsaných bezprostředně za sebou a končí dvojicí znaků `*/`, opět zapsaných bezprostředně za sebou. Vše, co je mezi těmito „komentářovými závorkami“, překladač ignoruje. (Přesvědčte se o tom: Smažte komentář v souboru `Prvni.cs`, potom program přeložte a spusťte. Program se přeloží a poběží úplně stejně jako předtím.)

Komentář ohraničený závorkami `/* a */` může zabírat i několik řádek. C# ovšem zná i jednořádkový komentář. Ten začíná dvojicí lomítek zapsaných těsně za sebou a končí na konci řádku. Náš program by tedy mohl začínat také takto:

```
// Náš první program v C# - soubor Prvni.cs
```

V C# existuje ještě třetí druh komentářů, tzv. dokumentační komentáře. Ty začínají znaky `///` a končí na konci řádku. Vrátime se k nim později (v dodatku).

Ze začínajícím programátorům komentáře často připadají zbytečné, neboť svému programu přece rozumějí. Jenže o to nejde: Nyní mu sice rozumíte, ale budete mu rozumět i za týden, až zjistíte, že v něm chcete něco opravit? A co za měsíc, za rok? Až po čase budete ve svém programu potřebovat něco změnit, budou pro vás dobře napsané komentáře představovat neocenitelnou pomoc. (A představu, že program lze napsat hned na poprvé tak, aby ho již nebylo třeba měnit, pusťte z hlavy; to se téměř nestává ani zkušeným programátorům.)

Deklarace třídy

Na dalším řádku začíná deklarace jediné třídy v našem programu. To říká slovo `class`, za kterým následuje jméno (*identifikátor*) této třídy, představované slovem `Banka`. Za jménem třídy následuje levá (otevírací) složená závorka `{`. Mezi ní a odpovídající pravou složenou závorkou `}` pak je *tělo třídy*, tj. programový popis třídy. Deklarace třídy `Banka` tedy vypadá takto:

```
class Banka {
    // ... Tělo třídy ...
}
```

Metoda Main()

Jak víme, mohou třídy obsahovat jednak datové složky, ve kterých se ukládají data, jednak metody, které vyjadřují operace s instancemi tříd. Třída `Banka` neobsahuje žádné datové složky (ke skutečné bance má opravdu daleko). Obsahuje jedinou metodu, která se jmenuje `Main()`.¹² Její deklarace je

```
static void Main()
{
    System.Console.WriteLine("Pozor, tunel!")
}
```

¹² Samotný identifikátor této metody je `Main`. Závorky za ním v zápisu `Main()` naznačují, že jde o metodu, nikoli o datovou složku; měly by usnadnit čtení textu.

Metoda `Main()` je nesmírně důležitá, neboť od ní bude program začít. Když zadáme v příkazové řádce příkaz

První

kterým program spouštíme, říkáme operačnímu systému našeho počítače: Najdi soubor `První.exe` a spusť jeho metodu `Main()`.

Soubor `První.exe` ovšem obsahuje příkazy, které způsobí, že se nejprve zavolá překladač JIT, který přeloží MSIL z tohoto souboru do strojového kódu. V tom, co vznikne překladem do strojového kódu, se vyhledá metoda `Main()` a ta se spustí. To znamená, že se začnou provádět příkazy, které tato metoda obsahuje, počínaje prvním.

První řádek, „hlavička“, metody `Main()`, bude v našich úvodních příkladech vypadat vždy tak, jak to vidíte v našem příkladu. Bude tedy obsahovat slova `static`, `void` a `Main` v uvedeném pořadí. Identifikátor `Main` musí mít první písmeno velké a ostatní malá. Za ním budou následovat prázdné kulaté závorky.



Později uvidíme, že v této deklaraci lze leccos změnit. Zatím se o to ale nepokoušejte, nejspíš byste se dostali do zbytečných problémů.

Za hlavičkou následuje „tělo“ metody `Main()`. To jsou příkazy, které má tato metoda provést, uzavřené mezi složené závorky `{ }`. Naše metoda `Main()` obsahuje jediný příkaz,

```
System.Console.WriteLine("Pozor, tunel!");
```



I když vypadá na pohled komplikovaně, neříká tento příkaz nic jiného než „vezmi text

Pozor, tunel!

a vypiš ho do standardního výstupu, tj. na obrazovku (konzolu)“.

řetězec

Místo o „textu“ ovšem zpravidla hovoříme o (znakovém) řetězci, řetězec, který má vystoupit, musíme zapsat mezi dvojici uvozovek, tento zápis označujeme jako řetězcovou konstantu.

Bude-li metoda `Main()` obsahovat více příkazů, budou se většinou provádět v pořadí, v jakém je zde zapíšeme; o výjimkách z tohoto pravidla si řekneme později. Program skončí, když přejde přes složenou závorku `}`, uzavírající tělo metody `Main()`.

Doplňující poznámky



Deklarace metody `Main()` začíná slovem `static`, které říká, že jde o tzv. statickou metodu. To znamená, že ji lze volat i v případě, že neexistuje ani jedna instance třídy `Banka`. Slovo `void` říká, že metoda `Main()` nevrací žádný výsledek.

Už jsme si řekli, že příkaz `System.Console.WriteLine("Pozor");` vypisuje zadaný text do standardního výstupu. To většinou znamená na konzolu, tj. buď na obrazovku v textovém režimu nebo do okénka v grafickém prostředí, které se chová jako obrazovka v textovém režimu (s příkazovým řádkem). Tento výstup můžeme příkazem operačního systému přeměrovat, např. do souboru. Spustíme-li program příkazem

```
První > realita.txt
```

nevypíše se text na obrazovku, ale do souboru `realita.txt`.

Ódy žlutého koně: čeština v programu

Nezadatelným právem uživatele počítače je, aby s ním počítač komunikoval v jeho rodném jazyce. Nápis, který předchází program vypisoval, byl sice český, ale neobsahoval žádné znaky s diakritickými znaménky, typické pro češtinu. Podívejme se tedy, jak to s naší řečí v programech v C# je.

Druhý program

Opraváři prý testovali dálnopisy tak, že na nich napsali větu „Quick brown fox jumps over the lazy dog.“ Obsahuje totiž všechny znaky anglické abecedy a přitom natukat ji a zkontrolovat, zda se vypsala správně, je mnohem snazší, než napsat a zkontrolovat jednotlivá písmena.

Pro testování, zda se správně zobrazují znaky specifické pro češtinu, se ze stejných důvodů používá věta o žlutoučkém koni, který příšerně ůpěl d'ábelské ódy. (Podobná věta se jistě používá i při testování slovenštiny, já ji ale bohužel neznám.)

Zajdeme ale ještě dál: Použijeme české znaky nejen ve výstupu, ale i v identifikátoru třídy a v názvu souboru. Náš druhý program tedy bude vypadat takto:

```
/* Test češtiny v C# - soubor ŽlutýKůň.cs */
```

```
class ŽlutýKůň
```

```
{
```

```
    public static void Main()
```

```
{
```

```
        System.Console.WriteLine("Žlutoučký kůň příšerně "+  
        "ůpěl d'ábelské ódy.");
```

```
}
```

```
}
```

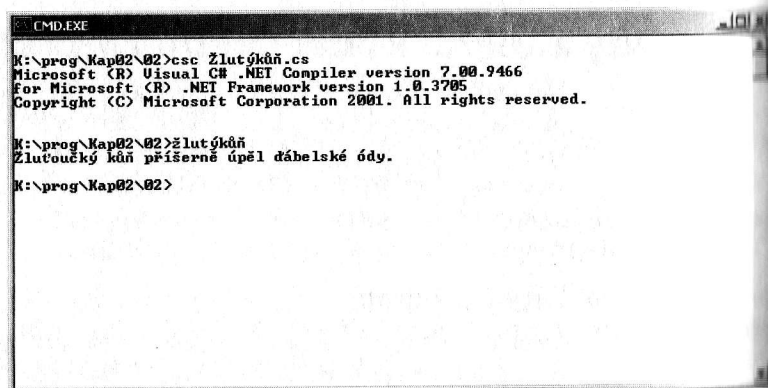
Program uložíme do souboru `ŽlutýKůň.cs`¹³, přeložíme ho příkazem

`csc ŽlutýKůň.cs`

a spustíme příkazem¹⁴

`ŽlutýKůň`

Na obr. 2.2, který ukazuje výsledek, je vidět, že české znaky nepůsobí žádné problémy – ať jde o výstup, o identifikátory nebo o jméno souboru. Zdrojový text tohoto programu najdete na WWW v adresáři `Kap02\02`.



Obr. 2.2 C# si s češtinou docela rozumí



Ovšem v chybových hlášeních mohou být s češtinou přece jen problémy. Jestliže se spletete a zapomenete při volání překladače uvést příponu `.cs`, dostanete chybové hlášení `Source file 'Žlutrk' could not be found`. Ve zprávách o syntaktických chybách však překladač vypisuje jméno souboru správně.



Přesměrujeme-li výstup programu `ŽlutýKůň.exe` do souboru příkazem `ŽlutýKůň > konina.txt`

zjistíme, že jej běžné editory pro Windows (např. notepad) nepřečtou správně. Příčina je jednoduchá: Tyto editory očekávají češtinu v kódování pro Windows, tedy v kódové stránce 1250. Ovšem pro výstup do konzolového okna používá systém Windows dosovské kódování Latin 2, tedy kódovou stránku 852. Proto i soubor `konina.txt` obsahuje češtinu v kódové stránce 852.

¹³ Zdrojový soubor se v jazyce C# může, ale nemusí jmenovat stejně jako třída, kterou obsahuje.

Rozdělený řetězec

Znakový řetězec, který jsme v našem druhém programu vypisovali, se nám nevešel na jeden řádek. Proto jsme ho rozdělili do dvou řetězcových konstant, které jsme spojili znaménkem plus:

`"Žluťoučký kůň přišerně " + "úpěl ďábelské ódy."`

Program si tyto dva znakové řetězce spojil v jeden a ten pak vypsal. Podobně lze samozřejmě spojit i více znakových řetězců.

Programátoři, čeština a my

Mnozí programátoři píší komentáře ke svým programům buď anglicky (kdyby to náhodou četl někdo cizí...) nebo alespoň česky (tj. bez háčeků a čárek). Důvod je jednoduchý: Už jsme si naznačili, že různých prostředích se můžeme setkat s různými způsoby kódování češtiny, a při přenosu programu z jednoho prostředí do jiného mohou vzniknout problémy.

V naší knize budeme ovšem psát komentáře důsledně česky, to znamená i s diakritickými znaménky. I zde je důvod jednoduchý: *Jak vidíte z této kratké veticky, cesky psane komentare byvaji spatne srozumitelne*. Nebudeme si tedy přidělovat zbytečnou práci s luštěním čestiny. Navíc programovací jazyk C# je určen – alespoň zatím – pouze pro Windows, a proto žádné problémy s přenosem do jiných prostředí nehrozí.

Také používání znaků s diakritickými znaménky v identifikátorech, tedy ve jménech tříd, proměnných ap., není příliš obvyklé – především proto, že jen málo programovacích jazyků to dovoluje. C# je zatím spíše čestnou výjimkou; většina programovacích jazyků požaduje, aby se identifikátory skládaly pouze z písmen anglické abecedy a z číslic. Ovšem v poslední době se stále častěji objevují překladače, které používání národních abeced podporují. Je to logické: Můžeme-li dávat součástí svého programu nezkomolená jména, můžeme se snáze soustředit na problém, který řešíme. *Proto budeme této možnosti využívat.*

2.4 Seskupení (assembly)

Když jsme si říkali, že překladem zdrojového programu vznikne spustitelný soubor, neříkali jsme si celou pravdu. Překladač ve skutečnosti vytvoří něco, co se nazývá *assembly* (budeme to překládat slovem *seskupení*).

¹⁴ Ve jménu souboru – na rozdíl od jména třídy – nezáleží na tom, zda použijeme malá či velká písmena.

Seskupení může obsahovat jeden nebo několik modulů, tj. souborů ve formátu PE pro prostředí .NET; může jít o spustitelné soubory (.exe), o dynamické knihovny (.dll) nebo o jiné součásti programů. Vedle toho obsahuje tzv. manifest, soubor s dodatečnými informacemi.

Soubory .exe nebo .dll obsahují kromě instrukcí v jazyku IL také tzv. metadata. To jsou doplňující informace o datových typech – např. jak se jmenují třídy v nich, jaké mají metody atd. Tyto informace mohou využívat nejen překladače, ale i ostatní programy, např. při automatickém generování dokumentace. Z programů jsou dostupné pomocí tzv. reflexe.

V seskupení je také uložena informace o *verzi* a o *kultuře* (culture). Verze jsou číslovány, číslo verze seskupení se skládá z hlavního a vedlejšího čísla, čísla překladu a čísla revize. Kultura znamená vlastně jazykovou mutaci seskupení.

Seskupení představují základní jednotku při šíření hotových programů. Pro prostředí .NET rozlišuje soukromá a sdílená seskupení (private, shared assembly). Soukromá seskupení jsou součástí jediné aplikace, sdílená seskupení slouží všem programům spouštěným v prostředí .NET. Číslování verzí se používá pouze pro sdílená seskupení.

Rozsah této knihy nám nedovolí zabývat se sdílenými seskupeními.

3 Jednoduché příklady

V minulé kapitole jsme napsali svůj první program v C#, ukázali jsme si, jak ho přeložit a spustit. V této kapitole si postupně ukážeme některé další možnosti jazyka C#. Mějte prosím stále na paměti, že jde o úvodní přehled, jejichž hlavním cílem je vytvořit si prostředky, které nám umožní ukazovat si rozumné příklady, až si začneme vykládat jazyk C# „naostro“.

3.1 Drobné úpravy programu

Podívejme se znovu na náš první nebo druhý program. Zatím víme, že zápis

```
System.Console.WriteLine("Hujajá");
```

představuje způsob, jak vypsát znakový řetězec "Hujajá". Rozeberme si to trochu podrobněji.

Volání metody

Při výstupu využíváme služeb objektu, který je pro nás připraven v knihovně jazyka C# (přesněji v knihovně prostředí .NET Framework) a který se jmenuje `System.Console`. Metody tohoto objektu mimo jiné umožňují výstup do standardního výstupu a čtení z něj.

Zápis `System.Console.WriteLine("Hujajá")` tedy představuje *volání metody* `WriteLine()` objektu `System.Console`.

Volání metody začíná jménem objektu, za nímž je tečkou připojeno jméno metody. Za jménem metody následují kulaté závorky, v nichž mohou být tzv. *parametry metody* (data, která má metoda zpracovat). Volání metody způsobí, že se metoda spustí – provedou se příkazy, které jsou v ní naprogramovány.

Nový řádek

Někdy potřebujeme rozdělit vystupující text do několika řádků. Vezměme např. druhý program z předchozí kapitoly a upravme ho tak, aby větu o žluťoučkém koni zapsal do tří řádků, do každého dvě slova.

Jednou z možností je rozdělit vystupující text do několika řetězcových konstant a každou z nich vypsát pomocí metody `WriteLine()`, neboť tato metoda po vypsání zadaného textu přejde na nový řádek. Náš program by tedy mohl vypadat takto:

```
/* Přejít na nový řádek C# - soubor zlom1.cs */
```

```
class ŽlutýKůň  
{  
    static void Main()  
    {  
        System.Console.WriteLine("Žlutoučký kůň");  
        System.Console.WriteLine("příšerně úpěl");  
        System.Console.WriteLine("dábelské ódy.");  
    }  
}
```



Zdrojový text tohoto programu najdete na WWW v adresáři Kap03\01. Jestliže ho přeložíme a spustíme, dostaneme

```
Žlutoučký kůň  
příšerně úpěl  
dábelské ódy.
```

Jazyk C# ovšem nabízí i jiné, a v tomto případě lepší řešení. Do takto zapsaného znakového řetězce sice přechod na nový řádek nemůžeme zapsat přímo, ale můžeme do něj vložit tzv. řídicí posloupnost, která přechod na nový řádek vyznačí.

\n Řídicí posloupnost pro přechod na nový řádek má tvar \n. Vložíme-li ji do vypisovaného řetězce, bude metoda Main() vypadat takto:

```
static void Main()  
{  
    System.Console.WriteLine("Žlutoučký kůň\npříšerně "+  
        "úpěl\ndábelské ódy.");  
}
```

Výstup takto upraveného programu bude stejný jako v předchozím případě. Jeho zdrojový text najdete na WWW v souboru zlom2.cs v adresáři Kap03\01.

Doslovný
řetězec

V C# máme ještě třetí možnost – použít tzv. doslovný řetězec. Ten může obsahovat i přímo zapsaný přechod na nový řádek. Zapisujeme ho do uvozovek, před ně ale umístíme ještě znak @ („zavináč“). Metoda Main() s doslovným řetězcem bude mít tvar

```
public static void Main()  
{  
    System.Console.WriteLine(@"Žlutoučký kůň  
příšerně úpěl  
dábelské ódy.");  
}
```

Výstup programu po této úpravě bude stejný jako v předchozích případech. Všimněte si, že tentokrát jsme nepotřebovali dělit řetězec na několik částí. Za toto pohodlí jsme ovšem zaplatili horší úpravou zdrojového textu. Jak druhý, tak třetí řádek totiž musí začínat těsně



Visual C#

u levého okraje stránky, neboť kdybychom je odsadili, staly by se vložené mezery jeho součástí a vypsaly by se i do výstupu.

Zdrojový text tohoto programu najdete na WWW v souboru zlom3.cs v adresáři Kap03\01.

Používáte-li Visual C#, nejspíš jste výstup žádného z našich programů dosud neviděli: Konzolové okno se po spuštění programu sice otevře, ale při ukončení programu se hned zase zavře. Pokud chcete u tohoto prostředí zůstat, můžete použít následující trik: Jako poslední příkaz v metodě Main() napište příkaz

```
System.Console.ReadLine();
```

Tento příkaz způsobí, že se program zastaví a bude čekat, až stisknete klávesu Enter; teprve pak skončí. Upravená metoda Main() našeho posledního programu tedy bude mít tvar

```
public static void Main()  
{  
    System.Console.WriteLine("Žlutoučký kůň\npříšerně "+  
        "úpěl\ndábelské ódy.");  
    System.Console.ReadLine();  
}
```

Prostor jmen

V předchozích odstavcích jsme si řekli, že používáme objekt, který se jmenuje System.Console. Tento název je pěkně dlouhý a jeho opakované vypisování nepříjemně únavné; naštěstí si ale můžeme část práce ušetřit. Než si povíme jak, musíme si ale něco povědět o tzv. *prostorech jmen*.

Ve velkých programech (a také ve velkých programových knihovnách) se vyskytuje obrovské množství objektů a všechny je třeba jednoznačně pojmenovat – přidělit jim identifikátor. Aby nedošlo ke zmatkům, zavádí jazyk C# možnost přidat ke jménu objektu ještě jakési „příjmení“ – zařadit ho do prostoru jmen. Jméno objektu se pak skládá z vlastního identifikátoru, před který je tečkou připojeno jméno odpovídajícího prostoru jmen. Různé objekty mohou mít stejná jména, pokud jsou v různých prostorech jmen.

V našem případě je System jméno prostoru jmen a Console jméno objektu.

using

Pokud v programu používáme více jmen z jednoho prostoru jmen, nebo pokud používáme jedno jméno opakovaně, můžeme překladači říci, že „příjmení“ – jméno prostoru jmen – budeme vynechávat. To vyjádříme tzv. direktivou using, jež má tvar

```
using prostor_jmen;
```

Použijeme-li ji, můžeme program zlom1.cs přepsat takto:

```

/* Prostory jmen - soubor zlom4.cs */
using System;
class ŽlutýKůň
{
    public static void Main()
    {
        Console.WriteLine("Žluťoučký kůň");
        Console.WriteLine("příšerně úspěš!");
        Console.WriteLine("dábelské ódy.");
    }
}

```



Jeho zdrojový text najdete na WWW v adresáři Kap03\01.

3.2 Počítač at' počítá

Zatím jsme si hráli se znakovými řetězci. Zkusme nyní napsat program, který vypočte součet dvou čísel a vytiskne ho.

Proměnná K tomu ale budeme potřebovat místo v paměti, kam uložíme hodnoty obou sčítanců, a také místo na uložení výsledku; jinými slovy, budeme potřebovat proměnné. Abychom mohli proměnné používat, musíme počítači přikázat, aby je vytvořil, musíme je *deklarovat*.

Typ int K našim prvním pokusům použijeme celá čísla, pro něž se v jazyce C# používá datový typ `int`. Deklaraci proměnné typu `int` v C# můžeme zjednodušeně popsat takto:

`int jméno = hodnota;`

kde *jméno* je identifikátor deklarované proměnné a *hodnota* představuje hodnotu, kterou bude tato proměnná obsahovat hned po vytvoření. (Pokud na počáteční hodnotě nezáleží nebo pokud ji v okamžiku deklarace neznáme, můžeme tuto tzv. *inicializaci* vynechat.)

Nyní můžeme napsat např. takovýto program:

```

// Sčítání dvou čísel - soubor Součet.cs
using System;
class Součet
{
    static void Main()
    {
        int a = 55;    // První sčítanec
        int b = 66;    // Druhý sčítanec
        int c;         // Součet
        c = a + b;
    }
}

```

```

Console.WriteLine(
    "Součet čísel {0} a {1} je {2}", a, b, c);
}
}

```

Přifazení

V prvních dvou řádcích metody `Main()` deklarujeme proměnné `a`, resp. `b`, s počátečními hodnotami 55, resp. 66. Ve třetím řádku deklarujeme proměnnou `c`, které počáteční hodnotu neurčíme. (Později do ní uložíme součet `a+b`, takže na její počáteční hodnotě nezáleží.)

V následujícím řádku vypočteme součet `a+b` a jeho hodnotu uložíme do proměnné `c`. Operátor `=` zde tedy neznamená rovnost, ale *přifazení* – uložení hodnoty do proměnné.

Podívejme se na příkaz, kterým vypisujeme výsledek. Volání metody `WriteLine()` má nyní 4 parametry – znakový řetězec a proměnné `a`, `b` a `c`. Jednotlivé parametry jsou odděleny čárkami.

Formátovací řetězec

Znakový řetězec, který předáváme metodě `WriteLine()`, obsahuje kromě očekávaného textu také specifikace `{0}`, `{1}` a `{2}`, které určují místa, na nichž má tato metoda vypsat hodnoty dalších parametrů. Ve složených závorkách jsou pořadová čísla těchto parametrů (ovšem počítaná od 0), takže `{0}` znamená místo, kde vystoupí hodnota proměnné `a`, `{1}` představuje místo, kde se vypíše hodnota proměnné `b` atd. Po přeložení a spuštění vypíše tento program

Součet čísel 55 a 66 je 121



Deklarujeme-li několik proměnných stejného typu, můžeme všechny deklarace zapsat do jednoho řádku za jedno jméno typu; v předchozím příkladu jsme např. mohli napsat

```
int a = 55, b = 66, c;
```

Proměnnou jsme `c` ale nemuseli deklarovat. Místo toho jsme mohli zapsat výraz `a+b` jako čtvrtý parametr metody `WriteLine()`:

```
Console.WriteLine("Součet čísel {0} a {1} je {2}", a, b, a+b);
```



Zdrojový text tohoto programu po uvedených úpravách najdete na WWW v souboru `Kap03\02\Součet.cz`.

Reálná čísla

Pokud bychom chtěli sčítat také reálná čísla, samozřejmě můžeme. Datový typ, používaný pro reálná čísla, se v C# nazývá `double`. Stačí tedy v předchozím programu místo `int` psát všude `double`.

Reálná čísla můžeme v programu zapsat podobně jako v běžném životě, pouze místo desetinné čárky užijeme desetinnou tečku. Inicializace proměnných `a` a `b` proto může mít např. tvar

```
double a = 55.1, b = 66.2;
```




Zdrojový text takto upraveného programu najdete na WWW v souboru Kap03\02\Součet2.cz. Jestliže ho přeložíte a spustíte v českých Windows, vypíše

Součet čísel 55,1 a 66,2 je 121,3

Všimněte si, že program při výstupu použil lokálních zvyklostí, tj. v reálném čísle zapsal desetinnou čárku, nikoli tečku.

Metoda, která vypočte hodnotu

Naučili jsme se sčítat, podívejme se na další početní operace. Nikoho asi nepřekvapí, že pro odečítání se používá operátor `-`; ovšem pro násobení se používá operátor `*`, který musíme vždy¹⁵ zapsat, a pro dělení používáme operátor `/` (lomítko).

Druhá mocnina

Jazyk C# nemá žádný zvláštní operátor pro operaci umocňování, a proto si napíšeme metodu, která ho nahradí. Může vypadat takto:

```
// Metoda pro výpočet druhé mocniny celých čísel
static int Dvojmoc(int n)
{
    return n*n;
}
```

Deklarace této metody začíná opět klíčovým slovem `static`, neboť jde o statickou metodu (můžeme ji volat, i když neexistuje žádná instance dané třídy). Dále následuje slovo `int`, které říká, že tato metoda vypočte a vrátí celé číslo. `Dvojmoc` je identifikátor této metody; za ním v závorkách následuje specifikace parametrů této metody.

Formální parametr

Metoda `Dvojmoc()` má jeden *formální parametr* typu `int`, který se jmenuje `n`. Tento parametr představuje číslo, jehož druhou mocninu chceme spočítat.

Druhou mocninu čísla `n` představuje výraz `n*n`. Slovo `return`, které mu předchází, říká, že `n*n` je výsledek, který má metoda `Dvojmoc()` vrátit. Provedení příkazu `return` také způsobí, že se program z metody vrátí na místo, odkud jsme ji zavolali.

Celý program, který metodu `Dvojmoc()` také použije, může mít tvar

```
// Metoda pro výpočet druhé mocniny - soubor Dvoj.cs
class Test
{
    static int Dvojmoc(int n)
```

¹⁵ V matematice znak násobení – tečku nebo křížek – často vynecháváme a prostě zapíšeme oba činitele vedle sebe. To v C# nelze. Nemůžeme napsat např. `i j` nebo dokonce `i j`, musíme vždy napsat `i*j`.

```
{
    return n*n;
}

static void Main()
{
    int n = 12;
    System.Console.WriteLine(
        "Druhá mocnina čísla {0} je {1}", n, Dvojmoc(n)
    );
}
```

Volání metody

Mohli jsme ovšem také napsat

```
int n = 12;
int m = Dvojmoc(n);
```

a v metodě `WriteLine()` vypsat hodnotu proměnné `m`; výsledek by byl stejný – program by vypsal

Druhá mocnina čísla 12 je 144

Podívejme se blíže na zápis `Dvojmoc(i)`, který představuje *volání metody* `Dvojmoc()`. Tento zápis v programu způsobí, že počítač provede příkazy v těle této metody a přitom si za formální parametr `n` dosadí hodnotu, kterou mu při volání předáme – v tomto případě hodnotu uloženou v proměnné `i`. (Říkáme, že jsme metodu *zavolali se skutečným parametrem* `i`.) Potom na místo, kde jsme volání metody zapsali, dosadí výsledek.

Všimněte si také, že jsme při volání metody použili samotný identifikátor, zatímco při volání metody `WriteLine()` jsme k jejímu jménu museli připojit i jméno objektu, k němuž patří. *Při volání metody též třídy stačí uvádět samotný identifikátor.*

Zdrojový soubor `Dvoj.cs` najdete na WWW v adresáři Kap03\02.



Hlavička metody

Podívejme se nyní na první řádek deklarace metody `Dvojmoc()`, který má tvar

```
static int Dvojmoc(int n)
```

Říkáme mu *hlavička metody* a obsahuje všechny informace důležité pro její volání. Zde se např. dozvíme, že tato metoda

- se jmenuje `Dvojmoc`,
- je statická (klíčové slovo `static`).
- vrací výsledek typu `int`
- a při volání jí musíme předat jeden skutečný parametr typu `int`.

3.3 Základní vstup

Předchozí programy měly jednu zásadní vadu: Uměly sečíst nebo umocnit pouze čísla, která jsme zapsali do programu. Kdybychom ale vždy, když budeme chtít sečíst dvě čísla, měli přepisovat a znovu překládat celý program, asi by nás programování rychle přestalo bavit. Je zřejmé, že potřebujeme nástroje, které by programu umožnili číst za běhu vstupní data.

Problematické vstupy a výstupy v C# se budeme věnovat podrobně později; zatím se naučíme alespoň číst jednoduchá data ze standardního vstupu (tedy z klávesnice). Napíšeme třídu VV, která nám poskytne metody pro vstup celých a reálných čísel a znakových řetězců. Protože je budeme chtít volat, aniž bychom vytvářeli instance třídy VV, budou statické.

Než se pustíme do programování, musíme si říci několik věcí:

1. Třída `System.Console` obsahuje metodu `ReadLine()`. Tuto metodu voláme bez parametrů; přečte ze standardního vstupu – tedy z klávesnice – znakový řetězec a vrátí ho.
2. V programu v C# můžeme znakový řetězec uložit do instance knihovny třídy `string`.
3. Načtený znakový řetězec, který obsahuje zápis čísla, je třeba převést na číslo. K tomu nám poslouží statické metody `ToInt32()`, případně `.ToDouble()` knihovny třídy `System.Convert`.

Vyzbrojeni těmito vědomostmi již můžeme zkusit napsat třídu VV. Nejjednodušší bude metoda `InString()`, která načte a vrátí znakový řetězec:

```
using System;

static string InString() // Načte a vrátí řetězec
{
    return Console.ReadLine();
}
```

Metoda `InInt()`, která načte a vrátí celé číslo, bude trochu složitější:

```
static int InInt() // Načte a vrátí číslo typu int
{
    string s = InString();
    int i = Convert.ToInt32(s);
    return i;
}
```



Nejprve do proměnné s typem `string` uložíme řetězec¹⁶, který vrátí metoda `ReadLine()`. Pak zavoláme metodu `Convert.ToInt32(s)`, která se pokusí převést řetězec `s` na číselnou hodnotu; výsledek vrátíme.

Stručněji bychom tuto metodu mohli zapsat následujícím způsobem:

```
public static int InInt()
{
    return Convert.ToInt32(InString());
}
```

Podobně bude vypadat i metoda `InDouble()`, která bude číst a vracet hodnotu typu `double` (reálné číslo).

Celý výpis třídy najdete v následujícím oddílu; její zdrojový text je na WWW v souboru `Kap03\04\01\VV.cs`.

Poznamenejme, že metody `InInt()` a `InDouble()` požadují, abychom každé vstupující číslo zapsali na samostatnou řádku, a mají ještě řadu dalších nechtností. Později pochopitelně napíšeme poněkud lepší verzi třídy VV.

Vytváříme dynamickou knihovnu

Nyní můžeme tuto třídu používat. Jak?

První, co nás napadne, je okopírovat třídu VV do každého z programů, ve kterém budeme potřebovat něco přečíst. To je ale velice nešikovné – při kopírování můžeme na něco zapomenout, můžeme omylem použít starou verzi, která nebude dělat, co potřebujeme, může se stát řada dalších nepříjemností. Daleko lepší by bylo přeložit si tuto třídu jednou provždy a pak ji už jen používat jako svou vlastní knihovnu.

Abychom ovšem mohli třídu VV používat i z jiných programů, musíme v ní provést několik drobných úprav: Musíme říci, že třídu VV jako celek a její jednotlivé metody smí používat kdokoli. K tomu poslouží slovo `public`, které napíšeme před slovo `class` a před deklaraci metody. Po této úpravě bude třída VV vypadat takto:

```
// Soubor VV.cs - základní prostředky pro vstup
using System;

public class VV
{
    public static string InString()
    {
        return Console.ReadLine();
    }
}
```

¹⁶ Pozor, ve skutečnosti je situace složitější: Do proměnné `s` se uloží pouze odkaz na instanci typu `string`. O tom budeme podrobně hovořit později.

```

public static int InInt()
{
    /*string s = InString();      // Tyto tři řádky
    int i = Convert.ToInt32(s);   // lze přepsat stručněji
    return i;*/                  // takto:
    return Convert.ToInt32(InString());
}

public static double InDouble()
{
    return Convert.ToDouble(InString());
}
}

```

Nyní tento soubor přeložíme. Tentokrát ale nechceme vytvořit spustitelný soubor¹⁷, nýbrž knihovnu, a proto použijeme v příkazové řádce přepínač `/t:library`. Program `csc` tedy zavoláme příkazem

```
csc /t:library VV.cs
```

Překladem vznikne soubor `VV.dll`, který představuje tzv. dynamickou knihovnu.

Použití dynamické knihovny

Nyní zkusíme naši knihovnu použít. Protože nám nejde o nic jiného než o vyzkoušení funkčnosti, napíšeme jednoduchý program, který přečte ze standardního vstupu číslo a vypíše jeho dvojnásobek:

```

// Test knihovny - soubor pokus.cs

using System;

class Test {
    static void Main()
    {
        Console.WriteLine("zadej číslo: ");
        int i = VV.InInt();
        Console.WriteLine("Jeho dvojnásobek je "+2*i);
    }
}

```



Tento soubor najdete na WWW v adresáři `Kap03\03`.

Při překladu ovšem narazíme na problém: Třidu `VV` překladač nezná a požaduje, abychom mu řekli, kde ji najde. K tomu nám poslouží přepínač `/r:`, za kterým uvedeme jméno knihovny (přesněji jméno seskupení), jež obsahuje třídu `VV`. Příkaz k překladu pak bude mít tvar

```
csc /r:VV.dll pokus.cs
```

¹⁷ To by ani nešlo, neboť třída v tomto souboru neobsahuje metodu `Main()`.

Příponu `.dll` ve jménu souboru s dynamickou knihovnou nesmíme vynechat. Po přeložení a spuštění tohoto programu se může odehrát následující konverzace:

```

K:\prog\Kap03\04>pokus
zadej číslo:
65
Jeho dvojnásobek je 130

```



Pokud se vám ale stane, že na výzvu zadej číslo napíšete něco, co nepředstavuje celé číslo, program skončí chybou. I to později napravíme.



Pozor: Pracujeme-li pod českými Windows, musí reálná čísla, která chceme číst pomocí metody `InDouble()`, obsahovat desetinnou čárku, nikoli desetinnou tečku.



Takto lze použít pouze dynamické knihovny pro .NET, které si vytvoříme v C# pomocí přepínače `/t:library` a které obsahují řízený kód. Dynamické knihovny, které obsahují neřízený kód (běžné dynamické knihovny, vytvořené v jazyce C nebo C++), používáme jiným způsobem; povíme si o něm později.

3.4 Ještě trochu počítání

Když už umíme číst data ze standardního vstupu, zkusíme toho využít. Naprogramujeme metodu, která vypočítá faktoriál zadaného celého čísla n , tedy součin čísel od 1 do n . To nám umožní seznámit se s programováním cyklu a podmínky a s některými dalšími operátory.

Cyklus

Faktoriál celého čísla n , označovaný $n!$, vypočteme pro $n > 1$ jako součin $1 \times 2 \times \dots \times n$. Faktoriál čísel 0 a 1 je podle definice roven 1, tedy $0! = 1$, $1! = 1$. Pro záporná čísla není faktoriál definován.

Výpočet faktoriálu pro hodnoty 0 a 1 tedy nebude představovat problém. Ovšem pro $n > 1$ potřebujeme spočítat součin $1 \times 2 \times \dots \times n$. Počet činitelů v něm předem neznáme a jazyk C# nám nedovolí použít ve výrazu tři tečky, jak se to někdy dělá v matematice. Co s tím?

Algoritmus
výpočtu

Můžeme zkusit následující postup (proměnná n zde obsahuje hodnotu, jejíž faktoriál počítáme):

1. Vytvoříme si pomocnou proměnnou s a uložíme do ní hodnotu 1.
2. Je-li $n < 2$, skončíme. Proměnná s obsahuje výsledek.
3. Vynásobíme $s \times n$ a výsledek uložíme zpět do s .
4. Zmenšíme hodnotu n o 1.

5. Vrátime se na krok 2.

Kroky 3 až 5 opakujeme, dokud je splněna podmínka $n > 1$. (Takovéto opakování se obvykle označuje jako *cyklus*.)

Obsahuje-li proměnná n hodnotu 0 nebo 1, tento algoritmus skončí, aniž kroky 3 až 5 proběhnou. Proměnná s proto bude obsahovat správný výsledek.

Obsahuje-li proměnná n celé číslo větší než 1, uloží se do s ve 3. kroku součin $s*n$ (tedy n , neboť s obsahuje na počátku 1). Pak se obsah n zmenší o 1, takže v s postupně vznikne součin $n*(n-1)*\dots*2$, což je požadovaný $n!$

Příkaz while Takovéto opakování kroků 3 až 5 lze naprogramovat pomocí příkazu `while`. (Hovoříme také o *cyklu while*.) Jeho tvar je

while(podmínka){příkazy}

a způsobí, že program bude provádět *příkazy* uzavřené ve složených závorkách, dokud je splněna *podmínka*. Přesněji:

- Program nejprve vyhodnotí *podmínku* (zjistí, zda je splněna).
- Pokud není, příkaz `while` skončí. To znamená, že *příkazy*, uzavřené mezi složenými závorkami (nazývané také „tělo cyklu“), se přeskočí a program bude pokračovat příkazem, který následuje za nimi.
- Je-li *podmínka* splněna, provedou se *příkazy* uzavřené mezi složenými závorkami. Pak se znovu vyhodnotí *podmínka*, a není-li splněna, příkaz `while` skončí atd.

Podmínky, relační operátory

Podmínka je výraz, který je buď *pravdivý* (říkáme, že podmínka je splněna) nebo *nepravdivý* (říkáme, že podmínka není splněna). Nejčastěji obsahuje porovnání hodnot proměnných pomocí tzv. relačních operátorů $>$, $<$, $<=$ nebo $>=$, které znamenají „je menší“, „je větší“, „je menší nebo rovno“ nebo „je větší nebo rovno“. Vedle toho můžeme použít operátor $=$, který vyjadřuje relaci „rovná se“, a operátor $!=$, který znamená „nerovná se“.

Typ bool

Doplňme, že podmínka představuje v C# hodnotu typu `bool`, který slouží k ukládání logických hodnot. Hodnota „pravda“ se vyjadřuje slovem `true`, hodnota „nepravda“ slovem `false`.

Nyní se můžeme pokusit naprogramovat metodu `f()`, která bude faktoriál počítat¹⁸:

¹⁸ Připomeňme si, že operátor $=$ nevyjadřuje v C# rovnost, ale přiřazení. Příkaz

```
static int f(int n)
{
    int s = 1;           // Pomocná proměnná, obsahuje výsledek
    while(n > 1)          // Opakuji, dokud je n > 1
    {
        s = s*n;          // Do s ulož s*n
        n = n - 1;        // Zmenši hodnotu n o 1
    }
    return s;            // s obsahuje výsledek
}
```

Zavoláme-li tuto metodu např. zápisem `f(3)`, bude proměnná (formální parametr) n obsahovat hodnotu 3. Na počátku se do pomocné proměnné s uloží hodnota 1.

Pak se vyhodnotí podmínka $n > 1$ v příkazu `while`. Ta bude splněna, a proto se tělo tohoto cyklu provede: Do s se uloží $s*n$, tedy 3. Dále se hodnota, uložená v n , zmenší o 1; proměnná n tedy bude obsahovat hodnotu 2.

Nyní se znovu vyhodnotí podmínka. Protože n obsahuje hodnotu 2, je splněna, a proto se tělo cyklu znovu provede: Do s se uloží $s*n$, tedy 6, do n hodnota 1 a znovu se vyhodnotí podmínka. Nyní již není splněna, neboť $n == 1$, a proto se tělo cyklu přeskočí. Program přejde na příkaz `return` a jako výsledek vrátí hodnotu 6, uloženou v s .

Na WWW v adresáři `Kap03\04` najdete soubor `Fakt1.cs`, který obsahuje třídu `Výpočet` používající metodu `f()`.

Užitečné operátory

I když náš program funguje, lze s trochou nadsázky lze tvrdit, že by ho takto žádný zkušený programátor nenapsal. Podívejme se proto, jak bychom ho mohli upravit. Výraz

```
s = s*n;
```

Operátory op=

obsahuje proměnnou s na obou stranách přiřazovacího operátoru, a pro tuto situaci nabízí C# zkratku

```
s *= n;
```

- Podobně můžeme v C# zkracovat i výrazy obsahující operátory $+$, $-$, $/$ a některé další. Je-li op některý z uvedených operátorů, znamená zápis

```
x op= y;
```

vždy totéž co

$n=n-1$; tedy znamená „vezmi hodnotu uloženou v proměnné n , zmenši ji o 1 a výsledek ulož zpět do proměnné n “.

```
x = x op y;
```

Ve skutečnosti ovšem nejde jen o úsporu při psaní: Použití těchto operátorů umožňuje překladači také vytvořit rychlejší program. S použitím těchto operátorů můžeme přepsat cyklus pro výpočet faktoriálu do tvaru

```
while(n > 1)           // Opakuji, dokud je n > 1
{
    s *= n;             // Totéž co s = s*n
    n -= 1;             // Totéž co n = n - 1
}
```

Operátory

++ a --

Ovšem zvětšení nebo zmenšení hodnoty proměnné o 1 je natolik časté, že pro to jazyk C# zavádí zvláštní operátory ++ a --.

Tyto operátory můžeme zapsat buď před proměnnou, na kterou má působit (pak hovoříme o *prefixovém* operátoru), nebo za ni (pak hovoříme o *postfixovém* operátoru). Postfixový operátor způsobí, že se nejprve použije *nezměněná* hodnota proměnné a teprve pak se tato hodnota zvětší nebo zmenší. Prefixový operátor způsobí, že se nejprve změní hodnota proměnné, a tato změněná hodnota se pak použije.

Příkaz

```
n -= 1;
```

v metodě `f()` proto můžeme nahradit příkazem

```
n--;
```

nebo

```
--n;
```

V tomto případě nezáleží na tom, zda použijeme prefixový nebo postfixový operátor. Jediné, co potřebujeme, je zmenšit o 1 hodnotu uloženou v proměnné `n`.

V předchozím výkladu jsme si ale řekli, že použijeme-li ve výrazu postfixový operátor --, vypočte program hodnotu výrazu z původní hodnoty proměnné a teprve pak tuto hodnotu změní. Proto můžeme cyklus v metodě `f()` přepsat takto:

```
while(n > 1)
{
    s *= n--;
}
```

Možná vám to na první pohled připadá složité a zbytečné. Programovací jazyky, podobně jako jazyky přirozené, mají ale své vžitě frázové idiomy – a patří k hrdosti programátorů je znát a používat. Zdrojový text programu s takto upravenou metodou `f()` najdete na WWW v souboru `Kap03\04\fakt2.cs`.



Podmínka

Metoda `f()` tedy umí počítat faktoriály celých kladných čísel. Ovšem zadáme-li ji jako skutečný parametr např. číslo -1, vrátí 1, i když pro záporná čísla není faktoriál definován. To není dobré, a proto se to pokusíme napravit.

Abychom mohli určit, co má program dělat, v závislosti na tom, zda je splněna určitá podmínka, používáme v C# příkaz `if`. Jeho zjednodušený popis je

```
if(podmínka) {příkazy_1}
```

nebo

```
if(podmínka) {příkazy_1} else {příkazy_2}
```

(První možnost označujeme jako *neúplné if*, druhou jako *úplné if*.) Za slovem `if` je v závorkách zapsána *podmínka*. Pak následuje skupina příkazů ve složených závorkách, zde označená *příkazy_1*. Pak může ještě následovat slovo `else` a za ním další skupina příkazů ve složených závorkách (*příkazy_2*). Význam tohoto příkazu je:

- Program nejprve vyhodnotí *podmínku*.
- Je-li *podmínka* splněna, provede *příkazy_1*. V případě, že jsme použili variantu s `else`, přeskočí *příkazy_2*.
- Není-li *podmínka* splněna, přeskočí *příkazy_1*. Jestliže jsme použili variantu s `else`, provede *příkazy_2*, jinak neprovede nic a přejde na příkaz následující za příkazem `if`.

Vyzbrojení tímto novým poznatkem se můžeme pustit do úprav předchozího programu. První, co nás může napadnout, je testovat před každým voláním, zda náhodou nepředáváme funkci `f()` záporný parametr. Upravená metoda `Main()` může mít tvar

```
static void Main()
{
    Console.WriteLine("Zadej celé nezáporné číslo: ");
    int n = VV.InInt();
    if(n >= 0)
    {
        Console.WriteLine("Faktoriál čísla {0} je {1}.",
                           n, f(n));
    }
    else
    {
        Console.WriteLine("Zadal jsi {0}. + "
                           " Faktoriál záporného čísla není definován.", n);
    }
}
```



Zdrojový text programu po této úpravě najdete na WWW v souboru Kap03\04\Fakt3.cs.

Indikace chyby

I když takto upravený program funguje a v případě zadání záporného čísla vypíše místo výsledku upozornění, není použitý postup dobrý. Pokud bychom totiž chtěli metodu `f()` používat v programu na více místech, museli bychom mít neustále na paměti, že pro záporná čísla nefunguje správně. Rozumnější proto bude upravit definici metody `f()` tak, aby v případě chybné hodnoty skutečného parametru nevracela nesmyslný výsledek.

Jedním z často používaných programátorských triků je vrátit v takovém případě hodnotu, která signalizuje chybu. Faktoriál je vždy kladné číslo, a proto by mohla metoda `f()` v případě záporného parametru vracet např. hodnotu 0:

```
static int f(int n)
{
    if(n < 0) return 0;      // Stačí neúplné if
    int s = 1; // Pomocná proměnná, obsahuje výsledek
    while(n > 1)
    {
        s *= n--;
    }
    return s;
}
```

Dostane-li metoda `f()` nezáporný parametr, nebude podmínka v příkazu `if` splněna. Program tedy přeskočí příkaz `return` v `if`, vypočte hodnotu faktoriálu a vrátí ji. Předáme-li této metodě `f()` záporný parametr, provede se příkaz `return 0` v `if` a program se z metody `f()` ihned vrátí. Cyklus `while` se tedy neprovede.



Tuto verzi programu najdete na WWW v souboru kap03\04\Fakt4.cs.

Výjimky

Skutečnost, že metoda `f()` indikuje chybnou hodnotu parametru, znamená určitě pokrok, ovšem nijak velký. Po každém volání metody `f()` totiž budeme muset kontrolovat, zda náhodou nevrátila 0. Metoda `Main()` našeho programu pak bude mít tvar

```
// Když f() indikuje chybu vrácenou hodnotou
static void Main()
{
    Console.WriteLine("Zadej celé nezáporné číslo: ");
    int n = VV.InInt();
    int m = f(n);
```

```
if(m != 0) {
    Console.WriteLine("Faktoriál čísla {0}"
        + " je {1}.", n, m);
}
else
{
    Console.WriteLine("Zadal jsi {0}." +
        " Faktoriál záporného čísla není definován.", n);
}
}
```

Používání metody `f()` se tím nijak nezjednodušilo. Navíc metody, které vracejí hodnoty (říká se jim také *funkce*), často používáme ve složitějších výrazech, jako je

```
x = (f(n)-f(n-1))/f(n-2);
```

Pokud bychom opravdu chtěli využít schopnosti této funkce indikovat chybu pomocí vrácené hodnoty, museli bychom nejprve vypočítat `f(n)`, vrácenou hodnotu uložit do pomocné proměnné a zkontrolovat, zda nedošlo k chybě, potom udělat totéž s `f(n-1)` a s `f(n-2)`. Program by pak byl nepřehledný, jeho zápis by neodpovídal běžnému zápisu řešeného problému.

Co je to výjimka

Proto moderní programovací jazyky používají v takovémto případě mechanismus *výjimka*. V tomto úvodu si ho popíšeme jen velmi zjednodušeně; nemůžeme se mu ale vyhnout, protože mnohé metody tříd z knihoven prostředí .NET výjimky používají.

Výjimka je chyba, kterou nedokáže daná metoda zvládnout. Typickým příkladem může být právě zadání špatného skutečného parametru (záporného čísla) metodě `f()` pro výpočet faktoriálu. Tato chyba může být důsledkem špatně zadané vstupní hodnoty, může ale mít i řadu jiných příčin a metoda `f()` nemá k dispozici dostatek informací, aby si s ní poradila. Proto *vyvolá výjimku*. To znamená, že nejprve vytvoří speciální objekt, který ponese informace o této chybě, a pak způsobí, že program přestane provádět příkazy v této metodě a bude hledat kód, který výjimku *ošetří* – tzv. handler. Pokud jej nenajde, program vypíše chybové hlášení a předčasně skončí.

throw

K vyvolání výjimky slouží příkaz `throw`, který budeme zatím používat pouze ve tvaru

throw new System.Exception(zpráva);

`System.Exception` je třída objektu, který ponese informace o výjimce; to znamená, že identifikátor `System` můžeme vynechat, použijeme-li předtím příkaz `using System`;

Zpráva je znakový řetězec s informací o chybě. (V následujícím od-
dílu této kapitoly se dozvíme, že zápisem `new System.Exception`
`on(zpráva)` vlastně vytváříme instanci třídy `System.Exception`.)

Upravíme tedy metodu `f()` do tvaru

```
static int f(int n)
{
    if(n < 0) throw
        new Exception("Faktoriál není definován");
    int s = 1; // Pomocná proměnná, obsahuje výsledek
    // ... a dále stejně jako předtím ...
}
```

Dostane-li tato metoda `f()` jako parametr nezáporné číslo, vypočte
jeho faktoriál. Dostane-li ale záporné číslo, proběhne příkaz `throw` a
program skončí chybovým hlášením¹⁹

```
Unhandled Exception: System.Exception: Faktoriál není
definován
   at Výpočet.f(Int32 n)
   at Výpočet.Main()
```

Tato zpráva nám oznamuje, že vznikla neošetřená výjimka typu `System`
`Exception`, a to v metodě `f()` třídy `Výpočet`, kterou jsme volali
v metodě `Main()` třídy `Výpočet`.

Náš program nyní při špatně zadaném vstupu vypíše zprávu o chybě a
skončí. To je skvělé při ladění, ale skutečný program se takto chovat
nemůže; my se s tím však zatím spokojíme. Později se samozřejmě
naučíme výjimky zachycovat a ošetřovat.

Tento program najdete na WWW v souboru `Kap03\04\Fakt5.cs`.

Při programování třídy `WV` jsme použili metody `Convert.ToInt32()` a
`Convert.ToDouble()`. Předáme-li těmto metodám jako skutečný para-
metr znakový řetězec, který nepředstavuje celé, resp. reálné číslo,
vyvolají také výjimku.

Magická čísla

Po chvíli experimentů s výpočty faktoriálu zjistíme, že pro hodnoty
větší než 12 nedává náš program správné výsledky. Důvod je jedno-
duchý: vypočtená čísla jsou příliš velká a nevejdou se do paměti vy-
hrazené pro proměnnou typu `int`.

¹⁹ Pokud máte zároveň nainstalováno i Visual Studio .NET, nejspíš se objeví
dialogové okno, které nám nabídne, že bude daný program ladit. Pro nás ale
nemá význam a proto ho klidně zavfete.

To ale znamená, že by metoda `f()` měla programátora, který ji zavolá
s parametrem větším než 12, upozornit, že je něco špatně; nejlépe opět
vyvolat výjimku. Nic nám samozřejmě nebrání upravit začátek této
metody do tvaru

```
static int f(int n)
{
    // Trik s řetězcí vysvětlíme o několik odstavců dále
    if(n < 0) throw new Exception("Faktoriál "+ n +
        " není definován");
    if(n > 12) throw new Exception("Příliš velké číslo: " + n);
    /* ... a dále stejně jako předtím ... */
}
```

Jenže takto zapsaná podmínka představuje opravdu skvělý způsob, jak
si v budoucnu přidělat problémy. V tuto chvíli sice dobře víme, co
číslo 12 znamená, ale až se k programu po nějaké době vrátíme, aby-
chom ho opravili nebo upravili, budeme muset zdoluhavě přemýšlet
nad jeho významem. Co když se později rozhodneme použít k výpo-
čtu faktoriálu jiný datový typ s větším rozsahem a budeme chtít tuto
konstantu změnit? V programu se totiž může vyskytovat více dvanác-
tek s různým významem a my budeme muset nad každou přemýšlet,
zda je to ta, kterou chceme změnit, nebo nějaká jiná.

Takovýmto přímo zapsaným konstantám (literálům) říkají programá-
toři „magická čísla“ a pokud možno se jim vyhýbají. Místo toho pou-
žívají pojmenované konstanty, které můžeme v C# deklarovat pomocí
slova `const`, např.

```
const int Mez = 12;
```

V dalším programu pak používáme identifikátor `Mez`; při jakýchkoli
úpravách stačí změnit deklaraci této konstanty, nemusíme hledat její
použití v programu.

Deklarace třídy `Výpočet` tedy může začínat

```
// Soubor Kap03\04\fakt6.cs na WWW
class Výpočet
{
    const int Mez = 12;
    static int f(int n)
    {
        if(n < 0) throw new Exception("Faktoriál "+ n +
            " není definován");
        if(n > Mez) throw new Exception("Příliš velké číslo: "
            + n);
        // ... dále jako předtím ...
    }
}
```

Povšimněte si také zápisu `"Příliš velké číslo: " + n`. Operátor `+`
můžeme použít pro spojení řetězce s hodnotou jiného typu. Program
nejprve převede hodnotu jiného typu – zde celé číslo `n` – na znakový
řetězec a pak oba řetězce spojí.

3.5 Univerzální nápis

Až dosud jsme programovali třídy, které obsahovaly pouze statické metody a nevyžadovaly vytváření instancí. To ale není v objektových jazycích příliš obvyklé, a proto se v tomto oddílu naučíme pracovat s instancemi.

Třída Text

Zatím jsme vypisovali text zadaný pomocí řetězcových konstant. Nyní se pokusíme zařídit, abychom mohli tento text programově měnit.

K ukládání znakových řetězců používáme v C# knihovni třídu `string`. Mohli bychom tedy v hlavní (a zatím jediné) třídě našeho programu definovat datovou složku typu `string`. Protože však budeme s nápisem chtít manipulovat i způsobem, který třída `string` neumožňuje, vytvoříme si pro nápis zvláštní třídu, kterou nazveme `Text`.

Tato třída bude obsahovat datovou složku `nápis` typu `string`, do níž budeme ukládat požadovaný text; a protože pravidla objektového programování říkají, že s datovými složkami by neměl zacházet nikdo jiný než metody dané třídy, deklarujeme ji jako soukromou – s přístupovými právy `private`.

```
private string nápis;
```

O změnu uloženého textu se postará metoda `NastavNápis(string)`, která bude mít jeden parametr typu `string`; to bude řetězec, který chceme uložit do složky `nápis`. Tato metoda bude veřejně přístupná (`public`).

```
public void NastavNápis(string nový)
{
    nápis = nový;
}
```

Dále budeme potřebovat metodu, která `nápis` vypíše na obrazovku. Nazveme ji `Vypis()`, bude také veřejně přístupná a nebude mít žádné parametry. Naprogramovat ji jistě dokážete sami (a pokud ne, najdete ji o pár řádků dále).

Konstruktor Zbývá ještě jeden problém: Jaká data – jaký nápis – bude instance obsahovat ihned po vytvoření? O to se v C# stará zvláštní metoda zvaná *konstruktor*.

Konstruktor se musí jmenovat stejně jako třída, jejíž instance vytváří. Konstruktor třídy `Text` se tedy musí jmenovat `Text()`. V jeho deklaraci neuvádíme typ vrácené hodnoty. Deklarace konstruktoru třídy `Text` může vypadat např. takto:

```
public Text(string sdělení)
{
    NastavNápis(sdělení);
}
```

Tento konstruktor má jediný parametr typu `string` a jeho úkolem je uložit do instance hned po jejím vytvoření zadaný řetězec.

Při pohledu na tento konstruktor můžete namítnout, že by bylo vhodnější místo volání metody `NastavNápis()` použít v těle konstruktoru příkaz

```
nápis = nový;
```

kteřý udělá totéž, ale bude efektivnější. Zatím máte nepochybně pravdu; ale co když se později rozhodneme text, ukládaný do instance, nějak upravovat – např. cenzurovat? Co když budeme chtít zkontrolovat, zda je vhodný pro malé děti? V takovém případě bychom museli kontrolu textu programovat na dvou místech, v konstruktoru a v metodě `NastavNápis()` – a to nikdy není dobré. Při pozdějších úpravách snadno na některé z míst, které chceme změnit, zapomeneme, a problémy budou na světě. Jednou z dobrých programátorských zásad je *neprogramovat totéž dvakrát*, a proto jsme i my v konstruktoru zavolali metodu `NastavNápis()`, místo abychom programovali uložení textu znovu.

Občas se stane, že pro svou třídu potřebujeme více konstruktorů. Ani to není problém: jazyk C# podporuje tzv. *přetěžování metod*. To znamená, že umožňuje definovat v jedné třídě více metod se stejným identifikátorem, pokud se liší počet nebo typ jejich parametrů (nebo obojí). Nic nám tedy nebrání definovat ještě konstruktor bez parametrů, který uloží do instance jakýsi implicitní text:

```
public Text() // Konstruktor
{
    NastavNápis("Jak tak na to koukám");
}
```

Nyní už si můžeme ukázat zdrojový text celé třídy `Text`:

```
using System;
class Text
{
    private string nápis; // Obsahuje nápis

    // Nastaví nový nápis
    public void NastavNápis(string nový)
    {
        nápis = nový;
    }

    public Text(string sdělení) // Konstruktor
    {
```

```

        NastavNápis(sdělení);
    }

    public Text()           // Konstruktor bez parametrů
    {
        NastavNápis("Jak tak na to koukám");
    }

    public void Vypis()     // Vypíše nápis
    {
        Console.WriteLine(nápis);
    }
}

```

Vytvoření instance

Nyní napíšeme program, který třídu `Text` použije. Než se do toho ale pustíme, musíme si říci, jak se v C# vytvářejí instance tříd.

Odkaz S instancemi pracujeme pomocí tzv. odkazů (referencí). Deklarace

```
Text t;
```

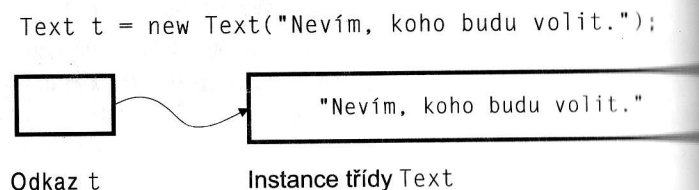
nevytvoří instanci třídy `Text`, ale odkaz na ni. Tento odkaz ovšem zatím na žádnou instanci neodkazuje. K vytvoření nové instance nám v C# slouží operátor `new`, za nímž zapíšeme volání konstruktoru. Na příklad takto²⁰:

```
new Text("Nevím, koho budu volit.");
```

Operátor `new` vrací odkaz na nově vytvořenou instanci, který můžeme přiřadit například²¹ proměnné `t`:

```
t = new Text("Nevím, koho budu volit");
```

Schématicky to znázorňuje obr. 3.1.



Obr. 3.1 Instance a odkaz na ni

²⁰ To je stejné jako v Javě. Pokud Javu neznáte, ale znáte C, C++ nebo Pascal, stačí vám říci, že jde vlastně o ukazatel na instanci. Všechny instance objektových typů jsou v C# dynamické.

²¹ `t` je proměnná typu *odkaz na instanci typu* `Text`.

Když máme odkaz na instanci, můžeme volat její nestatické metody. To už ale vlastně známe – k identifikátoru odkazu připojíme tečkou jméno metody. Budeme-li tedy chtít vypsat text uložený v instanci, na níž odkazuje `t`, zavoláme metodu `Vypis()` příkazem

```
t.Vypis();
```

Volání metod

Zatím jsme se setkali s několika způsoby volání metod. Abychom v tom měli jasno, shrňme si pravidla, která pro ně platí.

- Při volání nestatické metody zapíšeme nejprve identifikátor instance, pro níž tuto metodu voláme. K němu pak tečkou připojíme identifikátor metody. (Říkáme, že jméno metody *kvalifikujeme* odkazem na instanci.)
- Při volání statické metody napíšeme jméno třídy a k němu tečkou připojíme jméno metody. (Jméno metody kvalifikujeme jménem třídy.)
- Voláme-li z nestatické metody jinou nestatickou metodu téže třídy pro tutéž instanci, kvalifikaci vynecháme. Podobně můžeme kvalifikaci vynechat i při volání statické metody z jakékoli metody téže třídy.

Podívejme se znovu na konstruktor třídy `Text()`:

```
public Text(string sdělení) { NastavNápis(sdělení); }
```

V jeho těle voláme metodu `NastavNápis()` pro právě vytvářenou instanci, tedy pro instanci, s níž tento konstruktor pracuje. Proto jsme volání této metody nemuseli nijak kvalifikovat.

Program

Nyní se konečně můžeme podívat na třídu `Program`, která použije třídu `Text`.

```

class Program
{
    static void Main()
    {
        Text u = new Text();
        u.Vypis();
        Text t = new Text("Nevím, koho budu volit");
        t.Vypis();
        t.NastavNápis("A do voleb na to asi nepřijdu.");
        t.Vypis();
    }
}

```



Zdrojový text celého programu najdete na WWW v souboru `Kap03\05\Text.cs`. Když ho přeložíte a spustíte, vypíše

```
Jak tak na to koukám...
```

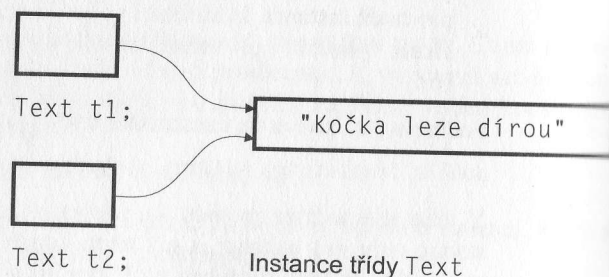

Nevím, koho budu volit.
A do voleb na to asi nepřijdu.

Z prvních dvou řádků tohoto výpisu je vidět, že každá z instancí má svou vlastní datovou složku nápis.

Přiřazování odkazů

Odkazy na instance lze samozřejmě přiřazovat proměnným odpovídajícího typu. Přitom ale musíme mít na paměti, že přiřazujeme odkazy, nikoli instance. Pozměňme metodu `Main()` z předchozího příkladu:

```
static void Main()
{
    Text t1 = new Text("Kočka leze dírou");
    Text t2 = t1;
    t1.Vypis();
    t2.NastavNapis("pes oknem");
    t1.Vypis();
}
```



Obr. 3.2 Po přiřazení ukazují dva odkazy na tutéž instanci

Nejprve vytvoříme novou instanci třídy `Text`, odkaz na ni uložíme do proměnné `t1` a hned také do `t2`. Poté vypíšeme text z instance, na níž odkazuje `t1`. Pak zavoláme metodu `NastavNapis()`, pomocí níž změníme text uložený v instanci, na níž odkazuje `t2`. Nakonec znovu vypíšeme text z instance, na níž odkazuje `t1`. Dostaneme

Kočka leze dírou
pes oknem

To znamená, že prostřednictvím odkazu `t2` jsme změnili data instance, na níž odkazuje `t1`. Situaci ukazuje obr. 3.2.



Zdrojový text našeho programu po této úpravě najdete na WWW v souboru `Kap03\05\Text2.cs`.

Doplňme, že proměnné představující odkazy mohou také obsahovat speciální hodnotu, kterou v programu vyjádříme slovem `null`. Tato hodnota říká, že daný odkaz neodkazuje na žádný platný objekt.

Automatická správa paměti

Nejspíš vás už napadlo, co se stane, když instance zanikne. Kdo uvolní paměť, kterou pro ni vyhradil operátor `new`? Jak to, že jsme se o to v našich dosavadních programech nestarali?

Jednou z výhod programovacího jazyka C# je, že se o to starat nemusíme. Uvolnění paměti po nepotřebných instancích zařídí v prostředí .NET automatická správa paměti, často označovaná anglickým termínem *garbage collector*²². Automatická správa paměti si vede evidenci odkazů, a když zjistí, že na nějaký objekt neexistuje žádný odkaz, zruší ho. Obvykle k tomu nedojde hned, ale až ve chvíli, kdy se to hodí; nemusí k tomu za běhu programu dojít vůbec.

Automatická správa paměti zpravidla znamená určité zpomalení běhu programu. Na druhé straně zabrání řadě chyb v práci s pamětí, které dobře znají programátoři používající C, C++ nebo Pascal, jež automatickou správu paměti standardně neposkytují.

Jak víme, je `string` jedna z předdefinovaných tříd v C#; ve skutečnosti je `string` jiné jméno – alias – pro třídu `System.String`. To ale znamená, že také instance této třídy se vytvářejí dynamicky. Jak je tedy možné, že jsme o tom dosud nemluvili?

Odpověď zní – protože to zatím nebylo třeba. Jediné, co jsme v této a v předchozí kapitole potřebovali, byly řetězcové konstanty a přiřazování odkazů. Z toho plyne, že složka `Napis` třídy `Text` představuje také odkaz na instanci třídy `string`, nikoli samotný `string`, a přiřazení

`napis = nový;`

v metodě `NastavNapis(string nový)` znamená přiřazení odkazu, nikoli kopírování instance. Podobně i parametr `nový` představuje odkaz na instanci.

Podívejme se nyní podrobněji, co se stane, když zavoláme metodu `NastavNapis()` příkazem

```
t1.NastavNapis("pes oknem");
```

1. Nejprve se vytvoří nová instance třídy `string`, obsahující řetězcovou konstantu `"pes oknem"`, a odkaz na ni se předá jako parametr nové metodě `NastavNapis()`. (Uloží ho do parametru `nový`.)
2. Pak se datové složce `napis` v instanci `t1` přiřadí odkaz, obsažený v parametru `nový`.

²² Doslova *sběrač smetí*. Američané nechodí pro termíny daleko a mají v tom naprostou pravdu. V literatuře najdeme ovšem i *Automatic memory management*.

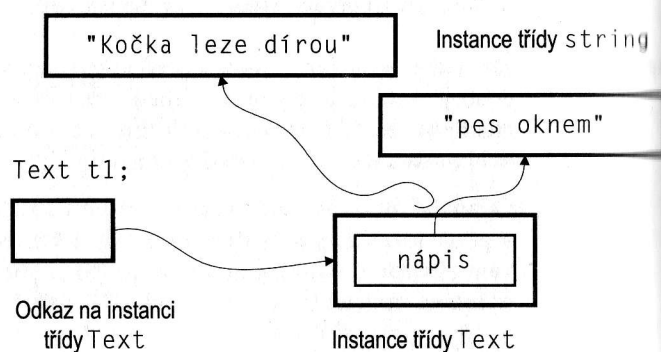
3. Tím zanikne odkaz na instanci třídy `string`, na niž předtím odkazovala datová složka `nápis`. Pokud na tuto instanci neexistuje žádný odkaz, zanikne – automatická správa paměti ji při první vhodné příležitosti odstraní.

Situaci ukazuje obrázek 3.3.

Na závěr ještě poznamenáme, že jazyk C# dovoluje také napsat

```
string s = "Nebude-li pršet, nezmoknem";
```

Význam už jistě uhodnete sami: Vytvoří se instance, která bude obsahovat daný text, a odkaz na ni se uloží do `s`. Poznamenejme, že tato konstanta "Nebude-li pršet, nezmoknem" také představuje instanci typu `string`.



Obr. 3.3 Metoda `NastavNápis()` změnila text uložený v instanci

Složitější příklady

V předchozích kapitolách jsme se seznámili s několika základními konstrukcemi, používanými v C#. V této kapitole se pustíme do trochu složitějších příkladů, na nichž uvidíme další možnosti tohoto jazyka. Mimo jiné se seznámíme i s kontejnery – třídami, jejich instance mohou sloužit k ukládání libovolného množství dat.

4.1 Řazení slov

Náš první úkol bude alespoň na pohled jednoduchý: Napíšeme program, který přečte ze standardního vstupu text rozdělený na řádky, vyhledá v něm různá slova, seřadí je podle abecedy a výsledek vytiskne. Pro jednoduchost budeme předpokládat, že text neobsahuje čárky, tečky a jiná interpunkční znaménka a různé gramatické tvary téhož slova budeme pokládat za různá slova.

Než se pustíme do programování, měli bychom si ujasnit, jak budeme postupovat. Náš program bude číst text po řádcích ze standardního vstupu – nic víc totiž zatím neumíme. Každý přečtený řádek pak rozloží na jednotlivá slova a ta si zapamatuje. Nakonec zapamatovaná slova seřadí podle abecedy a vytiskne.

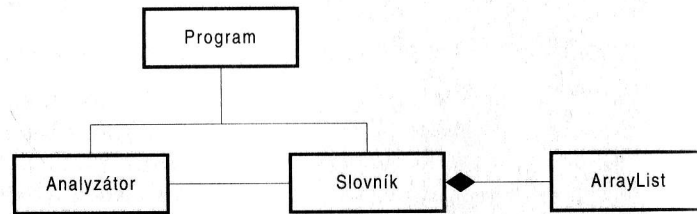
Třídy v programu

První návrh našeho programu by mohl vypadat asi takto: Napíšeme třídu, kterou nazveme třeba `AnalýzaTextu` a která se postará o přečtení jednotlivých řádků a jejich rozklad na slova. Tato třída si jednotlivá slova zapamatuje, setřídí a vytiskne. Protože nejde o příliš složitý program, svěříme vše jediné třídě.

Jakmile se o to ale pokusíme, zjistíme, že vznikající program je nepřehledný. Důvod je jednoduchý: Takto navržená třída `AnalýzaTextu` se stará o příliš mnoho věcí najednou (čtení textu a jeho analýzu, uložení jednotlivých slov a jejich zpracování). Rozumnější bude rozdělit program na několik menších tříd, z nichž každá bude mít přesně vymezené úkoly z jedné oblasti. Takovéto třídy lze pak i snáze znovu použít v jiných programech.

Náš program bude proto obsahovat třídu `Slovník`, která se bude starat o ukládání jednotlivých slov, jejich řazení a výpis, dále třídu `Analýzátor`, která bude číst text a rozkládat ho na jednotlivá slova, a třídu

Program, která bude obsahovat metodu Main(). Tato třída využije služeb zbývajících dvou (obr. 4.1).



Obr. 4.1 Vztah tříd v našem programu. Program využívá služeb Analyzátoru a Slovníku, Analyzátor využívá Slovníku (čára bez šipky vyjadřuje v UML obecnou asociaci)

Metoda Main() Třída Program bude obsahovat pouze metodu Main(). Začneme u ní, neboť zpravidla vyjadřuje základní funkčnost programu, nalezenou při analýze problému metodou shora dolů.

Tato metoda bude v cyklu číst jednotlivé řádky textu a předávat je analyzátoru ke zpracování. Analyzátor uloží jednotlivá slova do slovníku. Pak metoda Main() slovník seřadí a vytiskne.

Z předchozího popisu plyne, že v metodě Main() budeme potřebovat proměnnou typu²³ string, do níž uložíme přečtený řádek; nazveme ji výstižně řádek. Kromě toho budeme potřebovat instanci a třídy Analyzátor, která se postará o analýzu přečteného řádku, a instanci slovníku třídy Slovník, do níž se budou ukládat jednotlivá slova.

Pro čtení použijeme metodu Console.ReadLine(), která přečte a vrátí celý řádek jako instanci třídy string.²⁴ V dokumentaci se dočteme, že přečte-li tato metoda prázdný řádek, vrátí prázdný řetězec (instanci typu string, která neobsahuje žádné znaky). Narazí-li na konec souboru, vrátí null. (Pozor, prázdný řetězec "" není totéž co null.)

Cyklus pro čtení a analýzu vstupního textu tedy může vypadat takto:

```

while(řádek != null)           // Dokud se čtení daří
{
    řádek = řádek.Trim();       // Odstraň úvodní a závěrečné
    a.Analyzuj(řádek, slovník); // bílé znaky
    řádek = Console.ReadLine(); // Přečti další řádek
}
  
```

²³ Mějte prosím stále na paměti, že vyjádření „proměnná typu string“ vlastně znamená „proměnná představující odkaz na instanci třídy string“.

²⁴ Přesněji: Vytvoří instanci třídy string, uloží do ní obsah přečteného řádku a vrátí odkaz na tuto instanci.

Metoda Trim() třídy string vrátí instanci, která se od instance, pro niž byla volána, liší tím, že z ní byly odstraněny případné úvodní a závěrečné mezery.

Při vstupu do tohoto cyklu ovšem musí proměnná řádek již obsahovat první přečtený řádek; to obstará inicializace v deklaraci. Nyní už můžeme napsat celou metodu Main():

```

public static void Main()
{
    Analyzátor a = new Analyzátor(); // Potřebné proměnné
    Slovník slovník = new Slovník();
    string řádek = Console.ReadLine(); // Přečti první řádek

    while(řádek != null)           // Čtení a analýza řádků
    {
        řádek = řádek.Trim();
        a.Analyzuj(řádek, slovník);
        řádek = Console.ReadLine();
    }

    slovník.Seřaď();               // Seřazení a výpis slovníku
    slovník.Výpis();
}
  
```

Vypadá to jednoduše, že? Hlavní práci totiž obstarají pomocné třídy Slovník a Analyzátor.

Slovník

Požadavky

Nyní se podíváme na třídu Slovník. Co od ní požadujeme?

Tato třída by měla obsahovat metodu Vlož() pro přidání nového slova, dále metodu Seřaď(), která se postará o seřazení uložených slov podle abecedy, a metodu Výpis(), která obsah slovníku vypíše do standardního vstupu.

Pro zapamatování jednotlivých slov ve slovníku využijeme instanci np knihovny třídy ArrayList, která je definovaná v prostoru jmen System.Collections²⁵. Instance této třídy fungují jako pole, které může obsahovat libovolný počet prvků („nafukovací pole“). Deklarace třídy Slovník bude tedy mít tvar

```

using System;
using System.Collections;

class Slovník
{
    // Do np budeme ukládat přečtená slova
    private ArrayList np = new ArrayList(); // Nový prázdný

    public void Vlož(string slovo) { /*...*/ }
}
  
```

²⁵ Pro třídy sloužící k ukládání dat se vedle označení *kontejnery* používá alternativní název *kolekce*.


```

public void Vypis(){ /*...*/ }
public void Seřad(){ /*...*/ }
}

```

ArrayList

Podívejme se nyní na metodu Vlož(). Pomocí metody Add() třídy ArrayList můžeme do instance np ukládat odkazy na instance libovolných tříd; třída ArrayList totiž pracuje s odkazy na instance třídy object, která je společným předkem všech tříd v C#.

Při ukládání nového slova do slovníku np potřebujeme zjistit, zda už v něm toto slovo není. K tomu použijeme metodu IndexOf() třídy ArrayList, jež vrátí nezáporné číslo, pokud tam ukládaný objekt již je, a hodnotu -1, pokud tam ještě není:

```

public void Vlož(string slovo)
{
    if(np.IndexOf(slovo) == -1)
    {
        np.Add(slovo);
    }
}

```

Metoda Seřad bude velice jednoduchá – využije metodu Sort() třídy ArrayList:

```

public void Seřad()
{
    np.Sort();
}

```

Na konec nám zůstala metoda Vypis(). Ta musí probrat všechny instance, uložené v np, a vypsát je do standardního výstupu. K tomu bychom mohli použít příkaz foreach, o kterém si povíme v 8. kapitole; zde použijeme řešení založené na tzv. *enumerátorech*.

Enumerátor

Enumerátory (občas se také říká *iterátory*) jsou objekty, které se používají při procházení celých kontejnerů nebo jejich souvislých úseků. Můžeme se na ně dívat jako na zobecnění indexů, které určují prvky polí. V knihovně jazyka C# jsou definovány enumerátory v prostoru jmen System.Collections spolu s kontejnerovými třídami a v programech s nimi zacházíme jako s instancemi třídy IEnumerator.²⁶

Enumerátor pro danou instanci třídy ArrayList (ale i jiné kontejnerové třídy) získáme pomocí metody GetEnumerator(). Tato metoda vytvoří instanci a vrátí odkaz na ni.

Už jsme si řekli, že enumerátory se chovají tak trochu jako indexy pole, tj. určují prvek kontejneru. Každý enumerátor má metodu Move

²⁶ Později se dozvíme, že IEnumerator není třída, ale tzv. rozhraní. Zatím ale tento rozdíl není důležitý.

Next(), která ho „přesune“ na následující prvek. Tato metoda vrací logickou hodnotu (typu bool), která říká, zda se přesun podařil, tj. zda nějaký další prvek v kontejneru existuje. Vedle toho má složku²⁷ Current typu object, která představuje odkaz na prvek, který tento enumerátor určuje.

Zbývá doplnit, že enumerátor bezprostředně po vytvoření metodou GetEnumerator() neurčuje žádný prvek v kontejneru, ale chová se, jako kdyby určoval (neexistující) prvek před jeho prvním prvkem. První prvek bude určovat až po prvním volání metody MoveNext().

Vyzbrojení těmito vědomostmi už můžeme napsat metodu Vypis():

```

public void Vypis()
{
    System.Collections.IEnumerator i = np.GetEnumerator();
    while(i.MoveNext()) // Dokud existuje další prvek,
    { // vypiš ho
        Console.WriteLine(i.Current);
    }
}

```

Zde nejprve deklarujeme enumerátor i a inicializujeme ho hodnotou získanou voláním np.GetEnumerator().

Jak víme, metoda MoveNext() vrací logickou hodnotu, která říká, zda se „přesun“ enumerátoru na další prvek podařil; můžeme ji tedy použít jako podmínku v příkazu while. V cyklu pak pomocí metody Console.WriteLine() vypíšeme objekt určený složkou Current.

Nakonec si ukážeme deklaraci třídy Slovník vcelku:

```

class Slovník
{
    private ArrayList np = new ArrayList();
    public void Vlož(string slovo)
    {
        if(np.IndexOf(slovo) == -1)
        { np.Add(slovo);
        }
    }
    public void Vypis()
    {
        System.Collections.IEnumerator i = np.GetEnumerator();
        while(i.MoveNext())
        {
            Console.WriteLine(i.Current);
        }
    }
}

```

²⁷ Ve skutečnosti nejde o datovou složku v obvyklém smyslu, ale o tzv. vlastnost (property). Ani tento rozdíl zatím není důležitý.

```
public void Seřad()
{
    np.Sort();
}
```

Poznamenejme, že třída *Slovník* představuje tzv. *adaptér*: zapouzdřuje instanci jiné třídy, kterou zpřístupňuje prostřednictvím vlastního rozhraní (vlastních metod). Přejmenovává metodu *Sort()* třídy *ArrayList* na *Seřad()*, upravuje metodu *Add()* tak, aby kontrolovala, zda se žádný prvek neopakuje (a dává jí jiné jméno), a přidává novou metodu pro výpis. Řadu metod třídy *ArrayList*, které nejsou v případě slovníku potřebné, odstraňuje (nezveřejňuje je ve svém rozhraní).

Analýzátor

Na konec jsme si ponechali třídu *Analýzátor*, která má za úkol rozložit přečtený řádek na slova. Přitom za „slovo“ pokládáme jakoukoli posloupnost znaků jiných než je mezera nebo přechod na nový řádek. (Jinak řečeno, slova jsou oddělena mezerami nebo konci řádků. Později se ale můžeme rozhodnout, že ve vstupním textu mohou fungovat jako oddělovače slov i jiné znaky – např. tečky, čárky, otazníky ap.)

Třída *Slovník* proto bude mít jako datovou složku řetězec se znaky, které určují hranice slov. Většinu práce této třídy obstará metoda *Analýzuj()*, která bude mít jako parametry analyzovaný řádek a slovník.

Analýza textu

Vezměme tedy metodu *Analýzuj()*. Ta nejprve přeskočí mezery před slovem a pak bude kopírovat z proměnné *řádek* jednotlivé znaky tvořící slovo do pomocného řetězce, dokud nenarazí na mezeru nebo na jiný oddělovač.

StringBuilder

Ovšem k vytváření slova z jednotlivých znaků nemůžeme použít instanci typu *string*, neboť její obsah nelze měnit.²⁸ Místo toho použijeme instanci knihovny třídy *System.Text.StringBuilder*, která také slouží k ukládání znakových řetězců, ale její obsah lze měnit. Nový znak do ní vložíme pomocí metody *Append()*.

Dále potřebujeme vědět, že k jednotlivým znakům v instanci třídy *string* – např. v proměnné *řádek* – lze přistupovat zápisem *řádek[i]*. (Říkáme, že proměnnou *řádek* *indexujeme*; první znak má index 0, takže ho získáme zápisem *řádek[0]*.)

Metoda *Analýzuj()* nejprve zkontroluje, zda nedostala prázdný řádek; pokud ano, skončí, neboť není co analyzovat. Jinak vytvoří pomocnou

²⁸ Nenechte se zmást příkazy, jako je *řádek = řádek.Trim();*

Metoda *Trim()* vytvoří novou instanci a odkaz na ni přiřadí proměnné *řádek*. Původní instance, na níž odkazovala proměnná *řádek*, zanikne – odstraní ji automatická správa paměti.

proměnnou *i*, která nám bude sloužit jako index (ukazatel pozice) v analyzovaném řádku, a pomocnou proměnnou *slovo* typu *StringBuilder*. Poté v cyklu projde jednotlivá slova v řádku a každé z nich přeneso znak po znaku do slova.

Pro přeskakování mezer mezi slovy použijeme pomocnou metodu *PřeskočMezery()*, která vrátí index prvního znaku slova, pokud ho najde. Narazí-li tato metoda na konec řádku, vrátí -1.

Cyklus, který bude přenášet jednotlivé znaky z řádku do slova, může vypadat takto:

```
// Cyklus přes jednotlivá slova
while(i >= 0)           // Dokud nenarazíš na konec řádku:
{
    slovo = new System.Text.StringBuilder("");

    // Cyklus přes jednotlivé znaky slova
    // Dokud nepřekročíš délku řádku a nenarazíš na oddělovač:
    while((i < řádek.Length) &&
           (MEZERY.IndexOf(řádek[i]) == -1))
    { // Přenes znak a přejdi v proměnné řádek na další
      slovo.Append(řádek[i++]);
    }

    // Nakonec ulož slovo do slovníku a přejdi na další
    slovník.Vlož(slovo.ToString());
    i = PřeskočMezery(řádek, i);
}
}
```

Operátor &&

V podmínce cyklu *while* jsme potřebovali, aby zároveň platily dvě podmínky: Index *i* musí být menší než délka řádku a znak *řádek[i]* nesmí být mezera. To v jazyce C# zapíšeme tak, že obě podmínky spojíme pomocí operátoru *&&*, který vyjadřuje logickou konjunci. Kdybychom chtěli, aby platila alespoň jedna ze dvou podmínek, použili bychom operátor *||*.

PřeskočMezery()

Zbývá ještě napsat metodu *PřeskočMezery()*, která bude mít jako parametry analyzovaný řádek a index od znaku, od kterého má začít.

Tato metoda nejprve zkontroluje, zda již není na konci řádku; pokud ano, vrátí -1. Ke kontrole použije složku²⁹ *Length*, která udává počet znaků v řetězci. (Už jsme si řekli, že znaky jsou v instancích třídy *string* indexovány od nuly. To znamená, že např. naše instance *řádek* obsahuje znaky *řádek[0]*, *řádek[1]*, ..., *řádek[řádek.Length - 1]*, ale neobsahuje znak *řádek[řádek.Length]*.)

Tato metoda bude v cyklu procházet znaky *řádek[od]*, *řádek[od+1]* atd. a bude zkoumat, zda narazila na jiný znak než oddělovač nebo na

²⁹ Opět nejde o datovou složku, ale o vlastnost.

konec řetězce. Pokud narazí na konec řetězce, vrátí -1, jinak vrátí index prvního znaku slova. Její zdrojový text bude velice jednoduchý:

```
public int PřeskočMezery(string řádek, int od)
{
    if(od >= řádek.Length){ return -1; } // Konec řádku?
    while(MEZERY.IndexOf(řádek[od]) >= 0)
    {
        if(++od == řádek.Length) {return -1;}; // Konec řádku?
    }
    return od; // Index prvního znaku dalšího slova
}
```

I když jde o pomocnou metodu, deklarovali jsme ji jako veřejně přístupnou (public), neboť nemůžeme vyloučit, že ji při pozdějších úpravách programu použijeme i mimo tuto třídu, např. v metodě Main() místo metody Trim().

Pro úplnost si ukážeme zdrojový text celé třídy Analyzátor.

```
class Analyzátor
{ // Mezera a přechod na nový řádek
    private string MEZERY = " \n";
    public int PřeskočMezery(string řádek, int od)
    {
        if(od >= řádek.Length){ return -1; }
        while(MEZERY.IndexOf(řádek[od]) >= 0)
        {
            if(++od == řádek.Length) { return -1; }
        }
        return od;
    }

    public void Analyzuj(string řádek, Slovník slovník)
    {
        if(řádek == "") return;
        int i = 0;

        System.Text.StringBuilder slovo = null;
        i = PřeskočMezery(řádek, i);
        while(i >= 0)
        {
            slovo = new System.Text.StringBuilder("");
            while((i < řádek.Length) &&
                (MEZERY.IndexOf(řádek[i]) == -1))
            { slovo.Append(řádek[i++]); }
            slovník.Vlož(slovo.ToString());
            i = PřeskočMezery(řádek, i);
        }
    }
}
```



Celý program najdete na WWW v souboru Kap04\01\Analyza.cs.

Nyní můžeme program vyzkoušet. O tom, že bude správně řadit slova bez háček a čárek, asi nikdo nepochybuje; podívejme se tedy, jak dopadnou slova obsahující znaky s diakritickými znaménky. Zadejme text

alfa numero tygr Rek rek ňouma tuhýk Ája Řek řek duch duh azur

stiskneme klávesu Enter a pak kombinaci Ctrl+Z, jež znamená konec vstupních dat. Výsledek, který ukazuje obr. 4.2, naznačuje, že při porovnávání řetězců (instancí třídy string) se uplatňují pravidla českého abecedního řazení – spřežku *ch* umístil náš program mezi *h* a *j*, znaky jako *á* nebo *ň*, které mají sekundární řadící platnost, nebral v úvahu. (V souladu s normou [8] hovoříme o řazení, ale jde o třídění, jak jsme o něm hovořili v první kapitole.)

Obr. 4.2 Náš program seřadil zadaná slova správně podle abecedy

Zkusme mu zadat ještě výkřik *pádla padlá padla*. Odpovědí nám bude překvapující sdělení, že *padla padlá pádla*. I tentokrát program respektoval pravidla českého abecedního řazení.³⁰ Výstup ovšem také ukazuje, že program bere v úvahu rozdíl mezi malými a velkými písmeny, což by podle [8] neměl.

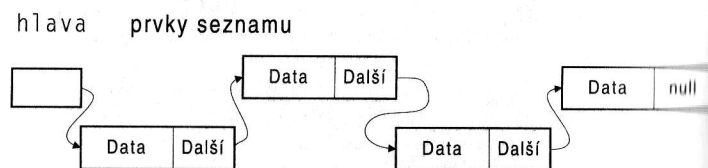
³⁰ Tato pravidla upravuje státní norma [8]. Stručně řečeno, tato norma říká, že řazení je dvouprůchodové. V prvním průchodu se uvažují písmena *a b c ě d e f g h c h i j k l m n o p q r ř s š t u v w x y z ž*, která mají primární řadící platnost. (Spřežka *ch* se pokládá za písmeno a je zařazena mezi *h* a *i*.) Písmena *á d' é ě í ň ó t' ú ů ý*, jež mají sekundární řadící platnost, se při prvním průchodu chovají jako odpovídající písmena bez diakritických znamének. Teprve pokud nelze pořadí dvou slov určit podle písmen s primární řadící platností, vezmou se v úvahu písmena se sekundární platností. Podobně se zachází i se znaky jiných národních abeced, jako je slovenské *ľ* nebo francouzské *ç*. Velká a malá písmena se nerozlišují.

4.2 Seznam

V tomto oddílu se budeme zabývat stejným problémem jako v předchozím, tedy řazením slov; místo knihovny třídy `ArrayList` si však naprogramujeme svůj vlastní kontejner. Použijeme ten nejjednodušší, jednosměrně zřetěžený seznam.

Jednosměrně zřetěžený seznam

Jednosměrně zřetěžený seznam (nebo prostě jen *seznam*) je datová struktura, která se skládá z prvků, z nichž každý obsahuje vedle užitečných dat ještě odkaz na další prvek; prvky seznamu jsou pomocí těchto odkazů „zřetězeny“. Známe-li první prvek, můžeme s pomocí odkazu, který obsahuje, najít druhý prvek, s jeho pomocí pak třetí atd. Poslední prvek bude místo odkazu na následující obsahovat hodnotu `null` (odkaz „nikam“). Schéma nejjednodušší verze jednosměrně zřetěženého seznamu ukazuje obr. 4.3.



Obr. 4.3 Schéma jednosměrně zřetěženého seznamu

Abychom mohli se seznamem pracovat, musíme znát odkaz na jeho první prvek (na obr. 4.3 označený *hlava*).

Prvek seznamu

Prvky seznamu budou instance třídy `Prvek`, kterou naprogramujeme v tomto souboru jako třídu `Seznam`. Prvek bude obsahovat odkaz `data` na uložený řetězec a odkaz `další` na následující prvek. Kromě toho bude obsahovat konstruktor, metody pro zjištění, zda prvek obsahuje daný řetězec, a pro jeho vrácení a metody pro zjištění a nastavení odkazu na další prvek. Podívejme se na její výpis.

```
class Prvek
{
    private string data;
    private Prvek další;

    public Prvek(string _data){
        NastavData(_data); další = null;
    }
}
```

```
public string VraťData(){ return data; }
public void NastavData(string _data){ data = _data; }
public Prvek VraťDalší(){ return další; }
public void NastavDalší(Prvek p){ další = p; }
}
```

Implementace seznamu

Třída `Seznam` bude obsahovat jedinou datovou složku, a to odkaz na první prvek. Stejně jako na obr. 4.3 ho nazveme *hlava*.

Vedle konstruktoru, který vytvoří prázdný seznam, budeme potřebovat metody `VložNaKonec()` a `VložNaZačátek()`, která vloží nový prvek na konec, resp. na počátek seznamu, metodu `Najdi()`, která projde seznam a zjistí, zda obsahuje prvek se zadanou hodnotou, a metodu `Seřad()`, která uspořádá prvky seznamu podle abecedy. Mimo to budeme potřebovat ještě metodu `Vypiš()`, která vypíše jednotlivé prvky. To je vše.

Konstruktor vloží do datové složky *hlava* hodnotu `null`, nic víc.

Metoda `VložNaKonec()` se nejprve podívá, zda není seznam náhodou prázdný. Pokud ano, tj. pokud *hlava* obsahuje `null`, vytvoří nový prvek a odkaz na něj uloží do složky *hlava*. Jinak projde celý seznam, najde jeho poslední prvek a připojí k němu nový prvek, obsahující zadaná data. Při procházení použije dvojici pomocných proměnných, `p` a `q`. Pomocí proměnné `p` budeme procházet jednotlivé prvky seznamu, až narazíme na jeho konec. Na další prvek seznamu přejdeme příkazem

```
p = p.VraťDalší();
```

Předtím si ale tento odkaz zapamatujeme, uložíme si ho do `q`. Když dojdeme na konec seznamu, bude `p` obsahovat `null` a `q` odkaz na poslední prvek. Implementace metody `VložNaKonec()` může mít tvar

```
public void VložNaKonec(string co)
{
    if(hlava == null) // Je-li seznam prázdný
    { hlava = new Prvek(co); }
    else // Není-li prázdný
    {
        Prvek p = hlava, q = null;
        while(p != null) // Najdi konec seznamu
        {
            // q bude obsahovat odkaz na
            // poslední prvek
            q = p;
            p = p.VraťDalší();
        }
        q.NastavDalší(new Prvek(co)); // Připoj nový prvek
    }
}
```

Metoda VložNaPočátek() bude podstatně jednodušší:

```
public void VložNaPočátek(string co)
{
    Prvek p = hlava;          // Ulož si odkaz na první prvek
    hlava = new Prvek(co);     // Vlož nový
    hlava.NastavDalší(p);      // a připoj zbytek
}
```

Metoda Najdi() projde seznam prvek po prvku a bude zjišťovat, zda obsahuje zadaný řetězec. Pokud ho najde, vrátí odkaz na nalezený prvek, jinak vrátí null.

```
public Prvek Najdi(string co) // Zjistí, zda je hodnota co
{                             // v seznamu, a pokud ano,
    Prvek p = hlava;          // vrať odkaz na prvek
    while(p != null && p.VraťData() != co)
    {
        p = p.VraťDalší();
    }
    return p;
}
```

Asi nejsložitější bude metoda Seřaď(), která seřadí řetězce v seznamu podle velikosti. Použijeme algoritmus třídění výběrem, o němž jsme hovořili v první kapitole v části 1.4:

Nejprve zkontrolujeme, zda náhodou není seznam prázdný. Pokud ano, skončíme, neboť není co dělat. Jinak projdeme celý seznam a najdeme v něm prvek obsahující nejmenší hodnotu. Hodnotu z tohoto prvku pak prohodíme s hodnotou v prvním prvku seznamu.

Výsledkem je, že nejmenší (tj. v abecedě první) řetězec je na správném místě; nyní projdeme zbytek seznamu (počínaje druhým prvkem), opět najdeme nejmenší a prohodíme řetězec v něm s řetězcem ve druhém prvku atd.

Přitom využijeme tři pomocných proměnných: p bude obsahovat odkaz na první prvek zkoumaného úseku, q bude sloužit k procházení tohoto úseku a r bude obsahovat odkaz na dosud nalezený nejmenší prvek ve zkoumaném úseku.

```
public void Seřaď()
{
    Prvek p = hlava, q = null, r = null;
    if(p != null) // Je-li seznam prázdný, není co řadit
    {
        // p: Odkaz na první prvek úseku
        while(p.VraťDalší() != null)
        {
            // q: prochází úsek
            q = p.VraťDalší();
            r = p; // r: pamatuje dosud nalezený nejmenší
            while(q != null) // Projdi prvky úseku
            {
```

```
if(q.VraťData().CompareTo(r.VraťData()) <= 0)
    r = q; // Pokud najdeš menší prvek,
    q = q.VraťDalší(); // zapamatuj si ho v r
}
```

```
// Prohod' nejmenší s prvním
string pom = r.VraťData();
r.NastavData(p.VraťData());
p.NastavData(pom);

// Nejmenší prvek úseku je nyní na svém místě,
// vezmi úsek začínající o jeden prvek dál
p = p.VraťDalší();
}
```

Pro porovnávání řetězců jsme použili metodu CompareTo() třídy string. Náš seznam neumožňuje používat enumerátory, a proto musíme při jeho procházení používat pomocné proměnné typu Prvek.

Podívejme se ještě na výpis celého seznamu. Metoda Vypiš() bude jednoduchá: Projde prvek po prvku, vyžádá si data a vypíše je do standardního výstupu.

```
public void Vypiš()
{
    Prvek p = hlava;
    while(p != null)
    {
        System.Console.Write(p.VraťData()+" ");
        p = p.VraťDalší();
    }
}
```

Budeme-li chtít použít třídu Seznam k pokusům s řazením slov, budeme muset poněkud upravit třídu Slovník, která slouží jako adaptér. Metody Seřaď() a Vypiš() se prostě odvolají na odpovídající metody Seznamu, metoda Vlož() si nejprve pomocí metody Najdi() zjistí, zda Slovník už zadané slovo obsahuje, a pokud ne, vloží ho na počátek:

```
if(seznam.Najdi(slovo) == null)
{
    seznam.VložNaPočátek(slovo);
}
```



Implementaci třídy Seznam a program, který ji využívá k řazení slov, najdete na WWW v souboru Kap04\02\Analyza2.cs. Samotnou implementaci seznamu najdete v souboru Kap04\02\Seznam.cs.

4.3 Vylepšujeme vstup

Naše nástroje pro čtení vstupních dat, které jsme si připravili v minulých kapitolách, nejsou právě nejdokonalejší, a proto se nyní pokusíme vytvořit lepší verzi.

Požadavky

V naší první verzi jsme mohli zapsat na řádek vždy jen jedno číslo. Naším hlavním cílem je toto omezení odstranit. Využijeme zkušenosti, které jsme získali v oddílu věnovaném řazení slov.

Úvodní úvahy

Předpokládáme, že na vstupu jsou čísla oddělená alespoň jednou nulou nebo přechodem na nový řádek.

Náš postup bude jednoduchý: Přečteme řádek, přeskočíme v něm mezery a pak – znak po znaku – z něj přeneseme „slovo“, tj. řetězec představující číslo. K převodu řetězce na číselnou hodnotu použijeme už známou třídu `System.Convert`. K vytvoření „slova“ použijeme opět třídu `System.Text.StringBuilder`.

Program tedy bude začínat direktivami

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

Datové složky

První, co musíme udělat, je přečíst řádek vstupního textu. Uložíme si ho do statické proměnné (datové složky) `řádek`. Abychom nemuseli opakovaně zjišťovat jeho délku, uložíme si ji do proměnné `délka`. Vedle toho budeme potřebovat pomocnou proměnnou `index`, která bude při zpracování řádku ukazovat aktuální pozici.

Nová verze třídy `VV` tedy bude začínat deklaracemi

```
private static string řádek = null;
private static int index = 0;
private static int délka = 0;
```

Čtení řádku

První, co musíme udělat, je přečíst řádek. K tomu použijeme jako obvykle metodu `Console.ReadLine()`. Zároveň nastavíme ukazatel pozice `index` na počátek řádku a zjistíme jeho délku:

```
private static void ČtiŘádek()
{
    řádek = Console.ReadLine();
    if(řádek == null) throw
        new Exception("Čtení za koncem souboru.");
    index = 0; // Jsme na počátku řádku
    délka = řádek.Length; // Zjistí jeho délku
}
```

Pokusíme-li se číst z klávesnice za znakem `Ctrl+Z`, nebo – použijeme-li přesměrování – za koncem souboru, vrátí metoda `Console.Read`

`Line()` hodnotu `null`. V tom případě postrádá jakékoli další zpracování smysl, a proto vyvoláme výjimku. (Zatím nám to připadá jako dobrý nápad; později ale uvidíme, že to v některých situacích není příliš vhodné, a budeme mít záminku k dalším úpravám.)

Metodu pro přeskočení oddělovačů jsme už jednou napsali. Tentokrát bude o něco jednodušší, neboť jako oddělovače uvnitř řádku uvažujeme jen mezery.

```
private static void PřeskočMezery()
{
    while(index < délka && řádek[index] == ' ')
    {
        // Nejsme-li na konci řádku a je-li to
        index++; // mezera, posuň se o jeden znak dále
    }
}
```

Vrať slovo

Nejvíce práce nám dá metoda `VraťDalšíSlovo()`, která vyhledá v řetězci řádek další slovo a vrátí ho. Ani ta ovšem nebude nijak složitá.

Na počátku obsahuje proměnná `řádek` hodnotu `null`, a proto musí metoda `VraťDalšíSlovo()` přečíst řádek. Pak přeskočí mezery. Může se ale stát, že přečtený řádek bude obsahovat jen mezery; v tom případě je třeba načíst další, a případně ještě další – dokud se nepřečte řádek, který bude obsahovat nějaké „slovo“. Skutečnost, že řádek obsahuje jen mezery, poznáme tak, že po návratu z metody `PřeskočMezery()` bude proměnná `index`, určující aktuální pozici, rovna délce řádku.

Pak vytvoříme instanci slova třídy `StringBuilder` a přeneseme do ní jednotlivé znaky slova. Nakonec toto slovo převedeme na `string` a vrátíme.

```
private static string VraťDalšíSlovo()
{
    // Na počátku je třeba
    if(řádek == null) ČtiŘádek(); // přečíst řádek a
    PřeskočMezery(); // přeskočit mezery

    while(index == délka) // Jsme-li na konci řádku
    {
        // nebo je-li prázdný,
        ČtiŘádek(); // přečti další a přeskoč
        PřeskočMezery(); // v něm mezery
    }

    StringBuilder slovo = new StringBuilder("");
    while(index < délka && řádek[index] != ' ')
    {
        // Pak přenes všechny
        slovo.Append(řádek[index++]); // znaky slova
    }
    return slovo.ToString();
}
```


Metody `InInt()` a `InDouble()` se ve srovnání s předchozí verzí změní jen velice málo: Použijí služeb metody `VraťDalšíSlovo()`.

```
public static int InInt()
{
    return Convert.ToInt32(VraťDalšíSlovo());
}

public static double InDouble()
{
    return Convert.ToDouble(VraťDalšíSlovo());
}
```

`private` Všimněte si, že metody `ČtiŘádek()`, `PřeskočMezery()` a `VraťDalšíSlovo()` jsme deklarovali jako soukromé (`private`); jde o implementační detaily, které nepotřebují uživatelé naší třídy používat. Uživatelé, klientskému programátorovi, jsou přístupné pouze metody `InInt()`, `InDouble()` a `InString()`, které slouží ke vstupu dat.



Celou třídu `VV` najdete na WWW v souboru `Kap04\03\vv.cs`. Podobně jako při překladu předchozí verze, i tentokrát je třeba předepsat, že výsledkem bude knihovna. K překladu tedy použijeme příkaz

```
csc /t:library vv.cs
```

Už jsme se zmínili, že ani tato verze knihovny ještě není dobrá. Uml sice číst více položek z jedné řádky, ovšem použijeme-li ji ke čtení ze souboru, ve kterém neznáme počet položek, skončí program výjimkou. I to později odstraníme. (Připomeňme si, že jde vlastně o čtení ze standardního vstupu. To sice znamená obvykle klávesnici, ale přesměrováním v příkazu, kterým program spustíme, můžeme dosáhnout čtení ze souboru. O tom jsme hovořili v kap. 1 na konci oddílu 1.2.)

6 Začínáme naostro

V předchozích kapitolách jsme si na řadě příkladů ukázali některé možnosti, které jazyk C# nabízí. To, co jsme se v nich naučili, nám umožní psát příklady, které půjdou spustit a budou něco užitečného dělat.

Nyní se tedy pustíme do soustavného výkladu jazyka C#. Nenechte se zmást tím, že vám některé věci budou připadat známé z předchozích kapitol; informace v nich nebyly úplné.

UNICODE

Už jsme zjistili, že C# umožňuje používat bez problémů znaky české abecedy. To je dáno skutečností, že vnitřně pracuje se znaky v kódování UNICODE, ve kterém je každý znak zobrazen ve 2 bajtech. Toto kódování umožňuje používat znaky většiny národních abeced.

Jak budeme C# popisovat

O formálních pravidlech, podle nichž sestavujeme deklarace, příkazy a jiné konstrukce jazyka C#, hovoříme jako o *syntaxi* tohoto jazyka. Při jejím popisu budeme postupovat takto:

- Před popisem uvedeme na samostatném řádku název popisované konstrukce ukončený dvojtečkou.
- Části, které lze do programu beze změny opsat (říkáme jim *terminální symboly*), zapíšeme **tučným písmem**.
- Části, které je třeba dále definovat nebo které jsme už někde definovali (říkáme jim *neterminální symboly*), zapíšeme *kurzívou*. Bude-li se označení neterminálního symbolu skládat z několika slov, budeme je zpravidla spojoval podtržítkem.
- Popisované konstrukce mohou mít několik možností. Každá z nich bude začínat odrážkou „•“.

Bude-li to vhodné, nahradíme přesný syntaktický popis neformálním vysvětlením; v mnoha situacích to bude srozumitelnější. Někdy také použijeme zjednodušený popis a doplníme ho výkladem. Popis syntaxe bude samozřejmě doprovázet i výklad o významu popisovaných konstrukcí. Ten bohužel takhle přehledně zpravidla zachytit nelze.

Příklad

Jako příklad si ukážeme část popisu syntaxe komentáře.

komentář:

- */* text_víceřádkového_komentáře */*

- *// text_jednořádkového_komentáře_do_konce_řádku*
- */// text_dokumentačního_komentáře_do_konce_řádku*

Tento popis říká, že v C# existují tři druhy komentářů. První z nich začíná znaky */**. V popisu jsou tučně (i když to v tisku asi není vidět), takže je můžeme přímo zapsat do svého programu. Pak následuje *text_víceřádkového_komentáře*. Tento symbol je zapsán kurzivou, takže budeme ještě muset upřesnit, o co jde: Buď musíme uvést další syntaktický popis, nebo ho nahradit výkladem. Tentokrát zvolíme neformální výklad: *text_víceřádkového_komentáře* je prostě jakýkoli text, který může zabírat i několik řádků.

Druhý typ komentáře začíná znaky *//* zapsanými těsně za sebou, za nimiž následuje *text_jednořádkového_komentáře_do_konce_řádku*.

Poslední možnost představuje komentář začínající třemi znaky *///* bezprostředně za sebou, za nimiž následuje *text_dokumentačního_komentáře_do_konce_řádku*. (Další podrobnosti o komentářích v C# najdete v následujícím oddílu.) ✧

5.1 Základní pojmy

Komentář

Komentář je součást zdrojového textu, kterou překladač ignoruje. Slouží programátorovi ke zřehlednění zdrojového textu. Syntaktický popis jsme viděli výše, takže už víme, že v C# máme k dispozici tři druhy komentářů:

- Jednořádkový komentář začínající znaky *//* a končící přechodem na nový řádek.
- Komentář začínající znaky */** a končící **/*. Tento komentář může zabírat i několik řádků.
- Dokumentační komentář začínající *///* a končící přechodem na nový řádek. Slouží k automatickému generování dokumentace ve formátu XML a podrobněji si o něm povíme v dodatku.

Komentáře začínající */** nelze do sebe vnořovat. To znamená, že následující řádky překladač označí za chybné:

```
/* CHYBA - KOMENTÁŘE NELZE VNOŘOVAT
while(i < n)
{
    s *= n--;    /* výpočet faktoriálu n! */
}
*/
```

Na podobnou situaci často narazíme při hledání chyb v programech. Při ladění totiž často potřebujeme některou část zdrojového textu dočasně odstranit, a abychom ji nemuseli později znovu psát (a znovu ladit), umístíme ji do komentáře. Abychom se podobným problémům vyhnuli, začínáme raději „skutečné“ komentáře dvěma lomítky, neboť jednořádkový komentář do komentáře začínajícího */** vnořit lze:

```
/* To je v pořádku
while(i < n)
{
    s *= n--;    // výpočet faktoriálu n!
}
*/
```

Klíčová slova

Vyhrazená slova, která v C# označují příkazy, předdefinované datové typy atd., nazýváme *klíčová slova*. Všechna klíčová slova v C# musíme zapisovat vždy malými písmeny.

V následující tabulce najdete přehled všech klíčových slov jazyka C#. Výklad o některých z nich ovšem přesahuje možnosti této knihy, a proto je uvádíme jen pro úplnost.

abstract	as	base	bool	break
byte	case	catch	char	checked
class	const	continue	decimal	default
delegate	do	double	else	enum
event	explicit	extern	false	finally
fixed	float	for	foreach	goto
if	implicit	in	int	interface
internal	is	lock	long	namespace
new	null	object	operator	out
override	params	private	protected	public
readonly	ref	return	sbyte	sealed
short	sizeof	stackalloc	static	string
struct	switch	this	throw	true
try	typeof	uint	ulong	unchecked
unsafe	ushort	using	virtual	void
volatile	while			

Identifikátor

Identifikátor je jméno, kterým v programu označujeme proměnné, třídy, metody, prostory jmen atd. Identifikátor je tvořen písmeny a číslicemi, začínat musí písmenem. Přitom za písmena se pokládají znaky národních abeced a znak podtržení. *V identifikátorech se rozlišují velká a malá písmena.* To znamená, že `lípa`, `Lípa` a `_lípa` jsou tři různé (a různé) identifikátory. Délka identifikátoru není omezena.

Identifikátor nesmí začínat číslicí a nesmí obsahovat mezeru, tabulátor ani jiný z tzv. bílých znaků. Nesmí také obsahovat tečku, čárku, vlnovku a jiné speciální znaky. To znamená, že např. `Žlutý kůň` není identifikátor, neboť obsahuje mezeru; také `Žlutý-kůň` není identifikátor, neboť obsahuje znak minus. Na druhé straně `Žlutý_kůň`, `ŽlutýKůň` a `ŽlutýKun` jsou různé správně utvořené identifikátory, neboť obsahují jen písmena a podtržítka.

@ Identifikátor může také začínat znakem `@`. Tento znak není součástí identifikátoru, určuje ale, že následující slovo představuje identifikátor. To znamená, že `ŽlutýKůň` a `@ŽlutýKůň` jsou dva naprosto stejné identifikátory.

Znak `@` lze také použít k rozlišení identifikátorů od klíčových slov. Např. `while` není možné použít jako identifikátor, protože se přece shoduje s jedním z klíčových slov. Naproti tomu `@while` nebo `while@` jsou správné identifikátory: První z nich je odlišen znakem `@`, druhý od klíčového slova `while` liší velikostí písmen.

Upřímně řečeno, neumím si představit, k čemu je používání znaku `@` dobré. I kdybychom v programu používali anglické identifikátory i v programátorských týmech, v nichž jsou i cizinci, je to zřejmě nezbytné – je rozumné se identifikátorům, které připomínají klíčová slova, zdaleka vyhnout, neboť znepráhledňují program.

Na rozdíl od některých jiných programovacích jazyků se v C# zatím nepoužívá žádná konvence, která by určovala, jak mají vypadat identifikátory tříd, proměnných, metod, konstant atd. V naší knize se budeme snažit dodržovat poměrně jednoduchá pravidla:

- Identifikátory tříd, metod a prostorů jmen budou začínat velkým písmenem, následující písmena budou malá. Bude-li název obsahovat několik slov, bude každé z nich začínat velkým písmenem.
- Identifikátory konstant budou obsahovat pouze velká písmena.
- Identifikátory proměnných a datových složek budou začínat malým písmenem. Bude-li název obsahovat několik slov, bude každé z následujících začínat velkým písmenem.

Zápis programu

Pravidla pro zápis programu v C# lze shrnout do následujících bodů:

- Identifikátory a klíčová slova nesmíme rozdělit bílým znakem (mezerou, přechodem na nový řádek, tabulátorem, komentářem). Nesmíme tedy napsat např. `Ma in()`.
- Také některé operátory, které se skládají z několika znaků, nelze rozdělit. To se týká např. operátorů `+=` (nesmíme napsat `+` a `=` s mezerou mezi znaky `+` a `=`), `--`, `++`, `==`, `<=`, `&&` a dalších.
- Je-li třeba zapsat v programu bezprostředně vedle sebe dva identifikátory, identifikátor a klíčové slovo nebo dvě klíčová slova, musíme je oddělit alespoň jedním bílým znakem.
- Na místě, kde může být jedna mezera, může být libovolný počet bílých znaků nebo komentář.

5.2 Prostory jmen

V tomto oddílu si ukážeme, jak se prostory jmen v C# deklarují a jak se používají. Vedle toho si povíme o některých prostorech jmen z knihovny prostředí .NET Framework.

Připomeňme si, že prostory jmen slouží jako jakási „přijmení“ k identifikátorům. Představují nástroj pro zvládnutí velkého množství identifikátorů v rozsáhlejších programech a usnadňují používání knihoven.

Deklarace prostoru jmen

Nejprve si ukážeme syntaktický popis deklarace prostoru jmen.

Deklarace prostoru jmen:

- `namespace jméno { součásti_prostoru_jmen }`

Deklarace prostoru jmen začíná klíčovým slovem `namespace`, za nímž následuje *jméno* prostoru jmen. Pak následují *součásti prostoru jmen*, uzavřené ve složených ve složených závorkách.

Jméno prostoru jmen je buď identifikátor nebo skupina identifikátorů spojených tečkou. *Součásti prostoru jmen* mohou být direktivy `using`, za nimi pak deklarace tříd nebo jiných datových typů nebo deklarace dalších (vnířných) prostorů jmen.

Všechny identifikátory, deklarované uvnitř prostoru jmen, budou mít „přijmení“ tvořené *jménem* tohoto prostoru jmen; při použití mimo tento prostor jmen je proto budeme muset kvalifikovat *jménem* prostoru jmen.

ru jmen (nebo budeme muset použít direktivu using). Při použití uvnitř prostoru jmen není kvalifikace nutná.

Příklad Začneme schématickým příkladem: Deklarujeme prostor jmen První a v něm třídy Y a Z.

```
namespace První { // Deklarace prostoru jmen
    class Y { // Deklarace uvnitř prostoru jmen
        public Y(){}
    }
    class Z {
        private Y yy;
        public Z(){ yy = new Y();}
    }
} // Konec deklarace prostoru jmen
```

K tomu připojíme třídu, která bude ležet mimo prostor jmen První, která použije třídu První.Z:

```
class X
{
    static void Main()
    {
        První.Z z = new První.Z();
        // ...A nějaká použití třídy Z
    }
}
```

Všimněte si, že v konstruktoru třídy Z jsme použili třídu Y bez kvalifikace. To můžeme, neboť obě leží ve stejném prostoru jmen. Na druhé straně ve třídě X jsme museli psát První.Z, neboť třída X neleží v prostoru jmen První. ♦

Prostor jmen můžeme deklarovat také uvnitř jiného prostoru jmen. Pro třídy z vnořeného prostoru jmen pak musíme používat kvalifikaci jménem vnějšího prostoru jmen, za níž připojíme kvalifikaci jménem vnitřního prostoru jmen. Do vnořeného prostoru jmen může být vnořen další prostor jmen atd. Identifikátory z vnějšího prostoru jmen jsou dostupné bez kvalifikace i ve vnitřním prostoru jmen.

Příklad Vezměme prostor jmen První z předchozího příkladu a vložme do něj deklaraci prostoru jmen Druhy, který bude obsahovat třídu U:

```
namespace První {
    class Y { /* ... jako předtím ... */ }
    class Z { /* ... jako předtím ... */ }
    // Deklarace vnořeného prostoru jmen
    namespace Druhy {
```

```
        class U
        {
            private Y y;
            public U(){ y = new Y(); }
        }
    } // Konec vnořeného prostoru jmen
} // Konec vnějšího prostoru jmen
```

Použití třídy U v metodě Main() třídy X, která leží mimo prostor jmen První, budeme muset kvalifikovat jmény obou prostorů jmen:

```
První.Druhy.U u = new První.Druhy.U();
```

Na druhé straně v konstruktoru třídy První.Druhy.U jsme mohli použít jméno Y z vnějšího prostoru jmen bez kvalifikace.



Zdrojový text tohoto příkladu najdete na WWW v souboru Kap05\01\ nsp1.cs. ♦

Spojování prostorů jmen

Deklarace prostorů jmen můžeme rozdělit na několik částí, překladač si s tím poradí. Jestliže se *jméno* prostoru jmen skládá z několika identifikátorů oddělených tečkami, jde o deklaraci vnořeného prostoru jmen.

Příklad

Např. deklarace z předchozího oddílu bychom mohli přepsat do tvaru

```
namespace První {
    class Y { /* ... jako předtím ... */ }
}
namespace První{
    class Z { /* ... jako předtím ... */ }
}
namespace První.Druhy {
    class U{ /* ... jako předtím ... */ }
}
```

a výsledek by byl stejný. Takto upravený program najdete na WWW v souboru Kap05\01\nsp2.cs. ♦

Globální prostor jmen

V našich programech jsme dosud prostor jmen nedeklarovali. Identifikátory, které nedeklarujeme v žádném prostoru jmen, budou ležet v jednom – tzv. globálním – prostoru jmen.

using

Ve druhé kapitole jsme se setkali s direktivou `using`, která umožňuje vynechat kvalifikaci jménem prostoru `jmen`. Tato direktiva musí být zapsána v prostoru `jmen` (příp. v souboru) před deklaracemi tříd a její syntaxe je

direktiva using:

- `using jméno_prostoru_jmen ;`

Použijeme-li direktivu `using` pro nějaký prostor `jmen`, smíme za ní v programu vynechávat kvalifikace *jménem_prostoru_jmen* u identifikátorů z prostoru `jmen` s tímto jménem. To se ale netýká identifikátorů deklarovaných ve vnořených prostorech `jmen`.

Příklady

Jako příklad použijeme knihovni prostor `jmen System`, s nímž jsme se setkali již v předchozích kapitolách. V tomto prostoru `jmen` je deklarována např. třída `Console`.

V prostoru `jmen System` je ovšem vnořena řada dalších prostorů `jmen`, např. kontejnerové třídy najdeme v prostoru `jmen System.Collections`.

Napišeme-li

```
using System;
```

můžeme v následujícím programu psát `Console` místo `System.Console`, nemůžeme ale psát `Collections.ArrayList` nebo dokonce jen `ArrayList`; musíme vždy napsat `System.Collections.ArrayList`. Chceme-li vynechávat kvalifikaci u jména `ArrayList`, musíme náš program začínat direktivami

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

Přejmenování

Klíčové slovo `using` můžeme použít také k přejmenování prostoru `jmen` nebo typu. Tato tzv. *deklarace using* má tvar

deklarace using:

- `using alias = jméno_prostoru_jmen ;`
- `using alias = jméno_typu ;`

Alias zde představuje nové jméno prostoru `jmen`, resp. typu, které můžeme používat místo jména původního. Poznamenejme, že *jméno typu* musí být vyjádřeno identifikátorem, nikoli klíčovým slovem. To znamená, že takto nelze zavést nové jméno např. pro typ `int`. (Lze ovšem zavést nové jméno pro typ `System.Int32` a tak dosáhnout stejného výsledku.)

Příklad

Použijeme-li v příkladu z tohoto oddílu deklaraci

```
using PD = Prvni.Druhy;
```

vytvoříme alias `PD`, který lze používat místo `Prvni.Druhy`. To znamená, že místo

```
Prvni.Druhy.U u = new Prvni.Druhy.U();
```

můžeme napsat

```
PD.U v = new PD.U();
```

Rozhodneme-li se přejmenovat typ `System.Collections.ArrayList` na `NafukovacíPole`, poslouží nám deklarace

```
using NafukovacíPole = System.Collections.ArrayList;
```

Novou instanci třídy `ArrayList` pak vytvoříme příkazem

```
NafukovacíPole p = new NafukovacíPole(); ✧
```

Při bližším studiu dokumentace zjistíme, že např. `string` je alias pro třídu `System.String` a `int` je alias pro typ `System.Int32`.

6.3 Atributy

Atributy³¹ jsou novinkou, s níž se v žádném jiném z běžně používaných programovacích jazyků nesetkáme. Umožňují vyjádřit dodatečné informace o seskupení, modulu, třídě, metodě, její složce nebo o parametru metody; zapisují se do hranatých závorek před deklarací.

deklarace atributů:

- `[seznam_atributů]`

Seznam atributů, to jsou jednotlivé *atributy*, oddělené čárkami. Samotný *atribut* můžeme popsat takto:

atribut:

- `cílnep jméno_parametrynep`

Atribut se implicitně vztahuje k následující deklaraci. Má-li se vztahovat k seskupení, musíme před ním jako *cíl* uvést `assembly:`, má-li se vztahovat k modulu, musíme před ním jako *cíl* uvést `module:`. Některé atributy lze aplikovat jen na některé cíle.

Za *jménem* atributu mohou následovat v závorkách parametry.

Podívejme se na příklad:

³¹ V OOP se termín *atributy* obvykle používá pro datové složky, příp. obecně pro složky tříd. Jeho význam v jazyce C# je odlišný.

```
[Serializable, Obsolete] public class Chacha{/* ... */ }
```

Zde jsme deklarovali třídu Chacha s atributy Obsolete a Serializable.

Některé atributy

Jazyk C# obsahuje řadu standardních atributů; my si zde řekneme jen o několika málo z nich. Vedle toho umožňuje definovat si své vlastní atributy. Toto téma však přesahuje rámec naší knihy.

Serializable Atribut Serializable říká, že třídu, strukturu, výčtový typ nebo delegát, na který je aplikován, lze serializovat (ukládat do souborů).

Obsolete Atribut Obsolete říká, že cíl, na který se vztahuje, je zastaralý a programátoři by ho neměli používat. Překladač při použití tohoto cíle vypíše varování.

Conditional Atribut Conditional(*symbol*) je definován v prostoru jmen System.Diagnostics a je určen pro ladění. Lze ho použít pouze na metodu typu void. Překladač vypustí všechna volání této metody, není-li pomocí direktivy #define definován³² *symbol*. Podívejme se na příklad:

```
#define LADÍM
using System.Diagnostics;
public class Příklad
{
    [Conditional(LADÍM)] public void Sleduj(int n)
    {
        System.Console.WriteLine("n = {0}", n);
    }
    public int f(int n) // Faktoriál, to tu už dlouho
    {
        // nebylo
        int s = 1;
        while(n > 1)
        {
            s *= n--; // Při ladění potřebujeme sledovat
            Sleduj(s); // změny hodnoty proměnné s
        }
        return s;
    }
}
```



Bude-li definován symbol LADÍM, bude metoda f() průběžně vypisovat hodnoty proměnné s. Po odstranění direktivy #define tyto výpisy zmizí. Tento příklad najdete na WWW v souboru Kap05\02\debug.cs.

C+

³² Pokud znáte jazyk C nebo C++, jistě si vzpomenete na podmíněný překlad pomocí direktiv #if a #endif. Smysl atributu Conditional je podobný, jeho výhodou ale je, že podmíněnost specifikujeme jen na jednom místě.

6 Proměnné a datové typy

V této kapitole se krátce zastavíme u deklarace proměnných v jazyce C# a pak si řekneme nejdůležitější informace o většině datových typech používaných v tomto jazyce.

6.1 Proměnné

Proměnná je, jak víme, pojmenované místo v paměti s určeným typem. Může jít o proměnnou deklarovanou v těle metody (tzv. lokální proměnnou), o formální parametr metody či o statickou nebo nestatickou datovou složku. Deklarace lokální proměnné má tvar

deklarace proměnné:

- **const**_{nep} *typ seznam_deklarátorů* ;

Použijeme-li modifikátor const, deklarujeme konstantu – proměnnou, jejíž hodnotu nebudeme moci měnit. *Typ* je typ proměnných zaváděných touto deklarací; může být vyjádřen některým z klíčových slov (např. int), identifikátorem třídy nebo některou ze složitějších konstrukcí, které poznáme později.

Seznam deklarátorů, to jsou jednotlivé deklarátory oddělené čárkami. *Deklarátor* je identifikátor deklarované proměnné, za kterým může následovat inicializace:

deklarátor:

- *identifikátor*
- *identifikátor = výraz*

Hodnota výrazu představuje počáteční hodnotu proměnné. V případě konstanty to musí být konstantní výraz, tj. výraz, který lze vypočítat už v době překladu.

Jazyk C# vyžaduje, abychom každé proměnné před prvním použitím přidělili hodnotu.

Typy a proměnné

Jazyk C# rozlišuje tři základní skupiny typů: První tvoří tzv. *hodnotové typy*, druhou pak *odkazy* (reference, referenční typy). Třetí skupinu tvoří ukazatele; zde je uvádíme jen pro úplnost, neboť leží mimo

hlavní směr programování v C# a lze je používat jen v tzv. nezabezpečeném kódu (unsafe code). V této knize o nich nebudeme hovořit.³³

Mezi hodnotové typy patří kromě tzv. primitivních (atomických) typů, což jsou datové typy vyjadřující čísla, znaky a logické hodnoty, také tzv. výčetové typy a struktury. Referenční typy obsahují odkazy na instance tříd, pole, rozhraní a některé další konstrukce.

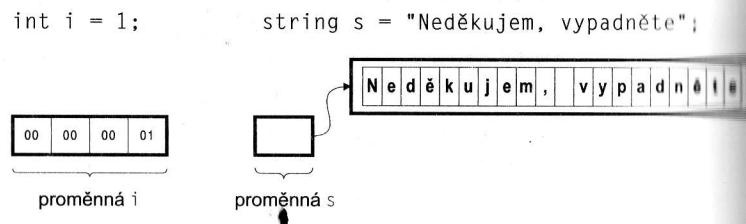
Základní rozdíl mezi hodnotovými a referenčními typy se týká zacházení s proměnnými. Deklarujeme-li proměnnou `i` typu `int`, který patří mezi hodnotové typy,

```
int i = 1;
```

bude identifikátor `i` označovat přímo místo v paměti, kde je uložena hodnota 1. Jestliže ale deklarujeme proměnnou `s` typu `string`, který patří mezi referenční typy,

```
string s = "Neděkujem, vypadněte";
```

bude identifikátor `s` označovat místo v paměti, v němž je uložen odkaz na jiné místo v paměti, obsahující instanci třídy `string` s uvedeným řetězcem. Situaci ukazuje obr. 6.1.



Obr. 6.1 Hodnotové a referenční typy

6.2 Hodnotové typy poprvé: atomické typy

V tomto oddílu se podíváme na typy vyjadřující celá a reálná čísla, znaky a logické hodnoty.

Celá čísla

V úvodních kapitolách jsme poznali typ `int`. Ovšem C# nabízí pro práci s celými čísly ještě 7 dalších datových typů; jejich vlastnosti shrnuje následující tabulka 6.1.

Typ	B	Rozsah
sbyte	1	-128 .. 127
short	2	-32 768 .. 32 767
int	4	-2 147 438 648 .. 2 147 438 647
long	8	-9 223 372 036 854 775 808 .. 9 223 372 036 854 775 807
byte	1	0 .. 255
ushort	2	0 .. 65535
uint	4	0 .. 4 294 877 295
ulong	8	0 .. 18 446 744 073 709 551 615

Tabulka 6.1 Celočíselné typy v C#. Ve sloupci **B** je uveden počet bajtů, které proměnná tohoto typu zabírá v paměti

V této tabulce si přečteme, že např. proměnná typu `sbyte` bude vždy zabírat jeden bajt a můžeme do ní uložit jakékoli celé číslo v rozmezí od -128 do +127, včetně těchto krajních hodnot. První čtyři typy, `sbyte`, `short`, `int` a `long`, jsou tzv. typy se znaménkem; to znamená, že mohou obsahovat jak kladná, tak i záporná čísla. Druhou čtveřici tvoří typy tzv. bez znaménka; proměnné těchto typů mohou obsahovat jen nezáporná čísla.

Literály

Ve třetí kapitole jsme se dozvěděli, že celočíselné konstanty (literály) v programu zapisujeme podobně jako v běžném životě, až na to, že nesmíme používat žádné oddělovače tisíců. Můžeme tedy napsat 12345, nesmíme ale napsat 12 345. Tento zápis vychází z desítkové číselné soustavy.³⁴

Celočíselné konstanty můžeme ovšem také zapisovat v šestnáctkové (hexadecimální) soustavě. Přitom jako číslice s významem 10, ..., 15 používáme písmena A, ..., F, případně a, ..., f. Konstanta v šestnáctkové soustavě začíná 0x (nula a písmeno x). Například šestnáctková konstanta 0xB znamená v desítkové soustavě číslo 11, konstanta 0x1B znamená 27 a 0xFFFF znamená 65535.

Celočíselná konstanta je implicitně typu `int`, pokud se její hodnota vejde do rozsahu tohoto typu. Pokud ne, bude typu `uint`, `long` nebo `ulong` (prvního, do jehož rozsahu se vejde).

³³ Jde o ukazatele, jak je známe z jazyků C a C++, včetně adresové aritmetiky.

³⁴ K číselným soustavám se vrátíme v příkladu za tímto oddílem.

Občas ale potřebujeme zapsat např. hodnotu 1 jako konstantu některého jiného typu. K tomu můžeme použít přípony U, L a UL. Literál s příponou U bude typu `uint` nebo `ulong` (pokud se nevejde do rozsahu typu `uint`). Literál s příponou L bude typu `long` nebo `ulong` a literál s příponou UL bude vždy typu `ulong`.

Například konstanta `1L` je typu `long`, konstanta `1U` je typu `uint`.

Přetypování

Jazyk C# dovoluje přiřadit proměnné typu s větším rozsahem hodnotu typu s menším rozsahem a přiřazovaná hodnota se automaticky převede na hodnotu cílového typu. (Říkáme, že proběhne *implicitní konverze* neboli *implicitní přetypování*.) Přesněji: Jestliže se všechny hodnoty typu výrazu na pravé straně přiřazení vejdou do rozsahu typu proměnné na levé straně, bude vše v pořádku. Jinak nám překladač toto přiřazení nedovolí.

Příklad

Máme dvě proměnné,

```
int i = 6554;
long l;
```

Přiřazení

```
l = i;
```

přijme překladač bez námitek, neboť všechny hodnoty typu `int` se vejdou do rozsahu typu `long`. Na druhé straně přiřazení

```
i = l; // NELZE
```

překladač označí za chybné, neboť některé hodnoty typu `long` nelze uložit do proměnné typu `int`. Pokud na takovém přiřazení přece jen trváme, musíme použít explicitní přetypování:

```
i = (int)l; // OK
```

Pokud hodnota proměnné `l` leží v rozsahu typu `int`, bude vše v pořádku; jinak se přenesou jen některé z bitů tvořících reprezentaci čísla `l` a výsledek nejspíš nebude mít žádný smysl. ✧

Podobná pravidla platí i při předávání parametrů metodám.

Operace celými čísly

Základní operace s celými čísly už vlastně známe. Operátory `+`, `-` a `*` slouží ke sčítání, odečítání a násobení; dělení vyjadřujeme lomítkem, tj. znakem `/`. Dělnice i dělitele zapisujeme do stejné řádky, např. `x/y`. Chceme-li na místě dělnice nebo dělitele použít složitější výrazy, musíme je uzavřít je do závorek, například `(x-y)/(z+v)`. Dělíme-li dvě celá čísla, je výsledkem opět celé číslo, tedy např. `22 / 8` je `2`.

Pomocí operátoru `%` zjistíme zbytek po dělení. Např. `22 % 8` je `6`.

Relační operátory

Dále můžeme použít unární operátory `plus` a `minus`. Pro celé číslo `x` znamená `+x` stejnou hodnotu jako `x` a `-x` znamená číslo se stejnou absolutní hodnotou a s opačným znaménkem.

Celá čísla můžeme porovnávat pomocí operátorů `>`, `<`, `>=` a `<=`, které znamenají po řadě větší, menší, větší nebo rovno a menší nebo rovno. Pomocí operátorů `==`, resp. `!=` zjišťujeme, zda jsou si dvě čísla rovna, resp. zda si nejsou rovna. Tyto tzv. relační operátory vytvářejí hodnotu typu `bool` (logickou hodnotu, s významem pravda nebo nepravda).

Na celočíselné proměnné (obecně na celočíselné l-hodnoty, tj. na jakékoli výrazy, které mohou stát na levé straně přiřazení) můžeme také použít operátory `++` a `--`. První z nich zvětší hodnotu proměnné o 1, druhý ji zmenší o 1. Podrobněji jsme o nich hovořili v kapitole 3.4.

Se všemi výše uvedenými operátory jsme se už setkali v předchozích kapitolách. Vedle nich můžeme pro celá čísla použít tzv. bitové operátory `&`, `|`, `^`, `<<` a `>>`. Výklad o nich ale přesahuje rámec naší knihy.

Kontrola přetečení

Ve 3. kapitole jsme napsali několik verzí metody `f(int n)`, která počítala faktoriál celého čísla. Přitom jsme zjistili, že pro hodnoty parametru `n > 12` nedává správný výsledek, neboť vypočtené číslo se nevejde do rozsahu typu `int`.

Mohli bychom zkusit použít typ `ulong`; to ale znamená, že musíme najít novou mez, pro níž má výpočet smysl. Po několika experimentech a troše otravného počítání zjistíme, že největší hodnota, pro kterou můžeme dostat správný výsledek, je 20. (Faktoriál roste opravdu velice rychle.)

checked

S podobnými problémy se můžeme setkat i v jiných výpočtech, a protože výsledek po celočíselném přetečení *obvykle* postrádá smysl, je rozumné zařídit, aby se o něm uživatel programu dozvěděl. Jazyk C# nám nabízí operátor `checked`, který způsobí, že celočíselné přetečení (při výpočtu nebo při přetypování) vyvolá výjimku.

Podobně je k dispozici i operátor `unchecked`, který naopak hlášení o přetečení potlačí.

Syntaxe jejich použití je

použití operátorů checked a unchecked:

- `checked(výraz)`
- `unchecked(výraz)`

Kromě toho můžeme předepsat nebo naopak potlačit kontrolu přetečení pro celou skupinu příkazů. Uzavřeme je do složených závorek a před ně napíšeme klíčové slovo `checked` nebo `unchecked`:

příkazy *checked* a *unchecked*:

- **checked** { *příkazy* }
- **unchecked** { *příkazy* }

Může se to zdát divné, ale některé výpočty aritmetického přetečení opravdu využívají. Proto nabízí C# také možnost jeho hlášení potlačit.

Příklad: ještě jednou faktoriál

Použijeme kontrolu celočíselného přetečení při výpočtu faktoriálu. V metodě `f()` z konce oddílu 3.4 (je v souboru `Kap03\04\fakt6.cs` na WWW) může k přetečení dojít jen na jediném místě, při násobení. Proto ho ošetříme pomocí operátoru `checked`:

```
static ulong f(int n)
{
    if(n < 0) throw new Exception("Faktoriál "
                                   + n + " není definován");
    ulong s = 1;    // Pomocná proměnná, obsahuje výsledek
    while(n > 1)
    {
        s = checked(s * (ulong)n--);    // Při přetečení
    }                                    // způsobí výjimku
    return s;
}
```

Všimněte si, že jsme z této metody mohli vypustit kontrolu, zda hodnota vstupního parametru nepřesahuje hodnotu konstanty `Mez`. Také deklaraci této konstanty jsme třídy `Výpočet` odstranili.

Místo operátoru `checked` jsme mohli také použít příkaz `checked` = mohli jsme do něj uzavřít celý cyklus `while`:

```
checked {    // Kontrola celočíselného přetečení
    while(n > 1)
    {
        s = s * (ulong)n--;
    }
}
```



Zdrojový text obou úprav programu pro výpočet faktoriálu najdete na WWW v adresáři `Kap06\01` v souborech `fakt6.cs` a `fakt7.cs`.

Znaky

Pro práci s jednotlivými znaky má C# datový typ `char`. Proměnná typu `char` zabírá dva bajty, neboť C# používá šestnáctibitové kódování UNICODE. (Proto také bez problémů zvládá nejen angličtinu, ale i jiné jazyky, mezi nimi i češtinu.)

Znakové konstanty (konstanty typu `char`) můžeme v programu zapsat několika způsoby:

- Běžné znaky prostě uzavřeme mezi apostrofy, např. `'a'`, `'ž'`.
- Řídící znaky, jako je přechod na nový řádek, tabulátor ap., můžeme vyjádřit některou z řídících posloupností (escape sequence) uvedených v tabulce 6.2 a uzavřených mezi apostrofy, např. `'\t'`.

Znak	UNICODE	Význam
<code>\n</code>	<code>\u000A</code>	Přechod na nový řádek
<code>\r</code>	<code>\u000D</code>	Návrat na počátek řádku
<code>\t</code>	<code>\u0009</code>	Tabulátor
<code>\b</code>	<code>\u0008</code>	Návrat o jeden znak
<code>\f</code>	<code>\u000C</code>	Nová stránka
<code>\v</code>	<code>\u000B</code>	Vertikální tabulátor
<code>\a</code>	<code>\u0007</code>	Zvukový signál
<code>\"</code>	<code>\u0022</code>	Uvozovky
<code>\\</code>	<code>\u005C</code>	Obrácené lomítko, znak „\“
<code>\'</code>	<code>\u0027</code>	Apostrof
<code>\0</code>	<code>\u0000</code>	Znak s kódem 0

Tab. 6.2 Řídící posloupnosti v C#

- Můžeme také použít řídící posloupnost tvaru `\xhhhh`, kde `hhhh` jsou nejvýše 4 šestnáctkové číslice vyjadřující kód daného znaku. Tak lze vyjádřit libovolný znak; např. zápis `'\x41'` představuje znak `'A'`, `'\xDD'` nebo `'\x00DD'` představuje znak `'Ý'`.
- Můžeme použít vyjádření v kódu UNICODE ve tvaru `'\uhhhh'`, kde `hhhh` jsou 4 šestnáctkové číslice (ani více, ani méně) vyjadřující kód daného znaku. Tabulka 6.3 na následující straně ukazuje kódy znaků české a slovenské národní abecedy.

Řetězcové literály

Ze 3. kapitoly víme, že v C# můžeme používat dva druhy řetězcových literálů: *běžné* a *doslovné* (*regular*, resp. *verbatim string literals*).

Běžný řetězcový literál se skládá z posloupnosti znakových konstant zapsaných buď přímo nebo pomocí řídících posloupností; znaky z tabulky 6.2 musíme vyjádřit pomocí řídících posloupností.

á	\u00E1	ó	\u00F3	Á	\u00C1	Ó	\u00D3
ä	\u00E4	ô	\u00F4	Ä	\u00C4	Ö	\u00D4
č	\u010D	ř	\u0155	Č	\u010C	Ř	\u0154
ď	\u010F	ř	\u0159	Ď	\u010E	Ř	\u0158
é	\u00E9	š	\u0161	É	\u00C9	Š	\u0160
ě	\u011B	ť	\u0165	Ě	\u011A	Ť	\u0164
í	\u00ED	ú	\u00FA	Í	\u00CD	Ú	\u00DA
ř	\u013E	ů	\u016F	Ř	\u013D	Ů	\u016E
í	\u013A	ý	\u00FD	Í	\u0139	Ý	\u00DD
ň	\u0148	ž	\u017E	Ň	\u0147	Ž	\u017D

Tab. 6.3 Kódy UNICODE českých a slovenských znaků s diakritickými znaménky

Doslovný řetězcový literál může obsahovat jakýkoli zobrazitelný znak kromě uvozovek a k tomu přechod na nový řádek a tabulátor. Řidič posloupnosti se v něm ignorují. To znamená, že např. příkaz

```
System.Console.WriteLine(@"\xDD
s");
```

vypíše

```
\xDD
s
```

Chceme-li v doslovném řetězcovém literálu zapsat uvozovky, zdvoji-
me je. Příkaz

```
System.Console.WriteLine(@"Program ""ŽlutýKůň""");
```

vypíše

```
Program "ŽlutýKůň"
```

Znaky jako čísla

Typ `char` se chová jako číselný typ bez znaménka s hodnotou odpoví-
dající kódu daného znaku. To znamená, že hodnoty tohoto typu mů-
žeme používat ve výrazech spolu s celými čísly. Můžeme na ně také
používat tytéž operátory jako na celá čísla.

Reálná čísla

Vedle označení reálná čísla uvidíte také názvy „desetinná čísla“ nebo
„čísla s plovoucí řádovou čárkou“. C# má dva datové typy pro repre-

zentaci reálných čísel, které se nazývají `double` a `float`. Jejich vlast-
nosti shrnuje tabulka 6.4.

Typ	B	Rozsah	Platných cifer
float	4	$1,4 \times 10^{-45} \dots 3,4 \times 10^{38}$	7–8
double	8	$4,9 \times 10^{-324} \dots 1,7 \times 10^{308}$	15–16

Tab. 6.4 Datové typy vyjadřující reálná čísla v C#. Ve sloupci **B** je uveden počet bajtů, které proměnná tohoto typu zabírá v paměti

Z této tabulky zjistíme, že např. do proměnné typu `double` můžeme uložit číslo x , pro které platí nerovnosti $4,9 \times 10^{-324} \leq x \leq 1,7 \times 10^{308}$ nebo je rovno nule nebo leží v rozmezí $-1,7 \times 10^{308} \leq x \leq -4,9 \times 10^{-324}$.

Vedle toho mohou obsahovat jisté speciální hodnoty, jako je „strojové nekonečno“ (hodnotu, která se chová jako $\pm\infty$) a NaN, hodnotu, která není číslem a která vznikne při některých výpočetních chybách. Těmi se v této knize ale nebudeme zabývat.

Literály

Reálné konstanty (tedy desetinná čísla) můžeme v programu zapsat několika způsoby.

- První možnost představuje tzv. zápis s pevnou řádovou tečkou. Číslo zapíšeme podobně jako v běžném životě, jen desetinnou čárku nahradíme tečkou a neděláme mezery mezi trojicemi číslic. Například číslo 65 987,626 221 zapíšeme 65987.626221.
- Můžeme použít tzv. semilogaritmický³⁵ tvar. To je zápis odvozený od způsobu, jakým běžně zapisujeme velmi velká nebo velmi malá čísla (např. $1,4 \times 10^{-45}$), ve kterém desetinnou čárku nahradíme desetinnou tečkou, vypustíme symbol násobení, desítku zaměníme znakem `e` nebo `E` a exponent zapíšeme do stejné řádky jako zbytek čísla. Číslo $1,4 \times 10^{-45}$ tedy v semilogaritmickém tvaru zapíšeme 1.4e-45 nebo 1.4E-45. Číslo před symbolem `e` nebo `E` nemusí obsahovat desetinnou část; můžeme také napsat např. 14E-46.
- Reálné literály jsou typu `double`. Chceme-li literál typu `float`, připojíme k němu příponu `f` nebo `F`. K literálu typu `double` můžeme připojit příponu `d` nebo `D`.

Výpis
reálných čísel

Při výstupu se reálná čísla formátují podle místních zvyklostí, nastavených v operačním systému. To znamená, že např. na počítači

³⁵ V návodech ke kalkulačkám se mu z jakýchsi záhadných důvodů říká „vědecká notace“, vzdor skutečnosti, že se učí v 8. třídě základních škol.

s českými Windows se bude při výstupu používat desetinná čárka, nikoli tečka.

Operace s reálnými čísly

Význam unárních operátorů $+$, $-$, $++$ a $--$ a binárních operátorů $+$, $-$, $*$ a $/$ pro reálná čísla je podobný jako pro celá čísla. Operátor $%$ vypočte opět zbytek po dělení, tentokrát bude ovšem výsledkem reálné číslo; např. $11.0 \% 2.5$ je rovno 1 . Při přiřazování můžeme kromě operátoru $=$ použít také složené operátory $+=$, $*=$ a další.

Pro porovnávání reálných čísel máme k dispozici operátory $<$, $<=$, $>$, $>=$, $!=$ a $==$, podobně jako pro porovnávání celých čísel. Bitové operátory nelze pro reálná čísla použít.

Ve výrazu se můžeme vedle sebe používat celá a reálná čísla. Výsledkem pak bude reálné číslo; o tom si podrobněji povíme v 7. kapitole.

Přiřazování

Přiřazení celočíselné hodnoty proměnné typu `float` nebo `double` je bez problémů. Podobně přiřazení hodnoty typu `float` proměnné typu `double`; vždy proběhne automatická konverze (přetypování), neboť se při něm nemůže ztratit žádná informace (cílový typ má vyšší přesnost než přiřazovaná hodnota)

Při ostatních přiřazeních musíme použít explicitní přetypování.

Zaokrouhlovací chyby

Protože jsou reálná čísla v paměti uložena ve 4, resp. v 8 bajtech, nelze v počítači zobrazit jakákoli čísla, ale jen některá. Naše data, stejně jako výsledky, nemohou být libovolně přesná – budou obsahovat jen několik platných cifer. Vyjde-li jako výsledek číslo, které nelze zobrazit, zaokrouhlí počítač výsledek na nejbližší zobrazitelnou hodnotu.

Ve většině případů vystačíme s přesností, kterou nám typy `float` a `double` poskytují. Nicméně jsou situace (a bohužel nijak vzácné), kdy chyby vzniklé zaokrouhlováním mohou zcela pokazit výpočet.

Tradiční příklad představuje cyklus

```
double x = 0;
while(x != 1)    // Nekonečný cyklus
{
    System.Console.WriteLine(x);
    x += 0.1;
}
```

Tento cyklus by měl v ideálním případě vypsát čísla 0.1 , 0.2 atd. až po 0.9 . Ve skutečnosti však nikdy neskončí, neboť číslo x nebude nikdy zcela přesně rovno 1 . Problém je v tom, že číslo 0.1 je ve dvojkové soustavě periodické – má tvar $0.00011001100\dots$. To znamená, že počítač si ho zaokrouhlí na nejbližší zobrazitelnou hodnotu, která ale

nebude přesně rovna 0.1 , takže ani její desetinásobek nebude roven přesně 1 . Na druhé straně číslo 1.0 se v počítači dá vyjádřit naprosto přesně, takže podmínka ukončení cyklu nikdy nenastane. Tento příklad ukazuje, že porovnávání reálných čísel pomocí operátorů $==$ a $!=$ může vést k nepříjemným chybám a proto je lepší se mu vyhnout.

Jednoduchý program, který obsahuje výše uvedený cyklus, najdete na WWW v souboru `Kap06\02\furt.cs`. Spustíte-li ho z příkazové řádky, budete ho muset ukončit stisknutím klávesové kombinace `Ctrl+C`.

Matematické konstanty a funkce

Téměř všechny programovací jazyky poskytují také nástroje pro matematické výpočty – goniometrické a jiné běžné funkce; v C# jsou soustředěny ve třídě `Math`, která obsahuje pouze statické datové složky a metody.

Najdeme tu důležité matematické konstanty E (základ přirozených logaritmů, $e = 2.718281\dots$) a PI (π , Ludolfovo číslo). Některé z metod ukazuje tabulka 6.5. Tyto metody očekávají parametr typu `double` a většina z nich také vrací hodnotu typu `double`.

Metoda	Význam	Metoda	Význam
<code>Abs()</code>	absolutní hodnota	<code>Asin()</code>	arkussinus
<code>Atan()</code>	arkustangens	<code>Atan2()</code>	arkustangens podílu
<code>Ceiling()</code>	nejbližší vyšší celé číslo	<code>Cos()</code>	kosinus
<code>Exp()</code>	e^x	<code>Floor()</code>	celá část čísla
<code>Log()</code>	přirozený logaritmus	<code>Max()</code>	větší ze dvou čísel
<code>Min()</code>	menší ze dvou čísel	<code>Sin()</code>	sinus
<code>Sqrt()</code>	druhá odmocnina	<code>Tan()</code>	tangens
<code>Tanh()</code>	hyperbolický tangens	<code>Round()</code>	zaokrouhlení

Tab. 6.5 Některé ze statických metod třídy `Math`

Při volání musíme tyto funkce kvalifikovat jménem třídy `Math`, např. `Math.sin(x)`. To je sice poněkud nezvyklé, plyne to ale ze skutečnosti, že C# je čistě objektový jazyk, a proto musí být každá funkce součástí nějaké třídy.

Logické hodnoty

Logické hodnoty vyjadřují, zda nějaké tvrzení platí nebo neplatí. V C# nám k tomu slouží datový typ `bool`, který má pouze dvě hodnoty,

vyjádřené klíčovými slovy `true` a `false`. První z nich znamená pravdu a druhé nepravdu.³⁶ Logické hodnoty vznikají jako výsledky použití relačních operátorů `<`, `<=` a dalších a můžeme je používat mj. v podmínce cyklů `while` nebo příkazů `if`.

Pro vytváření složitějších výrazů (např. z několika podmínek) nám poslouží tzv. logické operátory. Sem patří operátor negace `!`, operátor logického součinu (konjunkce) `&&` a operátor logického součtu (disjunkce) `||`. Operátor konjunkce `&&` použijeme, jestliže chceme, aby dvě podmínky platily zároveň, kdežto operátor disjunkce `||` použijeme, chceme-li, aby platila alespoň jedna z podmínek. Operátor negace je unární a použijeme ho, jestliže chceme, aby negovaná podmínka neplatila. Význam těchto operátorů shrnuje tab. 6.6.

A	B	!A	A && B	A B
false	false	true	false	false
false	true	true	false	true
true	false	false	false	true
true	true	false	true	true

Tab. 6.6 Význam logických operátorů

Z této tabulky vyčteme, že má-li proměnná `A` hodnotu `true` a proměnná `B` hodnotu `false`, má výraz `A || B` hodnotu `true`.

Vedle toho můžeme logické hodnoty porovnávat pomocí operátorů `==` a `!=`.

Z tabulky 6.6 je vidět, že má-li první operand hodnotu `false`, bude výsledek operace konjunkce `A && B` nepravdivý, ať má druhý operand jakoukoli hodnotu. Proto je-li první operand nepravdivý, druhý se již nevyhodnotí (nevypočte), protože ho není třeba k určení hodnoty výsledku.

Podobně má-li první operand hodnotu `true`, bude výsledek operace disjunkce `A || B` pravdivý, ať má druhý operand jakoukoli hodnotu. Proto je-li první operand pravdivý, druhý se již nevyhodnotí. (Tomu se říká *neúplné vyhodnocení logického výrazu*.)

Příklad V kapitole 4.1 jsme použili podmínku

```
(i < řádek.Length) && (MEZERY.IndexOf(řádek[i]) == -1)
```

³⁶ Na rozdíl od jazyků `C` a `C++` nepatří v `C#` typ `bool` mezi číselné typy. Číselné hodnoty nelze v `C#` konvertovat na typ `bool`.

která vyjadřovala požadavek, aby index `i` byl menší než délka řetězce `řádek` a zároveň aby znak s indexem `i` neležel v řetězci `MEZERY`.

Přitom jsme se nemuseli bát, že při `i >= řádek.Length` překročíme meze řetězce a způsobíme chybu: Není-li první podmínka splněna, druhá se již nebude vyhodnocovat.

Desítková čísla

Jazyk `C#` nabízí také typ `decimal`, který vnitřně pracuje s čísly v desítkové soustavě. Lze do něj ukládat kladné hodnoty v rozmezí přibližně od $1,0 \times 10^{-28}$ do $7,9 \times 10^{28}$ a záporné hodnoty v analogickém rozsahu, a to s přesností na 28–29 platných cifer.

Hodnoty typu `decimal` jsou v paměti uloženy v desítkové soustavě, takže např. číslo `0,1` se zobrazí přesně. (Počítač si pamatuje jednotlivé cifry a polohu desetinné tečky.)

Literály

Literál tohoto typu zapisujeme jako reálné číslo ve tvaru s pevnou řádovou tečkou, k němuž připojíme příponu `m` nebo `M`. Například

```
decimal x = 7654.321054m; // Bude uloženo přesně
```

Tento typ se používá především pro finanční výpočty. Jsou pro něj definovány podobné operace jako pro ostatní číselné typy; při přiřazování hodnot typu `decimal` proměnným ostatních číselných typů, stejně jako při přiřazování hodnot reálných typů proměnné typu `decimal` musíme použít explicitní přetypování.

Prázdný „typ“ void

C⁺

Z ryze formálních důvodů se mezi základními typy někdy uvádí i `void` (jde ovšem spíše o setrvačnost vzhledem k jazykům `C` a `C++`). Klíčové slovo `void` lze ovšem použít pouze ke specifikaci metod, které nevracejí žádný výsledek.

Příklad: Převod do jiné číselné soustavy

V tomto příkladu si ukážeme použití některých dalších operátorů pro práci s celými čísly. Naším úkolem bude napsat třídu `Konvertor`, jejíž veřejně přístupná metoda `konvertuj()` převede číslo typu `long` na znakový řetězec v zadané číselné soustavě o základu v rozmezí od 2 do 36.

Nejprve si zopakujeme školní vědomosti o číselných soustavách. My jsme zvyklí na tzv. desítkovou soustavu (soustavu se základem 10). V ní např. zápis `657` znamená číslo, které obsahuje 7 jednotek, 5 desítek a 6 stovek, tedy $6 \times 10^2 + 5 \times 10^1 + 7 \times 10^0 = 657$. Podobně se vyjadřují i

čísla v jiných číselných soustavách, pouze místo mocnin 10 se používají mocniny jiného základu. Např. číslo 1010 ve dvojkové soustavě znamená $1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$, tedy 10 v desítkové soustavě. Podobně číslo 0xab, zapsané v šestnáctkové soustavě, převedeme do desítkové soustavy, uvědomíme-li si, že tento zápis neznámá ne jiného než $10 \times 16^1 + 11 \times 16^0 = 10 \times 16 + 11 \times 1 = 171$.

Z toho vyplývá, že hodnotu poslední číslice čísla x v číselné soustavě se základem z zjistíme, když vypočteme zbytek po dělení čísla x základem z této soustavy. Např. $657 \% 10$ je 7, takže poslední číslice tohoto čísla v soustavě o základu 10 je 7.

Při zjišťování předposlední číslice můžeme postupovat podobně, než dříve se ale musíme zbavit poslední číslice. Toho dosáhneme například, že původní číslo vydělíme základem. (Abychom zjistili předposlední číslici čísla 657, vezmeme číslo $657/10 = 65$ a zjistíme jeho poslední číslici.)

Tedy už můžeme navrhnout algoritmus, jak postupně zjistit jednotlivé číslice vyjadřující dané číslo v soustavě o základu z .

1. Nejprve zjistíme, zda je dané číslo n záporné. Pokud ano, zapamatujeme si to a dále budeme pracovat s kladným číslem $-n$.
2. Hodnotu poslední číslice zjistíme vypočtením $n \% z$ a zapamatujeme si ji.
3. Číslo n nahradíme číslem n / z (celočíslné dělení základem).
4. Je-li $n != 0$ (není-li n rovno 0), vrátíme se na krok 2.
5. Na konec získané posloupnosti číslic připojíme znaménko.

Uvedený postup nám poskytne hodnoty jednotlivých číslic v soustavě o základu z , ovšem v obráceném pořadí. Není ale velký problém přepsat jej pak obráceně.

Třída `Konvertor` bude obsahovat soukromou datovou složku `základ`, v níž bude uložen základ cílové číselné soustavy. Dále bude obsahovat soukromou statickou (tj. pro všechny instance společnou) složku

```
private static const string číslice =
    "0123456789ABCDEFGHIJKLMNPOQRSTUVWXYZ";
```

obsahující všechny znaky, které použijeme jako číslice. (Je jich celkem 36, proto jsme si na počátku položili omezení, že nejvyšší základ bude právě 36. V soustavě s tímto základem bude znak `Z` představovat číslici s hodnotou 35.) Vlastní metoda `konvertuj()` bude jednoduchá:

```
public string konvertuj(int co)
{
    StringBuilder číslo = new StringBuilder("");
    bool záporné = false;    // Je co záporné?
```

```
    if(co == 0)                // Je-li to 0, jsem hotoví
    {
        return "0";
    }

    if(co < 0)                 // Je-li záporné, zapamatuj
    {                          // si to a pracuj s -co
        záporné = true;
        co = -co;
    }

    while(co != 0)             // Dokud nenajdeme všechny
    {                          // číslice
        číslo.Append(ČÍS_LICE[co % základ]);
        co /= základ;          // Odstraň poslední číslici
    }

    if(záporné) číslo.Append("-"); // K zápornému připoj "-"

    // Přepiš znaky v obráceném pořadí
    int i = 0, j = číslo.Length-1;
    while(i < j)               // Prohoď i-tý a j-tý znak
    {
        char pom = číslo[i];
        číslo[i++] = číslo[j];
        číslo[j--] = pom;
    }

    return číslo.ToString();   // Vrať nalezený řetězec
}
```

Poznámky:

- Jednotlivé znaky v instanci třídy `StringBuilder` jsou dostupné pomocí indexování, podobně jako u třídy `string`; jsou indexovány od 0 a jejich počet zjistíme pomocí složky `Length`.
- Výraz `ČÍS_LICE[co % základ]` představuje hledanou číslici. Výraz `co % základ` představuje zbytek po dělení čísla `co` základem soustavy a jednotlivé číslice jsou v řetězci `ČÍS_LICE` uloženy tak, že 0 je uložena na pozici s indexem 0, 1 je uložena na pozici s indexem 1, ..., Z je uloženo na pozici s indexem 35.
- Při otáčení řetězce použijeme dvě pomocné proměnné, `i` a `j`. Na počátku `i` obsahuje 0, tedy index počátečního znaku, a `j` obsahuje `Length-1`, tedy index posledního znaku. Po jejich prohození přejde `i` na druhý a `j` na předposlední znak atd. Když se tyto dvě proměnné setkají, prohazování skončí.



Zdrojový text tohoto příkladu najdete na WWW v souboru `Kap06\02\konvert.cs`

6.3 Pole: skupina proměnných stejného typu

O poli jsme si řekli, že to je skupina proměnných stejného typu, se kterou zacházíme jako s celkem. Proměnné v této skupině nazýváme prvky pole; jsou očíslovány (indexovány) a s jednotlivými prvky pracujeme pomocí jejich pořadových čísel (indexů).

Počet prvků Počáteční prvek má vždy index 0, takže v poli s N prvky má poslední prvek index $N-1$. Počet prvků v poli můžeme zjistit pomocí složky `Length`.

S poli v C# pracujeme pomocí referencí.

Deklarace a vytvoření pole

Deklarace pole Deklaraci odkazu na pole v C# můžeme zjednodušeně popsat takto:
deklarace pole:

- `typ [] seznam_identifikátorů ;`
- `typ [,] seznam_identifikátorů ;`
- ...

Typ je typ prvků tohoto pole (říkáme také *typ pole*). Následující prázdné hranaté závorky říkají, že deklarujeme pole, nikoli „obyčejnou“ proměnnou. *Seznam identifikátorů*, to jsou identifikátory deklarovaných odkazů.

V deklaraci odkazu neuvádíme počet prvků pole. Ten určíme později, až budeme pole vytvářet. (Jedna proměnná tedy může postupně obsahovat odkazy na pole s různým počtem prvků, mají-li tyto prvky stejný *typ*.)

První z možností v uvedeném popisu syntaxe obsahuje prázdné lomené závorky a slouží k deklaraci jednorozměrného pole (pole s jedním indexem). Druhá možnost obsahuje v lomených závorkách čárku a slouží k deklaraci pole se dvěma indexy (dvourozměrného). Ovšem tímto způsobem nekončí: Chceme-li pole s k indexy, zapíšeme v lomených závorkách $k-1$ čárek.

Dále si budeme povídat o jednorozměrných polích a nakonec si vysvětlíme, v čem se liší vícerozměrná pole.

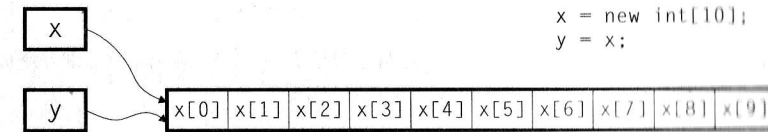
Vytvoření pole K vytvoření pole použijeme operátor `new`. Tento operátor vrátí odkaz na nově vytvořené pole. Za operátorem `new` uvedeme typ prvků a v lomených závorkách jejich počet.

Příklad Vezměme příkazy

```
int [] x, y; // Deklarace odkazů
```

```
x = new int[10]; // vytvoření nového pole  
y = x;          // přiřazení odkazu
```

Prvním z těchto příkazů deklarujeme dva odkazy na jednorozměrné pole s prvky typu `int`. Druhým vytvoříme pole o 10 prvcích a odkaz na ně uložíme do proměnné `x`. Třetím příkazem okopírujeme tento odkaz do proměnné `y`, takže jak `x`, tak `y` budou odkazovat na totéž pole. Výsledek ukazuje obr. 6.2. ✧



Obr. 6.2 Přiřazování odkazů na pole

Pole zanikne, jestliže zaniknou všechny odkazy na ně.

Práce s prvky

Už víme, že s prvky polí pracujeme pomocí indexů. Ty zapisujeme do lomených závorek za identifikátor odkazu. Prvek s indexem 5 z pole `x` získáme zápisem `x[5]` (nebo také `y[5]`, neboť `x` i `y` odkazují na totéž pole).

Program v C# vždy kontroluje, zda nepoužíváme neinicializovaný odkaz na pole a zda zadaný index leží v povoleném rozsahu. Pokud bychom se pokusili indexovat odkaz, který obsahuje `null`, vznikne výjimka typu `System.NullReferenceException`. Pokud bychom se pokusili použít index neexistujícího prvku, vznikne výjimka `System.IndexOutOfRangeException`.

Inicializace pole

Občas se stane, že prvky pole známe předem. V tom případě je můžeme zapsat přímo v deklaraci: Za identifikátor pole napíšeme rovnítko a pak ve složených závorkách jednotlivé prvky oddělené čárkami. Například takto:

```
double [] počátek = {0.0, 0.0, 0.0};
```

Program vytvoří pole o potřebném počtu prvků, uloží do něj dané hodnoty a odkaz na ně přiřadí proměnné `počátek`.

Také pole vytvořené pomocí `new` můžeme inicializovat. Prostě za zápis operátoru `new` připojíme do lomených závorek seznam hodnot:

```
double [] konec = new double[3] {1.0, 1.0, 1.0};
```

Předchozí možnost je jen zkratkou pro tento zápis.

Vícerozměrná pole

Při deklaraci a vytvoření pole s několika indexy postupujeme podobně jako v případě jednozměrného pole. Například deklarace

```
int [,] a = new int[2,3] {{1,2,3}, {4,5,6}};
```

zavádí pole *a* se 6 prvky typu *int*. Na toto pole se můžeme dívat jako na pole složené ze 2 jednozměrných polí o 3 prvcích typu *int*. Jeho prvky budou

```
a[0,0] a[0,1] a[0,2] a[1,0] a[1,1] a[1,2]
```

a budou v paměti ležet v uvedeném pořadí. Kdybychom si toto pole zapsali do tabulky (matice),

```
a[0,0] a[0,1] a[0,2]  
a[1,0] a[1,1] a[1,2]
```

určoval by první index číslo řádku a druhý číslo sloupce. Proto o prvcích se stejným prvním indexem hovoříme také jako o „řádku“ a o prvcích se stejným druhým indexem jako o „sloupci“ pole (matice). Pole *a* je tedy pole se dvěma řádky a třemi sloupci.

Počet rozměrů pole, tj. počet indexů, zjistíme pomocí složky *Rank*, např. *a.Rank == 2*.

Nepravidelná pole

Jazyk C# zná ještě tzv. *nepravidelná pole (jagged array)*. To jsou pole, jejichž řádky nemusí být stejně dlouhé. Deklarujeme je zápisem, v němž pro každý z indexů použijeme samostatné závorky, např.

```
int [][] z; // Deklarace odkazu
```

Nepravidelné pole vytváříme v několika krocích. Nejprve určíme počet řádků,

```
z = new int[2][]; // Vytvoření odkazů na řádky
```

Pak vytvoříme jednotlivé řádky:

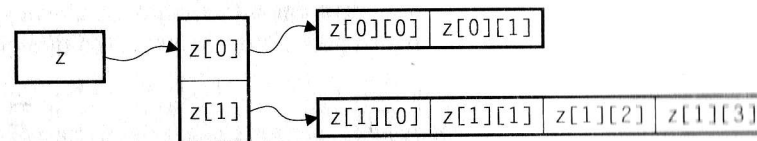
```
z[0] = new int[2]; // Vytvoření řádků  
z[1] = new int[4];
```

S prvky nepravidelného pole zacházíme podobně jako s prvky obyčejných polí, avšak každý index musíme uzavřít do samostatných závorek:

```
z[1][0] = 125;
```

Poznamenejme, že *z* je ve skutečnosti pole složené z prvků typu *int[]*, tedy pole obsahující odkazy na pole typu *int*. Situaci ukazuje obr. 6.3.

```
int[][] z = new int[2][];  
z[0] = new int[2];  
z[1] = new int[4];
```



Obr. 6.3 Nepravidelné pole

Jazyk C# dovoluje používat i vícerozměrná nepravidelná pole a dokonce složitější konstrukce, v nichž jsou prvky nepravidelných polí jiná pravidelná nebo nepravidelná pole.

Práce s poli

Kopírování polí

Protože s poli pracujeme pomocí odkazů, nemůžeme pro současné přenesení všech prvků jednoho pole do jiného použít přiřazení. Můžeme samozřejmě naprogramovat cyklus, který se o přenos hodnot postará:

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

Obvykle ale používáme elegantnější způsob, založený na metodě *CopyTo()*. Tato metoda je k dispozici ve všech polích a má dva parametry – cílové pole a počáteční index, od kterého se mají prvky do cílového pole uložit. Například příkazem

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

okopírujeme všechny prvky z pole *y* do pole *x* a uložíme je v něm počínaje indexem 0.

Pro okopírování části prvků z jednoho pole do jiného můžeme použít statickou metodu *Copy()* třídy *System.Array*. Obě metody lze ale bohužel použít pouze pro jednozměrná pole.

Další operace

Třída *System.Array* implementuje řadu užitečných operací. Zde se ve stručnosti zmíníme alespoň o statické metodě *Sort()*, která pole seřadí (seřadí jeho prvky podle velikosti) a o statické metodě *Reverse()*, která přerovná prvky pole v opačném pořadí. Metoda *BinarySearch()* slouží k vyhledání prvku v seřazeném poli. Podrobnější informace o nich najdete v dokumentaci.

Příklad

Napišeme třídu *Kalendář*, jež zjistí pořadové číslo dne v roce. Její metoda *ČísloDne()* dostane jako parametry datum rozložené na den,

měsíc a rok a vrátí celé číslo, které bude říkat, kolikátý den od počátku daného roku to je.

V této třídě si nejprve deklarujeme jako datovou složku pole `měsíce`, do kterého uložíme počet dnů v jednotlivých měsících:

```
int[] měsíce = { 31,28,31,30,31,30,31,31,30,31,30,31 };
```

V dalším poli, které nazveme `dny`, si uložíme počty dnů, které uplynuly od počátku roku vždy 1. dne daného měsíce. Prvek `dny[0]` bude obsahovat 0, neboť 1. ledna neuplynul ještě žádný den (a běží první), prvek `dny[1]` bude obsahovat 31, neboť prvního února uplynulo 31 dnů atd. Jinak řečeno, prvek `dny[i]` obsahuje součet prvků `mesice[0] + mesice[1] + ... + mesice[i-1]`. Obsah prvků tohoto pole vypočteme v konstruktoru třídy `Kalendář`.

```
class Kalendář
{
    int[] měsíce = {31,28,31,30,31,30,31,31,30,31,30,31};
    int[] dny = new int[12];

    public Kalendář()
    { // Konstruktor uloží do pole dny počty dnů,
      // které uběhly do začátku odpovídajícího měsíce
      int i = 1;
      while(i < dny.Length)
      {
          dny[i] = dny[i-1]+měsíce[i-1];
          ++i;
      }
    }
    // A další metody
}
```

Při výpočtu pořadového čísla dne musíme vzít v úvahu možnost, že existuje také přestupný rok, kdy má únor 29 dnů. V takovém případě musíme u dat následujících po 1. březnu přičíst 1.

Ke zjištění, zda je rok přestupný, použijeme metody `Přestupný()`, která bude vracet logickou hodnotu. Přestupný rok je rok, jehož číslo je dělitelné čtyřmi. Ovšem v případě celého století musí být číslo roku dělitelné 400:

```
public bool Přestupný(int rok) // Je daný rok přestupný?
{
    if((rok % 4 == 0) && (rok % 100 != 0) || (rok % 400 == 0))
    { return true; }
    else { return false; }
}
```

Nyní zbývá naprogramovat metodu `ČísloDne()`. Ta sečte počet dnů, které uplynuly od počátku roku do konce předchozího měsíce, a přičte den v měsíci.

```
public int ČísloDne(int den, int měsíc, int rok)
{
    int d = dny[měsíc-1] + den;
    if(Přestupný(rok) && měsíc > 2) ++d;
    return d;
}
```



Třidu `Kalendář` spolu s jednoduchým programem, který ji používá, najdete na WWW v souboru `Kap06\03\kalen.cs`. Poznamenejme, že tento skutečný program by měl kontrolovat, zda to co zadal uživatel, je platné datum, měl by umožňovat zadání data v obvyklém formátu atd. ✧

0.4 Instance tříd

S instancemi tříd pracujeme pomocí odkazů, podobně jako s poli. Je-li `X` třída, vytvoří deklarace

```
X x1;
```

proměnnou `x1` typu „odkaz na instanci třídy `X`“. Instanci třídy `X` vytvoříme pomocí konstruktoru a operátoru `new`:

```
x1 = new X(); // Konstruktor může mít i parametry
```

Podrobněji budeme o objektech hovořit ve zbytku knihy.

Automatická správa paměti

Všechny instance objektových typů (objekty) jsou dynamické; to znamená, že je vytváříme pomocí operátoru `new` a pracujeme s nimi pomocí odkazů. Podobně jsou dynamická i pole.

Už víme, že novou instanci vytvoříme zápisem, který se skládá z klíčového slova `new` a z volání konstruktoru. Vytvořený odkaz můžeme přiřadit proměnné vhodného typu, předat metodě jako parametr atd.

O rušení instancí objektových typů a polí se programátor v C# nestará.³⁷ Prostředí .NET Framework u každého z objektů a polí sleduje, kolik na ně existuje odkazů, a pokud zjistí, že žádný, zruší ho, tj. zavolá jeho metodu `finalize()` a pak uvolní jeho paměť. Mechanismus, který se o to stará, se nazývá automatická správa paměti (garbage collector).

To je na jedné straně velice pohodlné, neboť odpadá řada chyb v práci s pamětí, naprosto běžných v jazycích C, C++, Pascal aj., které auto-

C#

³⁷ Jazyk C# neobsahuje žádný příkaz, který by umožňoval explicitně zrušit objekt nebo pole. Proto je funkce destruktorků jiná než v C++ nebo v Pascalu.

matickou správy paměti nemají. Na druhé straně to ovšem znamená větší nároky na počítač, pomalejší běh programu ap.

Jak mohou odkazy na pole nebo objekty zaniknout? Je několik možností:

- Proměnné, která představuje odkaz na objekt nebo pole, přiřadí hodnotu `null` nebo odkaz na jiný objekt.
- Proměnná, která představuje odkaz, zanikne sama. Takto mohou zaniknout lokální proměnné nebo formální parametry metod poté, co program opustí tělo metody. Může také zaniknout objekt, který proměnnou obsahuje (protože ho zrušil garbage collector).

Příklad Ve 4. kapitole jsme implementovali jednosměrně zřetězený seznam. Metoda pro vyprázdnění seznamu, tj. pro odstranění všech jeho prvků, může mít tvar

```
public void Vyprázdni()
{
    hlava = null;
}
```

Tímto přiřazením zanikne jediný odkaz na první prvek, takže zanikne i tento prvek. Tím ovšem zanikne i odkaz na druhý prvek atd.

6.5 Hodnotové typy podruhé

V úvodu jsme si řekli, že mezi hodnotové typy patří také struktury a výčtové typy. Musíme si o nich tedy alespoň krátce povědět.

Výčtové typy

Představte si, že programujeme např. počítačovou hru, v níž se vedle sebe vyskytují různé druhy ptakoještěřů, jejichž chování závisí na tom, jaký den v týdnu právě je. (V počítačových hrách je možné leccos.) To znamená, že si nejspíš jak dny v týdnu, tak druhy ptakoještěřů očíslujeme.

Ovšem tak vytvoříme v programu opět jakousi odrůdu magických čísel. Při úpravách budeme nad každou čtyřkou přemýšlet, zda představuje čtvrtek nebo pterodaktyla.

O něco lepší bude zavedení pojmenovaných konstant. Ale ani pak nebudeme mít po starostech: Obojí budou celá čísla, a tak nám nenezabrání přiřadit hodnotu dne proměnné, která má vyjadřovat létající potvora.

Tento problém řeší v C# výčtové typy, které představují skupinu pojmenovaných konstant tvořící samostatný typ. Zjednodušený popis jejich deklarace vypadá takto:

deklarace výčtového typu:

- **enum jméno { seznam_konstant }**

Jméno je identifikátor výčtového typu, a *seznam konstant*, to jsou jednotlivé výčtové konstanty, oddělené čárkami. Například

```
enum dny {po, út, st, čt, pá, so, ne}
```

Za každou z výčtových konstant může následovat rovnítko a hodnota, kterou bude představovat:

```
enum stav {ok = 0, chyba=1, průšvih=2,
           veleprůšvih=4, katastrofa=8}
```

Při použití musíme výčtové konstanty vždy kvalifikovat jménem typu – musíme psát např. `dny.po` nebo `stav.ok`.

Výčtové konstanty představují celá čísla. Ovšem hodnotu výčtového typu nelze bez explicitního přetypování přiřadit číselné proměnné naopak, číslo nelze bez explicitního přetypování přiřadit proměnné výčtového typu. Proměnné jednoho výčtového typu nelze bez explicitního přetypování přiřadit hodnotu jiného výčtového typu. Například přiřazení

```
stav s = dny.po;    // NELZE
s = 11;              // NELZE
```

překladač označí za chybná, zatímco přiřazení

```
s = (stav)12;
```

přeloží bez námitek.³⁸

Na proměnné výčtových typů nelze použít `+`, `-` a další běžné aritmetické operátory; můžeme na ně však použít operátory `++` a `--` a bitové operátory.

Struktury

Deklarace struktury se podobá deklaraci třídy, pouze místo klíčového slova `class` použijeme `struct`. Pro struktury ovšem platí řada omezení – nelze je použít jako předka jiného typu, nemohou mít destruktory aj. Povíme si o nich později, až se seznámíme s třídami; zde si pouze ukážeme jednoduchý příklad:

```

struct Bod
{
    double x, y;
    public Bod(double _x, double _y){x = _x; y = _y;}
}

```

Mohlo by se zdát, že struktury jsou zbytečné, když máme k dispozici plnohodnotné třídy. Při některých výpočtech jsou však podstatně efektivnější, tj. počítač s nimi pracuje rychleji.

6.6 Balení a vybalování

Potomci
třídy object

Všechny typy – včetně hodnotových – jsou v C# potomky třídy `object`, takže i pro ně máme k dispozici metody `ToString()`, `GetHashCode()` aj. To znamená, že např. i pro proměnnou typu `int` můžeme zavolat metodu `ToString`:

```

int i = 22;
string s = i.ToString(); // OK, to lze

```

Balení

V některých situacích bychom potřebovali s hodnotami těchto typů zacházet jako s instancemi objektových typů – například při ukládání do kontejnerů; to lze. C# k tomu používá mechanismus zvaný *zabalení a vybalení* (*boxing a unboxing*).

Použijeme-li hodnotu hodnotového typu v situaci, kdy program očekává odkaz na instanci objektového typu, vytvoří instanci objektového typu, která bude předanou hodnotu obsahovat (to je to „zabalení“).

Například v dokumentaci se dočteme, že do instancí třídy `ArrayList` můžeme ukládat odkazy na instance třídy `object` – tedy instance libovolné třídy. Po nám ale nebrání uložit tak i celá čísla:

```

System.Collections.ArrayList al =
    new System.Collections.ArrayList();

int i = 0;
while(i < 10)
{
    al.Add(i++); // Celé číslo se automaticky zabalí
}

```

Po vyjmutí z kontejneru čísla vybalíme pomocí explicitního přetypování, tedy pomocí operátoru `(int)`:

```

System.Collections.IEnumerator e = al.GetEnumerator();
while(e.MoveNext())
{

```

³⁸ Smyslem uvedených omezení pochopitelně není zneptěmňovat programátorovi život, ale pomoci mu najít nepříjemné chyby z nepozornosti – chyby, které by se jinak projevy až za běhu programu.

```

i = (int)e.Current; // Přetypování se postará o
Console.Write(" "+i); // explicitní vybalení
}

```



Jednoduchý příklad, který uloží do kontejneru 10 celých čísel, a pak je zase vyjme a vypíše, najdete na WWW v souboru `Kap06\03\box.cs`.

6.7 Delegáty

Na delegáty se můžeme dívat jako na objektové zapouzdření odkazu na metody.³⁹ Uplatňují se zejména při zpracovávání událostí, lze je ale využít i v jiných situacích – například chceme-li předat metodu jako parametr jiné metodě.

Deklarace
delegátu

Deklarace delegátu zavádí datový typ; má tvar

deklarace delegátu:

- **delegát** *typ jméno(seznam_parametrů)* ;

Konstrukce *typ jméno(seznam_parametrů)* představuje hlavičku metody, na něž budou moci instance deklarovaného typu delegátů odkazovat. Přitom *jméno* je jméno nově zaváděného typu. Např. deklarace

```
delegate bool Filtr(char s);
```

zavádí typ `Filtr`, který představuje odkaz na metodu s jedním parametrem typu `char`, která vrací hodnotu typu `bool`.

Delegát může odkazovat buď na statickou metodu nebo na nestatickou metodu spolu s objektem, pro nějž ji chceme volat. Odkaz na instanci delegátu deklarujeme stejně jako odkaz na instanci jakéhokoli jiného objektového typu,

Vytvoření
delegátu

`Filtr f;`

Nový delegát vytvoříme pomocí operátoru `new`, za nímž uvedeme jméno delegátu a v závorkách jméno metody, na níž bude delegát odkazovat

Volání delegátu

Chceme-li zavolat metodu, na níž odkazuje instance delegátu, připojíme prostě k identifikátoru instance závorky obsahující skutečné parametry.

³⁹ Ne zcela přesnou analogií delegátů jsou v C++ ukazatele na funkce nebo ukazatele na metody; delegáty jsou ovšem zabezpečené v tom smyslu, že prostředím .NET kontroluje, zda opravdu na něco ukazují.

Java ve standardní implementaci analogii delegátů neobsahuje. Objevily se rozšíření ve Visual J++ 6.0, microsoftské implementaci Javy z roku 1998, staly se jednou z příčin nejprve slovních a pak i soudních sporů mezi firmami Sun Microsystems a Microsoft.

Příklad Napíšeme třídu, jejíž metody budou sloužit jako filtry k rozlišování různých druhů znaků:

```
public class TřídaFiltrů
{
    private string
        Hač = "áčďěéíňóřšťúůýžáčďěéíňóřšťúůýžäöøríłł";

    public bool HáčkyAČárky(char c) // Filtr pro rozlišení
    {                               // znaků s diakritickými
        return (Hač.IndexOf(c) != -1); // znaménky
    }
    // A další filtry...
}
```

Dále deklarujeme typ delegátu, který bude na takovýto filtr odkazovat:

```
public delegate bool Filtr(char s);
```

V jiném místě programu – např. ve třídě Program – pak deklarujeme metodu Přeber(string s, Filtr schvaluji), která bude mít jako druhý parametr deklarovaný delegát:

```
public static string Přeber(string s, Filtr schvaluji)
{
    System.Text.StringBuilder výsledek =
        new System.Text.StringBuilder("");
    for(int i = 0; i < s.Length; i++)
        if(schvaluji(s[i])) výsledek.Append(s[i]);
    return výsledek.ToString();
}
```

Tato metoda bere jednotlivé znaky z řetězce, který dostala jako parametr, a pro každý z nich zavolá metodu, na níž odkazuje daný delegát, zápisem schvaluji(s[i]).

V metodě Main() nejprve vytvoříme instanci třídy filtrů,

```
TřídaFiltrů tf = new TřídaFiltrů();
```

pak vytvoříme instanci delegátu,

```
Filtr f = new Filtr(tf.HáčkyAČárky);
```

a nakonec tento delegát předáme metodě Přeber():

```
s = Přeber(s, f);
```

Celý program najdete na WWW v souboru Kap06\05\deleg.cs.



Vícenásobné delegáty

Delegáty, které odkazují na metody typu void, mohou obsahovat několik odkazů. Nový odkaz lze přidat pomocí operátoru +, odstranit ho lze pomocí operátoru -.

Voláním takového delegátu zavoláme všechny metody, na něž delegát odkazuje, a to v pořadí, v jakém jsme je do delegátu uložili.

Vícenásobné delegáty se používají mj. při zpracování událostí.

7 Výrazy a operátory

Výraz v C# je konstrukce, která předepisuje výpočet hodnoty (výsledek výrazu). Skládá se z *operandů* (konstant, proměnných atd.) a *operátorů*, které říkají, jak se má výsledek vypočítat. Přesný syntaktický popis výrazu v C# je značně komplikovaný a pro nás ve skutečnosti nezajímavý. Místo toho si povíme o pravidlech, která omezují používání jednotlivých operátorů.

7.1 Vlastnosti operátorů

Priorita Z matematiky jsme zvyklí, že ve výrazu $a + b \times c$ se nejprve vypočte součin $b \times c$ a teprve pak se přičte a , neboť násobení má přednost před sčítáním. (Říkáme, že násobení má *vyšší prioritu* než sčítání.) Pokud se nám toto pořadí výpočtu nelíbí, můžeme předepsat jiné pomocí závorek – např. $(a + b) \times c$ znamená, že chceme nejprve vypočítat součet $(a + b)$ a ten pak vynásobit c .

Podobná pravidla platí i v C#: Ve výrazu $a + b * c$ se také nejprve vypočte součin $b * c$ a k výsledku se pak přičte a . Chceme-li toto pořadí změnit, můžeme použít závorek: $(a + b) * c$ opět znamená, že chceme nejprve sčítat a teprve pak násobit. V C# smíme k určování pořadí výpočtu použít pouze kulaté závorky, neboť hranaté $[]$ a složené $\{\}$ jsou vyhrazeny pro jiné účely.

Operátory jsou v C# rozděleny do skupin podle priorit (viz tabulka 7.1). Ve skutečnosti ale nemá smysl si priority operátorů pamatovat; stačí vědět, že násobení a dělení má přednost před sčítáním a odečítáním, aritmetické operace před relacemi (porovnávání ap.) a všechny tyto operace mají přednost před přiřazováním. (To lze považovat za „přirozená“ pravidla, protože víceméně odpovídají běžným zvyklostem.) Pokud si nebudeme jisti, použijeme závorky; výraz tak bude zpravidla i srozumitelnější.

Asociativita Někdy může být důležité i pořadí, ve kterém se vyhodnocují operátory se stejnou prioritou. Například výraz $a + b + c$ se vyhodnotí, jako kdyby byl uzávorkován $(a + b) + c$. (Říkáme, že operátor $+$ je asociativní zleva doprava, neboť nejdříve se vyhodnotí sčítání zapsané nejvíce vlevo.) Podobně se chová většina operátorů v C#; existují ovšem i operátory, které jsou asociativní zprava doleva. Mezi ně patří

např. přiřazovací operátor $=$. Výraz $a = b = c$ se v C# vyhodnotí jako $a = (b = c)$.

7.2 Aritmetické výrazy

Aritmetické výrazy představují výpočet číselné hodnoty. Jako operandy v nich mohou vystupovat hodnoty číselných a znakových typů. Jako operátory můžeme použít unární $+$ a $-$, binární operátory $+$, $-$, $*$, $/$ a $\%$ (sčítání, odečítání, násobení, dělení, zbytek po dělení) a bitové operátory $^$, $\&$, $|$, $\ll$ a $\gg$. Pro výpočty můžeme použít také operátory inkrementace a dekrementace $++$ a $--$ a složené přiřazovací operátory $+=$, $*=$ atd. O většině z nich jsme hovořili v předchozí kapitole.

Smíšené výrazy

V aritmetických výrazech mohou vedle sebe vystupovat operandy různých typů. V takovém případě je důležité znát typ výsledku: Bude-li výsledek např. typu `long`, nedovolí nám C# přiřadit ho jen tak proměnné typu `int`, budeme ho nejdříve muset přetypovat.

Pravidla pro typ výsledku při aritmetických výpočtech lze shrnout do následujících bodů:

Unární rozšíření

Ve výrazu obsahujícím unární operátor nejprve proběhnou s operandem tzv. unární numerická rozšíření. Výsledný typ bude i typem výsledku.

1. Nejprve se operandy typu `sbyte`, `byte`, `short`, `ushort` a `char` převedou (rozšíří) na `int`.
2. V případě unárního operátoru $-$ (minus) se operand typu `uint` převede na typ `long`.

Binární rozšíření

Ve výrazu obsahujícím binární operátor se s oběma operandy provedou nejprve tzv. binární numerická rozšíření. Jejich cílem je převést oba operandy na společný typ, do jehož rozsahu se vejdu všechny hodnoty typů obou operandů. Tento typ bude i typem výsledku. Poznamenejme, že *některé kombinace typů jsou zakázány*.

S jistou dávkou zjednodušení můžeme tvrdit, že číselné typy jsou v jazyce C# uspořádány, jak ukazuje obr. 7.1.

`int` $\rightarrow$ `uint` $\rightarrow$ `long` $\rightarrow$ `ulong` $\rightarrow$ `float` $\rightarrow$ `double` $\rightarrow$ `decimal`

Obr. 7.1 Přibližné uspořádání typů podle rozsahu

Šipky směřují od typů s menším rozsahem k typům s větším rozsahem. To znamená, že bude-li např. jeden z operandů typu `double` a druhý typu, který je v našem schématu vlevo od něj, převede se i druhý operand na typ `double`.

Rozšíření
malých typů

Obsahuje-li výraz pouze operandy typů `byte`, `sbyte`, `short`, `ushort` a `char`, převedou se na typ `int`.

Zakázané
kombinace

Bude-li jeden z operandů typu `decimal`, nesmí být druhý typu `double` nebo `float`. Bude-li jeden z operandů typu `ulong`, nesmí být druhý typu `sbyte`, `short`, `int` nebo `long`.

Výjimka

Smyslem uvedených pravidel je zabránit ztrátě přesnosti při výpočtu. Proto má pravidlo, plynoucí z obr. 7.1, jednu výjimku: Je-li jeden z operandů typu `uint` a druhý typu `sbyte`, `short` nebo `int`, převedou se oba operandy na typ `long`.

Příklady

Platí-li deklarace

```
short xx = 11;
```

nedovolí nám překladač napsat

```
xx = -xx;           // Nelze
```

neboť výraz `-xx` je typu `int` a implicitní konverze typu `int` na `short` není v C# dovolena. V takovém případě musíme použít přetypování

```
xx = (short)-xx;    // To je v pořádku
```

Podobně nám překladač nedovolí napsat

```
xx = xx + 1;        // Nelze
```

Tentokrát můžeme kromě explicitního přetypování napsat také

```
xx += 1;            // OK
```

nebo použít operátor `++`. Jestliže platí deklarace

```
int x;  
uint u;
```

nedovolí nám překladač napsat

```
u = u + x;
```

neboť výsledek je typu `long`.

Složitější výrazy se vyhodnocují po jednotlivých podvýrazech. Podívejme se na příklad:

```
int i = 11, j = 22;  
long k, l = 2;  
k = i * j + l;
```

Nejprve se bude vyhodnocovat výraz `i * j`. Protože jsou oba operandy typu `int`, bude i výsledek typu `int`. Pak se k výsledku přičte `l`. Tato proměnná je typu `long`, a proto se i předchozí výsledek nejprve převede na `long`. ♦

7.3 Relační výrazy

Relační výrazy poskytují hodnoty typu `bool`. Vyjadřují relace – tedy zkoumají, zda platí vztahy „je menší“, „je rovno“, „je menší nebo rovno“ atd. Mohou obsahovat relační operátory `<`, `<=`, `>`, `>=`, `==` a `!=`.

Mezi relační operátory patří také `is`, který umožňuje testovat příslušnost instance ke třídě.

Často potřebujeme zjistit, zda nějaká hodnota – řekněme proměnná `x` – splňuje podmínku tvaru $0 < x \leq 1$. Poměrně běžnou chybou začínajících programátorů je, že ji do programu opíší doslova:

```
if(0 < x <= 1) ...           // NELZE
```

Ve skutečnosti jde o spojení dvou podmínek, $0 < x$ a $x \leq 1$, které mají platit zároveň (konjunkce). Proto musíme v programu napsat

```
if((0 < x) && (x <= 1)) ...    // OK
```

7.4 Logické výrazy

Logické výrazy mají hodnotu `true` nebo `false` typu `bool`. Mohou obsahovat proměnné a konstanty typu `bool`, volání funkcí, které vracejí hodnotu typu `bool`, a relační výrazy.

K vytváření logických výrazů můžeme používat operátory `&&` a `||`, které znamenají konjunkci, resp. disjunkci s neúplným vyhodnocením a operátor `!`, který znamená negaci. (Podrobněji jsme o nich hovořili v kapitole 6.1.) Můžeme také použít operátory `&` a `|`, které znamenají konjunkci, resp. disjunkci s úplným vyhodnocením (vždy se vypočte hodnota obou operandů). Operátor `^` vrací pro hodnoty typu `bool` stejný výsledek jako operátor `!=`.

7.5 Přehled operátorů

V tabulce 7.1 najdete přehled operátorů, které máme v C# k dispozici. Vyšší priorita je vyjádřena nižším číslem a znamená přednost při vyhodnocování. O velké většině z nich jsme již hovořili; o bitových operátorech hovořit nebudeme. Zbývá zastavit se podrobněji několika dalších operátorech.

Podmíněný výraz

Často se setkáme s výpočtem, kde výsledek závisí na splnění nějaké podmínky. Chceme např. napsat metodu `Max()`, která bude vracet větší ze dvou čísel. S pomocí příkazu `if` ji napíšeme takto:

```
int Max(int a, int b) {
    if(a > b) return a;
    else return b;
}
```

Ovšem příkaz `if` nemůžeme použít jako součást výrazu. Proto nabízí C# operátor `?:`, který používáme takto:

Syntax *použití operátoru podmíněného výrazu:*

- *podmínka ? výraz_1 : výraz_2*

Nejprve se vyhodnotí *podmínka*. Je-li splněna, je výsledkem *výraz_1*, jinak je výsledkem *výraz_2*. (Oba výrazy musí být stejného typu nebo musí být možné převést je na stejný typ.)

S použitím tohoto operátoru můžeme metodu `Max()` napsat takto:

```
int Max(int a, int b) {
    return a < b ? b : a;
}
```

Konverze a přetypování

V mnoha situacích potřebujeme hodnotu jednoho typu převést – konvertovat – na hodnotu jiného typu. Konverze, při nichž se neztrácí informace (tzv. „rozšiřující“ konverze), mohou v jazyce C# proběhnout automaticky.

Z pravidel objektového programování, o nichž jsme hovořili v první kapitole, také plyne, že a je automatická konverze odkazu na instanci odvozené třídy na odkaz na instanci báze třídy. Protože v C# mají všechny třídy společného předka, třídu `object`, znamená to, že proměnné `obj`, deklarované příkazem

```
Object obj;
```

můžeme přiřadit odkaz na jakoukoli instanci jakékoli třídy. (Pomocí odkazu `obj` můžeme ovšem volat pouze metody třídy `object`, nikoli metody odvozených tříd.)

Vedle toho jsou konverze, které jazyk C# sice umí, ale nedělá je automaticky. To mohou být „zužující“ konverze (např. typu `int` na `byte`), konverze typu `int` na `char` nebo konverze odkazu na předka na odkaz na potomka.

Syntax

Tyto konverze musíme explicitně předepsat pomocí operátoru přetypování, který má tvar (*typ*) (jméno cílového typu uzavřené v závorkách) a zapisuje se před konvertovaný výraz.

Pokud přetypování není možné, protože ho pravidla jazyka C# zakazují, ohlásí překladač chybu.

Existují ovšem situace, kdy překladač nemůže poznat, zda je přetypování možné. Máme-li např. odkaz `obj` na `typ object` a pokusíme-li se přetypovat ho na odkaz na instanci třídy `Obj`, nemůže překladač vědět, zda bude `obj` opravdu obsahovat odkaz na instanci této třídy, a proto se správnost takovýto přetypování kontroluje až za běhu programu. Pokud přetypování není možné, vznikne výjimka typu `System.InvalidCastException`.

Přiřazovací operátor

Prostý přiřazovací operátor = používáme v přiřazovacích výrazech

Syntax

použití prostého přiřazovacího operátoru:

- *proměnná = výraz*

Takovýto výraz znamená „vypočti výraz na pravé straně operátoru = a jeho hodnotu ulož do místa, které představuje *proměnná* na levé straně“. Hodnotou (výsledkem) výrazu `a = b` je to, co se uloží do proměnné `a`.

Přiřazovací příkaz představuje sám o sobě hodnotu. Z toho plyne, že přiřazení můžeme zřetězit – můžeme napsat

```
a = b = c
```



Přiřazení lze ale použít všude, kde se očekává nějaká hodnota. Ve čtvrté kapitole jsme napsali program, který analyzoval slova v textovém souboru (viz soubor `Kap04\01\Analyza.cs` na WWW). Metoda `Analyzuj()` obsahovala cyklus

```
i = PřeskočMezery(řádek, i);
while(i >= 0)
{
    // Někaké zpracování
    i = PřeskočMezery(řádek, i);
}
```

ve kterém jsme před vstupem do cyklu nastavili hodnotu `i`, pak jsme testovali, zda je podmínka opakování splněna, zpracovali data v cyklu a nakonec jsme nastavili novou hodnotu `i`. To je správné, lze to ale zapsat jednodušeji:

```
while((i = PřeskočMezery(řádek, i)) >= 0) //Znamená TOTÉŽ
{
```

```
// nějaké zpracování
}
```

Hodnotu vrácenou metodou PřeskočMezery() uložíme do i, a pokud je tato hodnota nezáporná, zpracujeme ji. Poznamenejme, že výraz i PřeskočMezery(radek, i) jsme museli uzavřít do závorek, neboť operátor = má nižší prioritu než operátor >=.

Složené přiřazení Význam složených přiřazovacích operátorů +=, -= a dalších už známe. Výraz a op= b znamená téměř přesně totéž co a = a op b. Přitom op může být některý z operátorů +, -, *, /, %, <<, >>, &, |, ^. Kromě toho, že znamenají méně psaní, mohou tyto operátory usnadnit překladač optimalizaci.

sizeof Operátor sizeof lze aplikovat pouze na jméno hodnotového typu. Vrábí velikost odpovídající proměnné v bajtech. Na rozdíl od C++ ho nelze použít na výraz.

-> Tento operátor znají programátoři používající jazyky C a C++; slouží k přístupu ke složkám struktur prostřednictvím ukazatelů.

Operátory sizeof a -> lze používat jen v nezabezpečeném kódu (*unsafe code*), a proto se s nimi v této knize nebudeme zabývat.

Asociativita Všechny binární operátory kromě přiřazovacích jsou asociativní zleva doprava. Přiřazovací operátory a operátor podmíněného výrazu ? jsou asociativní zprava doleva.

Operátor	P	Význam
. (tečka)	1	Přístup ke složce instance: x.y
[]	1	Indexování (přístup ke složkám pole): a[x]
(výraz)	1	Uzávorkování výrazu: (a + b)*c
()	1	Volání metody: f(x)
++, --	1	Postfixový inkrement a dekrement: x++
typeof	1	Určení typu: typeof(x)
sizeof	1	Určení velikosti hodnotového typu: sizeof(int)
checked, unchecked	1	Určuje, zda se má při celočíselném přetečení vyvolat výjimka: checked(a+b)
->	1	Přístup ke složce struktury prostřednictvím ukazatele

Tabulka 7.1 Přehled operátorů - první část

Operátor	P	Význam
+, -	2	Unární plus, minus (otočení znaménka): +a
!	2	Negace logické hodnoty: !A
~	2	Bitový doplněk
++, --	2	Prefixový inkrement a dekrement: ++a
(typ)	2	Přetypování: (int)x
* / %	3	Násobení, dělení, modulo (zbytek po dělení)
+ -	4	Sčítání, odečítání; také spojování řetězců
<< >>	5	Bitové posuny
< > <= >=	6	Test nerovnosti (větší, větší nebo rovno atd.)
is, as	6	Test příslušnosti k typu, přetypování
== !=	7	Test rovnosti (rovná se, resp. nerovná se)
&	8	Logický součin (konjunkce) s úplným vyhodnocením, logický součin po bitech
^	9	Nonekvivalence po bitech
	10	Logický součet (disjunkce) s úplným vyhodnocením, logický součet po bitech
&&	11	Logický součin (konjunkce) s neúplným vyhodnocením
	12	Logický součet (disjunkce) s neúplným vyhodnocením
?:	13	Podmíněný výraz
=	14	Prosté přiřazení
+= -= *= /= %= ^= = &= <<= >>=	14	Složená přiřazení: Je-li @ některý z operátorů uvedených před = v tomto výčtu, znamená výraz a @= b totéž co a = a @ b.

Tab. 7.1 Přehled operátorů – dokončení

8 Příkazy

Na příkazy se můžeme dívat jako na elementární kroky, které slouží k zápisu algoritmu v C#. Obvykle se provádějí v pořadí, ve kterém jsou v programu zapsány; některé ovšem mohou způsobit, že dojde k „přenosu řízení“, tj. že program přejde na jiný než bezprostředně následující příkaz.

Středník Součástí některých příkazů v C# je středník. To znamená, že ho v těchto příkazech prostě uvádět musíme. (To je rozdíl např. oproti Pascalu, kde středníky slouží jako oddělovače a lze je v určitých situacích vynechat.)

8.1 Blok (složený příkaz)

V některých situacích nám syntaktická pravidla jazyka C# dovolují zapsat na určité místo jen jeden příkaz. Pokud jich tam potřebujeme více, použijeme blok neboli složený příkaz. To je skupina příkazů uzavřená mezi složené závorky { a }. Z hlediska jazyka C# se chová jako jediný příkaz; syntaktický popis bloku vypadá takto:

blok:

- { příkazy_{nep} }

Index_{nep} naznačuje, že blok může být i prázdný, tj. nemusí obsahovat žádný příkaz. Pak představuje tzv. prázdný příkaz, tj. příkaz, který nedělá nic. (I taková věc se v programech občas hodí.) Za ukončující závorku bloku nepíšeme středník.

Ve třetí kapitole této knihy jsme se poznali zjednodušenou podobu příkazu cyklu `while`, v níž jsme jako tělo cyklu používali vždy blok.

Vnořené bloky

Součástí bloku může být další blok; pak hovoříme o *vnitřním* nebo *vnořeném* bloku a o bloku *obklopujícím*. Proměnné, které deklarujeme ve vnořeném bloku, nelze mimo tento blok používat. Např. úsek

```
{                // Závorka otevírající blok
    int a = 55;    // Deklarace uvnitř bloku
    f(a);
}                // Závorka ukončující blok
a++;             // Chyba - neznámá proměnná
```

označí překladač za chybný, protože proměnnou `a` nezná. (Říkáme, že `a` je lokální v předcházejícím bloku).

Ve vnořeném bloku nesmíme deklarovat proměnnou se stejným identifikátorem jako v bloku, který ho obklopuje. Proto i následující úsek programu označí překladač za chybný:

```
int x = 11;
{ double x = 2.81; } // Chyba - opakovaná deklarace x
```

8.2 Elementární příkazy

Do této skupiny shrneme prázdný příkaz, výrazový příkaz a deklaraci; všechny tyto konstrukce už vlastně známe.

Prázdný příkaz

Prázdný příkaz je příkaz, který nedělá nic. Hodí se v situacích, kdy syntaktická pravidla požadují, abychom někde zapsali příkaz, a my přitom chceme, aby tam program nic nedělal. V C# ho můžeme vyjádřit buď samotným středníkem nebo prázdným blokem.

prázdný příkaz:

- ;
- {}

I když to na první pohled vypadá podivně, používají se prázdné příkazy poměrně často. Podívejme se na příklad: Chceme najít index prvního záporného čísla v poli `A` typu `int`. Možným řešením je cyklus

```
int i = -1;
while(i < A.Length && A[++i] >= 0)
;                // Prázdný příkaz tvořící tělo cyklu
```

jehož tělo je tvořeno prázdným příkazem.⁴⁰ Procházení jednotlivých prvků pole obstará operátor `++` v podmínce, takže po skončení cyklu bude `i` obsahovat hodnotu indexu prvního záporného prvku. (Pokud pole `A` záporný prvek neobsahuje, bude `i` obsahovat `A.Length`.)

Výrazový příkaz

Připojíme-li k výrazu středník, stane se z něj příkaz. Musí ovšem jít o výraz, který má nějaký vedlejší efekt – který změní hodnotu některé proměnné, zavolá metodu ap. Například přiřazení `i = 11` představuje

⁴⁰ Je dobrým zvykem zapisovat v takovýchto případech prázdný příkaz na samostatný řádek. Program je pak přehlednější.

výraz s hodnotou 11 a s vedlejším efektem, kterým je změna hodnoty proměnné `i`. Připojíme-li za tento výraz středník, dostaneme příkaz

```
i = 11;
```

Jako výrazový příkaz lze také použít např. volání metody nebo aplikaci či operátorů `++` nebo `--`.

Deklarace

Deklarace lokální proměnné nebo lokální konstanty se v jazyce C# také považuje za příkaz. Identifikátory, zavedené takovouto deklarací, lze používat od místa deklarace do konce bloku. Podrobněji jsme o deklaracích lokálních proměnných hovořili v oddílu věnovaném bloku.

8.3 Podmíněné příkazy

Tyto příkazy vyjadřují rozhodování, které určuje větvení algoritmu. Jinak řečeno, umožňují předepsat, která část programu se má provést a to v závislosti na hodnotě určitého výrazu. Jazyk C# nám poskytuje dva takové příkazy, `if` a `switch`.

Příkaz `if`

S tímto příkazem jsme se setkali už ve 3. kapitole. Má dvě podoby, které označujeme jako *neúplné if* a *úplné if*. Syntaktický popis tohoto příkazu má tvar

příkaz `if`:

- `if(podmínka) příkaz_1` // Neúplné `if`
- `if(podmínka) příkaz_1 else příkaz_2` // Úplné `if`

Podmínka je výraz s hodnotou typu `bool`.⁴¹ Musíme ji vždy uzavřít do závorek.

Nejdříve se vypočte hodnota *podmínky*; má-li hodnotu `true` (je-li splněna), provede se *příkaz_1*. Má-li hodnotu `false` (není-li splněna), neprovede se v případě neúplného `if` nic, v případě úplného `if` se provede *příkaz_2*.

O *příkazu_1*, občas hovoříme jako o „větvi“ příkazu `if` za podmínkou, o *příkazu_2* jako o „větvi“ za `else`. Potřebujeme-li na místě *příkazu_1* nebo *příkazu_2* zapsat skupinu příkazů, použijeme blok.

⁴¹ Na rozdíl od jazyků C a C++ nelze v C# použít jako podmínku výraz s číselnou hodnotou.

Řadu příkladů použití příkazu `if` jsme viděli ve 3. a ve 4. kapitole. Přesto si ukážeme několik dalších:

```
if(x == 0)
    System.Console.WriteLine("Tichá voda břehy mele");
else
    System.Console.WriteLine("Hlasitý oheň koryto zanáší");
```

Zde jsme v obou větvích úplného `if` použili jednoduché příkazy, nikoli blok. Pokud znáte jazyk Pascal, bude vám středník před `else` připadat podivný; v C# je ale nezbytný, neboť je součástí výrazového příkazu (volání metody).

Jako *příkaz_1* v neúplném `if` můžeme použít i další příkaz `if`, třeba úplné. Tak mohou vzniknout konstrukce, které jsou na první pohled nejasné:

```
if(x>0) if(y<0) x = 11; else x = 22;
```

Kterému `if` se vztahuje klíčové slovo `else`? Pravidla jazyka C# dávají jasnou odpověď: *V takovémto případě ho překladač sdruží s nejbližším předchozím nespárovaným if*. Z toho plyne, že předchozí příklad znamená totéž co

```
if(x>0) {if(y<0) x = 11; else x = 22;}
```

Pokud bychom chtěli, aby toto `else` patřilo k prvnímu `if`, museli bychom uzavřít vnořené `if` do bloku:

```
if(x>0) {if(y<0) x = 11;} else x = 22;
```

Příkaz `switch` (přepínač)

Příkaz `if` se hodí pro rozhodování mezi dvěma možnostmi. Máme-li na vybranou z většího počtu možností, je vhodnější příkaz `switch`. Jeho formální syntaktický popis je poměrně komplikovaný a nepřehledný, proto si zde uvedeme jen zjednodušenou podobu a tu doplníme podrobným výkladem:

příkaz `switch`:

- `switch(výraz) {`
 - `case konstanta: příkazy`
 - `case konstanta: příkazy`
 - `...`
 - `default: nep_příkazy``}`

Příkaz `switch` začíná klíčovým slovem `switch`, za kterým v závorkách následuje výraz. Výraz musí být celočíselný, `char`, výčtový nebo `string` (nebo musí existovat jednoznačná implicitní konverze, která

ho umožňuje na jeden z těchto typů převést). Pak následuje *tělo* příkazu `switch`, uzavřené ve složených závorkách.

Tělo příkazu `switch` se skládá ze *sekcí* (*alternativ*). To jsou skupiny příkazů, před nimiž jsou zapsána návěští ve tvaru `case konstanta`. Před jednou sekcí může být i několik návěští. Jedna ze sekcí může také mít návěští `default`. Typ *konstant* v návěštích musí odpovídat typu *výrazu*.

V jednom návěští `case` smíme uvést jen jednu konstantu. Žádná konstanta se v návěštích v jednom příkazu `switch` nesmí opakovat. Návěští `default` nemusí být poslední, i když se jako poslední zpravidla zapisuje.

Při zpracovávání příkazu `switch` se nejprve vyhodnotí *výraz*. Pak se vyhledá návěští `case`, v němž je uvedena konstanta se stejnou hodnotou, jako má *výraz*, a program přejde na první příkaz sekce, která za ním následuje. Pokud program odpovídající návěští nenajde, přejde na první příkaz sekce označené návěštím `default` (je-li uvedeno).

Jestliže program odpovídající návěští nenajde a tělo příkazu `switch` neobsahuje návěští `default`, nestane se nic – tělo příkazu `switch` se prostě přeskočí.



Každá ze sekcí musí být ukončena příkazem, který jednoznačně určí, jak má program pokračovat. Zpravidla to je příkaz `break`, který příkaz opouští tělo příkazu `switch` a přejít na příkaz za ním; může to ale být i příkaz `return` nebo `goto`.⁴² (O těchto příkazech budeme podrobně hovořit v oddílu 8.5.)

Příklad

Napišeme program, který si vyžádá malé celé číslo, a podle jeho hodnoty vypíše nějaké přísloví. (Vzpomeňme na tetu Kateřinu z Jirotkova Saturnina...) Nejprve si vytvoříme potřebné proměnné:

```
private string[] přísloví =
{
    "Ranní ptáče dál doskáče",
    "Komu se nelení, tomu se zelení",
    "Nechval dne před večerem"
};
private string implicitníPřísloví = "Vrána k vráně sedá";
private string omluva = "No dobře, dobře...";
```

C +

⁴² Na rozdíl od Javy, C nebo C++ není v C# dovoleno „propadnout“ z jedné sekce příkazu `switch` do sekce následující. Pokud bychom o něco takového stáli, musíme použít příkaz `goto case konstanta` nebo `goto default`, i když jde o „skok“ na následující řádek programu. Smyslem tohoto opatření je zabránit chybám, které v uvedených jazycích vznikají opomenutím příkazu `break`.

Metoda, která bude přísloví vypisovat, může obsahovat příkaz `switch`:

```
public void ŘekniPřísloví(int i)
{
    switch(i)
    {
        case -1:
            Console.WriteLine(omluva);
            break;
        case 0:
        case 1:
        case 2:
            Console.WriteLine(přísloví[i]);
            break;
        default:
            Console.WriteLine(implicitníPřísloví);
            break;
    }
}
```

Zavoláme-li tuto metodu např. s parametrem 1, vypíše komu se nelení, tomu se zelení.

Příkaz `break` jsme zapsali i za příkazy za návěštím `default`, i když je tam zdánlivě zbytečný. Překladač ho tam ale striktně vyžaduje. Na druhé straně před jednou sekcí může být několik návěští; zde jsme před jednu sekci umístili návěští `case 1`, `case 2` a `case 3`.



Program obsahující třídu chrlící přísloví najdete na WWW v souboru `Kap08\01\hlody.cs`. ♦

8.4 Cykly

Příkazy, které umožňují předepsat opakování jednoho kroku algoritmu, tedy jednoho nebo několika příkazů, označujeme jako *příkazy cyklu*, *smýčky* nebo *iterační příkazy*.

Příkaz, který se má opakovat, označujeme jako *tělo cyklu*. Opakování těla cyklu zpravidla závisí na hodnotě logického výrazu, který označujeme jako *podmínku opakování*. Jednotlivé průchody tělem cyklu označujeme také jako *iterace*.

Jazyk C# nabízí čtyři příkazy cyklu: `while`, `do-while`, `for` a `foreach`. Poznamenejme, že plnění příkazů těla kteréhokoli z cyklů lze přerušit pomocí příkazů `break` nebo `continue`, o nichž budeme hovořit v následujícím oddílu.

Příkaz while

Tento příkaz vlastně už známe. Jeho syntax je

příkaz while:

- **while**(podmínka) *příkaz*

Z tohoto syntaktického popisu plyne, že tělo cyklu *while* tvoří jediný příkaz; chceme-li jich tam zapsat více, musíme použít blok. *Podmínka* je logický (booleovský) výraz; závorky kolem ní jsou nezbytné.

Při zpracování tohoto příkazu se nejprve vyhodnotí *podmínka*. Není-li splněna (má-li hodnotu *false*), tělo program cyklu přeskóčí a provádění příkazu *while* tím končí. Je-li splněna (má-li hodnotu *true*), provede se *příkaz* a znovu se vyhodnotí *podmínka*. Není-li splněna, tělo cyklu se přeskóčí a provádění příkazu *while* tím končí. Je-li splněna, provede se tělo cyklu a znovu se vyhodnotí *podmínka* atd.

Tělo příkazu *while* se nemusí provést ani jednou, neboť *podmínka* opakování se vyhodnocuje *před* průchodem tělem cyklu.

Příkaz do-while

Příkaz *do-while* se od příkazu *while* liší jen tím, že se *podmínka* opakování vyhodnocuje až po průchodu tělem cyklu, takže se tělo provede vždy alespoň jednou. Jeho syntax je

příkaz do-while:

- **do** *příkaz while*(*podmínka*);

Pro *podmínku* opakování platí totéž co pro *podmínku* u příkazu *while*. Středník za tímto příkazem je nutný. Význam tohoto příkazu je:

Nejprve se provede *příkaz*, představující tělo cyklu. Pak se vyhodnotí *podmínka*. Není-li splněna, provádění příkazu *do-while* končí a program pokračuje prvním příkazem následujícím za ním. Je-li *podmínka* splněna, provede se znovu tělo cyklu, opět se vyhodnotí *podmínka* atd.



Cyklus *do-while* skončí, jestliže *podmínka* není splněna. Tím se liší od jinak velice podobného cyklu *repeat-until* známého z jazyka Pascal. Tato podobnost je bohužel častým zdrojem chyb.

Příklad

Oblíbeným příkladem na použití cyklu *do-while* je výpočet Eulerova čísla *e*, po Ludolfovu číslu nejznámější matematické konstanty, která slouží mj. za základ přirozených logaritmů. Napíšeme program, který ho vypočte se zadanou přesností.

Švýcarský matematik Leonhard Euler, po němž je číslo *e* pojmenováno, již před více než čtvrt tisíciletím dokázal, že ho lze vyjádřit jako nekonečný součet

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \dots + \frac{1}{n!} + \dots \quad (1)$$

kde $n! = 1.2. \dots .n$ je faktoriál čísla *n*. To sice vypadá ošklivě, protože ale chceme číslo *e* spočítat jen se zadanou přesností, stačí sečíst pouze několik prvních členů. V základním kurzu matematické analýzy se dokazuje, že sečteme-li pouze prvních *k* sčítanců, bude chyba, která se tím zde dopustíme, úměrná poslednímu sčítanci, tj. $1/k!$.

Napišeme tedy metodu `double Vypočítej(double přesnost)`, která dostane jako parametr *přesnost*, s níž chceme *e* vypočítat, a vrátí hodnotu, která se od *e* nebude lišit o více než *přesnost*. Bude v cyklu sčítat členy řady (1), dokud nepřičte sčítanec menší než *přesnost*; pak skončí. Protože chceme nejprve přičíst sčítanec a pak teprve zjišťovat, zda máme skončit, je rozumné použít příkaz *do-while*.

Na první pohled by se mohlo zdát, že pro výpočet sčítanců je vhodné použít metodu `f()` pro výpočet faktoriálu, kterou jsme naprogramovali ve 3. kapitole. Ovšem zdání občas klame: Počítali bychom naprosto zbytečně stále stejné součiny. Jestliže totiž ve třetím sčítanci vypočteme $1/1.2.3$, pak ve čtvrtém budeme počítat $1/1.2.3.4$, v pátém $1/1.2.3.4.5$ atd. Jak vidíte, můžeme *n*-tý sčítanec v (1) vypočítat, když vezmeme předchozí sčítanec a vydělíme ho *n*.

```
public static double Vypočítej(double přesnost)
{
    double s = 2, d = 1; // s je součet, d je sčítanec
    int i = 2;           // Počítá, kolikátý sčítanec
    do {
        d /= i++;        // Vypočti další sčítanec
        s += d;          // a přičti ho k součtu
    } while(d > přesnost); // Je-li menší než přesnost, hotovo
    return s;
}
```

V proměnné *s* budeme počítat součet (1); proměnná *d* bude postupně obsahovat jednotlivé sčítance tohoto součtu. První dva sčítance jsou rovny 1; ty sečteme sami, takže jako počáteční hodnotu do *s* uložíme 2 a do *d* hodnotu 1. Proměnná *i* bude sloužit jako „počítadlo“, kolikátý sčítanec právě počítáme.

V cyklu nejprve vypočteme novou hodnotu *d* (při prvním průchodu cyklem to bude $1/2$) a pak zvýšíme hodnotu *i* o jedničku. Pak sčítanec přičteme k celkovému součtu. Nakonec otestujeme, zda nebyl už menší než *přesnost*, a pokud ano, skončíme.



Zavoláme-li tuto metodu s parametrem $1e-5$, vrátí 2.71828180... Necháme-li si vypsat hodnotu konstanty Math.E, dozvíme se, že správný výsledek je 2.718281828459045. Program pro výpočet čísla e najdeme na WWW v souboru Kap08\02\Euler.cs. ✧

Příkaz for

Příkaz `for` je pravděpodobně nejpoužívanějším cyklem. Jeho syntax je

příkaz *for*:

- **for**(*inicializace*_{nep} ; *podmínka*_{nep} ; *aktualizace*_{nep}) *příkaz*

kde index *nep* naznačuje, že *inicializaci*, *podmínku* i *aktualizaci* lze vynechat, i když středníky musíme uvést. Příkaz `for` znamená téměř vždy totéž co posloupnost příkazů

```
inicializace;
while(podmínka) {
    příkaz           // Tělo cyklu
    aktualizace;
}
```

Pro *podmínku* platí totéž co v případě cyklu `while`. *Inicializace* je buď výrazový příkaz nebo deklarace lokální proměnné *s* s inicializací (příp. deklarace několika lokálních proměnných stejného typu). Zpravidla definuje počáteční hodnoty proměnných, které použijeme v *podmínce* opakování cyklu. *Aktualizace* je opět výrazový příkaz, který typicky upravuje hodnoty proměnných, jež řídí opakování cyklu.

Při provádění příkazu `for` nejprve proběhne *inicializace*. Pak se zjistí, zda je splněna *podmínka* opakování. Pokud ne, provádění příkazu `for` skončí. Pokud je *podmínka* splněna, proběhne se tělo cyklu a *aktualizace* a znovu se vyhodnotí *podmínka* opakování atd.

Vynecháme-li *podmínku*, dosadí si překladač `true`. To znamená, že příkaz

```
for(;;) { /* ... */ }
```

představuje nekonečný cyklus. Jeho opakování musíme ukončit jinak – např. pomocí příkazu `break`, `return` nebo `goto` v jeho těle.

Příklad Ve 3. kapitole jsme metodu, která počítala faktoriál přirozeného čísla, napsali pomocí cyklu `while`. Pomocí cyklu `for` bude zápis o něco přehlednější:

```
public static int f(int n)
{
    if(n < 0)throw
        new Exception("Nedovolená hodnota parametru");
    int s = 1;      // Proměnná s bude obsahovat výsledek
```

```
for(int i = n; i > 0; i--)
    s = checked(i*s);
return s;
}
```

Proměnná *s* obsahuje stejně jako předtím průběžně počítaný součin.

Podívejme se, jak bude tento cyklus probíhat. Nejprve se (jako součástí inicializace) vytvoří proměnná *i*, která zde bude sloužit jako *parametr cyklu*. Tato proměnná dostane počáteční hodnotu *n*. Pak se ověří, zda je splněna podmínka opakování, tj. zda je $i > 0$. Pokud ano, proběhne tělo cyklu, tj. příkaz `s = checked(i*s)`, který způsobí, že se hodnota uložená v *s*, nahradí hodnotou $s*i$. Pak proběhne aktualizace, v našem případě příkaz `i--`, který zmenší hodnotu uloženou v *i* o 1. Dále se ověří, zda je splněna podmínka opakování atd. Úplný text tohoto příkladu najdete na WWW v souboru Kap08\03\Fakt.cs. ✧

Proměnné, deklarované v inicializační části cyklu `for`, jsou k dispozici jen v tomto příkazu (v podmínce, aktualizaci a v těle). To znamená, že nemůžeme napsat

```
public static int faktorial(int n)
{
    if(n < 0)throw new Exception("Nedovolený...");
    for(int i = n, s = 1; i > 0; i--)
        s = checked(i*s);
    return s;           // CHYBA - neznámý identifikátor s
}
```

neboť v příkazu `return` proměnná *s*, deklarovaná v inicializační části příkazu `for`, už není k dispozici – tento příkaz už není součástí těla cyklu.

Příkaz foreach

Tento příkaz se hodí, chceme-li zpracovat všechny prvky v kontejneru; může jít o instance tříd `ArrayList`, `Queue`, `List` a dalších z prostoru jmen `System.Collection`, ale třeba také znaky v řetězci nebo o prvky pole. Jeho syntax je

příkaz *foreach*:

- **foreach**(*typ identifikátor in kontejner*) *příkaz*

Konstrukce *typ identifikátor* představuje deklaraci *iterační proměnné*, tj. proměnné, která bude zpřístupňovat jednotlivé prvky kontejneru. Její *typ* musí odpovídat typu hodnot ukládaných do kontejneru. Tato proměnná je k dispozici pouze v těle příkazu `foreach`.



Kontejner je procházený kontejner; musí představovat pole nebo instanci třídy, s níž lze zacházet pomocí enumerátorů. (O nich jsme hovořili ve 4. kapitole.)



Hodnotu iterační proměnné nelze v těle cyklu `foreach` měnit.

Příklad

Ve 4. kapitole, v oddílu 4.1, jsme napsali třídu `Slovník`, která ukládala slova (tedy instance třídy `string`) do instance np třídy `Array<string>`. Metoda `Vypis()` vypisovala celý obsah slovníku pomocí cyklu `while` a enumerátoru. Řešení naložené na příkazu `foreach` je jednodušší:

```
public void Vypis()
{
    foreach(string slovo in np)
        Console.Write(slovo + " ");
}
```



Celý program s takto upravenou metodou `Vypis()` najdete na WWW v souboru `Kap08\04\analiza.cs`.

8.5 Skokové příkazy

Tyto příkazy způsobí, že program přeruší přirozenou posloupnost plnění příkazů v pořadí, v jakém jsou zapsány, a přejde na jiné místo. (Říkáme, že tyto příkazy způsobí *přenos řízení*.) Patří sem příkaz `break`, s nímž jsme se setkali v oddílech věnovaných příkazům cyklu a příkazu `switch`, příkaz `continue`, `goto` a `return`. Přenos řízení může také být způsoben vznikem výjimky (příkazem `throw` nebo `checked`). O tom budeme hovořit později v kapitole 10.

Příkaz `break`

Příkaz `break` smíme použít pouze v těle některého cyklu nebo v těle příkazu `switch`. Jeho syntax je

příkaz break:

- `break ;`

Středník je součástí tohoto příkazu.

Příkaz `break` v těle příkazu `switch` způsobí, že program ukončí provádění těla příkazu `switch` a přejde na první příkaz za ním. V těle kteréhokoli příkazu cyklu způsobí, že program ukončí provádění těla cyklu a přejde za něj.

Příklad

Příklad použití příkazu `break` v příkazu `switch` jsme viděli v oddílu 8.3; podívejme se tedy na jeho použití v cyklu. Napíšeme metodu,

kteřá najde v poli první prvek se zápornou hodnotou a vrátí jeho index. Pokud tam záporný prvek není, vrátí -1.

```
static int NajdiPrvníZáporné(int[] a)
{
    int i;
    for(i = 0; i < a.Length; i++)
        if(a[i] < 0) break;
    if(i == a.Length) return -1;
    else return i;
}
```



Program obsahující tuto metodu najdete na WWW v souboru `Kap08\05\continue.cs`.

Příkaz `continue`

Tento příkaz lze použít jen v těle cyklu; umožňuje přeskočit zbytek těla cyklu a přejít k další iteraci.

příkaz continue:

- `continue ;`

Středník nelze vynechat.

Příklad

Vezměme pole znakových řetězců. Chceme vypsát všechny řetězce, které toto pole obsahuje, s výjimkou řetězců "necenzurovaný" (pokud se v něm vyskytují). Metoda `CenzurovanýVypis()`, která se o to postará, může vypadat např. takto:

```
public void CenzurovanýVypis(string[] řeči)
{
    for(int i = 0; i < řeči.Length; i++)
    {
        if(řeči[i].Equals("necenzurovaný")) continue;
        Console.Write(řeči[i] + " ");
    }
}
```

K porovnávání jednotlivých řetězců používáme metodu `Equals()`. Najdeme-li zakázaný řetězec, přeskočíme zbytek těla cyklu a začneme zpracovávat další. Bude-li tato metoda zpracovávat pole

```
string[] text= {"Toto", "je", "necenzurovaný", "řetězec"};
```

vypiše

Toto je řetězec

Kdybychom místo příkazu `continue` použili `break`, dostali bychom jen

Toto je



Program s touto metodou najdete na WWW v souboru Kap08\08\Continue.cs. ✧

Příkaz goto

Tento příkaz patří v běžném programování k nejméně používaným. Už od šedesátých let minulého století je známo, že kvalita programu je mj. nepřímo úměrná počtu použitých příkazů goto. Nicméně jazyk C# ho obsahuje, a proto si o něm musíme povědět. (Navíc v několika výjimečných situacích může být nezbytný.) Jeho syntax je

příkaz goto:

- **goto návěští ;**
- **goto case konstanta ;**
- **goto default ;**

Návěští je identifikátor, zapsaný před příkaz a oddělený od něj dvojtečkou.

První možnost znamená přechod na příkaz označený *návěští*. Další dvě možnosti se mohou vyskytnout pouze v těle příkazu switch a znamenají přechod na počátek sekce označené návěští default nebo case *konstanta*.

Příklad Metodu CenzurovanýVýpis() bychom mohli pomocí příkazu goto napsat také takto:

```
public void CenzurovanýVýpis(string[] řeči)
{
    int i = 0;
    Znova: // Návěští
        if (i == řeči.Length) goto Konec;
        if (řeči[i].Equals("necenzurovaný")) goto Pokračuj;
        Console.WriteLine(řeči[i] + " ");
    Pokračuj: // Návěští
        i++;
        goto Znova;
    Konec: // Návěští
}
```

Jistě se mnou budete souhlasit, že tato varianta programu není právě přehledná.



Všimněte si, že za návěští musí následovat nějaký příkaz. Za návěští Konec, které označuje konec metody, jsme museli zapsat prázdný příkaz. Program s touto metodou najdete na WWW v souboru Kap08\05\goto.cs.

Příkaz return

tento příkaz znamená ukončení metody. Jeho syntax je

příkaz return:

- **return ;**
- **return výraz ;**

Středník je povinnou součástí tohoto příkazu. První varianta (b) *výrazu* se smí objevit pouze v metodě typu void. Znamená „zde končí provádění těla metody, vrať se na místo, odkud byla zavolána“. Pokud jde-li program přes složenou závorku } uzavírající tělo metody, znamená to totéž co provedení příkazu return;.

Druhá varianta se smí objevit pouze v metodě jiného typu než void. Znamená „zde končí provádění těla procedury, vrať se na místo, odkud byla zavolána, a jako výsledek tam přenes hodnotu, kterou představuje *výraz*“. Typ výrazu musí být stejný jako typ vracený metodou nebo ho musí být možné na tento typ převést rozšiřující konverzí. (Např. znamená, že v metodě s deklarovaným typem int musí být výraz typu int nebo typu byte, short nebo char. Nemůže být např. typ long.)

Příkazy throw, checked a unchecked

Příkaz throw slouží k vyvolání výjimky. Jeho syntax je

příkaz throw:

- **throw výraz_{nep};**

Výraz musí představovat instanci třídy odvozené od System.Exception a lze ho za jistých okolností vynechat. Podrobněji o něm budeme hovořit v kapitole 10.

Příkazy checked, resp. unchecked slouží k označení bloků, v nichž celočíselné přetečení má, resp. nemá vyvolat výjimku typu System.OverflowException. Hovořili jsme o nich v kapitole 6.2.

9 Třídy a objekty

Až dosud jsme se zabývali příkazy, výrazy a podobnými konstrukcemi, které najdeme ve většině programovacích jazyků, ať objektových nebo neobjektových. C# je ale čistě objektový jazyk, a proto je nejvyšší čas, abychom se podívali na třídy a objekty. S většinou pojmů, které budeme potřebovat, jsme se seznámili v kapitole 1.5. V následujících kapitolách jsme si ukázali i některá jejich použití, takže můžeme hned začít naostro.

9.1 Deklarace třídy

Zjednodušeně můžeme deklaraci třídy popsat takto:

deklarace třídy:

- *modifikátor*_{nep} **class** *identifikátor předek_a_rozhraní*_{nep}
{
 tělo třídy
}

Klíčové slovo `class` říká, že jde o deklaraci třídy, a *identifikátor* bude její jméno.

Za identifikátorem může následovat specifikace předka (bázové třídy), implementovaných rozhraní nebo obojího; v syntaktickém popisu jsme to vyjádřili zápisem *předek_a_rozhraní*. Tato specifikace nám umožňuje definovat naši třídu jako potomka jiné třídy a určit rozhraní, které naše třída implementuje. (O rozhraních budeme hovořit v kapitole 11.)

Dále následuje ve složených závorkách tělo třídy. To mohou být deklarace datových složek, metod, vlastností, událostí a typů. Mohou to také být deklarace přetížených operátorů.

Před klíčovým slovem `class` v mohou stát modifikátory `public` nebo `internal` a `abstract` nebo `sealed`.

Modifikátory v deklaraci třídy

`public` Zapišeme-li před klíčové slovo `class` modifikátor `public`, říkáme tím, že jde o veřejně přístupnou třídu. To znamená, že ji může používat

`internal`

`abstract`

`sealed`

Specifikace předka a rozhraní

vytvářet její instance, volat její metody atd. – kdokoli ve kterékoli části programu⁴³, v němž jsme ji deklarovali, a v jakémkoli jiném programu, který obsahuje odkaz na seskupení, jež tuto třídu obsahuje. Klíčové slovo `internal` znamená, že tuto třídu můžeme používat pouze v rámci aktuálního programu.

Jestliže neuvedeme ani jeden z těchto specifikačních slov, bude se třída chovat, jako kdybychom použili specifikační slovo `internal`.

Dalším z modifikátorů, které se mohou objevit před klíčovým slovem `class`, je `abstract`. Označuje abstraktní třídu, tj. třídu, od níž nelze vytvářet instance. Abstraktní třída typicky shrnuje společné vlastnosti svých potomků.

Klíčové slovo `sealed` je v jistém smyslu opakem klíčového slova `abstract`. Znamená „zapečetěnou“ třídu, tedy třídu, od níž nelze odvozovat potomky. Abstraktní třída nemůže být zapečetěná, z těchto dvou klíčových slov můžeme uvést vždy jen jedno.

Specifikace předka a implementovaných rozhraní má tvar

předek a rozhraní:

- *:jméno_bázové_třídy*
- *:jméno_bázové_třídy, seznam_rozhraní*
- *:seznam_rozhraní*

kde *jméno_bázové_třídy* je identifikátor třídy, kterou chceme použít jako předka třídy právě deklarované. Je-li to nutné, připojíme k identifikátoru bázové třídy kvalifikaci jménem prostoru jmen. Pokud *specifikaci předka* vynecháme, doplní si překladač jako předka třídu `object`, která je společným předkem všech tříd v C#.

Seznam rozhraní, to jsou jména implementovaných rozhraní oddělená čárkami.



Jazyk C# připouští pouze jednoduchou dědičnost. To znamená, že každá třída může mít pouze jednoho přímého předka. Jako předka (bázovou třídu) nemůžeme použít právě deklarovanou třídu ani žádného z potomků právě deklarované třídy. (Třída nemůže být svým vlastním předkem, a to ani nepřímým.) Pokusíme-li se o to, ohlásí překladač chybu „Circular base class definition“.

Příklad

V kapitole 1.5 jsme hovořili o grafických objektech a navrhli jsme mimo jiné třídu `Body`, která měla za úkol reprezentovat bod na obrazovce.

⁴³ Samozřejmě pokud to dovolí přístupová práva ke složkám, které chceme použít; o tom budeme hovořit dále.

Bod
souřadnice x souřadnice y barva
zjistix() zjistiy() nastavx() nastavy() zjistibarvu() nastavbarvu()

ce. Nečíslovaný obrázek po straně připomíná její ikonu v UML. Deklarace této třídy v C# by mohla vypadat takto:

```
public class Bod {
    // Tělo třídy Bod
}
```

Třída Bod je veřejně přístupná, takže ji budou smět používat i jiné programy. Není abstraktní (v její deklaraci není modifikátor `abstract`) a je přímým potomkem třídy `object`, neboť jsme neuvedli žádného předka. Neimplementuje také žádné rozhraní. ✧

9.2 Tělo třídy

Tělo třídy je v deklaraci uzavřeno mezi složené závorky. Obsahuje deklarace datových složek, metod, vlastností atd. Pro deklarace, které tvoří tělo třídy, budeme používat souhrnné označení *složky*.

Pro každou ze složek třídy, tj. pro datové složky i pro metody, musíme určit tzv. přístupová práva. Tím předepisujeme, kdo bude smět tyto složky používat.

Jazyk C# rozlišuje u složek třídy pět úrovní přístupových práv, vyjádřených pomocí čtyř klíčových slov `public`, `protected`, `private` a `internal`. Podívejme se, jak je to s přístupovými právy ke složce třídy `T` v seskupení `A` pro jednotlivé úrovně.

<code>public</code>	Použijeme-li v deklaraci složky <code>S</code> modifikátor <code>public</code> , lze ji používat všude tam, kde lze používat <code>T</code> . (Říkáme, že složka <code>S</code> je <i>veřejně přístupná</i> .)
<code>protected internal</code>	Kombinace <code>protected internal</code> znamená, že složka <code>S</code> je dostupná z <code>A</code> a ze všech tříd, které jsou potomkem třídy <code>T</code> . (Jak dále uvidíme, znamená to, že složka <code>S</code> je <code>protected</code> nebo <code>internal</code> .)
<code>protected</code>	Modifikátor <code>protected</code> označuje <i>chráněné</i> složky. Ty jsou dostupné pouze ze třídy <code>T</code> a z jejích potomků. Občas totiž potřebujeme, aby potomek mohl manipulovat se složkami, které jsou ostatním částem programu nedostupné.
<code>internal</code>	Složky s tímto modifikátorem označujeme jako <i>interní</i> . Lze je používat pouze v seskupení <code>A</code> .
<code>private</code>	Modifikátor <code>private</code> označuje <i>soukromé</i> složky, které ze používat pouze ve třídě <code>T</code> . Soukromé složky představují <i>implementační detaily</i> . Tento modifikátor lze vynechat (pro složky tříd je implicitní).

Jednotlivé stupně přístupových práv shrnuje tabulka 9.1.

Stupeň	Třída	Seskupení	Potomci	Ostatní
<code>public</code>	ano	ano	ano	ano
<code>internal protected</code>	ano	ano	ano	ne
<code>protected</code>	ano	ne	ano	ne
<code>internal</code>	ano	ano	ne	ne
<code>private</code>	ano	ne	ne	ne

Tab. 9.1 Souhrn specifikací přístupových práv pro datové složky; **ano**, resp. **ne** znamená, zda je složka s odpovídajícím modifikátorem dostupná v téže třídě, téže seskupení atd.

9.3 Datové složky

V jazyce C# používáme dvě základní skupiny datových složek – statické a nestatické. Obojí mohou být dále deklarovány jako konstantní (`const`) nebo jen ke čtení (`readonly`); o rozdílu mezi nimi budeme hovořit na následující straně.

Nestatické datové složky

To jsou „obyčejné“ datové složky, s jakými jsme se dosud převážně setkávali. Jejich deklaraci je v podstatě stejná jako deklarace lokální proměnné, kterou známe z 6. kapitoly:

deklarace nestatické datové složky:

- `přístupnep crmodnep typ seznam_deklarátorů;`

Navíc je zde specifikace přístupových práv a možnost vyjádřit konstantnost složky pomocí modifikátorů `crmod` (tj. pomocí jednoho z modifikátorů `const` nebo `readonly`). Význam *deklarátoru* je stejný jako v kap. 6.1; může také obsahovat *inicializaci*.

Každá instance bude obsahovat své vlastní nestatické datové složky. Tyto složky typicky slouží k uchovávání údajů o stavu instancí.

Inicializace

Inicializace, zapsaná v deklaraci datové složky, proběhne bezprostředně po vytvoření instance, ještě před voláním konstrukturu. Pokud ji neuvedeme, bude hodnota složky ihned po vytvoření instance záviset na jejím typu.

Neinicializované atributy

Neinicializované datové složky číselných typů budou obsahovat hodnotu 0, znakové složky budou mít hodnotu `\"u0000\"`, logické datové složky `false` a odkazy na pole či objekty hodnotu `null`.

Příklad:
třída Bod

V první kapitole jsme navrhli – a této kapitole jsme oprášili – třídu Bod. Vraťme se k ní a doplníme její deklaraci o datové složky. Souřadnice i barva jsou údaje, které budou u různých bodů různé. Popíšeme je tedy pomocí nestatických datových složek typu `int`. Tyto složky deklarujeme jako soukromé, neboť jde o „implementační detail“. Zatím jsme se rozhodli ukládat „obrazovkové“ souřadnice, které používají jako jednotku jeden pixel, tedy obrazový bod na obrazovce. Co když se ale později rozhodneme vyjadřovat souřadnice v centimetrech? Aby případná změna co nejméně zasáhla zbytek programu, je nutné způsob reprezentace bodu co nejvíce utajit před uživateli této třídy (tj. před programátory, kteří ji budou užívat – když na to přijde, i před námi samými). Podobně deklarujeme jako soukromou i složku reprezentující barvu bodu.

```
public class Bod {
    private int x, y;
    private int barva = 0x00FFFFFF;
    // K metodám se vrátíme později
}
```

Jestliže nyní vytvoříme dvě instance třídy Bod,

```
Bod a = new Bod();
Bod b = new Bod();
```

bude mít každá instance své datové složky `x`, `y` a `barva`. Protože jsme složkám `x` a `y` počáteční hodnotu nepředepsali, budou nulové; složka `barva` bude mít hodnotu `0x00FFFFFF`, tedy 16777215.

Tyto složky jsou soukromé, a proto je mohou používat pouze metody třídy Bod. Kdybychom např. v metodě `Main()` třídy `Pokus` napsali

```
int x = a.barva; // V metodě jiné třídy CHYBA - soukromá složka
```



ohlásil by překladač chybu `'Bod.barva' is inaccessible due to its protection level` – složka `Bod.barva` není vzhledem k úrovni přístupových práv dostupná. ✧

Neměnitelné složky

V deklaraci datové složky třídy můžeme použít modifikátor `const`, například

```
public const int LIMIT = 20;
```

Hodnotu takovéto složky není možné programově měnit. Ovšem výraz v inicializaci musí také být konstantní, tj. jeho hodnotu musí umět určit už překladač. Ne vždy je ovšem tento požadavek splnitelný; někdy potřebujeme složku, jejíž hodnotu sice nelze měnit, ale v době psaní programu ji neznáme.

V takovém případě použijeme modifikátor `readonly`. Složky deklarované s tímto modifikátorem můžeme inicializovat buď v deklaraci, podobně jako konstanty, nebo v konstruktoru, ale pak už je nesmíme měnit.

Příklad

Použijeme opět naši oblíbenou třídu Bod. Barvu bodu určíme v konstrukturu a pak už ji nedovolíme měnit:

```
public class Bod
{
    private int x, y;
    private readonly int barva;
    public Bod(int _barva) // V konstrukturu určíme barvu
    {
        barva = _barva;
    }
    // ... a další metody
} ✧
```

Statické datové složky

Statické datové složky nejsou součástí jednotlivých instancí, ale celé třídy. Představují data společná všem instancím; někdy se o nich mluví jako o *datových složkách třídy*.⁴⁴ Toto označení je ale těžkopádné a snadno způsobuje nedorozumění, a proto se přidržíme vžitějšího označení *statické složky*, které je dáno tím, že v jejich deklaraci používáme klíčové slovo `static`. Syntax deklarace statické datové složky je

deklarace statické datové složky:

- *přístup_{nep} static crmod_{nep} typ seznam_deklarátorů;*

Klíčové slovo `static` říká, že jde o statickou složku; význam ostatních částí je stejný jako v případě nestatických datových složek.

`readonly`

Deklarace statické datové složky také může obsahovat modifikátor `const` nebo `readonly`. Statické datové složky s modifikátorem `readonly` lze inicializovat přímo v deklaraci nebo v tzv. statickém konstrukturu, o němž budeme hovořit v následujícím oddílu o metodách.

Protože jsou statické datové složky společné pro všechny instance, existují nezávisle na tom, kolik instancí vytvoříme. Existují od spuštění programu, před vytvořením první instance, a existují do zániku poslední instance své třídy.

Použití

V metodách třídy, v nichž jsme statickou složku deklarovali, ji můžeme používat bez kvalifikace. V metodách ostatních tříd musíme jméno kvalifikovat jménem třídy. Používání statických datových

⁴⁴ Můžeme se na ně dívat jako na globální proměnné, ukryté ve třídě.

žek lze, stejně jako používání ostatních složek, omezit pomocí specifikace přístupových práv.

Příklad: e a π Ve třídě `Math` jsou definovány dvě statické datové složky `E` a `PI`, představující známé matematické konstanty:

```
public static const E = 2.718281828459045;  
public static const PI = 3.141592653589793;
```

Na tyto konstanty se odvoláváme zápisem `Math.E`, resp. `Math.PI`, a nemusíme kvůli tomu vytvářet žádnou instanci třídy `Math`. ✧

9.4 Metody

Metody představují operace, které lze provádět s instancemi nebo s třídou jako celkem. To znamená, že metoda obsahuje naprogramovaný algoritmus této operace. Volání metody pak z hlediska objektově orientovaného programování představuje posílání zprávy instanci nebo třídě.

Metody, podobně jako datové složky, mohou být statické a nestatické. Vedle toho ovšem existuje i řada dalších „odrodnů“, vyjádřených různými modifikátory, o kterých si musíme povědět.

Přetěžování Metody lze v jazyce C# přetěžovat. To znamená, že jedna třída může obsahovat několik metod se stejným identifikátorem, pokud se navzájem liší počtem, typem nebo pořadím typů parametrů. K rozlišení přetížených metod nestačí, jestliže se liší typem vrácené hodnoty.

Deklarace metody

Deklarace metody se skládá z *hlavičky* a *těla* a lze ji popsat takto:

deklarace metody:

- *hlavička metody těla metody*

Tělo metody představuje buď skupina příkazů uzavřených mezi složené závorky – tedy blok –, nebo v případě abstraktní metody samotný středník:

tělo metody:

- *blok*
- *;* // jen v případě abstraktní metody

Hlavička metody vypadá takto:

hlavička metody:

- *modifikátory_{nep} typ_{nep} identifikátor (seznam_parametrů_{nep})*

Modifikátory určují přístupová práva a dále zda jde o statickou, veřejnou, abstraktní či zabezpečenou metodu. Budeme o nich hovořit dále.

Typ představuje typ výsledku, který metoda vrací. Pokud metoda nevytváří žádnou návratovou hodnotu, použijeme klíčové slovo `void`. Typ smíme vynechat pouze u konstruktorů a destruktorů (a to vynechat dokonce musíme).

Seznam parametrů představuje seznam formálních parametrů metody.

Parametry metod

Při výkladu o metodách narážíme na dva podobné termíny: *formální parametry* a *skutečné parametry*. Formální parametry uvádíme v hlavičce metody, skutečné parametry jsou hodnoty, které zaplňují místo formálních parametrů při volání metody.

Formální parametry

Specifikace formálních parametrů se podobá deklaraci lokálních proměnných bez inicializace, může však obsahovat modifikátory `out`:

Specifikace formálního parametru:

- *modifikátor typ identifikátor*

Typ je typ parametru a *identifikátor* je jeho jméno. Formální parametry vlastně představuje proměnnou, která vznikne v těle metody v okamžiku, kdy ji zavoláme, a bude obsahovat hodnotu, kterou jí předáme jako skutečný parametr.

Předávání parametrů

Hodnoty skutečných parametrů se předávají formálním parametrům po řadě: Prvnímu formálnímu parametru se předá hodnota prvního skutečného parametru, druhému formálnímu parametru se předá hodnota druhého skutečného parametru atd.

Vezměme metodu

```
void f(int x, double y) { /* ... */ }
```

a zavolejme ji příkazem

```
f(5, 6.1);
```

V těle metody `f()` pak bude mít formální parametr `x` počáteční hodnotu 5 a formální parametr `y` hodnotu 6.1.

Pokud ve specifikaci formálního parametru neuvedeme žádný modifikátor, bude se tento parametr předávat hodnotou.

Předávání hodnotou

To znamená, že formální parametr je kopií skutečného parametru. Změny formálního parametru v těle metody nezpůsobí změny skutečného parametru.

ného parametru. V našich příkladech jsme dosud používali pouze tento způsob předávání.

Příklad

Často ale potřebujeme, aby metoda mohla měnit své skutečné parametry. Oblíbeným učebnicovým příkladem je metoda `swap()`, která má za úkol prohodit hodnoty uložené ve skutečných parametrech. Pokud bychom ji napsali tak, jako jsme psali všechny dosavadní metody,

```
public static void swap(int a, int b) // Ne funguje
{
    int c = a;
    a = b;
    b = c;
}
```

Zjistili bychom, že není k ničemu. Příkaz

```
swap(x, y);
```

nezpůsobí vůbec žádnou změnu hodnot proměnných `x` a `y`, neboť prohodí jen obsahy jejich kopií lokálních v těle metody `swap()`. ✧

Předávání odkazem

Proto nabízí jazyk C# také možnost předávat parametry odkazem; k tomu slouží modifikátor `ref`. Při předávání odkazem představuje formální parametr jen jiné jméno pro skutečný parametr, takže veškeré operace, které v těle metody provedeme s formálním parametrem, představují operace se skutečným parametrem.



Skutečným parametrem předávaným odkazem musí být *inicializovaná proměnná odpovídajícího typu* a při předávání musíme před skutečný parametr zapsat klíčové slovo `ref`.

Příklad

Napišme metodu `swap()` ještě jednou, tentokrát tak, aby fungovala – tedy s použitím parametrů předávaných odkazem:

```
public static void swap1(ref int a, ref int b) // Funguje
{
    int c = a;
    a = b;
    b = c;
}
```

Nyní příkaz

```
swap1(ref x, ref y);
```



způsobí opravdu záměnu hodnot proměnných `x` a `y`. Zdrojový text třídy s oběma metodami najdete na WWW v souboru `Kap09\01\ref.cs`. ✧

Výstupní parametry

Jazyk C# nabízí ještě jednu možnost, a to tzv. výstupní parametry. Specifikujeme je pomocí modifikátoru `out` a oceníme je v případě, že chceme, aby nějaká metoda určila počáteční hodnoty proměnných, které jí předáme jako skutečné parametry. Chovají se stejně jako parametry předávané hodnotou, až na to, že skutečné parametry nemusí být před voláním metody inicializovány; při předávání před ně píšeme klíčové slovo `out`. Deklarujeme-li metodu

```
static void init(out int i) // Výstupní parametr
{
    i = 99;
}
```

můžeme ji zavolat příkazem

```
int h; // Neinicializovanou proměnnou
init(out h); // předáme jako výstupní par.
```

Nestatické metody

Zpráva instanci

Volání nestatické metody představuje zprávu poslanou instanci. Nestatickou metodu voláme vždy pro konkrétní instanci a tato metoda může pracovat s jejími daty. Při volání kvalifikujeme jméno nestatické metody jménem instance, pro niž ji voláme. Voláme-li v těle nestatické metody další nestatickou metodu pro tutéž instanci, kvalifikaci neuvádíme. Také datové složky instance, se kterou metoda pracuje, používáme v těle této metody bez kvalifikace.

Příklad

Vraťme se ke třídě `Bod`, kterou jsme deklarovali na počátku této kapitoly, a doplníme do ní metody pro nastavení a zjištění barvy:

```
public class Bod
{
    private int x, y;
    private int barva;
    // V konstruktoru určíme barvu a souřadnice bodu
    public Bod(int _barva, int _x, int _y)
    {
        x = _x; y = _y;
        NastavBarvu(_barva);
    }

    public void NastavBarvu(int _barva)
    {
        barva = _barva;
    }

    public int ZjistiBarvu()
    {
        return barva;
    }
    // a další metody...
}
```

Vytvoříme-li dvě instance,

```
Bod a = new Bod(0x00FF0000, 1, 2);  
Bod b = new Bod(0x000000FF, 3, 4);
```

vrátí volání `a.ZjistiBarvu()` hodnotu `0x00FF0000`, neboť příkaz `return barva`; v těle této metody vezme složku `barva` z instance `a`. Podobně volání `b.ZjistiBarvu()` vrátí hodnotu `0x000000FF`, neboť vezme složku `barva` z instance `b`.

Nic nám samozřejmě nebrání pracovat v těle metody i s jinými instancemi téže nebo jiné třídy. Při volání metod pro tyto instance musíme ovšem použít kvalifikaci.

Do třídy `Bod` bychom mohli přidat např. metodu `ObarviPodleBodu()`, která přebarví aktuální instanci podle skutečného parametru:

```
public void ObarviPodleBodu(Bod x)  
{  
    barva = x.barva;  
}
```

Složku `barva` na levé straně přiřazovacího příkazu jsme zapsali bez kvalifikace, a to znamená, že se použije složka aktuální instance. Na pravé straně přiřazovacího příkazu chceme použít složku `barva` formálního parametru, proto jsme ji museli kvalifikovat jménem instance.

Poznamenejme, že vhodnější by bylo využít metod `NastavBarvu()` a `ZjistiBarvu()`; o tom jsme ale hovořili již v kapitole 3.5.

```
public void ObarviPodleBodu(Bod x) // Totéž lépe  
{  
    NastavBarvu(x.ZjistiBarvu());  
}
```



Metodu `NastavBarvu()` voláme bez kvalifikace, týká se tedy aktuální instance. Metodu `ZjistiBarvu()` voláme pro instanci `x`, a proto jsme museli použít kvalifikaci. Zdrojový text tohoto příkladu najdete na WWW v souboru `Kap09\02\Bod.cs`. ✧

Statické metody

Zpráva třídě

Volání statické metody představuje zprávu poslanou celé třídě. I když to vypadá podivně, je to často potřebné. Objektový program se, jak víme, skládá z objektů, které si posílají zprávy. Jenže když chceme např. vytvořit nový objekt, komu máme poslat odpovídající zprávu? Instanci, která ještě neexistuje? To nedává smysl. Musíme ji tedy

poslat třídě, protože ta ví, jak mají její instance vypadat.⁴⁵ Podobně chceme-li se zeptat, kolik instancí dané třídy existuje: I takový dotaz je třeba adresovat třídě jako celku, nikoli jednotlivé instanci – už proto, že podobný dotaz lze vznést i v situaci, kdy ještě (nebo už) žádná instance neexistuje.

Z předchozího povídání plyne, že statické metody nevoláme pro žádnou určitou instanci. Při volání z těla jiné metody téže třídy ji nemusíme kvalifikovat vůbec, při volání z metody jiné třídy ji musíme kvalifikovat jménem třídy. Můžeme je volat i v případě, že neexistuje žádná instance „jejich“ třídy.

Příklady

Metoda `Main()` musí být statická, neboť operační systém ji zavolá ještě než můžeme vytvořit jakoukoli instanci jakékoli třídy.

Jako další příklad nám poslouží knihovná třída `Math`, v níž jsou definovány statické metody pro výpočet běžných matematických funkcí jako jsou goniometrické funkce ap. Chceme-li tedy spočítat sinu hodnoty `x`, napíšeme `Math.Sin(x)`.

Také naše třída `VV` pro čtení dat z konzole obsahuje pouze statické metody. Proto nemusíme vytvářet žádné instance této třídy a metod můžeme volat zápisem `VV.InInt()` ap. ✧

Omezení

Protože statické metody nepracují s žádnou konkrétní instancí, nemohou volat nestatické metody a používat nestatické atributy. Přesněji: Ve statické metodě samozřejmě můžeme vytvořit instanci a s ní pracovat – pro ni můžeme volat nestatické metody atd. Nemůžeme ale např. zavolat ve statické metodě nestatickou metodu bez kvalifikace.

Ve statických metodách samozřejmě smíme používat statické datové složky a volat jiné statické metody téže třídy.

Podívejme se na jednoduchý příklad, který nám tato omezení ukáže.

```
using System;
```

```
class Test {  
    int a; // Nestatický atribut  
    static int b; // Statický atribut  
    int f() { return a + b; }  
  
    public static void Main() {  
        Console.WriteLine(b); // OK – b je statický atribut  
        Console.WriteLine(a); // CHYBA – nestatický atribut  
        f(); // CHYBA – nelze volat  
              // nestatickou metodu  
    }  
}
```

⁴⁵ To ale znamená, že i třída je objekt, tedy instance jiné třídy (tzv. metatříd). To v C# – a v mnoha dalších objektově orientovaných jazycích – opravdu platí, ale výklad na toto téma přesahuje možnosti učebnice pro zelenáče.


```

Test t = new Test(); // Vytvoříme instanci
t.f()                // OK - pro konkrétní instanci lze
                    // volat nestatickou metodu
}

```



Metoda `Main()` je statická. Proto v ní můžeme používat bez omezení statický atribut `b`, nikoli však nestatický atribut `a`. Podobně nesmíme zavolat (bez kvalifikace) nestatickou metodu `f()`. Můžeme si ovšem vytvořit instanci třídy `Test` a zavolat `f()` pro ni. Zdrojový text této třídy najdete na WWW v souboru `Kap09\03\Static.cs`. ♦

Lokální proměnné

V metodách (statických i nestatických) si můžeme definovat lokální proměnné. To jsou proměnné, které slouží pro výpočet v metodě. Vzniknou v okamžiku volání metody a zaniknou ve chvíli, kdy se program z metody vrátí. *Mezi jednotlivými voláními se jejich hodnota nezachovává.* O tom, jak se deklarují, jsme hovořili v 6. kapitole. Připomeňme si, že deklarace lokální proměnné nesmí obsahovat specifikaci přístupových práv. Nelze je také deklarovat jako statické.

Na rozdíl od statických i nestatických datových složek instancí nejsou automaticky inicializovány a po vytvoření mají náhodnou hodnotu. Překladač ovšem kontroluje, zda nepoužíváme neinicializovanou proměnnou, a pokud ano, ohlásí chybu.

Rekurze

Metody mohou volat nejen jiné metody, ale i samy sebe, přímo nebo prostřednictvím jiných metod. Tomu se v programování říká rekurze a v C# ji lze bez problémů používat.

Příklad

V učebnicích se jako jednoduchý příklad často uvádí rekurzivní výpočet $n!$, tedy faktoriálu nezáporného celého čísla n . Jak víme, jde o součin všech čísel od 1 do n . Snadno se přesvědčíme, že ho můžeme definovat také následujícím způsobem:⁴⁶

- pro $n = 0$ platí $0! = 1$,
- pro $n > 0$ platí $n! = n \times (n - 1)!$.

Uvedené vztahy můžeme přímo opsat do programu:

```

public static int f(int n)
{
    if(n == 0) return 1;
}

```

⁴⁶ V těchto dvou vztazích znamená znak $=$ rovnost, nikoli přiřazení.

```

else return n*f(n-1);
}

```

Metodu jsme deklarovali jako statickou, protože nepracuje se žádnými daty instance a tedy není důvodu, proč bychom ji měli pro nějakou instanci volat.

Jestliže předáme této metodě hodnotu 0, vrátí 1. Jinak zavolá sama sebe s parametrem $n-1$ a vrátí výsledek vynásobený n . To znamená, že při volání `f(3)` vrátí $3 \times f(2)$. To ale způsobí nové volání, `f(2)`, které vrátí $2 \times f(1)$ atd.



Zdrojový text i s jednoduchým použitím najdete na WWW v souboru `Kap09\04\Fakt.cs`. ♦

Je jasné, že řetěz postupných volání musí někde skončit, tj. že pro nějakou hodnotu musíme umět naši úlohu vyřešit bez rekurze.

Příklad s faktoriálem je trochu umělý – přece jen počítat součin pomocí rekurze není obvykle to, co nás napadne první. Existují ovšem úlohy, kdy rekurzivní formulace přirozeně vyplývá z rozkladu úlohy na jednodušší kroky, když zjistíme, že po rozdělení jsme dostali úlohu téhož typu, jen s jinými daty.

Metoda Main()

Metoda `Main()` má zvláštní postavení, neboť představuje vstupní bod programu. Musí ji obsahovat alespoň jedna třída v programu, jinak překladač ohlásí chybu. Může ji ale obsahovat i více tříd⁴⁷; pak musí me překladači pomocí parametru `/main:jmeno_třidy` naznačit, kterou metodu `Main()` má použít jako vstupní bod.

Metoda `Main()`, která má sloužit jako vstupní bod programu, musí mít jednu z hlaviček

```

public static void Main();
public static void Main(string[] s);
public static int Main();
public static int Main(string[] s);

```

Pokud tato metoda vrátí `int`, musí pochopitelně obsahovat příkaz `return`, který určí, co má vrátit. Vrácená hodnota se interpretuje jako kód ukončení programu; 0 podle konvence znamená, že je vše v pořádku, nenulová čísla představují kód chyby.

Parametry

Metoda `Main()` může mít jediný parametr, a to pole řetězců, které obsahuje parametry příkazové řádky programu.

Příklad Napíšeme program, kterému zadáme v příkazové řádce číslo v desítkové soustavě a základ soustavy, do kterého ho chceme převést. Program vypíše číslo v cílové soustavě. Přitom využijeme služeb třídy Konvertor, kterou jsme vytvořili v 6. kapitole a kterou najdete v souboru Kap06\02\Konvert.cs. My napíšeme pouze třídu Převod, její metoda Main() zpracuje parametry příkazového řádku a předá je třídě

```
using System;
public class Převod {
    public static void Main(string[] zadání) {
        if(zadání.Length != 2)
        {
            Console.WriteLine("Použití:\n"+
                "Převod celé_číslo cílová_soustava\n"+
                "cílová soustava musí být v rozmezí 2 - 36");
            return;
        }
        int číslo = Convert.ToInt32(zadání[0]);
        int základ = Convert.ToInt32(zadání[1]);
        Console.WriteLine(
            new Konvertor(základ).konvertuj(čísl));
    }
}
```

Program nejprve zkontroluje, zda dostal v příkazové řádce dva parametry. To zjistí pomocí vlastnosti Length, která je v parametru zadání, jako v každém poli, k dispozici. Pokud jich dostal více nebo méně, vypíše informaci o tom, jak je ho třeba spouštět, a skončí. (Tak se chovají všechny slušně vychované programy spouštěné z příkazové řádky.)

Pokud má program správný počet parametrů, převede je z řetězcové podoby na čísla. K tomu využije statickou metodu ToInt32() třídy System.Convertor. Nakonec vytvoří instanci třídy Konvertor a jejímu konstruktoru předá jako parametr základ cílové soustavy, zavolá její metodu konvertuj() s parametrem rovným danému číslu a výsledek vypíše.

Tento příklad zároveň ukazuje, že v jednom programu můžeme použít i třídu definovanou v jiném programu, pokud je veřejně přístupná. Program je ovšem třeba překládat příkazem

```
csc /r:K:\prog\Kap06\02\konvert.exe prevod.cs
```

v němž v parametru /r: uvedeme cestu k souboru konvert.exe obsahujícímu třídu Konvertor. (Je jasné, že použijete skutečnou cestu, nikoli cestu zde uvedenou jako příklad.) Pokud se vám nechce vypí-

⁴⁷ To se může hodit např. při ladění. Třidu odladíme pomocí testů v její vlastní metodě Main() a pak ji použijeme v rozsáhlejšímu programu.

sovat celou cestu, můžete samozřejmě okopírovat soubor konvert.exe do adresáře se zdrojovým textem tohoto příkladu.

Spustíme-li tento program příkazem

```
Prevod 18 8
```

vypíše 22, neboť to je zápis čísla 18 v osmičkové soustavě.

Zdrojový text tohoto programu najdete na WWW v souboru Kap06\05\Prevod.cs. ✧



this

Podíváme-li se pozorně na volání nestatických metod, uvědomíme si, že v těle takovéto metody musí program znát instanci, pro kterou je metoda volána; jinak by totiž nedokázal určit, kde vzít složky, se kterými metoda pracovat. Občas bychom se ovšem na tuto instanci potřebovali explicitně odvolat také my – programátoři. To jde; v těle nestatické metody představuje odkaz na aktuální instanci klíčové slovo this.

Ve statických metodách není toto klíčové slovo k dispozici.

Příklad

Vezměme naši oblíbenou třídu Bod a její metodu

```
ObarviPodleBodu(Bod x)
{
    barva = x.barva;
}
```

Jestliže tuto metodu zavoláme příkazem a.ObarviPodleBodu(b), bude v jejím těle this znamenat odkaz na instanci a. Překladač si k identifikátoru proměnné barva doplní kvalifikaci this.barva, a proto volání a.ObarviPodleBodu(b) správně změní složku barva instance a. ✧

0.5 Konstruktory a destruktory

Konstruktor je zvláštní metoda, která má na starosti vytvoření instance. Měl by se postarat o inicializaci všech datových složek, které jsou inicializovány přímo v deklaraci.

Deklarace konstruktoru má tvar

deklarace konstruktoru:

- *hlavička_konstruktoru inicializátor_{nep} tělo*

Hlavička konstruktoru se podobá hlavičce metody, avšak nesmí obsahovat jiné modifikátory než specifikaci přístupových práv. Nesmí v ní být použit např. modifikátor sealed nebo abstract. Hlavička konstruktoru nesmí také obsahovat typ vrácené hodnoty (ani void).

Konstruktor se musí jmenovat stejně jako třída, ke které patří. Chceme-li, aby třída měla více konstruktorů, musí se lišit počtem, typem nebo pořadím typů parametrů.

Inicializátor Inicializátor může mít jednu z následujících podob:

inicializátor:

- **: base(skutečné_parametry)**
- **: this(skutečné_parametry)**

První možnost slouží k vyvolání konstruktoru předka a budeme o ní hovořit v následujícím oddílu. Druhá možnost říká, že před tělem tohoto konstruktoru chceme zavolat jiný konstruktor téže třídy. (Příklad volání jiného konstruktoru téže třídy najdete v oddílu 9.7 v části nadepsané *Příklad: grafické objekty*.)

Pokud v nějaké třídě nedefinujeme žádný konstruktor, vytvoří si ho překladač sám a my ho pak můžeme používat. Takovýto konstruktor bude mít prázdné tělo a bude veřejně přístupný. Jestliže však deklarujeme alespoň jeden konstruktor, překladač už žádné další konstruktory vytvářet nebude.

Konstruktor voláme při vytváření nové instance pomocí operátoru `new`. Syntax tohoto volání je

volání konstruktoru:

- **new identifikátor_třídy(skutečné_parametry_{nep})**

Hodnotou tohoto výrazu je odkaz na nově vytvořenou instanci.

Poznamenejme, že konstruktory se nedědí.

Příklad Deklarujeme-li třídu A bez konstruktoru:

```
class A { }
```

můžeme její instanci vytvořit zápisem

```
A a = new A();
```

neboť překladač si do deklarace této třídy doplnil konstruktor s prázdným tělem

```
class A { A(){} }
```

Kdybychom však do deklarace třídy A doplnili konstruktor s parametrem,

```
class A { A(int i){} }
```

C⁺

⁴⁸ Na rozdíl od C++ můžeme v C# pro právě vytvářenou instanci zavolat v jednom konstruktoru jiný konstruktor téže třídy.

způsobilo by volání `new A()` chybu, neboť jsme v této třídě již jeden konstruktor definovali a překladač si žádný další nevytvořil, takže konstruktor bez parametrů ve třídě A chybí. ✧

Statický konstruktor

Deklarace třídy může také obsahovat jeden konstruktor s modifikátorem `static` (statický konstruktor). Ten slouží k inicializaci statických složek třídy (včetně složek s modifikátorem `readonly`), které jsme neinicializovali přímo v deklaraci. Nesmí mít žádné parametry a volá se automaticky při zavedení třídy do paměti, tedy ve chvíli, kdy je prostředí .NET Framework připravuje k jejímu prvnímu použití.

Ani statický konstruktor se nedědí.

Příklad

Ve třídě *Pokus* chceme mít statickou datovou složku představující mocnin deseti celých čísel:

```
public class Pokus {
    private static int[] Pole;
    static Pokus() // Statický konstruktor
    {
        Pole = new int[10];
        for(int i = 0; i < 10; i++) Pole[i] = i*i;
    }
    // A další metody
} ✧
```

Destruktor

Každá třída může obsahovat jeden destruktory. To je zvláštní metoda, která má za úkol připravit instanci k zániku – tedy uvolnit prostředky, které instance v průběhu své existence získala. Může např. uzavřít databázové spojení nebo otevřené soubory.

Deklarace destruktory má tvar

deklarace destruktory:

- **~jméno_třídy() { tělo }**

Jméno destruktory je tvořeno jménem třídy, před nímž je znak tildy (vlnovka neboli tilda). Destruktor nesmí mít parametry, jeho deklarace nesmí obsahovat specifikaci přístupových práv ani typu návratové hodnoty. Nelze ho také v programu volat; zavolá ho prostředí .NET Framework těsně před tím, než automatická správa paměti uvolní instanci z paměti.⁴⁹ Poznamenejme, že ani destruktory se nedědí.

C⁺

⁴⁹ Význam destruktory je v C# daleko menší než v C++. Protože nevím (a zde vůbec) bude volán, nelze mu svěřit žádné opravdu důležité operace.

9.6 Vlastnosti



Vlastnosti, anglicky označované *properties*, usnadňují používání dat uložených v instanci. Hodnoty uložené v datových složkách lze často opravdu vyjadřují vlastnosti instance: Například skutečnost, je-li okno na obrazovce je nebo není viditelné, může být vyjádřena hodnotou datové složky jeVidět typu bool. Je-li okno viditelné a přiřazení této datové složce hodnotu false, nic se nestane, jen okno v datové složce nebude odpovídat pravdě; je totiž třeba ještě zavolat metodu, která okno skryje.

Předchozí situaci se lze vyhnout dvěma způsoby:

- Tradičním řešením je, že datová složka není přístupná a programátor smí volat pouze metody, které mění její hodnotu.
- Novinkou jazyka C# je možnost definovat vlastnost (*property*). Vlastnost v programu používáme jako datovou složku, ale přiřazení hodnoty vlastnosti může znamenat provedení řady operací (v podstatě volání metody). Podobně zjištění hodnoty vlastnosti např. tím, že ji použijeme na pravé straně přiřazovacího příkazu může znamenat provedení určitých doprovodných operací.

Při tradičním přístupu tedy programátor může zavolat metodu `stavViditelnost()`, která do proměnné `jeVidět` uloží hodnotu `true` nebo `false` a přitom se postará o zobrazení nebo skrytí okna.

Definujeme-li `Viditelnost` jako vlastnost okna, postačí např. přiřazení `Viditelnost = true;`

a tím nejen změníme hodnotu proměnné `jeVidět`, ale zároveň způsobíme, že se okno opravdu zobrazí.

Vlastnosti tedy nepřinášejí nic převratně nového, pouze nabízejí způsob, jak zapisovat program přehledněji. Mohou jich také využívat nástroje pro vizuální programování, které umožňují měnit jejich hodnoty v dialogových oknech v době návrhu programu.

Deklarace vlastnosti

Zjednodušený syntaktický popis deklarace vlastnosti vypadá takto:

deklarace vlastnosti:

- *modifikátory*_{nep} *typ identifikátor* { *část_get*_{nep} *část_set*_{nep} }

Vzhledem k automatické správě paměti se také stará pouze o jiné prostředky, než je paměť. Ve skutečnosti jde o analogii metody `finalize()` z Javy – ostatně destruktor se do IL překládá jako volání metody `Finalize()`, zděděné od třídy `object`.

Modifikátory mohou specifikovat přístupová práva., Mohou se tu také objevit modifikátory, které se používají u metod v souvislosti s děděním. *Typ* je datový typ vlastnosti a *identifikátor* je jméno, pomocí něhož se na tuto vlastnost budeme odvolávat. Za identifikátorem vlastnosti následují složené závorky, v nichž je *část get* a *část set*; jednu z nich lze vynechat. Syntax těchto částí je

část get:

- `get { příkazy }`

část set:

- `set { příkazy }`

Část get slouží ke zjištění hodnoty vlastnosti, *část set* k nastavení její hodnoty. Jestliže vynecháme *část set*, dostaneme vlastnost „pouze ke čtení“; vynecháme-li *část get*, dostaneme vlastnost, jejíž hodnotu můžeme nastavit, ale nemůžeme ji zjistit. V části *set* je vždy definována proměnná `value` představující hodnotu, kterou vlastnosti přiřazujeme.

Na obě tyto *části* vlastnosti se můžeme dívat jako na těla přístupových metod; vlastnost jako celek pak představuje syntaktickou zkratku pro definici a volání těchto metod.

Třída Bod

Jako příklad vezmeme opět třídu `Bod` a doplníme do ní metodu, která ho zobrazí. To sice zatím ještě neumíme, ale pro naše účely postačí, když vypíšeme zprávu, že kreslíme bod v dané barvě:

```
public void Zobraz()
{
    System.Console.WriteLine("Kreslím bod, barva = " + barva);
}
```

Barvu bodu deklarujeme jako vlastnost. Při změně barvy je zároveň třeba překreslit daný bod na obrazovce, a proto v části *set* zavola-
metodu `Zobraz()`:

```
public int Barva // Deklarace vlastnosti
{
    set // Uloží novou barvu
    { // do soukromé datové složky
        barva = value; // a překreslí bod
        Zobraz();
    }
    get // Pouze vrátí aktuální barvu
    {
        return barva;
    }
}
```

Jsou-li `a` a `b` instance takto upravené třídy `Bod` a napíšeme-li příkaz

```
a.Barva = 0x00110000;
```

uloží se do soukromé datové složky `barva` instance `a` daná hodnota a program vypíše

```
Kreslím bod, barva = 1114112
```



Takto upravenou třídu `Bod` najdete na WWW v souboru `Kap09\Bod2.cs`.

9.7 Dědění

V první kapitole jsme se seznámili s principy OOP a tam jsme hovořili i o dědění. Pro pohodlí si ale připomeneme, že:

- Odvozená třída (potomek) by měla vždy představovat zvláštní případ (specializaci) třídy předka.
- Proto můžeme instanci potomka vždy použít na místě, kde program očekává instanci předka.
- Potomek zdědí všechny datové složky a všechny metody předka. My k nim můžeme v odvozené třídě přidat nové datové složky a nové metody. Některé zděděné metody lze znovu definovat, význam takové deklarace závisí, jak uvidíme, na modifikátorech, které v deklaraci uvedeme.

Podívejme se nyní, jak se dědění programuje v C#. Jak víme, pokud neurčíme v deklaraci třídy žádného předka, bude třída odvozena přímo od třídy `object`, která je společným předkem všech tříd v C#. Chceme-li, aby třída měla jiného předka, musíme za identifikátor v deklaraci třídy zapsat dvojtečku a jméno předka (v případě potřeby včetně prostoru jmen).

V jazyce C# můžeme používat jen jednoduchou dědičnost, tj. každá třída může mít jen jednoho bezprostředního předka zapsaného v deklaraci za dvojtečkou.

Konstruktor potomka

Jak víme, konstruktory se nedědí. To znamená, že když v předkovi deklarujeme např. konstruktor s parametrem typu `int`, neplyne z toho, že tento konstruktor můžeme automaticky používat v potomkovi. Potomek bude mít jen ty konstruktory, které v něm explicitně deklarujeme (nebo konstruktor bez parametrů vytvořený překladačem).

`base` Konstruktor potomka vždy nejprve volá konstruktor předka. Pokud nám vyhovuje, že zavolá konstruktor bez parametrů (a pokud ho pře-

dek má), je vše v pořádku. Jinak musíme konstruktor předka zavolat sami. K tomu použijeme zápis

`: base(parametry_konstruktoru_předka)`

který umístíme bezprostředně za hlavičku konstruktoru, před otevírací složenou závorkou těla. Příklad uvidíme dále.

base

Občas se stane, že znovu definovaná metoda potomka dělá téměř totéž co původní metoda v předkovi a k tomu přidává cosi navíc.

Kdybychom např. od třídy `Bod`, se kterou se v této kapitole čas od času setkáváme, odvodili třídu `ZakroužkovanýBod`, potřebovali bychom pro kreslení nejprve nakreslit bod a pak okolo něj kroužek. Bylo by samozřejmě nesmyslné programovat kreslení bodu znovu, když jsme to už jednou udělali v předkovi, ve třídě `Bod`.

`base` Jazyk C# umožňuje volat v potomkovi metodu předka. K tomu postačí, když identifikátor metody kvalifikujeme klíčovým slovem `base`,⁵⁰

Příklad

Metoda `Nakresli()` ve třídě `ZakroužkovanýBod` by tedy mohla mít tvar

```
public void Nakresli() {  
    base.Nakresli(); // Voláme metodu předka - nakreslíme  
                    // "obyčejný" bod.  
    NakresliKroužek(); // Zde nakreslíme kroužek okolo bodu.  
}
```

Pro jednoduchost předpokládáme, že existuje metoda `NakresliKroužek()`, která se postará o nakreslení kroužku okolo daného bodu. ✨

Dovolí-li to přístupová práva, můžeme pomocí klíčového slova `base` přistupovat i k datovým složkám a vlastnostem báze třídy (předka).

Předefinované metody a vlastnosti

Jak víme, odvozená třída je zpravidla specializací předka. Proto musíme občas některé operace v potomkovi implementovat odlišně, musíme předefinovat odpovídající metody nebo vlastnosti.

Jazyk C# nabízí dvě možnosti, které se liší tím, zda implementujeme polymorfismus. Podívejme se nejprve na jednodušší, i když mnohem

⁵⁰ Každý objekt odvozené třídy se skládá z objektu báze třídy a ze souhrnu přidáných při odvozování. Klíčové slovo `base` představuje odkaz na „zdejší podobjekt“ báze třídy.

méně používanou možnost, kdy předdefinovaná metoda nebo vlastnost v potomkovi neimplementuje polymorfní chování.

Nepolymorfní chování předdefinovaných metod

Chceme-li v potomkovi definovat metodu se stejnou hlavičkou (tedy se stejným jménem a stejnými parametry), musíme v její deklaraci před specifikací přístupových práv připsat modifikátor `new`.

new Tento modifikátor upozorňuje překladač, že si opravdu přejeme definovat metodu se stejnou hlavičkou jako v předkovi a nechceme přitom použít pozdní vazbu. (O pozdní vazbě jsme hovořili v kap. 1.5 v části nazvané *Polymorfismus*.) Pokud na modifikátor `new` zapomeneme, neohlásí překladač chybu, pouze varování – tedy něco jako upozornění, abychom si rozmysleli, co vlastně chceme.

Příklad Ve své knihovně máme třídu `Číslo`, jež zapouzdřuje číslo typu `int`. (Něco takového se může občas hodit.) Tato třída obsahuje mj. metodu `Faktoriál()`, která vypočte hodnotu faktoriálu v ní obsaženého čísla, a metodu `CifernýSoučet()`, jež vypočte ciferný součet tohoto čísla; to znamená, že např. pro číslo 123 vrátí hodnotu $1 + 2 + 3 = 6$. Její implementace může vypadat takto:

```
using System;

public class Číslo
{
    protected int číslo; // Zapouzdřené číslo
    public Číslo(int n) // Konstruktor
    {
        číslo = n;
    }

    public long Faktoriál() // Výpočet faktoriálu, nehlídá
    { // se přetečení
        long s = 1;
        for(int i = 2; i <= číslo; i++) s *= i;
        return s;
    }

    public long CifernýSoučet()
    {
        long s = 0;
        for(int x = Math.Abs(číslo); x != 0; x /= 10)
            s += x % 10;
        return s;
    }
    // A další metody...
}
```

Metoda `CifernýSoučet()` zjišťuje jednotlivé cifry podobně jako metoda `Konvertor.konvertuj()` v kap. 6.2 a pak je sčítá. V metodě `Fakto`

`riál()` – a v žádné další metodě – nehlídáme, zda při výpočtu nedojde k celočíselnému přetečení.

Časem ovšem zjistíme, že potřebujeme třídu, jejíž metody budou kontrolovat, zda při výpočtu nedošlo k celočíselnému přetečení. Jednou z možností (i když určitě ne nejlepší) je odvodit od třídy `Číslo` potomka, jehož metody se takto budou chovat.

Deklarujeme tedy třídu `LepšíČíslo`, jež bude potomkem třídy `Číslo`; v ní předdefinujeme metody, v nichž může nastat přetečení:

```
public class LepšíČíslo: Číslo
{
    public LepšíČíslo(int n): base(n) {}

    // Hlídá se správnost výsledku
    new public long Faktoriál()
    {
        if(číslo < 0) throw new Exception("Není definován");
        long s = 1;
        checked { for(int i = 2; i <= číslo; i++) s *= i; }
        return s;
    }
}
```

Při výpočtu ciferného součtu nemůže dojít k přetečení, proto jsme tuto metodu nemuseli měnit. V metodě `Faktoriál()` přibyla kontrola, zda nepočítáme faktoriál záporného čísla, a zda nedošlo k přetečení.

Mohlo by se zdát, že stačí v metodě `Faktoriál()` zavolat metodu předka:

```
new public long Faktoriál() // NEFUNUJE
{
    if(číslo < 0) throw new Exception("Není definován");
    checked { return base.Faktoriál(); }
}
```

Takto naprogramovaná metoda sice opravdu zavolá metodu předka, ale tím, že ji uzavřeme do bloku `checked`, bohužel nedosáhneme toho, aby se při výpočtu v ní kontrolovalo přetečení.

Nepolymorfní chování

Podívejme se nyní na možné použití této třídy. Protože `LepšíČíslo` je potomkem třídy `Číslo`, můžeme proměnné typu odkaz na `Číslo` přiřadit odkaz na instanci třídy `LepšíČíslo`:

```
LepšíČíslo n1 = new LepšíČíslo(22);
Číslo n2 = new LepšíČíslo(23); // To lze, NEBUDE POLYMORFNÍ
```

Obě instance třídy `LepšíČíslo` obsahují hodnoty, pro které dojde při výpočtu faktoriálu k přetečení. Zavolejme tedy metodu `Faktoriál()`, kterou jsme v potomkovi předdefinovali:


```
long l1 = n1.Faktoriál(); // Vyvolá výjimku
long l2 = n2.Faktoriál(); // Nevyvolá výjimku!!!
```

Přestože jak `n1`, tak `n2` obsahují odkazy na instance třídy `LepšíČíslo`, bude se ve druhém případě volat metoda předka, neboť jsme instanci použili prostřednictvím odkazu na předka, na třídu `Číslo`! ♦

Při takto implementovaných metodách se potomek chová jako předek. Takovéto chování potřebujeme jen ve výjimečných případech. Většinou vyžadujeme, aby se instance chovala polymorfně.

Doplníme, že zdrojový text příkladu z tohoto oddílu najdete na WWW v souboru `Kap09\07\Nepoly.cs`.

Podobně lze implementovat i nepolymorfní chování vlastností.

Polymorfní chování metod

Prekrytí

Předefinování nestatické metody nebo vlastnosti s použitím pozdní vazby (budeme říkat *prekrytí*) je daleko častější – obvykle totiž od objektů požadujeme polymorfní chování. Jinak řečeno, použijeme-li instanci potomka, požadujeme, aby se volaly opravdu metody či vlastnosti potomka, a to bez ohledu na typ reference, kterou jsme použili při volání. V takovém případě musíme metodu nebo vlastnost v předkovi, kde je poprvé definována, označit modifikátorem `virtual`, a v potomcích modifikátorem `override`. (Modifikátor `new` v deklaraci překryté virtuální metody nepoužijeme.)

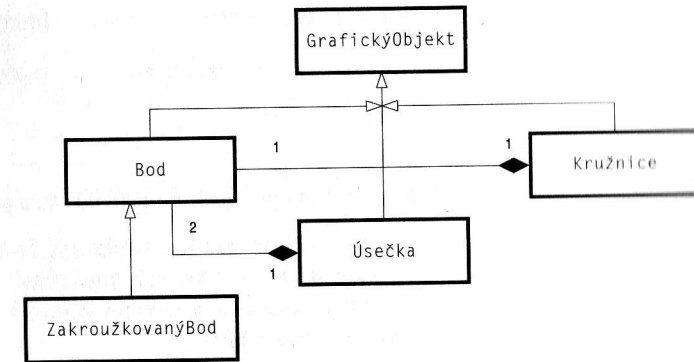
Musí mít stejnou hlavičku

Nová definice metody musí mít pochopitelně stejnou hlavičku jako metoda předka, tj. musí mít stejný identifikátor, parametry stejných typů ve stejném pořadí a musí vracet hodnotu téhož typu. Pokud toto pravidlo nedodržíme, metodu nepřekryjeme, ale přetížíme – v potomkovi budeme mít dvě různé metody vedle sebe a program se bude chovat jinak, než chceme.

Konstruktory a statické metody nemohou být virtuální (nelze je deklarovat s modifikátorem `virtual`, resp. `override`). V jejich deklaraci nelze také použít modifikátor `new`.

Příklad: grafické objekty

V kapitole 1.5 jsme navrhli hierarchii grafických objektů se společným předkem, třídou `GrafickýObjekt`. Od té doby jsme několikrát použili samostatnou třídu `Bod`; nyní se pokusíme implementovat všechny třídy z této hierarchie, do které navíc přidáme i třídu `ZakroužkovanýBod`. Diagram těchto tříd v UML ukazuje obr. 9.1 na následující stránce.



Obr. 9.1 Diagram tříd z následujícího příkladu; vyjadřuje jak dědičnost tak i skutečnost, že každá úsečka obsahuje dva body a každá kružnice jeden

Každý grafický objekt bude mít barvu vyjádřenou hodnotou výčtového typu barvy, kterou uložíme do soukromého atributu `barva`, konstruktor a metodu `Nakresli()`.⁵¹ Pracovat s ní budeme pomocí vlastnosti `Barva`.

Implementace třídy `GO` může vypadat takto (zdrojový text celého příkladu najdete na WWW v souboru `Kap09\08\go.cs`:

using System;

```
public enum barvy {červená, zelená, modrá, bílá, černá}

public class GrafickýObjekt // Obecný grafický objekt
{
    private barvy barva; // Barva objektu
    protected bool nakreslen = false; // Je nakreslen?

    public virtual barvy Barva
    {
        set // Má vlastnost barva, při změně barvy se překreslí
        {
            barva=value;
            Nakresli();
        }
        get { return barva; }
    }

    // Konstruktory: Druhý volá první
    public GrafickýObjekt(barvy _barva){Barva = _barva;}
    public GrafickýObjekt(): this(barvy.bílá){}
}
```

⁵¹ Protože stále ještě neumíme kreslit, použijeme známý trik – místo kreslení vypíšeme zprávu. Až se naučíme kreslit, prostě přepíšeme u jednotlivých metod `Nakresli()`.

```
public void Nakresli()
{
    Console.WriteLine("Kreslím GrafickýObjekt, barva "
        + barva);
}
```

Datová složka nakreslen obsahuje příznak, zda je bod již nakreslen.

Třída GrafickýObjekt má dva konstruktory. První má jeden parametr typu barvy a jeho hodnotu uloží do soukromé datové složky barva. Druhý nemá žádný parametr a nastaví bílou; k tomu využije volání jiného konstruktora téže třídy.

Vlastnost Barva při změně barvy překreslí způsobí překreslení grafického objektu – zavolá metodu Nakresli().

Všimněte si, že k nastavení barvy v konstruktoru používáme vlastnost Barva. To sice může vypadat jako zbytečné, ale ve skutečnosti to znamená, že se hodnota datové složky barva mění jedině pomocí vlastnosti Barva. V případě, že se později rozhodneme změnit způsob reprezentace barvy v grafických objektech, nám to může ušetřit práci. (Připomeňme si, že vlastnost je vlastně syntaktická zkratka pro dvojici přístupových metod.)

Vlastnost Barva nemusí být implementována stejně ve všech potomcích této třídy. Abychom ji mohli v případě potřeby v odvozených třídách předefinovat, deklarujeme ji jako virtuální.

Bod Třída Bod bude potomkem třídy GrafickýObjekt; to znamená, že zdědí datovou složku barva a vlastnost Barva. Navíc bude mít soukromé datové složky obsahující souřadnice a vlastnosti, které souřadnice zjišťují a mění. Metodu pro nakreslení bodu musíme pochopitelně překrýt, protože bod se kreslí jinak než obecný grafický objekt.

Implementace třídy Bod může být

```
public class Bod: GrafickýObjekt
{
    private int x, y;           // Souřadnice bodu
    public int X                // Souřadnice jako
    {                           // vlastnost
        set{ x = value; }
        get{ return x; }
    }
    public int Y
    {
        set{ y = value; }
        get{ return y; }
    }
}
```

```
public override void Nakresli() // "Kreslení" bodu
{
    nakreslen = true;
    Console.WriteLine("Kreslim Bod, barva " + Barva +
        ", (" + X + ", " + Y + ")");
}
public Bod(int _x, int _y, barvy _barva): base(_barva)
{
    X = _x;
    Y = _y;
}
}
```

Vlastnosti vyjadřující souřadnice a barvu není třeba komentovat, dívejme se na konstruktor. Ten má tři parametry – dvě souřadnice a barvu bodu. Barvy je třeba uložit do atributu barva; to jsme naprogramovali v konstruktoru předka, a proto ho zavoláme zápisem

base(_barva);

K uložení hodnot souřadnic do datových složek x a y použijeme vlastnosti X a Y, a to ze stejných důvodů, jako jsme použili vlastnost Barva. K uložení barvy v konstruktoru třídy GrafickýObjekt.

Zakroužkovaný Bod

Třída ZakroužkovanýBod je potomkem třídy Bod. Musíme v ní přepsat konstruktor a překrýt metodu Nakresli(); ostatní metody a atributy lze beze změny převzít od předka. Navíc v ní definujeme soukromou⁵² metodu NakresliKroužek(), která se používá k nakreslení kroužku kolem bodu. Implementace této třídy bude

```
public class ZakroužkovanýBod: Bod
{
    public ZakroužkovanýBod(int _x, int _y, barvy _barva)
    {
        base(_x, _y, _barva);
    }
    private void NakresliKroužek()
    {
        Console.WriteLine("Kreslim krouzek");
    }
    public override void Nakresli()
    {
        base.Nakresli(); // Předefinovaná metoda
        NakresliKroužek(); // Volá metodu předka
        // a něco navíc
    }
}
```



Podívejme se na konstruktor této třídy. I když má stejný parametr jako konstruktor předka, třídy Bod, musíme ho definovat, neboť konstruktory se nedědí. Kdybychom na to zapomněli, o

⁵² Je to implementační detail, který nechceme dát k dispozici samostatně.

překladač chybu. Jediné, co náš konstruktor udělá, je volání konstruktoru předka zápisem

```
base(_x, _y, _barva);
```

který se postará o vše potřebné.

Zajímavá je i metoda `Nakresli()`. Chceme-li nakreslit zakroužkovaný bod, musíme nejprve nakreslit samotný bod a pak kroužek. Kreslení bodu jsme ovšem již naprogramovali v předkovi, a proto nepotřebujeme zavolat (pomocí klíčového slova `base`) metodu předka. Pak nakreslíme kroužek.

Úsečka

Úsečka je určena svými dvěma krajními body. Ty si bude uchovávat v datových složkách `a` a `b` typu `Bod`. Kromě přístupových metod konstruktoru v ní musíme opět přepsat metodu `Nakresli()`. Implementace této třídy je:

```
public class Úsečka: GrafickýObjekt
{
    protected Bod a, b;           // Krajní body
    public Bod A                   // Vlastnosti pro práci
    {                             // s krajními body
        get{ return a;}
        set{ a = value;}
    }
    public Bod B
    {
        get{ return b;}
        set{ b = value;}
    }

    public override void Nakresli() // "Kreslení" úsečky
    {
        Console.WriteLine("Kreslim Usecku, barva " +
                           Barva + ", krajní body:");
        A.Nakresli();
        B.Nakresli();
    }

    public Úsečka(int _x1, int _y1, // Konstruktor
                  int _x2, int _y2, barvy _barva)
        :base(_barva)              // Konstruktor předka
    {
        A = new Bod(_x1, _y1, _barva); // Krajní body
        B = new Bod(_x2, _y2, _barva);
    }
}
```

Konstruktor třídy `Úsečka` zavolá konstruktor předka, třídy `GrafickýObjekt`, a svěří mu inicializaci složky `barva`. Pak vytvoří oba krajní body.



První test

Třidu `Kružnice` si můžete zkusit implementovat sami. (Její zdrojový text najdete spolu s ostatními třídami z tohoto příkladu na WWW v souboru `Kap09\08\go.cs`.)

Nyní si naše grafické objekty vyzkoušíme. Napíšeme jednoduchý program, který vytvoří postupně instance jednotlivých tříd a pro každou z nich zavolá metodu `Nakresli()`. (Proč bychom jinak vytvářeli grafické objekty, kdybychom je nechtěli kreslit?) Odkazy na vytvořené instance budeme ukládat do proměnné typu `GrafickýObjekt`. Tímto způsobem lze odkaz na `GrafickýObjekt` použít odkazující na `GrafickýObjekt` může obsahovat odkaz na jakoukoli třídu od `GrafickýObjekt` odvozenou – připomíná se si, že to je jedno ze základních pravidel OOP.

Metoda `Main()` třídy `Program` bude mít tvar

```
public static void Main()
{
    GrafickýObjekt g = new Bod(5,6,barvy.bílá);
    g.Nakresli();
    g = new Úsečka(1,2,3,4,barvy.modrá);
    g.Nakresli();
    g = new Kružnice(8, 9, 10, barvy.zelená);
    g.Nakresli();
    g = new ZakroužkovanýBod(6,9,barvy.červená);
    g.Nakresli();
}
```

Už první řádky výstupu ukazují, že se tyto objekty chovají polymorficky – tedy že se volají metody odpovídající skutečnému typu instance.

```
Kreslím Bod, barva bílá, (5, 6)
Kreslím Úsečku, barva modrá, krajní body:
Kreslím Bod, barva modrá, (1, 2)
Kreslím Bod, barva modrá, (3, 4)
...
```

Poznamenejme, že jsme zde uvedli pouze fragment výstupu: První řádek odpovídá výstupu prvního instance třídy `Bod`, další tři řádky představují výstup instance třídy `Úsečka`. (Metoda `Nakresli()` třídy volá metodu `Nakresli()` obou krajních bodů.)

Objekty
v kontejneru

Při skutečném použití (např. v grafickém editoru) budeme chtít vytvářené objekty zachovat, a proto si budeme odkazy na ně ukládat do vhodného kontejneru. V grafickém editoru ovšem nebudeme přetahovat objekty, kolik objektů uživatel našeho programu vytvoří, a proto nepotřebujeme pole, ale kontejner `System.Collections.ArrayList`. Objekty budeme ukládat pomocí metody `Add()`.

Enumerátor

Když budeme chtít nakreslit celý obrázek, tedy všechny uložené grafické objekty, budeme muset projít celý obsah kontejneru prvním

prvku. K tomu použijeme enumerátoru, nástroje, s nimiž jsme se seznámili již ve 4. kapitole.

Podívejme se, jak by mohla vypadat metoda `NakresliObrázek()`, která vezme instanci obrázek třídy `ArrayList` a nakreslí všechny grafické objekty v něm uložené:

```
public void NakresliObrázek(ArrayList obrázek)
{
    IEnumerator go = obrázek.GetEnumerator();
    while(go.MoveNext())
        ((GrafickýObjekt)go.Current).Nakresli();
}
```

Problém je, že standardní kontejnery mohou obsahovat jakékoli objekty, tedy instance jakékoli třídy odvozené od třídy `object`. Proto je vlastnost `Current` typu `object`. Abychom mohli s instancí, na ní odkazuje, zacházet jako s instancí typu `GrafickýObjekt`, musíme ho přetypovat, neboť přetypování na potomka není v C# automatické.⁵³



Na WWW v souboru `Kap09\09\Test.cs` najdete třídu `Test`, která naplní kontejner několika náhodně vytvořenými grafickými objekty a pak je nakreslí. Obsahuje mj. i uvedenou metodu `NakresliObrázek()`.

Abstraktní metody, abstraktní třídy

Už z první kapitoly víme, že `GrafickýObjekt` je typickým příkladem abstraktní třídy. Nebudeme vytvářet její instance – budeme potřebovat body, úsečky, ale nikdy obecné grafické objekty. Obecný grafický objekt nelze nakreslit: Nám se to podařilo jen proto, že jsme kreslení nahradili výpisem zprávy. Mohlo by se tedy zdát, že metoda nakresli ve třídě `GrafickýObjekt` nemá smysl. Pokud ale tuto metodu z deklarace třídy `GrafickýObjekt` odstraníme, ohlásí překladač v deklaraci potomků chybu – předek neobsahuje metodu, kterou by bylo možné překrýt. To znamená, že bychom museli v odvozených třídách změnit modifikátor `override` na `virtual`, ale pak by překladač hlásil chybu v příkazu

```
g.Nakresli(); // g je typu GrafickýObjekt
```

neboť třída `GrafickýObjekt` metodu `Nakresli()` neobsahuje. To znamená, že tuto metodu deklarovat musíme, i když nemá žádný smysl.

⁵³ Správnost tohoto přetypování nemůže kontrolovat překladač, kontroluje se až za běhu programu. Pokud bychom do kontejneru uložili objekt typu, který není potomkem třídy `GrafickýObjekt`, vznikla by při přetypování za běhu programu výjimka typu `System.InvalidCastException`.

Abstraktní metody

Takovou metodu deklarujeme jako abstraktní pomocí klíčového slova `abstract`. Abstraktní metoda nemá tělo, pouze „drží místo“ pro implementaci v odvozených třídách. Její deklarace neobsahuje tělo a končí středníkem. Podobně lze deklarovat abstraktní vlastnosti – jejich část `set`, resp. `get` obsahuje jen klíčové slovo `set`, resp. `get` a středník.

Třídou, která obsahuje alespoň jednu abstraktní metodu nebo vlastnost musíme také deklarovat jako abstraktní, to znamená, že v její deklaraci musíme před klíčovým slovem `class` uvést modifikátor `abstract`.

Příklad

Deklaraci třídy `GrafickýObjekt` upravíme následujícím způsobem:

```
public abstract class GrafickýObjekt {
    protected barvy barva;
    protected bool nakreslen = false;
    public abstract void nakresli(); // Abstraktní metoda
    // Ostatní deklarace stejné jako předtím
} ◆
```

Překladač nám nedovolí vytvářet instance abstraktních tříd.

Zapečetěné třídy, zapečetěné metody

Třídou s modifikátorem `sealed` označujeme jako „zapečetěnou“, neboť od ní nelze odvozovat potomky.

Metoda s modifikátorem `sealed` je „zapečetěná“, tj. nelze ji v odvozených třídách předefinovat. Jako zapečetěné lze deklarovat jen metody s modifikátorem `override`.

9.0 Struktury a třídy

Struktury se v C# chovají téměř jako třídy. Nejdůležitější rozdíly můžeme shrnout do následujících bodů:

- Jsou to hodnotové typy.
- Jsou automaticky zapečetěné – nemohou mít potomky ani být potomky jiné třídy.
- Nemohou mít destruktory.
- Mohou mít konstruktory, ten ale musí inicializovat všechny datové složky. Konstruktor struktury musí mít parametry.
- Datové složky struktury nelze inicializovat v deklaraci.

10 Další vlastnosti tříd

V této kapitole se podíváme na rozhraní, naučíme se přetěžovat metody a ukážeme si, jak se v jazyce C# pracuje s událostmi.

10.1 Rozhraní

Rozhraní nabízejí možnost implementovat pro třídy polymorfismus aniž bychom museli použít dědičnost. To zní možná trochu složité, proto začneme oklikou: Připomeneme si třídu *ZakroužkovanýBod*, kterou jsme navrhli v 9. kapitole a jejíž vztahy k ostatním „grafickým“ třídám ukazuje obr. 9.1 v oddílu 9.7. Odkaz na instanci této třídy vytvořenou např. příkazem

```
ZakroužkovanýBod zb =  
    new ZakroužkovanýBod(1, 55, Barvy.Cervená);
```

můžeme, jak víme, přiřadit proměnné typu odkaz na *Bod* nebo odkaz na *GrafickýObjekt*. Důvod je jasný – tyto dvě třídy jsou předky třídy *ZakroužkovanýBod*. Zatím jsme si říkali, že jde prostě o jedno ze základních pravidel OOP, a víc jsme o tom neuvažovali. Zkusme se na to nyní podívat z jiné strany: *Přiřazení*

```
Bod b = zb;
```

je správné, protože instance třídy ZakroužkovanýBod je zároveň také instancí třídy Bod. Máte-li dojem, že to už znáte, neboť odvozená třída je speciálním případem svého předka, pak jste četli pozorně. Řekli jsme si totiž, jen jinými slovy.

Instance může mít několik typů

Dědičnost tedy umožňuje mít instance, které náleží k několika vzájemně blízkým typům. Občas se ale stane, že potřebujeme, aby instance náležela k několika typům, které spolu logicky příliš nesouvisejí – a právě to umožňují rozhraní (interface).⁵⁴

Seznam metod

Rozhraní je vlastně seznam metod, vlastností a událostí. Třída může jedno nebo několik rozhraní implementovat – tím se zavazuje implementovat metody, vlastnosti a události z těchto rozhraní.

⁵⁴ Rozhraní jsou v objektovém programování poměrně novou věcí. Jedním z prvních široce používaných jazyků, které je programátorům nabídly, byla Java.

Příklad

V programu používáme třídy odvozené od společného předka *GrafickýObjekt*. Máme instanci *g* a potřebovali bychom napsat

```
zobraz(g); // Očekává instanci třídy GrafickýObjekt  
SaveToFile(g); // Očekává instanci třídy Uložitelné
```

Dvě různé metody, které chceme použít ke zpracování jedné instance očekávají parametry dvou různých tříd. Řešení založené na dědičnosti by vyžadovalo odvodit třídu *GrafickýObjekt* od třídy *Uložitelné*. To ale není nejšikovnější, protože do souborů můžeme chtít ukládat také záznamy o událostech, které v programu nastaly, záznamy o rozehraných partiích mariáše (příčemž karty kreslíme pomocí našich grafických objektů) atd. Měly by to všechno být třídy se společným předkem *Uložitelné*?

Položme si tuto otázku jinak: Co by měly společného? Samozřejmě schopnost nechat se zapsat do souboru a později přečíst, tedy metody, které nazveme třeba *Ulož()* a *Načti()*. Jenže tyto metody budou implementovány v každé třídě jinak, neboť grafický objekt se bude ukládat do souboru jinak než záznam o partii mariáše. To znamená, že tyto třídy budou mít společný pouze určitý *seznam metod* (někdy se tak říká *protokol*). Z toho plyne, že stačí, když budou implementovat stejné rozhraní. ♦

Deklarace a implementace rozhraní

Deklarace rozhraní

Deklarace rozhraní se podobá deklaraci třídy, ve které ale zapisujeme pouze hlavičky metod nebo vlastností. Jeho syntax je

deklarace rozhraní:

- *modifikátor_{nep} interface identifikátor { tělo }*

Modifikátory mají stejný význam jako v deklaraci třídy, tj. mohou specifikovat přístupová práva. *Identifikátor* je jméno deklarovaného rozhraní. *Tělo* obsahuje hlavičky metod ve tvaru

deklarace metody v těle rozhraní:

- *typ identifikátor (parametry_{nep});*

Vedle toho může obsahovat hlavičky vlastností ve tvaru

deklarace vlastnosti v těle rozhraní:

- *typ identifikátor { část_{set} část_{get} }*

Deklarace vlastnosti musí obsahovat *část set*, *část get* nebo *část set* obsahuje pouze klíčové slovo *set* následované středníkem. (*Část get* obsahuje pouze klíčové slovo *get*, opět následované středníkem. (Jde vlastně o deklaraci abstraktních metod a vlastností modifikátoru *abstract*.)



Identifikátor rozhraní obvykle začíná znakem I – např. jedno standardní rozhraní se jmenuje `IEnumerable`.

Všimněte si, že v deklaraci rozhraní nelze použít modifikátory specifikující přístupová práva ani modifikátory `virtual` nebo `override`.

Tělo rozhraní může být i prázdné. Takovéto rozhraní slouží vlastně jen k „označení“ třídy, k přidělení dalšího typu.

Implementace rozhraní

O rozhraní říkáme, že ho třída implementuje, nikoli, že ho zdědí.⁵⁵ To je sice na první pohled pouze slovní rozdíl, nicméně vyjadřuje významovou odlišnost.

Syntax pro specifikaci implementovaných rozhraní je ovšem velmi podobná jako syntax pro dědění; seznámili jsme se s ní v předchozí kapitole, a proto si jen připomeneme, že za jménem deklarované třídy zapíšeme dvojtečku a za ni pak čárkou oddělená jména implementovaných rozhraní. (Pokud zároveň deklarujeme dědění, píšeme seznam implementovaných rozhraní až za specifikaci předka.)

Jestliže nějaká třída implementuje dané rozhraní, musí deklarovat všechny jeho složky (metody, vlastnosti, události). Přitom může specifikovat přístupová práva a může deklarovat, že jde o virtuální metodu nebo vlastnost.

Implementaci rozhraní pak dědí odvozené třídy; nevirtuální složku nesmíme v odvozené třídě předdefinovat.

Použití rozhraní

Na rozhraní v C# se můžeme dívat jako na „lehkou“ náhradu vícenásobné dědičnosti, známé z C++. Třída sice nemůže mít více předků, může ale implementovat několik rozhraní. Tím dosáhneme toho, že její instance jsou několika různých typů.

Ukazuje se, že používání rozhraní patří k oblíbeným trikům programátorů, neboť dědičnost je v jistých případech příliš svazující. Podívejme se na některé příklady rozhraní z knihovny BCL.

`ICloneable`

Toto rozhraní obsahuje jedinou metodu `object Clone()`, která umožňuje klonování (vytváření kopií) objektů.⁵⁶ V její implementaci můžeme buď zavolat chráněnou metodu `MemberwiseClone()`, zděděnou od třídy `object`, nebo definovat vlastní způsob vytvoření kopie.

⁵⁵ Implementaci rozhraní ovšem třída může zdědit. To jest: Jestliže předek implementuje jisté rozhraní, implementují ho tím také všichni jeho potomci.

⁵⁶ Na rozdíl od C++ neobsahuje C# analogii kopírovacího konstrukturu.

C+

`IComparable`

Toto rozhraní obsahuje metodu `int CompareTo(object o)`, jež slouží k porovnávání dvou objektů. Výraz `a.CompareTo(b)` by měl být roven 0 při rovnosti objektů `a` a `b`, záporný při `a < b` a kladný při `a > b`.

`ICollection`

Toto rozhraní je definováno v prostoru jmen `System.Collections` a implementují ho standardní kontejnery.

`IComparer`

Také toto rozhraní je definováno v prostoru jmen `System.Collections`. Obsahuje funkci `int Compare(object x, object y)`, která slouží k porovnání dvou objektů pro účely třídění.

Některá další rozhraní poznáme v následujících příkladech.

Klonování objektů⁵⁷

Čas od času potřebujeme vytvořit nový objekt, který bude kopií objektu již existujícího. V C# neexistuje analogie kopírovacího konstrukturu z C++; třída `object`, která je společným předkem všech tříd v C#, však obsahuje metodu

`protected object MemberwiseClone()`,

jež vytvoří binární kopii instance *odvozené* třídy (tj. vyhradí místo pro novou instanci a okopíruje do ní bit po bitu data z kopírované instance). Někdy je to přesně to, co potřebujeme. Problémy ale mohou nastat při kopírování objektů, které jako složky obsahují odkazy na jiné objekty – metoda `MemberwiseClone()` třídy `object` totiž okopíruje i odkazy, nebude ale kopírovat objekty, na něž tyto složky odkazují.

Tato metoda není virtuální, a proto ji nelze v odvozených třídách přepsat. Protože je chráněná, nelze ji ani volat z jiné třídy než z `object` tomků. Tuto situaci lze řešit tím, že ve své třídě implementujeme rozhraní `ICloneable` a definujeme jeho metodu

`object Clone()`.

Ta může samozřejmě zavolat zděděnou chráněnou metodu `MemberwiseClone()` nebo definovat podle potřeby jiný způsob kopírování.

Příklad

Vezměme opět naši hierarchii grafických objektů; budeme je klonovat. To znamená, že jejich společný předek, třída `Graphics`, musí implementovat rozhraní `ICloneable` a definovat metodu `Clone()`. Abychom mohli definovat klonování v odvozených třídách,

⁵⁷ Na rozdíl od vědců, kteří se zabývají klonováním živých organizmů, programátoři nemusí obávat, že jim to politici zakážou.

⁵⁸ Říkáme, že vytvoří *mělkou* kopii. Chceme-li vytvořit *hlubokou* kopii, musíme kopírovat i objekty, na něž odkazují složky kopírované instance, a implementovat vhodným způsobem metodu `Clone()`.

jinak než ve společném předkovi, definujeme tuto metodu ve třídě `GrafickýObjekt` jako virtuální.

```
public class GrafickýObjekt: ICloneable
{
    public virtual object Clone()
    {
        return MemberwiseClone();
    }
    // Ostatní složky stejné jako předtím
}
```

V odvozených třídách `Bod` a `ZakroužkovánýBod` nemusíme metodu `Clone()` překrývat, neboť plně vyhovuje zděděná verze (stačí nám binární kopie vzoru). Ovšem ve třídách `Kružnice` a `Úsečka` budeme muset tuto metodu změnit, neboť chceme, aby klon `Kružnice` obsahoval svůj vlastní střed a aby klon `Úsečky` obsahoval své vlastní instance krajních bodů. Musíme tedy definovat metodu `Clone()` tak, aby se o to postarala (viz obr. 10.1):

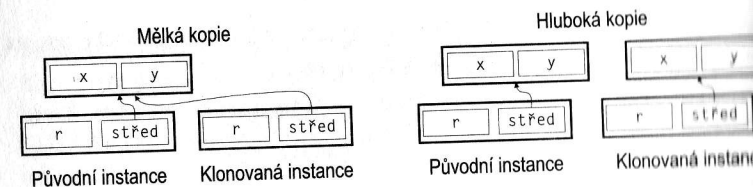
```
public class Kružnice: GrafickýObjekt
{
    protected Bod střed;
    protected int r;

    public override object Clone()
    {
        Kružnice výsledek = (Kružnice)base.Clone();
        výsledek.Střed = (Bod)Střed;
        return výsledek;
    }
    // Ostatní metody zůstanou stejné
}

public class Úsečka: GrafickýObjekt
{
    protected Bod a, b;

    public override object Clone()
    {
        Úsečka výsledek = (Úsečka)base.Clone();
        výsledek.A = (Bod)A.Clone();
        výsledek.B = (Bod)B.Clone();
        return výsledek;
    }
    // Ostatní metody zůstanou stejné
}
```

Obě tyto metody nejprve pomocí metody předka (tedy vlastně metody třídy `MemberwiseClone()` třídy `object`) vytvoří mělkou kopii klonované instance. Pak vytvoří klony středu, resp. krajních bodů, a takto upravený klon původní instance vrátí.



Obr. 10.1 Metoda `Clone()`, zděděná po třídě `object`, vytvoří mělkou kopii kružnice (vlevo); my ale potřebujeme hlubokou kopii (vpravo)

Metoda `Clone()` musí být schopna pracovat se všemi třídami v `CL` proto vrací odkaz na `object`. Vracenou hodnotu pak musíme přetypovat na skutečný typ instance.

Na WWW v souboru `Kap10\01\klon.cs` najdete zdrojové texty všech tříd z tohoto příkladu spolu s třídou `Program`, která v metodě `Main` vytvoří několik instancí tříd `Kružnice` a `Úsečka`, pak vytvoří jejich klony a v nich změní střed, resp. krajní bod. Z výstupu tohoto programu zjistíte, že se původní instance a její klon mají skutečně rozdílné složky.

Jestliže např. ve třídě `Kružnice` odstraníte metodu `Clone()`, ukážete, že změna středu klonu způsobí také změnu středu původní instance, neboť zděděná metoda používá `MemberwiseClone()`.

Seznam jako (téměř) standardní kontejner

Ve 4. kapitole jsme definovali třídu `Seznam`, jež implementovala jednosměrně zřetěžený seznam. Tento seznam jsme sice mohli považovat jako kontejner pro třídění slov, ale program se přitom musel znepokojovat, neboť náš seznam se nechoval jako standardní kontejnery – nemohli jsme s ním pracovat pomocí enumerátorů a nemohli jsme ho použít v příkazu `foreach`. To se nyní pokusíme napravit.

V dokumentaci se dočteme, že v příkazu `foreach` lze použít kontejner, který implementuje rozhraní `System.Collections.IEnumerable`. Toto rozhraní má tvar

```
IEnumerator
{
    public interface IEnumerable
    {
        IEnumerator GetEnumerator();
    }
}
```

tj. implementuje jedinou metodu, `GetEnumerator()`; tato metoda nám, vrátí instanci třídy představující enumerátor.

Enumerátor je instance třídy implementující rozhraní `System.Collections.IEnumerator`. Používá se trochu podobně jako index při procházení pole, ovšem v případě jednosměrného seznamu má značení

zenější možnosti – může se přesunout vždy jen na následující prvek nebo na začátek seznamu.

IEnumerator

Rozhraní IEnumerator má tvar

```
public interface IEnumerator
{
    bool MoveNext();           // Existuje další prvek? Pokud
                              // ano, přejdi na něj a vrať true
    object Current{get;}      // Aktuální prvek
    void Reset();             // Návrat před počátek kontejneru
}
```

Deklarujeme tedy třídu Enumerátor, která toto rozhraní implementuje. Protože jde ale o implementační detail, který nechceme navenek vystupnit, deklarujeme tuto třídu jako soukromou složku třídy Seznam.

Než se pustíme do dalšího výkladu, ukážeme si deklaraci celé třídy.

```
// Třída vnořená do třídy Seznam (definovaná uvnitř této třídy)
private class Enumerátor: IEnumerator
{
    private Seznam seznam;      // Odkaz na "svůj" seznam
    private Prvek aktuální=null; // odkaz na aktuální prvek

    internal Enumerátor(Seznam s){seznam = s;}

    public bool MoveNext()      // Zkusí přejít na další prvek
    {                           // podaří-li se to, vrátí true
        if(aktuální == null) aktuální = seznam.hlava;
        else aktuální = aktuální.VraťDalší();
        return aktuální != null;
    }

    public void Reset(){aktuální = null;} // Návrat na počátek

    public object Current      // Data z aktuálního prvku
    {
        get
        {
            if(aktuální == null) throw
                new InvalidOperationException();
            return aktuální.VraťData();
        }
    }
}
```

Třída Enumerátor obsahuje soukromou složku typu Seznam, odkazující na seznam, s nímž tento enumerátor pracuje; předáme ho jako parametr konstruktoru. Vedle toho obsahuje odkaz aktuální na „aktuální“ prvek, tedy na prvek, který zpřístupňuje vlastnost Current. Bezprostředně po vytvoření nezpřístupňuje enumerátor žádný prvek, proto složku aktuální inicializujeme hodnotou null.

MoveNext()

Metoda MoveNext() má dva úkoly: zkontrolovat, zda v seznamu existuje další prvek, a podle toho vrátit true nebo false, a pokud tento prvek existuje, změnit hodnotu složky aktuální, aby na něj odkazovala. Nejprve tedy zjistíme, zda aktuální na nějaký prvek opravdu odkazuje.

- Pokud ano, získáme z něj odkaz na jeho následovníka.
- Pokud ne, znamená to nejspíš, že jsme enumerátor ještě nepoužili. Uložíme tedy do aktuální odkaz na první prvek seznamu, který je ve složce hlava instance třídy Seznam, s níž enumerátor pracuje.

Jestliže následující prvek neexistuje, bude po předchozích krocích ve složce aktuální uložena hodnota null, jinak je tam odkaz na nový aktuální prvek. (Připomeňme si, že poslední prvek seznamu obsahuje null jako odkaz na následující prvek a že v prázdném seznamu obsahuje složka hlava hodnotu null.) Metoda MoveNext() tedy vrátí hodnotu relace aktuální != null.

Current

Vlastnost Current zpřístupňuje data z prvku seznamu, na který ukazuje složka aktuální. Je pouze ke čtení (obsahuje pouze část get) a data získá voláním metody VraťData() třídy Prvek.

Reset()

Metoda Reset() nastaví enumerátor, aby ukazoval před počátek seznamu, tak, že po následujícím volání MoveNext() bude zpřístupňovat první prvek. K tomu stačí uložit do složky aktuální hodnotu null.

Jestliže si nyní vytvoříme instanci třídy Seznam a uložíme do ní několik řetězců,

```
Seznam s = new Seznam();      // Vytvoříme seznam
s.VložNaKonec("Ahoj");        // a uložíme do něj nějaká data
s.VložNaKonec("lidi");         // .. a další vtipné texty ...
```

můžeme je vypsát příkazem

```
foreach(string ss in s) System.Console.Write(ss + " ");
```

Všimněte si, že v metodě MoveNext() jsme v příkazu

```
if(aktuální == null) aktuální = seznam.hlava;
```

použili soukromou složku hlava třídy Seznam. To lze, protože Enumerátor je třída deklarovaná uvnitř třídy Seznam.

Za zmínku stojí i skutečnost, že třída Enumerátor je soukromá, nelze ji použít mimo třídu Seznam. Uživatel s ní zachází jen prostřednictvím veřejně přístupného rozhraní IEnumerator a má k dispozici jen metody a vlastnosti tohoto rozhraní.

Program, který obsahuje upravenou třídu Seznam s enumerátorem najdete na WWW v souboru Kap10\02\seznam3.cs.



10.2 Přetěžování operátorů

Jazyk C# nabízí možnost rozšířit definici některých operátorů na nové datové typy definované uživatelem – tedy přetěžovat operátory. To vlastně o nic nového, neboť v překladačích nejrozšířenějších programovacích jazyků je tato možnost k dispozici odedávna. Zamyšlíme-li se nad výrazem `a + b`, uvědomíme si, že jde o různé operace podle toho, zda jsou `a` a `b` hodnoty typu `int`, `double` nebo třeba `string`. Překladač podle typu operandů určí, kterou operaci má použít.

Neexistuje tedy žádný rozumný důvod, proč podobnou možnost neobdobnout také programátorovi. Navíc používání přetížených operátorů může výrazně zpřehlednit program.⁵⁹ (To ale neznamená, že přetěžování operátorů nemá své odpůrce.)

Základní pravidla

Začneme tím, že si řekneme několik základních pravidel, jež platí pro přetěžování operátorů v jazyce C#.

- Přetížením můžeme v C# rozšířit použití některého z operátorů na nové datové typy. Nemůžeme zavést nový operátor – nelze například vzít symbol `**` a prohlásit ho za operátor.
- Jazyk C# dovoluje přetěžovat pouze operátory uvedené v tabulce 10.1. Ostatní operátory uvedené v tabulce 7.1 přetěžovat nelze.

Unární operátory	<code>+ - ! ~ ++ --</code>
binární operátory	<code>+ - * / % & ^ << >> != == < > <= >=</code>
logické konstanty	<code>true false</code>
indexování	<code>[]</code>
konverzní funkce	<code>(typ)</code>

Tab. 10.1 Přetěžovatelné operátory v C#

- Při přetěžování nelze změnit prioritu ani asociativitu operátoru. Nelze také změnit počet jeho operandů.

⁵⁹ Všeho s mírou: Při nevhodném použití může naopak vést k téměř nečitelnému, a tedy špatně opravovatelnému a udržovatelnému programu. Nic nám totiž nebrání definovat například operátor `*` tak, že bude sčítat, a operátor `/` tak, že bude násobit... Právě možnost takovýchto a podobných „zneužití“ je častým argumentem odpůrců přetěžování operátorů. Jenže – zřekneme se nože jen proto, že se s ním někdo méně šikovný může také pořídit?

Deklarace přetíženého operátoru

Unární nebo binární operátor nebo konverzní funkci (viz dále) deklarujeme jako *veřejně přístupnou statickou metodu* třídy nebo struktury; jméno takovéto metody je tvořeno klíčovým slovem `operator`, za nímž následuje symbol operátoru, například `operator +`.

Parametry unárního nebo binárního operátoru nebo konverzní funkce musíme předávat hodnotu.

Přetížené binární operátory deklarujeme jako metody se dvěma parametry, z nichž první odpovídá levému a druhý pravému operandu. Přetížené unární operátory deklarujeme jako metody s jedním parametrem.

Logické konstanty `true` a `false` se přetěžují jako unární operátory; povídání o nich však přesahuje rámec naší knihy. O operátorech `[]` a `(typ)` si povíme zvlášť.

Nepřímo přetěžované operátory

Operátory op=

Jazyk C# neumožňuje přetěžovat složené přiřazovací operátory. Ovšem překladač odvozuje význam těchto operátorů z významu operátorů `+`, `-` atd., takže například přetížením operátoru `+` automaticky definujeme význam operátoru `+=`.

|| a &&

Také operátory `||` a `&&` není třeba přetěžovat. Překladač C# totiž odvozuje význam operátoru `||` (logické disjunkce s neúplným vyhodnocením) z významu logického operátoru `|` (logické disjunkce s úplným vyhodnocením) a význam operátoru `&&` (logické konjunkce s neúplným vyhodnocením) z významu logického operátoru `&` (logické konjunkce s úplným vyhodnocením).

Některé operátory tvoří dvojice



Některé operátory tvoří při přetěžování logické dvojice, v nichž platí, že pokud přetížíme jeden z těchto operátorů, musíme přetížit i druhý. Jedná se o páry `<a>`, `<= a>`, `= a`, `!=`, `true` a `false`.

Komplexní čísla

Přetížené operátory používáme nejčastěji v programech, které opravdově počítají. V některých složitějších výpočtech potřebujeme komplexní čísla, a proto si ukážeme, jak je lze v C# definovat. Při výpočtech ovšem zpravidla záleží na rychlosti, a proto je deklarujeme jako strukturu, tedy jako hodnotový typ.

Připomeňme si, že komplexní číslo je vlastně dvojice reálných čísel, které označujeme jako reálnou a imaginární část. Součet dvou komplexních čísel je komplexní číslo, jehož reálná část je součtem reálných částí a imaginární část je součtem imaginárních částí sčítance. Podobně rozdíl dvou komplexních čísel je komplexní číslo tvořený rozdílem reálných a rozdílem imaginárních částí.

Číslo i Komplexní číslo $(0, 1)$ se označuje symbolem i a platí, že $i^2 = -1$.

Deklarace struktury `cplx` může tedy vypadat např. takto:

```
public struct cplx
{
    double re, im;

    public cplx(double _re, double _im){re = _re; im = _im;}
    public cplx(double _re):this(_re, 0){}

    public double Re
    {
        set {re = value;}
        get{return re;}
    }

    public double Im
    {
        set {im = value;}
        get{return im;}
    }

    public double Abs
    {
        get{ return Math.Sqrt(Re*Re+Im*Im); }
    }
    // zde budou deklarace operátorů
}
```

Vedle soukromých složek `re` a `im`, které obsahují reálnou a imaginární část, jsme deklarovali veřejně přístupné vlastnosti `Re` a `Im`, které hodnoty těchto složek zpřístupňují.⁶⁰ Kromě toho jsme definovali vlastnost `Abs`, která představuje absolutní hodnotu komplexního čísla, tj. vzdálenost od bodu $(0, 0)$ v rovině komplexních čísel. Je jen ke čtení,

⁶⁰ Možná vám připadá, že to je zbytečné, že by stačilo zpřístupnit přímo složky `re` a `im`. Uvažte ale, že komplexní číslo lze reprezentovat několika způsoby. My jsme zvolili způsob založený na dvojici čísel představujících souřadnice v rovině. Jiná používaná reprezentace je založena na dvojici čísel, z nichž jedno znamená úhel sevřený spojnicí komplexního čísla v komplexní rovině a kladnou poloosou reálných čísel a druhé vzdálenost od bodu $(0,0)$. V případě, že se později rozhodneme změnit reprezentaci komplexních čísel (např. kvůli efektivitě), ušetří nám přístup založený na vlastnostech spoustu starostí, neboť nebudeme muset měnit už hotové části programu, které komplexní čísla používají.

nelze ji měnit. (Absolutní hodnota komplexního čísla $a = (x, y)$ je definována vztahem $|a| = \sqrt{x^2+y^2}$.)

Dále jsme zde definovali dva konstruktory; druhý z nich má jen jeden parametr, za imaginární část si doplní 0. (C# bohužel nedovoluje ve struktuře deklarovat konstruktor bez parametrů.) Nyní definujeme pro komplexní čísla alespoň některé základní matematické operace.

Překladač také z nějakých záhadných důvodů odmítá inicializaci složek struktury pomocí vlastností. Konstruktor

```
public cplx(double _re, double _im){Re = _re; Im = _im;}
```

označí za chybný se zdůvodněním, že „ve struktuře musí být explicitně inicializovány všechny složky“; inicializace pomocí vlastností zřejmě není dostatečně explicitní...

Unární – Unární minus vrátí kopii instance, na kterou působí, s opačnými znaménky u reálné a imaginární části:

```
public static cplx operator-(cplx a)
{
    return new cplx(-a.Re, -a.Im);
}
```

Porovnání Dvě komplexní čísla jsou si rovna, jestliže se rovnají jak jejich reálné, tak imaginární části. Ke zjišťování rovnosti používáme operátor `==`. Překladač ovšem vyžaduje, abychom zároveň definovali i operátor `!=`:

```
public static bool operator ==(cplx a, cplx b)
{
    return (a.Re==b.Re)&&(a.Im==b.Im);
}

public static bool operator !=(cplx a, cplx b)
{
    return (a.Re!=b.Re)|| (a.Im!=b.Im);
}
```

Pro komplexní čísla se nezavádějí operace `<`, `>`, `<=` a `>=`. Proto ani my se nepokusíme definovat odpovídající operátory `<`, `>`, `<=` a `>=`.

K porovnávání objektů se zpravidla používá překrytá metoda `Equals(object o)`, definovaná již ve třídě `object`, a přetížené operátory `==` a `!=` ji obvykle využívají. Není to nezbytné, ale pokud to neuděláme – jako v tomto případě –, vypíše překladač upozornění.

Aritmetické operace

Také definice operátoru `+` pro sčítání komplexních čísel je jednoduchá:

```
public static cplx operator+(cplx a, cplx b)
{
    return new cplx(a.Re+b.Re, a.Im+b.Im);
}
```

Definice operátoru – pro odečítání je podobná a ani definice operátorů $*$ a $/$ nejsou o mnoho složitější, a proto vám jejich naprogramování přenechávám. (Ostatně můžete si je vyhledat ve zdrojovém textu tohoto příkladu na WWW.) Nyní už můžeme začít počítat:

```
cplx a = new cplx(1,1);
cplx d = -a;
cplx b = new cplx(1,1), c = new cplx(1,2), i = new cplx(0,1);
if (c == b)
    d = new cplx(11);
d = (a+b)*(c-i);
```



Kdybychom nemohli přetěžovat operátory, museli bychom místo toho deklarovat metody `Plus()`, `Krát()` ap. Poslední příkaz by pak měl vypadat

```
d = Krát(Plus(a, b), Minus(c, i));
```

už tohle je nepřehledné, a to ještě není žádný skutečný výpočet. Zkuste si představit, jak by dopadl vzorec, který obvyklým matematickým zápisu zabírá tři řádky.



Zdrojový text tohoto příkladu i s dalšími operátory najdete na WWW v souboru `Kap10\03\cplx1.cs`. V souboru `cplx2.cs` ve stejném adresáři najdete strukturu `cplx`, v níž jsme předefinovali metodu `Equals()` a tu jsme pak využili k přetížení operátorů `==` a `!=`. ♦

Konverzní operátor

V matematice se reálná čísla považují za zvláštní případ komplexních čísel, takže všude, kde se očekává komplexní číslo, můžeme zaplatit číslo reálné. Ovšem překladač C# nám nedovolí napsat např.

```
d = 1;
```

je-li `d` typu `cplx`. Zde nám mohou pomoci konverzní operátory; syntax jejich deklarace je

deklarace konverzního operátoru:

- **public static** modifikátor_užití
operator cílový_typ (typ identifikátor) { tělo }

Deklarace konverzního operátoru musí – stejně jako deklarace unární a binární operátorů – začínat klíčovými slovy `static public`. Potom následuje modifikátor užití vyjádřený jedním z klíčových slov `explicit` nebo `implicit`, které určuje, zda si přejeme, aby bylo třeba volat tento konverzní operátor vždy explicitně, nebo aby ho překladač mohl používat k implicitním konverzím. Za klíčovým slovem operátor pak následuje cílový typ – typ, na který tento operátor převěd svůj operand.

Příklad

Deklarujeme ve třídě `cplx` operátor, který umožní implicitní konverze reálných čísel na čísla komplexní:

```
public static implicit operator cplx(double x)
{
    return new cplx(x, 0);
}
```

Nyní nám překladač dovolí nejen přiřazení

```
d = 1;
```

ale i výrazy jako

```
e += 5;
```

Vraťme se ještě jednou k naší oblíbené třídě `Bod`, s níž jsme se setkali naposledy v kapitole 9.7. Protože komplexní čísla lze chápat jako body v komplexní rovině, je pochopitelné, že čas od času budeme chtít některé výsledky výpočtů s komplexními čísly graficky zobrazit. Potřebujeme tedy operátor, který umožní převod komplexního čísla na `Bod`. Pro určitost se rozhodneme, že body, které budou výsledkem takového převodu, budou mít červenou barvu. V tomto případě ale deklarujeme konverzní operátor jako explicitní:

```
public static explicit operator Bod(cplx x)
{
    return new Bod((int)x.Re, (int)x.Im, barvy.červená);
}
```

Protože komplexní čísla obsahují reálné složky a naše body mají celočíselné souřadnice, museli jsme složky komplexního čísla převést na celá čísla.

Překladač nám nyní nedovolí napsat

```
Bod b1 = e; //e je komplexní číslo
```

neboť konverzní funkci pro převod na `Bod` nesmí použít k implicitním konverzím, ale přijme příkaz obsahující explicitní konverzi

```
Bod b1 = (Bod)e;
```

Zdrojový text uvedeného příkladu najdete na WWW v souboru `Kap10\04\cplx-go.cs`. ♦

V deklaraci konverzního operátoru musí být buď cílový typ nebo typ operandu shodný s typem, v němž je tento operátor deklarován. Nelze např. v typu `cplx` deklarovat konverzní operátor pro převod celých čísel na body.

Konverze na řetězec

Často potřebujeme převést instanci nějakého typu na znakový řetězec. Tato konverze má zvláštní postavení; objeví-li se instance jakéhoko



typu v situaci, kdy překladač očekává řetězec, zavolá překladač virtuální metodu ToString(), zděděnou po třídě object. Stačí tedy předefinovat tuto metodu a není třeba definovat konverzní operátor (Připomeňme, že tato metoda je k dispozici i pro atomické typy.)

Příklad Deklarujeme-li ve třídě cplx metodu

```
public override string ToString()
{
    return "(" + Re + ", " + Im + ")";
}
```

vypiše příkaz

```
Console.WriteLine(i);
```

do standardního výstupu

```
(0, 1)
```



I tuto metodu najdete v souboru Kap10\04\cplx-go.cs. ✧

Indexování

Operátor indexování zpravidla slouží ke zpřístupnění složek pole, za pouzdrěného v instanci třídy nebo struktury; není to ale nezbytné. Definujeme ho podobně jako vlastnosti, tj. bez klíčového slova operátor. Neplatí pro něj také omezení, o nichž jsme hovořili v souvislosti s ostatními operátory: Nemusí být veřejně přístupný, nemusí být statický, může být abstraktní, virtuální nebo zapečetěný.

Specifikace jazyka C# ve skutečnosti nehovoří o operátoru indexování, ale o *indexeru*, a my tento termín budeme používat také. Zjednodušený syntaktický popis deklarace indexeru má tvar

deklarace indexeru:

- *modifikátory_{nep} typ this [seznam_parametrů] { část_get_{nep} část_set_{nep} }*

Modifikátory mohou specifikovat přístupová práva, mohou definovat indexer jako abstraktní, virtuální nebo zapečetěný. Typ je typ hodnoty, kterou indexer zpřístupňuje; zpravidla je to typ prvku pole. Klíčové slovo this určuje, že nejde o „obyčejnou“ vlastnost, ale indexer.

Pak následuje *seznam parametrů* v hranatých závorkách. Jednotlivé parametry odpovídají jednotlivým indexům. Poznamenejme, že indexy nemusí být celočíselné; k indexování lze využít třeba znakových řetězců – záleží jen na našich potřebách a programátorské dovednosti.

Část *get* určuje vrácenou hodnotu, část *set* určuje způsob uložení hodnoty. Vedle proměnné *value* v nich můžeme využít i indexy, formální parametry indexeru.

Definujeme-li v nějaké třídě indexer, můžeme pak indexovat instanci a tak přistupovat k datům, která jsou v nich uložena.

Příklad

Bod *a* v rovině má dvě složky, které označujeme *x* a *y*. Vedle toho nich často hovoříme jako o první a druhé složce (případě souřadnici); označujeme je *a₁* a *a₂*. Bude tedy rozumné definovat ve třídě Bod indexer, který nám zpřístupní tyto dvě složky:

```
public int this[int i]
{
    set
    {
        switch(i)
        {
            case 1: X = value; break;
            case 2: Y = value; break;
            default: throw new IndexOutOfRangeException(
                "Chyba v indexování Bodu");
        }
    }
    get
    {
        switch(i)
        {
            case 1: return X;
            case 2: return Y;
            default: throw new IndexOutOfRangeException(
                "Chyba v indexování Bodu");
        }
    }
}
```

Je-li nyní *x* instance třídy Bod, můžeme napsat

```
x[1] = 215;
```

a bude to znamenat totéž jako

```
x.X = 215;
```



Všimněte si, že indexy nemusí začínat od nuly. Zdrojový text t upravené třídy Bod s jednoduchým příkladem použití najdete WWW v souboru Kap10\04\cplx-go.cs. ✧

10.3 Události

I když s vyvoláváním a ošetřováním událostí se v C# setkáváme přibližně všim při programování aplikací s grafickým uživatelským rozhraním, nástroje pro to jsou k dispozici pro všechny třídy.

Jazyk C# používá pro práci s událostmi tradiční návrhový vzor nazývaný *vydavatel–předplatitel* (publisher–subscriber):

- Instance, jež může vyvolat událost (vydavatel), si udržuje seznam instancí, které vznik události zajímá (předplatitelů). Když událost nastane, vydá o tom oznámení – odešle zprávu – všem instancím ve svém seznamu, tedy všem předplatitelům.
- Instance, která chce nějak reagovat na vznik události v jiné instanci, se u ní musí zaregistrovat jako předplatitel.

Protože odeslání zprávy instanci znamená v C# zavolání její metody, je pochopitelné, že jako seznam předplatitelů slouží delegáty – tedy ukazatele na metody typu `void`. Chceme-li deklarovat delegát jako událost, použijeme navíc modifikátor `event`.

Třída, která chce na událost reagovat, musí obsahovat odpovídající metodu (tzv. *handler*).

Delegát pro událost

Prostředí .NET definuje řadu delegátů pro různé události; nic nám ale nebrání deklarovat si svůj. Metoda pro ošetřování události má typicky dva parametry. První je typu `object` a představuje odkaz na instanci, jež událost vyvolala, a druhý je typu odvozeného od třídy `EventArgs`. `EventArgs` a obsahuje „parametry události“ – doplňující informace o vzniklé události. I zde můžeme využít některou z předdefinovaných tříd nebo deklarovat svou vlastní.

Deklarace události

Syntaktický popis deklarace⁶¹ události vypadá takto:

deklarace události:

- *modifikátory*_{nep} **event** *typ deklarátor*

Modifikátory mohou vedle specifikace přístupových práv určovat, že jde o statickou, virtuální, abstraktní nebo zapečetěnou událost. *Typ* je

⁶¹ V C# lze deklarovat události také jako vlastnosti. Výklad o nich však přesahuje možnosti naší knihy.

typ delegátu představujícího handler a deklarátor je identifikátor datové složky.

Deklarujeme-li ve třídě událost pomocí klíčového slova `event`, deklarujeme vlastně delegát. Modifikátor `event` ovšem zajistí, že mimo tuto třídu pro něj lze používat pouze operátory `+=` a `-=`, nelze ho zavolat.

Vyvolání události

Nastane-li událost, musí instance, v níž k události došlo, zavolat metodu „předplatitelů“. K tomu použijeme obvyklý zápis pro volání delegátů.

Příklad

V programu máme třídu `Tlačitko`, reprezentující tlačítko v grafickém uživatelském rozhraní programu. Každé tlačítko obsahuje mj. znakový řetězec představující nápis na něm. Stisknutí tlačítka je událost, o níž chceme informovat ostatní třídy v programu. Deklarujeme proto nejprve třídu `TlačitkoEventArgs`, jež bude sloužit k přenosu informací o události:

*Třída pro přenos
informací*

```
public class TlačitkoEventArgs : EventArgs
{
    private string kdo;
    public TlačitkoEventArgs(string _kdo){kdo = _kdo;}
    public string Kdo{get{return kdo;}}
```

Tato třída je potomkem třídy `EventArgs`; navíc obsahuje znakový řetězec, který bude identifikovat tlačítko, jež událost vyvolalo.

Delegát

Pak deklarujeme delegát pro událost stisknutí tlačítka:

```
public delegate
    void UdálostTlačitka(object zdroj, TlačitkoEventArgs e);
```

Tlačítko

Třída `Tlačitko` bude obsahovat událost stisknutí:

```
public class Tlačitko
{
    private string název;
    public Tlačitko(string jméno){název = jméno;}

    // deklarace události
    public event UdálostTlačitka stisknutí;

    // Stisknutí vyvolá událost
    public void Stiskni()
    {
        if(stisknutí != null)
            stisknutí(this, new TlačitkoEventArgs(název));
    }
}
```

V metodě `Stiskni()` nejprve zkontrolujeme, zda se nějaký posluchač zaregistroval, a pokud ano, zavoláme odpovídající delegát.

Nyní deklarujeme třídu `Program`, jež bude mj. obsahovat handler. To bude velice jednoduchý, pouze vypíše informaci o tom, co se stalo:

```
static void StiskTlačítka(object zdroj, TlačítkoEventArgs e)
{
    Console.WriteLine("Někdo stiskl tlačítko " + e.Kdo);
}
```

V metodě `Main()` deklarujeme několik tlačítek,

```
Tlačítko T1 = new Tlačítko("Ano"), T2 = new Tlačítko("Ne");
```

a zaregistrujeme se jako zájemce o událost stisknutí tlačítka `T1`:

```
T1.stisknutí += new UdálostTlačítka(StiskTlačítka);
```

Pak stiskneme obě tlačítka,

```
T2.Stiskni(); // To může vyvolat událost
T1.Stiskni();
```

O události tlačítka `T2` se nikdo nezajímá, takže jeho stisknutí nebude mít žádné důsledky. Na druhé straně stisknutí `T1` způsobí, že program vypíše

Někdo stiskl tlačítko Ano



Zdrojový text tohoto příkladu najdete na WWW v souboru `Kap10\09\event.cs`. ♦

11 Výjimky

S výjimkami jsme se poprvé setkali ve 3. kapitole a od té doby je od času vyvoláváme, ale jinak se tváříme, že se o ně nemusíme starat. Výjimky ale představují mocný nástroj pro psaní bezpečných a přehledných programů. Než se ovšem pustíme do výkladu o syntaktických pravidlech, podívejme se, proč jsou vlastně tak dobré.

Proč výjimky?

Co dělat, když v programu dojde k chybě? Zatím jsme se tvářili, že nás to netýká, že se to v našich programech nemůže stát – to jen nepozorný uživatel může zadat nesprávná data.

Jenže takovýto postoj je snad možný ve školních programech, u reálných ale neobstojí ve vážně míněných aplikacích. Počítače dnes už nejsou jen hračkářské desky a programy musí počítat s tím, že něco bude jinak, než by mělo. Zkuste si např. představit, že uprostřed přistávacího manévru odmítnou palubní počítač v letadle vysunout podvozek, neboť v programu, který se o to stará, došlo k chybě, protože pilot zmáčkl špatné tlačítko.

Nebo si zkuste představit, že máte otevřený soubor, upravujete v něm data, a uprostřed úprav dojde k chybě. Pokud by program v tom okamžiku skončil, mohlo by dojít k poškození dat v něm a tím i k finančním škodám. (A nemusí jít jen o soubor s bankovními účty.)

Program by měl počítat nejen s chybami svého okolí, ale i se svými vlastními chybami. Výzkumy ukazují, že software špičkové kvality obsahuje jednu chybu přibližně v každých 10 000–20 000 řádcích zdrojového kódu [6].

Dnes se stává samozřejmým požadavkem odolnost softwaru proti chybám (fault tolerance). Přitom „chybou“ může být nesprávně zadaný vstup, chybějící nebo poškozený soubor, přerušení síťového spojení, ale také třeba dělení nulou. Chybu zjistíme v nějaké metodě, která volala jiná metoda ... a v metodě, kde chybu zjistíme, ji nedokážeme napravit, protože k tomu nemáme potřebné informace.

Příklad

Zkuste znovu spustit program `Prevod`, který jsme vytvořili v kapitole 9.4 v oddílu věnovaném metodě `Main()`, příkazem

```
Prevod Nazdar, programe
```

Tento program očekává v příkazové řádce dvě čísla v desítkové soustavě. Začne tím, že zjistí počet parametrů v příkazové řádce, a pokud je jich více nebo méně než dva, vypíše instrukce, jak se má spustit; více kontrol jsme neuměli naprogramovat.

My jsme mu zadali dva parametry, ovšem ani jeden z nich nepředstavuje celé číslo; proto vypíše obsáhlé chybové hlášení a skončí. To způsobí metoda `Convert.ToInt32()`, která jako vstup očekává znakový řetězec představující celé číslo v desítkové soustavě. Pokud dostane jiný řetězec, je to chyba, kterou ale nelze v této metodě napravit; to musí dát do pořádku ten, kdo ji se špatným parametrem zavolal.

Takže ještě jednou: Co dělat, když v programu dojde k chybě, kterou nedokážeme v daném místě napravit? V principu máme několik možností:

- Můžeme vypsat zprávu a ukončit program. To je sice dobré pro ladění, ale dodat takovýto program zákazníkovi, to prostě nelze. I když je chyba na straně uživatele. Náš program by měl alespoň popsat chybu, ke které došlo, způsobem srozumitelným uživateli, a doporučit mu, jak s programem zacházet.
- Můžeme ukončit metodu a vrátit hodnotu, která bude signalizovat chybu. To bylo oblíbené řešení ve starších programovacích jazycích – mj. v C a v prvních verzích C++. (Zkusili jsme to ve 3. kapitole: faktoriál je definován pouze pro nezáporná čísla a jeho hodnoty jsou vždy kladné. Jedna z verzí metody `f()`, počítající faktoriál, proto v případě záporné hodnoty parametru vracela 0.) Programátora ale nikdo nenutí vrácenou hodnotu kontrolovat, a tak se může stát, že program bude počítat s nesmyslným výsledkem. Navíc tento postup nelze použít vždy; např. výsledkem metody `Convert.ToInt32()` může být jakékoli celé číslo, takže nelze žádnou hodnotu vyhradit jako příznak chyby.
- Můžeme do každé instance přidat datovou složku, která bude sloužit jako příznak chyby. Jestliže metoda volaná pro tuto instanci zjistí chybu, uloží do ní kód této chyby. Programátor pak musí po ukončení metody kontrolovat, zda proběhla úspěšně. (Tady je opět slabé místo: Kdo k tomu programátora donutí? Když na to zapomeneme, může se stát, že program poběží dál a bude dělat nesmysly.)
- Můžeme použít mechanismus výjimek. Ten umožňuje bezpečný přenos řízení z místa, kde jsme zjistili chybu, na místo, kde ji můžeme ošetřit. O něm bude tato kapitola.

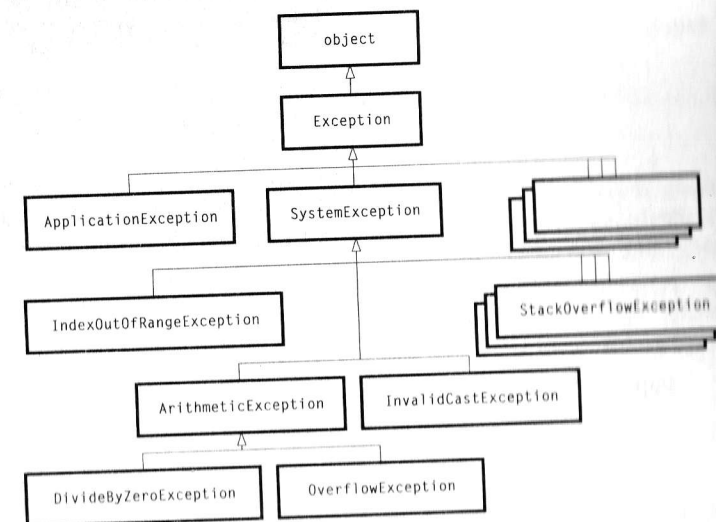
Výjimka v C#

Když v programu v C# vznikne výjimka, vytvoří se objekt, který nese informace o vzniklém problému do místa, kde tento problém bude možné vyřešit. Nositel informací o druhu problému je předem určený typ tohoto objektu; navíc do něj lze uložit ještě znakový řetězec s dalšími informacemi.

Aby nebyly věci tak snadno srozumitelné, říká se těmto objektům také „výjimky“. Slovo „výjimka“ tedy může znamenat buď chybu vzniklou v programu nebo objekt, který o ní nese informace. Podle typu tohoto objektu se pak hovoří o „výjimce typu `System.IndexOutOfRangeException`“ ap.

Třídy pro přenos informací o výjimkách

Mechanismus výjimek v C# používá k přenosu informací o výjimkách třídy odvozených od standardní třídy `System.Exception`, která je buďto prostředním potomkem třídy `object`, tj. `System.Object`.



Obr. 11.1 Základ hierarchie objektových typů pro práci s výjimkami v C#; většina z nich je deklarována v prostoru jmen `System`. Prázdné obdélníky naznačují typy, o nichž nebudeme v této knize hovořit.

Exception

Od třídy `Exception` je odvozena řada potomků; mezi jejími prostředními potomky najdeme třídu `SystemException`, od níž jsou odvozeny třídy `ArithmeticException` (aritmetická výjimka), `IndexOutOfRangeException` (index mimo povolený rozsah), `InvalidCastException` (nedovolené přetypování) a další, jež slouží pro indikaci ne

nějších chyb. Od třídy `ArithmeticException` jsou ještě odvozeny třídy `DivideByZeroException` (dělení nulou) a `OverflowException` (celočíselné přetečení).

Dalším bezprostředním potomkem třídy `Exception` je třída `ApplicationException`, jež slouží k signalizaci méně závažných chyb aplikace.

Všechny zmíněné třídy leží v prostoru jmen `System`. Některé další, zde neuvedené třídy výjimek mohou ležet v prostorech jmen vnořených do prostoru `System`. Všechny třídy výjimek mají vedle konstruktoru bez parametrů také konstruktor, kterému lze předat znakový řetězec s bližšími informacemi o vzniklé chybě.

Vznik výjimek

Některé výjimky může vyvolat samo prostředí .NET Framework, jak např. o celočíselné přetečení v bloku `checked`, o důsledek dělení nulou nebo o vyčerpání volné paměti programu. Ostatní výjimky vyvoláváme příkazem `throw`, za nímž uvedeme odkaz na objekt, který nese informace o chybě. Formální popis syntaxe vyvolání výjimky už známe:

vyvolání_výjimky:

- **throw** výraz_{nep} ;

kde výraz musí představovat odkaz na instanci třídy odvozené od `System.Exception`. Můžeme použít odkaz na existující objekt nebo můžeme vytvořit nový objekt pomocí `new`. Po vzniku výjimky se přeruší přirozené pořadí provádění příkazů a program začne hledat handler, část kódu, která vzniklou situaci ošetří. Tento výraz lze za jistých okolností vynechat; o tom si povíme v následujícím oddílu.

Příklad Jako příklad si připomeneme metodu `f()` z kapitoly 3.4, jež počítala faktoriál celého čísla. Začínala takto:

```
static int f(int n)
{
    if(n < 0) throw new Exception("Faktoriál není definován");
```

Zde nejprve otestujeme, zda je parametr v předepsaném rozmezí, a pokud není, vyvoláme výjimku. (Místo třídy `Exception` by bylo vhodnější použít třídu `System.ArgumentException`, jež se standardně používá k indikaci chybně zadaných parametrů metody.)

Jestliže tuto metodu zavoláme příkazem `fakt(-5)`, skončí program chybovým hlášením. To sice na první pohled vypadá, že jsme neudělali žádný pokrok, neboť program opět končí, ale jak uvidíme dále, máme nyní možnost s chybou něco dělat. Za prvé se brzy naučíme

výjimku zachytit a ošetřit, a za druhé už víme, že z chybového hlášení můžeme při ladění leccos vyčíst – tom jsme hovořili v kap. 3.4. ✧

11.3 Ošetřování výjimek

Syntax Práce s výjimkami je v C# založena na tzv. hlídaném bloku⁶². Operace, jež se nemusí podařit, tj. při nichž může vzniknout výjimka, uzavřeme do bloku, před nějž zapíšeme klíčové slovo `try`. Za něj pak připojíme jeden nebo několik handlerů nebo koncovek (případně obojí). Ukažme si syntaktický popis hlídaného bloku:

hlídaný blok:

- **try** { příkazy } handlery_{nep} koncovka_{nep}

Hlídaný blok musí obsahovat alespoň jeden handler nebo koncovku (nebo obojí). Handlerů může být více, koncovka může být nejvýše jedna. Koncovky zatím ponecháme stranou, vrátíme se k nim později.

Handler

Handler je část programu, která může ošetřit výjimku. Jeho syntax je

handler:

- **catch**(typ identifikátor){ tělo_handleru }
- **catch**{ tělo_handleru }

První možnost začíná klíčovým slovem `catch`⁶³, za kterým následuje v závorkách typ výjimky, kterou může tento handler ošetřit („typ handleru“). Musí jít o třídu odvozenou od `Exception`. Identifikátor, který následuje, bude v těle handleru představovat odkaz na objekt, který nese informace o výjimce. Budeme mu také říkat „parametr handleru“. Tělo handleru je tvořeno příkazy, které se zachycenou výjimkou „něco udělají“, tedy které řeší vzniklou situaci.

C+

⁶² Celý mechanismus práce s výjimkami je velmi podobný jako v Javě nebo v C++. Ve srovnání s C++ zde navíc najdeme koncovku bloku (`finally`), která je zjevně inspirována strukturovanými výjimkami (běžné rozšíření jazyka C na PC pod Windows).

⁶³ `Try` znamená „zkus, pokus se – tj. zkus provést následující operace. O bloku s prefixem `try` budeme občas hovořit jako o „pokusném bloku“. `Throw` znamená „hod“, vrhni – „vyhod“ výjimku“ česky moc pěkně nezní, a proto budeme raději říkat, že se výjimky vyvolávají. `Catch` znamená „chyt“ – handler výjimku zachytí a ošetří. U koncovek se ještě setkáme se slovem `finally`, které znamená „na konec“ – tyto operace se provedou na konci pokusného bloku, ať v něm vznikla výjimka nebo ne.

Když vznikne výjimka

Druhá možnost, v níž chybí závorky se specifikací typu výjimky a klíčovým slovem `catch`, představuje tzv. univerzální handler, který zachytí všechny výjimky bez rozdílu.

Už víme, že při vzniku výjimky se přeruší přirozená posloupnost plnění příkazů a začne se hledat vhodný handler. To znamená, že v bloku, ve kterém vznikla výjimka, se již následující příkazy neprovedou, program tento blok opustí. Je-li to pokusný blok a je-li k němu připojen vhodný handler, přejde do něj. Jinak přejde do nadřazeného bloku (pokud v dané metodě nějaký je) a bude hledat vhodný handler u něj atd. Nenajde-li program vhodný handler v dané metodě, přejde do metody, která ji zavolala, a bude se chovat, jako kdyby výjimka vznikla v příkazu, kterým jsme danou metodu volali. To znamená, že opět přeskočí všechny příkazy do konce bloku a bude hledat vhodný handler. O tomto procesu hovoříme jako o *šíření výjimky*. Nalezení handleru a vstupem do něj se výjimka pokládá za ošetřenou.

Šíření výjimky

Pokud v handleru nevznikne další výjimka, bude program po opuštění handleru pokračovat za hlídaným blokem. To znamená, že přeskočí případné další handlersy připojené k témuž pokusnému bloku (Provede ale příkazy z koncovky, pokud je k bloku `try` připojena.)

Nenajde-li program vhodný handler ani v metodě `Main()`, převezme výjimku prostředí .NET Framework, jež vypíše informace o typu, místu vzniku a postupu šíření výjimky a program ukončí.

Jestliže v pokusném bloku nevznikne výjimka, proběhnou všechny příkazy v jeho těle. Program pak přeskočí handlersy, které za ním následují, a pokračuje příkazy za nimi. (Pokud je k bloku `try` připojena koncovka, provede nejprve příkazy v ní.)

Výpis informací

Všechny instance výjimkových tříd obsahují vlastnost `Message`, jež zpřístupňuje znakový řetězec s chybovým hlášením, a vlastnost `StackTrace`, jež obsahuje informace o postupu šíření výjimky od místa vzniku po místo zachycení handlerem. Obojí lze využít zejména při ladění.

Příklad

Ponecháme zatím stranou koncovku a podíváme se, jak bychom mohli využít toho, co jsme se právě dozvěděli. Ve 3. kapitole – v oddílu 4.3 – jsme napsali funkci `f()`, jež počítala faktoriál, a pokud její parametr neležel v rozmezí od 0 do 12, vyvolala výjimku typu `Exception` (žádný jiný jsme ještě neznali). Připomeňme si alespoň začátek zdrojového kódu metody `f()`:

```
const int Mez = 12;

static int f(int n)
{
```

```
    if(n < 0) throw
        new Exception("Faktoriál "+ n + " není definován");
    if(n > Mez) throw new Exception("Příliš velké číslo "+
n);
    // Vlastní výpočet není zajímavý
}
```

Upravme nyní metodu `Main()` takto:

```
static void Main()
{
    try {
        Console.WriteLine("Zadej celé nezáporné číslo: ");
        int n = VV.InInt();
        Console.WriteLine("Faktoriál čísla {0}"
            + " je {1}.", n, f(n));

        Console.WriteLine("Výpočet proběhl bez problémů.");
    }
    catch(Exception e)
    {
        Console.WriteLine
            ("Špatně zadaný vstup\n"+e.Message+e.StackTrace);
    }
    Console.WriteLine("Nashledanou.");
}
```

Podívejme se nejprve na běh programu za „normálních“ okolností. Program vypíše žádost o zadání celého čísla. Uživatel zadá řekněme číslo 8. Volání metody `f()` proběhne, aniž nastane výjimka, a program vypíše výsledek a oznámení, že výpočet proběhl bez problémů a přeskočí handler, vypíše závěrečný pozdrav a skončí.

Jestliže však uživatel zadá např. číslo -9 , vznikne výjimka v metodě `f()` v prvním příkazu `throw`. To znamená, že se nejprve vytvoří instance výjimky typu `Exception`. Pak se přeskočí všechny následující příkazy do konce bloku, tj. do konce těla metody. Protože v této metodě není žádný handler, skončí tím běh metody `f()` a program se vrátí na místo, odkud byla volána, k příkazu označenému vykřičníkem v koncovce řádku. Tento příkaz se už nedokončí a přeskočí se všechny příkazy do konce bloku. To znamená, že se nevytisknou slova Faktoriál 8, je, ani hlášení Výpočet proběhl bez problémů.

K tomuto bloku je ovšem připojen handler, který může ošetřit výjimku typu `Exception`, a proto program do přejde tohoto handleru a vypíše upozornění Špatně zadaný vstup, dále zprávu Faktoriál 8 není definován a posloupnost volání metod, jež vedla ke vzniku výjimky. Po ukončení handleru pokračuje program prvním příkazem za hlídaným blokem. To znamená, že – stejně jako v případě, že k chybovému došlo – vypíše závěrečný pozdrav.



Zdrojový text takto upraveného program najdete na WWW v souboru
Kap10\01\Fakt.cs. ✧

Všimněte si, že jsme do pokusného bloku uzavřeli všechny operace, které se týkají normálního běhu programu, tj. i ty, které výjimku ne
mohou vyvolat; zde se jedná o volání metody výpis zprávy, že vše
proběhlo bez problémů. *Blok try totiž obsahuje kód, který se při
normálním běhu programu, a handlers se starají o ošetřování chyb.*
Příkazy za voláním metody `f()` nemá smysl provádět, pokud se výpočet faktoriálu nepovede. Aparát pro ošetřování výjimek tak přispívá k
ke zpřehlednění programu.

Výběr handleru

Handler zachytí výjimku typu, který je uveden v závorkách za klíčovým slovem `catch`, nebo jakéhokoli typu, který je od něj odvozen. *I zde se tedy uplatňuje základní pravidlo objektového programování, že potomek může vždy zastoupit předka.* Speciálně handler, v němž deklarujeme typ výjimky `Exception`, zachytí naprosto všechny výjimky.

Přitom program použije první handler, na který při šíření výjimky narazí a který může výjimku ošetřit. To nám umožňuje výjimky tříditi. Můžeme se např. rozhodnout, že budeme zvlášť ošetřovat výjimky typu `DivideByZeroException`, ale všechny ostatní výjimky „hodíme do jednoho pytle“.

Příklad

Předchozí příklad obsahoval jeden skrytý problém: Neumožňoval rozlišit, zda výjimka vznikla v metodě `f()` kvůli nevhodné hodnotě parametru nebo v metodě `InInt()` kvůli špatně napsanému číslu. V obou metodách totiž vzniká výjimka typu `Exception`.

Upravme tedy metodu `f()` pro výpočet faktoriálu tak, aby v případě nesprávného parametru vyvolávala výjimku typu `ArgumentException`:

```
static int f(int n)
{
    if(n < 0) throw
        new ArgumentException
            ("Faktoriál "+ n + " není definován");
    if(n > Mez) throw
        new ArgumentException("Příliš velké číslo: " + n);
    // Vlastní výpočet stále není zajímavý
}
```

V metodě `Main()` můžeme rozlišit druh chyby podle jejího původu:

```
static void Main()
{
    try {
        Console.WriteLine("Zadej celé nezáporné číslo: ");
        int n = VV.InInt();
    }
```

```
        Console.WriteLine("Faktoriál čísla {0}"
            + " je {1}.", n, f(n));
        Console.WriteLine("Výpočet proběhl bez problémů.");
    }
    catch(ArgumentException e)
    {
        Console.WriteLine("Špatně zadaný vstup\n"+e.Message
            +e.StackTrace);
    }
    catch
    {
        Console.WriteLine("Vstup nepředstavuje celé číslo.");
    }
    Console.WriteLine("Nashledanou.");
}
```

Tentokrát jsme k pokusnému bloku připojili dva handlers. První chycuje už známý případ – špatný parametr. Druhý zachytí všechny ostatní výjimky (v našem případě výjimky vzniklé při zpracování vstupu). Jestliže dojde k výjimce typu `ArgumentException`, zachytí první handler, protože na něj program narazí dříve. Dojde-li k jaké jiné výjimce, zachytí ji až druhý handler. (Už víme, že to je univerzální handler, takže zachytí všechny výjimky, které k němu projdou.)

Pošli to dál

V handleru můžeme vyvolat novou výjimku stejného nebo jiného typu. Tu už nezachytí žádný z následujících handlerů u téhož bloku, ale až handler nadřazeného bloku. Jestliže uvnitř handleru použijeme příkaz

`throw;`

v němž chybí výraz, vyvoláme tím výjimku stejného typu a se stejnou instancí, jako byla aktuální výjimka.

To se hodí, když potřebujeme rozdělit ošetření výjimky do několika míst. Např. v metodě `f1()` musíme při vzniku výjimky uzavřít otevřené soubory, které bychom jinak ponechali otevřené. V metodě `f2()` je třeba zavolat `Dispose()`, je třeba uzavřít připojení k databázi, ale celkové ošetření výjimky je možné až v metodě `Main()`. V takovém případě v každé z těchto metod výjimku zachytíme, uděláme, co je třeba, a pak ji „pošleme dál“.

Koncovka

Když vznikne výjimka, postará se program o správné zrušení všech proměnných v blocích a metodách, které při hledání handleru prošel.

opustí. Instance polí a objektových typů zaniknou, když zanikne odkazy na ně; o to se postará automatická správa paměti.

Může se ale stát, že si v metodě otevřeme soubor a při jeho zpracování vznikne výjimka. Před opuštěním metody bychom ho potřebovali zavřít. Může se stát, že napíšeme na obrazovku nápis, který bychom zpracování probíhá úspěšně, a tento nápis bychom při výjimce potřebovali odstranit.

Může se zdát, že by se o to měl postarat handler; jenže jak uzavřít soubor, tak odstranění nápisu musíme udělat nejen v případě, že akce podaří, ale i v případě, že v průběhu zpracování dojde k výjimce. Abychom je nemuseli programovat dvakrát, použijeme koncovku daného bloku. Operace v ní se provedou, ať skončí pokusný blok normálně nebo výjimkou.



Poznamenejme, že koncovka se provede i v případě, že pokusný blok opustíme příkazem `return`, `break`, `continue` nebo `goto`. Je-li k bloku `try` připojen jak handler, tak koncovka a odpovídá-li typ výjimky typu handleru, provede se nejdříve handler, pak koncovka. Koncovka proběhne i v případě, že v handleru vznikne další výjimka.

Syntax koncovky je:

koncovka hlídaného bloku:

- **finally{ příkazy }**

Použití si ukážeme nejprve jen schématicky:

```
try {
    OtevřiSoubor(F);      // Zkus to
    Zpracuj(F);
}
catch(Exception e){
    NecoStímUdělej();    // a když se to nepovede...
}
finally{
    ZavřiSoubor(F);      // Tohle udělej v každém případě
}
```

Příklad Trochu podrobněji ukáže fungování koncovky následující příklad, v němž předpokládáme, že kódu předchází deklarace `using System`. Metoda `Main()` obsahuje dva do sebe vnořené bloky `try`, k nimž je připojen vždy jeden handler a koncovka.

Ve vnitřním bloku `try` vznikne výjimka typu `ArgumentException`, kterou zachytí až handler vnějšího bloku:

```
public static void Main(string[] args)
{
    try {
        // Vnější blok
```

```
try
{
    Console.WriteLine("před výjimkou");
    throw new ArgumentException();
}
catch(ArithmeticException e) // Handler vnitřního
{                             // bloku
    Console.WriteLine("handler vnořeného bloku");
}
finally                        // Koncovka vnitřního
{                              // bloku
    Console.WriteLine("koncovka vnořeného bloku");
}
Console.WriteLine("za vnořeným blokem try");
}
catch(ArgumentException e) // Handler vnějšího
{                             // bloku
    Console.WriteLine("handler vnějšího bloku");
}
finally                        // Koncovka vnějšího
{                              // bloku
    Console.WriteLine("koncovka vnějšího bloku");
}
}
```

Vnitřní blok skončí, aniž se použil jeho handler. Proto se provede je koncovka. Pak ale proběhne handler vnějšího bloku a teprve po něm koncovka tohoto bloku. Uvedený program vypíše

```
před výjimkou
koncovka vnořeného bloku
handler vnějšího bloku
koncovka vnějšího bloku
```



Zdrojový text tohoto příkladu najdete na WWW v souboru `Kap11\koncovka.cs`.

11.4 Složitější příklady

V příkladech ošetřování výjimek se vrátíme ke čtení dat z klávesnice k přetěžování operátorů.

Ještě jednou vstupy

Ve 4. kapitole jsme navrhli třídu `WV`, jejíž statické metody `ToInt()` a `ToIntDouble()` sloužily ke čtení čísel typu `int`, resp. `double` ze standardního vstupu. Obě metody byly dobře použitelné pro čtení z koncovky, nicméně pro čtení ze souboru se příliš nehodily. Pokud jsme neznali předem počet položek v souboru, skončilo čtení výjimkou, jsme neuměli ošetřit.

Podívejme se na tuto třídu ještě jednou.

Metody `InInt()` nejsou špatné, ale konec souboru je při vstupu natolik častá záležitost, že se výjimky pro jeho ošetření příliš nehodí. Koneckonců, výjimky by měly být výjimečné. (I když si lze představit situaci, kdy se nám bude hodit, že konec souboru způsobí výjimku. Ostatně v příští kapitole uvidíme, že některé metody standardních proudových tříd se takto opravdu chovají – jen třída výjimky se jmenuje jinak.)

Jak tedy zajistit, aby se uživatel třídy `VV` dozvěděl, že při čtení narazil na konec souboru? Jedním z možných řešení je napsat metodu, která přečte údaj do proměnné předané odkazem a vrátí logickou hodnotu signalizující, zda se čtení podařilo.

Ponecháme tedy metodu `InInt()` jak je a přidáme do třídy `VV` přečištěnou metodu `bool Čti(ref int x)`, která bude signalizovat konec souboru tím, že vrátí `false`. Tato metoda vnitřně využije už hotovou metodu `InInt()`. (Metoda `InInt()`, stejně jako `InDouble()`, bude i nadále vyvolávat výjimku při dosažení konce souboru.)

První, o co se postaráme, je rozlišení výjimky způsobené koncem souboru od výjimky způsobené chybou v řetězci. Definujeme vlastní třídu pro výjimky způsobené koncem souboru,

```
public class EOFException: Exception
{
    public EOFException(string s): base(s){}
}
```

Dále upravíme metodu `ČtiŘádek` tak, aby při dosažení konce souboru vyvolala výjimku tohoto typu:

```
private static void ČtiŘádek()
{
    řádek = Console.ReadLine();
    if(řádek == null) throw
        new EOFException("Čtení za koncem souboru.");
    // a dále stejně jako ve 4. kapitole
}
```

Nyní už můžeme napsat metodu `Čti()`:

```
public static bool Čti(ref int x)
{
    try {
        x = InInt();
        return true;
    }
    catch(EOFException e) {
        return false;
    }
}
```

Je to jednoduché: Metoda `Čti()` zkusí přečíst další číslo ze standardního vstupu. Pokud se to podaří, uloží přečtenou hodnotu do skutečného parametru předávaného odkazem a vrátí `true`, jinak vrátí `false`. Velmi podobná je i metoda `bool Čti(ref double x)`, která se pokusí přečíst hodnotu typu `double` a uložit ji do skutečného parametru.

Poznamenejme, že chyba ve znakovém řetězci, který představuje číslo, za výjimku opravdu stojí, a tuto výjimku v žádné z metod neošetřujeme; tu musí ošetřit až programátor, který třídu `VV` použije.

Podívejme se na příklad použití metody `Čti()`: V souboru `data.dta` máme neznámý počet celých čísel; chceme je přečíst a zpracovat (pro jednoduchost je prostě vypíšeme na obrazovku):

```
// Soubor TEST.CS
using System;

public class Test
{
    public static void Main()
    {
        int x = 0;
        try {
            while(VV.Čti(ref x)) Console.WriteLine(x);
        }
        catch(FormatException e) {
            Console.WriteLine(e.Message+
                "\nTo není číslo, končím");
        }
        catch(OverflowException e) {
            Console.WriteLine(e.Message+
                "\nMoc velké číslo, končím");
        }
    }
}
```

Tento program spustíme z příkazové řádky příkazem

```
test < data.dta
```

Budou-li data v souboru `data.dta` pořádku, bude cyklus `while` probíhat, dokud při čtení nenarazíme na konec souboru; pak skončí, protože metoda `Čti()` vrátí hodnotu `false`.

Vyskytne-li se v datech chyba, vznikne výjimka typu `FormatException` (řetězec nepředstavuje číslo) nebo `OverflowException` (číslo je moc velké). V tom případě vypíšeme odpovídající upozornění.

Zdrojový text nové verze třídy `VV` najdete na WWW v souboru `Kap11\03\VV.cs`. Testovací program je v souboru `test.cs` spolu se souborem `data.dta` je v témže adresáři. ♦



Rovnost komplexních čísel

V oddílu 10.1, věnovaném přetěžování operátorů, jsme navrhli třídu `cplx` a přetížili jsme pro ni mj. operátory `==` a `!=`. Překladač nás tehdy upozorňoval, že vhodnější je nejprve překrýt metodu `Equals()`, zde děnou ve třídě `object`, a tu využít k definici zmíněných operátorů. Metoda `Equals()` má hlavičku

```
public override bool Equals(object o);
```

tj. jejím parametrem může být odkaz na libovolný objekt v C#. Chceme-li testovat jeho rovnost s aktuální instancí třídy `cplx`, musíme parametr `o` přetypovat. Pokud se přetypování nepodaří – odkaz `o` obsahuje `null` nebo odkazuje na objekt jiného typu – vznikne výjimka, kterou zachytíme a v handleru vrátíme `false`. Jestliže se přetypování podaří, vrátíme výsledek porovnání reálné a imaginární části.

```
public override bool Equals(object o)
{
    try {
        cplx c = (cplx)o;
        return c.Re == Re && c.Im == Im;
    }
    catch(Exception e) { // Přetypování se nepodařilo,
        return false;    // nejsou stejného typu a proto
    }                    // se nemohou rovnat
}
```

Implementace přetížených operátorů `==` a `!=` je pak už přímočará:

```
public static bool operator ==(cplx a, cplx b)
{
    return a.Equals(b);
}

public static bool operator !=(cplx a, cplx b)
{
    return !a.Equals(b);
}
```



Úplný zdrojový text tohoto příkladu, který obsahuje navíc přetížené metody⁶⁴ `GetHashCode()` a `ToString()`, najdete na WWW v souboru `Kap11\04\cplx3.cs`.

⁶⁴ Metoda `int GetHashCode()` vrátí hodnotu, jež slouží pro ukládání instancí do tzv. hešových tabulek; překladač vypíše varování, překryjeme-li metodu `Equals()`, aniž bychom překryli metodu `GetHashCode()`. Výklad o hešových tabulkách ale přesahuje možnosti knihy pro zelenáče.

12 Soubory, vstupy a výstupy

Soubor a proud

Jazyk C# rozlišuje *soubory* (file) a *proudy* (stream).⁶⁵ *Soubor* je množina dat, která existuje na disku nebo v jiném paměťovém médiu víceméně nezávisle na běhu programu a se kterou zacházíme jako s logickým celkem.

Na druhé straně *proud* si můžeme představovat jako nástroj pro přenos dat mezi *zdrojem* (například programem) a *spotřebičem* (to může být nejen soubor, ale i síťové spojení, místo v paměti...). Zdrojem dat ovšem může být také soubor nebo síťové spojení a spotřebičem program – záleží na okolnostech.

System.IO

Nástroje pro práci se soubory a vstupními a výstupními proudy jsou v C# součástí knihovny; najdeme je v prostoru jmen `System.IO`. Ten prostor jmen ovšem obsahuje značné množství různých tříd, což může být pro začátečníka nejen nepřehledné, ale i znechucující. Brzy však zjistíte, že to není zdaleka tak zlé, jak to na první pohled vypadá.

V této kapitole se ovšem omezíme jen na nákladní informace.

Obrázek 12.1 ukazuje hierarchii základních tříd pro vstupní a výstupní operace. Podíváme-li se na něj pozorně, zjistíme, že obsahuje několik skupin tříd.

První z nich tvoří třídy `File` a `Directory`, jež obsahují nástroje manipulace se soubory a adresáři – umožňují zjišťovat, zda soubor nebo adresář existuje, vytvořit ho, přejmenovat ho, přesunout, zrušit ho atd.

Dále tu najdeme abstraktní třídu `Stream`, která je společným předkem tříd pro vstupní a výstupní operace. Od ní je odvozena mj. třída `FileStream`, sloužící ke čtení ze souboru a k zápisu do něj, dále třída `MemoryStream` pro přímou práci s pamětí nebo třída `NetworkStream` pro odesílání a přijímání dat prostřednictvím síťového spojení.

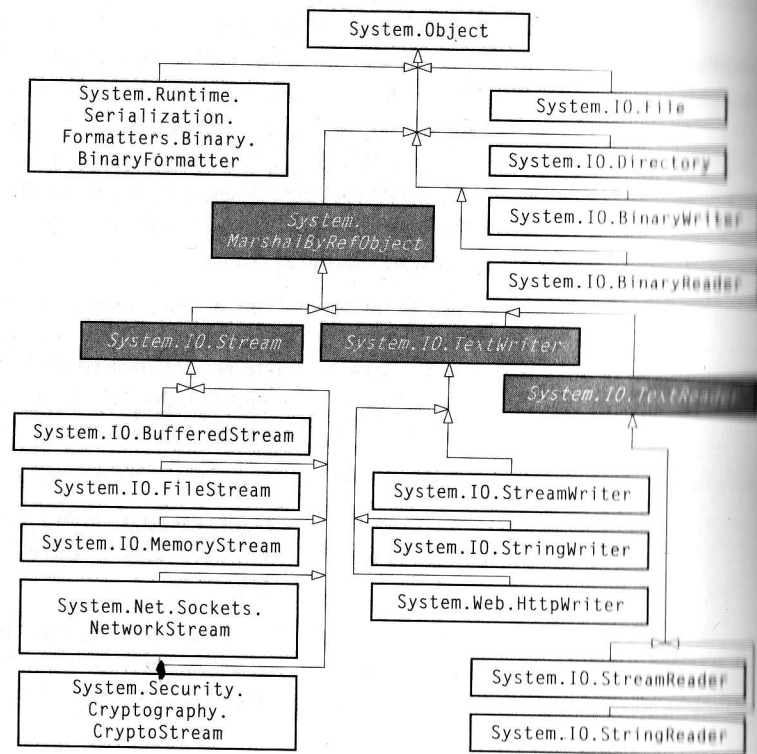
Další abstraktní třída, `TextWriter`, má jako přímé potomky třídy `StreamWriter`, `StringWriter` a `HttpWriter`, jež poskytují nástroje pro práci se znakovými proudy (tedy s textovými soubory ap.). Podobné abstraktní třídy `TextReader` jsou odvozeny instančními třídami `StreamReader`, `StringReader` a `HttpReader`.

C +

⁶⁵ Podobně jako např. jazyky C, C++ nebo Java. Návrh proudových způsobů jejich skládání je trochu podobný jako v Javě. Na tuto podobnost nelze příliš spoléhat.

der a StringReader, jež slouží pro čtení z textového souboru, resp. pro čtení z paměti (z instance třídy string).

Většina metod těchto tříd může vyvolat výjimku typu System.IO.IOException. Vedle toho mohou vyvolat řadu dalších výjimek; podrobnosti najdete v dokumentaci.



Obr. 12.1 Nejběžnější třídy pro vstupní a výstupní operace. Šedé jsou abstraktní třídy

12.1 Soubor a adresář

Už víme, že třída File obsahuje nástroje pro práci se souborem. Všechny její metody jsou statické, takže není třeba vytvářet její instance. Obsahuje výčetové typy, jež umožňují specifikovat režim práce se souborem, režim sdílení souboru a způsob jeho otevření.

Způsob práce se souborem

Podívejme se nejprve na výčetové typy, jež umožňují specifikovat způsob práce se souborem.

Výčetový typ FileMode určuje způsob otevření souboru. Obsahuje hodnotu Open, jež příkazuje otevřít existující soubor, dále tu najdeme hodnotu Truncate, jež příkazuje otevřít existující soubor a vymazat jeho obsah, a hodnotu Create, jež příkazuje vytvořit nový soubor (pokud existuje, bude přemazán). Hodnota CreateNew příkazuje vytvořit nový soubor, a pokud existuje, vznikne výjimka typu IOException. Možnost Append otvírá soubor pro připojování na konec a možnost OpenOrCreate příkazuje otevřít existující soubor, a pokud neexistuje, vytvořit ho.

Výčetový typ FileAccess určuje možnosti přístupu k souboru. Obsahuje tři hodnoty, a to Read, Write a ReadWrite. První z nich povoluje čtení ze souboru, druhá zápis do něj a třetí povoluje obojí.

Výčetový typ FileShare vymezuje možnosti sdílení souboru mezi více programy. Obsahuje mj. hodnoty None, Read, Write a ReadWrite. První z nich zakazuje sdílení, další umožňují po řadě čtení sdíleného souboru, zápis do něj a obojí.

Metody třídy File

Zde se zmíníme jen o některých z metod této třídy; podrobnější informace najdete v dokumentaci.

Metoda bool Exists(string path) umožňuje zjistit, zda existuje soubor se jménem určeným parametrem path. Chceme-li vytvořit nový soubor s daným jménem, použijeme metodu Create(string path). Metoda Delete(string path) soubor smaže, metoda Move(string oldPath, string newPath) soubor přesune na nové místo.

Metoda FileStream Open(string s, FileMode fm) vytvoří proud typu FileStream připojený k danému souboru. Přetížené verze této metody mají jako další parametry FileAccess a příp. FileShare. Podobné metody FileStream OpenRead(string s) a FileStream OpenWrite(string s) vytvoří datové proudy připojené k souboru a otevřou soubor pro čtení, resp. zápis.

Třída Directory

Tato třída obsahuje nástroje pro vytváření, rušení a procházení adresářů. Podobně jako ve třídě File zde najdeme pouze statické metody, takže nemusíme vytvářet její instance.

Metody `Exists()`, `Delete()` a `Move()` mají podobný význam jako u třídy `File`. Metoda `CreateDirectory(string path)` vytvoří adresář se zadaným jménem. Metoda `string[] GetFiles(string path)` vrátí pole řetězců se jmény souborů v adresáři daném parametrem `path`; metoda `string[] GetDirectories(string path)` vrátí jména podadresářů. Metoda `DirectoryInfo GetParent(string path)` vrátí informace o nadřazeném adresáři.

Příklad Napišeme jednoduchý program, který vypíše obsah zadaného adresáře. Požadovaný adresář budeme zadávat jako parametr v příkazovém řádku při spuštění programu.

K tomu použijeme třídu `Directory` a její metody `GetFiles()` a `GetDirectories()`. Vlastní program bude nesmírně jednoduchý:

```
using System;
using System.IO;

class VypisAdresare
{
    static void Main(string[] zadani)
    {
        string cesta = "."; // Implicitně se vypíše
        if(zadani.Length == 1) // aktuální adresář
            cesta = zadani[0];

        string[] soubory = Directory.GetFiles(cesta);
        foreach(string jmeno in soubory) // Soubory
            Console.WriteLine(jmeno);

        string[] podadresare = Directory.GetDirectories(cesta);
        foreach(string jmeno in podadresare) // Podadresare
            Console.WriteLine(jmeno);
    }
}
```



Zdrojový text tohoto příkladu najdete na WWW v souboru `Kap12\01\vypis.cs`. ✧

12.2 Čtení a zápis binárních dat

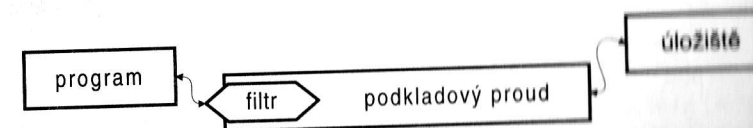
Jako „binární“ označujeme data ve tvaru, v jakém s nimi pracuje počítač. Ze 6. kapitoly např. víme, že jedno číslo typu `int` zabere v paměti 4 bajty a obsahuje hodnotu vyjádřenou ve dvojkové soustavě. Jestliže tyto 4 bajty okopírujeme do souboru, dostaneme soubor, který nebude čitelný pro člověka, ale počítač s ním bude pracovat velice snadno.

Pro čtení binárních dat ze souboru, resp. zápis těchto dat do souboru používáme třídu `FileStream`. Její instanci můžeme získat buď pomocí

metody `Create()` třídy `File` nebo voláním konstruktoru třídy `FileStream`. Parametrem metody `Create()`, stejně jako parametrem konstruktoru je znakový řetězec představující cestu a jméno souboru. (Dále hodnota typu `FileMode` udávající způsob otevření souboru. (Dále mohou být, jak víme, hodnoty typu `FileAccess`, udávající způsob přístupu k souboru, a `FileShare`, určující režim sdílení souboru.)

Proud je po vytvoření automaticky otevřen; po skončení práce bychom ho měli uzavřít pomocí metody `Close()`.⁶⁶

Pro čtení, resp. zápis můžeme použít metody `Read()`, resp. `Write()` třídy `FileStream`. Ve skutečnosti ale nejsou tyto metody mnoho plněné, protože očekávají parametry typu `byte[]` a s ničím jiným pracovat neumějí. Proto proud `FileStream`, stejně jako `MemoryStream` nebo `NetworkStream`, zpravidla skládáme s proudy jako `BinaryWriter` nebo `BinaryReader` či s některými dalšími. Jeden proud pak funguje jako „filtr“, jež poskytuje programu vhodné metody, a druhý („podkladový“) proud obstarává práci se souborem či jiným úložištěm. Symbolicky to naznačuje obr. 12.2.



Obr. 12.2 Typické použití proudů v C#. Filtr upravuje data, podkladový proud pracuje s úložištěm (souborem, sítí...)

Čtení a zápis atomických typů

Pro práci s atomickými typy použijeme filtry `BinaryReader` a `BinaryWriter`. Tyto proudy nabízejí metody jako `ReadInt32()`, `ReadInt64()`, `ReadDouble()`, resp. `WriteInt32()`, `WriteInt64()`, `WriteDouble()` a další, které umožňují číst, resp. zapisovat hodnoty atomických typů. Poznamenejme, že pokud tyto metody narazí na konec souboru, vyvolají výjimku typu `System.IO.EndOfStreamException`.

Parametrem konstruktoru tříd `BinaryReader`, resp. `BinaryWriter` může být instance třídy odvozené od třídy `Stream` – `FileStream`, `MemoryStream` atd., jež představují „podkladový“ proud.

Příklad:

Napišeme metodu `Vypis()`, jež uloží do souboru `data.dta` v aktuální adresáři čísla typu `int` od 0 do 9. Dále napíšeme metodu `PrecitajVypis()`, jež otevře soubor `data.dta` pro čtení, bude předstírat, že kolik položek v souboru je, všechny je přečte a vypíše na konzolu.

Metoda `Vypiš()` může vypadat takto:

```
public static void VypišČísla()
{
    FileStream fs =
        new FileStream("data.dta", FileMode.Create);
    BinaryWriter bw = new BinaryWriter(fs);
    for(int i = 0; i < 10; i++) bw.Write(i);
    bw.Close();
}
```

Při vytváření proudu jsme jako druhý parametr použili `FileMode.Create`, jenž předepisuje, že se soubor má vytvořit (a pokud již existuje, má se jeho obsah přepsat).

Metoda `PřečtiAVypiš()` je jen o málo složitější: V nekonečném cyklu budeme pomocí metody `ReadInt32()` číst obsah souboru. Cyklus skončí, když při čtení narazíme na konec souboru a vznikne výjimka, proto celý cyklus uzavřeme do bloku `try`.

Handler pro ošetření výjimky `EndOfStreamException` bude prázdný, neboť není důvodu při nalezení konce souboru podnikat nějaké zvláštní akce.

```
public static void PřečtiAVypiš()
{
    FileStream fs = new FileStream("data.dta", FileMode.Open);
    BinaryReader br = new BinaryReader(fs);
    try
    {
        for(;;) // Nekonečný cyklus
        {
            int i = br.ReadInt32(); // Vyvolá výjimku, když
            Console.WriteLine(i); // narazí na konec souboru
        }
    }
    catch(EndOfStreamException e) // Ošetření konce souboru
    { // neděláme nic
    }
}
```

Metoda `ReadInt32()`, stejně jako metoda `WriteInt32()`, může ještě vyvolat výjimku typu `IOException`, jež znamená to, že došlo k chybě při čtení nebo zápisu dat. Tyto výjimky ovšem nemůžeme v metodách `Vypiš()` a `PřečtiAVypiš()` ošetřit, to přenecháme metodě `Main()`:

```
static void Main(string[] args)
{
    try{
        VypišČísla();
        PřečtiAVypiš();
    }
}
```

⁶⁶ Zapomeneme-li na to, může se stát, že se část dat ztratí.

```
catch(IOException e)
{
    Console.WriteLine("Něco je špatně.");
}
}
```



Zdrojový text tohoto příkladu najdete na WWW v souboru `Kap12\02\semam.cs`. ✧

Ukládání objektů (serializace)

Ukládání hodnot primitivních typů je sice skvělé, ale při programování v C# dříve či později narazíme na nutnost ukládat do souborů či jinam celé objekty tak, abychom je mohli později načíst a obnovit v původním stavu, včetně vztahů k jiným objektům. Tomu se říká *perzistence* objektů (přetrvávají z jednoho běhu programu do jiného) a v C# se to řeší pomocí tzv. *serializace*. Problematika perzistence je poměrně komplikovaná, ovšem její implementace založená na serializaci je pro uživatele velice jednoduchá.

Třída pro ukládání objektů

K ukládání objektů v binární podobě do souborů a k jejich následnému čtení slouží třída `System.Runtime.Serialization.Formatters.Binary.BinaryFormatter`. Není to proud (i když je zakreslena na obr. 12.1). Tato třída poskytuje mj. metodu `Serialize(Stream proud, object co)`, jejímž prvním parametrem je proud, do něhož chceme data uložit, a druhým parametrem je ukládaný objekt.

Kromě toho najdeme ve třídě `BinaryFormatter` také metodu `object Deserialize(Stream proud)`, jejímž parametrem je proud, z něhož chceme objekt načíst. Tato metoda vrátí odkaz na objekt, a proto musíme vrácenou hodnotu před použitím přetypovat.

Třída, jejíž instance, chceme ukládat a číst pomocí mechanismu serializace, musí být deklarována s atributem `Serializable` a musí obsahovat hodnoty primitivních typů, odkazy na objekty, které jsou deklarovány s tímto atributem, nebo pole primitivních typů nebo objektů, které jsou deklarovány s tímto atributem – jinak vznikne při pokusu o čtení nebo zápis výjimka typu `System.Runtime.Serialization.SerializationException`. Většina knihovnických tříd tuto podmínku splňuje (mj. i třída `ArrayList` a další kontejnery).



Příklad

Vraťme se zase jednou k našim oblíbeným grafickým objektům. Mají-li být opravdu použitelné v grafickém editoru, musíme vyřešit i jejich

ukládání do souboru – a zde se serializace nabízí jako nejjednodušší cesta.⁶⁷ Zásahy do programu s grafickými objekty budou minimální.

- Třidu `GrafickýObjekt` a všechny třídy od ní odvozené deklarujeme s atributem `[Serializable]`.
- Deklarujeme třídu `Obrázek` představující kontejner, v němž budou jednotlivé grafické objekty uloženy. Ta bude obsahovat i metody `Ulož()` a `Načti()`.

To je vše. Podívejme se na odpovídající části zdrojového kódu. Nejprve třída `GrafickýObjekt`:

```
[Serializable]
public class GrafickýObjekt
{
    // Datové složky a metody stejné jako předtím
}
```



Atributy se nedědí, a proto musíme atribut `Serializable` připojit i ke třídám `Bod`, `Kružnice` atd.

Třída `Obrázek` obsahuje složku `obr` typu `ArrayList`, v níž jsou uloženy jednotlivé grafické objekty. Při zapisování do souboru stačí uložit tento kontejner. Metoda `Ulož()` může vypadat např. takto:

```
public void Ulož(string soubor)
{
    FileStream fs = File.Create(soubor);
    BinaryFormatter bf = new BinaryFormatter();
    bf.Serialize(fs, obr);
    fs.Close();
}
```

Zde nejprve vytvoříme proud typu `FileStream`. K tomu tentokrát použijeme metodu `Create()` třídy `File`. Pak vytvoříme `BinaryFormatter`, zavoláme jeho metodu `Serialize`, které předáme proud a ukládaný objekt, a ta se postará o vše potřebné. To znamená, že např. při ukládání `Kružnice` se uloží i `bod`, na nějž tato instance odkazuje (střed).

Postup při čtení je podobný:

```
public void Načti(string soubor)
{
    FileStream fs = File.Open(soubor, FileMode.Open);
    BinaryFormatter bf = new BinaryFormatter();
}
```

⁶⁷ Nikoli ovšem nejúspornější. Při serializaci se kromě obsahu objektů ukládají i další data, která umožní následné obnovení objektů a vztahů mezi nimi. Pokud by nám tedy šlo o velikost souboru s uloženým obrázkem, bylo by rozumnější nespolehat na serializaci a navrhnout vlastní vektorový formát.

```
obr = (ArrayList)bf.Deserialize(fs);
fs.Close();
}
```

Pro vytvoření proudu tentokrát použijeme metodu `Open()`, neboť chceme otevřít existující soubor. Metoda `Deserialize()` vrací objekt, proto musíme vrácenou hodnotu přetypovat na `ArrayList`. (Přitom ovšem může vzniknout výjimka typu `System.InvalidCastException`, kterou zde neošetřujeme.)

V konstruktoru třídy `Program` vytvoříme kontejner `Kresba` a v metodě `VytvořObrázek()` do něj přidáme několik grafických objektů:

```
public static void VytvořObrázek()
{
    Kresba.Přidej(new Bod(5,6,barvy.bílá));
    Kresba.Přidej(new Úsečka(1,2,3,4,barvy.modrá));
    Kresba.Přidej(new Kružnice(8, 9, 10, barvy.zelená));
    Kresba.Přidej(new ZakroužkovanýBod(6,9,barvy.červená));
}
```

Pak tuto kresbu uložíme příkazem

```
Kresba.Ulož("data.dta");
```

Později můžeme obrázek přečíst a nakreslit:

```
Kresba = new Obrázek();
Kresba.Načti(jméno);
Kresba.Nakresli();
```

Protože kreslit ještě neumíme, vypisuje metoda `Nakresli()` pouze zprávy na obrazovku. Ovšem i z tohoto výpisu zjistíme, že nová `Kresba` obsahuje tytéž objekty, jaké jsme předtím uložili.

Úplný zdrojový kód tohoto příkladu najdete na WWW v souboru `Kap12\03\serial.cs`. ♦



12.3 Čtení a zápis znakových dat

Za poněkud neobvyklým označením „znaková data“ se skrývají především tzv. textové soubory, tedy soubory, které obsahují znaky – znakovou reprezentaci nejrůznějších dat. Data v textových souborech jsou navíc rozdělena na řádky. Vedle souborů ale může jít třeba o síťová spojení, přenášející textová data (např. HTML).

Textové soubory (a znaková data vůbec) slouží převážně pro komunikaci mezi počítačem a člověkem. Práce s nimi je pro počítač obtížnější, neboť chceme-li do textového souboru uložit nějakou hodnotu, musí ji počítač nejprve převést na znakový řetězec a ten pak zapsat.

Pro čtení znakových dat používáme převážně proudy odvozené od třídy `System.IO.TextReader` a pro zápis proudy odvozené od třídy `System.IO.TextWriter`. Tyto třídy obsahují nástroje pro formátování dat. Jako podkladový proud pro ně používáme – stejně jako v případě binárních dat – třídy odvozené od `System.IO.Stream`.

Už jsme si řekli, že tyto třídy pracují se znaky. Mohli bychom tedy předpokládat, že data čtou a ukládají v kódování UNICODE. Ovšem naše zkušenosti s konzolovým výstupem tomu odporují: Data, vypisovaná metodou `Console.WriteLine()`, vystupují v kódové stránce 852 (středoevropské jazyky pro DOS).

Protože řada nástrojů pod Windows stále používá kódování založené na kódu ASCII (jednobajtovém), můžeme při vytváření proudu rozhodnout, jakého kódování chceme použít. Jak uvidíme později, vedle kódové stránky 852 máme k dispozici kódovou stránku 1250 (středoevropské jazyky pod Windows), kódování UNICODE, kódování UNICODE s velkým endiánem a některá další.

Zápis do textového souboru

Pro výstup znakových dat použijeme proud `StreamWriter`. Parametrem jeho konstruktoru je instance podkladového proudu; při výstupu do textového souboru to bude instance třídy `FileStream`. Konstruktorem proudu `StreamWriter` může mít vedle znakového řetězce ještě druhý parametr typu `Encoding`, který určuje kódování (kodovou stránku).

Všechny proudy odvozené od třídy `TextWriter`, a tedy i proud `StreamWriter`, obsahují mj. metody `Write()` a `WriteLine()` pro formátovaný zápis. Tyto dvě metody se liší jen tím, že po výpisu pomocí metody `Write()` nenásleduje přechod na nový řádek, kdežto po výpisu metodou `WriteLine()` následuje.

Příklad: zápis čísel do souboru

Napišeme velice jednoduchý program, který vytvoří textový soubor `data.txt` a zapíše do něj celá čísla od 10000 do 10009. Jméno souboru je deklarováno jako statická složka třídy:

```
static string jméno = "data.txt";
```

V metodě `Zapiš()` nejprve vytvoříme podkladový proud `fs` typu `FileStream`. Jako režim vytvoření zadáme `FileMode.Create`, neboť si přejeme, aby se vždy vytvořil nový soubor.

Proud `fs` pak předáme jako parametr konstruktoru třídy `StreamWriter`. Pro zápis použijeme známou metodu `WriteLine()`.

```
public static void Zapiš()
{
    FileStream fs = new FileStream(jméno, FileMode.Create);
```

```
StreamWriter sw = new StreamWriter(fs);
for(int i = 10000; i < 10010; i++)
    sw.WriteLine(i);
sw.Close();
}
```



Otevřete-li výsledný soubor např. v poznámkovém bloku pod Windows, přesvědčíte se, že se do něj zapsala všechna požadovaná čísla, na každý řádek jedno. Zdrojový text tohoto příkladu najdete na WWW v souboru `Kap12\04\Textovy.cs`. ✧

Připomeňme si, že při zápisu reálných čísel se použije – v souladu s nastavenými lokálními zvyklostmi – desetinná čárka. Podobně musí obsahovat desetinnou čárku (nikoli tečku) vstupující hodnoty.

Čtení z textového souboru

Pro čtení znakových dat použijeme proud `StreamReader`. Parametrem jeho konstruktoru je opět instance podkladového proudu; při čtení z textového souboru to bude instance třídy `FileStream`. Také konstruktorem proudu `StreamReader` může mít vedle znakového řetězce ještě druhý parametr typu `Encoding`, který určuje kódování.

Všechny proudy odvozené od třídy `TextReader`, a tedy i proud `StreamReader`, obsahují mj. metody `Read()` a `ReadLine()` pro čtení.

- Metoda `int Read()` bez parametrů přečte následující znak ze vstupního proudu a vrátí ho jako hodnotu typu `int`. Není-li žádný znak k dispozici, vrátí hodnotu `-1`.
- Metoda `int Read(char[] buffer, int index, int kolik)` přečte kolik znaků a uloží je do pole `buffer` počínaje indexem `index`.
- Metoda `string ReadLine()` bez parametrů přečte řádek a vrátí ho jako znakový řetězec. Není-li ve vstupním proudu k dispozici žádný znak, vrátí hodnotu `null`.

Pomocí těchto metod ze vstupního proudu získáme znakové řetězce, jejichž obsah si musíme analyzovat sami. K tomu můžeme využít už hotovou třídu `VV`, v níž stačí nahradit třídu `Console` obecným datovým proudem odvozeným od třídy `StreamReader`.

Příklad: čtení ze souboru

V předchozím příkladu jsme vytvořili soubor `data.txt` a zapsali jsme do něj jakási celá čísla. Nyní obsah tohoto souboru přečteme a vypíšeme na konzolu.

I tentokrát nejprve vytvoříme podkladový proud a ten předáme konstruktoru třídy `StreamReader`. Pak přečteme jednotlivé řádky, a protože víme, že každý řádek obsahuje jediné číslo, ihned je převedeme

číslo. K tomu použijeme tentokrát statickou metodu `Parse(string s)` třídy `Int32`, která umí totéž co `Convert.ToInt32`.

```
public static void Přečti()
{
    FileStream fs = new FileStream(jméno, FileMode.Open);
    StreamReader sr = new StreamReader(fs);
    string s = sr.ReadLine();
    while(s != null)                // Dokud je co číst...
    {
        int i = Int32.Parse(s);      // Převed' řetězec na číslo,
        Console.WriteLine(i);       // vypiš ho
        s = sr.ReadLine();          // a čti další
    }
}
```



Zdrojový text tohoto příkladu najdete spolu s předchozím na WWW v souboru `Kap12\04\Textovy.cs`. ✧



Chceme-li takto číst reálná čísla, použijeme např. metodu `Parse(string s)` třídy `System.Double`. Pracujeme-li pod českými Windows, musí tato čísla obsahovat desetinnou čárku, nikoli tečku.

Čeština a kódování

Zatím jsme vypisovali pouze čísla. Podívejme se nyní na výstup textu, samozřejmě českého. Upravme předchozí program tak, aby místo čísel vypsal zase jednu větu o žluťoučkém koni:

```
public static void Zapiš()
{
    FileStream fs = new FileStream(jméno, FileMode.Create);
    StreamWriter sw = new StreamWriter(fs);
    sw.WriteLine("Žluťoučký kůň příšerně úpěl ďábelské ódy");
    sw.Close();
}

public static void Přečti()
{
    FileStream fs = new FileStream(jméno, FileMode.Open);
    StreamReader sr = new StreamReader(fs);
    string s = sr.ReadLine();
    Console.WriteLine(s);
}
```

UTF-8 Věta přečtená ze souboru se správně vypíše na obrazovku, ale soubor nebude v některých jiných programech čitelný, neboť bude v kódování UTF-8. Toto kódování vychází z UNICODE, ale pro znaky bez diakritických znamének používá 1 bajt, zatímco pro znaky s diakritickými znaménky používá dva bajty.

Encoding

Už jsme si řekli, že konstruktory proudů `StreamWriter` a `StreamReader` mohou mít jako druhý parametr odkaz na instanci třídy `System.Text.Encoding`, vyjadřující požadované kódování. `Encoding` je abstraktní třída a v prostoru jmen `System.Text` jsou k dispozici její potomci:

- Třída `ASCIIEncoding` představuje sedmibitové kódování ASCII, které obsahuje pouze znaky bez diakritických znamének. Mimo anglicky hovořící země tedy nemá příliš použití.
- Třída `UnicodeEncoding` představuje kódování UNICODE, v němž jsou pro každý znak vyhrazeny 2 bajty. Toto kódování pokrývá prakticky všechny znakové sady. V konstruktoru této třídy lze předepsat i pořadí ukládání bajtů tvořících jeden znak (malý nebo velký endián⁶⁸).
- Třída `UTF8Encoding` představuje výše zmíněné kódování UTF-8.



Pro všechna tato kódování jsou k dispozici již připravené instance. Jsou definovány jako statické vlastnosti třídy `Encoding` pod názvy `ASCII`, `Unicode`, `BigEndianUnicode` a `UTF8`. Vedle toho je tu k dispozici také hodnota `Default`, jež představuje implicitní kódování (v českých Windows kódovou stránku 1250).

Příklad

Změníme metodu `Zapiš()` z předchozího příkladu, aby vytvořila soubor v kódové stránce 1250:

```
public static void Zapiš()
{
    FileStream fs = new FileStream(jméno, FileMode.Create);
    StreamWriter sw = new StreamWriter(fs, Encoding.Default);
    sw.WriteLine("Žluťoučký kůň příšerně úpěl ďábelské ódy");
    sw.Close();
}
```

Pokud bychom chtěli vytvořit soubor v kódové stránce 852, nezbudeme nám, než si odpovídající kódování „vypůjčit“ od objektu `Console`:

```
StreamWriter sw = new StreamWriter(fs,
    System.Console.Out.Encoding);
```



Podobným způsobem upravme i metodu pro čtení. Program, v němž můžete zvolit druh kódování a vypsat do souboru větu o žlutém koni najdete na WWW v souboru `Kap12\04\konina.cs`. ✧

⁶⁸ PC používají pro ukládání celých čísel a znaků malý endián, tj. méně významné bajty se ukládají na nižší adresu. To znamená, že bajty tvořící číslo `0x1234` budou v paměti uloženy v pořadí `0x34`, `0x12`. Jiné počítače mohou používat velký endián, tj. opačné pořadí ukládání bajtů. Velký endián používají vždy nezávisle na užitém počítači programovací jazyk Java.

Formátování

Až dosud jsme se nezabývali tvarem vystupujících údajů. Spokojili jsme se s tím, že jsme je převedli na nějaký znakový řetězec, a ten jsme vypsalí na obrazovku nebo do souboru. Čas od času se ale vyskytnou situace, kdy nám na formátu těchto čísel bude záležet. Podívejme se proto na některé z možností formátování. Podrobnější informace najdete v dokumentaci k rozhraní `System.IFormattable`.

Zatím víme, že v metodě `WriteLine()` třídy `TextWriter` můžeme zapísat libovolný znakový řetězec a v něm ve složených závorkách pořadové číslo parametru, který má na tomto místě vystoupit. Např. příkaz

```
s.WriteLine("faktoriál {0} je {1}", n, x);
```

způsobí, že se za slovem faktoriál vypíše hodnota `n` a za slovem je hodnota `x`. Takto formátované hodnoty vystoupí v implicitním tvaru – to znamená bez oddělovačů tisíců, v desítkové soustavě apod.

Specifikace	Použití	Příklad	výstup
N	Číslo s oddělovači tisíců	{0:N}	12 349,00
X	Šestnáctková soustava	{0:X}	303D
x	Šestnáctková soustava	{0:x}	303d
E	Semilogaritmický tvar	{0:E}	1,234900E+004
e	Semilogaritmický tvar	{0:e}	1,234900e+004
F	Pevná řádová čárka	{0:F}	12349,00
G	Obecný formát (podle okolností N nebo E)	{0:G}	12349
C	Měna	{0:C}	12 349,00 Kč
D	Desítkové číslo	{0:D7}	0012349

Tab. 12.1 Specifikace numerických formátů. Ve všech příkladech předpokládáme, že se formátuje hodnota 12349 typu `int`.

Specifikace
formátu

Za pořadové číslo parametru ve složených závorkách ovšem můžeme připsat specifikaci formátu; ta se od pořadového čísla parametru odděluje dvojtečkou. Zpravidla se skládá z jednoho písmene, za nímž může následovat číslo bez znaménka určující „přesnost“ (typicky počet cifer za desetinnou čárkou). Například specifikace `{0:E}` předepisuje, že reálné číslo má vystoupit v semilogaritmickém tvaru, specifikace `{0:E0}` navíc určuje, že za desetinnou čárkou nemají být žádné cifry. Možnosti formátování číselných hodnot ukazuje tabulka 12.1.

Vzor Pokud nám tyto formáty nestačí, můžeme specifikovat obecný vzor (picture), např. takto:

```
Console.WriteLine("{0:00000000.###}", 12349);
```

V této specifikaci znak „0“ předepisuje výstup číslice; pokud na daném místě žádná významná číslice není, vystoupí 0. Znak „#“ předepisuje také výstup číslice, avšak pokud na daném místě žádná významná číslice není, vystoupí mezera. Znak „.“ předepisuje výstup desetinné čárky. V tomto případě požadujeme, aby za desetinnou čárkou vystupovaly pouze významné číslice, zatímco před ní musí vystoupit vždy alespoň osm číslic. Výsledkem bude, že program vypíše 00012349.

Formát data

Pro práci s datem a časem je v BCL k dispozici struktura `System.DateTime`. Deklarujeme-li proměnnou

```
DateTime dt = new DateTime(2002, 5, 15, 11, 30, 21);
```

vypíše příkaz

```
Console.WriteLine("{0}", dt);
```

datum a čas ve tvaru

```
15.5.2002 11:30:21
```

Některé další možnosti ukazuje tabulka 12.2.

Specifikace	Význam	Příklad
D	den. měsíc (slovem) rok	15. květen 2002
T	hodina:minuta:sekunda	11:30:21
t	hodina:minuta	11:30
f	den. měsíc (slovem) rok hodina:minuta	15. květen 2002 11:30
g	den. měsíc. rok hodina:minuta	15.5.2002 11:30

Tab. 12.2 Některé z možností formátování data a času



Program, který ukazuje některé z možností formátování, najdete v WWW v souboru `kap12\05\format.cs`.

13 Grafické uživatelské rozhraní

Všechny naše dosavadní programy spadaly mezi tzv. konzolové aplikace. To znamená, že s uživatelem komunikovaly prostřednictvím příkazové řádky v dosovském okně Windows. V jazyce C# lze ovšem vytvářet stejně dobře aplikace pro internet, aplikace s grafickým uživatelským rozhraním („okenní“ programy), databázové aplikace atd. Prostředky pro to se skrývají v knihovně komponent BCL a v její nadstavbě v prostředí .NET Framework. V této kapitole si ukážeme základy programování aplikací s grafickým uživatelským rozhraním.

Pokud se rozhodnete používat C# ke skutečnému programování, budete tyto aplikace ve skutečnosti zpravidla vytvářet pomocí vizuálních vývojových nástrojů, jako je Visual C# .NET firmy Microsoft (ať už dodané samostatně nebo jako součást Visual Studio .NET). Ale i v takovém případě budete potřebovat znát alespoň základní principy výstavby těchto aplikací.

13.1 První okenní program

Nástroje pro tvorbu aplikací s grafickým uživatelským rozhraním jsou v převážné většině v prostoru jmen `System.Windows.Forms`. Pokud neuvědeme něco jiného, budou všechny třídy, které v tomto odstavení použijeme, z tohoto prostoru.

Základem okenních aplikací je třída `Application`, která obsahuje řadu statických metod, vlastností a událostí sloužících pro správu aplikace. Z nich asi nejdůležitější je pro nás přetížená metoda `Run()`. Nás bude nyní zajímat varianta s hlavičkou

```
public static void Run(System.Windows.Forms.Form f);
```

která očekává jako parametr odkaz `f` na instanci třídy `Form` (nebo jejího potomka) představující hlavní okno aplikace.

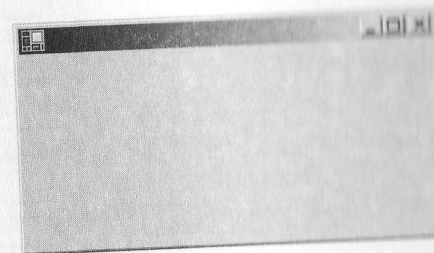
Příklad Nejjednodušší okenní program může vypadat takto:

```
using System.Windows.Forms;

public class Program
{
    public static void Main()
    {
        Form vokno = new Form();
```

```
Application.Run(vokno);
    }
```

V prvním řádku metody `Main()` vytvoříme instanci třídy `Form`, ve druhém zavoláme metodu `Run()` a tuto instanci jí předáme. Po překladu a spuštění se na obrazovce objeví prázdné okno (obr. 13.1). Toto okno není sice k ničemu, ale má veškeré základní vlastnosti okna pod Windows: Lze ho přemísťovat, měnit jeho velikost, minimalizovat nebo maximalizovat. Kliknutím na ikonu s křížkem ho lze uzavřít a tím program skončí.



Obr. 13.1 Okno našeho prvního okenního programu v C#

Ve skutečnosti je okno tohoto programu čtvercové. Na obr. 13.1 jsme ho na výšku poněkud zmenšili, neboť prázdné okno není příliš zajímavé.



Zdrojový text tohoto programu najdete na WWW v souboru `Kap13\01\vokno0.cs`. ♦

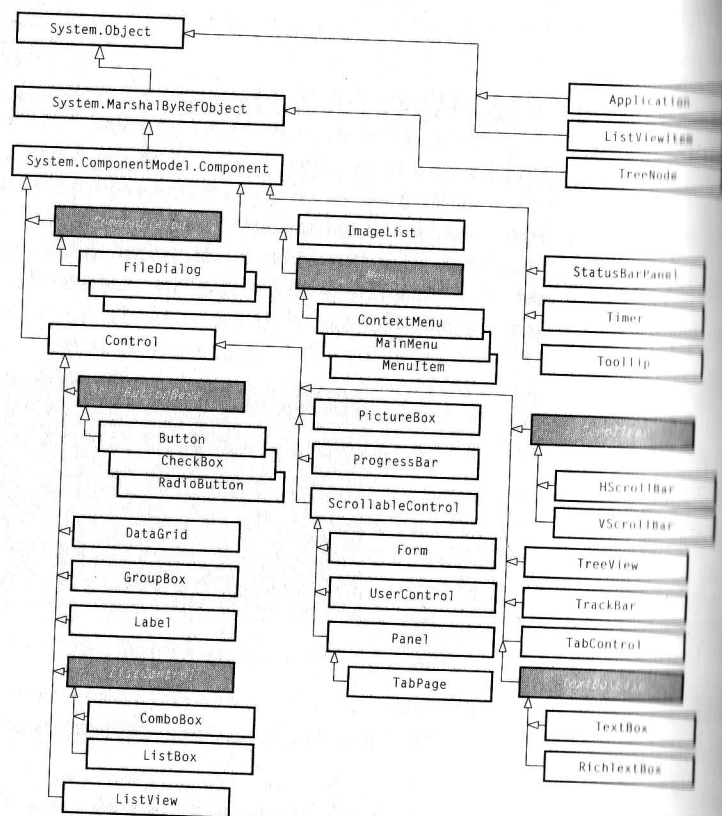
Doplňme, že kliknutí na ikonu s křížkem (na pravém okraji titulkové lišty) způsobí, že skončí metoda `Run()`. To má našem případě za následek, že skončí celý program.

Je asi jasné, že standardní třída okna nám sama o sobě k ničemu není, neboť ji nemůžeme měnit. Ve skutečných programech od ní vždy odvodíme potomka, svoji vlastní okenní třídu.

Knihovna Windows Forms

Knihovna Windows Forms obsahuje nástroje pro vytváření grafického uživatelského rozhraní. I když nás v tuto chvíli asi nejvíc zajímá třída `Form`, neuškodí, podíváme-li se alespoň povšechně na širší souvislosti.

Třídy pro vytváření grafického uživatelského rozhraní jsou, podobně jako všechny ostatní třídy v C#, odvozeny od třídy `object` (tj. `System.Object`). Výtah z jejich hierarchie ukazuje obr. 12.2.



Obr. 12.2 Převážná většina tříd pro tvorbu grafického uživatelského rozhraní. Šedé jsou abstraktní třídy. Kde není uveden prostor jmen, rozumí se *System.Windows.Forms*

Většina tříd vizuálních komponent je odvozena od třídy *System.Windows.Forms.Control*. Mezi jejími potomky najdeme třídu *ScrollableControl*, od níž jsou odvozeny třídy *Form* (okno) a *Panel*, jež slouží jako jakési okno v okně; umožňuje totiž seskupit další ovládací prvky grafického uživatelského rozhraní. Dále tu najdeme abstraktní třídu *ButtonBase*, od níž jsou odvozeny třídy *Button*, *CheckBox* a *RadioButton*, jež představují tlačítko, zaškrtnávací pole a přepínač.

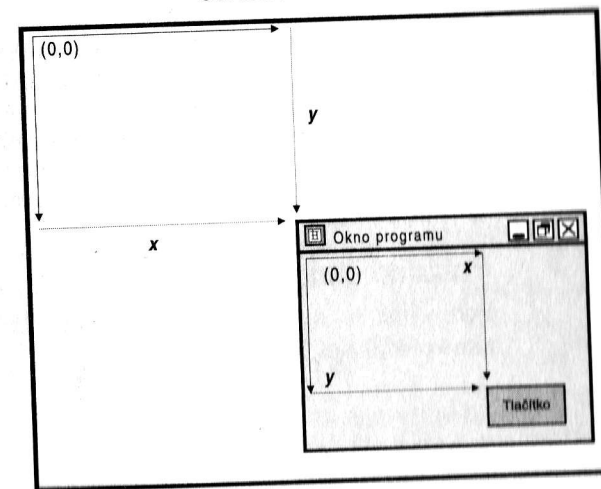
Od třídy *System.ComponentModel.Component* je odvozena třída *CommonDialog*, jejímiž bezprostředními potomky jsou dialogová okna pro otevření souboru, pro výběr barvy apod. (třídy *FileDialog*, *ColorDialog* a další). Jiným potomkem třídy *Component* je abstraktní třída *Menu*, od níž jsou odvozeny mj. třídy *MainMenu* (hlavní nabídka) a *MenuItem* (položka nabídky). Od třídy *Control* je také odvozena abstraktní třída *TextBoxBase*, jejímž potomkem je *TextBox* (vstupní pole).

Vlastnosti komponent

Velikost, umístění, způsob zobrazení a mnohé další charakteristiky komponent – tedy instancí tříd pro vytváření grafického uživatelského rozhraní programu – nastavujeme pomocí vlastností. Podívejme se alespoň na některé, jež se opakují u všech komponent odvozených od třídy *Control*.

- Visible** Tato vlastnost typu *bool* určuje, zda je komponenta viditelná.
- Enabled** Typu *bool*. Určuje, zda je komponenta aktivní, tj. zda může reagovat na vstup z klávesnice nebo na kliknutí myši.
- Text** Typu *string*. Obsahuje nápis na komponentě.
- Focused** Typu *bool*. Obsahuje hodnotu *true*, má-li komponenta fokus (může-li přijímat vstup z klávesnice).
- Size** Velikost komponenty. Zadáváme ji pomocí instance struktury *System.Drawing.Size*, což je uspořádaná dvojice celých čísel vyjadřující šířku a výšku komponenty v pixelech (bodech obrazovky).
- Bounds** Vyjadřuje velikost a pozici komponenty v okně. Je typu *System.Drawing.Rectangle*, což je uspořádaná čtveřice celých čísel vyjadřující po řadě velikost a umístění komponenty. Souřadnice v okně se počítají od levého horního rohu, kde je bod (0, 0). Souřadnice *x* roste zleva doprava, souřadnice *y* shora dolů.

Obrazovka monitoru



Obr. 12.3 Souřadnice na obrazovce a v okně jsou celočíselné a vyjadřují vzdálenost v pixelech

Left, Right	Tyto vlastnosti typu <code>int</code> vyjadřují souřadnici x levého, resp. pravého okraje komponenty v pixelech.
Top, Bottom	Tyto vlastnosti typu <code>int</code> vyjadřují souřadnici y horního, resp. dolního okraje komponenty v pixelech.
Location	Souřadnice levého horního rohu komponenty v pixelech. Je typu <code>System.Drawing.Point</code> , což je dvojice celých čísel vyjadřujících souřadnice (x, y) bodu.
Controls	Vlastnost pouze ke čtení. Představuje kontejner na komponenty, které jsou do dané komponenty vnořeny. Je typu <code>Control.ControlCollection</code> .

Třída okna

Okno obvykle obsahuje řadu vizuálních i nevizuálních komponent, jako jsou tlačítka, vstupní pole, nápisy apod. Může také obsahovat kontejner, do něhož se tyto komponenty uloží. Deklarujeme ho jako instanci třídy `System.ComponentModel.Container`, např.

```
private System.ComponentModel.Container components = null;
```

Dále bude třída okna obsahovat konstruktor, jehož úkolem bude postarat se o vytvoření a inicializaci komponent. Vzhledem k tomu, že půjde zpravidla o poměrně rozsáhlý úsek programu, vložíme tyto operace do zvláštní metody, kterou nazveme⁶⁹ `InitializeComponent()`.

Ve třídě okna musíme také překrýt chráněnou virtuální metodu

```
void Dispose(bool disposing),
```

jejímž úkolem je postarat se o uvolnění komponent při zániku okna. Musí zavolat metodu `Dispose` kontejneru `components` (pokud ho používáme) a nakonec také metodu `Dispose` předka.

Prázdné okno ještě jednou

Podívejme se nyní, jak může vypadat třída, představující prázdné okno o velikosti 300×400 pixelů.

```
using System.ComponentModel;
using System.Windows.Forms;

public class Vokno: Form
{
```

⁶⁹ V této kapitole budeme často používat jména, která metodám a datovým složkám dává standardně vývojové prostředí Visual C# .NET.

```
private Container components = null; // Můžeme vypustit
// Konstruktor
public Vokno()
{
    InitializeComponent();
}

protected override void Dispose(bool disposing)
{
    if(disposing)
    {
        if (components != null)
        {
            components.Dispose();
        }
        base.Dispose(disposing); // Toto je nezbytné
    }
}

private void InitializeComponent()
{
    // Nastavíme vlastnosti okna zděděné od třídy Form
    Size = new System.Drawing.Size(400, 300);
    Text = "Naše první okno";
}

}

class Program
{
    static void Main()
    {
        Application.Run(new Vokno());
    }
}
```

V metodě `InitializeComponent()` vytvoříme kontejner `components` a nastavíme velikost okna (vlastnost `Size`) a nápis v záhlaví okna (vlastnost `Text`). Po spuštění programu uvidíme prázdné okno, jež se bude od obr. 13.1 lišit jen tím, že bude mít v záhlaví titulek Naše první okno.



Zdrojový text tohoto programu najdete na WWW v souboru `Kap13\01\vokno1.cs`. ♦

13.2 Používáme komponenty

I když se výsledek našeho snažení na pohled neliší od prvního programu, je tu jeden podstatný rozdíl: Do třídy `Vokno` můžeme přidávat komponenty a používat je. Napišme tedy program, který vytvoří okno s jediným tlačítkem, po jehož stisknutí program skončí. I když to není nic převratného, ukážeme si na tom, jak se s komponentami pracuje.

Přidáváme komponentu do okna

Chceme-li do okna přidat komponentu, musíme ji nejprve vytvořit, a protože s ní budeme později chtít zacházet – nastavovat její vlastnosti ap. –, definujeme ji jako soukromou datovou složku třídy okna.

V metodě `InitializeComponent()` pak nastavíme její vlastnosti (nápis na ní, velikost, umístění v okně) a přidáme ji do kontejneru obsahujícího komponenty našeho okna. Tento kontejner je přístupný prostřednictvím vlastnosti `Controls` a k přidání použijeme metodu `Add()`,

```
using System.ComponentModel;
using System.Windows.Forms;

public class Vokno: Form
{
    private Button tlačítko = null; // Odkaz na tlačítko

    public Vokno() // Konstruktor
    {
        InitializeComponent();
    }

    protected override void Dispose(bool disposing)
    {
        base.Dispose(disposing);
    }

    private void InitializeComponent()
    {
        SuspendLayout();
        tlačítko = new Button();

        // Nastavení vlastností tlačítka
        tlačítko.Location =
            new System.Drawing.Point(118, 192);
        tlačítko.Size = new System.Drawing.Size(160, 50);
        tlačítko.Text = "Šmytec";

        // Nastavení vlastností okna
        ClientSize = new System.Drawing.Size(392, 273);
        StartPosition = FormStartPosition.CenterScreen;
        Text = "Okno s tlačítkem";

        // Přidání tlačítka do okna
        Controls.Add(tlačítko);

        ResumeLayout(false);
    }
}
```



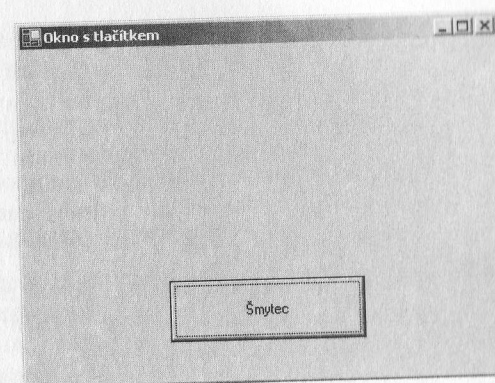
Všimněte si, že jsme po vstupu do metody `InitializeComponent()` volali metodu `SuspendLayout()` a před návratem jsme zavolali `ResumeLayout()`. Metoda `SuspendLayout()` zakáže vznik událostí, spoje-

ných s nastavováním vzhledu komponent (velikosti ap.) a metoda `ResumeLayout()` ho opět povolí.

Také nastavování vlastností okna je nyní trochu jiné než v předchozím příkladu. Vlastnost `ClientSize` vyjadřuje velikost *klientské oblasti* okna, tj. oblasti, do níž můžeme vkládat komponenty, kreslit ap. Klientská oblast okna nezahrnuje okraje a titulkovou lištu.

V předchozím příkladu se okno objevovalo na obrazovce ve víceméně náhodném místě. Tentokrát jsme předepsali, že se má zobrazit uprostřed obrazovky. K tomu nám posloužila vlastnost `StartPosition` a hodnota `FormStartPosition.CenterScreen`.

Výsledek naší práce ukazuje obr. 13.4.



Obr. 13.4 Výsledek naší usilovné práce: okno obsahuje tlačítko

Zatím ovšem má náš program jednu obrovskou vadu: stisknutí tlačítka nic nezpůsobí.

Předdefinované události

Z 10. kapitoly víme, že pro práci s událostmi používáme v C# delegáty a události. Třída `Button`, představující tlačítko, stejně jako většina tříd v knihovně `Windows Forms`, obsahuje řadu předdefinovaných událostí, takže se o deklarace odpovídajících typů delegátů, datových složek pro události a typů pro přenos informací o události nemusíme starat. Podívejme se alespoň na ty z nich, s nimiž se setkáme u většiny komponent:

Click

Nastane po stisknutí tlačítka myši na komponentě. Je definována pomocí delegátu typu `EventHandler`, druhý parametr handleru musí být typu `EventArgs`.

Enter	Nastane před tím, než komponenta dostane fokus (než může přijímat vstup z klávesnice). Je definována pomocí delegátu typu <code>EventHandler</code> , druhý parametr handleru musí být typu <code>EventArgs</code> .
GotFocus	Nastane poté, co komponenta dostane fokus. Tato událost je definována pomocí delegátu typu <code>EventHandler</code> , druhý parametr handleru musí být typu <code>EventArgs</code> .
Leave	Nastane před tím, než komponenta ztratí fokus (fokus přejde na jinou komponentu). Je definována pomocí delegátu typu <code>EventHandler</code> , druhý parametr handleru musí být typu <code>EventArgs</code> .
LostFocus	Nastane poté, co komponenta ztratí fokus. Je definována pomocí delegátu typu <code>EventHandler</code> , druhý parametr handleru musí být typu <code>EventArgs</code> .
Paint	Nastane, když je třeba překreslit ⁷⁰ obsah komponenty. (Většina komponent tuto událost ošetřuje automaticky a nemusíme se o to starat.) Je definována pomocí delegátu typu <code>PaintEventHandler</code> , druhý parametr handleru musí být typu <code>PaintEventArgs</code> .
Resize	Nastane, změní-li se velikost komponenty. Je definována pomocí delegátu typu <code>EventHandler</code> , druhý parametr handleru musí být typu <code>EventArgs</code> .
KeyDown	Nastane při stisknutí klávesy, pokud má komponenta fokus. Je definována pomocí delegátu typu <code>KeyEventHandler</code> , druhý parametr handleru musí být typu <code>EventArgs</code> .
KeyPress	Nastane při stisknutí klávesy, pokud má komponenta fokus (po události <code>KeyDown</code> a před <code>KeyUp</code>). Je definována pomocí delegátu typu <code>KeyPressEventHandler</code> , druhý parametr handleru musí být typu <code>EventArgs</code> .
KeyUp	Nastane při uvolnění klávesy, pokud má komponenta fokus. Je definována pomocí delegátu typu <code>KeyEventHandler</code> , druhý parametr handleru musí být typu <code>EventArgs</code> .
MouseEnter	Nastane, když kurzor myši vstoupí na plochu komponenty. Je definována pomocí delegátu typu <code>EventHandler</code> , druhý parametr handleru musí být typu <code>EventArgs</code> .

⁷⁰ V programech pro Windows platí, že každý program se stará o obsah svého okna a je povinen ho kdykoli umět znovu nakreslit. Jestliže uživatel počítače odkryje okno programu, jež bylo zakryto jiným oknem, jestliže změní rozměry okna, jestliže okno minimalizuje a pak ho obnoví a v mnoha jiných situacích se stane obsah okna (nebo komponenty) neplatný a operační systém vyzve program, aby ho obnovil. V programech v C# bude výsledkem události `Paint`.

MouseMove	Nastane po události <code>MouseEnter</code> , když se kurzor myši pohybuje po ploše komponenty. Je definována pomocí delegátu typu <code>EventHandler</code> , druhý parametr handleru musí být typu <code>EventArgs</code> .
MouseDown	Nastane, je-li kurzor myši na ploše komponenty a uživatel stiskne tlačítko myši. Druhý parametr handleru musí být typu <code>EventArgs</code> .
MouseUp	Nastane, je-li kurzor myši na ploše komponenty a uživatel uvolní tlačítko myši. Je definována pomocí delegátu typu <code>EventHandler</code> , druhý parametr handleru musí být typu <code>EventArgs</code> .
MouseLeave	Nastane, když kurzor myši opustí na plochu komponenty. Je definována pomocí delegátu typu <code>EventHandler</code> , druhý parametr handleru musí být typu <code>EventArgs</code> .

Ze složek druhého parametru lze získat dodatečné informace. V případě událostí týkajících se myši to jsou tlačítka, jež byla stisknuta, souřadnice bodu, kde se nacházel kurzor, a další. V případě událostí, jež se týkají klávesnice, je to stisknutá klávesa, stisknuté přepínače atd. Podrobnější informace lze získat v dokumentaci.

Učíme tlačítko fungovat

Nyní už nebude problém dokončit náš program. Použijeme událost `Click`, kterou třída `Button` již obsahuje. Protože jde o předdefinovanou událost, nemusíme se starat o deklaraci delegátu, události ani třídy parametrů. Vše, co potřebujeme, je deklarovat handler (jako metodu třídy okna) a zaregistrovat ho u instance tlačítka. Deklarace handleru je jednoduchá:

```
private void tlačítko_Click(object sender, EventArgs e)
{
    Close();
}
```

Metoda `Close()` třídy `Form` uzavře okno našeho programu a tím skončí metoda `Application.Run()` a v důsledku toho i celý program.⁷¹

Registraci handleru v metodě `InitializeComponent()` obstará příkaz `tlačítko.Click += new System.EventHandler(tlačítko_Click);`



Tím jsme hotovi; okno programu bude vypadat stejně jako na obr. 13.4, po stisknutí tlačítka však program skončí. Úplný zdrojový text najdete na WWW v souboru `Kap13\02\vokno3.cs`. ✧

⁷¹ V programech pro Windows je obvyklé, že uzavření hlavního okna aplikace způsobí ukončení celého programu.

13.3 Příklad: problém N dam

Na závěr této kapitoly se podíváme na složitější příklad, na němž se seznámíme s několika komponentami a ukážeme si vytvoření programu s grafickým uživatelským rozhraním. Naším úkolem bude napsat program, který bude hledat a zobrazovat řešení problému osmi dam.

Jde o klasickou kombinatorickou úlohu, kterou formulovat někdy před rokem 1850 známý německý matematik Karl Friedrich Gauss. Chceme rozmístit na běžné šachovnici osm dam tak, aby se žádné dvě navzájem neohrožovaly. (To znamená, že libovolné dvě dámy jsou navzájem nepřátelské. Nejde tedy o šachovou úlohu, spíše o problém ze skutečného života ;)

My si vezmeme obecnější úlohu: použijeme šachovnici o $N \times N$ polích a pokusíme se na ni rozmístit N dam. Náš program by měl umět najít a zobrazit všechna řešení.

Pro $N = 1$ je naše úloha triviální, neboť jednu jedinou dámu na šachovnici o 1×1 polí lze umístit jedním způsobem a nikdo ji nebude ohrožovat.

Pro $N = 2$ a $N = 3$ tato úloha nemá řešení. O tom se snadno přesvědčíte, proberete-li všechna možná postavení dam na šachovnicích o velikosti 2×2 a 3×3 .

Pro $N = 4$ má tato úloha dvě řešení. Pro $N > 4$ má vždy řešení a jejich počet s rostoucím rozměrem šachovnice N velmi rychle roste. Pro běžnou šachovnici, tedy pro $N = 8$, má 92 řešení (existuje 92 různých rozmístění dam, která vyhovují zadání), pro $N = 13$ už existuje více než 70 000 řešení.

Jak najít řešení

Než se pustíme do programování, měli bychom si ujasnit, zda (a jak) dokážeme najít řešení problému N dam. Přitom se omezíme na $N \geq 4$.

Hrubá síla

První možnost, která nás napadne, bude nejspíš použití hrubé síly: Probrat všechny možnosti rozestavení N dam na šachovnici o velikosti $N \times N$ a vyloučit ta, jež nevyhovují zadání. Ovšem například 8 dam lze na šachovnici rozmístit více než $4,4 \cdot 10^9$ způsoby, z nichž jen 92 je správných. Metoda hrubé síly tedy nejspíš nebude tou nejlepší možností.

Brzy ale zjistíme, že počet probíraných možností můžeme výrazně omezit. Šachová dáma ohrožuje jinou figuru jen v případě, že obě stojí ve stejném sloupci, ve stejné řádce nebo na stejné diagonále.

To mj. znamená, v každém sloupci šachovnice může být nejvýše jedna dáma. Odtud plyne, že chceme-li na šachovnici o velikosti $N \times N$ umístit N dam, musí být v každém sloupci *právě jedna* dáma. Podobně můžeme uvažovat i o řádkách a diagonálách šachovnice. Tyto úvahy nás přivedou k následujícímu postupu:

Zápis řešení

■ Řešení úlohy N dam lze vyjádřit jako N -tici čísel $(a_1, a_2, \dots, a_N)$, kde a_1 vyjadřuje polohu dámy v prvním sloupci šachovnice, a_2 polohu ve druhém sloupci atd. Je-li např. $a_1 = 2$, znamená to, že dáma je v prvním sloupci ve druhém řádku.

Postupné umísťování

■ Nejprve umístíme dámu do prvního sloupce. Začneme pozici v prvním řádku.

■ Pak zkusíme umístit další dámu do druhého sloupce. Začneme pozici v prvním řádku. Ihned zkontrolujeme, zda tato pozice odpovídá pravidlům, tj. zda ji tam neohrožuje některá z dříve umístěných dam.

■ Pokud nově umístěnou dámu žádná z předchozích neohrožuje, a tedy pokud nově umístěná dáma neohrožuje žádnou z dříve umístěných dam, ponecháme ji tam a zkusíme umístit dámu do následujícího sloupce. V opačném případě zkusíme dámu umístit na následující řádku v témže sloupci.

■ Nepodaří-li se nám dámu v daném sloupci umístit, vrátíme se k předchozí dámě a zkusíme pro ni najít nové umístění. Pokud se nám nepodaří umístit ani tu, vrátíme se ještě o sloupec zpět atd.

■ Umístíme-li úspěšně dámu v N -tém sloupci, našli jsme řešení. Zapišeme je a pokusíme se najít další pozici pro dámu v posledním sloupci (tedy najít další řešení).

■ Není-li možné najít další umístění pro první dámu, skončíme, neboť další řešení již neexistují.

Backtracking

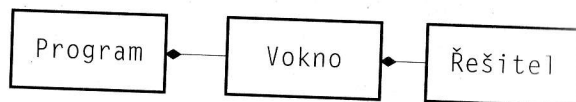
Tomuto postupu se říká metoda prohledávání s návratem (často se používá anglický termín *backtracking*). Ve srovnání s metodou hrubé síly výrazně snižuje počet zkoumaných možností. Jestliže totiž zjistíme, že žádné řešení nemůže začínat např. $(1, 2, \dots)$, protože takto umístěné dámy stojí na stejné diagonále, vyloučíme všechny pozice, které takto začínají, z dalšího zkoumání.

Program

Když už víme, jak najít řešení, ujasníme si, jak bude náš program vypadat. Jako obvykle definujeme třídu `Program`, která bude obsahovat metodu `Main()`.

O komunikaci s uživatelem programu se bude starat třída `Vokno`, jež bude obsahovat hlavní okno naší aplikace. Bude obsahovat nabídku řídicí prvky pro zadání rozměru šachovnice, pro nalezení dalšího řešení a pro jeho zobrazení.

Třída okna bude využívat služeb třídy `Řešitel`, jež bude mít na starosti nalezení jednotlivých řešení úlohy. Vztah těchto tříd ukazuje obr. 13.5. (Připomeňme si, že šipka zakončená černým kosočtvercem vyjadřuje kompozici – tj. třída `Program` využívá služeb třídy `Vokno` a ta využívá služeb třídy `Řešitel`.)



Obr. 13.5 Vztah základních tříd v programu

Třída `Program` je natolik jednoduchá, že ji jistě zvládnete napsat sami. Třída `Vokno` bude naopak pro nás jako začátečníky v oblasti grafického rozhraní natolik komplikovaná, že ji prozatím ponecháme stranou a začneme třídou `Řešitel`.

Třída Řešitel

Z našich předchozích úvah plyne, že tato třída bude jako soukromé datové složky obsahovat

- hodnotu `n` typu `int` vyjadřující rozměr šachovnice a počet umísťovaných dam (N),
- pole řešení typu `int`, obsahující řešení nebo jeho dosud nalezenou část.

Datové složky

K nim přidáme ještě proměnnou počet vyjadřující počet dosud nalezených řešení. Deklarace třídy `Řešitel` tedy bude začínat

```
using System;

class Řešitel
{
    private readonly int n;      // Počet dam
    private int[] řešení;       // Řešení úlohy (viz výše)
    int počet = 0;              // Počet nalezených řešení
}
```

Přitom budeme muset mít na paměti, že pole v C# jsou indexována od 0, ale políčka šachovnice od 1. Pole řešení vyplníme v konstruktoru hodnotou `-1`. Hodnotu proměnné `n`, tj. počet dam, zadáme jako parametr konstruktoru, a proto jsme tuto proměnnou deklarovali s modifikátorem `readonly`.

`Bezpečné()`

Dále budeme potřebovat metodu `Bezpečné()`, která určí, zda je daná pozice „bezpečná“, tj. zda na ni lze dámu umístit, aniž by ji ohrožovala některá z těch, které na šachovnici již jsou. Vzhledem k tomu, že budeme zkoumat vždy bezpečnost nově umísťované dámy, stačí zjistit, zda ve stejném řádku nebo na stejných diagonálách nestojí jiná dáma v některém z předchozích sloupců. Funkce `Bezpečné()` může mít tvar

```
// Je pozice v k-tém řádku, m-tém sloupci bezpečná?
private bool Bezpečné(int k, int m)
{
    // V prvním sloupci lze dámu umístit kamkoli
    if(m == 0) return true;
    bool výsledek = true;
    int j = 0;
    // Projdi v cyklu všechny předchozí sloupce
    while(výsledek && (j < m))
    {
        výsledek = (řešení[j] != k) && // stejný řádek?
                   (řešení[m-j-1] != k-j-1) && // stejná diagonála?
                   (řešení[m-j-1] != k+j+1);
        if(!výsledek) return false;
        j++;
    }
    return true;
}
```

Dále budeme potřebovat pomocnou metodu `NajdiDalšíK`, která najde v daném sloupci další polohu vyhovující pravidlům:

```
// Najde další možné umístění v m-tém sloupci,
// začne za k-tým řádkem
private int NajdiDalšíK(int k, int m)
{
    while(++k < n) if(Bezpečné(k, m)) break;
    return k; // Neuspěje-li, vrátí n
}
```

Tato metoda očekává jako vstupní parametr předchozí pozici dámy v daném sloupci, proto nejprve zvýší `k` o 1. Pak testuje další pozice, až najde bezpečnou nebo až vyčerpá všechny možnosti. Pokud neuspěje, tj. pokud v daném sloupci žádná další bezpečná pozice neexistuje, vrátí `n`. (Připomeňme si, že řádky jsou číslovány od 0 do `n-1`, neboť je vyjadřujeme pomocí indexů pole.)

Nyní už můžeme napsat metodu `Řeš()`, která najde jedno řešení a vrátí odkaz na pole s ním; pokud řešení nenajde, vrátí `null`.

Této metodě předáme jako parametr index sloupce, od kterého má při hledání začít. Na počátku to bude 0, neboť musíme umístit nejprve první dámu, pak druhou atd. Při hledání druhého – a všech dalších řešení – ji ovšem předáme jako parametr hodnotu `m-1`, neboť musíme

začít tím, že se pokusíme najít nové umístění pro poslední dámu. Pokud to nepůjde, poslední dámu odstraníme (do posledního sloupce zapíšeme -1 a snížíme hodnotu m o 1) a pokusíme se přemístit před poslední dámu atd.

// Začne hledat další řešení, a to počínaje m -tým sloupcem

```
private int[] Řeš(int m)
{
    int k;
    while(m > -1)
    {
        k = NajdiDalšíK(řešení[m], m);
        if(k < n)
        {
            řešení[m++] = k;
            if(m == n) return řešení;
        }
        else řešení[m--] = -1;
    }
    return null;
}
```

Jádrem této metody je cyklus `while` s podmínkou $m > -1$. Tato situace nastane, když už žádné další řešení neexistuje, takže se nepovedlo umístit první dámu.

V cyklu se vždy pokusíme najít další umístění dámy v m -tém sloupci. Pokud uspějeme, vrátí metoda `NajdiDalšíK()` hodnotu menší než n . V tom případě nalezenou pozici zapíšeme do složky `řešení[m]` a přejdeme na další sloupec (příkaz `m++`). Pokud jsme umístili dámu v posledním sloupci (po zvýšení hodnoty m platí $m == n$), našli jsme řešení, vrátíme tedy odkaz na ně, jinak cyklus pokračuje hledáním umístění v dalším sloupci.

Jestliže se v daném sloupci nepodařilo najít vhodnou pozici, vrátí metoda `NajdiDalšíK()` hodnotu n . V takovém případě se musíme vrátit o jeden sloupec zpátky. Proto uložíme do `řešení[m]` hodnotu -1 a zmenšíme m o 1. Pokud cyklus skončí, protože je $m == -1$, znamená to, že další řešení neexistuje, a proto vrátíme `null`.

Metodu `Řeš()` je třeba při hledání prvního řešení volat s parametrem 0 a při hledání dalších řešení vždy s parametrem $n-1$. To je nepohodlné a pro případného uživatele takového třídy, který nezná postup implementace, i matoucí. Proto jsme ji deklarovali jako soukromou a napíšeme ještě metodu `NajdiDalšíŘešení()`, jež tyto problémy odstraní a zároveň spočítá, kolik řešení bylo nalezeno:

```
public bool NajdiDalšíŘešení(out int[] výsledek)
{
    if(počet==0) výsledek = Řeš(0);
    else výsledek = Řeš(n-1);
}
```

```
if(výsledek != null) počet++;
return výsledek != null;
}
```

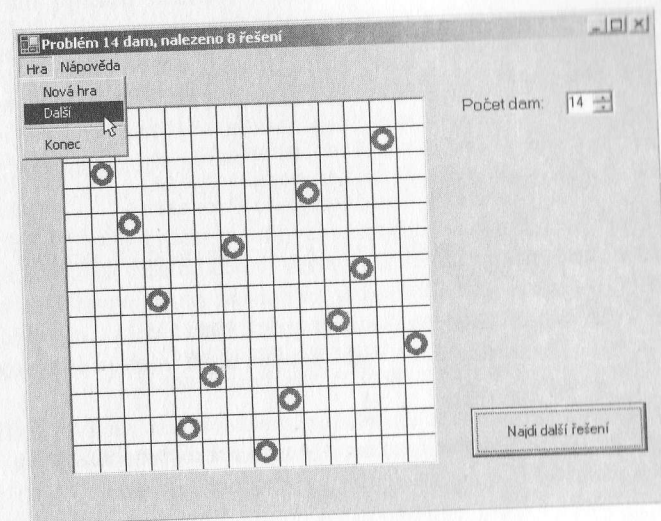
Tato metoda vrací hodnotu typu `bool` signalizující, zda se podařilo další řešení najít, a jako výstupní parametr také odkaz na pole obsahující řešení.



V souboru `Kap13\03\damy.cs`, který můžete najít na WWW, je vedle uvedených metod také rekurzivní metoda `NajdiŘešení()`, jež implementuje týž algoritmus rekurzivně, najde všechna řešení najednou. Také tato metoda je soukromá a voláme ji prostřednictvím metody veřejně přístupné metody `NajdiVšechnaŘešení()`.

Třída okna

Nyní naprogramujeme okno tvořící grafické uživatelské rozhraní našeho programu; bude vypadat jako okno na obr. 13.6. Vedle nabídky bude obsahovat komponentu pro zadávání počtu dam a tlačítko, po jehož stisknutí program vyhledá a zobrazí další řešení. Nabídka bude obsahovat dvě položky, Hra a Nápověda.



Obr. 13.6 Okno naší budoucí aplikace

Nabídka Hra bude mít příkazy Nová hra, po němž si program přečte z řídícího prvku požadovaný počet dam, Další, jenž bude mít stejný význam jako tlačítko Najdi další řešení, a Konec, jež ukončí program. Nabídka Nápověda bude obsahovat pouze položku 0 programu, který vyvolá stejnojmenné dialogové okno.

Třidu okna nazveme správně česky Vokno; jako datové složky bude obsahovat odkazy na komponenty a kromě toho pomocné proměnné

```
private int kolik = 4;           // Počet dam
private Řešitel ř = null;       // Instance řešitele
int[] řešení = null;           // Pole s řešením
bool nalezeno = false;         // Našlo se řešení?
const int VELIKOST_PANELU = 300; // pixelů
```

Nabídka Jako první vytvoříme nabídku; jejím základem je komponenta MainMenu. Tato komponenta je nevizuální, funguje pouze jako kontejner obsahující jednotlivé položky.

Její jednotlivé položky jsou instance třídy MenuItem. Nápisů na položkách nabídky jsou dány vlastností Text. Také separátor (vodorovná čára oddělující skupinu položek nabídky) je instance třídy MenuItem, ovšem s vlastností Text obsahující znak minus.

To znamená, že budeme potřebovat jednu instanci třídy MainMenu a sedm instancí třídy MenuItem (včetně separátoru mezi příkazy Další a Konec, viz obr. 13.6). Nejprve deklarujeme ve třídě okno odkazy na ně:

```
private MainMenu nabídka;       // Nabídka jako celek
private MenuItem menuHra;       // Hra
private MenuItem menuNová;      // Hra| Nová hra
private MenuItem menuDalší;     // Hra| Další
private MenuItem menuSeparátor; // Separátor
private MenuItem menuKonec;     // Hra| Konec
private MenuItem menuNápověda;  // Nápověda
private MenuItem menuOProgramu; // Nápověda | 0 programu
```

V metodě InitializeComponent() nejprve potřebné komponenty vytvoříme,

```
this.nabídka = new MainMenu();
this.menuHra = new MenuItem();
this.menuNová = new MenuItem();
// atd...
```

a pak určíme nápisy na položkách nabídky:

```
this.menuHra.Text = "&Hra";
this.menuNápověda.Text = "&Nápověda";
// atd...
```

Klávesová zkratka

Znak & v textu nabídky způsobí, že následující znak se zobrazí podtržený a bude pro odpovídající příkaz sloužit jako klávesová zkratka (v kombinaci s klávesou Alt). To znamená, že např. klávesovou zkratkou Alt+H způsobíme vybrání položky Hra. Podobně definujeme klávesové zkratky i pro jiné řídicí prvky.

Propojení nabídek

Dalším krokem je propojení nabídek tak, aby např. položka Hra obsahovala položky Nová, Další a Konec. K tomu využijeme skutečnost, že instance třídy MenuItem má vlastnost MenuItems, jež představuje položky podřízené nabídce. Tato vlastnost je typu System.Windows.Forms.Menu.MenuItemCollection a má metodu AddRange(MenuItem[]) pro přidávání podřízených položek. Tutéž vlastnost má a tedy i tutéž metodu – obsahuje i třída MainMenu. K seskupení položek nám proto poslouží příkazy

```
this.nabídka.MenuItems.AddRange(new MenuItem[] {
    this.menuHra, this.menuNápověda });
this.menuHra.MenuItems.AddRange(new MenuItem[] {
    this.menuNová, this.menuDalší,
    this.menuSeparátor, this.menuKonec });
this.menuNápověda.MenuItems.AddRange(new MenuItem[] {
    this.menuOProgramu });
```

Připojení k oknu

Nakonec připojíme nabídku k oknu; teprve tím způsobíme, že se nabídka při otevření okna zobrazí. K tomu nám poslouží vlastnost Menu, zděděná po třídě Frame:

```
this.Menu = this.nabídka
```

Vybrání příkazu nabídky ovšem zatím nic nezpůsobí, neboť jsme nedefinovali a nezaregistrovali odpovídající handlers. To ale už známe: Např. pro položku Konec definujeme handler jménem menuKonec_Click(object sender, EventArgs e) a zaregistrujeme ho příkazem

```
this.menuKonec.Click +=
    new EventHandler(this.menuKonec_Click);
```

Enabled

Některé příkazy ovšem v určitých situacích nemají smysl; např. požadovat nalezení dalšího řešení dříve, než zadáno počet dam, může způsobit problémy. Proto nastavíme u komponenty menuDalší vlastnost Enabled na false. Tato komponenta pak bude „zašedlá“ a nebude reagovat na vybrání myši nebo příkazy z klávesnice.

Tlačítko

Dalším prvkem uživatelského rozhraní je tlačítko, jež způsobí nalezení dalšího řešení. Představuje alternativu k položce nabídky Hra|Další, jeho stisknutím způsobíme nalezení a zobrazení dalšího řešení.

Tlačítko deklarujeme jako soukromou datovou složku,

```
private Button dalšíŘešení;
```

vytvoříme ho v metodě InitializeComponent() volané z konstruktoru, kde zároveň určíme jeho velikost, polohu a nápis na něm,

```
this.dalšíŘešení = new Button();
this.dalšíŘešení.Location = new Point(360, 280);
```

```
this.dalšíŘešení.Size = new Size(144, 40);
this.dalšíŘešení.Text = "Najdi &další řešení";
```

(znak & opět znamená klávesovou zkratku), a zaregistrujeme pro něj stejný handler jako pro položku nabídky menuDalší:

```
dalšíŘešení.Click += new EventHandler(this.menuDalší_Click);
```

Také tlačítko bude mít na počátku nastavenou vlastnost Enabled na false.

Číselník

Ke stanovení počtu dam a velikosti šachovnice použijeme komponentu číselník (typu NumericUpDown),

```
private NumericUpDown početDam;
```

Tato komponenta obsahuje textové pole, jež zobrazuje číslo, a tlačítka se šipkami nahoru a dolů, jež umožňují zobrazené číslo měnit. Hodnota zobrazeného čísla je dána vlastností Value, maximální a minimální hodnota vlastnostmi Maximum, resp. Minimum. Všechny tři tyto vlastnosti jsou typu decimal. Vytvoření, umístění a počáteční nastavení této komponenty v metodě InitializeComponent() může vypadat takto:

```
this.pocetDam = new NumericUpDown();
this.pocetDam.Location = new Point(448, 24);
this.pocetDam.Size = new Size(40, 20);
this.pocetDam.Maximum = new Decimal(20);
this.pocetDam.Minimum = new Decimal(4);
this.pocetDam.Value = new Decimal(4);
```

Minimální počet dam (a zároveň počáteční hodnotu) jsme nastavili na 4, neboť pro menší hodnotu úloha nemá význam. Maximální počet jsme nastavili na 20. Změna zobrazené hodnoty této komponenty vyvolá událost ValueChanged; my ji však nevyužijeme.

Text u číselníku vytvoříme pomocí komponenty Label.

Panel

Nalezená řešení budeme zobrazovat v okně. Jako kreslicí plochu použijeme komponentu Panel

```
private Panel panel;
```

V metodě InitializeComponent() ho vytvoříme a určíme jeho vlastnosti:

```
this.panel = new Panel();
this.panel.BackColor = Color.White;
this.panel.BorderStyle = BorderStyle.Fixed3D;
this.panel.Location = new Point(32, 24);
this.panel.Size = new Size(VELIKOST_PANELU, VELIKOST_PANELU);
```

Zde nejprve určíme barvu pozadí, pak nastavíme styl okrajů, umístění a velikost. Poznamenejme, že výčtové typy Color a BorderStyle jsou definovány v prostoru jmen System.Drawing, stejně jako struktury Point a Size.

Nakreslení výsledku na plochu panelu obstará metoda panel.Paint(), kterou zaregistrujeme jako handler pro událost Paint:

```
this.panel.Paint += new
    System.Windows.Forms.PaintEventHandler(this.panel_Paint);
```

K vlastnímu kreslení se vrátíme později.

Vložení komponent

Nakonec vložíme všechny komponenty do okna příkazem

```
this.Controls.AddRange(new Control[] {this.panel,
    this.NápisKolikDam, this.pocetDam, this.dalšíŘešení});
```

Dialogová okna se zprávami

Vzhled našeho programu je již v podstatě hotov, můžeme se začít věnovat jeho chování. Než se pustíme do programování handlerů, ujasníme si, jak bude komunikovat s uživatelem.

Jestliže např. program nenajde další řešení, měl by to oznámit uživateli. Podobně před ukončením – ať už pomocí příkazu Hra|Konec nebo pomocí tlačítka s křížkem na titulkové liště – by se program měl zeptat, zda uživatel chce opravdu skončit.

K takovýmto dotazům a oznámením se používají předdefinovaná dialogová okna, anglicky označovaná Message Box. Ukážeme si, jak je lze používat v C#. Máme dvě možnosti a obě si ukážeme:

- Tato okna jsou součástí Windows API (knihovny user32.dll) a v programech přímo využívajících toto rozhraní lze je volat pomocí funkce MessageBox(). Tuto možnost poskytuje i C#.
- Můžeme použít statickou metodu Show() třídy MessageBox, kterou najdeme v knihovně BCL.

Volání API

Chceme-li volat funkci z Windows API, musíme ji deklarovat jako statickou externí metodu nějaké třídy a v parametru atributuDllImport uvést jméno dynamické knihovny, v níž je definována. To znamená, že ve třídě okno deklaruje metodu

```
[DllImport("user32.dll")]
private static extern int MessageBoxA(int okno,
    string nápis, string titulěk, int typ);
```

Deklarace této metody neobsahuje tělo a je ukončena středníkem. Její první parametr obsahuje identifikační číslo (handle) okna, z něhož je toto dialogové okno voláno; většinou tam uvedeme hodnotu 0. Druhý

parametr obsahuje nápis, který chceme zobrazit, a třetí pak titulek tohoto okna. Poslední parametr obsahuje číslo, jež určuje počet a typ tlačítek, která se v okně mají zobrazit, a případně některý z předdefinovaných obrázků. Zde můžeme použít výčtových typů `MessageBoxButtons` a `MessageBoxIcons`, jež jsou oba definovány v prostoru jmen `System.Windows.Forms`. Tato funkce vrací hodnotu určující tlačítko, které uživatel stiskl; k porovnání můžeme použít výčtový typ `DialogResult`, který je definován ve stejném prostoru jmen. Podívejme se na některé z hodnot těchto typů:

MessageBox-Buttons	OK	Dialogové okno obsahuje tlačítko OK.
	OKCancel	Dialogové okno obsahuje tlačítka OK a Storno.
	YesNo	Dialogové okno obsahuje tlačítka Ano a Ne.
	YesNoCancel	Dialogové okno obsahuje tlačítka Ano, Ne a Storno.
MessageBox-Icons	Exclamation	Ikona s vykřičníkem (znamená obvykle upozornění)
	Information	Písmeno i v kroužku (informace)
	Question	Otazník (program potřebuje znát rozhodnutí uživatele)
	Stop	Znak X v kruhu s rudým pozadím (chyba)
DialogResult	OK	Uživatel stiskl tlačítko OK.
	Yes	Uživatel stiskl tlačítko Ano.
	No	Uživatel stiskl tlačítko Ne.
	Cancel	Uživatel stiskl tlačítko Storno.

Metoda `Show()` třídy `MessageBox` existuje v několika přetížených verzích. Pro nás bude nejzajímavější

```
DialogResult Show(string text, string titulek,
    MessageBoxButtons tlačítka, MessageBoxIcon ikona)
```

První, resp. druhý parametr představuje text, resp. titulek okna, třetí určuje tlačítka a čtvrtý ikonu. (Čtvrtý parametr lze vynechat.) Příklady použití obou způsobů vytvoření těchto dialogových oken uvidíme v následujícím oddílu.

Handlery

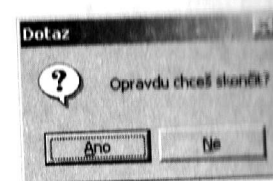
Ukončení programu Dobře napsaný program před ukončením zkontroluje, zda uživatel uložil soubory, v nichž došlo ke změnám, a pokud ne, zeptá se, zda si je přeje uložit. Náš program sice se soubory nepracuje, ale přesto se uživatele zeptá, zda smí skončit.

První řešení Handler pro událost `Click` komponenty `menuKonec` proto může mít tvar

```
private void menuKonec_Click(object sender, System.EventArgs e)
```

```
{
    if(MessageBox.Show("Opravdu chceš skončit?", "Dotaz",
        MessageBoxButtons.YesNo, MessageBoxIcon.Question)
        == DialogResult.Yes)
        Close();
}
```

Dialogové okno, které tento handler vyvolá, ukazuje obr. 13.7. Pokud-li uživatel tlačítko Ano, zavolá metodu `Close()`, jinak neudělá nic.



Obr. 13.7 Slušně vychovaný program se zeptá, zda smí skončit

Poznamenejme, že stejného výsledku bychom dosáhli také příkazem

```
if(MessageBox.A(0, "Opravdu chceš skončit?", "Dotaz",
    (int)MessageBoxButtons.YesNo|(int)MessageBoxIcon.Question)
    == (int)DialogResult.Yes) Close();
```

Zde ovšem musíme výčtové typy přetypovat na `int`, a to je nepohodlné. Proto budeme nadále používat jen metodu `MessageBox.Show()`.

Problém

Brzy ovšem zjistíme, že toto řešení v sobě skrývá problém: Program se zeptá, jen když ho ukončíme příkazem `Hra|Konec` z nabídky. Ukončíme-li ho pomocí systémové nabídky nebo stisknutím tlačítka s křížkem v titulkové liště, skončí bez ptaní.

Closing

Řešení nám poskytuje událost `Closing`, která nastane těsně před uzavřením okna. Její handler má druhý parametr typu `System.ComponentModel.CancelEventArgs`; přiřazením hodnoty `true` vlastnosti `Cancel` tohoto parametru můžeme událost zrušit a tak zabránit uzavření okna a ukončení programu. Napišeme tedy handler

```
private void Vokno_Closing(
    object sender, System.ComponentModel.CancelEventArgs e)
{
    if(MessageBox.Show("Opravdu chceš skončit?", "Dotaz",
        MessageBoxButtons.YesNo, MessageBoxIcon.Question) ==
        DialogResult.No)
        e.Cancel = true; // Zruš událost
}
```

Handler zaregistrujeme příkazem

```
this.Closing += new
    System.ComponentModel.CancelEventHandler(this.Vokno_Closing);
```


Další problém

Po této úpravě se program zeptá, zda smí skončit, i když ho uzavřeme pomocí systémové nabídky. Ihned se ale vynoří nový problém: Ukončíme-li ho příkazem `Hra|Konec`, zeptá se dvakrát.

Lepší řešení

Příčinou je, že událost `Closing` nastává také v důsledku volání metody `Close()`.

Řešení je jednoduché – handler `menuKonec_Click()` jen zavolá metodu `Close()`,

```
private void menuKonec_Click(object sender, System.EventArgs e)
{
    Close();
}
```

a o dotaz při ukončení se postará pouze handler `Vokno_Closing()`,

Používání programu

Než se pustíme do programování ostatních handlerů, ujasníme si, jak bude uživatel náš program používat. Předpokládáme, že nejprve pomocí číselníku nastaví požadovaný počet dam. Pak tuto hodnotu potvrdí zadáním příkazu `Nová hra` z nabídky `Hra`. Nakonec opakovaným příkazem `Hra|Další` nebo stisknutím tlačítka `Najdi další řešení` způsobí postupné zobrazení prvního, druhého a dalších řešení.

Nová hra

To znamená, že při zadání příkazu `Hra|Nová` nejprve zjistíme hodnotu nastavenou na číselníku `početDam`. Připomeňme si, že jde o vlastnost typu `decimal`, a proto ji musíme přetypovat. Uložíme ji do datové složky `kolik`. Pak vytvoříme novou instanci třídy `Řešitel`, hodnotu `kolik` použijeme jako parametr konstruktoru. Pak nastavíme text v titulku okna, povolíme používání položky `Hra|Další` a tlačítka `Najdi další řešení` (vlastnosti `Enabled` přiřadíme `true`), do datové složky `nalezeno` uložíme `false` (ještě nebylo nalezeno žádné řešení) a necháme překreslit panel. K tomu použijeme metodu `Invalidate()`,

Zdrojový text handleru `menuNová_Click()` může mít tvar

```
private void menuNová_Click(object sender, System.EventArgs e)
{
   olik = (int)this.početDam.Value;
    ř = new Řešitel(kolik);
    this.Text = "Problém " +olik + " dam, nalezeno " +
        ř.PočetŘešení + " řešení";
    this.dalšíŘešení.Enabled = true; // Povolíme hledání
    this.menuDalší.Enabled = true; // dalšího řešení
    nalezeno = false; // Ještě jsme nenašli žádné
    panel.Invalidate(); // Překreslí obsah panelu
}
```

Metoda `Invalidate()` označí obsah panelu za neplatný, a to způsobí vznik události `Paint`.

Další řešení

Handler `menuDalší_Click()`, který je společný pro položku `Hra|Další` a tlačítko `Najdi další řešení`, nejprve zavolá metodu `NajdiDalšíŘešení` instance ř třídy `Řešitel`. Pokud tato metoda vrátí `true`, nechá překreslit obsah panelu (zavolá `panel.Invalidate()`) a upraví text v titulku. Jinak zobrazí dialogové okno s upozorněním, že další řešení neexistují, a zakáže používání tlačítka a příkazu `Další`.

```
private void menuDalší_Click(object sender, EventArgs e)
{
    nalezeno = ř.NajdiDalšíŘešení(out řešení);
    if(!nalezeno) // Pokud jsme nenašli žádné řešení
    {
        MessageBox.Show("Další řešení neexistují", "To je vše",
            MessageBoxButtons.OK, MessageBoxIcon.Information);
        this.dalšíŘešení.Enabled = false;
        this.menuDalší.Enabled = false;
    }
    panel.Invalidate();
    this.Text="Problém " +olik + " dam, nalezeno "
        + ř.PočetŘešení + " řešení";
}
```

Kreslení

Při kreslení pracujeme s instancí třídy, jež je potomkem abstraktní třídy `Graphics`. Tato třída představuje abstraktní kreslicí plochu – v našem případě povrch panelu. Získáme ho prostřednictvím vlastnosti `Graphics` druhého parametru handleru.

Pero

Při kreslení používáme *pero* (`pen`); to je objekt třídy `System.Drawing.Pen`, který shrnuje barvu, sílu a jiné vlastnosti čáry. Vedle toho používáme občas také *štětec* (`brush`), který vyjadřuje vlastnosti pozadí; zde ho ale nebudeme potřebovat.

Pero má vlastnosti `Width` (síla čáry), `Color` (barva) a další. Barvu můžeme předepsat přímo v konstruktoru. Při kreslení můžeme používat metody `DrawLine()`, `DrawEllipse()` třídy `Graphics`; jejich prvním parametrem je vždy *pero*.

V metodě `panel_Paint()` nejprve získáme odkaz na objekt třídy `Graphics`, představující povrch panelu. Jestliže jsme při předchozím volání metody `ř.NajdiDalšíŘešení()` řešení našli, vytvoříme nové *pero* o síle 1 pixel a černé barvě a nakreslíme s ním sít svíslých a vodorovných čar představujících šachovnici. Pak vytvoříme *pero* o síle 5 pixelů v červené barvě, které použijeme k nakreslení dam na pozice určené nalezeným řešením (uloženým v poli `řešení`).

Jestliže jsme žádné řešení nenašli, vyplníme panel bílou barvou pomocí metody `Clear()`.

```
private void panel_Paint(object sender, PaintEventArgs e)
{
}
```

```

Graphics g = e.Graphics;
if(nalezeno)
{
    // Pero pro čáry
    Pen pen = new Pen(Color.Black);
    pen.Width = 1;
    int delta = VELIKOST_PANELU/kolik;
    for(int i = 1; i < kolik; i++)
    {
        // Nakresli políčka šachovnice
        g.DrawLine(pen, 0, i*delta, 360, i*delta);
        g.DrawLine(pen, i*delta, 0, i*delta, 360);
    }
    pen.Width = 5;
    // Pero pro dámy
    pen.Color = Color.Red;
    for(int m = 0; m < kolik; m++)
    {
        // Nakresli dámy
        g.DrawEllipse(pen, m*delta+delta/6,
            řešení[m]*delta+delta/6, 2*delta/3, 2*delta/3);
    }
}
else g.Clear(Color.White);
}

```

Dialogové okno O programu

Nakonec zbývá vytvořit dialogové okno „O programu“ se základními informacemi o tomto programu. Typicky v něm najdeme jméno programu, číslo verze, informace o autorských právech a případně logo firmy, která program vytvořila. Vedle toho obsahuje tlačítko OK. Většinu věcí již umíme, zastavíme se jen u těch, které pro nás budou nové.

Nové okno vytvoříme v samostatném souboru, který nazveme Dialog.cs. Jeho třídu OProg opět odvodíme od třídy Frame. Bude obsahovat několik komponent Label, představujících nápisy, tlačítko a komponentu PictureBox pro obrázek.

```

private PictureBox obrázek;
private Label cancl;
private Label cancl2;
private Label cancl3;
private Button tlačítko;

```

Písmo

Již umíme vytvořit komponentu, umístit ji a určit její velikost. Chceme-li změnit velikost písma v komponentě Label, použijeme vlastnost Font, která je typu System.Drawing.Font. Konstruktor této třídy má jako první parametr řetězec obsahující název písma, dále velikost v typografických bodech a styl (zda jde o kurzivu, tučné písmo atd.) Velikost zadáváme jako číslo typu float, styl jako hodnotu výčetového typu FontStyle. Můžeme použít např. konstant Bold, Italic nebo

Regular, jež znamenají tučné písmo, kurzivu, resp. normální písmo. Písmo změníme v metodě InitializeComponent() příkazem

```

this.cancl.Font = new Font("Microsoft Sans Serif", 12F,
    FontStyle.Bold);

```

Zarovnání textu

Chceme-li zarovnat text doprostřed plochu komponenty Label, použijeme vlastnost TextAlign.

```

this.cancl.TextAlign = ContentAlignment.MiddleCenter;

```

Obrázek

S obrázkem v komponentě PictureBox můžeme pracovat pomocí vlastnosti Image, která je typu System.Drawing.Image. Jako obrázek použijeme např. soubor Strom.gif, který najdete v adresáři tohoto programu na WWW.

Poznamenejme, že Image je abstraktní třída, takže musíme použít některého z potomků. Nám se bude hodit třída Bitmap, protože jde o bitmapový obrázek. Jako parametr konstruktoru zadáme řetězec představující jméno souboru (včetně cesty):

```

Image im = new Bitmap("K:\\prog\\kap13\\03\\strom.gif");
this.obrázek.Image = im;

```

Problém ale je, že velikost našeho obrázku neodpovídá zvolené velikosti komponenty obrázek. To můžeme vyřešit vyřešíme nastavením vlastnostiSizeMode komponenty PictureBox:

```

this.obrázek.SizeMode = PictureBoxSizeMode.StretchImage;

```

Použijeme hodnotu StretchImage typu PictureBoxSizeMode, jež předepisuje, že se má obrázek přizpůsobit velikosti komponenty.

Tlačítko

S tlačítkem již zacházet umíme. Dialogové okno bude obsahovat jediné tlačítko s nápisem OK. Protože však chceme, aby uživatel mohl okno O programu zavřít nejen stisknutím tlačítka myši, ale i stisknutím klávesy Enter nebo Esc, nastavíme ve třídě okna v metodě InitializeComponent() vlastnosti AcceptButton (tlačítko, které bude reagovat na Enter) a CancelButton (tlačítko, které bude reagovat na Esc):

```

this.AcceptButton = this.tlačítko;
this.CancelButton = this.tlačítko;

```

Handler okno pouze schová – přiřadí false vlastnosti Visible.

```

private void tlačítko_Click(object sender, System.EventArgs e)
{
    this.Visible = false;
}

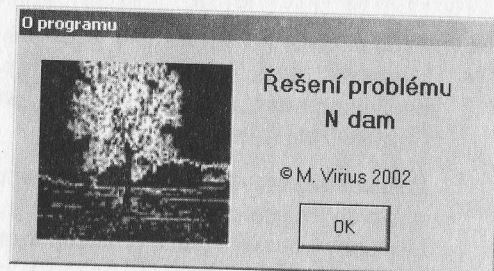
```

Systémová nabídka

Nakonec odstraníme systémovou nabídku (ikony v titulkové liště), neboť ta u tohoto okna nemá smysl, a zamezíme změnám velikosti okna. K tomu použijeme vlastnosti `ControlBox` a `FormBorderStyle`:

```
this.ControlBox = false;  
FormBorderStyle = FormBorderStyle.FixedDialog;
```

Výsledné okno ukazuje obr. 13.8.



Obr. 13.8 Dialogové okno O programu

Zobrazení okna

Nyní zbývá zobrazit okno v handleru položky `menuOProgramu` v hlavním oknu programu. To se vypadá jednoduše – prostě vytvoříme instanci a zobrazíme ji:

```
// Ve třídě Vokno  
private void menuOProgramu_Click(object sender, EventArgs e)  
{
```

```
    OProg op = new OProg();  
    op.Visible = true;  
}
```

Jenže takto lze zobrazit obyčejné okno, nikoli dialogové okno. Takto zobrazené okno může ztratit fokus, může být překryto hlavním oknem – a to je netypické. Chceme-li aby se chovalo jako modální dialogové okno, tj. jako okno, které nemůže ztratit fokus, dokud ho uživatel neobslouží a neuzavře, musíme ho zobrazit pomocí metody `ShowDialog()`. Konečná podoba handleru nabídky `menuOProgramu` tedy bude

```
private void menuOProgramu_Click(object sender, EventArgs e)  
{  
    OProg op = new OProg();  
    op.ShowDialog();  
}
```



Celý program, najdete na WWW v adresáři `Kap13\03` v souborech `Vokno.cs`, `Dialog.cs` a `Damy.cs`. Poznamenejme, že druhé dva soubory je třeba překládat jako knihovny, tj. s parametrem `/t:library`.

Literatura

- [1] *Microsoft C# Language Specifications*, Microsoft Press 2001. ISBN 0-7356-1448-2.
- [2] B. Alahari, P. Drayton, B. Merrill: *C# Essentials*, O'Reilly & Associates 2001. ISBN 0-596-00079-0.
- [3] Tom Archer: *Inside C#*, Microsoft Press 2001. ISBN 0-7356-1288-9. Český překlad *Myslím v jazyku C#*, Grada Publishing 2002, ISBN 80-147-0301-7.
- [4] D. E. Knuth: *The Art of the Computer Programming*, Vol. 1–3, Addison-Wesley. (Tato kniha existuje v řadě vydání od 70. let po současnost. K dispozici je také ruský překlad *Iskusstvo programirovaniya dlja EVM*, Mir, Moskva 1978.)
- [5] N. Wirth: *Algoritmy a struktury údajov*, Alfa, Bratislava 1987.
- [6] *Software Fault Tolerance*, Editor M. R. Lyu, John Wiley & Sons, New York 1995. ISBN 0-471-95068-8.
- [7] John Gough: *Compiling for the .NET Common Language Runtime (CLR)*, Prentice Hall 2002. ISBN 0-13-062296-6.
- [8] ČSN 97 6030 – *Abecední řazení*. Český normalizační institut, březen 1994.
- [9] Bloch, A.: *Murphyho zákon*, Svoboda-Libertas, Praha 1993. ISBN 80-205-0322-3.
- [10] Kačmář, D.: *Programujeme .NET aplikace*, Computer Press, Praha 2001. ISBN 80-7226-569-5.