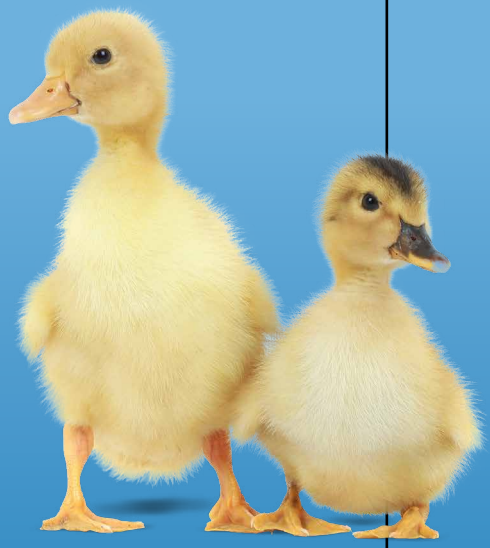
EDICE
myslíme v ...

knihovna programátora

- Přehledná a praktická učebnice C# pro začátečníky i uživatele ostatních programovacích jazyků
- Základní programovací konstrukce a potřebné pojmy (program, operační systém, programovací jazyk, objektově orientované programování a další)
- Programovací jazyk C# od základů k pokročilým konstrukcím
- Úvod do nejdůležitějších knihoven, .NET Core, verze .NET 5 a do používání vývojových nástrojů



Programování v C#

od základů k profesionálnímu použití

MIROSLAV VIRIUS

soubory ke stažení na WWW.GRADA.CZ

GRADA

EDICE
myslíme v ...

knihovna programátora

Programování v C#

od základů k profesionálnímu použití

Miroslav Virius

GRADA
Publishing

Miroslav Virius

Programování v C#

Od základů k profesionálnímu použití

Vydala Grada Publishing, a.s.
U Průhonu 22, Praha 7
obchod@grada.cz, www.grada.cz
tel.: +420 234 264 401
jako svou 7866. publikaci

Odborná korektura Jan Hájek, NETWORG
Odpovědný redaktor Petr Somogyi
Sazba Petr Somogyi
Počet stran 424
První vydání, Praha 2021
Vytiskla Tiskárna v Ráji, s.r.o., Pardubice

ISBN 978-80-271-4004-6 (ePub)
ISBN 978-80-271-4003-9 (pdf)
ISBN 978-80-271-1216-6 (print)

Obsah

Předmluva

Kniha, kterou jste právě otevřeli, vás seznámí se základy programování v jazyce C#. Je určena jak čtenářům, kteří ještě vůbec neprogramovali, tak i čtenářům, kteří již v nějakém jazyce programovali a chtějí se s tímto programovacím jazykem seznámit. Přitom předpokládám, že jste sice začátečníky v programování, ale umíte s počítačem zacházet uživatelsky, tedy že umíte spustit program, zkopírovat nebo smazat soubor atd. V průběhu osmnácti kapitol této knihy se vás pokusím dovést na úroveň lehce pokročilého programátora.

Programovací jazyk C#

Programovací jazyk C# uveřejnila firma Microsoft v roce 2002. I když byl původně určen pouze k programování pro Windows, v současné době se používá k vytváření aplikací i pro řadu dalších platforem, jako je Linux, macOS, Android a další a lze v něm vytvářet cloudové aplikace.

Co v této knize najdete

Jediným způsobem, jak se naučit nějaký programovací jazyk, je psát v něm programy, a proto už ve druhé kapitole začneme programovat. Cílem druhé, třetí a čtvrté kapitoly je uvést na scénu vybrané konstrukce jazyka C# a umožnit vám zkoušet si vše, o čem si v následujících kapitolách povíme, na vlastních programech – jednoduchých a možná nešikovných, ale fungujících.

Než se ovšem pustíme do programování, musíme si ujasnit některé pojmy a sjednotit terminologii. Proto se v první kapitole seznámíme s pojmy, které budeme používat. Povíme si, co je třeba vědět o počítači, o programovacích jazycích, algoritmech, objektově orientovaném programování a některých dalších věcech, bez nichž se při programování neobejdeme.

Ve druhé kapitole napíšeme svůj první program, ukážeme si, jak ho přeložit – neboť jazyk, v němž ho píšeme, je jiný než jazyk, v němž „myslí" počítač – a jak ho spustit. Seznámíme se s integrovaným vývojovým prostředím Visual Studio 2019, které budeme v této knize používat.

Ve třetí kapitole napíšeme několik verzí jednoduchých programů, které přečtou číslo, vypočtou z něj jistou hodnotu a výsledek vypíší. To nám umožní seznámit se alespoň povrchně s některými základními programovacími konstrukcemi, s rozdělením programu do několika souborů, s pojmem projektu apod. Napíšeme také jednoduchou vlastní programovou knihovnu pro čtení dat z klávesnice. Se znalostmi z této kapitoly byste měli být schopni zkoušet si vše, co se v dalších kapitolách naučíte, na vlastních programech.

Ve čtvrté kapitole se seznámíme s kolekcemi – nástroji, které umožňují uchovávat v programu libovolné množství dat –, seznámíme se velmi povrchně s generickými třídami a ukážeme si, jak ladit program – tedy jak zjistit, proč dělá něco jiného, než co jsme chtěli naprogramovat.

Od páté kapitoly začíná výklad „naostro". Po seznámení se základními pojmy postupně poznáme vestavěné datové typy, příkazy jazyka C#, s výrazy a operátory, které máme k dispozici, naučíme se používat tzv. pole. Pak přijdou na řadu kapitoly věnované programátorem definovaným negenerickým objektovým typům, rozhraním, genericitě, delegátům a lambda-výrazům (to vše jsou zajímavé a užitečné konstrukce, které jazyk C# nabízí) a tzv. výjimkám (nástrojům pro ošetřování chyb za běhu). Samostatná kapitola je věnována tzv. nulovatelným typům. Nelze samozřejmě pominout nástroj označovaný zkratkou LINQ, který umožňuje získávat vybraná data z různých zdrojů – mohou to být už zmíněné kolekce, soubory a mnohé další.

Pak přijdou na řadu kapitoly věnované použití vybraných programových knihoven. Začneme nástroji pro práci se znakovými řetězci, tedy s textem. Pak se podíváme na vstupní a výstupní operace, na vytváření grafického uživatelského rozhraní pomocí knihovny Windows Forms a pomocí knihovny Windows Presentation Foundation a na nástroje pro paralelní počítání.

Na co se nedostalo

Víc se mi bohužel do této knihy nevešlo. Nedostalo se na tvorbu databázových aplikací, na tvorbu webových aplikací, na tvorbu mobilních aplikací, na práci s datovými typy, které nebyly v době překladu k dispozici (tzv. reflexi), na použití nespravovaného kódu a dynamických knihoven napsaných v jiných programovacích jazycích, na práci s místním nastavením (kulturou) a na mnoho dalších zajímavých a užitečných témat.

Nástroje

Výklad v této knize je založen na jazyce C# verze 8 publikované v roce 2019. Pro psaní, překlad, sestavení a ladění programů používám vývojové prostředí Visual Studio 2019 Community Edition, které lze zdarma získat na stránkách firmy Microsoft.

Není to ovšem jediná možnost: Dobrou alternativou je nástroj Visual Studio Code, který je také zdarma a který lze instalovat pod operačními systémy Windows 7, 8 a 10, Debian, Ubuntu, Red Hat, Fedora, Suse a macOS 10.10 a pozdějších. Jeho ovládání se poněkud liší od ovládání Visual Studia; to ale nic nemění na platnosti výkladu o jazyce C# a jeho knihovnách, které v této knize najdete.

Terminologie

V celé knize používám důsledně českou terminologii. Jsem přesvědčen, že použití vhodných českých názvů výrazně usnadní pochopení, oč jde. Anglické termíny samozřejmě uvádím alespoň při prvním výskytu také.

Příklady

Výklad v této knize doprovází velké množství příkladů. Jejich zdrojové texty si můžete stáhnout z webových stránek nakladatelství Grada Publishing (vyhledejte na *www.grada.cz* stránku této knihy), nebo z mých osobních stránek, jejichž adresu najdete dále. V příkladech důsledně používám české identifikátory, a to včetně háčků a čárek. Vím, že profesionální programátoři se tomu vyhýbají a že v mezinárodních týmech jsou anglické identifikátory samozřejmostí. Mám ale dlouholetou zkušenost, že při výkladu určeném začátečníkům mohou české identifikátory výrazně usnadnit orientaci v ukázkách zdrojového kódu – a to je můj hlavní cíl.

Na závěr

I přes veškerou péči, kterou jsem této knize věnoval, se do ní mohly vloudit chyby. Jestliže při studiu nějakou najdete, pošlete mi prosím zprávu na níže uvedenou adresu; bude-li to možné, uveřejním na svých webových stránkách opravu.

Miroslav Virius, Praha, 21. srpna 2020
miroslav.virius@fjfi.cvut.cz
http://people.fjfi.cvut.cz/virius

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

Kdesi – myslím, že šlo o návod, jak vyplnit daňové přiznání – jsem četl, že počítač je „stroj na zpracování informací". To je samozřejmě pravda, ale člověk, který se chce naučit programovat, o něm musí vědět přece jen víc.

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

- Součást, která opravdu „počítá", tedy zpracovává informace, a která 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, zařízení na čtení CD/DVD 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).
- Trvalá paměť slouží – jak název napovídá – 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 (tzv. pevný disk) nebo polovodičové disky (obvykle označovaný zkratkou SSD z anglického *Solid State Drive*). Vedle trvalé paměti napevno zabudované v počítači se často používá i externí trvalá paměť, kterou k počítači připojujeme pouze v případě potřeby, pro účely zálohování dat atd.
- Součást, jež propojuje všechny ostatní části počítače, se nazývá *sběrnice* (anglicky *bus*).

1.2 Operační paměť

Bity

Operační paměť 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 pracujeme jen výjimečně, protož je to nepohodlné, a to jak pro člověka, tak pro počítač. Bity v operační paměti se v dnešních počítačích sdružují do skupin velikosti 8 bitů a tyto skupiny 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 ve skutečnosti trochu složitější, ale to můžeme v souvislosti s jazykem C# ponechat stranou. Pro pochopení dalšího výkladu stačí, co jsme si řekli.)

1.3 Datové typy a proměnné

Není těžké zjistit, že nejmenší číslo, které může jeden bajt obsahovat, se skládá z osmi nul a představuje ve dvojkové i v desítkové soustavě nulu. Největší takové číslo se bude skládat z osmi jedniček a v desítkové soustavě vyjadřuje 255. To je pro téměř jakékoli počítání málo; v běžných programech se obvykle objevují daleko větší čísla. Proto se pro ukládání dat zpravidla 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 65 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 naprosto jiným významem.

Odtud je vidět, že počítači nestačí znát adresu bajtu nebo bajtů, s nimiž 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 *řetězců*, 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 adresou byla nepohodlná, a proto ji pojmenujeme, dáme jí *identifikátor*. Tomu se v programování říká *deklarace proměnné*.

1.4 Programy a programovací jazyky

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

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 monitoru 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* (v programátorštině *stroják*) 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.

Vyšší programovací jazyky

Je asi jasné, že programování ve strojovém kódu je velice namáhavé a nepřehledné. 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 jazyce je textový soubor, který obsahuje popis řešené úlohy vyjádřený pomocí vybraných anglických slov a pomocí výrazů zapsaných podobně jako v matematice. Napsat program ve vyšším programovacím jazyce je samozřejmě daleko snazší než napsat odpovídající program ve strojovém kódu. Ovšem nic není zadarmo: Program ve vyšším programovacím jazyce nelze na počítači přímo spustit, neboť počítač mu nerozumí. Takovýto program se proto musí buď *přeložit* do strojového kódu, nebo *interpretovat*. V obou případech k tomu potřebujeme další program, který to za nás udělá.

Překlad

Textový soubor, který obsahuje zápis programu ve vyšším programovacím jazyce, se zpravidla nazývá *zdrojový kód* nebo *zdrojový program*, v programátorské hantýrce *„zdroják"*. K *překladu* do strojového kódu (hovoříme také o *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. Dnes je však sestavovací program zpravidla součástí překladače.

Překladem (a následným sestavením) programu vznikne soubor 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 Basic.[1] Interpretované programy obvykle běží podstatně pomaleji

1 To se netýká současných verzí Visual Basicu; s programem v tomto jazyce se zachází stejně jako s programem v C#, jak o tom budeme hovořit dále.

než překládané programy. Vedle toho musíme na cílový počítač spolu s naším programem instalovat také interpretační program.

Jazyk 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 *Common Intermediate Language* (*CIL*, dříve označovaný *Microsoft Intermediate Language*, *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, 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.)

1.5 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 myší 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, macOS, IOS a Android. Poznamenejme, že pro všechny tyto systémy lze v jazyce C# vytvářet programy, to znamená, že je pro ně k dispozici prostředí .NET.

1.6 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 říkají, 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 (musíme mu napsat program). 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 – z 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 nám 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 algoritmus, 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.

PŘÍKLAD 1.1

Metodu shora dolů si ukážeme na následujícím příkladu: Máme skupinu `N` číselných proměnných, které označíme `A[0],A[1],...,A[N-1]`. Takovouto 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], ...,A[N-1]`. Čísla, označující jednotlivé prvky pole, se nazývají *indexy*. Naším úkolem je uspořádat pole `A` tak, aby platila podmínka

```
A[0]≤ A[1]≤ ... ≤ A[N-1]                                    (1)
```

Úloha „zpřeházet" hodnoty, uložené v prvcích pole, aby splňovaly podmínku (1), se nazývá *třídění* pole.[2] Naším úkolem tedy je setřídit pole `A`.

První pokus

Náš první pokus o napsání algoritmu třídění by mohl vypadat takto:[3]

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. A takto postupuj dále, dokud není pole setříděné.

pokračování ►

2 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, a podobné je to i v mnoha dalších (lidských i programovacích) jazycích.

3 Popsaná metoda se nazývá *třídění výběrem* (anglicky *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.

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

Klíčem k úspěchu bude následující úvaha: Poté, co v prvním kroku vyhledáme nejmenší prvek a dáme ho na první místo, je nejmenší hodnota již na svém místě a nemusíme se i ni dále starat. Stačí tedy setřídit 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],...,A[N-1]` a ten dát na druhé místo, ve třetím kroku najít nejmenší hodnotu mezi `A[2],...,A[N-1]` atd. V posledním kroku budeme hledat nejmenší z prvků `A[N-2]` a `A[N-1]`. Pak bude pole setříděné, tj. bude vyhovovat podmínce (1).

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

Naše nová formulace může vypadat takto („počítadlo" označíme `i`):

1. Do proměnné `i` vlož hodnotu 0.
2. Mezi prvky `A[i],A[i+1],...,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 rozdělit na menší.[4] 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],A[i+1],...,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] ≤ 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. K prohození dvou hodnot budeme potřebovat pomocnou proměnnou, kterou označíme `x`.

3. Vyměň prvek `A[min]` s prvkem `A[i]`.

3a. Do pomocné proměnné `x` ulož hodnotu `A[min]`.

3b. Do `A[min]` ulož hodnotu z `A[i]`.

3c. Do `A[i]` ulož hodnotu z `x`.

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

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

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 provždy, pojmenujeme ho a později ho budeme už jen používat.

Knihovna

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

1.7 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* (*OOP*; budeme také říkat jen *objektové programování*). Až se trochu seznámíme s jazykem C#, budeme si moci povědět více.

1.7.1 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 řídící vstřikování paliva v automobilovém motoru. Program musí v každém případě odrážet vybrané stránky řešené úlohy, musí být jejím *počítačovým modelem*.

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 provždy 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 bodu, úsečky atd. a bude muset naprogramovat také operace, které lze s grafickými objekty dělat.

Bod jako objekt

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 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.

Bod jako třída

V programu budeme určitě potřebovat větší množství bodů. Abychom nemuseli popisovat každý zvlášť, definujeme `Bod` jako nový datový typ – jako *třídu* (*class*). 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*, které popisují jeho polohu a barvu.[5]

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 (datové složky objektu),
- operace s ním (metody).

Datové složky spolu s metodami shrnujeme pod společné označení *složky třídy*. Později uvidíme, že třída v jazyce C# může mít ještě další druhy složek.

Oprávnění přístupu (přístupová práva)

Jak pro datové složky, tak pro metody můžeme navíc určit tzv. přístupová práva. Můžeme např. předepsat, že určitou složku smějí používat pouze metody dané třídy (pak hovoříme o *soukromé složce*), nebo že je smí používat kdokoli (složka je *veřejně přístupná*, *veřejná*). O souboru všech veřejně přístupných složek hovoříme také jako o *rozhraní třídy*.

Poznamenejme, že datové složky deklarujeme zpravidla jako soukromé a pro zacházení s nimi mimo třídu používáme tzv. *přístupové metody*. Přesněji, 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.

Zapouzdření

Skutečnost, že jsme deklarovali společně datovou reprezentaci objektů a operace s nimi a že jsme zároveň určili, kdo má právo kterou složku používat, se označuje *zapouzdření* (anglicky *encapsulation*).

1.7.2 Objektový program

Objektový program se skládá pouze z objektů, tedy z instancí objektových typů, a tyto objekty si navzájem *posílají zprávy*. To zní jako zaklínadlo, 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.

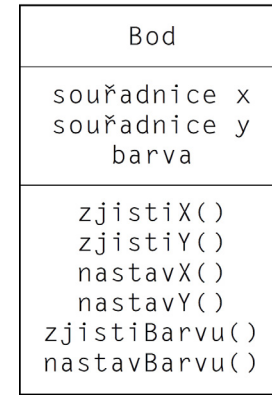
---

5 Datovým složkám se v teorii objektově orientovaného programování říkalo také *atributy*. V souvislosti s C# se ale tomuto termínu vyhneme, neboť jej budeme používat pro něco podstatně jiného.

1.7.3 Modelovací jazyk UML

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 – oddělený vodorovnou čarou – je seznam datových složek a pod ním, opět pod vodorovnou čarou, seznam metod. (Pokud seznamy datových složek nebo metod nepotřebujeme, můžeme obrázek zjednodušit a tyto seznamy vynechat.) Možné znázornění třídy `Bod` je zachyceno na obrázku 1.1.



Obrázek 1.1 Možné znázornění třídy Bod v diagramu tříd

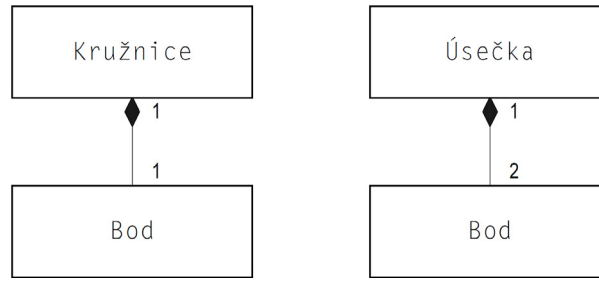
1.7.4 Skládání objektů

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* (i když ve skutečnosti jde spíše o skládání tříd).

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 obrázek 1.2.



Obrázek 1.2 Znázornění vztahu mezi třídami Bod, Úsečka a Kružnice. Čísla, připojená ke spojnicím, vyjadřují, že Kružnice obsahuje jeden Bod, zatímco Úsečka obsahuje dva Body

1.7.5 Dědění

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 `ÚsečkaRotující`. 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.

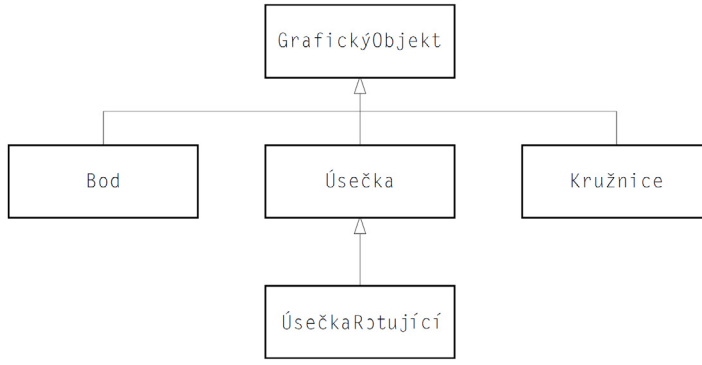
Bylo by jistě nesmyslné při programování otáčivé úsečky opisovat znovu vše, co jsme naprogramovali pro obyčejnou úsečku. Ostatně jedna ze základních zásad, kterou se při programování řídíme, zní NEOPAKUJTE SE, nepište stejný kód vícekrát.[6] Programovat jednu věc několikrát, to je jednak zbytečná práce navíc, jednak rozhodneme-li se něco změnit, musíme to měnit na několika místech a snadno se stane, že na některé místo zapomeneme.

Objektově orientované programování proto nabízí mechanizmus nazvaný nepříliš šťastně *dědění* (nebo též *dědičnost*, anglicky *inheritance*), který umožňuje od existující třídy odvodit třídu novou.

Odvozená třída bude obsahovat všechny datové složky a všechny metody třídy, od níž byla odvozena. K nim můžeme přidat nové datové složky a nové metody. Můžeme také *překrýt* (změnit; anglicky *override*) některou z metod předka. Při dědění nemůžeme žádnou datovou složku ani žádnou metodu odstranit.[7]

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

- Stejné datové složky jako rodičovská třída. My k nim přidáme novou datovou složku vyjadřující rychlost otáčení.
- Stejné 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.



Obrázek 1.3 Dědická hierarchie tříd z příkladu o grafickém editoru

Ovšem dědění 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… Jinými slovy, budeme se opakovat, a to, jak už víme, nikdy není dobré.

6 Anglicky *Don't Repeat Yourself*, ve zkratce DRY. Proto se někdy v žertu hovoří o „suchém principu".

7 Pro třídu, *od* které odvozujeme, se používá označení *rodičovská* nebo *bázová* třída nebo *předek*; v anglické terminologii dnes převažuje označení *base class*. Pro *odvozenou třídu* se používá také označení *potomek* nebo *dceřiná třída*. Tyto termíny budeme dále používat.

Proto vezmeme společné vlastnosti všech tříd grafických objektů z našeho programu a přeneseme 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`, jež objekt nakreslí, bude tuto metodu obsahovat i každá z odvozených tříd. Je ovšem pravděpodobné, že ji budeme muset v každém z potomků překrýt; 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 dědění 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 1.3 ukazuje dědickou hierarchii tříd, o kterých jsme hovořili v příkladu o grafickém editoru.

1.7.6 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. 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.

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`.)

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

1.7.7 Abstraktní třída

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, který to bude.

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

1.7.8 Dědění 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 bázové 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 `ÚsečkaRotující` jako potomka třídy `Úsečka`. Na druhé straně kružnice *není* bod, takže odvodit třídu `Úsečka` jako potomka třídy `Bod` nemá smysl.

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

Test je – má

Předchozí úvahy ukazují něco, čemu se občas říká *test je – má* (anglicky *is a – has a*, někdy psáno *isa – hasa*). 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.

1.8 Jazyk C#

Programovací jazyk C# je dílem firmy Microsoft. První zmínky o něm pronikly na veřejnost někdy v roce 2000 a oficiálně byl uveden na trh počátkem roku 2002 jako součást integrovaného vývojového prostředí Visual Studio .NET 2002. V současné době je k dispozici verze 8, publikovaná v roce 2019 jako součást Visual Studia 2019. Je to čistě objektový programovací jazyk určený k vývoji aplikací pro prostředí .NET, jež bylo původně pouze nadstavbou operačního systému Windows, nyní je jeho standardní součástí a jsou k dispozici i implementace pro řadu dalších prostředí – o tom jsme hovořili v oddílu 1.4.

Označení C# – mříž se někdy také píše jako horní index, $C^{\#}$ – se přepisuje **C sharp**. Tento název ve skutečnosti znamená C♯, tedy tón **Cis**. Znak ♯ ovšem není na klávesnici běžných počítačů k dispozici, proto se místo něj používá mříž. (Filosofie názvu je tedy podobná jako u jazyka C++: jde o následovníka jazyka C, v tomto případě posazeného o půl tónu výš.)

Tento programovací jazyk byl několikrát standardizován, poslední verze jeho mezinárodního standardu nese označení ISO/IEC 23270:2018 [1].

Mezijazyk CIL

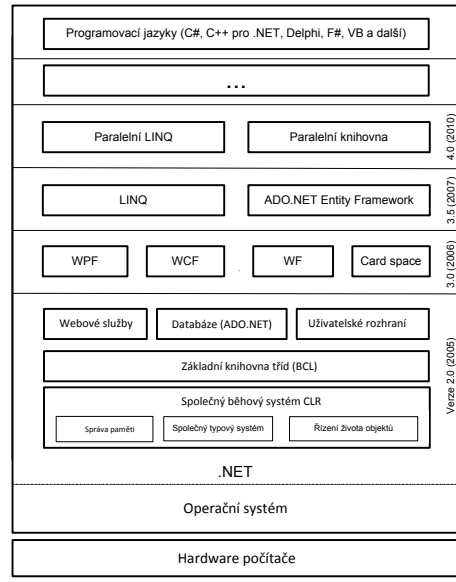
Už víme, že jazyk C# se překládá do společného mezijazyka označovaného zkratkou CIL. Ve skutečnosti se do tohoto jazyka překládají programy pro prostředí .NET napsané v jakémkoli programovacím jazyce, nejen v C#. Lze samozřejmě programovat i přímo v CIL, i když to není obvyklé, neboť jde o jazyk nižší úrovně (leží někde mezi strojovým kódem a vyššími programovacími jazyky).

1.8.1 Prostředí .NET

Už jsme se zmínili, že toto prostředí je k dispozici na různých platformách (pod různými operačními systémy), ovšem je třeba poznamenat, že jeho možnosti se na různých platformách zatím liší, i když je zřejmá tendence ke sjednocení). V této knize se budeme zabývat především vývojem aplikací pro Windows, protože tam jsou (zatím) možnosti nejširší.

V současné době pod Windows je k dispozici prostředí .NET verze 4.7, která je standardní součástí Windows 10 a lze ji doinstalovat do Windows 8.1. Toto prostředí se skládá z několika částí, o nichž se zde alespoň krátce zmíníme (viz též obr. 1.4).

Už také víme, že v okamžiku spuštění se program přeloží do strojového kódu překladačem JIT. Doplňme, že to je sice typický postup, není však nezbytný.



Obrázek 1.4 Struktura prostředí .NET

Jádrem prostředí .NET je *společný běhový systém*, anglicky označovaný *Common Language Runtime* (CLR). Tento systém zajišťuje běh programů přeložených z různých programovacích jazyků do mezijazyka CIL. CLR umožňuje jejich vzájemnou spolupráci, takže různé součásti programu mohou být napsány v různých programovacích jazycích. Stará se mj. o tzv. *automatickou správu paměti*, o řízení doby života objektů a o další věci nezbytné pro běh programů v tomto prostředí. (Dále se dozvíme, co to znamená.)

Další důležitou součástí prostředí .NET je *Knihovna tříd* (*Framework Class Library*, dříve nazývaná *Basic Class Library*). 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 touto knihovnou jsou ještě knihovny pro tvorbu grafického uživatelského rozhraní programů, pro tvorbu databázových aplikací, knihovny pro webové služby, alternativní nástroje pro tvorbu grafického uživatelského rozhraní (*Windows Presentation Foundation*, WPF), nástroje pro tvorbu systémů pro oběh dokumentů („správu pracovních postupů", Workflow Foundation, WF) a mnohé další specializované knihovny. Najdeme tu také vnořený dotazovací jazyk LINQ, nástroje pro paralelní programování a mnohé další; téměř každá nové verze prostředí .NET přinese nové nástroje.

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#.

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

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

Nejvyšší vrstvu prostředí .NET tvoří překladače z různých programovacích jazyků a překladač JIT. Přitom jsou možné tři režimy:

- Celý program se přeloží v okamžiku spuštění. To je nejobvyklejší režim.
- Celý program se přeloží v době instalace, uloží se ve strojovém kódu a tento „nativní obraz" se pak spouští.
- Při spuštění se přeloží jen část, která je právě nezbytná, a další části se spouštějí, až když jsou potřeba.

Tím se však zde nebudeme zabývat, budeme využívat výchozí režim.

1.8.2 Metadata

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

1.8.3 Společný systém typů

Společný systém typů (*Common Type System*, CTS) je základním nástrojem prostředí .NET, který umožňuje snadnou spolupráci kódu napsaného v různých programovacích jazycích. Určuje, jak jsou datové typy popsány a reprezentovány v paměti, jaké operace jsou s nimi dovoleny, ale také jakými pravidly se musí řídit programovací jazyky pro .NET, aby bylo možno kód napsaný v jednom jazyce použít v jazyce jiném.

CTS definuje objektový model závazný pro všechny programovací jazyky určené pro prostředí .NET – mimo jiné pravidla doby života objektů, pravidla dědění, pravidla viditelnosti datových typů a jejich složek, tedy práva pro přístup ke datovým typům a k jejich složkám, jako jsou metody, vlastnosti atd. Vedle toho definuje CTS sadu základních datových typů.

Hodnotové a odkazové typy

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 *odkazové* (*referenční*) *typy*, což jsou 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 překrývat.

Potomky třídy `Object` jsou i hodnotové typy. To znamená, že např. i pro proměnné celočíselných typů můžeme volat metody zděděné od třídy `Object`. Hodnotové typy ovšem nepodporují dědění (nelze od nich odvozovat potomky).

Podrobněji si o třídě `Object` povíme v 6. kapitole.

1.8.4 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) pracujeme v C# pouze s dynamickými objekty (vytváříme je pomocí operátoru `new`). O úklid vytvořených objektů se nemusíme starat, .prostředí NET 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. Obsahují mj. třídy pro vytváření grafického uživatelského rozhraní, pro programování vstupních a výstupních operací atd. Některé z 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`. (Jestliže jste se s těmito pojmy dosud nesetkali, nevadí; vysvětlíme si je později, v 9. kapitole.)
- Na rozdíl od Javy (a podobně jako C++) podporuje C# přetěžování operátorů.

1.9 Kde získat překladač C#

Překladač jazyka C# je součástí instalace vývojových nástrojů, jako je Visual Studio nebo Visual Studio Code. Informace v tomto oddílu mohou bohužel za čas pozbýt platnosti, neboť s příchodem další verze těchto nástrojů se může změnit licenční politika firmy Microsoft, může se změnit způsob instalace atd. To, co si zde řekneme, platí pro tyto nástroje v roce 2020.

Visual Studio 2019 Community Edition

Už víme, že překladač jazyka C# je součástí všech provedení integrovaného vývojového prostředí[8] **Microsoft Visual Studio 2019**. Pro nás bude zajímavá edice zvaná Community Edition,

8 Pod označením *integrované vývojové prostředí*, anglicky *Integrated Development Environment* (zkráceně *IDE*), se rozumí nástroj, který v sobě spojuje editor zdrojového textu, správu projektu, překladač, ladicí a další nástroje potřebné k vývoji aplikací.

která je k dispozici zdarma a v současné době ji lze získat na webových stránkách [3]. Z této stránky si stáhneme instalátor, program, který se postará o instalaci. Po spuštění se tento program zeptá na několik věcí, mimo jiné nám nabídne možnost vybrat si doplňky; žádný z nich však nebudeme potřebovat. Pak se rozeběhne stahování potřebných souborů a vlastní instalace. Počítejte s tím, že je zdlouhavá; skutečný čas bude samozřejmě záviset na kvalitě připojení a na možnostech vašeho počítače, ale i při velmi rychlém připojení a na kvalitním počítači trvá přes hodinu.

Visual Studio 2019 je k dispozici v řadě lokalizací (jazykových mutací); mezi nimi je i česká, která se – pokud máte česká Windows – instaluje automaticky. Profesionální programátoři dávají přednost nelokalizovaným vývojovým nástrojům, ovšem pro úplného začátečníka je podle mého názoru česky mluvící prostředí mnohem pohodlnější.

Při prvním spuštění po instalaci je třeba Visual Studio konfigurovat; pro naše účely stačí přijmout nabídnuté možnosti.

Pro instalaci Visual Studia je potřebný počítač PC s procesorem o rychlosti alespoň 1.8 GHz; doporučuje se, aby měl alespoň 4 jádra. Paměť by měla být alespoň 2 GB, ovšem doporučená velikost je 8 GB. Minimální verze Visual Studia vyžaduje 800 MB diskového prostoru.

Výsledkem instalace je 30denní zkušební verze Visual Studia 2019; abychom ho mohli používat trvale, je třeba se zaregistrovat (popřípadě vytvořit si účet u Microsoftu; obojí je samozřejmě také bezplatné).

Další podrobnosti najdete na adrese [4].

Visual Studio Code 1.48

Dobrou alternativou je nástroj Visual Studio Code, který je také zdarma a který lze instalovat pod operačními systémy Windows 7, 8 a 10, Debian, Ubuntu, Red Hat, Fedora, SUSE a macOS 10.10 a pozdějších. V době, kdy jsem dokončoval tuto knihu, byla k dispozici verze Visual Studio Code 1.48, která je ke stažení z webových stránek [17]. Tam najdete i dokumentaci a další zdroje informací o tomto nástroji. Jeho ovládání se poněkud liší od ovládání Visual Studia, ale jinak se jeho použitím nic nezmění – vše, co si povíme o jazyce C# a o jeho knihovnách, zůstane beze změny v platnosti, všechny programy, které napíšeme, budou bez jakékoli změny přeložitelné i pod Visual Studio Code.

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. Začneme překladem a spuštěním programu z příkazové řádky, abychom si předvedli, jak věci vlastně fungují. Pak si ukážeme tytéž úlohy v integrovaném vývojovém prostředí Visual Studia 2019.

2.1 Nápis na obrazovce

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

2.1.1 Zdrojový text

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 poznámkový blok (notepad) z Windows. Později budeme používat editor, který je součástí Visual Studia. Jako editor pro vývoj programů se nehodí například Word neb jiný editor, který text formátuje, tj. vkládá do něj i dodatečné informace o velikosti a typu písma apod.

Nyní tedy spusťte editor a vytvořte v něm soubor `První.cs`, který bude obsahovat následující text. Vytvořený soubor uložte do samostatného adresáře – pro jednoduchost budu předpokládat, že to je adresář `C:\Pokusy`:

```
/* Náš první program v C# - soubor První.cs */
class Program
{
   static void Main()
   {
      System.Console.WriteLine("Nazdárek, lidičky");
   }
}
```

Je důležité, abyste při opisování dodrželi velikost písmen, neboť v C# se velká a malá písmena důsledně rozlišují. Soubor také musí mít příponu `.cs` a uložíme ho v kódové stránce 1250.

Tento zdrojový text najdete také na WWW v adresáři `02` mezi soubory ke stažení.[9]

9 Adresu stránky, z níž si můžete stáhnout zdrojové texty příkladů, najdete na straně 16.

2.1.2 Příprava překladu z příkazové řádky

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 CIL. Postará se o něj překladač `csc.exe`, který je součástí prostředí .NET. Najdeme ho v adresáři, který je součástí instalace Visual Studia; přesné jméno a umístění tohoto adresáře bohužel závisí na verzi Visual Studia; ukážeme si však, že to není důležité.

Začneme tím, že si otevřeme si okno příkazové řádky (*command prompt*) s potřebnými nastaveními. Myší klepneme na tlačítko s ikonou okna v levém rohu hlavního panelu Windows a vyvoláme nabídku programů. Pak napíšeme `cmd`; Windows nám nabídne několik programů a my z nich zvolíme *Developer Command Prompt for VS 2019*; objeví se konzolové okno (okno příkazové řádky) s nastaveními pro Visual Studio.

Pak zadáme v příkazové řádce příkaz

```
CD \Pokusy
```

kterým přejdeme do pracovního adresáře.

2.1.3 Překládáme program z příkazové řádky

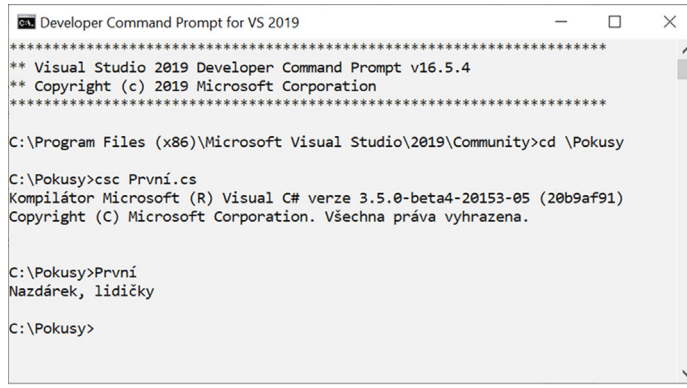
Nyní můžeme náš první program přeložit. K tomu použijeme příkaz

```
csc Program.cs
```

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

Soubory

Jestliže se nyní podíváte do adresáře, kde je uložen zdrojový program, najdete v něm kromě souboru `První.cz` také soubor `První.exe`. To je spustitelný soubor, tedy hotový program. Je ve formátu PE (portable executable, standardní formát spustitelných souborů pro Windows), neobsahuje ovšem téměř žádný strojový kód, ale instrukce v jazyce CIL. Budeme-li chtít hotový program přenést na jiný počítač, budeme přenášet právě tento soubor.



Obrázek 2.1 Překlad a spuštění našeho prvního programu z příkazové řádky. Hlášení vypsaná překladačem mohou být anglicky – záleží na jazykové mutaci Visual Studia, kterou máte k dispozici

Spuštění našeho prvního programu

Přeložený program spustíme z příkazové řádky příkazem

```
První
```

Náš program se rozeběhne a vypíše

```
Nazdárek, lidičky
```

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

2.1.4 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 `'csc' is not recognized as an internal or external command, operable program or batch file` nebo podobná zpráva, znamená to, že operační systém nenašel program `csc.exe`. Jestliže máte správně instalováno Visual Studio a použili jste *Developer Command Prompt for VS 2019*, nemělo by se to stát.
- Objeví-li se hlášení `error CS2001: Source file 'C:\Pokusy\Prvni' could not be found`, popřípadě `error CS2001: Zdrojový soubor 'C:\Pokusy\Prvni' se nenašel`, znamená to nejspíš, že jsme ve jménu souboru zapomněli uvést příponu `cs`.

Syntaktické chyby

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

```
/* Náš první program v C# - soubor První.cs */
class Program
{
   static void Main()
   {
      System.Console.WriteLine("Nazdárek, lidičky")
   }
}
```

a výsledek zkuste přeložit. Překladač vypíše zprávu

```
První.cs(5, 48): error CS1002: Očekával se středník (;)
```

jež nám říká, že v souboru `První.cs` našel překladač na šestém řádku chybu, kterou lze v dokumentaci vyhledat pod označením `CS1002`. Následuje stručná charakteristika chyby v češtině (nebo v angličtině, ale pod českými Windows hovoří překladač csc víceméně česky): *očekával se středník*. Bohužel, zdaleka ne vždy jsou chybová hlášení tak přehledná a snadno srozumitelná jako v tomto případě.

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 První.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 `První.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 řádky. Náš program by tedy mohl začínat také takto:

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

V C# existuje ještě třetí druh komentářů, tzv. dokumentační komentáře. Ty začínají znaky `///` a končí na konci řádky.

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 napoprvé tak, aby ho již nebylo třeba měnit, pusťte z hlavy; to se téměř nestává ani nejzkušenějším 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 `Program`. 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 `Program` tedy vypadá takto:

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

Metoda Main()

Jak víme, mohou třídy obsahovat jednak datové složky, ve kterých se ukládají data, jednak metody, které vyjadřují operace s instancemi tříd. Třída `Program` neobsahuje žádné datové složky. Obsahuje jedinou metodu, která se jmenuje `Main()`.[10] Její deklarace je

10 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 této knihy.

```
static void Main()
{
   System.Console.WriteLine("Nazdárek, lidičky");
}
```

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

```
První
```

kterým program spouštíme, říkáme vlastně 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 CIL 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 `{` a `}`. Naše metoda `Main()` obsahuje jediný příkaz,

```
System.Console.WriteLine("Nazdárek, lidičky");
```

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

```
Nazdárek, lidičky
```

a vypiš ho do standardního výstupu, tj. do konzolového okna, z něhož byl program spuštěn".

Místo o „textu" ovšem zpravidla hovoříme o (*znakovém*) *řetězci*. Řetězec, který má vystoupit, musíme zapsat mezi dvojici uvozovek; tento zápis označujeme jako *řetězcovou konstantu* nebo *řetězcový literál*.

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 `Program`. Slovo `void` říká, že metoda `Main()` nevrací žádný výsledek.

Už jsme si řekli, že příkaz `System.Console.WriteLine("Nazdárek, lidičky");` vypisuje zadaný text do standardního výstupu. Tento výstup můžeme příkazem operačního systému přesměrovat, např. do souboru. Spustíme-li program příkazem

```
První > pozdrav.txt
```

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

2.3 Ódy žlutého koně: čeština v programu

Nezadatelným právem uživatele počítače je, aby s ním počítač komunikoval v jeho rodném jazyce. 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 naťukat 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 žluťoučkém koni, který příšerně úpěl ďábelské ódy. (Podobná věta se jistě používá i při testování slovenštiny, já ji ale bohužel neznám.)

Zajděme ale ještě dál: Použijeme české znaky nejen ve výstupu, ale i v identifikátoru třídy a – stejně jako u prvního programu – 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("Žluťoučký kůň příšerně " +
                      "úpěl ďábelské ódy.");
   }
}
```

Program uložíme do souboru `ŽlutýKůň.cs`,[11] přeložíme ho příkazem

```
csc žlutýkůň.cs
```

a spustíme příkazem

```
žlutýkůň
```

Pod Windows se malá a velká písmena nerozlišují, takže jsme si dovolili je nerozlišovat ani v uvedených příkazech. Program po spuštění vypíše

```
Žluťoučký kůň příšerně úpěl ďábelské ódy.
```

Zdrojový text tohoto programu najdete na WWW mezi soubory ke stažení v adresáři 02.

Pozor na kódovou stránku

Přesměrujeme-li výstup programu `ŽlutýKůň` do souboru příkazem

```
ŽlutýKůň > výstup.txt
```

zjistíme, že jej běžné editory pro Windows (např. notepad) nepřečtou správně – uvidíme něco jako

```
¦luśoúźkě k…Í pý˘çernŘ ŁpŘl Ô belsk, ˘dy.
```

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

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

kódování Latin 2, tedy kódovou stránku 852. Proto i soubor `výstup.txt` obsahuje češtinu v kódové stránce 852.

Rozdělený řetězec

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

```
"Žluťoučký kůň příšerně " + "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 cesky (tj. bez háčků a čárek). Důvod je jednoduchý: Už jsme si naznačili, že v 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 vidite z teto kratke veticky, cesky psane komentare byvaji spatne srozumitelne.* Nebudeme si tedy přidělávat zbytečnou práci s luštěním „cestiny".

Také používání znaků s diakritickými znaménky v identifikátorech, tedy ve jménech tříd, proměnných ap., není příliš obvyklé – především proto, že to donedávna dovolovalo jen málo programovacích jazyků. C# bylo spíše čestnou výjimkou; většina programovacích jazyků požadovala, aby se identifikátory skládaly pouze z písmen anglické abecedy a z číslic. Ovšem dnešní standardy řady programovacích jazyků to dovolují a většina současných překladačů to podporuje. Je to logické: Můžeme-li dávat součástem svého programu nezkomolená jména, můžeme se snáze soustředit na problém, který řešíme. *Proto budeme této možnosti využívat.*

2.4 Sestavení (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 *sestavení* (*assembly*).

Seatavení 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 jazyce CIL také tzv. metadata. To jsou doplňující informace o datových typech – např. jak se jmenují třídy v sestavení, 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 sestavení je také uložena informace o *verzi* a o *kultuře*, tedy o jazykové mutaci (*culture*). Verze jsou číslovány, číslo verze sestavení se skládá z hlavního a vedlejšího čísla, čísla překladu a čísla revize.

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

Rozsah této knihy nám nedovolí zabývat se sdílenými sestaveními.

2.5 Opět žlutý kůň, tentokrát s Visual Studiem 2019

Nyní si ukážeme, jak vytvořit, přeložit a spustit program ve Visual Studiu 2019. Nejprve si ale musíme vysvětlit některé pojmy.

2.5.1 Projekt a řešení

Naše programy se zatím skládaly z jednoho zdrojového souboru; u rozsáhlejšího programu jich však bude více. Všechny soubory, které jsou potřebné k vytvoření programu, shrnujeme pod označení *projekt*. Ovšem ne vždy se aplikace, kterou chceme vytvořit, skládá z jediného projektu, a proto používáme ve Visual Studiu ještě pojem *řešení* (*solution*). To je skupina souvisejících projektů. Visual Studio rozlišuje několik druhů projektů:

- *Konzolová aplikace* (*console application*). To je program, který spouštíme z příkazové řádky, který čte data z příkazové řádky a do příkazové řádky zapisuje také své výsledky. (Může pracovat i se soubory na disku, ale nemá grafické uživatelské rozhraní, tedy okna). V současné verzi Visual Studia je ve dvou variantách.
- *Aplikace založená na knihovně Windows Forms.* To je aplikace s grafickým uživatelským rozhraním.
- *Knihovna tříd.* Slouží k vytvoření vlastní programové knihovny.
- … a další.

Naše první programy byly konzolové aplikace a zpočátku u nich zůstaneme, k některým pokročilejším druhům projektů se dostaneme později.

Platforma

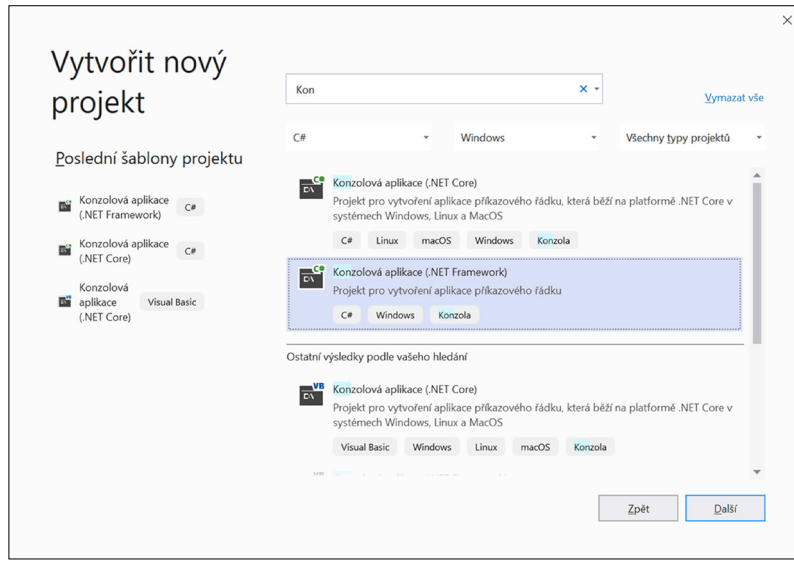
U každého projektu musíme určit *cílovou platformu*. Tím se myslí, zda jde o projekt pro Windows, Linux nebo některé z dalších prostředí, pro které můžeme v jazyce C# psát programy. Visual Studio nabízí (v abecedním pořadí) tyto možnosti: Android, Azure, iOS, Linux, macOS, tvOS, Windows, Xbox. My se budeme zabývat projekty pro Windows, neboť tato platforma nabízí v současné době nejvíce možností.

2.5.2 Projekt nového programu

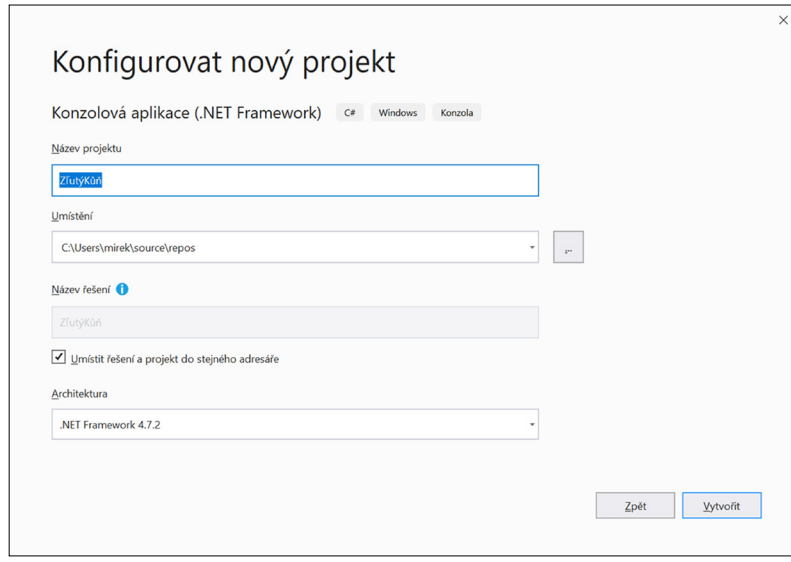
Po spuštění se zobrazí pomocné okno, které nám nabízí, zda chceme vytvořit nový projekt, otevřít existující projekt atd.; pokud jsme ve Visual Studiu už pracovali, nabídne nám také nedávno otevřené projekty. Vybereme možnost *Vytvořit nový projekt*. (Jestliže jsme v IDE už pracovali, vybereme v nabídce příkaz *Soubor* | *Nový* | *Projekt*...) Zobrazí se dialog *Vytvořit nový projekt* (viz obrázek 2.2).

Vedle titulního nápisu tohoto okna je textové pole; zapíšeme do něj *Konzolová aplikace* (ve skutečnosti stačí prvních několik písmen). IDE ze seznamu možných druhů projektů vybere ty, jejichž jména takto začínají. Z nich zvolíme *Konzolová aplikace (.NET Core).* Prostředí nám nabídne ještě možnost *.NET Framevork*; tato možnost byla ve starších verzích Visual Studia jediná, dnes je však na ústupu, a proto se jí budeme věnovat jen okrajově.

Pod záhlavím tohoto dialogu, nad seznamem typů projektů, je rozbalovací seznam s nápisem *Všechny jazyky*; vybereme v něm C#, neboť v tomto jazyce chceme programovat. Vedle je rozbalovací seznam s nápisem *Všechny platformy*, v němž určíme, pro jakou platformu (pro jaké zařízení a pro jaký operační systém) bude náš program určen; vybereme Windows. Ve třetím rozbalovacím seznamu ponecháme *Všechny typy projektů* a stiskneme tlačítko *Další*. Objeví se dialog *Konfigurovat nový projekt* (obr. 2.3).



Obrázek 2.2 Dialog Vytvořit nový projekt, v němž jsme nastavili volby



Obrázek 2.3 Dialog Konfigurovat nový projekt, v němž jsme zadali jméno projektu

V prvním vstupním poli zadáme jméno projektu, např. `ŽlutýKůň`; umístění projektu, nabídnuté ve druhém textovém poli, ponecháme (je v adresáři `Users\jméno\source\repos`, kde `jméno` zastupuje uživatelské jméno programátora) a zaškrtneme políčko *Umístit řešení a projekt do stejného adresáře*. V posledním textovém poli, nadepsaném *Architektura*, můžeme zvolit prostředí, pro které bude náš program určen; ponecháme nabídnutou hodnotu a stiskneme tlačítko *Vytvořit*.

Visual Studio vytvoří soubor `Program.cs` obsahující následující zdrojový text a zobrazí ho v editoru (obr. 2.4):

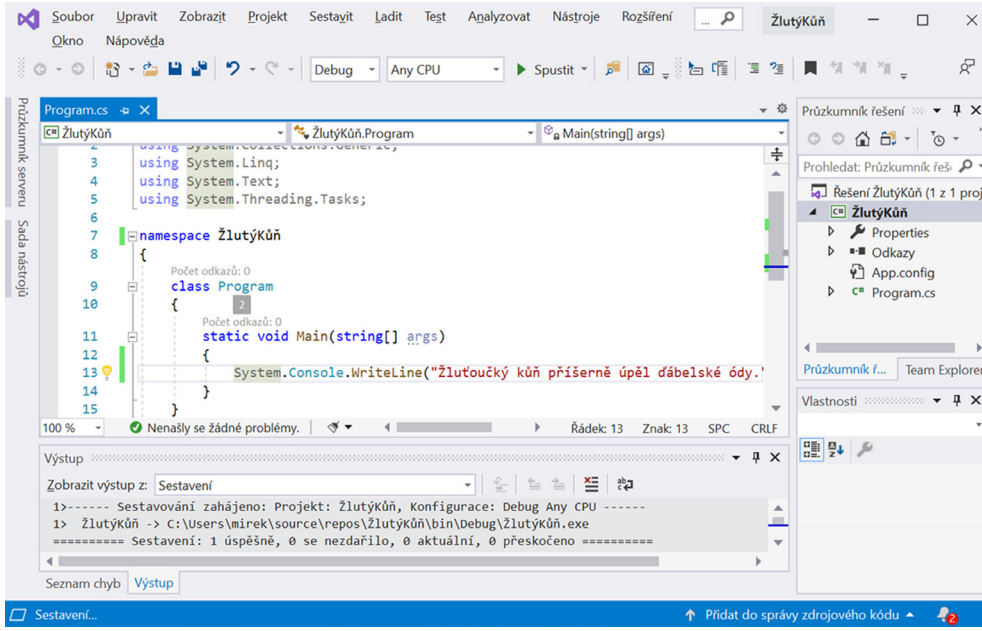
```
using System;
using System.Collections.Generics;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ŽlutýKůň
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

Jak vidíte, poněkud se liší od našich dosavadních programů, i když jeho smysl je v podstatě stejný; k tomu se vrátíme dále, na konci tohoto oddílu. Začneme však tím, že do metody `Main()` doplníme příkaz vypisující větu o žluťoučkém koni – to pro nás nebude nic nového – a pak upravený soubor uložíme příkazem *Soubor* | *Uložit* z nabídky nebo klávesovou zkratkou Ctrl+S. Můžeme také použít tlačítko s ikonou diskety na panelu nástrojů.

Překlad

Naším dalším krokem bude překlad programu. K tomu nám poslouží příkaz *Sestavit* | *Sestavit ŽlutýKůň* z nabídky nebo klávesová zkratka Shift+B. Výsledek překladu a sestavování nám Visual Studio oznámí na panelu ve spodní části okna; bude-li vše v pořádku, bude tato zpráva končit textem

```
Sestavení: 1 úspěšně, 0 se nezdařilo, 0 aktuální, 0 přeskočeno
```



Obrázek 2.4 Okno Visual Studia 2019 se zdrojovým textem našeho programu a se zprávou o výsledku překladu