Microsoft

Ján Hanák

Praktické objektové programování v jazyce C# 4.0

Microsoft Visual Studio

Microsoft .NET

Ján Hanák

Praktické objektové programování v jazyce C# 4.0

Artax
2009

Autor: Ing. Ján Hanák, MVP

Praktické objektové programování v jazyce C# 4.0

Vydání:	první
Rok prvního vydání:	2009
Náklad:	150 ks
Jazyková korektura:	Ing. Ján Hanák, MVP
Vydal:	Artax a.s., Žabovřeská 16, 616 00 Brno pro Microsoft s.r.o., Vyskočilova 1461/2a, 140 00 Praha 4
Tisk:	Artax a.s., Žabovřeská 16, 616 00 Brno
ISBN:	978-80-87017-07-4

Obsah

Předmluva

Objektově orientované programování je stále v kurzu. Konec konců, není se ani čemu divit, vždyť programování s objekty je denním chlebem naprosté většiny programátorů, softwarových vývojářů a tvůrců počítačových aplikací. Jazyk C# 4.0 je pro objektové programování jako dělaný: na jedné straně obsahuje mohutné programové konstrukce pro implementaci všech prvků koncepce OOP a na straně druhé poskytuje sofistikované nástroje, jejichž pomocí je práce s objekty spíše příjemnou zábavou než nutnou povinností. Jelikož je to právě ona magická zkratka OOP, která je hitem dnešních dnů, rozhodli jsme se věnovat programování s objekty více prostoru, než je obvykle běžné. Do této publikace jsme proto zařadili celkem 16 praktických programátorských ukázek, které pokrývají všechny stěžejní prvky objektově orientovaného programování v jazyce C# 4.0. A přestože v jednotlivých ukázkách najdete také nezbytnou teoretickou průpravu, hlavní důraz je kladen na zcela praktickou aplikaci vybraných pilířů objektové teorie. Naší intencí je představit vám OOP jako životaschopné programátorské paradigma, jemuž jazyk C# 4.0 umožňuje vyniknout v plné kráse.

Praktická ukázka č. 1: Deklarace třídy, generování instancí třídy a manipulace s instancemi třídy

Přestože má termín třída v obecném slova smyslu více možných výkladů, programátoři znalí objektově orientované koncepce vývoje počítačového softwaru vědí, že v tomto ponímaní plní třída roli továrny na objekty. Jazyková specifikace C# 4.0 předepisuje základní elementy, jimiž musí být každá třída vybavena. V první praktické ukázce přijdeme třídám na chuť, přičemž sestavíme třídu, jejíž instance budou zobrazovat bitové mapy na obrazovce počítače.

Praktická ukázka č. 2: Modelování tříd pomocí vizuálního návrháře tříd (Class Designer)

Integrované vývojové prostředí Visual Studio 2010 disponuje zabudovaným vizuálním návrhářem tříd. Návrhář tříd je velice užitečným pomocníkem, jenž nám dovolí vytvářet třídy plně vizuální cestou – třídy jednoduše nakreslíme podobně, jako navrhujeme grafické symboly při konstrukci diagramů v modelovací aplikaci Microsoft Visio 2010. Když jsme začali s vizuálním návrhářem tříd pracovat, ani jsme nevěděli, jak báječný asistent to ve skutečnosti je. Práce s návrhářem tříd je rychlá, přehledná a přináší potěšení. Přínosy návrháře tříd jistě pocítíte i vy, a to zejména pro jeho designérské, vizualizační a refaktorizační schopnosti.

Praktická ukázka č. 3: Implementace jednoduché a úrovňové dědičnosti

Pokud bychom vyhlásili soutěž o nejproslulejší prvek objektově orientované koncepce programování, zcela jistě by se na první příčce objevila dědičnost. Dědičnost je očividně nejznámějším atributem OOP, jehož význam je někdy až démonizován. Nicméně pravdou je, že bez dědičnosti bychom si jenom sotva mohli představit plnohodnotné programování s objekty. Jazyk C# 4.0 podporuje jednoduchou dědičnost a její speciální variaci, které se říká úrovňová dědičnost (o té mluvíme tehdy, působí-li jedna třída v rámci vymezeného řetězce tříd současně jako třída bázová i odvozená). Z uvedeného vyplývá, že použití vícenásobné dědičnosti, známé zvláště z prostředí jazyka C++, není v C# 4.0 povoleno. Toto omezení ovšem není nijak tragické – právě naopak, je totiž ku prospěchu věci, neboť napomáhá přehlednějšímu a lépe spravovatelnému programovému kódu. V praktické ukázce vás provedeme procesem vývojem časomíry, na níž budeme demonstrovat použití jednoduché dědičnosti. Pak přistoupíme k úrovňové dědičnosti, jejíž pomocí vybudujeme sérii tříd uskutečňujících výpočet vybraných statistických ukazatelů.

Praktická ukázka č. 4: Instanční konstruktory

Věděli jste, že instanční konstruktor třídy jazyka C# 4.0 je v instrukcích mezijazyka MSIL reprezentován metodou s identifikátorem **.ctor**? A víte, že před provedením kódu nacházejícím se v těle instančního konstruktoru se implicitně volá konstruktor instance bázové třídy? Jestliže jste odpověděli alespoň na jednu z položených otázek záporně, máte dobrý důvod, proč nalistovat čtvrtou praktickou ukázku objektového programování v jazyce C# 4.0.

Praktická ukázka č. 5: Implicitní instanční konstruktory

Nevložíme-li do těla třídy definici instančního konstruktoru, překladač jazyka C# 4.0 pro nás vygeneruje implicitní veřejně přístupný bezparametrický instanční konstruktor. Tato skutečnost s sebou nese několik zajímavých implikací, které si blíže rozebereme v praktické ukázce.

Praktická ukázka č. 6: Instanční konstruktory a inicializace soukromých datových členů instancí tříd

Instanční konstruktor je první metoda, kterou běhové prostředí CLR volá bezprostředně po vytvoření instance třídy. Primárním účelem použití instančního konstruktoru je inicializace

soukromých datových členů zrozeného objektu. Aplikaci instančního konstruktoru si představíme na příkladu Informátora, jenž bude na požádání zjišťovat informace o předem určených datových souborech.

Praktická ukázka č. 7: Přetěžování instančních konstruktorů

Někdy je zapotřebí, aby jedna třída obsahovala více verzí instančního konstruktoru. V těchto situacích se ke slovu dostává technika přetěžování, díky níž mohou programátoři sestrojit přetížený konstruktor. O tom, jak je realizováno přetěžování konstruktoru, budete vědět víc po přečtení této praktické ukázky. Dodejme, že diferenciace definic instančního konstruktoru se uskutečňuje pomocí rozdílného počtu formálních parametrů, rozdílných datových typů formálních parametrů a použitých modifikátorů **ref** a **out**.

Praktická ukázka č. 8: Použití finalizéru

Finalizér byl vždycky považován za jakýsi „doplněk“ konstruktoru. Zatímco konstruktor měl na starosti začáteční inicializaci objektu, finalizér prováděl konečný úklid alokovaných objektových zdrojů. Finalizér jazyka C# 4.0 je však jiný než destruktor známý z C++, což je významná informace zejména pro programátory přicházející z nativního prostředí „céčka se dvěma plusy“. Ačkoliv z hlediska syntaxe nejsou modifikace takřka vůbec patrné, k zásadním změnám dochází „pod povrchem“. Finalizéry v C# 4.0 jsou vyvolávány běhovým prostředím CLR v procesu, který označujeme hrozivě znějícím názvem „nedeterministická finalizace objektů“. Po základní charakteristice finalizérů na platformě .NET Framework 4.0 uvidíte rovněž praktickou ukázku ilustrující vzájemný vztah konstruktoru a finalizéru.

Praktická ukázka č. 9: Vztah finalizéru a finalizační metody Finalize

Když programátor v C++ řekl destruktor, měl na mysli opravdu destruktor, tedy metodu, k aktivaci které dochází těsně před ukončením doby životnosti objektu. V jazyce C# 4.0 však situace není tak průzračně čistá: je to proto, že zdejší „destruktor“ se nazývá „finalizér“. Finalizér je ve skutečnosti pouze přestrojenou finalizační metodou s identifikátorem **Finalize**. Pečlivým zkoumáním vztahu finalizéru a finalizační metody se zabývá devátá praktická ukázka.

Praktická ukázka č. 10: Implicitní a explicitní dealokace objektových zdrojů

Prostředky obsazené objektem mohou být uvolněny dvěma způsoby: buď implicitně finalizérem, anebo explicitně voláním čistící metody **Dispose** rozhraní **IDisposable**. Oba dealokační scénáře se mohou vzájemně doplňovat tak, aby nedošlo k závažným konfliktům. Na příkladu volání funkcí aplikačního programového rozhraní Win32 si předvedeme, jak naprogramovat finalizér a metodu **Dispose** tak, abychom dodrželi doporučeníhodné návrhové vzory a vyhnuli se tak skrytým léčkám a pastím.

Praktická ukázka č. 11: Implicitní volání metody Dispose pomocí příkazu using

Příkaz **using** jazyka C# 4.0 představuje mocnou zbraň, neboť umožňuje programátorům definovat samostatný blok kódu, po opuštění kterého bude zcela automaticky volána metoda **Dispose** používaného objektu. Tím pádem je jisté, že objektové zdroje budou určitě uvolněny metodou **Dispose**, a to i bez její přímé aktivace. Nevyhnutným předpokladem pro úspěšnou realizaci nastíněné operace je, aby třída, z níž je objekt zrozen, implementovala rozhraní **IDisposable** a zaváděla definici metody **Dispose**. Příkaz **using** byl po dlouhou dobu konkurenční výhodou jazyka C#: kdybychom se totiž ohlédli do historie, zjistili bychom, že C# byl prvním „dotnetovým" programovacím nástrojem, jenž použití příkazu **using** povoloval. Ve verzi 2005 se k jazyku C# přidal také Visual Basic, jenž ovšem používá příkaz **Using**, jehož název je psán s velkým začátečním písmenem.

Praktická ukázka č. 12: Statické třídy

Statické třídy představují jednu z konkurenčních výhod jazyka C# 4.0, která nadchne nemálo vývojářů. Ano, je to tak: nyní můžeme opatřit deklarační příkaz třídy klíčovým slovem **static** a nemusíme se již pouštět do křížku s rádoby statickými třídami definujícími soukromé instanční konstruktory. Statické členy statických tříd lze používat bez nutnosti instanciace těchto tříd, což je zcela jistě nejvíce viditelné pozitivum statických tříd jako takových. Povídání o statických třídách bude zpestřeno ukázkou naprogramování 2D grafu s využitím dovedností grafického rozhraní GDI+.

Praktická ukázka č. 13: Aktivace členů bázové třídy z odvozené třídy

Dovolují-li to pravidla viditelnosti programových entit, je možné z těla odvozené třídy přímo volat metodu či vlastnost umístěnou v těle třídy bázové. Aktivace členů bázové třídy z podtřídy je v jazyce C# 4.0 uskutečnitelná použitím příkazu **base**. V této praktické ukázce

pochopíte, jak celý proces funguje. Mimo to se seznámíme s praktickou aplikací, která bude tentokrát ekonomického ražení. Na příkladu dvou tříd se dozvíme, jak determinovat bod zvratu a kritické využití výrobní kapacity, což jsou důležité ukazatele vypovídající o výrobně-obchodní činnosti kterékoliv produkční podnikatelské jednotky.

Praktická ukázka č. 14: Abstraktní a zapečetěné třídy

O abstraktních a zapečetěných třídách se v dávných dobách mezi vývojáři mluvilo tiše a ještě k tomu při zhasnutém světle. Těžko říct, zda pravým důvodem tohoto počínání byl nějaký programátorský rituál nebo snad až pekelná obtížnost, která byla s přípravou těchto tříd spojována. Ať tak či onak, jazyk C# 4.0 vám otevírá dveře i do této na první pohled obtížnější oblasti vývoje softwaru. O tom, že budování abstraktních a zapečetěných tříd patří do „vyšší školy programovacích technik“, není pochyb. Ovšem na druhou stranu je zapotřebí dodat, že se nejedná o nic, co by průměrně zdatnému vývojáři mělo činit nějaké větší potíže.

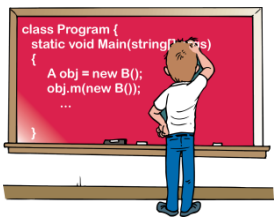
Praktická ukázka č. 15: Polymorfismus implementovaný prostřednictvím dědičnosti

Zatímco dědičnost je neznámějším pilířem objektově orientované koncepce programování, polymorfismus je pro mnohé vývojáře synonymem obrovité příšery, která je straší ve snech. Abychom napomohli odbourání této fobie, zařazujeme praktickou ukázku implementace polymorfismu pomocí veřejné jednoduché dědičnosti. Společně připravíme třídy s polymorfně se chovajícími metodami, jejichž prostřednictvím sestrojíme uživatelské nabídky s novými vizuálními styly.

Praktická ukázka č. 16: Binární serializace objektů

Výpravu do universa objektově orientovaného programování zakončíme exkurzí binární serializace objektů. Pod pojmem serializace rozumíme proces transformace paměťové reprezentace objektu do formy datového proudu, který může být uložen do souboru a posléze podroben transportu. V praktické ukáce si posvítíme na binární serializaci, která uskutečňuje hluboké otisky objektů. Nevynecháme však ani ukázku mělké serializace, jejímž výstupem budou s daty asociované instrukce značkovacího jazyka XML.

1 Praktická ukázka č. 1: Deklarace třídy, generování instancí třídy a manipulace s instancemi třídy



Cíl praktické ukázky:
Vysvětlit proces deklarace a instanciace tříd v jazyce C# 4.0.
Vědomostní náročnosť: .
Časová náročnosť: 20 minut.
Programovací jazyk: C# 4.0.
Vývojové prostředí: Microsoft Visual Studio 2010.

Programovací jazyk C# 4.0 je těsně svázán s bázovou knihovnou tříd vývojově-exekuční platformy .NET Framework 4.0, takže vývojáři mohou s výhodou využívat dovedností několikatisícového zástupu předem připravených tříd. Ačkoliv je opětovná použitelnost entit integrovaných uvnitř BCL (Base Class Library) zcela jistě ohromná, poměrně často se před programátorem vynoří požadavek na vytvoření vlastní třídy, která bude zapouzdřovat uživatelskou funkcionalitu. Jazyk C# 4.0 nám samozřejmě umožňuje deklarovat naše vlastní třídy a posléze vytvářet jejich instance. Přesněji řečeno, v úvahu připadá více postupů, jak zhotovit novou třídu. Přitom záleží jenom na vás a vašich potřebách, pro který z možných scénářů se rozhodnete.

1.1 První scénář: Implementace nové funkcionality

Jestliže potřebujete vytvořit zbrusu novou třídu, která bude plnit specifický úkol, zvolíte pravděpodobně tuto variantu. Při ní navrhnete složení třídy z hlediska požadované cílové funkcionality, obohatíte třídu o datové členy a nakonec zvolíte vhodný systém, jenž bude pracovat s daty, s nimiž třída nebo její instance přijdou do kontaktu.

Pokud pracujete v týmu, je možné, že budete muset své rozhodnutí pro navrhnutí nové třídy nejspíše konzultovat se softwarovým architektem – to proto, aby se dodržela stanovená linie a přijaté standardy, jimiž se vývoj softwarového systému řídí. Jste-li jedním kolečkem v týmu, musíte být při stavění nové třídní funkcionality mnohem obezřetnější než tehdy, když pracujete na svém individuálním projektu. Komunikace se softwarovým architektem nebo analytikem však bývá zpravidla prospěšná pro obě zúčastněné strany.

Naopak, leží-li odpovědnost za vytvoření nové třídy jenom na vás, máte volnější pozici. Ovšem ani tak byste neměli zapomínat na zlaté pravidlo projektového managementu: lépe je věnovat návrhu tříd více času a dospět k logickému a fungujícímu systému, než přidávat třídy stylem hlava nehlava a pak si neuváženě komplikovat další fáze vývojového cyklu aplikace.

Máme-li na mysli zařazení nové funkcionality prostřednictvím nové třídy, míníme tím, že nová třída nebude mít žádného explicitního předka. I takováto třída však bude disponovat svou implicitní nadtřídou, kterou je v našem případe třída **System.Object**.

1.2 Druhý scénář: Rozšiřování stávající funkcionality

Bázová knihovna třídy je vskutku bohatou studnicí, která se může pochlubit množstvím různorodých tříd, jež jsou uloženy v hierarchicky členěných jmenných prostorech. BCL zavádí účinnou podporu pro mnoho programátorských činností, datovým zpracováním počínaje a programování počítačové grafiky konče. Při uvažování o nové třídě byste proto měli vždy zvážit, zda je nutné vytvářet zcela novou třídu. Možná, že stejné, anebo dokonce i lepší výsledky můžete dosáhnout tehdy, rozhodnete-li se rozšířit již existující třídu, respektive funkcionalitu, kterou tato třída nabízí. Tento pracovní model vychází z velice jednoduchého postulátu: vezmeme to, co je již k mání, a přidáme to, co potřebujeme. Bezesporu vám nemusíme připomínat, že předestřený postup se pojí s aplikací dědičnosti a případně také s polymorfismem (chceme-li upravit chování instancí budoucí třídy). Pokud není vestavěná třída bázové knihovny výslovně deklarována jako zapečetěná (s modifikátorem **sealed**), můžete ji použít jako mateřskou třídu. Odvozená třída zdědí dovednosti svého přímého předka (nadtřídy) a také nepřímého předka (systémové třídy **Object**). Máme-li před sebou podtřídu, můžeme její funkcionalitu dále rozšiřovat přidáváním nových datových členů, metod, vlastností, událostí, operátorů a delegátů. Rovněž můžeme překrývat metody a vlastnosti mateřské třídy (a donutit je tak, aby se chovali polymorfně), či v příhodném okamžiku aktivovat přístupné členy bázové třídy a s povděkem využívat připravenou funkcionalitu.

Pokud diskutujeme o rozšiřování existující funkcionality, měli bychom poukázat na skutečnost, že tento model neimplikuje pouze jednoduché dědění, jehož cílem je sestrojení nové odvozené třídy. Společně s jazykem C# 4.0 totiž můžeme postoupit ještě o krok dále. Stačí, když třídu hezky zabalíme do uživatelsky přívětivého ovládacího prvku nebo komponenty. Posléze není nic jednoduššího než sestavený ovládací prvek nabídnout uživatelům. Vedle kreace ovládacího prvku můžeme rozšířenou funkcionalitu

přetransformovat rovněž do knihovny tříd. Za těchto podmínek sice uživatel ztrácí možnost vizuální práce s třídou, ovšem použití třídy je stále velice rychlé a práce s ní komfortní.

1.3 Deklarace třídy v jazyce C# 4.0

Jakoukoliv třídu v jazyce C# 4.0 deklarujeme pomocí příkazu **class**. Každá třída musí mít své jméno a tělo – to jsou nezbytné předpoklady pro to, abychom mohli o určitém bloku zdrojového kódu mluvit jako o třídě. Jméno třídy musí vyhovovat konvencím pro pojmenovávání programových entit jazyka C# 4.0, a proto nesmí začínat číslicí. Tělo třídy je vymezeno blokem příkazů, které jsou uzavřeny ve složených závorkách ({}), a které následují bezprostředně za hlavičkou třídy. V hlavičce deklarované třídy se mohou nacházet atributy a přístupové modifikátory. Tak atributy jako i modifikátory jsou nepovinné. Ovšem zatímco chybějící atribut nemá na funkci třídy v podstatě žádný vliv, o absentujícím přístupovém modifikátoru totéž říci nemůžeme. Není-li v deklaraci třídy uveden žádný z přípustných přístupových modifikátorů, jazyk C# 4.0 implicitně dosazuje modifikátor **internal**. To znamená, že ke třídě může přistupovat pouze zdrojový kód, jenž se nachází ve stejném sestavení řízené aplikace. Jinými slovy, třídu mohou vidět také jiné třídy, které jsou deklarovány v identickém aplikačním projektu, ovšem pro třídy umístěné v externích aplikačních projektech není naše třída přístupná. Častěji se ale setkáváme s veřejnými třídami (což jsou třídy, v jejichž hlavičkách se objevuje přístupový modifikátor **public**), na které se může odkazovat veškerý zdrojový kód situovaný v daném sestavení anebo v jakýchkoliv externích sestaveních řízených aplikací.

Poznámka: Přestože ve výbavě jazyka C# 4.0 nalezneme ještě tři přístupové modifikátory (**private**, **protected** a **protected internal**), tyto se vztahují spíše k datovým členům třídy než ke třídě samotné. Kdybychom kupříkladu třídě přiřadili soukromý modifikátor **private**, překladač by nebyl schopen kód přeložit. Potíž vězí v tom, že jazyk C# 4.0 nepřipouští aplikaci výše zmíněných tří modifikátorů ve spojení s třídou, která je součásti jistého jmenného prostoru. Po založení projektu nebo přidání souboru s třídou je implicitně vytvořen hlavní (takzvaný kořenový) jmenný prostor (obvykle s názvem aplikace) a teprve do něj je třída uložena. Možná si myslíte, že toto omezení lze odstranit vyjmutím třídy a jejím umístěním mimo hlavní jmenný prostor. Bohužel, takovéto řešení nefunguje: důvodem je, že jazyk C# 4.0 implicitně pracuje s globálním jmenným prostorem, do něhož patří všechny programové entity, s nimiž ve svém projektu pracujeme.

Povězme, že budeme chtít vytvořit třídu, jejíž instance bude schopna uchovat grafický obrázek v podobě bitové mapy. Bitmapa je virtuálním obrazem souborové struktury grafického formátu, jenž má podobu datové mřížky. Tato mřížka nese informace o jednotlivých bodech obrazu. Kvalita obrazu je přímo závislá na množství informací uložených v bitové mapě. Toto množství informací se měří v bitech, přičemž vypovídajícím atributem kvality je počet bitů na jeden obrazový bod datové mřížky (jde o známý pixel).

Naše třída se bude jmenovat **Bitmapa** a vytvoříme ji takto:

1. Založíme nový projekt standardní aplikace pro systém Windows pomocí projektové šablony **Windows Forms Application** jazyka C# 4.0.
2. Otevřeme nabídku **Project** a klepneme na příkaz **Add Class**.
3. Vybereme ikonu **Class** a aktivujeme tlačítko **Add** (chceme-li, můžeme soubor se zdrojovým kódem třídy pojmenovat, no pro naše potřeby si vystačíme s implicitně zvoleným názvem).
4. Vygenerovaný zdrojový kód upravíme podle níže uvedeného vzoru. Ten představuje syntaktický skelet třídy **BitovaMapa**.

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace kniha_ppcs40_pu_01
{
    // Deklarace třídy, jejíž instance dovedou pracovat s bitovými mapami.
    public class BitovaMapa
    {
        // Definice soukromých datových členů třídy.
        private string cestaKBitmape;
        private Bitmap bitmapa;

        // Definice veřejného instančního bezparametrického konstruktoru.
        public BitovaMapa()
        {
            cestaKBitmape = Environment.CurrentDirectory + @"\Západ slunce.bmp";
            NacistBitovouMapu(cestaKBitmape);
        }
        // Definice veřejné instanční parametrické metody, která provede
        // načtení bitové mapy z fyzického souboru.
```

```
        public void NacistBitovouMapu(string cesta)
        {
            cestaKBitmape = cesta;
            bitmapa = new Bitmap(cesta);
        }
        // Definice veřejné instanční metody, která zobrazí bitovou mapu
        // na formuláři.
        public void ZobrazitBitovouMapu()
        {
             Form formular = Form.ActiveForm;
             Graphics grafickyObjekt = formular.CreateGraphics();
             grafickyObjekt.DrawImage(bitmapa, 10, 10, 200, 150);
             grafickyObjekt.Dispose();
        }
    }
}
```

Verbální shrnutí zdrojového kódu jazyka C# 4.0: Jak si můžeme všimnout, třída **BitovaMapa** je veřejně přístupná a je umístěna ve kořenovém jmenném prostoru. V těle třídy se nachází celkem pět členů. Z toho jde ve dvou případech o datové členy (**cestaKBitmape** a **bitmapa**), dále o jeden konstruktor (má stejný název jako třída samotná) a nakonec o dvojici metod (**NacistBitovouMapu** a **ZobrazitBitovouMapu**). Názvy členů dávají tušit, že naše třída se bude soustředit na načítávání a zobrazování bitových map.

Abychom mohli bitovou mapu načíst a posléze prezentovat na obrazovce počítače, musíme mít po ruce několik artefaktů. Především je to fyzický soubor s rastrovými daty, který je obvykle uložen někde na pevném disku, případně na jiném externím záznamovém médiu. Na tento soubor se můžeme kdykoliv odkázat prostřednictvím absolutní nebo relativní cesty popisující jeho pozici. Pro uchování cesty k souboru slouží v naší třídě datový člen s identifikátorem **cestaKBitmape**. Protože cesta k souboru je ve skutečnosti ztělesňována pouze řetězcem textových znaků, jež jsou propojeny speciálními symboly, roli datového člena **cestaKBitmape** hravě sehraje proměnná primitivního odkazového datového typu **string**.

Dobrá, když víme, kde grafický soubor leží, máme napůl vyhráno. Abychom mohli data z fyzického souboru načíst do operační paměti počítače, potřebujeme instanci třídy **Bitmap** ze jmenného prostoru **System.Drawing**. Samozřejmě, data z grafického souboru by bylo možné načíst také manuálně v rámci vstupní operace s využitím dovedností datových proudů. Tato alternativa je však mnohem složitější, a proto jsme raději poprosili o pomoc třídu **Bitmap**, s jejíž instancí je práce daleko rychlejší a přímočařejší. Soukromá odkazová

proměnná **bitmapa** reprezentuje druhý datový člen třídy **BitovaMapa**. Tuto proměnnou využijeme v procesu instanciace třídy **System.Drawing.Bitmap**, přičemž odkaz na vzniklou instanci uložíme právě do datového členu **bitmapa**.

Datový člen **cestaKBitmape** explicitně inicializujeme v instančním konstruktoru třídy. Jak je známo, konstruktor je speciální metoda, která je automaticky volána poté, co je zrozena instance třídy. Primárním účelem konstruktoru je inicializace všech datových členů instance třídy. Podobně se konstruktor chová i v případě naší třídě. V těle konstruktoru je do odkazové proměnné **cestaKBitmape** přiřazena cesta ke grafickému souboru (s příponou .bmp). Konkrétně, do zmíněného datového členu bude uložena cesta ke grafickému souboru, jenž je umístěn ve stejné složce jako spustitelný soubor řízené aplikace.

Poznámka: Na jakékoliv soubory, jež jsou umístěny ve stejné složce jako spustitelný soubor řízené aplikace, se lze dotazovat pomocí vlastnosti **CurrentDirectory** třídy **Environment** ze jmenného prostoru **System**. Třída **Environment** je však užitečným pomocníkem i za jiných okolností, třeba když chceme zjistit jméno aktuálně přihlášeného uživatele či velikost pracovní sady (což je množství virtuální operační paměti mapované na právě běžící proces). Druhou variantu demonstruje následující fragment zdrojového kódu jazyka C# 4.0:

```
static void Main(string[] args)
{
    // Zjištění velikosti pracovní sady běžícího procesu pomocí
    // vlastnosti WorkingSet třídy Environment.
    long pracovniSada = System.Environment.WorkingSet;
    Console.WriteLine("Velikost pracovní sady pro tento proces je " +
        (pracovniSada / (1024 * 1024)) + " MB.");
}
```

Výsledným efektem této ukázky je vypočtení celočíselné hodnoty, která představuje přibližné množství alokované operační paměti pro aktuálně spuštěný proces. Když jsme program spustili na našem testovacím počítači, dozvěděli jsme se, že pracovní sada je 6 MB.

Co dále provádí konstruktor? Inu, volá metodu **NacistBitovouMapu**, která zabezpečuje načtení obrazových dat z fyzického rastrového souboru do objektu třídy **Bitmap**. V signatuře metody se nachází jeden formální parametr typu **string**, jenž je připraven přijmout absolutní cestu k cílovému souboru s bitmapou. Poté, co je JIT-překladačem zpracován kód metody **NacistBitovouMapu**, máme k dispozici paměťový otisk obrazových dat, které jsou uchovány

v souboru s extenzí .bmp. Zdůrazněme, že jednotlivé pixely jsou prozatím pouze v paměti a nikoliv na obrazovce počítače.

Abychom data převedli do oku lahodící podoby, zavoláme si na pomoc další metodu, jejíž název je **ZobrazitBitovouMapu**. Metoda má jediný úkol: přenést obrazová data z paměti na obrazovku a namapovat je na plochu aktivního formuláře. Vykreslovací operace se uskutečňuje za přispění grafického aplikačního rozhraní GDI+, přičemž měrnými jednotkami souřadnicového systému jsou pixely. Obrazová data naneseme na formulář pomocí grafického objektu **Graphics** (tento objekt je instancí stejnojmenné třídy, jež je deklarována ve jmenném prostoru **System.Drawing**). Grafický objekt je vždy spojen s jistým kontextem – v našem případě jako kontext vystupuje plocha aktivního formuláře (k němu získáme přístup přes vlastnost **ActiveForm** třídy **Form**). Instanci třídy **Graphics** získáme voláním metody **CreateGraphics**.

Po zrození grafického objektu voláme jeho instanční metodu **DrawImage**, které předáváme odkaz na soubor s bitovou mapou. A jelikož bychom rádi obrázek vykreslili na námi zvolené pozici, specifikujeme ještě další argumenty:

- souřadnice levého horního bodu obdélníkové oblasti, v níž bude obrázek umístěn,
- šířku obrázku,
- výšku obrázku.

Když metoda **DrawImage** dokončí svoji práci, na ploše formuláře se budou vyjímat všechny pixely vybrané bitmapy. Vzápětí po finalizaci vykreslovací operace již grafický objekt nebudeme potřebovat, a proto alokované grafické zdroje uvolňujeme explicitní aktivací spřízněné metody **Dispose**.

Jakmile máme napsaný zdrojový kód třídy **BitovaMapa**, můžeme na formulář aplikace přidat jedno tlačítko (instanci ovládacího prvku **Button**) a do zpracovatele jeho události **Click** vložit programové instrukce pro vytvoření instance naší třídy.

```
namespace kniha_ppcs40_pu_01
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
```