Speciál pro programátory

Visual Basic .NET a Managed Extensions for C++

Časová náročnost (min):	Začátečník	Pokročilý	Profesionál
65			

Použitý operační systém :	Windows 2000 SP3
Hlavní vývojový nástroj :	Visual Studio .NET 2002
Další vývojový software :	Žádný
Jiný software :	Žádný

Milí čtenáři,

rád bych vás přivítal u nového Speciálu pro programátory, který rozšiřuje nabídku tohoto vydání rubriky Visual Basic. Programátorský speciál plní v našem případě úlohu „zásuvného modulu", jenž je dalším stupínkem v procesu zkvalitňování prostředí a obsahu rubriky Visual Basic. Náplň speciálu pro programátory bude ovšem poněkud širší, než na jakou jste byli doposud zvyklí z jiných částí rubriky. Znamená to, že v speciálech se budou objevovat mnohá složitější témata, dokonce se střetnete i s tématy, které nebyly ještě nikdy uveřejněny. Ačkoliv bude náročnost probírané problematiky vyšší, nemusíte mít obavy, že byste ji snad nezvládli. Nadále bude dodržena již standardní linie, kterou můžete spatřit ve všech součástích rubriky Visual Basic, a která se vyznačuje interaktivním výkladem doprovázeným značným množstvem grafických ilustrací, obrázků a skic. Prozatím se ovšem nepočítá, že by programátorský speciál byl součástí každého vydání rubriky Visual Basic, nicméně můžete si být jisti, že se s ním setkáte při speciálních příležitostech.
Výborně, organizační záležitosti jsme vyřídili, a teď se již můžeme plně soustředit na dnešní téma, kterým je jazyková interoperabilita mezi jazyky Visual Basic .NET a Managed Extensions for C++.

Obsah

Managed Extensions for C++ a Visual Basic .NET
Jazyk Managed Extensions for C++ a Visual C++ .NET
Vytváření knihovny DLL v Managed Extensions for C++
Vytváření podtřídy ve Visual Basicu .NET
Testování jazykové interoperability

Managed Extensions for C++ a Visual Basic .NET

Předtím, než se vrhneme na samotné programování, si budeme muset říct pár slov o významu a postavení programovacího jazyka Managed Extensions for C++. Ano, i když se to možná na první pohled nezdá, můžeme o Managed Extensions for C++ mluvit jako o novém programovacím jazyce. Managed Extensions for C++ představují „řízená rozšíření" jazyka C++, protože právě pomocí těchto rozšíření mohou programátoři v C++ psát řízené (**managed**)

aplikace pro cílovou platformu .NET Framework. Řízené aplikace disponují následujícími znaky:

1. Programový kód řízených aplikací je uložen v podobě kódu jazyka **Microsoft Intermediate Language (MSIL)**, což je objektově orientovaný nízkoúrovňový programovací jazyk, instrukcím kterého rozumí společné běhové prostředí (**Common Language Runtime**, **CLR**). Kód jazyka MSIL ovšem nemůže být podroben přímé exekuci, a proto je nutné jej přeložit do odpovídajícího strojového kódu (přesněji kódu, jemuž rozumí instrukční sada daného typu CPU počítače). Převod MSIL kódu je realizován za běhu programu pomocí **Just-In-Time** (**JIT**) kompilátoru (pomineme-li možnost vytvoření tzv. nativního obrazu assembly, při které je převeden MSIL kód assembly do podoby nativního programového kódu).

2. MSIL kód je společně s metadaty uložen v jedné assembly. Jak je známo, assembly je základní aplikačně-logická jednotka platformy .NET a její velikou výhodou je skutečnost, že je samopopisná. Assembly může být složena s jednoho nebo i několika modulů. Podobně, data assembly mohou být uložena v jednom či více souborech, avšak jeden soubor assembly vždy obsahuje manifest, jenž poskytuje informace o dané assembly. Jak již bylo řečeno, assembly je samopopisná, a tudíž není problémem její přenositelnost mezi různými cílovými prostředími (v tomto okamžiku ovšem pouze v rámci operačních systémů Windows 98, Windows 98 Druhé Vydání, Windows Millenium Edition, Windows NT 4, Windows 2000, Windows XP verze Home a Professional plus operační systémy z rodiny Windows .NET Server). Assembly není závislá od systémového registru, a budete-li chtít uskutečnit její rychlou portaci na jinou počítačovou stanici, můžete použít příkaz XCOPY (v tomto případě ovšem musí být .NET Framework nainstalován na cílovém počítači). Pro pokročilejší potřeby můžete samozřejmě použít instalaci pomocí instalační služby Windows Installer. Assembly s metadaty rovněž přináší nové světlo do problematiky verzí (zjednodušeně by se dalo říci, že jsou již pryč časy s nekompatibilními verzemi různých externích součástí, na nichž byla aplikace závislá).

3. Managed Extensions for C++ vyhovují specifikacím společného typového systému (**Common Type System**) a společné jazykové specifikace (**Common Language Specification**) platformy .NET (těmto specifikacím a pravidlům samozřejmě vyhovují i další jazyky platformy .NET, tedy Visual Basic .NET, Visual C# .NET a Visual J# .NET). Tím je zaručena vzájemná typová kompatibilita, která pohání myšlenku úzké spolupráce všech programovacích jazyků, které operují pod křídly .NET. Jazykové interoperabilitě napomáhá také uložení výstupů jednotlivých kompilátorů do jednotné formy, kterou představuje již zmíněný Microsoft Intermediate Language. Právě bezproblémová spolupráce různých programovacích jazyků na platformě .NET je metou, do které vkládá Microsoft velké naděje a prostředky. Pokud jste někdy programovali v jazycích C nebo C++, jistě víte, že spolupráce těchto jazyků s Visual Basicem nebyla nikdy ideální. Například při tvorbě dynamicky linkovaných knihoven ve Visual C++ jsme museli často

exportovat pouze funkce jazyka C. Dále bylo nutné použít správnou volací konvenci (**__stdcall**) a rovněž vhodně určit datové typy formálních parametrů funkcí a jejich návratových hodnot (pokud existovaly). A samotná tvorba definičních souborů celý postup také zrovna neusnadňovala. Pamatujete-li tyto doby, můžete vzpomínky na ně již hodit za hlavu. Jazyková interoperabilita v podání .NET Frameworku je, jedním slovem, pohádka. Typová kompatibilita odstraňuje potíže s rozličným rozsahem datových typů v různých jazycích. Můžete tedy přímo používat systémové datové typy .NET Frameworku, např. **System.Boolean**, **System.Int32**, **System.Object** a další. Každý jazyk, či už je to Visual Basic .NET nebo Managed Extensions for C++, obsahuje vlastní datové typy, které odkazují na systémové typy .NET Frameworku. Pokud ve Visual Basicu .NET pracujete s proměnnými typu **Integer**, ve skutečnosti pro uchovávaní celých čísel nepoužíváte typ **Integer**, ale systémový typ **System.Int32**. Klíčové slovo **Integer** tedy ve Visual Basicu .NET slouží pouze jako odkaz (neboli alias), jenž je nasměrován na příslušný systémový datový typ. Budete-li programovat v Managed Extensions for C++, pro uchování celých čísel z intervalu <-2 147 483 648, 2 147 483 647> můžete také použít systémový datový typ **System.Int32**. Tak jako Visual Basic .NET, i Managed Extensions for C++ obsahují alias, pomocí kterého se můžete odkázat na typ **System.Int32**. Tímto aliasem je klíčové slovo **int** (resp. **__int32**).

V Managed Extensions for C++ dochází při práci s celočíselnými proměnnými k poněkud nepřehledné situaci, protože pro jeden systémový typ (**System.Int32**) se zde nacházejí až tři aliasy (**int**, **__int32** a **long**). Aliasy **int** a **__int32** jsou ekvivalentní, a proto se budeme soustředit pouze na komparaci odkazů **int** a **long**. I když oba odkazují na společný systémový datový typ **System.Int32**, při použití aliasu **long** bude kompilátor generovat speciální modifikátor **Microsoft.VisualC.IsLongModifier**, čímž bude naznačeno, že byl použit alias **long**. I když aliasy **int** a **long** odkazují na systémový datový typ se stejným rozsahem platných hodnot, mohou se tyto aliasy stát rozlišovacím prvkem při vytváření přetížených funkcí.

Protože se v různých jazycích mohou vyskytovat různé aliasy pro stejné systémové datové typy, mnoho programátorů se přiklání ke koncepci využívaní pouze systémových datových typů. Pokud se chcete zabývat jazykovou interoperabilitou hlouběji, měli byste tuto skutečnost vzít na vědomí.

Skutečně snadná je spolupráce mezi jazyky v oblasti objektově orientovaného programování – základní implementaci třídy můžete vytvořit v jednom programovacím jazyku (třeba v Managed Extensions for C++) a v dalším (např. ve Visual Basicu .NET) odvodit ze základní třídy podtřídu. Když jsme se dostali k OOP koncepci, je třeba zdůraznit, že .NET Framework podporuje pouze jednoduchou dědičnost (tedy jednostranný vztah mezi supertřídou a množinou odvozených tříd). Ano, odvozené třídy mohou mít jenom jednoho přímého předka, ovšem odvozené třídy mohou implementovat jedno, nebo i několik rozhraní. Jste-li zvyknutí na vícenásobnou dědičnost z nativního C++, je možné, že budete trochu zklamaní, ovšem zařazení jednoduché dědičnosti je na platformě .NET jenom ke prospěchu věci (dochází

k odstranění chyb při špatném návrhu vztahu mezi rodičovskými třídami a jejich potomky).

4. Řízené aplikace jsou pod kontrolou společného běhového prostředí (**Common Language Runtime**). **CLR** nabízí řízeným aplikacím mnoho dodatečných služeb, mezi které patří kupříkladu automatické správa paměti prostřednictvím **Garbage Collection** (**GC**), která pomáhá programátorům spravovat vytvořené objekty a rovněž podporuje automatickou likvidaci neužitečných objektů (programátor se tedy již nemusí starat o explicitní likvidaci objektů a následnou dealokaci alokovaného paměťového prostoru). Je evidentní, že právě automatická správa paměti je velikou devizou jazyka Managed Extensions for C++. V nativním, nebo také neřízeném (**unmanaged**) C++, je životní cyklus objektů plně v rukou programátora. To znamená, že sám programátor je zodpovědný za řádné zrození a posléze i za příslušnou likvidaci objektů. V prostředí Managed Extensions for C++ stačí, když programátor jenom vytvoří objekt, o jeho destrukci se již nemusí starat (o tu se postará **Garbage Collection**). Automatická správa zabezpečuje, že již nikdy nedojde k závažným programátorským chybám, při kterých byl objekt zrušen příliš brzo, nebo došlo k pokusu o opětovnou likvidaci již odstraněného objektu. Na druhou stranu, programátoři v nativním C++ si mohou být jisti deterministickou finalizací objektů, tedy skutečností, že cílový objekt je zrušen okamžitě poté, co mu tento příkaz odevzdá aplikační logika programu. Naproti tomu, **CLR** ve spolupráci s **Garbage Collection** poskytují nedeterministickou finalizaci, která znemožňuje likvidaci objektu v přesně stanovený okamžik. Ve většině případů je uvedené chování **Garbage Collection** přijatelné, ovšem jisté potíže nastávají ve chvíli, kdy je likvidace objektu spjata s jinou událostí.

Jazyk Managed Extensions for C++ a Visual C++ .NET

Abychom předešli terminologickým nejasnostem a potížím, musíme si povědět, jaká je souvislost mezi pojmy Managed Extensions for C++ a Visual C++ .NET. Tak předně Visual C++ .NET představuje vývojové prostředí pro vytváření rozmanitého spektra aplikací. Pomocí Visual C++ .NET můžete programovat tři základní typy aplikací:

1. Neřízené (**unmanaged**) aplikace použitím jazyka C a Win32 API.
2. Neřízené (**unmanaged**) aplikace pomocí nativního jazyka C++ a knihovny MFC.
3. Řízené (**managed**) aplikace prostřednictvím Managed Extensions for C++.

Z uvedeného výčtu je zřejmé, že Managed Extensions for C++ jsou rozšířeními jazyka C++ pro vytváření aplikací pro **CLR** a .NET Framework. Na druhé straně ovšem nelze Managed Extensions for C++ chápat jako jenom pouhé rozšíření, protože jde vskutku o nový programovací jazyk s mnoha novými konstrukcemi a prvky, které se v prostředí nativního C++ nenachází (jde např. o podporu delegátů, nové koncepce pro implementaci vlastností, práci s řízenými typy, automatickou správu paměti, výjimky a další). Visual C++ .NET je, jako momentálně jediné vývojové prostředí, schopné generovat jak řízený, tak i neřízený

programový kód. Tato skutečnost ovšem nabývá na významu teprve při použití Managed Extensions for C++, protože v jednom projektu můžete libovolně míchat kód nativního C++ společně s řízenými rozšířeními. Kromě použití nových programovacích prvků se mnoho změn odehrává na pozadí, při práci s pamětí (nejmarkantnější změny souvisejí s kontrolou životních cyklů objektů). Objekty v řízených aplikacích jsou ukládány do vyhrazené paměťové oblasti, která je označovaná jako řízená hromada (**managed heap**), zatímco objekty v neřízených aplikacích jsou uchovávány na neřízené hromadě (**C++ heap**). Rozdíl mezi těmito dvěmi hromadami spočívá v tom, že řízená hromada je pod správou **Garbage Collection**, ovšem neřízená hromada není kontrolována žádnou podobnou entitou. Pokud bychom se podívali na správu paměti blíže, zjistili bychom, že situace je poněkud složitější. Objekty, které se nacházejí na řízené hromadě jsou rozděleny do tří generací: generace 0, 1 a 2. V nulté generaci jsou umístěny „nejmladší" objekty, tedy objekty, které byly vytvořené naposledy a u kterých se nepředpokládá, že doba jejich existence bude příliš dlouhá. Samozřejmě, paměť, která tvoří nultou generaci řízené hromady, není nekonečná (ve skutečnosti je její velikost přibližně 2^8 KB). V okamžiku, kdy dojde k alokaci veškeré paměti, je spuštěn úklid paměti pomocí **Garbage Collection**. **Garbage Collection** zjistí, zdali v nulté generaci nejsou zastoupeny objekty, které už nejsou potřeba (**GC** při hledání již nepotřebných objektů aplikuje speciální mechanizmus, který využívá strom odkazů). Najdou-li se takovéto objekty, jsou z paměti uvolněny. Pochopitelně, ne všechny objekty bude možné z paměti odstranit, některé totiž budou stále aktivní, a tyto nelze podrobit destrukci. **Garbage Collection** rozhodne, které objekty jsou pořád zapotřebí a tyto přesune do generace 1. Výsledkem práce jednoho cyklu **Garbage Collection** je tedy uvolnění místa v generaci 0 a přenos aktivních objektů do generace 1. Tím však celá mašinérie ještě nekončí, protože je nutné také aktualizovat ukazatele, které směrují na jednotlivé objekty (je totiž nezbytně nutné, aby i po uskutečnění paměťového úklidu bylo možné získat přístup k objektům pomocí referenčních proměnných). Při přemísťování objektů je brán ohled na kompaktnost této činnosti, protože je velmi důležité, aby objekty byly snadno a co možné nejrychleji přístupné. A co ostatní generace? Generace 1 obsahuje objekty se střední dobou životnosti a generace 2 pak sdružuje dlouho žijící objekty, kterých využití nemusí být příliš frekventované, ovšem je nutné, aby tyto objekty byly k dispozici po dlouhou dobu. Když po úklidu generace 0 není získán dodatečný paměťový prostor, uskuteční se průzkum generace 1 a naleznou se objekty, které mohou být zlikvidovány. Pokud stále není dostatek prostoru, **Garbage Collection** „přeskenuje" také generaci 2 a zjistí, zdali nemůže být uvolněn jistý paměťový prostor právě z této generace. Ve většině případů ovšem situace nezachází až do tohoto extrémního bodu. Důvodem je samotný charakter činnosti aplikací, přičemž mnoho aplikací využívá intenzivně pouze nultou generaci (aplikace v poměrně krátkém sledu vytvářejí objekty a tyto následně podléhají likvidaci).

Vytváření knihovny DLL v Managed Extensions for C++

Jazykovou interoperabilitu mezi jazyky Visual Basic .NET a Managed Extensions for C++ si předvedeme na vytvoření knihovny DLL v Managed Extensions for C++. Postupujte takto:

1. Spusťte Visual Studio .NET 2002 a vytvořte Visual C++ .NET projekt typu **Managed C++ Class Library**.

2. V okně **Solution Explorer** poklepejte na položku s hlavičkovým souborem Jméno_projektu.h. Otevře se okno editoru pro zápis programového kódu.
3. Veškerý kód hlavičkového souboru upravte podle níže uvedeného vzoru:

```
// interop_02.h
//Direktiva preprocesoru #pragma once.
#pragma once

//Pouziti direktivy #using pro import metadat z uvedenych assembly.
#using <System.dll>
#using <System.Drawing.dll>
#using <System.Windows.Forms.dll>

//Pouziti direktivy using pro import potrebnych jmennych prostoru.
using namespace System;
using namespace System::Drawing;
using namespace System::Drawing::Drawing2D;
using namespace System::Windows::Forms;

//Hlavni jmenny prostor projektu.
namespace Interop_01
{
//Deklaracni prikaz verejne __gc tridy s nazvem Kresleni.
      public __gc class Kresleni
            {
//Definice verejne bezparametricke funkce ZacitKreslit.
            public:
                  void ZacitKreslit(void)
                  {
                  //Deklarace referencnich promennych f a g.
                  Form  __gc * f;
                  Graphics __gc * g;

                  //Prirazeni odkazu na aktivni formular do referencni
                  //promenne f (je pouzita skalarni staticka
//vlastnost ActiveForm tridy Form).
                  f = Form::ActiveForm;

                  //Vytvoreni grafickeho objektu pomoci metody CreateGraphics.
                  g = f->CreateGraphics();

                  //Vytvoreni instance struktury System::Drawing::Rectangle.
                  System::Drawing::Rectangle rec;

                  //Modifikace vlastnosti instance struktury.
                  rec.set_Location(Point(0,0));
                  rec.Width = f->Width;
                  rec.Height = f->Height;

//Vytvoreni instance tridy System::Drawing::Drawing2D::LinearGradientBrush.
                  LinearGradientBrush __gc * sb = new
                        LinearGradientBrush(rec, Color::Orange,
                        Color::Magenta, LinearGradientMode::ForwardDiagonal);

//Metoda FillRectangle grafickeho objektu zabezpeci vyplneni obdelnikoveho
//regionu gradientni vyplni.
                  g->FillRectangle(sb, rec);
```

Programový kód pokračuje na následující straně

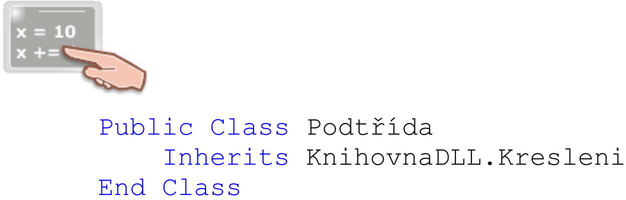
```
//Uvolneni zdroju, ktere byly alokovany grafickym objektem.
                  g->Dispose();
                  }
            };
}
```

Tento programový kód ukazuje třídu **Kresleni**, která obsahuje jednu veřejnou členskou funkci s názvem **ZacitKresleni**. V deklaračním příkazu třídy se nachází klíčové slovo **__gc**, které naznačuje, že jde o řízenou (**garbage-collected**) třídu. Instance řízené třídy budou po vytvoření umístěné na řízenou hromadu. Členská funkce **ZacitKreslit** je bezparametrická a rovněž nevrací ani návratovou hodnotu (obě skutečnosti jsou výslovně naznačeny použitím klíčového slova **void**). V těle funkce jsou vytvořeny dvě referenční proměnné **f** a **g**. Jak si můžete všimnout, jde o referenční proměnné, do kterých budou později uloženy ukazatele na příslušné instance. Všimněte si, že také obě odkazové proměnné jsou deklarované pomocí klíčového slova **__gc**. Pokud se před referenční proměnnou nachází klíčové slovo **__gc**, znamená to, že tato proměnná může uchovávat odkaz (ukazatel) jenom na řízenou instanci. Po pravdě řečeno, použití klíčového slova **__gc** není v deklaračním příkazu povinné, protože i když jej neuvedete, bude vytvořena řízená referenční proměnná. Pro programátory ve Visual Basicu .NET je důležitá jedna zpráva: K členům instancí referenčního typu (jakými jsou např. třídy) se přistupuje pomocí operátoru **->**, zatímco k členům instancí hodnotových typů (jakými jsou např. struktury) lze přistupovat prostřednictvím tečkového operátoru (**.**). Pokud bude někdy v budoucnu zavolána členská funkce **ZacitKreslit** instance třídy **Kresleni**, získá přístup k právě aktivní instanci třídy **Form** a plochu této instance vyplní grafickým štětcem s lineární výplní. Ještě předtím, než budete pokračovat, proveďte sestavení projektu (nabídka **Build**, příkaz **Build Solution**). V další části přidáme do stávajícího řešení projekt Visual Basicu .NET a od právě vytvořené třídy odvodíme ve Visual Basicu .NET podtřídu. Více se však dozvíte až na následujících řádcích.

Vytváření podtřídy ve Visual Basicu .NET

Postupujte podle těchto instrukcí:

1. Vyberte nabídku **File**, ukažte na položku **Add Project** a klepněte na položku **New Project**.
2. Vyberte projekt Visual Basicu .NET pro standardní aplikaci pro Windows (**Windows Application**) a klepněte na tlačítko OK.
3. Do vytvořeného projektu přidejte soubor s kódem třídy. Postupujte tak, že na název projektu Visual Basicu .NET v okně **Solution Explorer** klepněte pravým tlačítkem myši. Po zobrazení kontextové nabídky ukažte na položku **Add** a vyberete položku **Add Class** (soubor s kódem třídy můžete nechat implicitně pojmenovaný).
4. Ještě na chvíli zůstaňte v okně **Solution Explorer**. Klepněte pravým tlačítkem myši na uzel **References** a z kontextového menu vyberte příkaz **Add Reference**. V dialogu s titulkem **Add Reference** aktivujte tlačítko **Browse** a vyhledejte soubor knihovny DLL, kterou jste vytvořili ve Visual C++ .NET. Poté klikněte na tlačítko OK.
5. Kód třídy **Class1** upravte následovně:

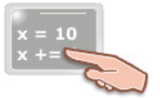


```
Public Class Podtřída
    Inherits KnihovnaDLL.Kresleni
End Class
```

Programový kód třídy s názvem **Podtřída** je vskutku jednoduchý. Třída obsahuje jenom jeden řádek s příkazem **Inherits**, za kterým následuje název jmenného prostoru a třídy, ze které má být odvozená **Podtřída**. Protože jazyková interoperabilita pracuje na nízké úrovni (na bázi kódu MSIL), není nutné provádět jakékoliv jiné operace.

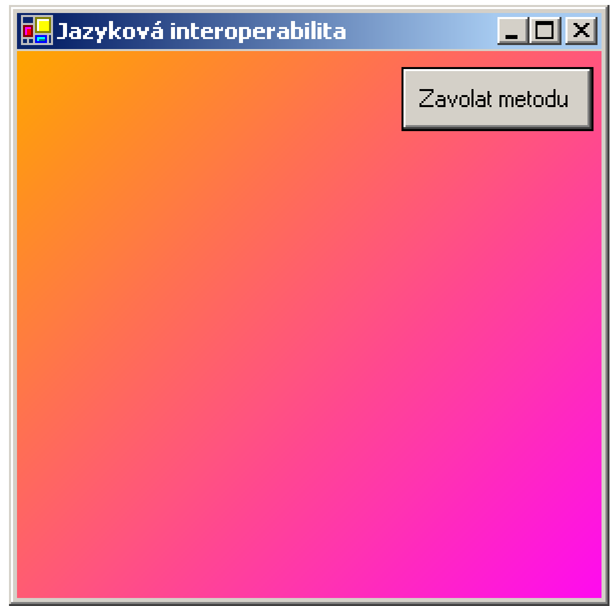
Testování jazykové interoperability

Na formulář projektu Visual Basicu .NET přidejte jednu instanci ovládacího prvku **Button**. Na vytvořenou instanci poklepejte a do její událostní procedury **Click** vložte tyto řádky kódu:



```
Dim x As New KnihovnaDLL.Kresleni()
x.ZacitKreslit()
```

Předtím, než budeme moci jazykovou interoperabilitu vyzkoušet v praxi, musíme určit, který projekt našeho řešení se bude spouštět jako výchozí. To uděláte tak, že v okně **Solution Explorer** klepnete pravým tlačítkem myši na název projektu Visual Basicu .NET a zvolíte položku **Set as StartUp Project**. Stiskněte klávesu F5 a po spuštění aplikace klepněte na tlačítko. V tuto chvíli bude vytvořena instance třídy **Podtřída** a zavolána metoda **ZacitKreslit**. Jazyková interoperabilita bude provedena bez jakýchkoliv potíží, a vy uvidíte, jak se zabarví plocha formuláře (obr. 1).



Obr. 1 – Jazyková interoperabilita v praxi