Elektronická příručka pro programátory
pracující s programovacími jazyky
Visual Basic 6.0 a Visual Basic .NET

Souhrn všech vydání rubriky Visual Basic za rok 2003

Úvod

Vážení čtenáři,

do rukou se vám dostává elektronická podoba všech vydání rubriky Visual Basic za rok 2003. Tato elektronická příručka je určena zejména programátorům pracujícím s programovacími jazyky Visual Basic 6.0 a Visual Basic .NET (verze 2002). Hlavním podnětem pro vytvoření příručky bylo podání obrovského kvanta informací ze světa programování v jednotní a hlavně uživatelsky přívětivé podobě. Tak vznikl PDF dokument, který právě čtete. Rozsah elektronické příručky je vskutku pozoruhodný – přesahuje totiž 300 stran a z tohoto pohledu se může směle měřit s jakoukoliv tištěnou a na programování ve Visual Basicu orientovanou publikaci.

Když Visual Basic vzešel z dílny společnosti Microsoft, psal se rok 1991. Snad nikdo, ani samotní tvůrci tohoto programovacího jazyka, neměli představu o tom, jakým způsobem se bude k programování počítačových aplikací přistupovat za více než deset let později. Ano, Visual Basic byl vždy tím elementem, který se hrdě hlásil ke svému přídomku „Visual" a který jako první přinesl opravdové vizuální programování pro platformu Windows. Je samozřejmé, že prvotní návrh a implementace programovacího jazyka nebyly zrovna ideální. Jak plynul čas, byl jazyk obohacován o nové programovací konstrukce a dovednosti. Jednou z přelomových verzí jazyka byla verze 4.0, uvedená na softwarový trh v roce 1995. Právě „čtyřka" přinesla mnoho revolučních prvků, mezi něž můžeme zařadit třeba vývoj aplikací pro plně 32bitové prostředí operačního systému Windows, implementaci hlavních prvků objektově orientovaného programování, funkční automatizaci aplikací a mnohé další. S dalšími verzemi přicházely další vylepšení: vývoj ActiveX prvků, zařazení kompilátoru nativního kódu, lepší podpora OOP, zdokonalené integrované vývojové prostředí... Po pravdě řečeno, Visual Basic již urazil hodně velký kus své cesty k dokonalosti. Toto tvrzení dokazuje rovněž zatím poslední verze jazyka, honosně označována jako Visual Basic .NET. Vize platformy .NET a „všeho, co k ní patří", sehrává v strategii Microsoftu značně důležitou roli. S příchodem platformy .NET Framework byly uvedeny, kromě Visual Basicu .NET, také další programovací jazyky, které plně vyhovují specifikacím nového rámce pro vývoj aplikací pro Windows a web.

Visual Basic .NET je nyní velice dobře připraven na to, aby vám mohl poskytnout interaktivní a uživatelsky přívětivé prostředí pro vývoj vašich aplikací. Nabízí plnou podporu všech prvků OOP včetně dědičnosti a práce s konstruktory a destruktory. A protože ve verzi .NET pracuje Visual Basic pod křídly platformy .NET, aplikace v něm vytvořené jsou stejně výkonné jako jejich protějšky napsané v C# či v řízeném C++. Ano, Visual Basic v dnešní době nepředstavuje pouze programovací jazyk, nýbrž životní styl.

Ať se tedy Visual Basic .NET stane součástí vašeho životního stylu!

Přeji vám mnoho příjemných chvil při listování virtuálními listy této elektronické příručky.

Ján Hanák

O autorovi

Autor této elektronické příručky se zabývá programováním a vývojem aplikací pro platformu **.NET Framework** společnosti Microsoft použitím programovacích jazyků **Visual Basic .NET**, **Visual C# .NET** a **Visual C++ .NET**. Na CD/DVD příloze odborného počítačového magazínu CHIP vede každý měsíc dvě programátorské rubriky (**Visual Basic** a **C#**). Kromě programování se zabývá také plánováním, návrhem, vývojem a implementací progresivních dokumentačních systémů, pracujících na bázi technologií **Microsoft WinHelp**, **Microsoft HTML Help** a **Microsoft Help 2.0**. Ve volných chvílích se zabývá programování umělé inteligence počítačových pomocníků pracujích pod křídly technologie **Microsoft Agent 2.0** a studuje trojrozměrnou grafiku a aplikační programové rozhraní **DirectX 9**.
Autor je k zastižení na adrese hanja@stonline.sk.

Obsah

Integrované prostředí rubriky Visual Basic	Autor: **Ján Hanák**

Téma měsíce

Programování ve více jazycích

Časová náročnost (min): **120**	Začátečník	Pokročilý	Profesionál
			

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

Vážení čtenáři,

hlavním námětem pro dnešek je problematika „míchání" programovacích jazyků. V následujících kapitolách se pokusíme přijít na to, jak může být pro programátora přínosná práce ve dvou (nebo i více) programovacích jazycích. Dále se budeme soustředit na praktickou demonstraci filozofie programování ve více jazycích a vytvoříme dynamicky linkovanou knihovnu (DLL) v prostředí Visual C++ a posléze využijeme možností vytvořené knihovny ve Visual Basicu. Věřím, že vás námět dnešní diskuse zaujal, a proto neztrácejme čas a ponořme se hlouběji do problematiky.

Obsah

Filozofie programování ve více jazycích

Možná, že s uvedeným termínem „míchání programovacích jazyků" (angl. **Mixed Language Programming**) se střetáváte poprvé, možná, že jste o něm již „něco" slyšeli, no stále nevíte, o co ve skutečnosti jde. Pojďme se tedy podívat, jak se tato myšlenka dostala do světa, ve kterém vládnou optimalizované algoritmy a počítačové instrukce. Jak jistě víte, v počítačovém světe existuje poměrně početná množina programovacích jazyků, od těch nejstarších (ALGOL, COBOL, FORTRAN), přes C a BASIC až po nejnovější instance, jakými jsou Visual Basic .NET, Visual C# .NET nebo Visual C++ .NET. Každý programovací jazyk, který se nachází na určitém vývojovém stupni, byl v počátcích své tvorby koncipován jako pomůcka pro řešení jistých problémových úkolů. Zjednodušeně se dá říct, že každý programovací jazyk se na svou podstatu existence (tedy řešení úkolů, pro které byl stvořen) dívá z vlastního pohledu. Důkazem tohoto tvrzení jsou někdy zcela specifické postupy, které musí programátor ve dvou jazycích uskutečnit, aby vyřešil stejnou úlohu (nejjednodušším příkladem může být zapsání cyklu **For** ve Visual Basicu a C++). I když je každý

programovací jazyk specifický, všechny jazyky disponují mnoha „standardními" konstrukcemi (v analogii s předchozím příkladem, oba jazyky, Visual Basic i Visual C++ obsahují cyklus **For**). Rozvineme-li myšlenku, že každý programovací jazyk byl primárně určen na řešení jistého okruhu problémů, můžeme dojít k poznání, že jeden jazyk může být vhodnější na řešení jisté skupiny problémů, zatímco jiný jazyk si zase dobře poradí s jinou, specifickou množinou úkolů. Jestliže popojedeme v našich úvahách ještě o krok dále, objevíme samotnou myšlenku programování ve více jazycích. Posuďte sami: Proč používat na vyřešení jistého problému jazyk, ve kterém je toto řešení když ne nemožné, tak alespoň hodně obtížné? Nebylo by lepší jednoduše napsat onu kritickou část programu v jiném jazyku, který si s naším problémem umí lépe poradit?

Použití teorie míchání programovacích jazyků může přinést mnoho výhod:

Úspora času	Otázka rozřešení programátorského problému, se kterým nevíme v prostředí jednoho jazyka „ani hnout", se v jiném jazyce může stát otázkou několika málo okamžiků.
Zlepšení výkonu	Napsání kritických částí programu v jiném, nižším, programovacím jazyku může způsobit citelné zlepšení výkonu aplikace.
Zlepšení rychlosti	S předchozím bodem se úzce váže i zlepšení rychlosti programu. V případě, že napíšeme optimalizovaný kód nižší úrovně, a ten pak využijeme ve stávající aplikaci, zlepšení rychlosti může být signifikantní.
Znovupoužitelnost programového kódu	Pokud jsme schopni napsat nízkoúrovňový kód, který později „zabalíme" do komponenty pro další použití, nemusíme již tuto část kódu psát opět později. Jednoduše využijeme to, co již máme připravené.

Pro úplnost výkladu je zapotřebí vzpomenout i některé stinné stránky této teorie programování.

Vědomostní úroveň	Jde o úroveň znalostí a schopností, kterou musí programátor disponovat. Ne každý programátor totiž umí dobře programovat ve více jazycích. V této souvislosti není možno zapomenout na prvotní zvýšené náklady na zvýšení znalostní úrovně programátora. Může jít např. o absolvování specializovaného programátorského kurzu, nákup literatury a pod.
Nutnost vlastnit další programovací nástroj	Pokud se rozhodnete pro programování ve více jazycích, budete pravděpodobně muset investovat do nákupu dalšího vývojového nástroje, cena kterého může značně ovlivnit váš rozpočet pro vyvíjenou aplikaci.

V dnešní případové studii budeme společně řešit jeden z nejčastějších úkolů programátora pracujícího v prostředí dvou programovacích jazyků. Půjde o vytvoření dynamicky linkované knihovny (DLL) ve Visual C++ a následně její implementace do aplikace vyvíjené ve Visual Basicu.

Pokud náhodou patříte mezi skupinu programátorů VB, kterým se při vyslovení názvů „C/C++", případně „Visual C++" udělá špatně, nemusíte se již více obávat a můžete zůstat zcela klidní. V tomto článku totiž nepředpokládám, že máte nějaké zkušenosti jak s jazykem C/C++, tak s vývojovým prostředím Visual C++. V celém článku se uplatňuje ověřená technika postupu stylem „krok za krokem", proto se nemusíte obávat jakýchkoliv nejasností. Uvidíte, že po přečtení následujících řádků budete i vy schopni vytvořit jednoduchou DLL ve Visual C++.

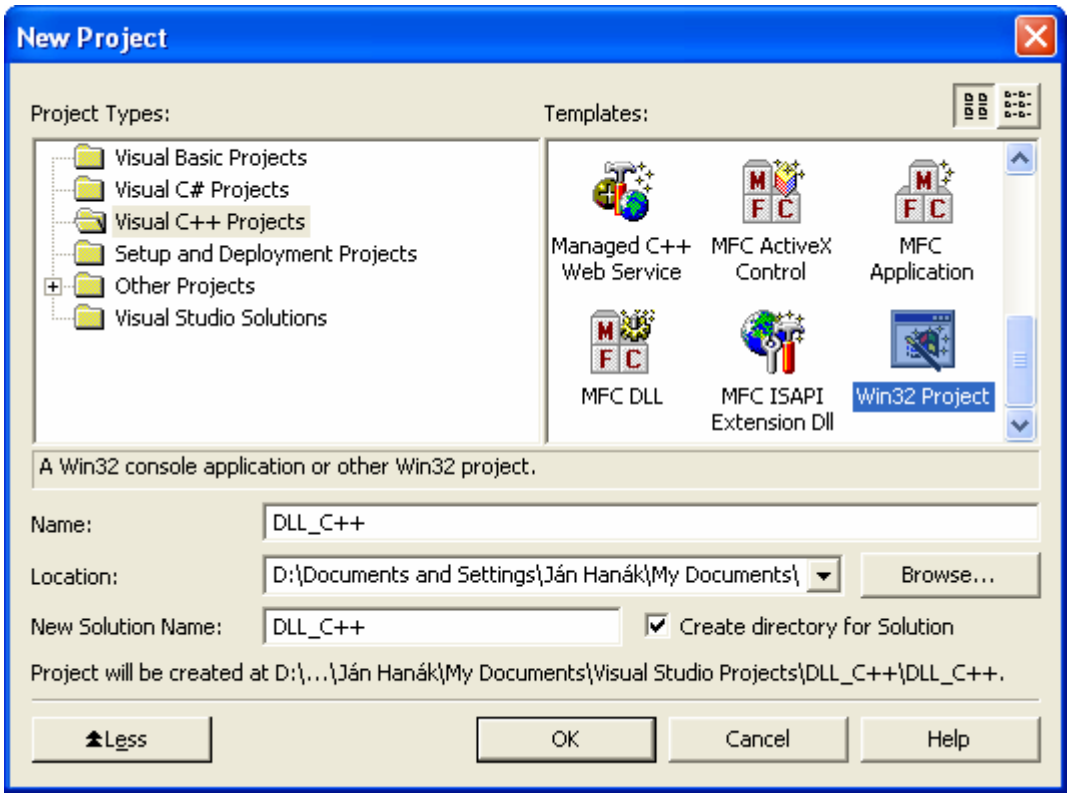
Visual Basic .NET a Visual C++ .NET
Krok 1: Tvorba dynamicky linkované knihovny (DLL) ve Visual C++ .NET

V následující ukázce vytvoříme standardní dynamicky linkovou knihovnu typu **Win32**, která bude obsahovat jednu funkci pro výpočet třetí mocniny čísla. Postupujte takto:

1. Spusťte Visual Studio .NET a z nabídky **File** vyberte položku **New** a pak **Project**.

Stejného účinku můžete docílit i klepnutím na ikonu **New Project** na standardním panelu ikon, nebo klepnutím na tlačítko **New Project** v okně **Start Page**. Pokud okno **Start Page** po spuštění Visual Studia není viditelné, klepněte na nabídku **Help** a aktivujte položku **Show Start Page**.

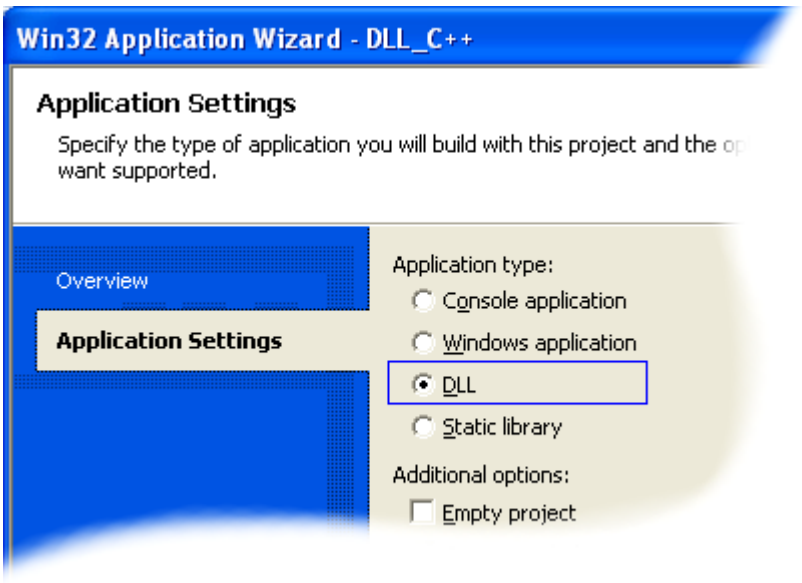
2. Okno **New Project** upravte podle obr. 1.



Obr. 1 – Okno pro výběr nového typu projektu

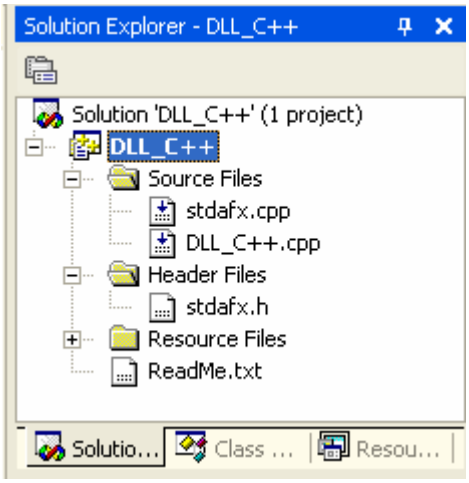
V levé části okna, v sekci **Project Types**, vyberte jako typ projektu **Visual C++ Projects**. V pravé části okna, v sekci **Templates**, vyberte položku **Win32 Project**. V textovém poli **Name** zadejte jméno pro váš nový projekt a zaškrtněte volbu **Create directory for Solution**, čímž zabezpečíte vytvoření samostatné složky pro vaše řešení. Když jste se změnami spokojeni, klepněte na tlačítko **OK**.

3. Visual C++ okamžitě zobrazí průvodce s názvem **Win32 Application Wizard**.
4. Klepněte na položku **Application Settings**. V sekci **Application Type** vyberte možnost **DLL** (obr. 2) a nakonec klikněte na tlačítko **Finish**.



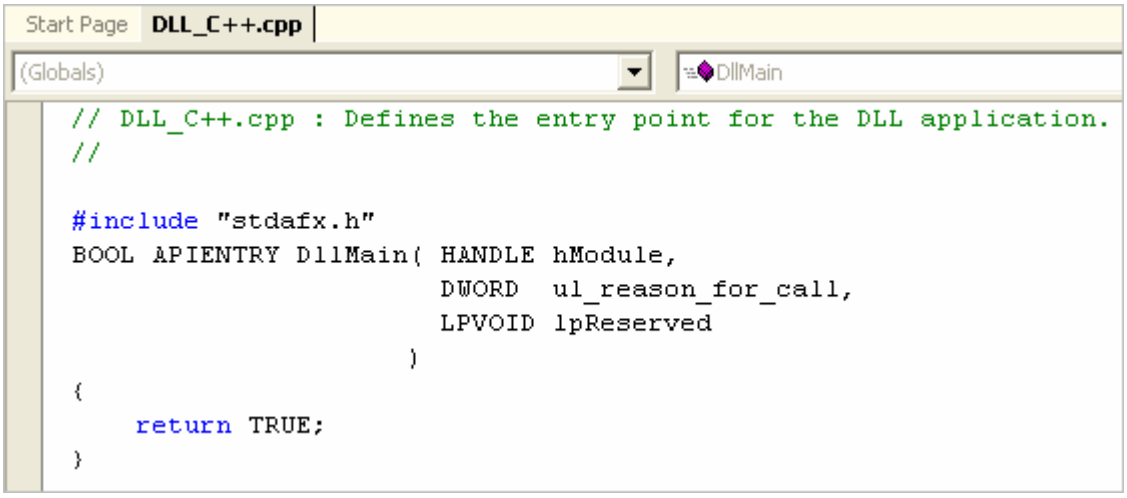
Obr. 2 – Výběr typu aplikace v průvodci

Visual C++ posléze vytvoří všechny nezbytné soubory, které si aplikace typu **Win32 DLL** vyžaduje. Seznam těchto souborů si můžete prohlédnout v Průzkumníkovi řešení (**Solution Explorer**). Okno Průzkumníka řešení by se mělo nacházet v pravé horní části okna Visual C++ a jeho podoba je zachycena na obr. 3.



Obr. 3 – Okno **Solution Explorer**

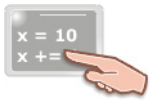
Nyní poklepejte na položku **DLL_C++.cpp**, která se nachází ve stromové struktuře pod uzlem **Source Files**. Visual C++ následně otevře vybraný zdrojový soubor projektu, kterého obsah pro vás zobrazí v okně pro zápis programového kódu (obr. 4).



Obr. 4 – Okno pro zápis programového kódu ve Visual C++

V tomto okně si můžete všimnout více věcí, které se vám (jako programátorovi VB), můžou zdát neznámé a podivné. Pojďme tedy pěkně popořádku a obsah okna si vysvětleme. Nuže, první dva řádky představují komentáře, které do kódu umístil samotný Visual C++. Zapamatujte si, že každý komentář (ve stylu jazyka C++) je uveden dvěma lomítky, které jsou symbolem pro jednořádkové komentáře. Znamená to tedy, že když uvedete v kódu dvě lomítka a za nimi text, tento je později kompilátorem chápán jako komentář a je zcela ignorován.

Další nejasností je pravděpodobně tento řádek:




```
#include "stdafx.h"
```

První část výrazu **#include** je takzvaná **direktiva preprocesoru**. Ta „říká" preprocesoru, aby soubor s názvem stdafx.h začlenil do programového kódu projektu. Řečeno jinými slovy, když preprocesor při své práci „narazí" na direktivu **#include**, vloží obsah souboru, který tato direktiva specifikuje, do samotného programového kódu vyvíjené aplikace. Soubor stdafx.h, jenž tvoří druhou část výrazu, je tzv. **hlavičkovým** souborem.

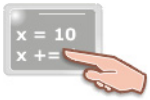
Preprocesor je typem textového procesoru, který jistým způsobem modifikuje zdrojový text programu ještě před samotnou kompilací (ve skutečnosti se preprocesor používá jenom v první etapě kompilace programu).

Následuje definice funkce **DllMain**, která je vstupním bodem (**entry point**) do dynamické knihovny. Tuto funkci volá samotný operační systém při inicializaci a terminaci procesů a vláken. Protože v naší ukázce nebude chtít vykonávat žádné operace při uvedených činnostech, můžeme ponechat funkci **DllMain** v její původní podobě.

Ve všeobecnosti je dynamicky linkovaná knihovna naplněna funkcemi, které je možné z jejího prostředí později volat. Abyste do stávajícího kódu přidali také kód pro naší exportovanou funkci, udělejte následovní:

1. Umístěte kurzor myši pod všechen programový kód (ještě přesněji pod řádek s uzavírací složenou závorkou).

2. Zde zadejte tento fragment kódu:



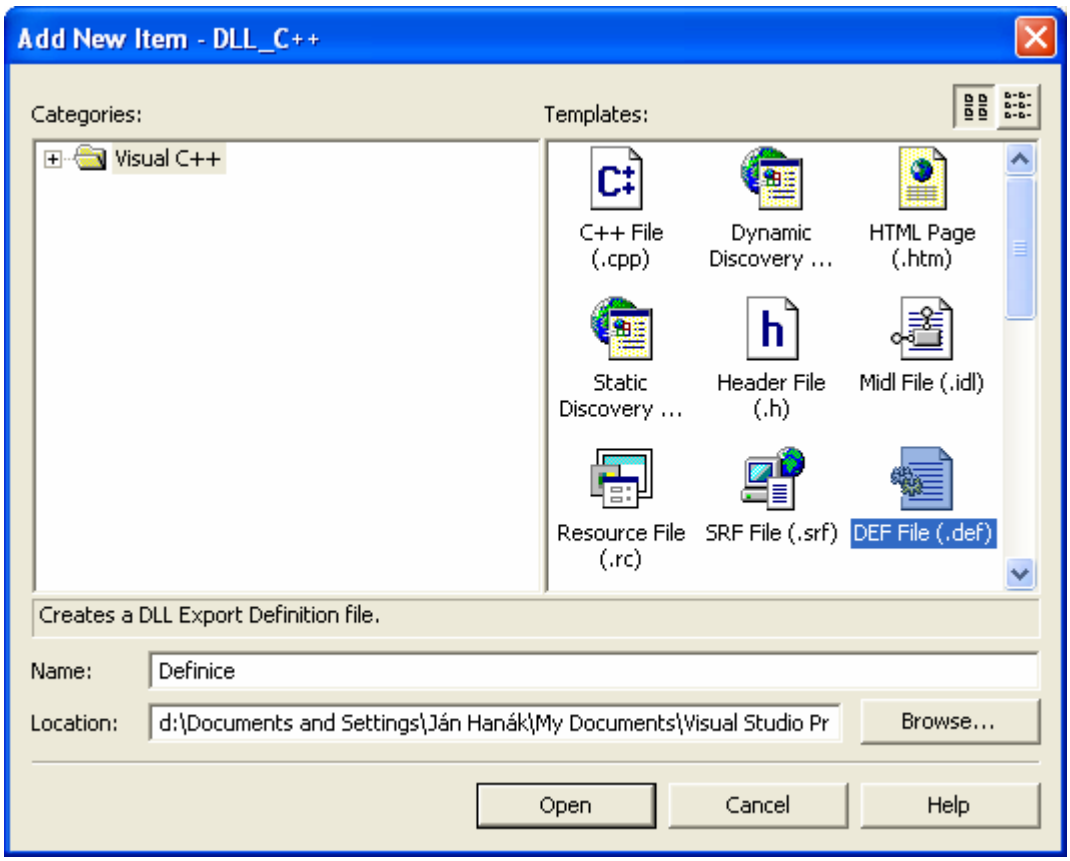

```
extern "C" __declspec(dllexport) __stdcall int Mocnina(short cislo)
{
return cislo*cislo*cislo;
}
```

Toto je zápis pro exportovanou funkci. Pod pojmem „exportovaná funkce" se rozumí funkce, která bude v budoucnosti volána z programovacího jazyka Visual Basic .NET. Aby bylo možné tuto funkci zavolat z programu Visual Basicu .NET, je zapotřebí uvedená definice funkce. Pokud odhlédneme od použití modifikátorů extern „C", __declspec(dllexport) a volací konvence __stdcall, dostáváme se k samotné funkci. Její název je **Mocnina** a jak je vidět, funkce má jeden parametr, kterým je proměnná typu short s názvem **cislo**.

V jazyce C++ se proměnné deklarují jinak než je tomu ve Visual Basicu. Zde je zapotřebí nejdříve uvést datový typ proměnné a až poté samotné jméno proměnné.

Protože jde o parametr funkce, je nutno jej uzavřít do kulatých závorek. Návratová hodnota funkce je specifikována před jménem funkce; z uvedeného příkladu vidíme, že půjde o návratovou hodnotou typu **Integer**, jak specifikuje klíčové slovo int. Tělo každé funkce v prostředí jazyka C/C++ je uzavřeno do složených závorek. Tělem funkce se rozumí příkazy, které se mají provést tehdy, je-li funkce zavolána. V našem případě si můžete všimnout, že v těle funkce je jenom jediný příkaz. Tento příkaz vypočte třetí mocninu čísla, kterou funkce při dokončení své práce vrací. Všechno, co následuje za klíčovým slovem return představuje návratovou hodnotu funkce. Konečně, každý příkaz v jazyku C/C++ je ukončen nikoliv koncem řádku, nýbrž středníkem.

3. Aby bylo později možné volat exportovanou funkci z Visual Basicu, je nutné do projektu s DLL knihovnou začlenit tzv. **definiční soubor** (.DEF), který explicitně popisuje seznam funkcí určených na export. Vyberte tedy nabídku **Project** a klepněte na položku **Add New Item**. Objeví se dialogové okno, ve kterém označte položku **DEF File** (obr. 5). Po zapsání jména souboru stiskněte tlačítko **Open**.



Obr. 5 – Výběr definičního souboru pro export funkce z knihovny DLL

4. Visual C++ ihned vytvoří a zobrazí definiční soubor. V souboru by se měl nacházet jeden řádek, ve kterém je zapsáno požadované slovo **LIBRARY**, které je následováno jménem dynamické knihovny.
5. Do definičního souboru přidejte další řádek, ve kterém pomocí klíčového slova **EXPORTS** určíte funkce, které se mají exportovat (jinými slovy, které mají být „viditelné" z externího programu). V našem případě jde o jedinou funkci s názvem **Mocnina**. Výslednou podobu definičního souboru můžete vidět na obr. 6.

Při zápisu jména funkce mějte na paměti, že Visual C++ je „case sensitive", tedy rozlišuje malá a velká písmena ve slovech. Pokud byste zadali místo řetězce **Mocnina** řetězec **mocnina**, jednalo by se o zcela jinou funkci.

Obr. 6 – Výslední podoba souboru DEF

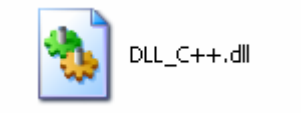
Všimněte si hvězdičky, která se nachází v záložce za jménem DEF souboru na obr. 6. Tímto způsobem vás Visual C++ upozorňuje na skutečnost, že obsah souboru se změnil od té doby, co jste tento soubor naposled uložili.

6. Vyberte nabídku **File** a ukažte na příkaz **Save All**, nebo klikněte na stejnojmennou ikonu na panelu tlačítek.

7. Dále aktivujte nabídku **Build** a vyberte příkaz **Build Solution**, anebo opět klepněte na příslušné tlačítko.

Po vydání příkazu na sestavení aplikace Visual C++ zkompiluje všechny nezbytné soubory a vybuduje jedinou dynamickou knihovnu. Jestliže vše proběhlo tak, jak mělo, ve spodním panelu uvidíte text „Build succeeded". Výborně, dokázali jste vytvořit svou první dynamicky linkovanou knihovnu!

Podíváte-li se do složky vašich projektů a otevřete složku **Debug** právě vytvořeného projektu, můžete vidět knihovnu DLL v celé její kráse tak, jak ji znázorňuje obr. 7.



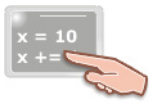
Obr. 7 – Vytvořená dynamicky linkovaná knihovna (DLL)

Abyste nemuseli později zadávat celou cestu k vaší knihovně DLL, zkopírujte ji do systémové složky Windows (\Windows\System).

Visual Basic .NET a Visual C++ .NET
Krok 2: Využití dynamické knihovny ve Visual Basicu .NET

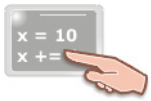
Pro předvedení možností knihovny DLL v prostředí Visual Basicu .NET nejdříve přidáme do stávajícího projektu projektovou šablonu pro standardní aplikaci pro Windows. Postupujte takto:

1. Zvolte nabídku **File**, příkaz **New** a pak **Project**.
2. V okně pro výběr projektu vyberte projekty VB (**Visual Basic Projects**) a jako konkrétní typ projektu vyberte **Windows Application**. Dále vyberte volbu **Add to Solution**, čímž přidáte projekt VB k již stávajícímu projektu Visual C++. Nakonec nový projekt pojmenujte a klepněte na tlačítko **OK**.
3. Visual Basic .NET vytvoří všechny nezbytné soubory pro nový projekt, který následně přidá do již vytvořeného řešení.
4. Přidejte do projektu Visual Basicu standardní modul. Uděláte to tak, že vyberete nabídku **Project** a klikněte na položku **Add New Item**. V objevivším se okně vyberte ikonu modulu a jeho vytvoření potvrďte stisknutím tlačítka **OK**. Do projektu VB se automaticky přidá modul. Poklepejte na ikonu nového modulu v Průzkumníkovi řešení, čímž si vyžádáte otevření okna pro zápis kódu. Do modulu umístěte deklaraci exportované funkce z naší DLL:




```
Public Declare Function Mocnina Lib "DLL_C++.dll" _
(ByVal cislo As Short) As Integer
```

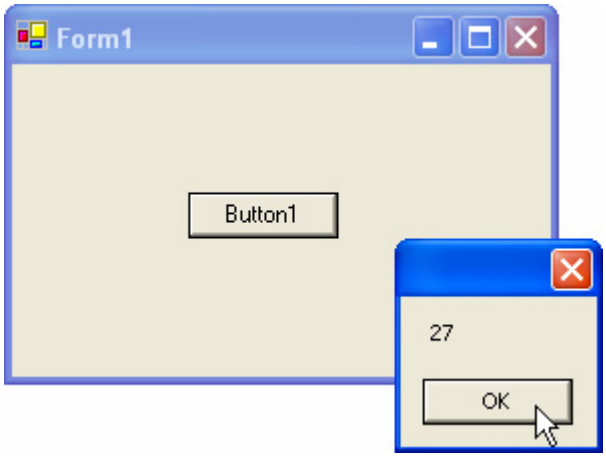
5. Pokračujte tím, že na formulář přidáte jednu instanci ovládacího prvku **Button**. Na vytvořenou instanci poklepejte, čímž otevřete okno pro zápis programového kódu. Do obsluhy události **Click** tlačítka doplňte tento řádek kódu:




```
MessageBox.Show(Mocnina(3).ToString)
```

6. Uložte projekt VB klepnutím na tlačítko **Save All**.
7. V Průzkumníkovi řešení klepněte pravým tlačítkem myši na název projektu VB a z kontextové nabídky vyberte položku **Set as Startup Project**.
8. Klikněte na tlačítko **Start**, případně stiskněte klávesu **F5** pro spuštění projektu.

Jakmile klepnete na tlačítko, VB .NET zavolá exportovanou funkci s názvem **Mocnina**, která vypočte třetí mocninu čísla 3 a návratovou hodnotu zobrazí v okně se zprávou (obr. 8).

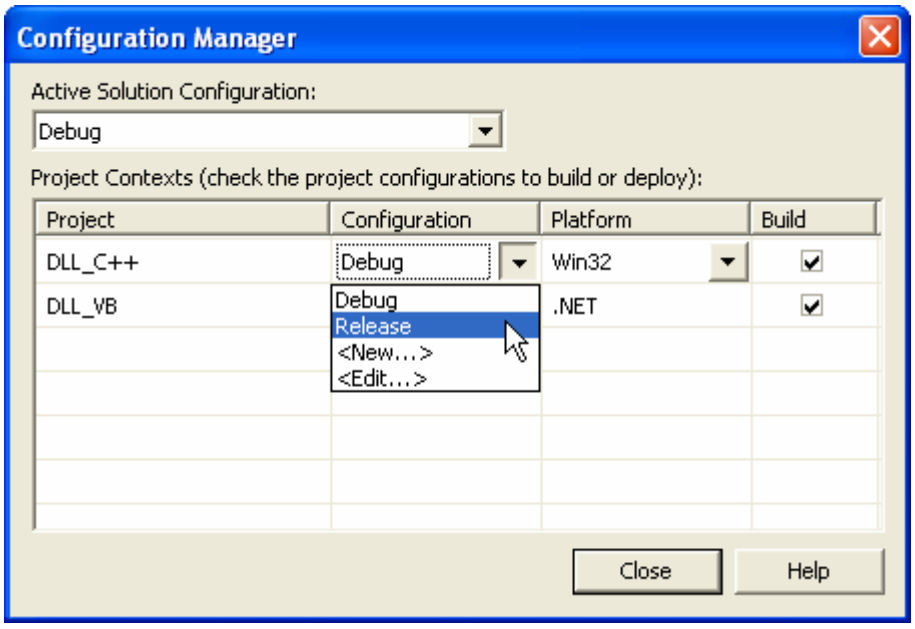


Obr. 8 – Finální podoba spolupráce aplikace VB a knihovny DLL

Výborně, dynamicky linkovaná knihovna odvádí svou práci na jedničku!

Abyste zmenšili velikost výslední knihovny DLL, můžete ji také sestavit v distribučním provedení. V tomto provedení se kód plně optimalizuje a rovněž se z něj odstraní informace, které jsou určeny pouze pro proces odlaďování knihovny. V prostředí Visual C++ udělejte následovní:

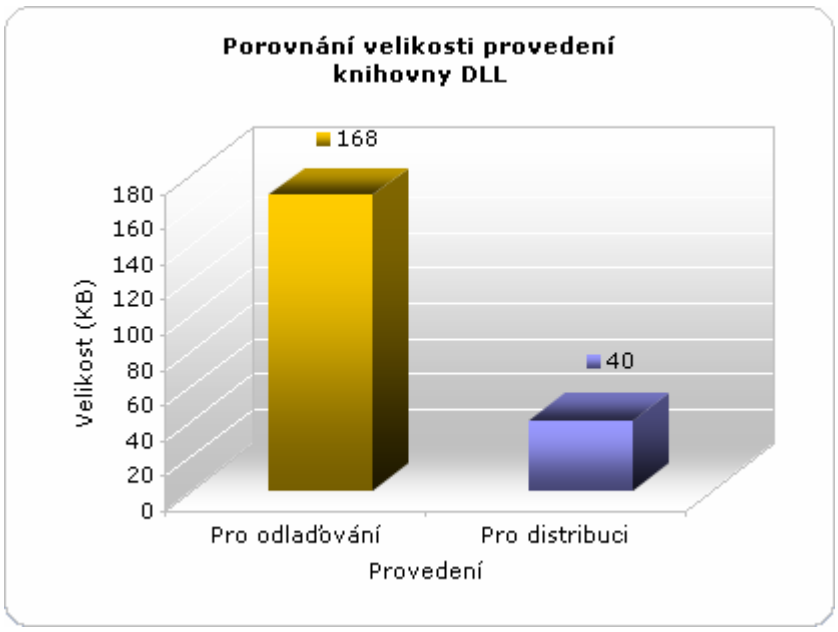
1. Z nabídky **Build** vyberte příkaz **Configuration Manager**. Objeví se dialogové okno, ve kterém určíte, aby se knihovna DLL kompilovala v provedení, které bude určeno pro distribuci. Z otevíracího seznamu v sloupci **Configuration** vyberte položku **Release** (obr. 9).



Obr. 9 – Výběr distribučního provedení knihovny DLL v okně **Configuration Manager**

2. Klepněte na tlačítko **Close** a z nabídky **Build** vyberte položku **Build DLL_C++**.

Visual C++ uskuteční sestavení dynamicky linkované knihovny a ve stávající složce projektu vytvoří další složku s názvem **Release**. Ve složce **Release** se nachází knihovna DLL, připravená na distribuční proces. Abyste měli lepší představu o velikosti knihovny DLL, podívejte se na obr. 10, který znázorňuje velikost knihovny DLL určené pro odlaďování a pro distribuci.



Obr. 10 – Porovnání velikosti variant provedení knihovny DLL

Integrované prostředí rubriky Visual Basic	Autor: **Ján Hanák**

Začínáme s VB .NET

Úvod do světa .NET (1. díl)

Časová náročnost (min):	Začátečník	Pokročilý	Profesionál
30			

Použitý operační systém :	Windows XP
Hlavní vývojový nástroj :	Visual Basic .NET 2002
Další vývojový software :	Žádný
Jiný software :	Žádný

Milí čtenáři,

vítám vás u zbrusu nového seriálu, který se bude zabývat momentálně nejmodernějším nástrojem pro tvorbu .NET aplikací, a sice Visual Basicem .NET. Seriál je určen zejména pro začátečníky, no věřím, že bude užitečný rovněž pro pokročilé uživatele a také softwarové profesionály, kteří budou chtít povznést formu vývoje svých aplikací na .NET úroveň. Společně tak prozkoumáme všechny „zákoutí" nového programovacího jazyka a už teď vás mohu ujistit, že o zábavu budeme mít vystaráno. Vysvětlíme si použití jazyka, způsob sestavování .NET aplikací, použití ovládacích prvků, základy objektově orientovaného programování a spoustu dalšího. Cílem seriálu je naučit i naprosté začátečníky budovat, používat a udržovat programové aplikace pro platformu Windows. Pokud jste ještě neprogramovali ve Visual Basicu, případně neprogramovali vůbec, nevěšte hlavu. I vám se totiž otvírají dveře do kouzelného světa .NET technologií. Stačí jenom vykročit...

Obsah

Co budete potřebovat

Kromě chuti naučit se něco nového a ochoty věnovat studiu i trochu času budete rozhodně potřebovat ty správné nástroje k vaší práci. Především byste měli disponovat poměrně výkonným počítačem a odpovídajícím softwarovým zázemím. Co se týče hardwaru, společnost Microsoft uvádí tuto minimální počítačovou konfiguraci, na které by měl Visual Basic .NET jakž takž pracovat. Co tedy budete potřebovat?

- 450 megahertzový procesor třídy Pentium II nebo lepší,
- 160 MB operační paměti RAM,
- 3,5 GB volného místa na instalačním disku a dalších 500 MB volného místa na systémovém disku,
- mechaniku CD-ROM, případně DVD-ROM,
- a obrazovkové rozlišení alespoň 800x600 obrazových bodů při 8bitové hloubce barev.

I když se v mnoha materiálech uvádí tato konfigurace jako poměrně dostačující, není tomu zcela tak. Pro praktické využití je nutno počítat s daleko silnější „mašinou". Osobně bych vám doporučil

alespoň gigahertzový procesor a nejméně 256 megabajtů operační paměti. Velkou výhodu získáte, jestliže vlastníte kvalitní monitor s úhlopříčkou minimálně 17 palců a rozlišením 1024x768 obrazovkových bodů při 32bitové barevné hloubce. Protože VB .NET obsahuje velké množství oken, podoken, vysouvacích oken a ohromné množství dalších nástrojů, které budete chtít mít na obrazovce, investice do větší plochy monitoru se zde zcela jasně vyplatí.

Po prozkoumání hardwarové části se pojďme podívat, jaké jsou nároky nového Visual Basicu na softwarové zabezpečení. Aby mohl VB dýchat, musí být podepřen operačním systémem Windows 2000 nebo XP. I když první zkušebné verze běhaly i pod Win9x, ostrá verze je v tomto směru náročnější a není je tedy možné instalovat pod starší typy operačního systému Windows.

Pomalu, ale jistě se dostáváme k představení samotného programovacího nástroje. Visual Basic .NET můžete získat v těchto vyhotoveních:

- Visual Basic .NET Standard,
- Visual Basic .NET Professional,
- Visual Basic .NET Enterprise Developer
- a Visual Basic .NET Enterprise Architect.

Pro úplnost je zapotřebí dodat, že zkušební (trial) verzi Visual Basicu .NET, případně i Visual Studia .NET lze získat jako přílohu k zahraniční literatuře, která se zabývá programováním v uvedených nástrojích. Literatura tohoto druhu se ovšem vyznačuje poněkud vysokou cenou, která se může vyšplhat až na několik tisíc korun.

Dobrá, technické specifikace máme za sebou a teď se můžeme podívat, na čem stojí celá .NET technologie.

První kroky s .NET Framework

Veškerá funkcionalita technologie .NET leží na pověstném základném kameni, který je známý jako .NET Framework. .NET Framework představuje prostředí pro vývoj a běh aplikací, přičemž do jeho působnosti patří takřka všechny operace, jež jsou s vývojem a exekucí programového kódu spjaty (alokace a dealokace paměti, provádění programových instrukcí atd.) Prostředí rámce .NET Framework je tvořené dvěma hlavními součástmi: **Common Language Runtime** a **Base Class Library**. Podívejme se, co se za těmito záhadnými názvy skrývá.

Common Language Runtime (CLR)

CLR je zodpovědná za skutečný běh aplikací .NET. Zabezpečuje vše potřebné: kompilaci kódu, jeho běh, alokaci paměti, zprávu procesů a vláken. **CLR** s sebou přináší velmi zajímavou koncepci pro kompilaci a následnou exekuci programového kódu. Pokud jste pracovali s předchozí verzí Visual Basicu, víte, že aplikaci jste mohli přeložit jak do tzv. P-kódu, tak i do nativního (strojového) kódu počítače. Zcela jiná je v tomto směru situace v prostředí .NET Framework. Když budete kompilovat aplikaci .NET, kompilátor nebude překládat výsledný kód aplikace do nativního formátu. Místo toho je všechen kód standardně zkompilován do určité jazykové mezivrstvy, které se říká **Microsoft Intermediate Language**, zkráceně **MSIL** (někdy se můžete střetnout i se zkratkou **IL**).

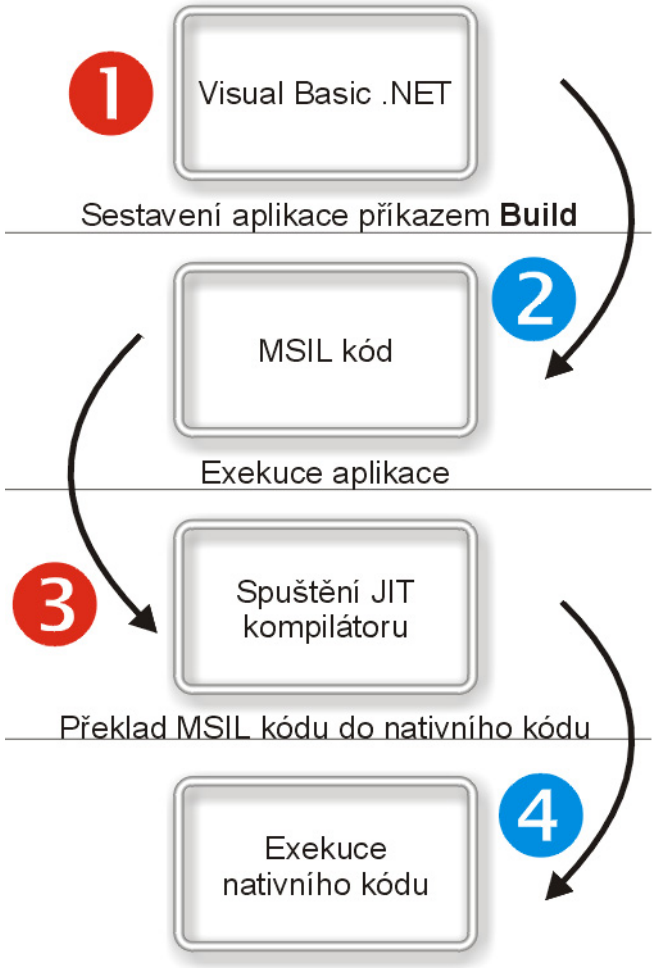
MSIL je nízkoúrovňový jazyk, pozůstávající z instrukcí, kterým rozumí **Common Language Runtime**. Protože jde o jazyk, který nemůže být přímo prováděn, dochází později (v době běhu aplikace) ke kompilaci kódu **MSIL** do nativního formátu počítače.

Abyste měli lepší představu o kódu jazyka **MSIL**, můžete si tento kód představit jako jistý druh polotovaru. Tak jako musí být polotovar upraven do podoby finálního produktu, tak i segment kódu **MSIL** musí být před použitím „upraven" do strojového jazyka, kterému je schopen počítač

bezprostředně porozumět. Úprava **MSIL** kódu do nativního kódu je realizována právě prostřednictvím **CLR**, která při této operaci využívá služeb **Just-In-Time** (zkráceně **JIT**) kompilátoru.

JIT kompilátor přeloží potřebné kódové instrukce, vyhradí pro ně prostor v paměti a spustí je. Jestliže v budoucnosti programová logika poručí, že je třeba použít další instrukce programu, je opět zavolán **JIT** kompilátor, který přeloží další kód do jeho nativní podoby, a tak zabezpečí spuštění žádaných programových instrukcí.

Situaci sestavení .NET aplikace a její exekuce znázorňuje obr. 1.



Obr. 1 – Schematické zobrazení sestavení a způsobu běhu .NET aplikace

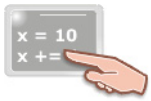
Protože je všechen kód nejdříve překládán do **MSIL** kódu, mohou různé aplikace, vytvořené v různých programovacích jazycích standardu .NET (Visual Basic .NET, Visual C# .NET a Visual C++ .NET) společně komunikovat. Z uvedeného důvodu musí všechny kompilátory .NET jazyků vyhovovat určitým pravidlům, která definuje **Common Language Specification (CLS)**. Tím je garantována mezijazyková kompatibilita.

Base Class Library (BCL)

Base Class Library, neboli jednodušeji řečeno knihovna základních tříd, představuje robustní kolekci tříd, typů, rozhraní a objektů, které můžete při vývoji vašich vlastních aplikací využít. Příchuť koncepce objektově orientovaného programování je v **BCL** přítomna v daleko větší míře jako kdykoliv předtím. Prakticky na každém kroku vývoje máte do činění s třídami, jejich instancemi (objekty) a po pravdě řečeno, ani si tuto skutečnost příliš neuvědomujete. Pro zachování pořádku je **BCL** členěna do tzv. jmenných prostorů. Pod pojmem jmenný prostor lze chápat určitou množinu spřízněných funkcí. Všechny jmenné prostory jsou organizovány hierarchicky, od základní úrovně,

kterou představuje jmenný prostor **System** až po další a další větvě, které ukrývají další kolekce jmenných prostorů. Pokud budete vyvíjet standardní „okenní" aplikaci pro Windows, v hojné míře vám poslouží třídy jmenného prostoru **System.Windows.Forms**. Jak si můžete z uvedeného zápisu jmenného prostoru všimnout, jestliže se chcete dostat k dalším (vnořeným) jmenným prostorům jednoho jmenného prostoru, použijete operátor **tečka (.)**.

Abyste ovšem nemuseli v programovém kódu neustále zadávat plně kvalifikovaný název jmenného prostoru, můžete si situaci zjednodušit použitím klíčového slova **Imports** a zadáním kýženého jmenného prostoru. Předpokládejme, že pracujeme s jmenným prostorem **System.Windows.Forms.Form**. Tento nám umožňuje přistupovat k vlastnostem a metodám třídy **Form**, ze které se tvoří samotné objekty formulářů. Abychom ale nemuseli vždy zadávat tento dlouhý název jmenného prostoru, vložíme odkaz na jmenný prostor takto:




```
Imports System.Windows.Forms
```

A nyní se na formulář budeme odkazovat jenom pomocí klíčového slova **Form**.

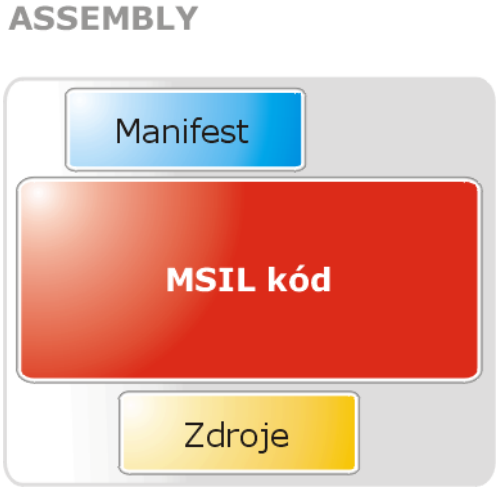
Klíčové slovo **Imports** se musí ve VB .NET uvádět na samém **začátku** programového kódu, ihned za příkazy typu **Option** (např. **Explicit**), jestliže nějaké používáte.

Anatomie .NET aplikace

Změny v stavbě aplikací, které přinesl .NET Framework, jsou vskutku revolučního charakteru. Předně, základní stavební jednotkou se stává tzv. **assembly**. Assembly lze chápat jako jistou kolekci programového kódu, programových zdrojů a metadat. Každá assembly obsahuje **manifest**, který popisuje její vnitřní strukturu. Manifest obvykle poskytuje informace o následujících skutečnostech:

- identifikaci assembly, kterou tvoří její jméno a číslo verze,
- seznamu typů použitých v assembly,
- seznamu jiných assembly, které daná assembly vyžaduje ke své činnosti,
- a konečně o bezpečnostních nařízeních pro popisovanou assembly.

V této souvislosti je důležité si uvědomit, že každá assembly může obsahovat pouze jeden manifest. Assembly dále tvoří jeden nebo i několik **modulů**, které obsahují programový kód .NET aplikace a také metadata, popisující tento kód. Veškerý programový kód, který assembly obsahuje, se nachází ve formě **MSIL** kódu. Příklad jednoduché assembly můžete vidět na obr. 2.



Obr. 2 – Struktura jednoduché assembly

Common Type System (CTS)

Common Type System je odpovědný za typovou kompatibilitu .NET aplikací. Jak již dobře víte, v procesu kompilace aplikací dochází k přebudování programového kódu do podoby **MSIL**. Toto se děje u všech programovacích jazyků platformy .NET, ne jenom u Visual Basicu .NET. Chceme-li hovořit o kompatibilitě na nízké úrovni, musí být nějakým způsobem upravený i typový systém všech aplikací. V kódu jazyka **MSIL** jsou všechny odpovídající si datové typy jednotlivých programovacích jazyků .NET vyjádřeny ve stejné podobě. Ukažme si příklad. Visual Basic .NET i Visual C++ .NET mohou pracovat s datovým typem **Integer** (ve VB je označení typu **Integer**, ve VC++ **int**). Aby byla zabezpečená jednotnost, při překladu aplikací do **MSIL** kódu jsou všechny výskyty datového typu **Integer** (resp. **int**) nahrazeny jednotným systémovým typem **System.Int32**. Podobně je to i u ostatních datových typů.

CTS tvoří dvě skupiny typů:
- **hodnotové typy**
- **referenční typy**

Proměnné hodnotových typů obsahují konkrétní hodnotu, která je uložená v paměti. Touto hodnotou může být např. číslo v pevné nebo pohyblivé řádové čárce. Proměnné referenčního typu v sobě uchovávají odkaz na paměťové místo, na němž se nachází jistá hodnota.

K **hodnotovým typům** patří:
- základní datové typy (**Integer**, **Boolean**, **Char**, ...)
- uživatelsky definované typy (struktury)
- enumerační (výčtové) typy

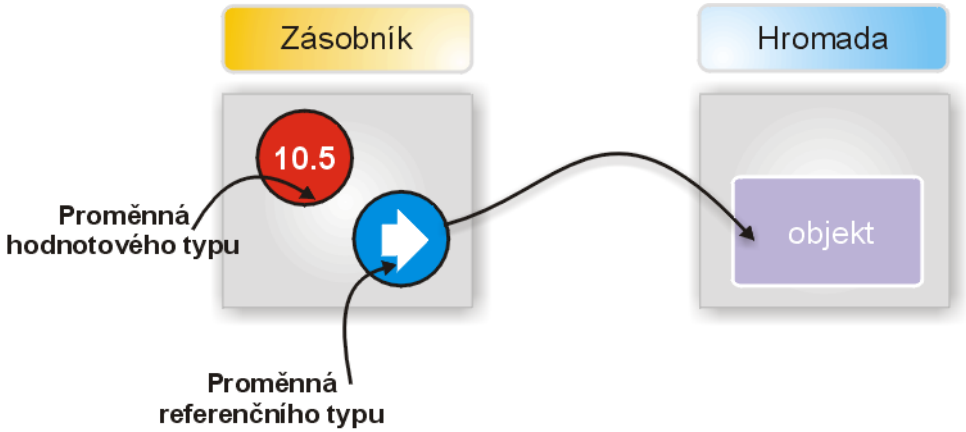
K **referenčním typům** patří:
- třídy
- rozhraní
- pole
- delegáti
- ukazatele

Pro dokonalejší probádání problematiky hodnotových a referenčních typů bychom si měli povědět o tom, jak se s těmito typy zachází v operační paměti počítače. Ještě předtím ovšem musíte porozumět dvěma základním pojmům z oblasti paměťového managementu. Jde o pojmy **zásobník** (**stack**) a **hromada** (**heap**).

Zásobník si můžete představit jako oblast paměti, která je vyhrazena pro běh jisté programové aplikace. V okamžiku, kdy je jako odpověď na vzniklou událost volána programová funkce, jsou všechny proměnné, které tato funkce používá, umístěny na zásobník. Když je volána další funkce, tak i její proměnné jsou uloženy na zásobník, ovšem „nad" předchozí uložené proměnné. Zásobník pracuje na principu **LIFO** (angl. **Last-In-First-Out**, tedy Poslední-Dovnitř-První-Ven). To je ve skutečnosti hlavní princip fungování zásobníku. Podle tohoto principu je možné ze zásobníku vybírat jenom ty proměnné, které tam byly uložené naposled. Jestliže volaná funkce ukončí svou činnost, všechny její proměnné jsou odstraněny ze zásobníku. Jinak řečeno, paměť, kterou tyto proměnné okupovaly, je uvolněna a přístupná pro další operace.

Oblast paměti, která je rezervována pro tvorbu objektů, se nazývá **hromada** (**heap**). Tato oblast je pod správou **Common Language Runtime**, která za tímto účelem využívá služeb pokročilé správy paměti označované jako **Garbage Collection** (**GC**).

Hodnoty přiřazené do hodnotových typů jsou ukládány vždy na zásobník. Naproti tomu u proměnných referenčních typů je situace zajímavější. Zde je samotný objekt uložený na hromadě, avšak proměnná, která obsahuje odkaz na tento objekt se nachází na zásobníku. Tuto situaci lze pozorovat na obr. 3.



Obr. 3 – Ilustrace práce hodnotových a referenčních typů

Pokud je o to požádána, proměnná poskytne adresu paměti, na které je objekt uložen. Jestliže je proměnná zrušena, zruší se i odkaz na objekt, který byl v dané proměnné umístěn. Ovšem pozor! Jestliže existují další reference na daný objekt, tento není zlikvidován. Destrukce objektu (prostřednictvím **Garbage Collection**) nastává až ve chvíli, když na něj nejsou navázány žádné další odkazy.

Integrované prostředí rubriky Visual Basic	Autor: Ján Hanák

Programátorská laboratoř

Prostor pro experimentování

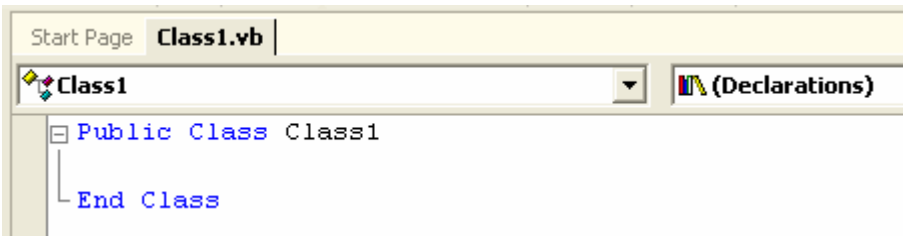
Časová náročnost (min): 60	Začátečník	Pokročilý	Profesionál

Použitý operační systém :	Windows XP
Hlavní vývojový nástroj :	Visual Basic .NET 2002
Další vývojový software :	Žádný
Jiný software :	Žádný

Tvorba třídy

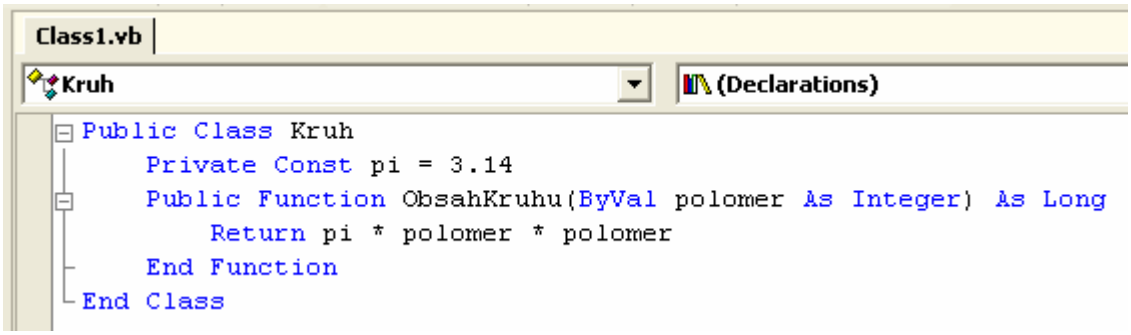
V následující ukázce společně vytvoříme jednoduchou třídu, které bude obsahovat jednu metodu pro výpočet obsahu kruhu. Postupujte takto:

1. Spusťte Visual Basic .NET, v okně pro výběr nového projektu vyberte položku **Class Library**, projekt pojmenujte jako **Trida_01**, zatrhněte volbu **Create directory for Solution** a klepněte na tlačítko **OK**.
2. Visual Basic .NET vytvoří projekt příslušného typu a otevře pro vás okno pro zápis kódu (obr. 1).



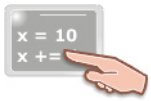
Obr. 1 – Kód třídy vygenerovaný Visual Basicem .NET

3. Změňte jméno třídy na **Kruh** a jestliže chcete, můžete přejmenovat i soubor, ve kterém je programový kód třídy uložen. Standardně má tento soubor jméno **Class1.vb**.
4. Přidejte do třídy deklaraci konstanty a kód pro metodu **ObsahKruhu**. Výslednou podobu kódu třídy **Kruh** můžete vidět na obr. 2.



Obr. 2 – Finální podoba třídy Kruh

5. Uložte projekt a proveďte jeho kompilaci, čímž vytvoříte assembly ve formě DLL s kódem třídy.
6. Přidejte ke stávajícímu projektu nový testovací projekt typu **Windows Application** (**File>Add Project>New Project**).
7. Do testovacího projektu přidejte odkaz na assembly s třídou (**Project>Add Reference**). Klepněte na tlačítko **Browse** a vyhledejte potřebnou assembly.
8. Umístěte na formulář testovacího projektu instanci ovládacího prvku **Button**, kterou můžete nechat implicitně pojmenovanou. Její obsluhu události **Click** vyplňte tímto kódem:




```
Dim f As New Trida_01.Kruh()
MessageBox.Show(f.ObsahKruhu(10).ToString)
```

9. V okně **Solution Explorer** klepněte pravým tlačítkem myši na jméno testovacího projektu. Záhy se objeví kontextové menu, ve kterém klikněte na položku **Set as StartUp Project**.
10. Spusťte projekt a klepněte na tlačítko. V okně se zprávou se zobrazí hodnota, která přináleží obsahu kruhu, jehož poloměr je roven 10.

Sestrojení průhledného formuláře

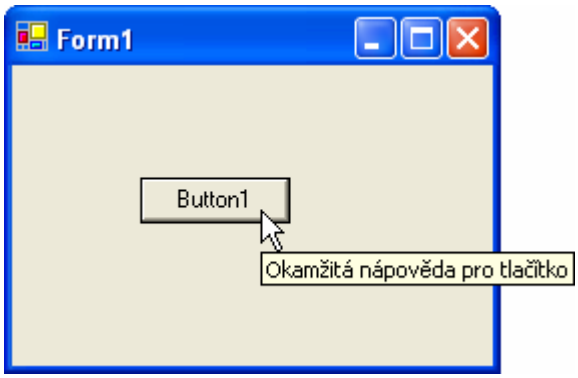
Visual Basic .NET obsahuje řadu nových a doposud nevídaných možností, mezi něž patří také vlastnost, dovolující vám nastavit průhlednost formuláře. Jméno této vlastnosti je **Opacity** a můžete ji nalézt mezi standardními vlastnostmi formuláře v okně **Properties**. Platné hodnoty pro uvedenou vlastnost se nacházejí v intervalu <0,1>, nebo, jestliže chcete, mezi 0 a 100 procenty. Hodnota 0 znamená, že formulář je zcela neviditelný, zatímco hodnota 1 představuje formulář v jeho plné kráse.

Ukázka použití ovládacího prvku ToolTip

Prostřednictvím ovládacího prvku **ToolTip** můžete přiřadit k dalším ovládacím prvkům, které jsou umístěny na formuláři okamžitou nápovědu. Prvek **ToolTip** patří mezi tzv. **extender** prvky, což jsou prvky, které rozšiřují možnosti jiných ovládacích prvků. Instance prvku **ToolTip** se proto z uvedeného důvodu automaticky neumístí na plochu formuláře, nýbrž zaujme své místo na podnosu komponent (obr. 4). V prostoru podnosu komponent se nacházejí prvky, jež nejsou vidět za běhu programu. Jakmile je ovládací prvek **ToolTip** přidán do projektu, ovlivní vlastnosti dalších ovládacích prvků na formuláři tím, že do jejich seznamu vlastností v okně **Properties** přidá instanci svoji vlastní vlastnosti s názvem **ToolTip on ToolTip1**.

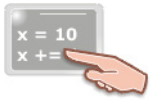
Obr. 4 – Ovládací prvek **ToolTip**

Předpokládejme nyní, že se na formuláři nachází tlačítko. Vyplníte-li vlastnost tlačítka **ToolTip on ToolTip1** informačním textem, po spuštění projektu a umístění kurzoru myši se po chvilce objeví okamžitá nápověda (obr. 5).



Obr. 5 – Zobrazení okamžité nápovědy pomocí prvku **ToolTip**

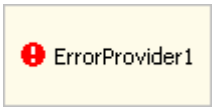
Text pro okamžitou nápovědu můžete dynamicky měnit i za běhu programu pomocí programového kódu:




```
ToolTip1.SetToolTip(Me.Button1, _
"Chcete-li uložit projekt, klepněte na toto tlačítko")
```

Ukázka použití ovládacího prvku ErrorProvider

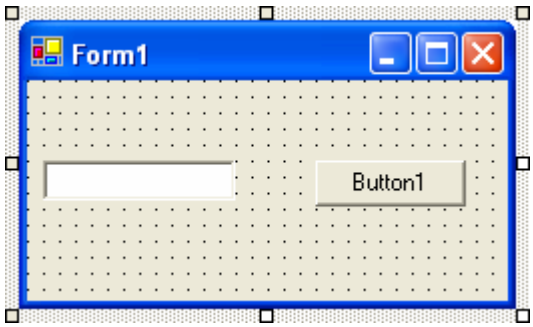
Ovládací prvek **ErrorProvider** je, podobně jako prvek **ToolTip**, extender prvkem. Hlavní úkol prvku **ErrorProvider** spočívá ve vizuálním upozornění uživatele na vzniklou chybovou situaci. Dojde-li k chybě, kterou umí prvek rozeznat, zobrazí speciální ikonu při tom ovládacím prvku, s kterým je integrován (jinak řečeno, kterého schopnosti rozšiřuje). Umístí-li uživatel kurzor myši nad vzpomínanou ikonu, zobrazí se text, identifikující vzniklou chybu. Pokud přidáte prvek do projektu, uloží se na podnos komponent. Podobu ovládacího prvku **ErrorProvider** můžete vidět na obr. 6.



Obr. 6 – Podoba prvku **ErrorProvider** na podnosu komponent

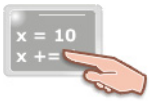
V následujícím příkladu si ukážeme, jak integrovat prvek **ErrorProvider** s ovládacím prvkem **TextBox**. Postupujte takto:

1. Přidejte na formulář textové pole (**Textbox1**) a tlačítko (**Button1**).
2. Umístěte na podnos komponent ovládací prvek **ErrorProvider**.
3. Podoba vašeho projektu by měla být podobná té z obr. 7.



Obr. 7 – Ukázka podoby projektu

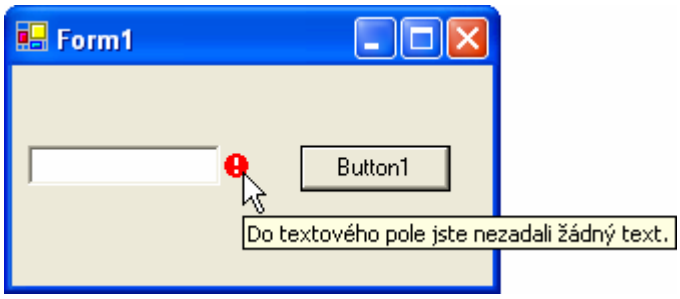
4. Do obsluhy události **Click** tlačítka **Button1** přidejte tento programový kód:




```
If Len(TextBox1.Text) = 0 Then
    ErrorProvider1.SetError(TextBox1, "Do textového pole " & _
    "jste nezadali žádný text.")
Else
    ErrorProvider1.SetError(TextBox1, "")
End If
```

Tento kód testuje, zdali je textové pole prázdné či nikoliv. V případě, že uživatel nezadal do textového pole vůbec žádný text, aktivuje se prvek **ErrorProvider**. Ten zobrazí vedle textového pole výstražnou ikonu, která upozorní uživatele, že něco není v pořádku. Pokud uživatel ukáže kurzorem na tuto ikonu, zobrazí se informační text, popisující danou chybu. V opačném případě, není-li textové pole prázdné, nestane se nic.

5. Spusťte projekt a klikněte na tlačítko **Button1** bez toho, abyste cokoliv zapisovali do textového pole (obr. 8).



Obr. 8 – Ovládací prvek **ErrorProvider** v akci

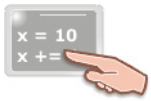
Zadáte-li nyní do textového pole libovolný textový řetězec a klepnete-li poté opět na tlačítko **Button1**, uvidíte, že výstražní ikona zmizí.

V situacích, ve kterých je nutné upozornit uživatele na vzniklou chybu přívětivou cestou, lze vhodně využít možností ovládacího prvku **ErrorProvider**.

Vytvoření přetížené funkce

VB .NET vám dovoluje vytvořit tzv. **přetížené funkce**. Přetížené funkce jsou funkce, které mají stejné jméno, ovšem rozličnou signaturu (seznam parametrů). Schopnosti přetížených funkcí mohli doposud využívat jenom programátoři C/C++, avšak nyní se podobná možnost naskýtá i vám.

Přetížení funkce je velmi užitečná technika, pomocí které můžete snadno „vyrobit" sadu funkcí se stejným jménem, no odlišnými vstupními parametry. Zápis přetížené funkce s názvem **Vypocet** můžete vidět na následující ukázce kódu:

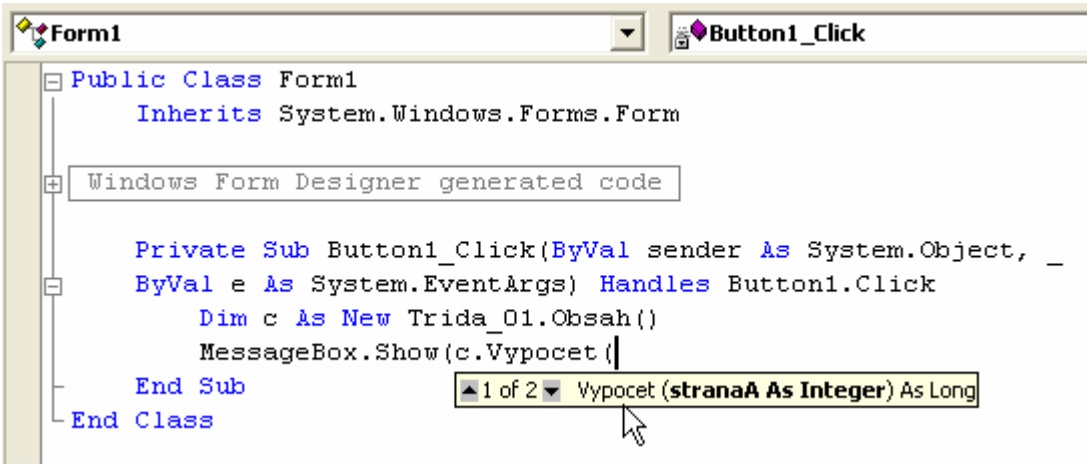



```
Public Class Obsah

    Public Overloads Function Vypocet(ByVal stranaA As Integer) _
    As Long
        Return stranaA * stranaA
    End Function

    Public Overloads Function Vypocet(ByVal stranaA As Integer, _
    ByVal stranaB As Integer) As Long
        Return stranaA * stranaB
    End Function

End Class
```

V hlavičkách obou funkcí můžete vidět i modifikátor **Overloads**, jenž indikuje, že jde o přetíženou funkci. I když je použití tohoto modifikátoru volitelné, v kódu jsem ho uvedl proto, aby bylo na první pohled viditelné, že máme do činění s přetíženou funkcí.

Jak si můžete všimnout, jsou zde definované dvě funkce. První funkce je určená na výpočet obsahu čtverce, a proto přebírá pouze jeden parametr. Druhá funkce ovšem počítá obsah obdélníku, a z tohoto důvodu pracuje se dvěma vstupními parametry.

Jestliže vytvoříte instanci této třídy, při výběru vhodné varianty funkce **Vypocet** vám pomůže VB .NET prostřednictvím technologie **IntelliSense** (obr. 9).



Obr. 9 – Výběr přetížené funkce

Klepnete-li v okně okamžité nápovědy na ikonu se šipkou dolů, objeví se deklarace přetížené podoby funkce **Vypocet**.

Integrované prostředí rubriky Visual Basic	Autor: Ján Hanák

Téma měsíce

Grafický subsystém GDI+ (1. díl)

Časová náročnost (min):	Začátečník	Pokročilý	Profesionál
45			

Použitý operační systém :	Windows XP
Hlavní vývojový nástroj :	Visual Basic .NET 2002
Další vývojový software :	Žádný
Jiný software :	Žádný

Milí čtenáři,

v dnešním a budoucím vydání se podíváme blíže na nový grafický subsystém, neboli grafické aplikační programové rozhraní (API), které přináší platforma .NET. Jméno tohoto grafického systému je GDI+ (angl. **Graphics Device Interface Plus**). Ve výkladu se podíváme na nové možnosti, které jsou pro programátory více než přínosné. Dnes se naučíme kreslit jednoduché geometrické obrazce, jako jsou úsečky, obdélníky, vybarvené obdélníky a obdélníky s gradientní výplní. Příště se podíváme na Bezierovy křivky, kardinální spline křivky a jiná pokročilá témata. Tedy, vzhůru do GDI+!

Obsah

GDI+ a jeho místo v .NET dílně

Grafický systém GDI+ je nástupcem již staršího a poněkud zastaralého systému GDI. Pokud situaci hodně zjednodušíme, můžeme prohlásit, že grafický systém GDI+ je zodpovědný za vše, co se nachází na počítačové obrazovce. Pro vás, jako programátory ve VB .NET je ovšem důležitější skutečnost, že nový systém vám dovoluje provádět doposud nevídané věci, a to cestou velmi jednoduchou, dokonce bych řekl, že až intuitivní. Vše co GDI+ nabízí je moderně „zaobaleno" do velkého množství tříd a jmenných prostorů. Pokud přicházíte z Visual Basicu verze 6.0, možná budete po prvních zkušenostech s GDI+ poněkud zklamáni tím, že „všechno je jinak". Ano, způsob, jakým prováděl grafické operace VB 6.0 je na míle vzdálený tomu, se kterým se setkáte ve VB .NET. Ovšem nevěšte hlavu! Pokud se naučíte správně a efektivně aplikovat grafické algoritmy relevantní pro .NET platformu, budete moci daleko snadněji generovat mnohem složitější a také působivější grafické obrazce.

GDI+ nabízí rozšíření především v následujících oblastech:

1. **Práce s grafikou ve dvojrozměrném (2D) prostoru**
   Nových metod a technik, které můžete při kreslení 2D obrazců využít je opravdu velké množství. Níže uvedená tabulka představuje výčet těch nejmarkantnějších změn:

Antialiasing	Při renderování křivek, ale i jiných grafických objektů dochází velmi často k vzniku „zubatých" hran. Pokud nebudete pracovat jenom s horizontálními nebo vertikálními křivkami, určitě se s neželaným efektem „zubatosti hran" střetnete. Aby i různorodé křivky a geometrické tvary působily uhlazeně, můžete využít výhod techniky zvané **Antialiasing**. V podstatě jde o vytvoření plynulého barevného přechodu mezi barvou křivky a pozadím, na kterém je křivka nakreslena.
Kardinální spline křivky	Kardinální spline křivky si můžete představit jako soubor menších křivek, ze kterých je vytvářena jedna větší křivka. Zvláštností je, že přechod mezi jednotlivými body je zcela plynulý, tedy nedochází ke vzniku ostrých hran při „napájení" jedné křivky do druhé. Další zajímavostí je možnost definice tzv. faktoru napětí, který určuje, jak moc bude křivka v „přechodních" bodech ohýbána.
Alpha Blending	Technika pro nastavování průhlednosti grafických objektů. Potenciálních stupňů průhlednosti je 256, stupně lze tedy volit z intervalu <0, 255>. Stupeň 0 znamená nulovou průhlednost, stupeň 255 pak kompletní průhlednost.
Gradientní výplně	Gradientní výplní se rozumí plynulý přechod z jedné barvy do druhé. Pomocí tohoto oblíbeného efektu můžete snadno vygenerovat region s různým typem barevného přechodu.
Bezierovy křivky	Bezierova křivka je speciální křivka, která je (v nejjednodušším provedení) tvořena počátečním a koncovým bodem a dvěma řídícími body. Nastavováním polohy řídících bodů lze plynule měnit tvar samotné křivky.
Transformace	Pomocí nové třídy **Matrix** lze uskutečňovat transformace grafických objektů (rotace, posunutí atd.).
Škálovatelné regiony	Regiony jsou ukládány ve světových souřadnicích a lze na ně aplikovat jakékoliv grafické transformace pomocí transformační matice.

2. Práce s obrázky

Velké množství grafických formátů	Grafický subsystém GDI+ obsahuje podporu pro větší počet nativních grafických formátů. Jde o tyto formáty: .BMP, .GIF, .JPEG, .Exif, .PNG, .TIFF, .ICON, .WMF, .EMF.
Nové grafické operace	GDI+ vám dovoluje aplikovat na rastrové obrázky nové algoritmy např. přizpůsobení jasu, kontrastu, rotace, zešikmení atd.

3. Typografická rozšíření

Lepší čitelnost textu na LCD obrazovkách	Technologie **ClearType** od společnosti Microsoft zvyšuje čitelnost textu na LCD obrazovkách. Do oblasti působnosti technologie spadají také pokročilé algoritmy pro vyhlazování hran textových jednotek.

Jmenné prostory pro GDI+

Jádro funkčnosti grafického systému GDI+ je zapouzdřeno do více jmenných prostorů, které si nyní představíme. Hlavním jmenným prostorem je **System.Drawing**, obsahující třídy pro základní manipulaci s grafickými objekty. Pokud budete chtít renderovat pokročilejší geometrické obrazce, zcela jistě „zavadíte" i o prostor jmen s názvem **System.Drawing.Drawing2D**. Budete-li potřebovat

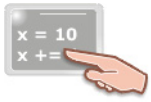
generovat specifické fonty, je vám k dispozici jmenný prostor **System.Drawing.Text** a pro modifikaci rastrových obrázků lze zase využít dovedností jmenné prostoru **System.Drawing.Imaging**.
Aby byl výčet prostorů jmen úplný, je zapotřebí vzpomenout ještě dva jmenné prostory, a to **System.Drawing.Design** a **System.Drawing.Printing**.

Souřadnicový systém

Renderování, neboli vykreslování grafických obrazců, se ve většině případů bude uskutečňovat na ploše formuláře. Proto bude vhodné, když si tento grafický region popíšeme podrobněji. Plocha formuláře, na kterou lze umísťovat grafické útvary, se nachází v rovině, ve které pozici objektů determinují jednotlivé souřadnice na horizontální (x-ové) a vertikální (y-ové) ose. Počátkem souřadnicového systému formuláře je jeho levý horní roh, jenž disponuje nulovými souřadnicemi (0,0). Koncový bod je pak určen jako pravý dolní roh formuláře. Nejmenší bod formuláře se nazývá pixel, nebo také obrazovkový bod. Grafický systém GDI+ nabízí několik struktur, které určují pozici objektu na grafické ploše. Mezi struktury patří **Point**, **Size** a **Rectangle**. Tyto struktury určují pozici regionů pomocí parametrů typu **Integer**. Všechny uvedené struktury mají ještě své dvojníky **PointF**, **SizeF** a **RectangleF**, ve kterých jako parametry vystupují čísla s pohyblivou řádovou čárkou a jednoduchou přesností (**Single**).

GDI+ v akci: Tvorba objektu Graphics

Pokud budete chtít kreslit grafiku, je potřebné za tímto účelem vytvořit speciální grafický objekt s názvem **Graphics**. Tento objekt představuje plochu, na kterou lze umísťovat grafické obrazce. Touto plochou může být třeba plocha formuláře, nebo plocha určeného ovládacího prvku. Při tvorbě grafického objektu **Graphics** tak ve skutečnosti získáme přístup k objektu **Graphics**, jenž představuje kreslící plochu blíže určeného objektu (např. formuláře). Tuto situaci zachycuje následující výpis zdrojového kódu:

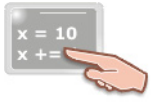



```
Dim graf_01 As Graphics
graf_01 = Me.CreateGraphics()
```

V kódu je deklarována objektová proměnná **graf_01**, do které je následně přiřazen, pomocí metody **CreateGraphics**, odkaz na objekt **Graphics** aktivního formuláře (**Me**). Takto lze získat přístup k objektu **Graphics** formuláře, neboli k samotné kreslící ploše formuláře. Na tuto plochu pak můžeme začít umísťovat další grafické obrazce.

GDI+ v akci: Kreslení úsečky

Jestliže jste úspěšně vytvořili hlavní grafický objekt, můžete pokračovat v „malování" dalších geometrických útvarů. Následující programový kód zabezpečí vykreslení úsečky.




```
Dim graf_01 As Graphics
Dim pero_01 As New Pen(Color.Black, 2)
graf_01 = Me.CreateGraphics()
graf_01.DrawLine(pero_01, 50, 40, 120, 200)
pero_01.Dispose()
graf_01.Dispose()
```

Zde si můžete všimnout použití nového objektu **Pen**, jenž reprezentuje „grafické pero", se kterým je vykreslena samotná úsečka. Při deklaraci grafického pera je specifikována jeho barva (**Color.Black**)

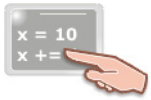
a tloušťka (**2**). Abyste vyrenderovali úsečku, použijte metodu **DrawLine** objektu **Graphics**. Metoda **DrawLine** pracuje s několika parametry. Především je to objekt **Pen** a x-ové a y-ové souřadnice počátečního a koncového bodu úsečky. Po realizaci primárního programovacího algoritmu jsou prostřednictvím volání metody **Dispose** uvolněny z paměti grafické objekty **Graphics** a **Pen**.

Po vykonání požadovaných grafických operací byste měli vždy zavolat metodu **Dispose**, která zabezpečí, že se uvolní paměťové zdroje spjaté s alokací příslušných grafických objektů.

Výsledek práce kódu můžete vidět na obr. 1.

Obr. 1 – Generování úsečky pomocí metody **DrawLine**

Pokud si prohlédnete obr. 1 pozorněji, určitě spatříte efekt „zubatosti" při vykreslování úsečky. Abyste tento nežádoucí efekt eliminovali, doplňte předchozí ukázku programového kódu takto:




```
Dim graf_01 As Graphics
Dim pero_01 As New Pen(Color.Black, 2)
graf_01 = Me.CreateGraphics()
graf_01.SmoothingMode = Drawing.Drawing2D.SmoothingMode.AntiAlias
graf_01.DrawLine(pero_01, 50, 40, 120, 200)
pero_01.Dispose()
graf_01.Dispose()
```

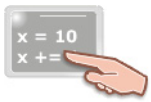
Výsledek můžete spatřit na obr. 2.

Obr. 2 – Generování úsečky pomocí metody **DrawLine** a aktivního **Antialiasingu**

Vhodným nastavením vlastnosti **SmoothingMode** objektu **Graphics** lze minimalizovat vytváření „schodkovitých" přechodů při malování úsečky.

GDI+ v akci: Kreslení obdélníku

Tvorba obdélníku je stejně snadná jako kreslení úsečky. Podívejte se nejdříve na programový kód, a pak si k němu něco řekneme.



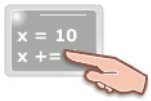

```
Dim obdl_01 As New Rectangle(10, 10, 200, 200)
Dim pero_01 As New Pen(Color.Black, 2)
Dim graf_01 As Graphics
graf_01 = Me.CreateGraphics()
graf_01.SmoothingMode = Drawing.Drawing2D.SmoothingMode.AntiAlias
graf_01.DrawRectangle(pero_01, obdl_01)
pero_01.Dispose()
graf_01.Dispose()
```

Pro vykreslení obdélníku je nevyhnutné deklarovat objekt **Rectangle**, jenž charakterizuje podobu vytvářeného obdélníku. Obdélník je definován x-ovými a y-ovými souřadnicemi levého horního rohu, šířkou a výškou. Proces samotného vykreslování je pod patronátem metody **DrawRectangle**, která přijímá dva parametry (objekty **Pen** a **Rectangle**). Finální podobu obdélníku znázorňuje obr. 3.

Obr. 3 – Generování obdélníku pomocí metody **DrawRectangle**

GDI+ v akci: Kreslení vybarveného obdélníku

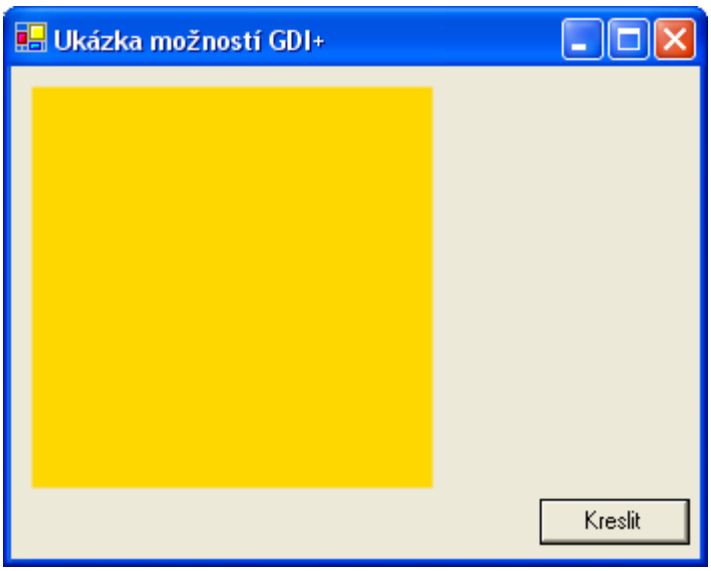
Snadnou obměnou předchozího fragmentu programového kódu lze docílit toho, aby byl obdélník vyplněn uživatelskou barvou.




```
Dim obdl_01 As New Rectangle(10, 10, 200, 200)
Dim br_01 As New SolidBrush(Color.Gold)
Dim graf_01 As Graphics
graf_01 = Me.CreateGraphics()
graf_01.SmoothingMode = Drawing.Drawing2D.SmoothingMode.AntiAlias
graf_01.FillRectangle(br_01, obdl_01)
br_01.Dispose()
graf_01.Dispose()
```

Pamatujte, že vždy, když budete chtít vytvořit grafický objekt s výplní, budete muset definovat objekt grafický štětec (**Brush**), jenž specifikuje barvu výplně. V našem příkladě jde o objekt **SolidBrush**, který specifikuje výběr barvy pro zabarvení objektu **Rectangle**. Vykreslení pak uskuteční metoda **FillRectangle**, která pracuje se dvěma parametry (objekt grafický štětec **SolidBrush** a objekt **Rectangle**).

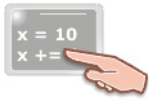
Výslední efekt přibližuje obr. 4.



Obr. 4 – Generování obdélníku s barevnou výplní pomocí metody **FillRectangle**

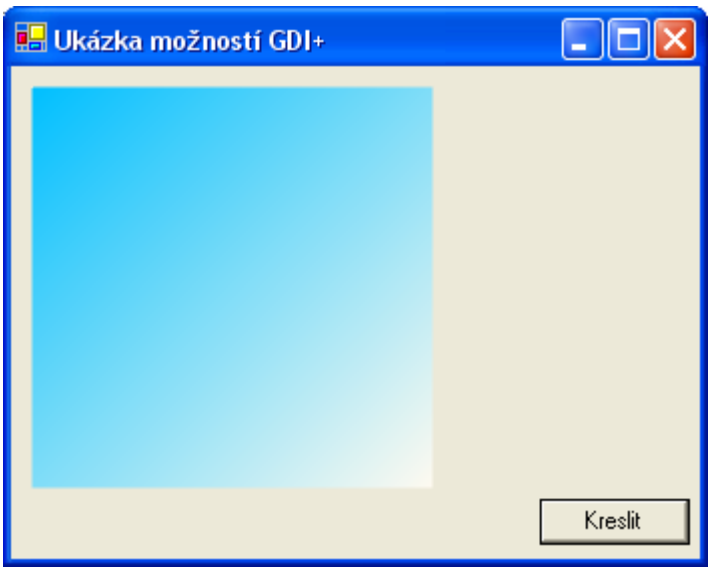
GDI+ v akci: Kreslení obdélníku s gradientní výplní

Chcete-li dosáhnout skutečně impresivních efektů, můžete nakreslit obdélník s gradientní výplní.




```
Dim obdl_01 As New Rectangle(10, 10, 200, 200)

Dim br_01 As New System.Drawing.Drawing2D.LinearGradientBrush _
(obdl_01, Color.DeepSkyBlue, _
Color.FloralWhite, _
Drawing.Drawing2D.LinearGradientMode.ForwardDiagonal)

Dim graf_01 As Graphics
graf_01 = Me.CreateGraphics()
graf_01.SmoothingMode = Drawing.Drawing2D.SmoothingMode.AntiAlias
graf_01.FillRectangle(br_01, obdl_01)
br_01.Dispose()
graf_01.Dispose()
```

Renderování obdélníku s gradientní výplní můžete vidět na obr. 5.



Obr. 5 – Generování obdélníku s **gradientní** výplní

Pro vytvoření plynulého barevného přechodu se používá speciální grafický štětec (objekt **LinearGradientBrush**), jenž se nachází v jmenném prostoru **System.Drawing.Drawing2D**. Pomocí tohoto štětce můžete rovněž určit počáteční a koncovou barvu, podobně jako i styl přechodu barev, jenž má být aplikován. Pro vykreslení grafického obrazce použijeme již „starou známou" metodu **FillRectangle**.

Integrované prostředí rubriky Visual Basic	Autor: **Ján Hanák**

Lab

Téma

Začínáme s VB .NET

Úvod do světa .NET (2. díl)

Časová náročnost (min):	Začátečník	Pokročilý	Profesionál
45	1	2	3

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

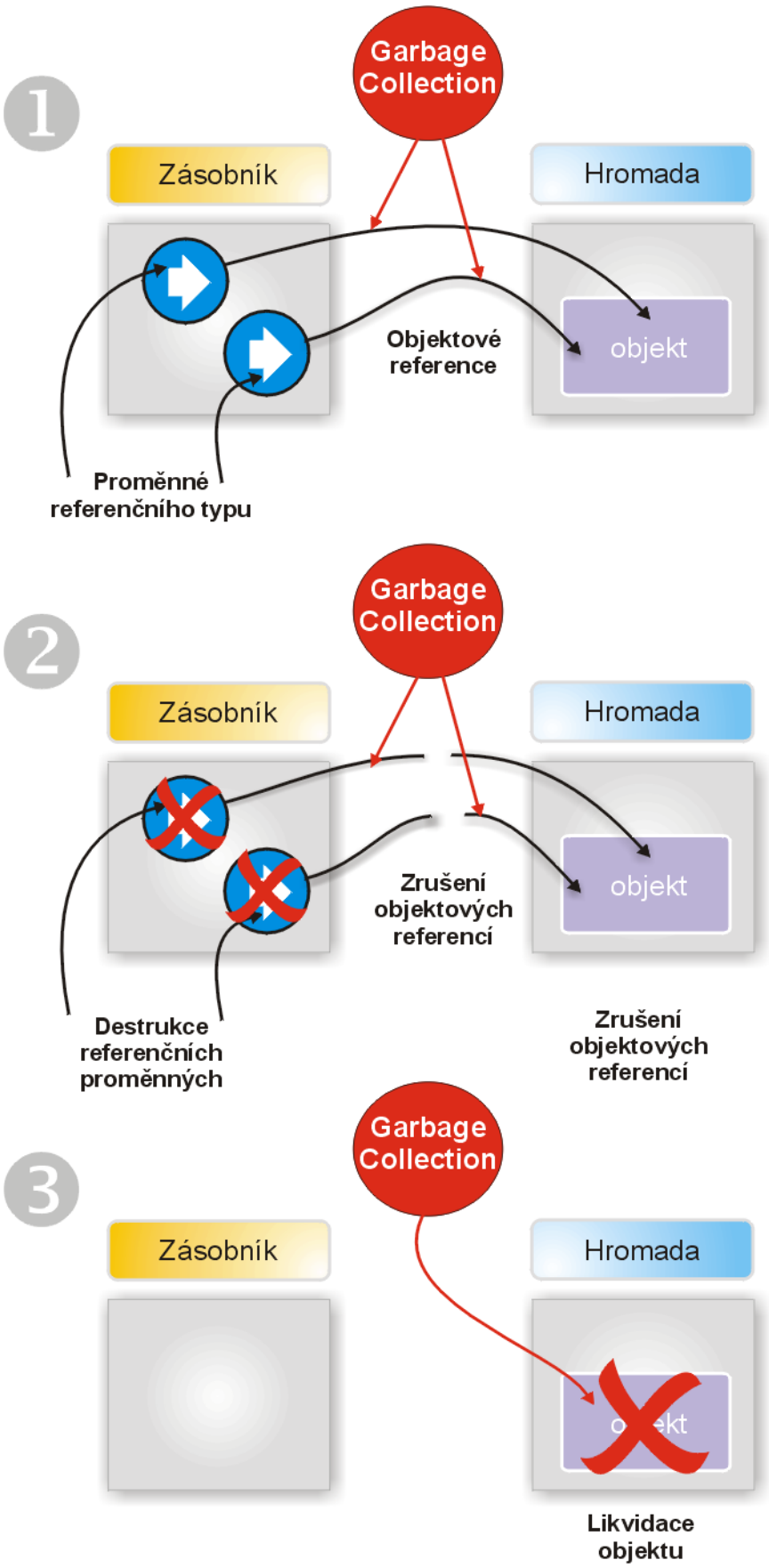
Vážení čtenáři,

hlavní náplní druhé části seriálu bude představení pracovního prostředí aplikace Visual Basic .NET, ovšem ještě před tím si povíme pár slov o dynamickém managementu paměti, jenž zabezpečuje .NET Framework prostřednictvím **Garbage Collection (GC)**.

Obsah

Představení Garbage Collection

V minulém dílu jsme si pověděli, co je to zásobník a hromada a rovněž jsme si ukázali, jak je to s použitím hodnotových a referenčních proměnných. Jak už víte, objekty jsou v paměti ukládány na hromadu (**heap**). Po každém úspěšném pokusu o vytvoření objektu je pro tento objekt alokován jistý prostor v paměti počítače. Jednoduše řečeno, objekt existuje tak dlouho, dokud na něj „ukazuje" některá proměnná referenčního typu. **Garbage Collection** permanentně sleduje vytvořený objekt a rovněž tak i reference, které jsou na tento objekt navázané. Ve většině svého životního cyklu působí **Garbage Collection** jako vlákno s nízkou prioritou. Ovšem v případě, že se dostaví nedostatek paměti, je priorita vlákna zvýšena a **GC** začne vykonávat svou práci tím, že odstraní z paměti nepotřebné elementy. Pokud **GC** splní svou úlohu (zabezpečí maximalizaci paměťového prostoru), je opět „přepnut" do módu s nízkou prioritou. **Garbage Collection** si můžete představit jako inteligentní paměťovou „policii", která neustále sleduje zákoutí paměti a hledá nepotřebné objekty, které následně zlikviduje. Princip práce **GC** je zjednodušeně zachycen na obr. 1.



Obr. 1 – Schematické zobrazení činnosti **Garbage Collection**

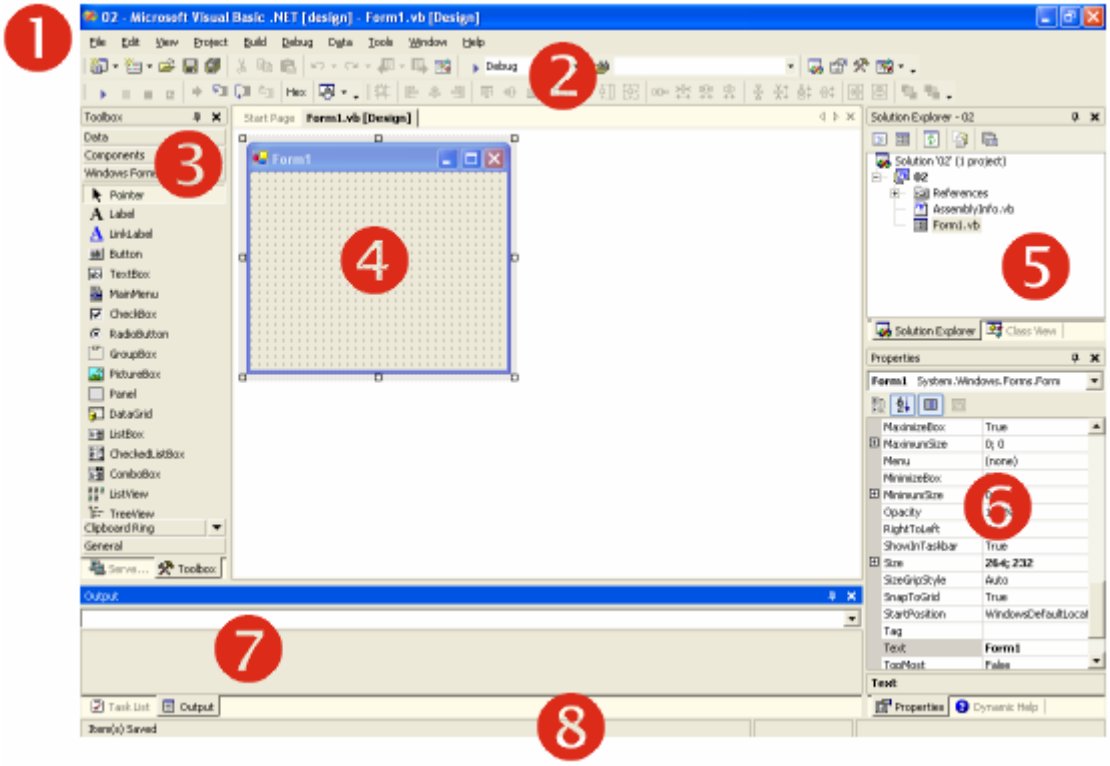
Popišme si nyní jednotlivé části obrázku podrobněji. V prvním kroku jsou znázorněny dvě referenční proměnné, pomocí kterých lze přistupovat k objektu, jenž je uložen na hromadě. Předpokládejme,

že v dalším kroku dojde k likvidaci referenčních proměnných (proměnné se dostanou „mimo" svého oboru platnosti). V tomto okamžiku nastane „prolomení" objektových referencí. Připomenu, že celou vzniklou situaci sleduje **Garbage Collection**, který následně podrobí destrukci i samotný objekt (třetí krok). Přibližně takto bychom mohli ve zkratce objasnit využití služeb správy paměti pomocí **Garbage Collection**.

Pro samotného programátora představuje existence **Garbage Collection** jasnou výhodu v tom, že již může hodit za hlavu všechny starosti s explicitním odstraňováním objektů z paměti. **GC** se sám postará o to, aby byla správa paměti opravdu bezpečná. Na druhé straně, i když si můžeme být jisti skutečností, že nepotřebné objekty budou z paměti odstraněny a paměť tak uvolněna, nevíme s určitostí povědět, kdy k této likvidaci objektů dojde. **Garbage Collection** tedy nepracuje s tzv. deterministickou finalizací, při které vždy programátor vydá pokyn na likvidaci objektu v přesném časovém okamžiku. I tento problém se ovšem dá vyřešit tím, že zavoláme specifickou metodu **Dispose**, abychom přinutili **GC** k okamžité reakci.

Charakteristika pracovního prostředí Visual Basicu .NET

Spustíte-li Visual Basic .NET a vytvoříte nový projekt typu **Windows Application**, můžete se střetnout s podobnou strukturou pracovního prostředí aplikace, jaká je zobrazena na obr. 2.



Obr. 2 – Ukázka integrovaného vývojového prostředí (**IDE**) Visual Basicu .NET

Je možné, že jako začínajícímu, nebo mírně pokročilému uživateli, se vám může zdát vývojové prostředí poněkud „přeplněné". Ano, máte pravdu. Visual Basic .NET je ve srovnání se svým starším bratříčkem daleko víc propracovanější a robustnější. Pojďme si tedy povědět pár informací o jednotlivých částech vývojového prostředí, které jsou znázorněny na obr. 2.

Část 1: Titulní pruh okna aplikace a pruh s hlavními nabídkami aplikace

V titulním pruhu jsou obsaženy informace o právě otevřeném projektu; jde o název projektu a jeho stav. Domnívám se, že bude vhodné, když pojem „stav" blíže vysvětlím. Projekt Visual Basicu se může v zásadě nacházet ve dvou stavech nebo režimech:

- **režim návrhu aplikace** (design)
- **režim běhu aplikace** (run)

Při režimu návrhu aplikace pracuje programátor se samotnou aplikací, např. upravuje parametry hlavního okna aplikace, píše programový kód atd. Režim návrhu aplikace lze přirovnat k opravě automobilu. Podobně jako se servisní technik stará o svěřený automobil, tak i programátor pečuje o svou aplikaci.

Při běhu aplikace je prováděn programový kód aplikace. V tomto režimu je aplikace spuštěna a programátor s ní manipuluje stejným způsobem jako finální uživatel. Pokud budeme chtít rozvinout předchozí analogii s automobilem, režim běhu aplikace můžeme přirovnat k procesu jízdy automobilu, tedy pohybu automobilu po určité trajektorii.

Pod titulním pruhem aplikace se nachází pruh s hlavními nabídkami. Ve Visual Basicu .NET se můžete střetnout s těmito typy základních nabídek:

- **File**
  Nabídka obsahuje velké množství standardních příkazů, které můžete znát i z jiných softwarových aplikací. Jsou zde příkazy pro založení nového projektu, otevření stávajícího projektu, uložení projektu a všech jeho součástí, seznam posledně otevřených projektů a souborů a mnohé další.

- **Edit**
  Nabídka sdružuje příkazy pro práci s textem, případně jinými objekty. Nacházejí se zde příkazy pro vyjmutí, kopírování, vkládání, výběr, dále příkazy pro hledání a záměnu a podobně.

- **View**
  Pomocí příkazu v této nabídce můžete zviditelnit okno pro zápis kódu či okno pro návrh vzhledu formuláře. Nabídka obsahuje také příkazy pro zobrazení mnohých dalších panelů nástrojů či speciálních oken, jako např. Průzkumníka řešení (**Solution Explorer**), nebo okno se seznamem dostupných vlastností pro různé objekty (**Properties Window**).

- **Project**
  Všechny příkazy, které se vážou k právě otevřenému projektu tvoří čtvrtou nabídku. Prostřednictvím dostupných příkazů můžete do projektu přidávat nové, nebo již existující programové součásti, dále moduly, třídy, komponenty a odkazy na další externí součásti.

- **Build**
  Nabídka **Build** pozůstává z příkazů pro sestavení aplikace a vygenerování spustitelného souboru (**assembly**).

- **Debug**
  V okamžicích, kdy budete pilně odstraňovat veškeré chyby z vaší aplikace, jistě využijete možností, které vám poskytuje nabídka **Debug**. Přidružené příkazy vám dovolí převzít chod aplikace „do svých rukou". Můžete využívat např. krokování programu či umísťování zarážek na požadovaná místa.

- **Tools**
  V této nabídce je umístěna velice důležitá položka **Options**, která vám umožňuje upravit si vzhled a chování aplikace k obrazu svému. Kromě ní se v nabídce nachází příkazy spouštějící externí vývojové nástroje.

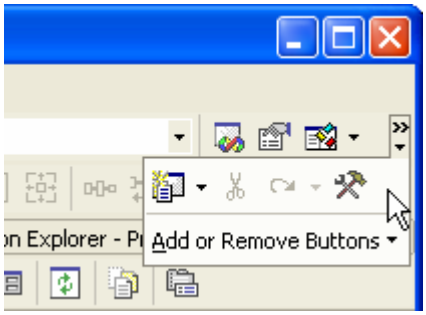
- **Windows**
  Slouží pro práci s jednotlivými okny aplikace.

- **Help**
  Obsahem poslední nabídky jsou položky pro zobrazení nápovědy, „zavolání" technické podpory či zobrazení informací o verzi aplikace.

K základním nabídkám jsou v případě potřeby přidány ještě dodatečné nabídky. Např. při práci s formuláři se k standardním nabídkám přidají další dvě, a to **Data** a **Format**.

Část 2: Panely nástrojů

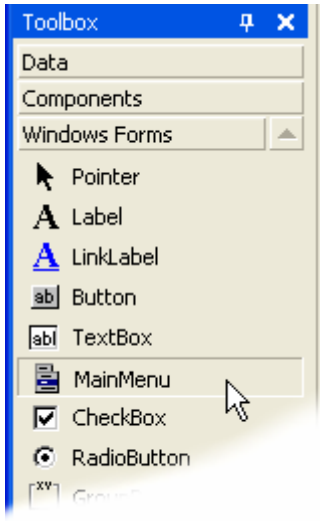
Panely nástrojů jsou tvořeny soustavou tlačítek s rozličnými ikonami, které v značné míře urychlují práci a orientaci v pracovním prostředí aplikace. Kromě základních panelů nástrojů můžete zobrazit také další, které budete chtít „mít na očích". Panely obdržely i jistou dávku inteligence, a tudíž vám umožní shlédnout jejich obsah i v případě, že prostor pro panel je menší než jeho rozsah. Tehdy se při pravém okraji panelu zobrazí dvojitá šipka, která po aktivaci nabídne seznam těch položek panelu, které nebylo doposud vidět (obr. 3).



Obr. 3 – Inteligentní panel nástrojů

Část 3: Souprava nástrojů (Toolbox)

Souprava nástrojů v ucelené formě uvádí všechny dostupné ovládací prvky, které můžete ve většině případů umístit na formulář a dále s nimi pracovat. Její podobu lze vidět na obr. 4.

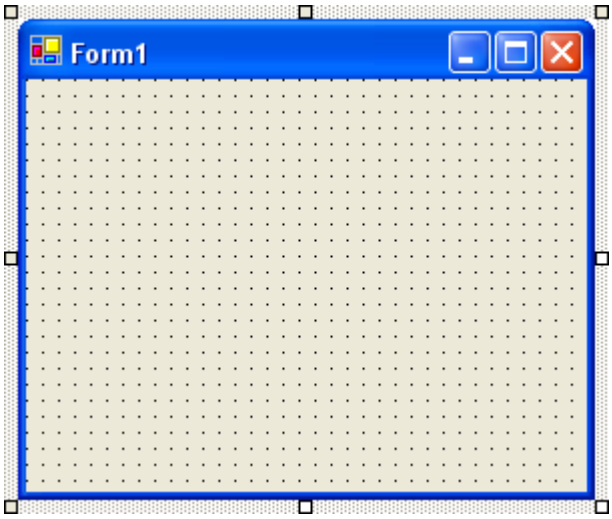


Obr. 4 – Souprava nástrojů aplikace Visual Basic .NET

Souprava nástrojů obsahuje skutečně velké množství ovládacích prvků, které můžete použít ve vašich aplikacích. S mnohými prvky se obeznámíme v dalších částích tohoto seriálu. Instance, neboli kopie těchto ovládacích prvků se zpravidla umísťují na formulář, případně na tzv. podnos komponent.

Část 4: Formulář

Formulář je základní vizuální jednotkou aplikace ve Visual Basicu .NET. Po technické stránce jde o standardní okno obdélníkového tvaru, které disponuje všemi potřebnými elementy: systémovou nabídkou, titulkovým pruhem, tlačítky pro minimalizaci, maximalizaci a uzavření okna, dále aktivní plochou a okrajem. Podobu formuláře můžete vidět na obr. 5.



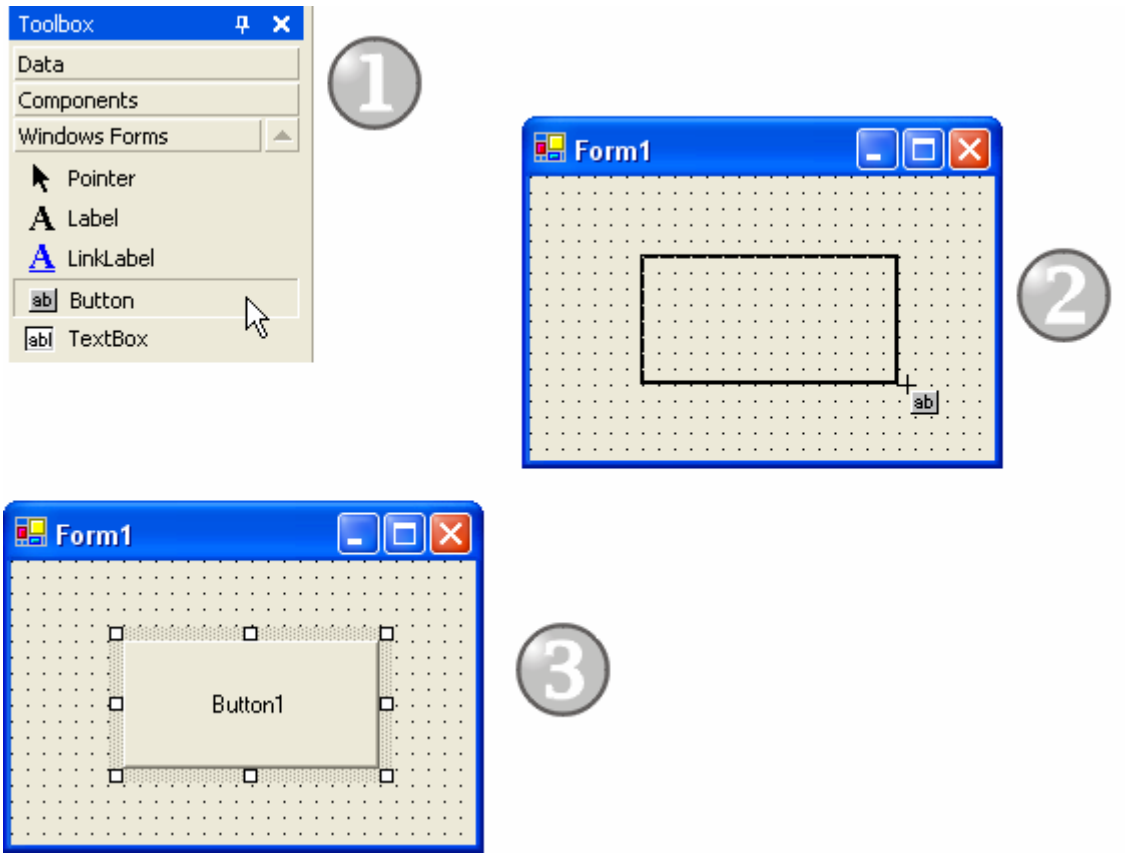
Obr. 5 - Formulář

Nachází-li se vaše aplikace v režimu návrhu, můžete při okrajích formuláře spatřit malé čtverce, které vám umožňují měnit velikost jednotlivých stran formuláře. Pokud budete chtít změnit horizontální nebo vertikální velikost formuláře, ukažte myší na tyto speciální „kotvící" značky a tažením je přemístěte do požadované pozice.

Visual Basic je velice dobře proslulý tím, že umožňoval vývojářům velmi lehce vytvářet formuláře a umísťovat na ně instance příslušných ovládacích prvků. Tuto myšlenku lze považovat za základní kámen filozofie vizuálního programování. Ještě jednodušeji řečeno, pokud chtěl programátor přidat na formulář některý z ovládacích prvků, prostě jej „namaloval" na formulář a bylo. Pokud jste zatím nikdy neměli tu čest pracovat s vizuálním návrhem formulářů, zde je postup, jak umístit na formulář instanci ovládacího prvku **Button** (tlačítko):

1. Vyberte v soupravě nástrojů prvek **Button** a přemístěte kuzor myši nad plochu formuláře.
2. Stiskněte levé tlačítko myši a při jeho soustavném držení nakreslete obrysy instance ovládacího prvku na formulář.
3. Jestliže jste s rozměry ovládacího prvku spokojeni, uvolněte stisknutí levého tlačítka myši.

Uvedený proces lze pozorovat na obr. 6.



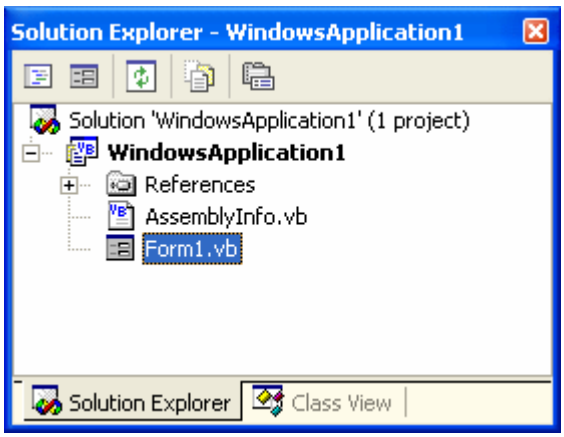
Obr. 6 – Postup přidání instance ovládacího prvku **Button** na formulář

Všimněte si, že po vybrání ovládacího prvku **Button** ze soupravy nástrojů se změní kurzor myši na křížek s ikonou právě vybraného ovládacího prvku. V našem příkladě vypadá kurzor myši takto:

Visual Basic .NET vás i tímto způsobem informuje o typu ovládacího prvku, se kterým právě pracujete.

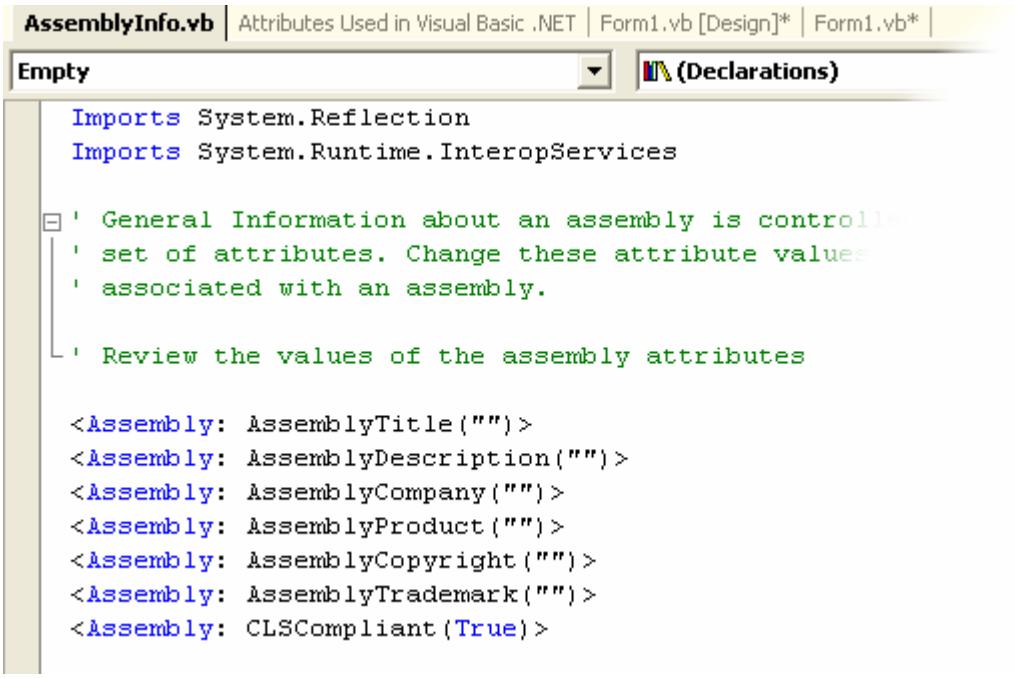
Část 5: Průzkumník řešení (Solution Explorer)

Průzkumník řešení představuje správce vašeho řešení. Pod pojmem „řešení" si můžete představit komplexní rámec vašeho vývojářského projektu. Řešení může obsahovat jeden, nebo i víc projektů, či jiných softwarových součástí. Přitom tyto součásti nemusí být napsány jenom ve Visual Basicu .NET. Do stávajícího řešení lze přidávat i jiné prvky, které byly vytvořeny v dalších programovacích jazycích podporujících standard .NET (např. Visual C# .NET nebo Visual C++ .NET). Architektura řešení je zobrazena na obr. 7.



Obr. 7 – Okno **Průzkumníka řešení (Solution Explorer)**

Popišme si nyní okno **Průzkumníka řešení** podrobněji. Jak můžete vidět, všechny součásti vašeho řešení jsou seskupeny do přehledné stromové struktury prvků. V našem případě se v řešení nachází jeden projekt, jenž je standardně pojmenován jako **WindowsApplication1**. Pokud nezadáte při výběru nového řešení váš vlastní název, Visual Basic použije standardní název. Všimněte si, že jméno projektu je zobrazeno tučným písmem. To znamená, že tento projekt se spustí okamžitě poté, co vydáte příkaz na vybudování a spuštění aplikace. Na další úrovni stromové struktury se nachází položka **References**. Tato sdružuje všechny potřebné odkazy na prostory jmen (např. **System**, **System.Data**, **System.Drawing**, **System.Windows.Forms** atd.) či externí komponenty. Výčet pokračuje souborem **AssemblyInfo.vb**. Pokud poklepete na položku **AssemblyInfo.vb**, zobrazí se obsah přidruženého souboru, který uchovává informace o assembly daného řešení. Ukázku podoby tohoto souboru můžete spatřit na obr. 8.



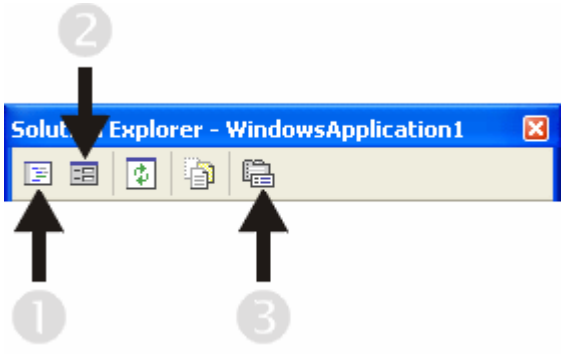
Obr. 8 – Obsah souboru **AssemblyInfo.vb**

Soubor obsahuje množství atributů, které lze měnit, např. **AssemblyTitle**, **AssemblyDescription**, **AssemblyCompany** a mnohé další. V tomto vydání Programátorské laboratoře se můžete naučit, jak modifikovat některé z atributů v souboru **AssemblyInfo.vb**.

Poslední položkou v stromové struktuře **Průzkumníka řešení** je formulář (**Form1**).

V prostředí VB .NET mají všechny soubory se zdrojovým kódem (formuláře, moduly, třídy a další) jednotnou příponu **.vb**.

Na závěr této části bych si dovolil ještě připomenout, že v horní části okna **Průzkumníka řešení** se nachází panel nástrojů s několika tlačítky. Počet přístupných tlačítek se mění v závislosti od vybrané položky v stromové struktuře **Průzkumníka řešení**. Pokud máte vybranou položku s formulářem, panel by měl mít tuto podobu (obr. 9).



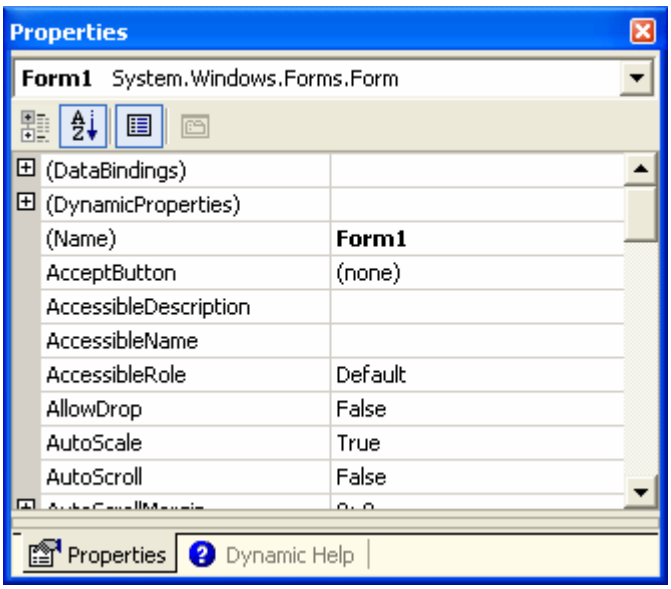
Obr. 9 – Panel nástrojů **Průzkumníka řešení**

My se budeme prozatím soustředit na následující tři tlačítka:

1. View Code	Po aktivaci tlačítka se zobrazí editor pro zápis zdrojového kódu aplikace.
2. View Designer	Klepnete-li na toto tlačítko, zobrazí se podoba formuláře, kterou můžete libovolně měnit (jde o režim návrhu aplikace).
3. Properties	Po stisknutí se zviditelní okno s vlastnostmi souboru formuláře (**Form1.vb**).

Část 6: Okno s vlastnostmi objektů (Properties Window)

Properties Window zobrazuje hodnoty vlastností všech objektů, se kterými v prostředí Visual Basicu pracujete. Ve skutečnosti je Visual Basic .NET plně objektově orientovaným vývojovým nástrojem, a pokud to poněkud přeženeme, můžeme přijít k tvrzení, že „všude, kam se podíváte, je objekt". Na vysvětlenou, objekt je instancí určité třídy a jako taký disponuje svými vlastnostmi a metodami. Rovněž je schopný reagovat na vnější podněty, kterým se ve vizuálním prostředí říká události. Objektem je samotný formulář a objekty jsou také všechny instance ovládacích prvků, které na formulář umístíte. Pro modifikaci hodnot vlastností objektů v režimu návrhu je vám k dispozici speciální okno **Properties Window**, které je zobrazeno na obr. 10.



Obr. 10 – Okno s vlastnostmi objektů (**Properties Window**)

Pokud klepnete na instanci jakéhokoliv ovládacího prvku, seznam jeho vlastností se objeví v tomto okně. Změnu hodnoty příslušné vlastnosti uskutečníte tak, že klepněte myší do pravého pole a zadáte, nebo vyberete požadovanou hodnotu pro danou vlastnost. Vlastnosti mohou být seřazeny podle abecedy (tak je to na obrázku), nebo podle předem definovaných kategorií.

Část 7: Okno Output

Pokud vydáte příkaz pro sestavení aplikace, v tomto okně se zobrazí všechny informace o procesu generace aplikace. Jestliže vše proběhne v pořádku, VB .NET zobrazí v okně **Output** správu „Build: 1 succeeded, 0 failed, 0 skipped".

Část 8: Stavový pruh aplikace

V stavovém pruhu se zobrazují další dodatečné správy, které vám VB .NET v jistých okamžicích zobrazuje. Například při úspěšném sestavení aplikace je vypsán řetězec „Build succeeded".

Integrované prostředí rubriky Visual Basic	Autor: Ján Hanák

Téma

VB .NET

Programátorská laboratoř

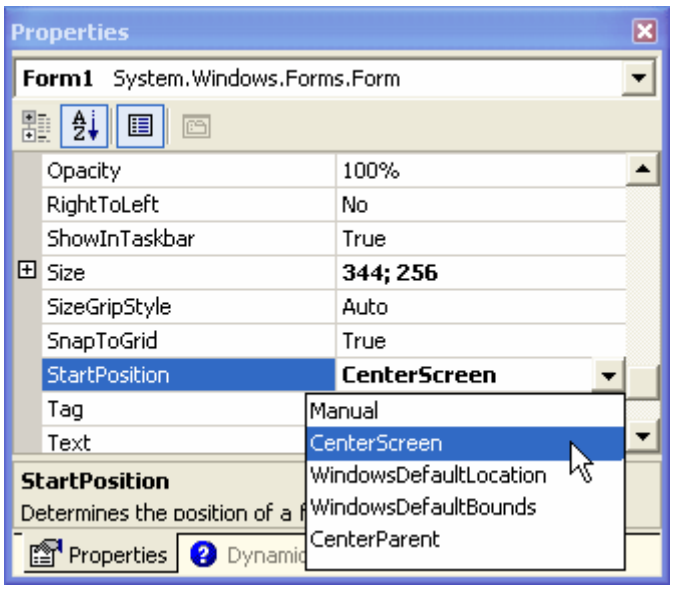
Prostor pro experimentování

Časová náročnost (min):	Začátečník	Pokročilý	Profesionál
60	1	2	3

Použitý operační systém :	Windows XP
Hlavní vývojový nástroj :	Visual Basic .NET 2002
Další vývojový software :	Žádný
Jiný software :	Žádný

Určení startovní pozice pro formulář

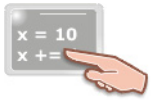
Startovní pozici pro formulář můžete určit vhodným nastavením vlastnosti **StartPosition** v době návrhu aplikace. Pokud například chcete, aby byl formulář při běhu aplikace centrován, nastavte vlastnost **StartPosition** na hodnotu **CenterScreen** (obr. 1).



Obr. 1 – Nastavení vlastnosti **StartPosition** na hodnotu **CenterScreen**

Plynulé zmiznutí formuláře

Jak jsme si v minulém dílu Programátorské laboratoře ukázali, pro změnu průhlednosti formuláře lze vhodně využít jeho vlastnost **Opacity**. V následujícím experimentu si budeme demonstrovat použití této vlastnosti v praxi a vytvoříme tak efekt „plynulého zmiznutí" formuláře při stisknutí tlačítka. Do obsluhy události **Click** tlačítka zapište tento fragment programového kódu:




```
Dim x As Double
For x = 1 To 0 Step -0.1
    System.Threading.Thread.Sleep(50)
    Application.DoEvents()
    Me.Opacity = x
Next x
Environment.Exit(0)
```

Popis kódu
Nejdřív deklarujeme proměnnou **x**, která je typu **Double**. Poté napíšeme konstrukci cyklu **For-Next**, do které umístíme uvedené řádky s kódem. Aby byla každá dekrementace hodnoty vlastnosti **Opacity** viditelná, „uspíme" aplikaci metodou **Sleep** při každém průchodu cyklem na 50 milisekund. Dále použijeme metodu **DoEvents**, která zabezpečí, že aplikace bude i nadále reagovat na vnější podněty. Poslední příkaz uvnitř cyklu upravuje hodnotu **Opacity** aktivního formuláře (**Me**). Poslední řádek kódu pak ukončí exekuci aplikace.

Modifikace souboru AssemblyInfo.vb

Soubor **AssemblyInfo.vb** sdružuje prostřednictvím různých atributů informace o vyvíjené aplikaci. Vhodnou úpravou těchto atributů lze docílit toho, aby některé informace byly viditelné i pro finálního uživatele vaší aplikace. Pokud v prostředí operačního systému Windows XP zvolíte při zobrazování souborů pohled **Tiles**, při ikoně aplikace se budou nacházet i další, dodatečné informace. Na obr. 2 můžete vidět informace o aplikaci Notepad.

Obr. 2 – Podoba ikony aplikace při použití pohledu **Titles**

Chcete-li, aby i vaše aplikace ovládala tuto „vymoženost", udělejte toto:

1. V okně **Solution Explorer** klepněte pravým tlačítkem myši na položku **AssemblyInfo.vb**.
2. Z kontextové nabídky vyberte příkaz **View Code**, na což se obsah souboru otevře v editoru.
3. Upravte hodnoty atributů **AssemblyTitle** a **AssemblyCompany** (obr. 3).

```
<Assembly: AssemblyTitle("Textový procesor")>
<Assembly: AssemblyDescription("")>
<Assembly: AssemblyCompany("Ján Hanák")>
<Assembly: AssemblyProduct("")>
<Assembly: AssemblyCopyright("")>
<Assembly: AssemblyTrademark("")>
<Assembly: CLSCompliant(True)>
```

Obr. 3 – Modifikovaný soubor **AssemblyInfo.vb**

4. Proveďte sestavení aplikace a podívejte se do složky s vygenerovanou assembly. Použijete-li pohled **Tiles**, podoba ikony aplikace by měla zodpovídat té z obr. 4.

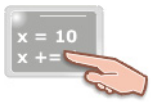
Obr. 4 – Výslední podoba ikony aplikace

Přidání nabídky a položky za běhu programu

V této ukázce si popíšeme postup, jak lze „programově" přidat do projektu hlavní nabídku **Soubor** a dále položku **Nový**, po aktivaci které se zobrazí uživatelská zpráva.

Postupujte dle instrukcí:

1. Založte nový projekt typu **Windows Application**.
2. Na formulář přidejte instanci ovládacího prvku **Button**.
3. Na vytvořenou instanci poklepejte, čímž se vám otevře okno pro zápis programového kódu, do kterého zapište tento kód:



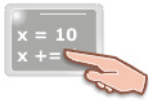

```
Dim menu As New MainMenu()
'Vytvoření hlavní nabídky.

Me.Menu = menu
'Přiřazení hlavní nabídky k aktivnímu formuláři.

Dim Soubor As MenuItem
Soubor = New MenuItem("Soubor")
'Vytvoření nabídky "Soubor".
```

Programový kód pokračuje na následující straně

```
Dim Novy As MenuItem
Novy = New MenuItem("Nový", _
New EventHandler(AddressOf Zobrazit_zpravu))
'Vytvoření podřazené položky "Nový" nabídky "Soubor".

menu.MenuItems.Add(Soubor)
'Přidání nabídky "Soubor" do hlavní nabídky.

Soubor.MenuItems.Add(Novy)
'Přidání položky "Nový" do nabídky "Soubor".
```

Nejdříve ze všeho vytvoříme hlavní nabídku **MainMenu**, která představuje kolekci objektů typu **MenuItem**. Pokračujeme tím, že tuto hlavní nabídku přiřadíme k právě aktivnímu formuláři. Dále vytvoříme nabídku **Soubor** a poté i položku **Nový**, která se bude nacházet v nabídce **Soubor**. Při tvorbě položky **Nový** dále specifikujeme i název procedury (**Zobrazit_zpravu**), která se bude aktivovat vždy, když uživatel klikne na tuto položku. Nakonec do hlavní nabídky (**MainMenu**) přidáme vytvořenou nabídku **Soubor** a do nabídky **Soubor** přidáme položku **Nový**.

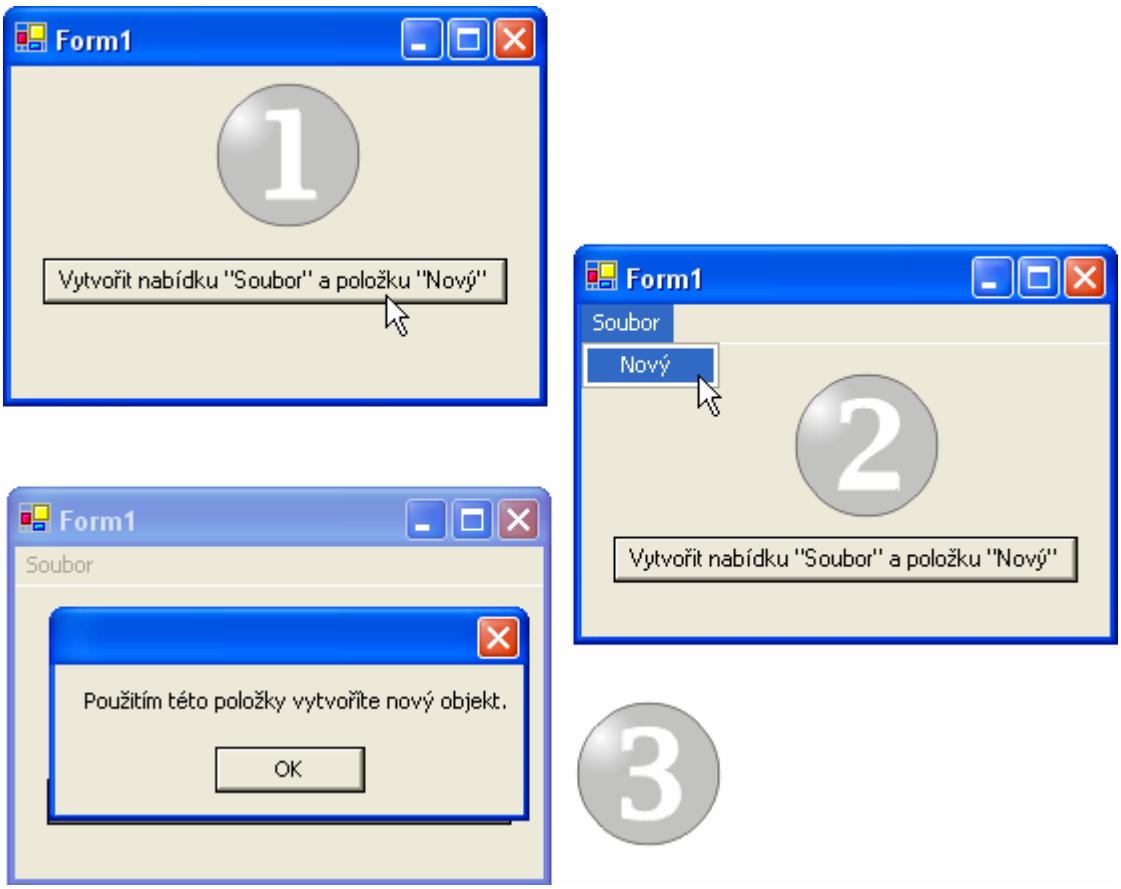
4. Do editoru kódu vložte zápis pro proceduru **Zobrazit_zpravu**:




```
Private Sub Zobrazit_zpravu(ByVal sender As Object, _
ByVal e As System.EventArgs)

MessageBox.Show("Použitím této položky vytvoříte nový objekt.")

End Sub
```

Do procedury **Zobrazit_zpravu** můžete zadat kód podle vašich potřeb, například by zde mohl být vložen kód pro skutečné vytvoření nového souboru či nového objektu.

Když spustíte aplikaci a klepnete na tlačítko, vytvoří se nabídka **Soubor**, která bude obsahovat položku **Nový**. Aktivujete-li tuto položku, zobrazí se zpráva s textem. Situaci ilustruje obr. 5.



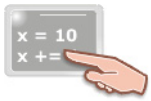
Obr. 5 – Proces přidání nabídky **Soubor** a položky **Nový** na formulář

Ukázka práce Garbage Collection

Princip práce **Garbage Collection** (**GC**) je podrobně popsán v tomto vydání seriálu Začínáme s VB .NET. V této chvíli se ovšem zaměříme na praktickou ukázku činnosti **Garbage Collection**. Vytvoříme několik tisíc instancí ukázkového objektu v paměti počítače a budeme pozorovat, jak se s touto situací vypořádá **Garbage Collection**.

Následujte uvedený postup:

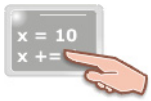
1. Vytvořte nový projekt typu **Windows Application**.
2. Přidejte do projektu **modul třídy** (**Class1.vb**).
3. Upravte kód v modulu třídy do této podoby:




```
Public Class Objekt_01
    Public Shared Instance_objektu As Long

    Public Sub New()
        Instance_objektu += 1
    End Sub

    Protected Overrides Sub Finalize()
        Instance_objektu -= 1
    End Sub
End Class
```

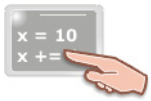
4. Na formulář přidejte instanci ovládacího prvku **Button** a instanci ovládacího prvku **Timer**.

5. Hodnotu vlastnosti **Enabled** prvku **Timer** nastavte na **True** a do vlastnosti **Interval** zadejte hodnotu **50**.
6. Obsluhu události **Click** prvku **Button** vyplňte následujícím programovým kódem:



```
Dim x As Integer, obj
For x = 1 To 5000
    obj = New Objekt_01()
Next x
```

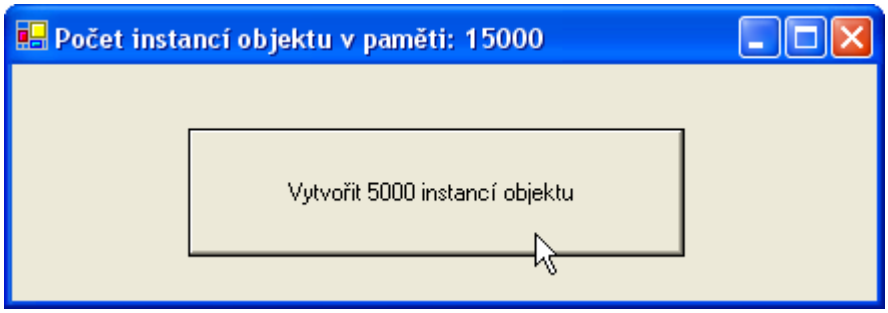
7. Obsluhu události **Tick** prvku **Timer** modifikujte následujícím způsobem:



```
Me.Text = "Počet instancí objektu v paměti: " &
Objekt_01.Instance_objektu
```

8. Proveďte sestavení projektu a spusťte jej.

Vždy, když klepnete na tlačítko, vytvoří se v paměti počítače 5000 instancí objektu **Objekt_01**. Vždy, když je objekt vytvořen, je paralelně inkrementována i proměnná **Instance_objektu** (kód pro inkrementaci se nachází v konstruktoru třídy). Počet vytvořených instancí sleduje v 50milisekundových intervalech ovládací prvek **Timer**, který v titulkovém pruhu okna aplikace také zobrazuje zprávu o počtu platných objektových instancí v paměti počítače (obr. 6).

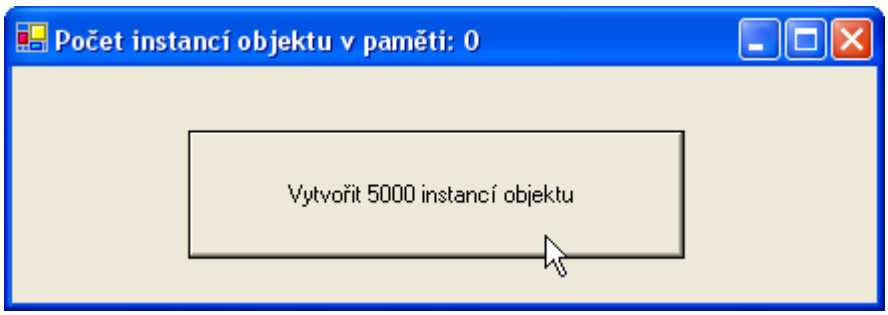


Obr. 6 – Vytvoření 15000 instancí objektu v paměti

V čem tedy spočívá samotné jádro problematiky? **Garbage Collection** sleduje všechny referenční proměnné a rovněž tak reference na jednotlivé instance objektu. Abyste důkladně pochopili tuto situaci, uvažujte o objektové proměnné **obj**, která je deklarována v události **Click** tlačítka. Proměnná **obj** je proměnnou referenčního typu a obsahuje tedy referenci na příslušnou instanci objektu **Objekt_01**. Protože oborem platnosti této proměnné je jenom obsluha události **Click** tlačítka, je ihned po ukončení exekuce programového kódu dané obsluhy události objektová proměnná zlikvidována. V této chvíli dojde i k destrukci reference na instanci objektu **Objekt_01**. To všechno sleduje **Garbage Collection** a zabezpečuje odstranění již nepotřebných instancí objektu z paměti počítače. Při likvidaci instance objektu se provede programový kód umístěný v destruktoru třídy (dojde k dekrementaci proměnné **Instance_objektu**).

Vyzkoušejte tento postup:

Po rozběhnutí aplikace klepněte několikrát na tlačítko a vytvořte dostatečný počet instancí objektu v paměti. Poté chvíli vyčkejte a sledujte titulkový pruh okna aplikace, ve kterém se budou zobrazovat informace o počtu instancí, které se ještě nacházejí v paměti. Po uplynutí jistého časového okamžiku (může jít i o několik minut) by měl počet instancí klesnout na nulu, což je důkazem toho, že **Garbage Collection** odvádí svou práci svědomitě (obr. 7).



Obr. 7 – Konec práce **Garbage Collection** a nulový počet instancí objektu v paměti

Integrované prostředí rubriky Visual Basic	Autor: **Ján Hanák**

Téma měsíce

Grafický subsystém GDI+ (2. díl)

Časová náročnost (min):	Začátečník	Pokročilý	Profesionál
60	1	2	3

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

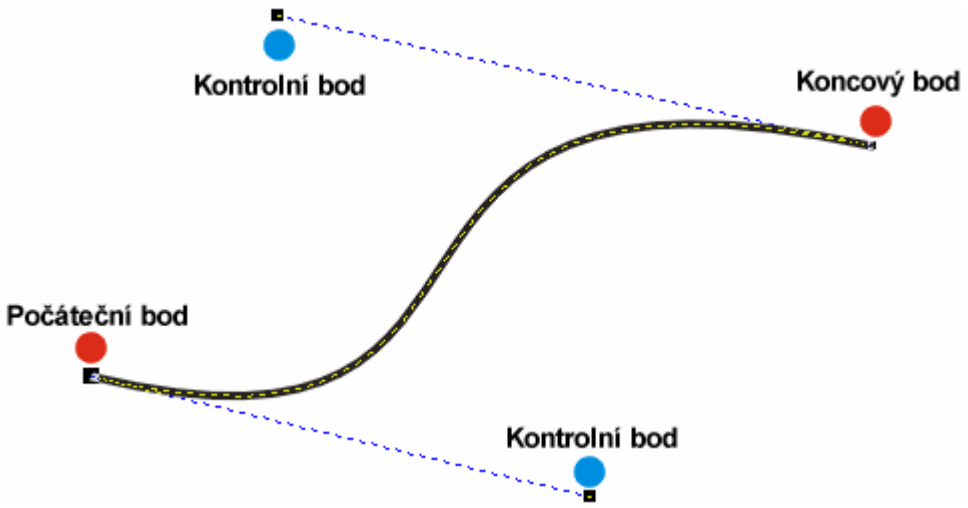
Vážení přátelé,

v minulém vydání jsme si pověděli základní informace ohledně aplikace grafického subsystému platformy .NET, jehož název je GDI+. Naučili jste se, jak vytvořit primární grafický objekt **Graphics** a pomocí něj realizovat kreslení jednoduchých geometrických obrazců a útvarů. V dnešní části se více přikloníme k praktickým ukázkám pokročilých dovedností nového grafického aplikačního rozhraní. Naučíte se pracovat s Bezierovymi křivkami a přijdete na chuť kardinálním spline křivkám. Dále uvidíte, jak pomocí objektu **GraphicsPath** vytvářet pokročilé grafické struktury a kombinovat jednotlivé geometrické útvary. Závěr bude patřit základům kreslení textových řetězců.

Obsah

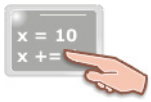
Vytvoření Bezierovy křivky

Bezierova křivka je speciální křivka, která je definována čtyřmi základními body. Jde o počáteční bod, koncový bod a dva kontrolní body. Ukázku křivky můžete vidět na obr. 1.



Obr. 1 – Bezierova křivka

Tvar křivky lze plynule měnit podle pozice jednotlivých kontrolních bodů. Visual Basic .NET vám prostřednictvím grafického subsystému GDI+ umožňuje navrhovat a kreslit vlastní Bezierovy křivky, dokonce i různé série těchto křivek. Pro vytvoření jednoduché Bezierovy křivky použijte tento programový kód:

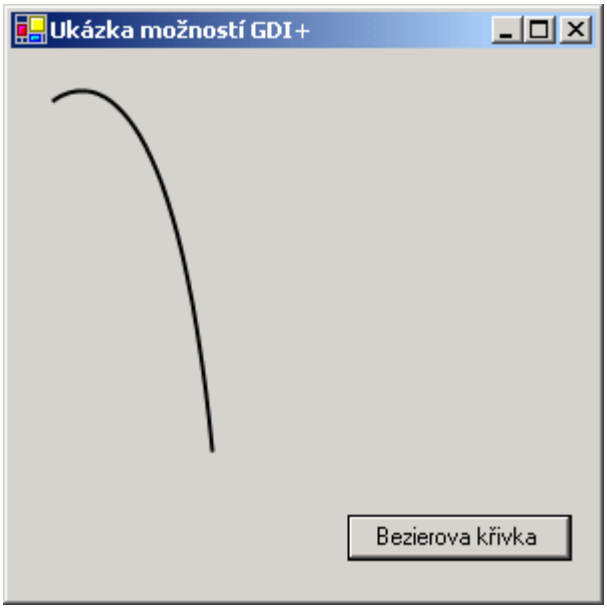



```
Dim pero As New Pen(Color.Black, 2)
Dim grafika As Graphics = Me.CreateGraphics
grafika.SmoothingMode = Drawing.Drawing2D.SmoothingMode.AntiAlias
grafika.DrawBezier(pero, 20, 25, 40, 10, 82, 15, 100, 200)
pero.Dispose()
grafika.Dispose()
```

Z uvedeného kódu vyplývá, že k tomu, abyste vygenerovali požadovanou Bezierovu křivku, potřebujete vytvořit dva objekty: objekt **Pen** pro definici barvy a tloušťky čáry, kterou bude křivka nakreslena a grafický objekt **Graphics**. Poté, co jsou tyto objekty úspěšně vytvořeny, můžete na objekt **Graphics** aplikovat metodu **DrawBezier**, která jako parametry přijímá objekt **Pen** a souřadnice všech bodů, které tvoří křivku. Aby byl grafický výsledek co nejlepší, můžete použít i vlastnost **SmoothingMode** objektu **Graphics** a nastavit její hodnotu na **AntiAlias**. Tím dojde k eliminaci „zubatých" hran, které se objeví při vykreslování Bezierovy křivky.

Pokud se rozhodnete pro použití vlastnosti **SmoothingMode**, musíte její nastavení provést ještě před samotným kreslením křivky (tedy před aplikací metody **DrawBezier**).

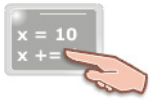
Výsledkem bude Bezierova křivka, která je zobrazena na obr. 2.



Obr. 2 – **Bezierova křivka** s aktivním antialiasingem v prostředí VB .NET

Vytvoření série Bezierovych křivek

Jak jsem se již zmínil, ani zdaleka nejste omezeni pouze na kreslení samostatných Bezierovych křivek. Budete-li chtít vytvářet mnohonásobné série křivek tohoto typu, zcela jistě se střetnete s metodou **DrawBeziers**. Uvedená metoda se může vyskytovat ve dvou různých podobách (jde tedy o přetíženou metodu):



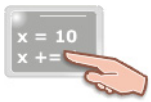
1. **Overloads Public Sub DrawBeziers(Pen, Point())**

2. **Overloads Public Sub DrawBeziers(Pen, PointF())**

Z deklarace obou variant metod je zřejmé, že metoda pracuje se dvěma parametry. Prvním parametrem je objekt **Pen**, funkci kterého již dobře znáte. Pole struktur typu **Point** (resp. **PointF**) pak tvoří druhý parametr metody. Pole struktur obsahuje x-ové a y-ové souřadnice všech bodů, které tvoří celou sérii křivek. Série křivek se vytváří přibližně podle tohoto scénáře:

1. Nejdřív se definují platné souřadnice pro počáteční bod Bezierovy křivky.
2. Dále se určí souřadnice dvou kontrolních bodů křivky.
3. Následně se určí souřadnice koncového bodu křivky.
4. Tímto je nakreslena základní Bezierova křivka. Na vykreslení další křivky je nutné determinovat ještě tři dodatečné body. Dva z těchto bodů působí jako kontrolní body druhé křivky a třetí bod je koncovým bodem této křivky. Platí zásada, že koncový bod jedné Bezierovy křivky je současně počátečním bodem následující křivky.

Pro lepší pochopení celé problematiky bude nejlepší demonstrace tvorby série Bezierovych křivek na praktickém příkladu. Předpokládejme tedy, že budete chtít vytvořit sérii křivek, která bude pozůstávat z třech křivek. Programový kód, jenž řeší nastíněnou situaci by mohl mít tuto podobu:

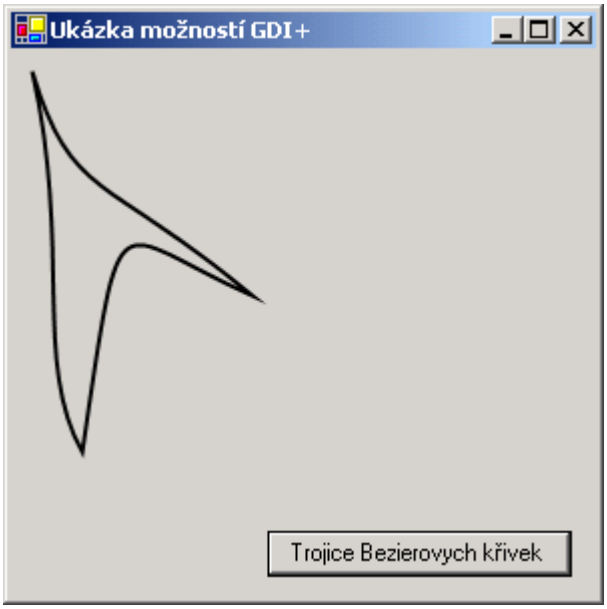


```
Dim pero As New Pen(Color.Black, 2)
Dim grafika As Graphics = Me.CreateGraphics
grafika.SmoothingMode = Drawing.Drawing2D.SmoothingMode.AntiAlias

Dim b1 As New Point(10, 10)
Dim b2 As New Point(30, 75)
Dim b3 As New Point(45, 60)
Dim b4 As New Point(120, 122)
Dim b5 As New Point(48, 90)
Dim b6 As New Point(55, 65)
Dim b7 As New Point(35, 200)
Dim b8 As New Point(10, 155)
Dim b9 As New Point(30, 99)
Dim b10 As New Point(10, 10)

Dim body As Point() = {b1, b2, b3, b4, b5, _
b6, b7, b8, b9, b10}

grafika.DrawBeziers(pero, body)

pero.Dispose()
grafika.Dispose()
```

Pokud budeme abstrahovat od základních aspektů, jako je inicializace potřebných grafických objektů, můžeme zaměřit svou pozornost na definici všech bodů (**b1** až **b10**), které budou tvořit požadovanou sérii křivek. Body **b1** až **b4** určují pozici a tvar první ze tří Bezierovych křivek, které chceme nakreslit. Připomenu, že bod **b4** plní v tomto případě dvojí úlohu; je jednak koncovým bodem první křivky a současně počátečním bodem křivky druhé. Body **b5** a **b6** představují kontrolní body druhé křivky a bod **b7** je koncovým bodem této křivky. Situace u dalších tří bodů je poté analogická: Bod **b7** je počátečním bodem třetí křivky, body **b8** a **b9** jsou jejími kontrolními body a konečně bod **b10** plní funkci koncového bodu této křivky.

Po určení polohy všech potřebných bodů je vytvořeno pole struktur typu **Point** s názvem **body**, do kterého jsou následně vloženy všechny údaje, které se týkají pozic všech operačních bodů. Zlatým

hřebíkem představení je pak zavolání metody **DrawBeziers** vytvořeného objektu **Graphics**. Tato metoda zabezpečí správné vykreslení série Bezierovych křivek, která je zobrazena na obr. 3.

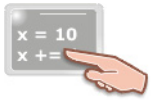


Obr. 3 – Vykreslení **série Bezierovych křivek**

Vytvoření kardinální spline křivky

Charakteristiku kardinální spline křivky jsme rozebírali v minulém dílu, proto jen stručně. Spline křivka je křivka, která vzniká agregací několika dalších křivek, kterých počáteční a koncové body jsou definovány. Jedinečnou vlastností kardinální spline křivky je skutečnost, že místa, kterými se jedna kompletační křivka napájí na jinou křivku jsou bez jakýchkoliv prudkých zlomů, přechod je tedy zcela plynulý. Další zajímavostí je možnost definice faktoru napětí, který stanovuje mez ohýbání kompletačních křivek v jednotlivých „přechodních" místech. Z ryze technického hlediska je tvorba kardinální spline křivky velmi podobná tvorbě série Bezierovych křivek.

Pro vykreslení jednoduché kardinální spline křivky můžete použít tento fragment programového kódu:




```
Dim pero As New Pen(Color.Black, 2)
Dim grafika As Graphics = Me.CreateGraphics
grafika.SmoothingMode = Drawing.Drawing2D.SmoothingMode.AntiAlias

Dim b1 As New Point(45, 77)
Dim b2 As New Point(21, 88)
Dim b3 As New Point(120, 90)
Dim b4 As New Point(110, 155)

Dim body As Point() = {b1, b2, b3, b4}

grafika.DrawCurve(pero, body, 0.65)

pero.Dispose()
grafika.Dispose()
```

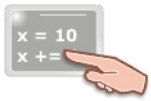
Strůjcem všeho je metoda **DrawCurve** objektu **Graphics**. Tato metoda pracuje se třemi parametry:

1. Objekt **Pen**.
2. Pole bodů, jejichž souřadnice jsou uložené v strukturách typu **Point**.
3. **Faktor napětí** v podobě čísla typu **Single**.

Výsledek práce uvedeného kódu je přehledně znázorněn na obr. 4.

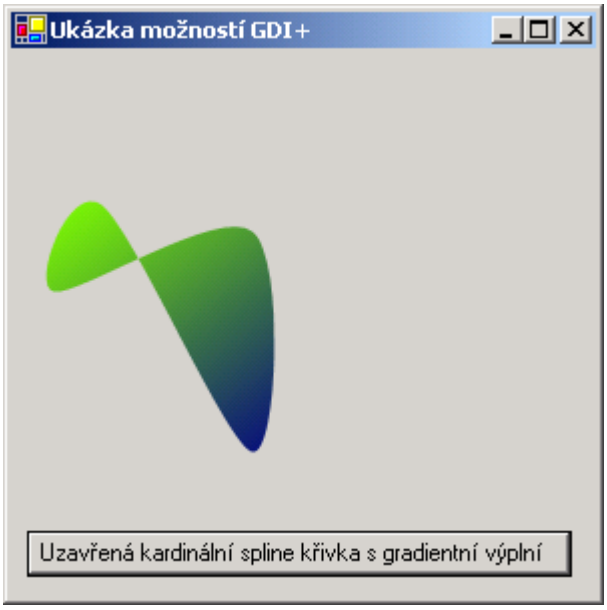
Obr. 4 – Tvorba jednoduché **kardinální spline křivky**

Velice efektních výsledků můžete dosáhnout aplikací metody **FillClosedCurve**, vybarvující plochu, která je ohraničena uzavřenou kardinální spline křivkou. Pokud ovšem není kardinální spline křivka uzavřena (jak je to ve výše uvedeném příkladě), je automaticky přidána další křivka, která celou křivku uzavře. Výplň interiéru vzniklého útvaru realizuje grafický štětec, pomocí kterého můžete vykouzlit různé efekty. Prostudujte si tento programový kód:




```
Dim stetec As New _
System.Drawing.Drawing2D.LinearGradientBrush(New Point(30, 70), _
New Point(130, 210), Color.LawnGreen, Color.Navy)

Dim grafika As Graphics = Me.CreateGraphics
grafika.SmoothingMode = Drawing.Drawing2D.SmoothingMode.AntiAlias

Dim b1 As New Point(45, 77)
Dim b2 As New Point(21, 120)
Dim b3 As New Point(120, 90)
Dim b4 As New Point(120, 200)

Dim body As Point() = {b1, b2, b3, b4}

grafika.FillClosedCurve(stetec, body, 0.65)

stetec.Dispose()
grafika.Dispose()
```

Výsledek operace je znázorněn na obr. 5.



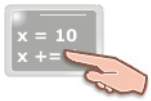
Obr. 5 – Uzavřená kardinální spline křivka s gradientní výplní

Použití objektu GraphicsPath

Pokud jste skutečně nadšeni mnoha vzrušujícími možnostmi, které vám dovoluje grafický subsystém GDI+ realizovat na obrazovce počítače, pravděpodobně budete chtít tvořit co možná nejdříve komplexní grafické obrazce. Pod tímto pojmem se rozumí také obrazce, které jsou tvořeny z několika, nebo i velkého počtu jiných geometrických útvarů. Řečeno poněkud jednodušším jazykem, můžete nakreslit velké množství parciálních obrazců, které následně „zamícháte dohromady", čímž ve skutečnosti vygenerujete zcela nový, komplexní grafický útvar, na který pak můžete aplikovat další grafické efekty. Slovy matematika, jde o sjednocení značného počtu množin grafických obrazců a vytvoření jedné obrovské agregované množiny. Objektem, který má tuto agregaci na starosti, je **GraphicsPath**. Objekt zastupuje roli grafického kontejneru, můžete do něj přidávat jak základní, tak i uživatelsky definované grafické obrazce.

Ve skutečnosti můžete pomocí objektu **GraphicsPath** kombinovat jak kompletní geometrické útvary (obdélníky, elipsy), tak i křivky, či série křivek (Bezierovy křivky, kardinální spline křivky).

Na následujícím příkladě si ukážeme a vysvětlíme, jak uvedený proces „navěšování" geometrických útvarů pracuje. Vytvoříme dvě elipsy, které přidáme do objektu **GraphicsPath**. Poté na vzniklý grafický objekt aplikujeme gradientní výplň.



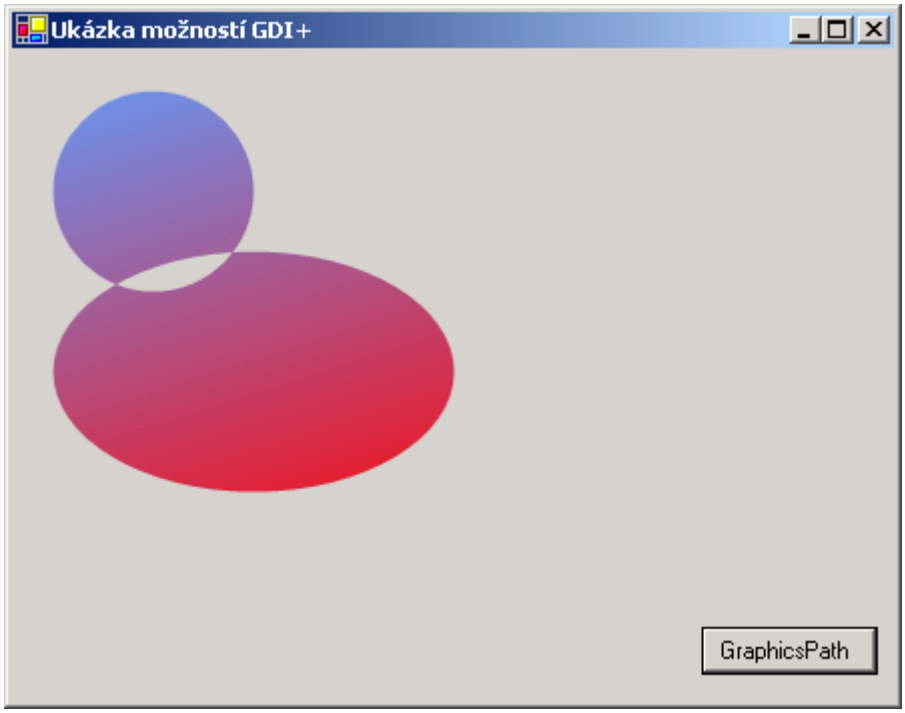

```
Dim grafika As Graphics = Me.CreateGraphics

Dim stetec As New _
System.Drawing.Drawing2D.LinearGradientBrush( _
New Point(10, 20), _
New Point(100, 300), Color.CornflowerBlue, Color.Red)
Dim cesta As New System.Drawing.Drawing2D.GraphicsPath()

cesta.AddEllipse(20, 20, 100, 100)
```

Programový kód pokračuje na následující straně

```
cesta.AddEllipse(20, 100, 200, 120)

grafika.SmoothingMode = Drawing.Drawing2D.SmoothingMode.AntiAlias

grafika.FillPath(stetec, cesta)

cesta.Dispose()
stetec.Dispose()
grafika.Dispose()
```

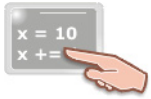
V tomto výpisu programového kódu si prosím všimněte vytvoření objektu třídy **GraphicsPath**. Poté, co vzpomínaný objekt vytvoříme, přidáme do něj pomocí metody **AddEllipse** dvě elipsy. Při definici elips jsou zadány hraniční obdélníky, do prostoru kterých jsou elipsy vykresleny. Uvědomte si, že po procesu přidání elips do grafického kontejneru (přesněji do objektu třídy **GraphicsPath**) vzniká nový grafický útvar, jenž pozůstává z ploch, které ohraničují obě elipsy. Na tento agregovaný útvar pak použijeme gradientní výplň prostřednictvím metody **FillPath** objektu **Graphics**.

Vytvořený obrazec můžete pozorovat na obr. 6.



Obr. 6 – Ukázka možností objektu **GraphicsPath**

Další ukázka, kterou si předvedeme, bude demonstrovat spojení kardinální spline křivky a Bezierovy křivky. Také použijeme nový štětec, jde o štětec se vzorkem (objekt **HatchBrush**) a vzniklý grafický útvar vybarvíme zvoleným vzorem.

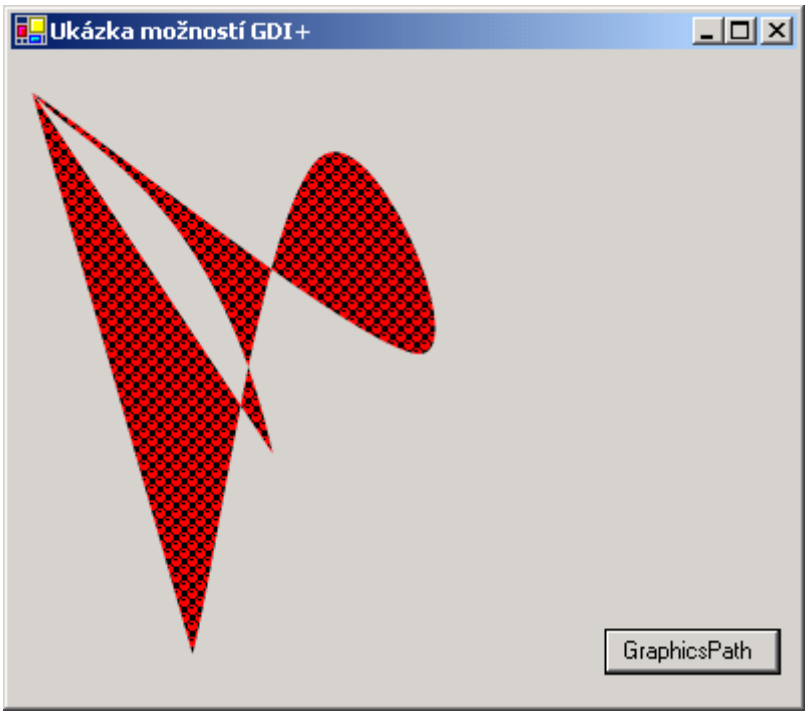



```
Dim stetec_se_vzorkem As New _
System.Drawing.Drawing2D.HatchBrush(Drawing.Drawing2D.HatchStyle.Sphere, _
Color.Red)

Dim grafika As Graphics = Me.CreateGraphics

grafika.SmoothingMode = Drawing.Drawing2D.SmoothingMode.AntiAlias
```

Programový kód pokračuje na následující straně

```
Dim body As Point() = {New Point(10, 20), _
New Point(200, 150), _
New Point(150, 56), _
New Point(90, 300)}

Dim cesta As New System.Drawing.Drawing2D.GraphicsPath()

cesta.AddCurve(body, 0.75)
cesta.AddBezier(New Point(10, 20), _
New Point(30, 45), _
New Point(100, 66), _
New Point(130, 200))

grafika.FillPath(stetec_se_vzorkem, cesta)

stetec_se_vzorkem.Dispose()
grafika.Dispose()
```

Výsledek můžete vidět na obr. 7.

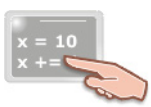


Obr. 7 – Ukázka možností objektu **GraphicsPath** podruhé

Kreslení textového řetězce

GDI+ vám dovoluje snadno kreslit textové řetězce kdekoliv na formulář. Stačí, když vytvoříte objekt **Font** pro určení typu, velikosti a stylu písma a poté použijete metodu **DrawString** objektu **Graphics**. Této metodě předáte několik argumentů: **textový řetězec**, který se má vykreslit, objekt **Font**, **grafický štětec** (v našem případě objekt **LinearGradientBrush**) a nakonec **souřadnice levého horního bodu** (v podobě čísel datového typu **Single**), od kterého se má začít text vykreslovat.

Více prozradí výpis programového kódu:




```
Dim grafika As Graphics = Me.CreateGraphics
```

Programový kód pokračuje na následující straně

```
Dim obdl As New Rectangle(10, 10, 200, 200)

Dim stetec As New _
System.Drawing.Drawing2D.LinearGradientBrush(obdl, _
Color.DodgerBlue, Color.Green, _
Drawing.Drawing2D.LinearGradientMode.ForwardDiagonal)

Dim font As New Font("Times New Roman", 20, FontStyle.Bold)

grafika.SmoothingMode = Drawing.Drawing2D.SmoothingMode.AntiAlias

grafika.DrawString("Ukázka renderování textu", _
font, stetec, 10.0F, 10.0F)

font.Dispose()
stetec.Dispose()
grafika.Dispose()
```

Ukázka použití kódu:

Obr. 8 – Kreslení textového řetězce pomocí metody **DrawString**

Integrované prostředí rubriky Visual Basic	Autor: **Ján Hanák**

Lab

Téma

Začínáme s VB .NET

Úvod do světa .NET (3. díl)

Časová náročnost (min):	Začátečník	Pokročilý	Profesionál
60	1	2	3

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

Vážení čtenáři,

v dnešní části seriálu vytvoříme konečně první plnohodnotní aplikaci pro Windows. Vysvětlíme si proces, jakým se ve Visual Basicu .NET vytvářejí aplikace, a to jak z teoretické, tak i z praktické stránky. Také získáte přehled o základních pojmech, které se s vývojem aplikací pojí. Předmětem naší lekce bude vytvoření tradiční aplikace, která bude zobrazovat okno se zprávou. Přeji příjemné čtení.

Obsah

Tvoříme první aplikaci pro Windows (teoretická část)

Pokud jste absolvovali naše předchozí sezení, měli byste v této chvíli mít základní přehled o vývojovém prostředí Visual Basicu .NET. V této kapitole si ukážeme, jak vytvořit jednoduchou aplikaci pro Windows. Protože snad všechny učebnice a knihy o programování uvádějí své studenty do problematiky vytvořením aplikace typu „Ahoj světe!", ani my nebudeme „předělávat" tradiční zvyky a tuto aplikaci vytvoříme také.

Programovací jazyk Visual Basic již od své počáteční verze nabízel (v porovnání s jinými jazyky, jako například C nebo C++) poněkud netradiční styl vývoje aplikací. Tato vskutku inovační koncepce vychází ze samotného jádra stylu vizuálního vývoje, na filozofii kterého celý VB .NET stojí. Budeme-li chtít činnosti spjaté s procesem vývoje aplikace v tomto programovacím jazyce graficky znázornit, můžeme použít vývojový diagram se třemi částmi. Zobrazení tohoto diagramu můžeme vidět na obr. 1.

Obr. 1 – Schéma vývoje aplikací ve VB .NET

Popišme si nyní jednotlivé části detailněji.

Krok 1: Umístění instance ovládacího prvku na formulář
Hlavním stavebním prvkem aplikací pro Windows jsou tzv. ovládací prvky. Tyto se nacházejí v soupravě nástrojů, ve které jsou přehledně seskupeny. Každý ovládací prvek je vizuálně prezentovaný grafickou ikonou a svým názvem. Souprava nástrojů sdružuje všechny podstatné ovládací prvky, které budete při své práci nejčastěji využívat.

Ve skutečnosti umožňuje souprava nástrojů také import dalších (externích) ovládacích prvků, které můžete ve svých aplikacích rovněž použít. Tyto externí ovládací prvky se někdy označují i jako „ovládací prvky třetích stran", čímž je taktně poukázáno na skutečnost, že tyto prvky vytvářejí softwarové společnosti, nebo počítačoví nadšenci, kteří je v zápětí nabízejí na trhu ovládacích prvků.

Předpokládejme, že budete chtít umístit na formulář instanci jednoho ovládacího prvku (třeba prvku **Button**).

Pod pojmem „**instance** ovládacího prvku" rozumíme **kopii** ovládacího prvku. Rovněž tak slovním spojením „umístění instance ovládacího prvku na formulář" je míněno vybrání ovládacího prvku v soupravě nástrojů a vložení kopie tohoto prvku na formulář jednoduchým nakreslením obrysů daného prvku.

V tomto případě vyberete kýžený ovládací prvek se soupravy nástrojů a nakreslíte jeho obrysy na plochu formuláře. Ihned po ukončení „kreslící" fáze je instance prvku vložena na plochu formuláře. Po vložení je instance ovládacího prvku dále editovatelná, můžete kupříkladu měnit její rozměry nebo pozici na ploše formuláře.

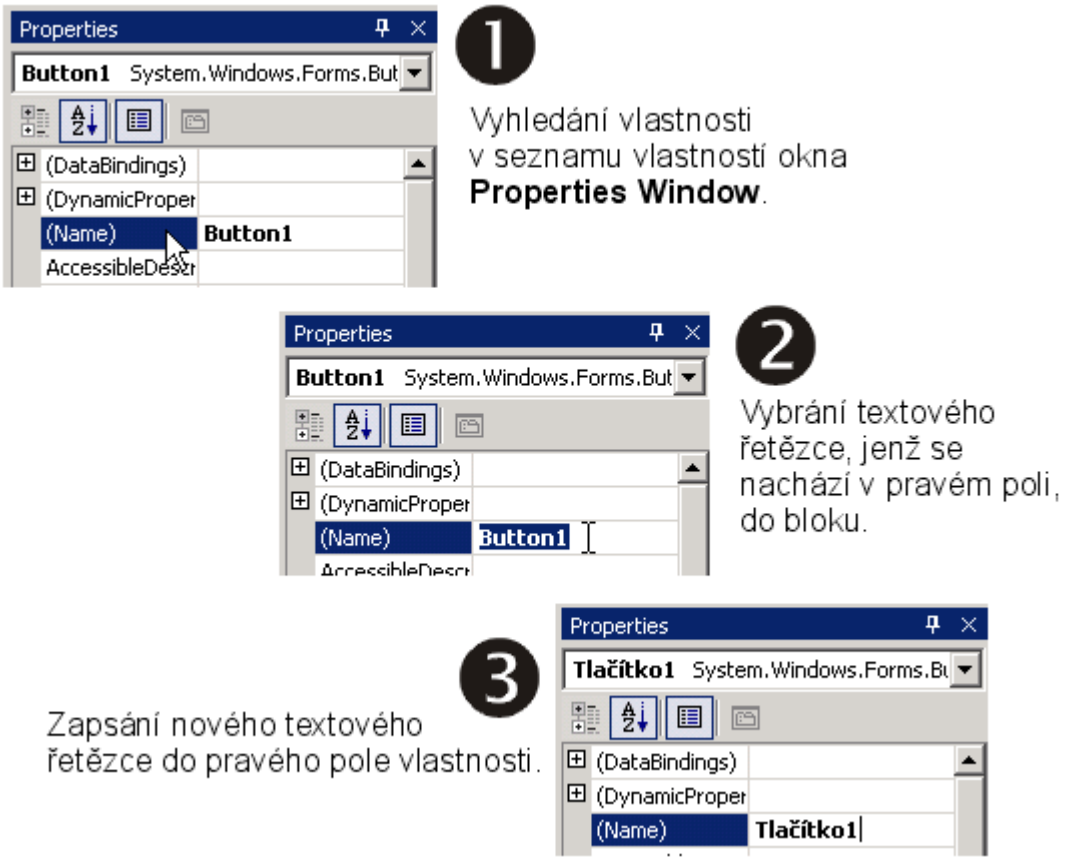
Tento proces jsme si podrobně popsali již v minulé části, a proto předpokládám, že vkládání instancí ovládacích prvků na formulář vám již nečiní žádné potíže.

Většina přístupných ovládacích prvků je v režimu běhu aplikace viditelná na formuláři. Umístíte-li na formulář instanci ovládacího prvku **Button**, a poté aplikaci spustíte, je tlačítko viditelné a nachází se právě tam, kde jste jej „nakreslili". Na druhou stranu se však můžete střetnout i s ovládacími prvky, jejichž instance nejsou za běhu aplikace přímo viditelné na ploše formuláře (tyto jsou známé rovněž pod názvem **komponenty**). Příkladem je ovládací prvek **Timer**, jenž se používá pro vykonávání programových operací v předem stanovených časových intervalech. Použijete-li ve své aplikaci tento ovládací prvek, jeho instance se umístí na takzvaný **podnos součástí**, nebo **podnos komponent**. Podnosem komponent se míní speciální okno, které se nachází pod plochou pro návrh podoby formuláře.

Krok 2: Nastavení vlastností instance ovládacího prvku
Jestliže již máte na formuláři umístěnou instanci požadovaného ovládacího prvku, můžete měnit její vlastnosti. Bezprostředný výklad pojmu **vlastnost** je v ponímání jazyka Visual Basic .NET poněkud komplikovanější. Budeme-li abstrahovat od exaktní definice vlastnosti z pohledu objektově orientovaného programování, můžeme říct, že vlastnosti představují jisté charakteristiky ovládacích prvků, které lze nastavit vhodným způsobem. Každá instance ovládacího prvku, kterou na plochu formuláře umístíte, musí být jednoznačně identifikována. Na tuto identifikaci instance poslouží její **vlastnost Name**.

Seznam všech vlastností, kterými instance ovládacího prvku disponuje, se zobrazují v okně vlastností (**Properties Window**).

Pokud se vám nebude líbit standardní název instance ovládacího prvku, který instanci implicitně přidělil Visual Basic .NET, nic vám nebrání v tom, abyste hodnotu této vlastnosti upravili, nebo dokonce zcela změnili. Jednoduše v okně **Properties Window** vyhledáte požadovanou vlastnost, klepnete do pravého pole vlastnosti a změníte její hodnotu. Předpokládejme, že budete chtít změnit standardní název (**Button1**) vlastnosti **Name** instance ovládacího prvku **Button**. Budete postupovat tak, jak je to ukázáno na obr. 2.



Obr. 2 – Proces změny hodnoty vlastnosti **Name** ovládacího prvku **Button**

I když samozřejmě můžete modifikovat hodnotu vlastnosti **Name** podle svého uvážení, po několika týdnech či měsících praxe s programovacím jazykem a prostředím zjistíte, že bude velmi vhodné, když budete instance ovládacích prvků pojmenovávat podle jistého systému. Význam systematického pojmenovávání se uplatní zejména v případě, kdy se na formuláři bude nacházet větší množství instancí různých ovládacích prvků. V dřívějších dobách se jednoznačně doporučovala speciální notace, kdy se před samotné jméno instance prvku umísťoval zpravidla třípísmenný prefix, sloužící na snadnější identifikaci té-které instance. Například pro tlačítko by podoba názvu mohla vypadat takto: **btnTlačítko_OK**. Pochopitelně, že použití této notace je dobrovolné, pokud se vám nelíbí, můžete si vymyslet „tu svou", podle které se budete řídit.

Pokud nevidíte celý název vlastnosti, kterou si přejete editovat, umístěte nad název vlastnosti kurzor myši a chvíli posečkejte. Za okamžik se v bublinovém okně objeví plný název zvolené vlastnosti.

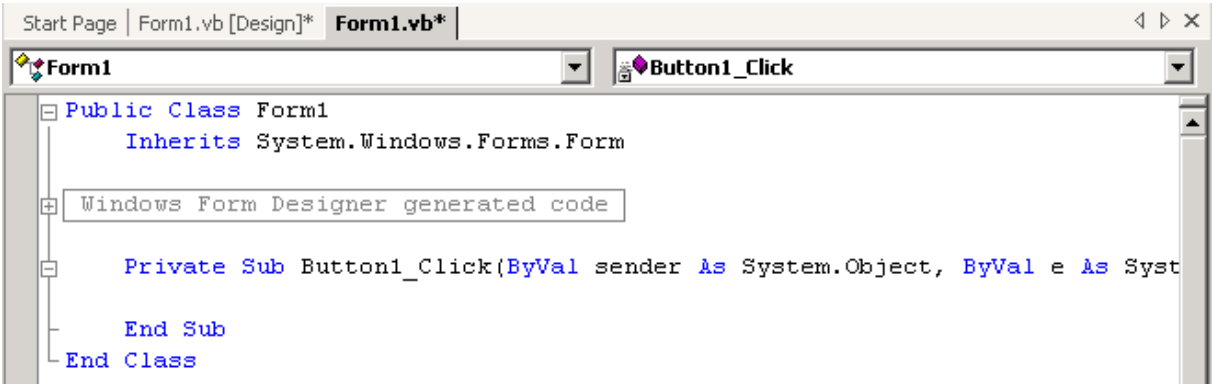
Krok 3: Psaní programového kódu

Na první pohled by se mohlo zdát, že programování ve vizuálním jazyce se obejde bez poctivého psaní programových konstrukcí. I když vývoj ve stylu práce programovacích nástrojů se žene mílovými kroky vpřed, etapě psaní programového kódu se díkybohu (ještě) nevyhneme. Programový kód vdechuje jinak poněkud mrtvým instancím ovládacích prvků život. Pomocí kódu můžete určit, co se uskuteční poté, co uživatel klepne na tlačítko, zvolí položku nabídky, nebo provede jinou akci. Psaní kódu je rovněž nejnáročnější a také nejdůležitější činností v životě programátora.

Tvoříme první aplikaci pro Windows (praktická část)

Po nezbytném teoretickém výkladu problematiky se můžeme vrhnout na její praktické znázornění. Na následujících řádcích navrhneme jednoduchou aplikaci, která bude složena z formuláře a jednoho tlačítka. Jakmile uživatel klepne na tlačítko, zobrazí se zpráva s textem. Postupujte podle uvedených instrukcí:

1. Spusťte Visual Basic .NET a vytvořte novou aplikaci typu **Windows Application**.
2. Na formulář umístěte instanci ovládacího prvku **Button**.
3. Poklepejte na vytvořenou instanci, abyste získali přístup do okna editoru pro zápis programového kódu. Okno editoru je znázorněno na obr. 3.



Obr. 3 – Okno **editoru** pro zápis programového kódu

Okno editoru je rozděleno do tří hlavních částí. První část tvoří záložka okna, ve které se nachází název souboru s formulářem (**Form1.vb**). Dva otevírací seznamy tvoří základ druhé části okna. V levém seznamu je vybrána položka **Form1**, která představuje celkový pohled na formulář (přesněji řečeno na třídu formuláře). Náplní pravého seznamu je položka **Button1_Click**, která identifikuje speciální proceduru typu **Sub**. Programový kód, jenž se nachází v těle této procedury bude spuštěn pokaždé, když dojde k aktivaci události **Click** instance ovládacího prvku **Button**.

Pro pochopení informačního smyslu předchozí věty bude nutné, abyste porozuměli několika zásadním termínům technologie vizuálního programování. Jde o termíny **událost** a **procedura**.

Pojmem **událost** se označuje uskutečnění jisté operace (zejména ze strany uživatele), která podnítí vznik nějaké akce. Událostí je například klepnutí myší na tlačítko, vybrání položky ze seznamu, nebo zavření formuláře. To všechno jsou příklady událostí, které může uskutečnit uživatel.

Druhá část definice zdůrazňuje provedení jisté akce jako reakce na vzniklou událost. Touto akcí je programový kód, jenž se nachází v příslušné přidružené programové **proceduře**. Ano, s každou událostí je spjata procedura, programový kód které se provede vždy, když k této události dojde.

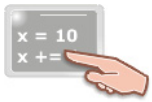
Bližší charakteristika procedury odhalí fakt, že se jedná o soukromou **Sub proceduru**.

Sub procedura je deklarována jako **soukromá**, což naznačuje použití klíčového slova **Private**. Ona soukromost procedury znamená, že k této metodě lze přistupovat jenom z jistého kontextu (tím je v tomto případě třída formuláře).

Sub procedury jsou typické tím, že nenavracejí žádnou hodnotu, jenom provádějí programový kód, který se nachází v jejich těle (tělem procedury se rozumí prostor mezi řádky **Private Sub** a **End Sub**). **Sub** procedury rovněž pracují s několika parametry, no do těchto dálav se pouštět ještě nebudeme.

OK, víme, že **Sub** procedura s názvem **Button1_Click** se bude spouštět vždy poté, co uživatel klepne na tlačítko. Aby bylo možné zobrazit zprávu s textem, musíme do procedury zapsat programový kód. Tato úloha je předmětem dalšího kroku postupu.

4. Mezi řádky **Private Sub** a **End Sub** procedury **Button1_Click** vložte tento řádek kódu:




```
MessageBox.Show("Ahoj světe!")
```

Pro zobrazení okna se zprávou použijeme **metodu Show** třídy **MessageBox**. Protože jsme ještě neprobírali základy objektově orientovaného programování, nemůžeme si exaktně vysvětlit pojem třída. Proto si můžete prozatím třídu **MessageBox** představit jako „černou skříňku". Nebojme se rozvinout tuto filozofii a povězme, že tato černá skříňka obsahuje metodu **Show**.

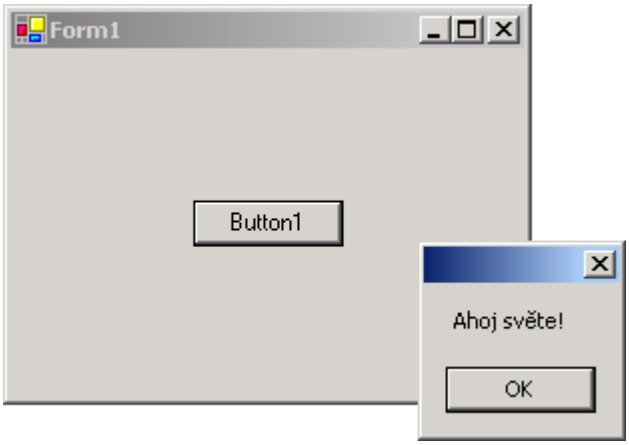
Metoda je speciální sada programového kódu, která provádí jistou prospěšnou činnost. V našem případě metoda **Show** zabezpečuje zobrazení dialogového okna s textem.

Všimněte si, že vztah „černá skříňka obsahuje metodu" je v programovém kódu vyjádřen pomocí **tečkového operátoru** (**.**).

Text, přesněji textový řetězec, jenž chceme zobrazit, je metodě předán ve formě argumentu. Podstatu argumentů si lépe vysvětlíme tehdy, až budeme rozumět pojmu proměnná. Zatím si argument můžete představit jako „vstupnou surovinu" pro potřeby metody.

5. Výborně, to je všechen kód, jenž potřebujeme napsat. Teď nám zbývá jenom sestavit a spustit aplikaci. Automatické sestavení a spuštění aplikace docílíte stisknutím klávesy **F5**, nebo klepnutím na tlačítko **Start** (), které se nachází na standardním panelu tlačítek.

Pokud stiskněte tlačítko, objeví se okno se zprávou, které by mělo vypadat jako to na obr. 4.



Obr. 4 – První aplikace pro Windows v akci

Zatímco jste napsali jenom jeden řádek programového kódu, VB .NET musí sám uskutečnit mnoho operací pro vytvoření hlavního okna aplikace a instance ovládacího prvku. Výtečnou vlastností Visual Basicu je, že značné množství vaší programátorské práce převezme na svá ramena. Vy se proto můžete plně soustředit na návrh a programování hlavního účelu vyvíjené aplikace. Jenom pro zajímavost: Pokud byste chtěli napsat aplikaci Windows v čistém Céčku s pomocí Win32 API, potřebovali byste jenom pro tvorbu hlavního okna aplikace několik desítek řádků kódu. Díky Bohu za Visual Basic!

Výborně, dokázali jste vytvořit vaší první aplikaci a k tomu všemu vám stačilo napsat jenom jeden řádek programového kódu. Na druhou stranu zase musíte uznat, že smysl aplikace je ve „skutečném" světě programování takřka mizivý (co bychom také mohli čekat od jednoho řádku kódu). Nicméně, první krok na cestě ke ovládnutí jazyka Visual Basic .NET jste udělali, a to je důležité.

Integrované prostředí rubriky Visual Basic	Autor: Ján Hanák

Téma

VB .NET

Programátorská laboratoř

Prostor pro experimentování

Časová náročnost (min):	Začátečník	Pokročilý	Profesionál
60	1	2	3

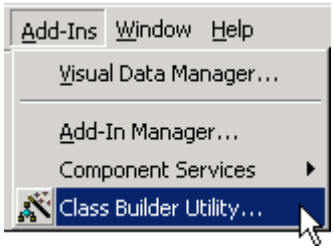
Použitý operační systém :	Windows 2000 SP3
Hlavní vývojový nástroj :	Visual Basic 6.0 SP 5
Další vývojový software :	Visual Basic .NET 2002
Jiný software :	Žádný

Použití aplikace Class Builder za účelem vytváření tříd

Pokud patříte mezi pokročilé uživatele a chtěli byste si zjednodušit námahu při práci s vytvářením tříd, kolekcí, metod, vlastností či událostí, je vám k dispozici chytrá aplikace s názvem **Class Builder**. Tato aplikace je plně integrována do IDE Visual Basicu 6.0, no ještě předtím, než ji budete moci přímo vyvolat z nabídky, musíte Visual Basicu sdělit, aby ji do IDE přidal. To provedete takto:

1. Spusťte Visual Basic 6.0 a vytvořte aplikaci typu **Standard EXE**.
2. Vyberte nabídku **Add-Ins** a klepněte na položku **Add-In Manager**.
3. V seznamu najděte položku **VB 6 Class Builder Utility**.
4. V rámci **Load Behavior** zatrhněte volby **Loaded/Unloaded** a **Load on Startup**.
5. Klepněte na tlačítko OK.

Ihned poté se do nabídky **Add-Ins** přidá položka **Class Builder Utility**, po stisknutí které se spustí zmiňovaná aplikace (obr 1.).

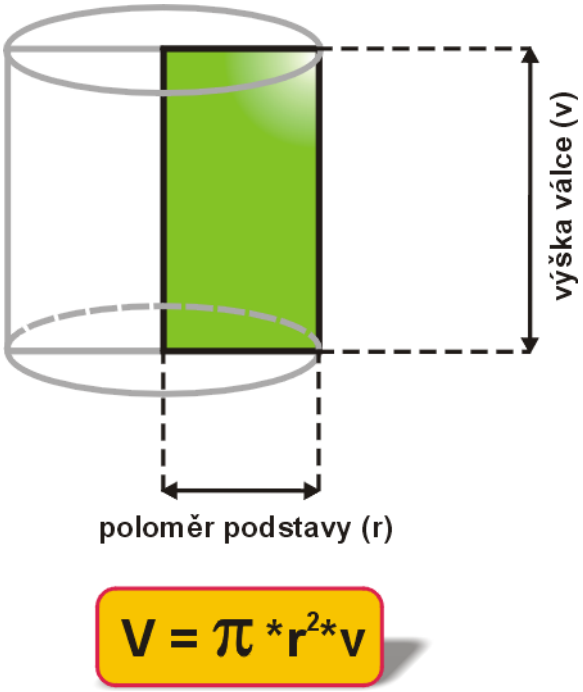


Obr. 1 – Přidání položky **Class Builder Utility** do nabídky **Add-Ins**

Jakmile spustíte aplikaci, přivítá vás její hlavní okno. Abyste nabyli základní zkušenost s tvorbou třídy prostřednictvím tohoto programu, předvedeme si příklad, v němž sestavíme třídu s jednou metodou pro výpočet objemu rotačního válce.

Pokud jste již zapomněli, co to vlastně rotační válec je, zde je malý výlet do matematiky.

Rotační válec vnikne rotací obdélníka o 360 stupňů kolem jedné jeho strany.



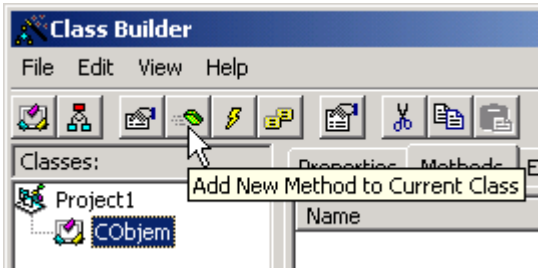
Obr. 2 – Podoba rotačního válce a vzorec pro výpočet jeho objemu

Postupujte podle uvedených instrukcí:

1. Vyberte nabídku **File**, ukažte na položku **New** a klepněte na položku **Class**.
2. VB zobrazí okno **Class Module Builder**.
3. Ujistěte se, že je vybrána záložka **Properties**.
4. V poli **Name** zadejte jméno pro třídu, kterou si přejete vytvořit (v našem případě to může být třeba název **CObjem**). Když jste s názvem spokojeni, stiskněte tlačítko OK.

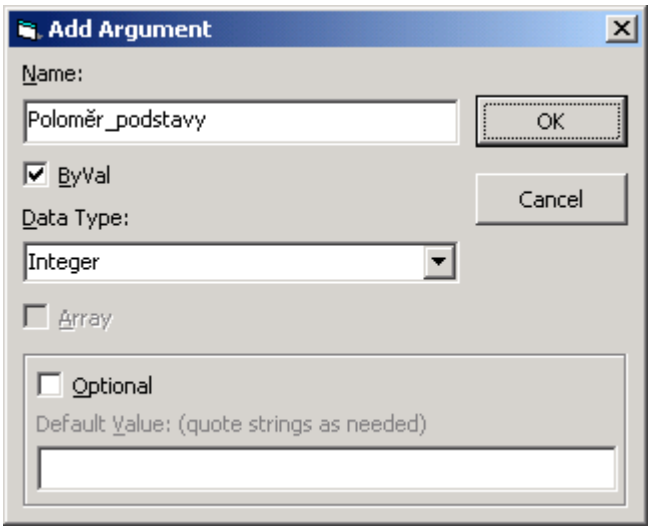
Pokud chcete zadat i stručnou charakteristiku třídy, ještě před klepnutím na tlačítko OK vyberte záložku **Attributes**. Zde se nachází textové pole **Description**, do kterého můžete zapsat informace ohledně vyvíjené třídy.

5. Všimněte si, že v stromové struktuře, která se nachází v levé části okna aplikace, se vytvořila větev s třídou **CObjem**. Aby třída nabízela alespoň základní funkčnost, musíme ji opatřit metodou.
6. Přidání metody zahájíte nejrychleji aktivací tlačítka **Add New Method to Current Class**, které můžete najít na panelu tlačítek (obr. 3).



Obr. 3 – Znázornění tlačítka pro přidání metody do vytvořené třídy

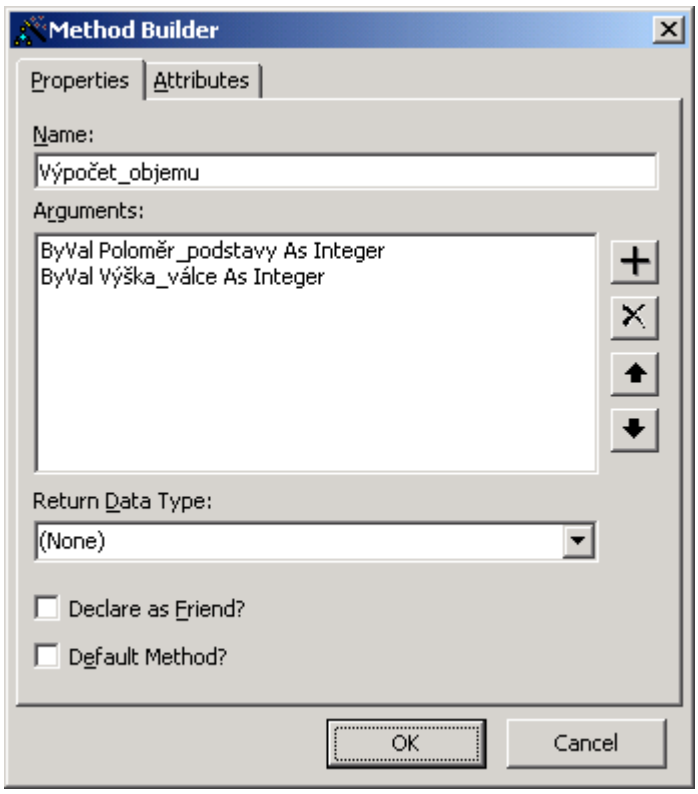
7. Po aktivaci vzpomínaného tlačítka se objeví dialogové okno **Method Builder**.
8. Do pole **Name** zadejte název pro metodu (**Výpočet_objemu**).
9. Protože naše metoda bude pracovat s dvěma parametry, musíme je vytvořit. Proto pokračujte stisknutím tlačítka **Add New Argument** (+).
10. Objevivší se okno **Add Argument** upravte do podoby uvedené na obr. 4.



Obr. 4 – Přidání prvního parametru metody

Jméno prvního parametru je **Poloměr_podstavy**. Dále je specifikováno, že hodnoty, které bude parametr přijímat budou předávány hodnotou (zatržené pole **ByVal**). Protože nebudeme vyžadovat přílišnou přesnost, můžeme parametr deklarovat v podobě datového typu **Integer**. Pokud jste všechny položky vyplnili, stiskněte tlačítko OK.

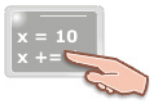
11. Opakujte postup z předchozího bodu pro přidání dalšího parametru, tentokrát s názvem **Výška_válce** (všechny ostatní náležitosti budou stejné). Jestliže jste úkol úspěšně splnili, dialogové okno **Method Builder** by mělo vypadat jako na obr. 5.



Obr. 5 – Obraz okna **Method Builder** pro přidání dvou parametrů

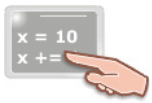
I při tvorbě metody je vám k dispozici záložka **Attributes** s polem **Description**, do kterého můžete zapsat stručný popis metody.

12. Naše metoda, resp. funkce bude vracet i návratovou hodnotu ve tvaru celého čísla (**Integer**). Ze seznamu **Return Data Type** proto vyberte položku **Integer**. Nakonec klepněte na tlačítko OK.
13. Abyste výsledek aplikace **Class Builder** promítli do Visual Basicu, vyberte menu **New** a zvolte příkaz **Update Project**. Posléze zavřete okno aplikace **Class Builder** a otevřete vytvořenou třídu (s názvem **CObjem**).
14. Ujistěte se, že je vybrána metoda, resp. funkce **Výpočet_objemu** a do jejího těla zapište tyto příkazy:




```
Výpočet_objemu = 3.14 * Poloměr_podstavy * _
Poloměr_podstavy * Výška_válce
```

15. Zavřete okno s programovým kódem metody, přemístěte se na formulář a umístěte na něj instanci ovládacího prvku **CommandButton**.
16. Na vytvořenou instanci poklepejte, čímž vyvoláte okno pro zápis programového kódu. Událostní proceduru **Click** vyplňte takto:

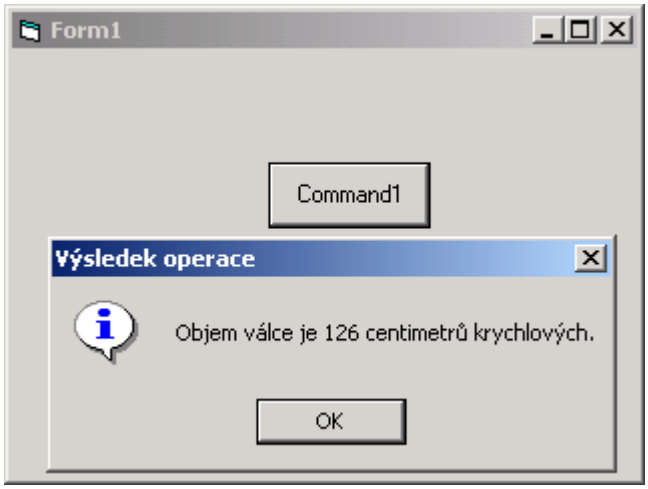



```
Private Sub Command1_Click()
      Dim obj_CObjem As CObjem
```

Programový kód pokračuje na následující straně

```
        Set obj_CObjem = New CObjem
        Dim odp As Integer

        odp = MsgBox("Objem válce je " & _
        obj_CObjem.Výpočet_objemu(2, 10) & _
        " centimetrů krychlových.", _
        vbInformation, "Výsledek operace")
End Sub
```

Pokud chcete vědět, jaký objem má rotační válec, jehož poloměr podstavy se rovná 2 centimetrům a výška 10 centimetrům, podívejte se na obr. 6.



Obr. 6 – Finální podoba experimentu

Vytvoření šablony z libovolného formuláře

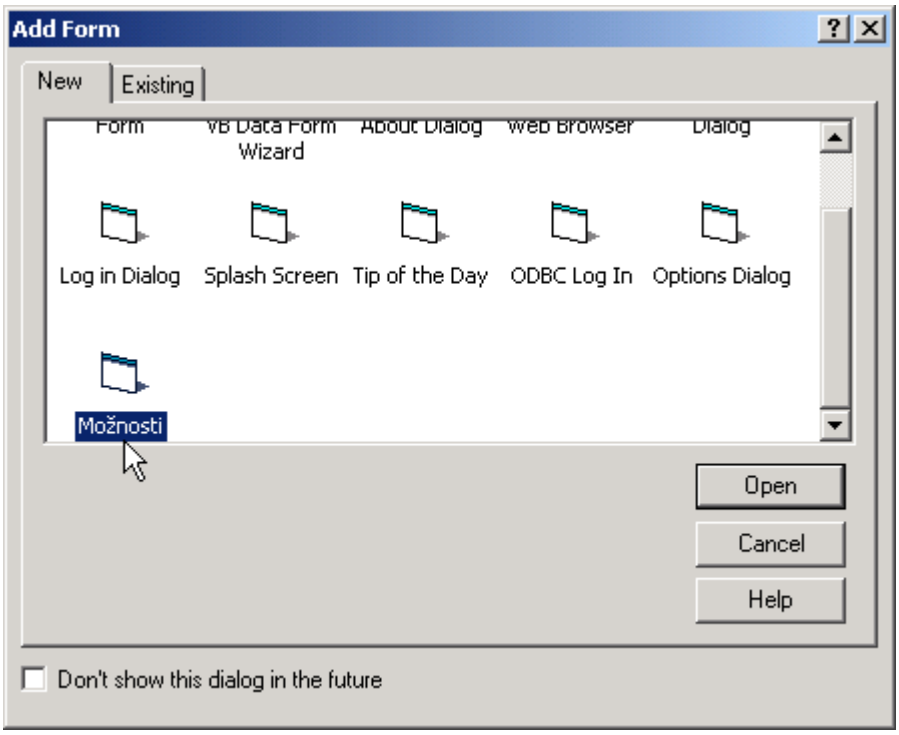
Tento experiment bude vhodný obzvláště pro ty z vás, kteří při návrhu aplikace používáte často speciální druh formuláře. Myslím, že budete se mnou souhlasit, když prohlásím, že neustále sestavování formulářů kopírovací metodou je poněkud zdlouhavé a nepraktické. Proto si pomůžeme malým trikem. Vytvoříme formulář, jenž bude představovat typ, který ve svých aplikacích potřebujete implementovat. Předpokládejme, že půjde o formulář, jenž bude uživateli zobrazovat okno s různými možnostmi (obr. 7).

Obr. 7 – Formulář, jenž poslouží jako šablona

Takto navržený formulář můžete „zaplnit" programovým kódem, tedy můžete napsat kód pro ty událostní procedury, o kterých si myslíte, že jejich kód se v budoucnosti nebude modifikovat (jde o jistý způsob pro zabezpečení znovupoužitelnosti programového kódu) .

Formulář pojmenujte jako **frmMožnosti** a uložte jej pod jménem **Možnosti**. Následně soubor s formulářem (**Možnosti.frm**) a jeho binárním ekvivalentem (**Možnosti.frx**) překopírujte do složky **Forms**, která se nachází ve složce **Template** (kompletní cesta: složka **Visual Studio\VB98\Template\Forms**). Dále postupujte takto:

1. Vytvořte novou aplikaci typu **Standard EXE**.
2. Zvolte nabídku **Project** a klepněte na položku **Add Form**.
3. Objeví se dialogové okno **Add Form**.
4. Ujistěte se, že je vybrána záložka **New** a vyhledejte ikonu s názvem **Možnosti** (obr. 8).



Obr. 8 – Přidání šablony formuláře do nového projektu

5. Poklepete-li na ikonu, formulář s možnostmi se přidá do projektu. Nyní již stačí jenom důkladně zakomponovat vložený formulář do logiky vyvíjené aplikace.

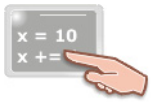
Opatření ovládacího prvku ikonou, která se bude zobrazovat v soupravě nástrojů

Pokud vyvíjíte ovládací prvek, resp. aplikaci typu **Windows Control Library**, možná, že jste se dostali do potíží ve chvíli, kdy jste chtěli zvolit vaší ikonu pro ovládací prvek, která by se zobrazovala v soupravě nástrojů klientské aplikace. Ve Visual Basicu 6.0 byla za tímto účelem vytvořena vlastnost **ToolBoxBitmap**, která se nacházela přímo v okně s vlastnostmi objektů. Možná, že jste i ve VB .NET hledali podobnou vlastnost, ovšem nepodařilo se vám ji najít.

Problematika přiřazení vhodné ikony ovládacího prvku pro soupravu nástrojů je v tomto jazyce o poznání komplikovanější. Abyste určili bitmapu (s rozměry 16 krát 16 pixelů), která bude hrát roli ikony, musíte využít nabídku třídy **ToolBoxBitmapAttribute**. Nejde o standardní třídu, nýbrž o třídu, která obsahuje popisnou charakteristiku (metadata) o vašem ovládacím prvku.

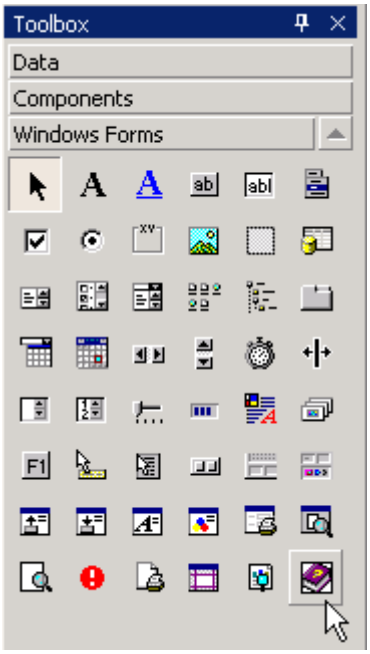
Zápis atributu **ToolboxBitmap** s cestou k souboru s bitmapou se musí nacházet před deklarací třídy, ovšem na stejném řádku. Aplikovatelná je i situace, kdy se zápis atributu bude nacházet na jednom řádku a bude od deklarace třídy oddělen mezerou a znakem pro pokračování řádku (_).

Ukázka zakomponování atributu do programového kódu ovládacího prvku:




```
<ToolboxBitmap("d:\Ikona16x16.bmp")> _
Public Class UserControl1
    Inherits System.Windows.Forms.UserControl

#Region "Windows Form Designer generated code"

    Private Sub UserControl1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load

    End Sub
End Class
```

A potenciální výsledek:



Obr. 9 – Vlastní ikona ovládacího prvku v soupravě nástrojů

Integrované prostředí rubriky Visual Basic	Autor: **Ján Hanák**

Téma měsíce

Inteligentní počítačoví asistenti

Časová náročnost (min): **60**	Začátečník	Pokročilý	Profesionál
			

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

Vážení přátelé,

v dubnovém vydání rubriky se budeme věnovat poznávání programové architektury technologie s názvem **Microsoft Agent 2.0**, která je zodpovědná za správnou funkčnost digitálních počítačových pomocníků, se kterými se můžete střetnout v stále větším množství moderních aplikací. Předmětem našeho zájmu bude vysvětlení základů uvedené technologie a posléze praktické ukázky, které vám pomohou s implementací počítačových pomocníků do vašich vlastních programů.

Obsah

Představení možností technologie

Pojem „počítačoví asistenti" byl do povědomí uživatelů a posléze i programátorů zaveden již před nějakým časem. Pokud bychom chtěli být přesnější, musíme se vrátit až do doby, kdy byl na trh uveden kancelářský balík Office společnosti Microsoft ve verzi 97. Sada programů tehdy, samozřejmě kromě jiného, přinesla na výsluní revoluční prvek, jenž byl navíc velice pečlivě zakomponován do uživatelského rozhraní nabízených aplikací. Ano, šlo o takzvaného pomocníka Office, v podobě snad již legendární kancelářské sponky. Počítačový asistent vás sprovázel od prvních chvil. Snažil se vám pomoci při uskutečnění různých úkolů, neustále vás upozorňoval na to, jak lze vaší práci zefektivnit použitím jiného postupu či aplikací klávesových zkratek. Zkrátka, byl stále „na dohled", což u některých uživatelů vyvolávalo záchvaty vzteku. V žádném případě bych na tomto místě nechtěl polemizovat o skutečnosti, zdali byla umělá inteligence digitálního asistenta na požadované úrovni nebo nikoliv. Pro nás je myslím nejdůležitější fakt, že pomocník byl postaven na dostatečně pevném základu, který dovoloval jeho použití v různých situacích (např. pomocí jazyků Visual Basic a Visual Basic pro aplikace (VBA)).

Budeme-li sledovat chronologii následujících verzí aplikační sady Office, uvidíme, že pomocníci si také v nich vydobyly neotřesitelnou pozici. Domnívám se, že v okamžiku, kdy jste spatřili animovanou postavičku pomocníka, chtěli jste něco podobného mít i ve vašem programu. Dobrou zprávou je, že na následujících řádcích se dozvíte vše potřebné pro to, aby se váš sen stal skutečností. Ještě předtím se ale zkusme podívat na technologii MS Agent 2.0 podrobněji.

Technologie MS Agent 2.0 nabízí množství objektů, kolekcí, vlastností a metod, které můžete kdykoliv použít a převzít tak chování počítačových pomocníků do svých rukou. Všechny programové součásti technologie jsou nerozlučně spjaty s operačním systémem Windows a po jejich nainstalovaní je již nelze ze systému bezpečně odstranit. Cílovou skupinou pro aplikaci technologie jsou programátoři, kteří používají vývojové nástroje, které podporují standard ActiveX (resp. COM, Component Object Model). Kolekce programových komponent totiž obsahuje ovládací prvek ActiveX s názvem **Microsoft Agent Control 2.0**, pomocí kterého je možné snadnou cestou zadávat pomocníkovi programové pokyny.

Programový kód ovládacího prvku **MS Agent Control 2.0** je uložen v souboru **agentctl.dll**. Tento soubor se nachází ve složce MSAGENT kořenové složky Windows.

Do množiny programovacích nástrojů, které lze použít při programování počítačových pomocníků, můžeme zařadit Visual Basic a Visual C++, dále pak také Visual FoxPro, editor jazyka VBA v MS Office a rovněž MS Internet Explorer. Všechna tato vývojová prostředí vám mohou posloužit v případě, že budete chtít zavolat počítačové pomocníky.

Systémové soubory technologie **MS Agent 2.0** se nacházejí ve složce MSAGENT (tato složka je umístěna v kořenové složce systému Windows). Systémové soubory technologie jsou na novějších verzích operačního systému Windows (Millennium Edition, 2000 a XP) již standardně zahrnuty do instalace systému, a proto je nemusíte opětovně instalovat. Jste-li ovšem majitelem starší verze systému Windows (95 a 98), na domové stránce technologie (**www.microsoft.com/msagent/**) najdete vše potřebné.

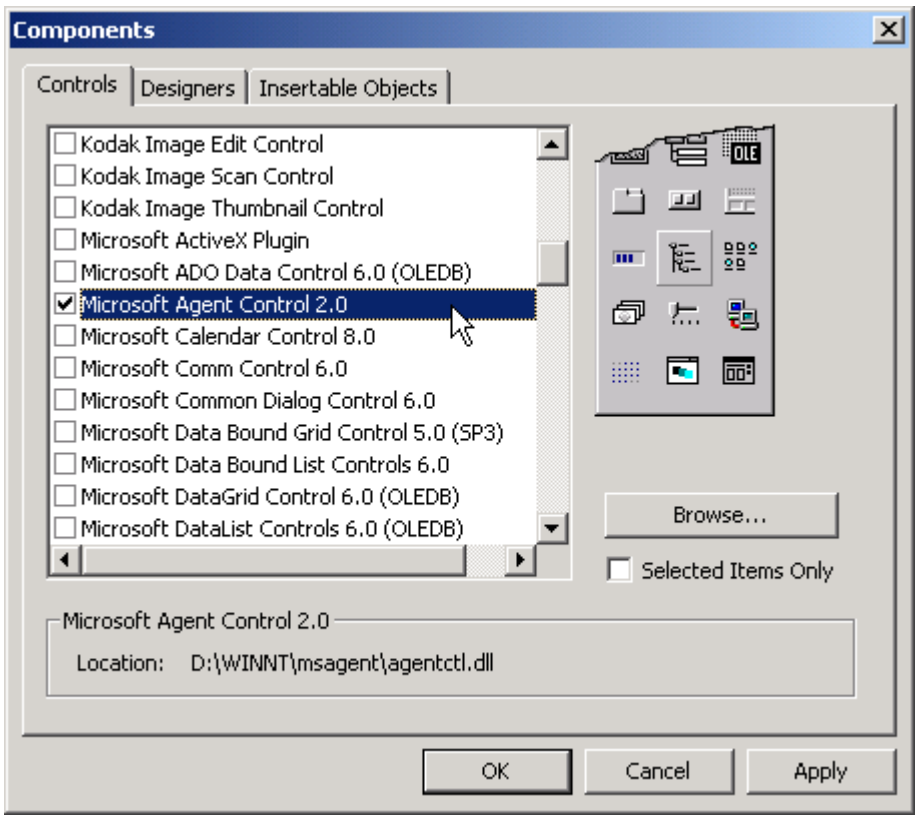
Velmi důležitou informací je, že počítačový pomocník působí jako server typu **out-of-process**, co znamená, že asistent je vždy spouštěn ve svém vlastním procesu. Tato skutečnost ostatně vysvětluje, proč je pomocník zcela nezávislý od aplikace, která si vyžádala jeho služby (jde o tzv. klientskou aplikaci). Nezávislá povaha technologie počítačového asistenta vám dovoluje aplikovat kousky doposud nevídané. Můžete například nechat zobrazit na obrazovce několik pomocníků a třeba uskutečnit mezi nimi rozhovor.

Osobně si myslím, že možnosti počítačových asistentů jsou poněkud přehlíženy. Vždyť jenom samotná představa, že ve světe vašeho počítače se nachází inteligentní virtuální bytost, je skutečně jedinečná. A pokud je také „vybroušen" programový kód, jenž nařizuje pomocníkovi, co má kdy dělat, je počítačový asistent pomocníkem v pravém slova smyslu. Dobře, věřím, že můžeme přejít k programovým ukázkám toho, co všechno může pomocník dělat.

Přidání ovládacího prvku MS Agent Control 2.0 do soupravy nástrojů

Při našich pokusech budeme vždy pracovat s ovládacím prvkem **MS Agent Control 2.0**. Aby bylo možné vytvářet instance tohoto prvku a následně je ukládat přímo na plochu formuláře, je nejdříve zapotřebí přidat do soupravy nástrojů odkaz na příslušný ovládací prvek. Postupujte teda podle následujících instrukcí:

1. Spusťte Visual Basic 6.0 a vytvořte nový projekt typu **Standard EXE**.
2. Označte nabídku **Project** a klepněte na položku **Components**.
3. Dávejte pozor, aby byla v dialogovém okně vybrána záložka **Controls**. Zde v seznamu vyhledejte položku s názvem **Microsoft Agent Control 2.0** a klepnutím ji označte (obr. 1).



Obr. 1 – Vložení odkazu na ovládací prvek do soupravy nástrojů

Pokud by se požadovaná položka nenacházela z nějakého blíže nespecifikovaného důvodu v seznamu, zkuste klepnout na tlačítko **Browse** a vyhledat soubor **agentctl.dll**.

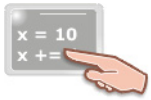
4. Aktivujte tlačítko OK a zkontrolujte, jestli se ikona ovládacího prvku nachází v soupravě nástrojů (obr. 2).

Obr. 2 – Ikona ovládacího prvku **MS Agent Control 2.0** v soupravě nástrojů

Výborně, první krok jste úspěšně absolvovali. Dále již stačí jenom umístit instanci ovládacího prvku na formulář a začít s psaním programového kódu. To vše bude námětem další sekce této studie.

Zobrazení počítačového pomocníka

Umístěte na formulář instanci ovládacího prvku **MS Agent Control 2.0** a také instanci prvku **CommandButton**. Označte instanci prvku s pomocníkem a podívejte se do okna vlastností (**Properties Window**). Můžete si všimnout, že asistent podporuje jisté vlastnosti, no není jich moc. Pokud budete chtít, můžete změnit jméno pomocníka modifikací hodnoty vlastnosti **Name** (standardně nese první instance jméno **Agent1**). Pokračujte dvojitým klepnutím na instanci tlačítka a zapsáním uvedeného programového kódu do okna editoru.

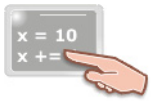



```
Agent1.Characters.Load "Pomocník1", "C:\Soubor_s_asistentem.acs"
Agent1.Characters("Pomocník1").Show
```

Protože může tento fragment programového kódu působit na první pohled lehce komplikovaně, pokusme se na něj podívat zblízka. Především je nutné nahrát správný profil asistenta ze souboru .ACS a tento pak přidat do kolekce objektů s názvem **Characters**. To provádí metoda **Load**, signaturu které tvoří dva parametry:

1. **CharacterID** – jde o textový řetězec, který identifikuje profil pomocníka. Pokud se budete chtít v budoucnosti odkázat na tento profil pomocníka, vždy použijete název (textový řetězec), který jste danému profilu přiřadili při volání metody **Load**. V našem případě je vybraný profil asistenta pojmenován jako **Pomocník1**.
2. **Cesta k souboru s profilem asistenta**. Určení cesty k požadovanému souboru může být uskutečněno dvojakým způsobem. V nejběžnějším případě bude tento parametr naplněn explicitním určením cesty k souboru s asistentem, jenž je fyzicky přítomný na vašem pevném disku (např. C:\Soubor1.acs). Pokud ovšem chcete použít soubor s pomocníkem, jenž je uložen na síti, můžete zadat také příslušnou HTTP adresu.

V souvislosti s druhým parametrem je nutné vzpomenout i relativní určení cesty k souboru s profilem pomocníka. Můžete použít kód v této podobě:




```
Agent1.Characters.Load "Pomocník1", "Genie.acs"
Agent1.Characters("Pomocník1").Show
```

Zde není určena plná cesta k souboru, ale cesta relativní. Ta předpokládá, že soubor s asistentem (Genie.ACS) se nachází ve složce CHARS, která je podřízenou složkou složky MSAGENT.

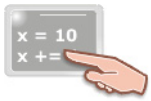
Zobrazení pomocníka na obrazovce počítače má na starosti metoda **Show**. Pokud byste použili soubor s čarovným džinem, uvedené dva řádky kódu by ho zviditelnily na obrazovce v podobě uvedené na obr. 3.

Obr. 3 – Zobrazení pomocníka pomocí metody **Show**

Při nahrávání profilu pomocníka platí pravidlo, že použitím jedné instance ovládacího prvku můžete nahrát jenom jeden profil asistenta ve stejnou dobu. Pokud byste tedy chtěli aktivovat dalšího pomocníka, bylo by potřebné vytvořit také další instanci ovládacího prvku.

Vytvoření bublinového okna pomocníka a zobrazení uvítací zprávy

Aby mohl asistent zobrazit bublinové okno s uvítací zprávou, použijeme metodu **Speak** s textovým parametrem:

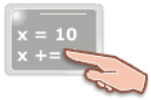


```
Agent1.Characters("Pomocník1").Speak "Dobrý den, já jsem váš počítačový
pomocník."
```

Výsledek lze pozorovat na obr. 4.

Obr. 4 – Bublinové okno pomocníka

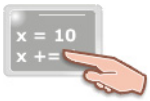
Chcete-li, můžete asistentovi poskytnout několik uvítacích zpráv a nechat na něm, kterou zobrazí. V tomto případě je nutné oddělit jednotlivé věty symbolem svislé čáry (|). Prostudujte si níže uvedený programový kód:



```
Agent1.Characters("Pomocník1").Speak _
"Dobrý den, já jsem váš počítačový pomocník.|" & _
"Ahoj, jak se máte?|" & _
"Potřebujete pomoc?"
```

Pokud budete zkoušet činnost tohoto kódu, pamatujte na pravidlo, že v jednom okamžiku může být z jedné instance ovládacího prvku zobrazen pouze jeden asistent. Proto kdybyste dvakrát klepli na tlačítko na formuláři, Visual Basic by ohlásil chybu, v popisu které by vám bylo sděleno, že postavička pomocníka již byla jednou načtena. Tento (mimochodem poměrné častý) problém lze vyřešit například takto:

1. V deklarační části formuláře vytvořte proměnnou typu **Boolean** s názvem **aktivace**.
2. V událostní proceduře **Form1_Load** inicializujte tuto proměnnou na hodnotu **False**.
3. Kód v obsluze události **Click** tlačítka upravte do následující podoby:



```
Private Sub Command1_Click()
If aktivace Then
```

Programový kód pokračuje na následující straně

```
symbol:
    Agent1.Characters("Pomocník1").Show
    Agent1.Characters("Pomocník1").Speak _
    "Dobrý den, já jsem váš počítačový pomocník.|" & _
    "Ahoj, jak se máte?|" & _
    "Potřebujete pomoc?"
Else
    Agent1.Characters.Load "Pomocník1", "Genie.acs"
    GoTo symbol
End If

aktivace = True
End Sub
```

Znázorněný kód je poněkud netypický hlavně použitím konstrukce **GoTo**. Průběh celého algoritmu vypadá asi takto:

1. Je testována hodnota proměnné **aktivace**. Při prvním spuštění programu je hodnota této proměnné nastavena na **False** (toto nastavení hodnoty proměnné se děje v událostní proceduře **Form1_Load**). Proto je dále proveden kód, jenž se nachází ve **druhé části** rozhodovací konstrukce **If-Then**, a tedy dochází k načtení profilu pomocníka prostřednictvím metody **Load**. Připomínám, že jde jenom o „načtení" pomocníka, nedojde tedy k jeho zobrazení na monitoru počítače.
2. Provede se příkaz **Goto** a exekuce programu se přenese na řádek, jenž následuje za řádkem, který je označen návěstím (v našem případě je návěstím slovo **symbol**).
3. Provedou se další příkazy, které následují za návěstím a dojde k opuštění konstrukce **If-Then**.
4. Proměnná **aktivace** je posléze nastavena na hodnotu **True**. Pokud klepnete ještě jednou na tlačítko, proměnná již bude nastavena na hodnotu **True** a provede se tedy jenom **první část** rozhodovací konstrukce **If-Then**.

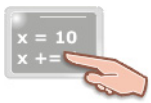
Jestliže teď spustíte aplikaci, můžete klepnout na tlačítko i poté, co je pomocník již viditelný na obrazovce počítače. Nedojde k žádné chybě, asistent jednoduše vybere další zprávu a přečte ji.

Změna pozice pomocníka na obrazovce počítače

Při standardním způsobu aktivace se pomocník zobrazuje v levém horním rohu plochy vašeho monitoru. Jestliže vám toto umístění asistenta nevyhovuje, můžete ho změnit pomocí dvou vlastností **Top** a **Left**. Tyto vlastnosti mají stejný význam jako jejich ekvivalenty ve Visual Basicu, společně určují pozici levého horního rohu okna pomocníka.

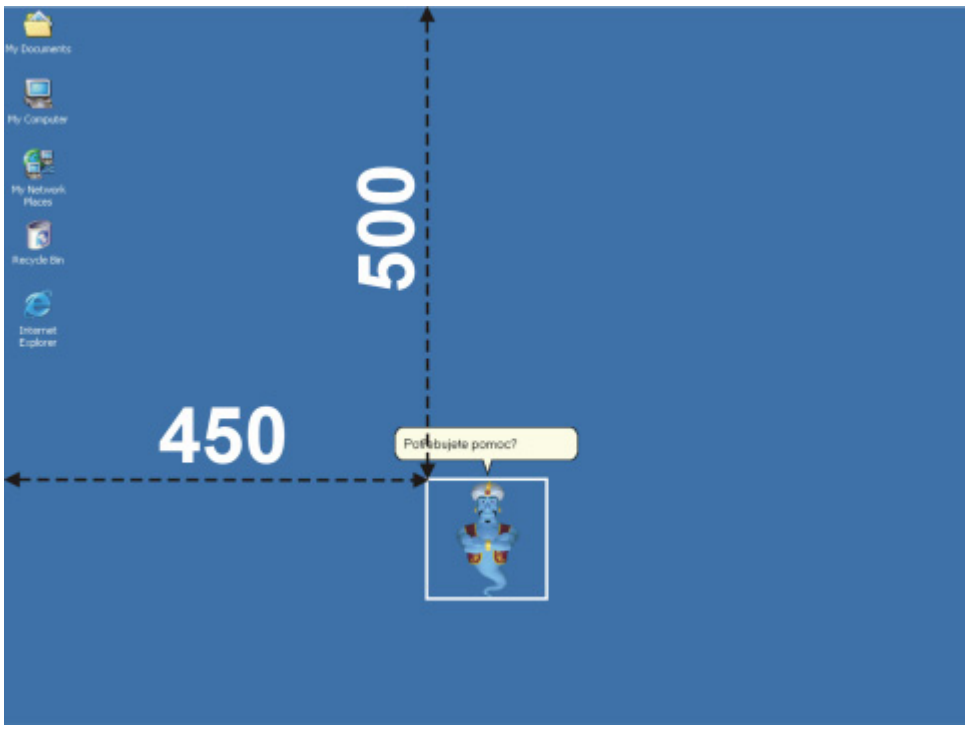
Ačkoliv to není na pohled zřejmé, pomocník se nachází v okně (rozměry tohoto okna by neměly překročit hranici 128 krát 128 pixelů). Pozadí okna a okraje okna jsou průhledné, proto je nemůžete spatřit.

Budete-li chtít přesunout asistenta 500 pixelů směrem dolů a 450 pixelů směrem doprava, použijte tento kód:




```
Agent1.Characters("Pomocník1").Top = 500
Agent1.Characters("Pomocník1").Left = 450
```

Výsledek je demonstrován na obr. 5.



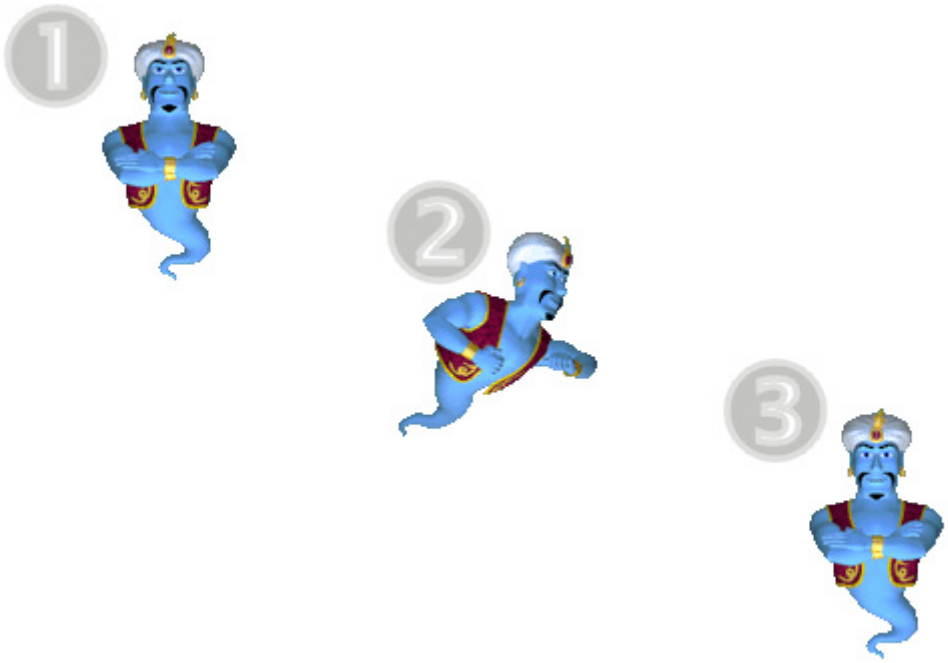
Obr. 5 – Přesné umístění pomocníka na kýženou pozici

Přestože vlastnosti **Top** a **Left** odvádějí svou práci znamenitě, existuje další způsob, pomocí kterého lze změnit pozici asistenta. Jde o metodu **MoveTo** objektu **Character**. Představme si nejprve zápis metody a pak si k ní povíme pár slov.

agent.**Characters ("***CharacterID***").MoveTo** *x,y,* [*Speed*]

Metoda **MoveTo** pracuje se třemi parametry, přičemž poslední z nich je volitelný. První dva parametry určují souřadnice cílového levého horního bodu okna pomocníka. Již vzpomínaný volitelný parametr determinuje rychlost přehrávaní snímků animace pomocníka při změně jeho pozice. Tato rychlost je měřena v milisekundách.

Velkou výhodou metody **MoveTo** je skutečnost, že změna pozice asistenta je vizuálně umocněna animací (obr. 6).



Obr. 6 – Změna pozice pomocníka použitím metody **MoveTo**

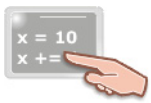
Ukončení práce s pomocníkem

Jestliže již nebudete chtít využít služeb počítačového asistenta, máte v zásadě dvě možnosti, jak jej odstranit z obrazovky počítače. Pomocníka můžete buď skrýt aplikací metody **Hide**, nebo jej zcela odstranit z paměti použitím metody **Unload**.

Dávejte si pozor na použití metody **Hide**. Tato metoda zabezpečuje pouze skrytí pomocníka, nikoliv jeho odstranění z paměti počítače. Paměťové zdroje, které obsazoval asistent můžete bezpečně uvolnit jenom použitím metody **Unload**.

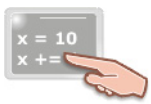
Příkladné použití obou metod můžete vidět níže.

Když budete chtít skrýt pomocníka, zapište tento kód:




```
Agent1.Characters("Pomocník1").Hide
```

V případě nutnosti odstraňte pomocníka z paměti tímto kódem:




```
Agent1.Characters.Unload "Pomocník1"
```

Integrované prostředí rubriky Visual Basic	Autor: **Ján Hanák**

Začínáme s VB .NET

Úvod do světa .NET (4. díl)

Časová náročnost (min):	Začátečník	Pokročilý	Profesionál
45			

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

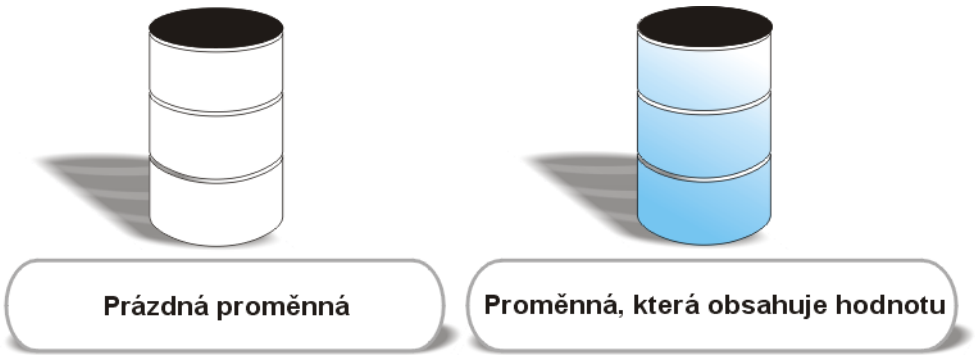
Vážení čtenáři,

v dnešním díle seriálu začněme s vysvětlováním pojmu proměnná. Dozvíte se, na co vlastně proměnné slouží, jak se deklarují a inicializují. Také bude ozřejměn pojem obor proměnné. Přeji příjemné počtení.

Obsah

Pojem proměnná

Tak, jak je základní stavební jednotkou aplikace stylu .NET forma zvaná assembly, tak ve světě programového kódu je základní jednotkou pro ukládání hodnot proměnná. Do proměnné lze umísťovat různé hodnoty, jako například celá či desetinná čísla, text, datum atd. Proměnné jsou někdy také označovány jako kontejnery, aby se ještě více zvýraznila jejich schopnost uchovávat jistou hodnotu (obr. 1).



Obr. 1 – Vizuální přiblížení pojmu proměnná

Pokud byste se chtěli podívat na proměnné z techničtějšího hlediska, zjistili byste, že každá proměnná alokuje jisté místo v paměti počítače (proměnná má tedy svůj rozsah, který je zpravidla měřen v bitech). Právě toto místo v paměti je schopno uchovat hodnotu, kterou má proměnná „na

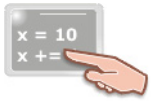
starosti". Aby ovšem bylo možné do proměnné umístit jakoukoliv hodnotu, je potřebné ji nejdříve deklarovat.

Deklarace proměnné

Na deklarování proměnných slouží ve Visual Basicu .NET příkaz **Dim**. Obecný zápis procesu deklarace proměnné je:

Dim jméno_proměnné As datový_typ_proměnné

Pokud byste tedy chtěli deklarovat proměnnou **číslo**, která by byla datového typu **Integer**, použili byste tento zápis programového kódu:




```
Dim číslo As Integer
```

Jste-li začínajícími programátory, může se vám mnoho věcí na způsobu deklarace proměnné zdát nejasných, a proto si dovolím tento proces rozebrat poněkud podrobněji. Snad nejdůležitější informací je, že neexistuje nic jako „ryzí proměnná", vždy se můžete střetnout jenom s proměnnou, které reprezentuje jistý datový typ. Datový typ si můžete představit jako charakteristiku, která blíže určuje použití proměnné. Ve Visual Basicu .NET se koneckonců můžete střetnout s poměrně velkým počtem rozličných datových typů (jejich charakteristikou se budeme zabývat v příštím dílu seriálu). Příkaz **Dim** zabezpečí rezervaci místa v paměti pro vytvořenou proměnnou (přitom samozřejmé dbá na to, aby byla velikost rezervovaného místa v souladu s rozsahem použitého datového typu proměnné). V zápisu je také důležité klíčové slovo **As**, za kterým následuje specifikace zvoleného datového typu. Celý zápis ukončuje název datového typu, v našem případě je to typ **Integer**, který je vhodný pro ukládání celých čísel.

Použijete-li pouze uvedený zápis s příkazem **Dim**, je pro proměnnou **číslo** jenom vyhrazeno místo v paměti počítače, ale samotná proměnná neobsahuje žádnou hodnotu. Aby bylo možné umístit do proměnné hodnotu, je zapotřebí proměnnou **inicializovat**. Jak na to si ukážeme v další kapitole.

Inicializace proměnné

Inicializace proměnné se uskutečňuje pomocí takzvaného přiřazovacího příkazu.

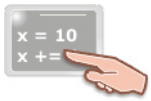
V této souvislosti bychom si také měli ozřejmit pojem **příkaz**. Příkazem se obyčejně rozumí jeden řádek programového kódu. Příkazem může být ovšem také klíčové slovo programovacího jazyka, po použití kterého se uskuteční nějaká operace. Nakonec příkazem může být chápána i programová konstrukce typu přiřaďování, kdy dochází k umístění hodnoty do proměnné.

Přiřazovací příkaz obecně přiřazuje jistou hodnotu do proměnné. Řečeno jinými slovy, jenom pomocí přiřazovacího příkazu můžete do proměnné uložit nějakou hodnotu. Jak ovšem takový příkaz vypadá? Asi takto:

jméno_proměnné = hodnota, kterou chceme do proměnné přiřadit

Přiřazovací příkaz dělá přesně to, co říká jeho název, tedy přiřazuje „něco někam". Jaký je cíl přiřazení? No přece proměnná. A co do proměnné přiřazujeme? Libovolnou hodnotu v závislosti od rozsahu příslušného datového typu proměnné. Ještě prostěji, obsah pravé strany příkazu je „vložen" do proměnné, název které se nachází na levé straně přiřazovacího příkazu.

Budeme-li chtít navázat na předchozí deklaraci proměnné **číslo**, můžeme použít tento zápis programového kódu:

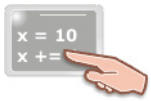


```
číslo = 1000
```

Tímto způsobem naznačíme Visual Basicu .NET, aby do proměnné **číslo** umístil hodnotu 1000. Porovnání mezi deklarací a inicializací proměnné znázorňuje obr. 2.

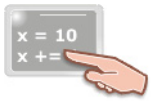
Obr. 2 – Rozdíl mezi deklarací a následnou inicializací proměnné

Pokud ji v budoucnu nezměníte, proměnná **číslo** bude stále obsahovat tuto hodnotu. Abyste se přesvědčili, že proměnná opravdu obsahuje zadanou číselní hodnotu, můžete použít metodu **Show** třídy **MessageBox** v následujícím tvaru:



```
MessageBox.Show(číslo)
```

Metodě **Show** je v tomto případě jako vstupní hodnota (přesněji jako argument) předána proměnná **číslo**, a protože metoda zobrazuje v dialogovém okně hodnotu svého parametru, udělá tak i v našem případě a zobrazí okno s hodnotou 1000. Dobrá, přesvědčili jste se, že proměnná opravdu obsahuje vloženou hodnotu. Pokud byste chtěli změnit hodnotu proměnné třeba na rovný milion, opět použijete přiřazovací příkaz, jenom s novým určením hodnoty:



```
číslo = 1000000
```

Takto je předchozí obsah proměnné **číslo** smazán a nahrazen novou hodnotou. Demonstrovaným postupem můžete měnit hodnotu proměnné, no nesmíte v žádném případě zapomenout na rozsah proměnné. Ten totiž určuje interval hodnot, které můžete do proměnné uložit (v případě datového typu **Integer** je tento interval <-2 147 483 648, 2 147 483 647>).

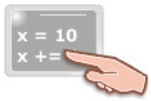
Datový typ **Integer** je uchováván jako 32bitová, resp. 4bajtová hodnota. Jde o datový typ, jenž poskytuje nejefektivnější výkon při celočíselných operacích na dvaatřicetibitových procesorech.

Jestliže byste do proměnné **číslo** přiřadili hodnotu mimo povolený interval, Visual Basic .NET by vygeneroval chybové hlášení.

V souvislosti s uchováváním hodnot pomocí proměnné by vás ovšem mohla napadnout otázka, **dokdy** ve skutečnosti proměnná bude přiřazenou numerickou hodnotu obsahovat. Aby bylo možné vyřešit toto dilema, musíme si vysvětlit další důležitý pojem, kterým je **obor** proměnné.

Obor proměnné

Oborem proměnné rozumíme časový interval, v rámci kterého proměnná udržuje svou hodnotu. Umístíte-li na formulář projektu tlačítko a poklepete na něj, otevře se editor pro zápis kódu. V prostředí editoru můžete zapsat kód třeba pro událostní proceduru **Click** tlačítka. Kdybychom v této událostní proceduře deklarovali proměnnou **číslo** a přiřadili do této proměnné numerickou hodnotu, proměnná by obsahovala tuto hodnotu jenom dokud by byl vykonáván kód dané procedury. Když se provádění událostní procedury ukončí, zlikvidují se všechny proměnné, které byly v této proceduře deklarovány (a pochopitelně dojde i k destrukci hodnot, které tyto proměnné obsahovaly). Předveďme si tuto situaci prakticky na ukázce programového kódu obsluhy události **Click** tlačítka **Button1**:




```
Private Sub Button1_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles Button1.Click
    Dim číslo As Integer
    číslo = 1000
    MessageBox.Show(číslo)
End Sub
```

Uvnitř procedury typu Sub je deklarována proměnná **číslo** typu **Integer**. Proměnná je také inicializována na hodnotu 1000. Další příkaz zobrazuje dialogové okno s hodnotou. Tím je úkol událostní procedury splněn a procedura je ukončena. V té chvíli dochází k likvidaci proměnné **číslo** a také „mizí" hodnota, která byla do proměnné **číslo** uložena. Proto říkáme, že oborem proměnné **číslo** je událostní procedura **Click** tlačítka **Button1**.

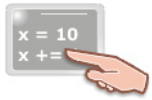
Proměnná může mít i daleko širší obor. Oborem proměnné může být celý formulář (resp. celá třída formuláře) nebo také celý projekt VB .NET (jestliže je proměnná deklarována v doprovodném modulu).

Modulem se rozumí speciální soubor, do kterého se ukládají tzv. veřejné proměnné, funkce a další platné programové konstrukce.

Zkusme si první z těchto variant ukázat na praktickém příkladu.

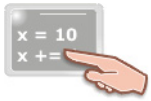
Příklad: Formulář (třída formuláře) jako obor proměnné

1. Vytvořte ve Visual Basicu .NET nový projekt typu **Windows Application**.
2. Na formulář umístěte dvě tlačítka, která nechte standardně pojmenovaná.
3. Poklepejte na první tlačítko a upravte podobu událostní procedury **Click** podle vzoru:



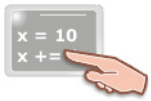
```
Private Sub Button1_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles Button1.Click
    číslo = 1000
    MessageBox.Show(číslo)
End Sub
```

4. Klepněte na druhé tlačítko a zadejte programový kód pro zobrazení dialogového okna se zprávou:



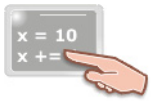
```
Private Sub Button2_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles Button2.Click
    MessageBox.Show(číslo)
End Sub
```

5. V této chvíli je nutné učinit nejvýznamnější krok, a sice deklarovat proměnnou, oborem které bude třída formuláře. Přemístěte kurzor pod první dva řádky veškerého programového kódu formuláře, tedy pod tyto dva řádky:



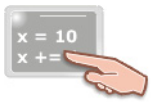
```
Public Class Form1
       Inherits System.Windows.Forms.Form
```

A deklarujte proměnnou **číslo** zápisem:



```
Dim číslo As Integer
```

6. Pohled na programový kód formuláře by měl mít tuto podobu:



```
Public Class Form1
    Inherits System.Windows.Forms.Form
    Dim číslo As Integer

"Windows Form Designer generated code"

    Private Sub Button1_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Button1.Click
        číslo = 1000
        MessageBox.Show(číslo)
    End Sub

    Private Sub Button2_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Button2.Click
        MessageBox.Show(číslo)
    End Sub
End Class
```

Když pustíte aplikaci a klepněte na první tlačítko, objeví se dialogové okno, které bude ukazovat numerickou hodnotu (1000) proměnné **číslo**. Po ukončení událostní procedury **Button1_Click** ovšem nedojde k likvidaci proměnné **číslo**, protože oborem této proměnné je v tomto případě celý formulář a ne jenom procedura. Klepnete-li na druhé tlačítko, zobrazí se aktuální stav proměnné **číslo**, tedy opět numerická hodnota 1000. Ptáte se, kdy je v tomto případě proměnná zlikvidována? K destrukci proměnné dochází ve chvíli likvidace formuláře (resp. odstranění formuláře z paměti počítače).

Integrované prostředí rubriky Visual Basic	Autor: Ján Hanák

Programátorská laboratoř

Prostor pro experimentování

Časová náročnost (min):	Začátečník	Pokročilý	Profesionál
45			

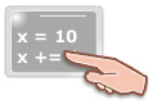
Použitý operační systém :	Windows 2000 SP3
Hlavní vývojový nástroj :	Visual Basic .NET 2002
Další vývojový software :	Visual Basic 6.0
Jiný software :	Žádný

Přehrávání animací počítačového pomocníka

Přehrávání animací počítačových pomocníků, kterých základy spočívají na technologii **MS Agent 2.0**, má na starosti metoda **Play**. Obecné použití metody má tuto podobu:

agent.**Characters ("***CharacterID***").Play** "*AnimationName*"

Jak můžete vidět, metoda disponuje jediným parametrem (**AnimationName**), jenž specifikuje název animace, která se má přehrát. Kdybychom například chtěli, aby nám pomocník zatleskal, použijeme tento programový kód:




```
Agent1.Characters.Load "Pomocník1", "genie.acs"
Dim pomocník As Object
Set pomocník = Agent1.Characters("Pomocník1")
pomocník.Show
pomocník.Speak "Gratuluji vám!"
pomocník.Play "Congratulate"
```

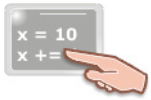
Výsledek můžete vidět na obr. 1.

Obr. 1 – Přehrání vybrané animace pomocníka použitím metody **Play**

Sady animací, které podporují standardní počítačoví pomocníci (jako je například čarovný džin či kouzelník Merlin) můžete získat z dokumentu Agent: Platform SDK, jenž si můžete stáhnout z webové stránky **www.microsoft.com/msagent/**.

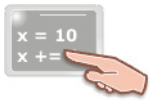
Současná deklarace a inicializace proměnné

Visual Basic .NET uvádí nový a nutno říci, že mnohem rychlejší a efektivnější způsob procesu deklarace a následní inicializace proměnných. Oba tyto parciální úkony můžete od nynějška spojit dohromady, protože VB .NET vám dovoluje proměnné ihned při deklaraci inicializovat. Nejdříve se ovšem podívejme, jaká byla situace při deklaraci a inicializaci proměnných před příchodem VB .NET:




```
Dim proměnná1 As Integer
'Deklarace proměnné s názvem proměnná1.
proměnná1 = 1024
'Inicializace proměnné.
```

Teď se podívejme, jak lze oba kroky spojit (ovšem pamatujte, že pouze ve Visual Basicu .NET!):



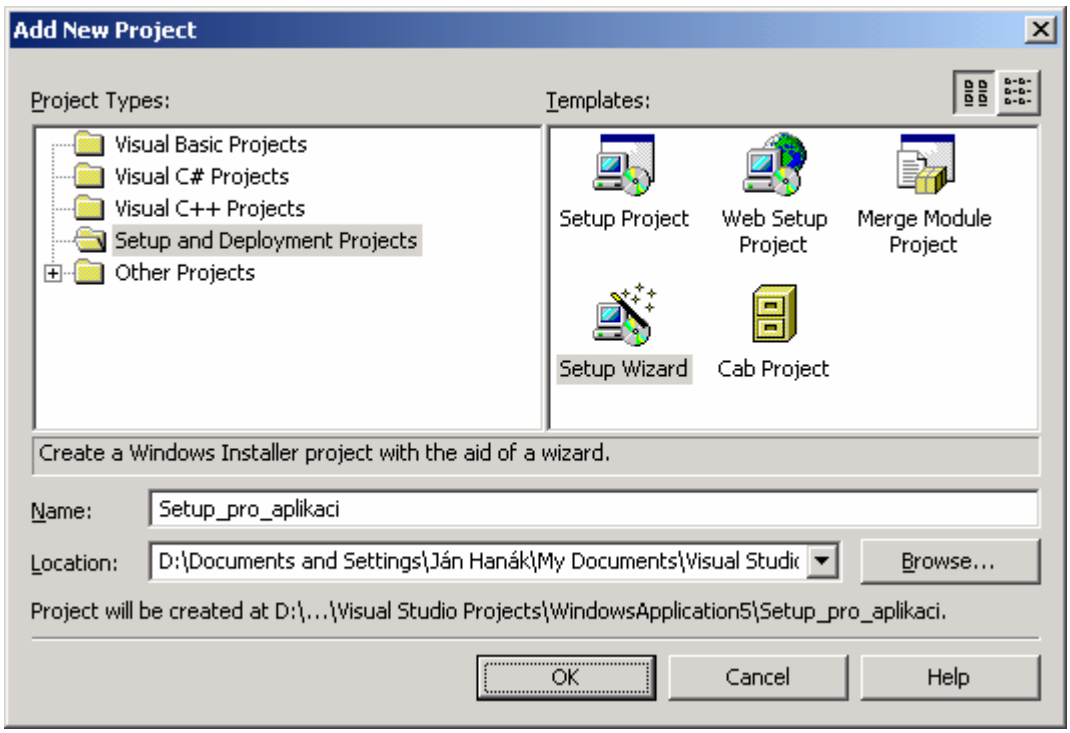

```
Dim proměnná1 As Integer = 1024
```

Pokud se rozhodnete pro současnou deklaraci a inicializaci proměnných, nejenom, že si ušetříte trochu psaní, ale můžete snadno optimalizovat svůj programový kód.

Vytvoření distribuční jednotky aplikace

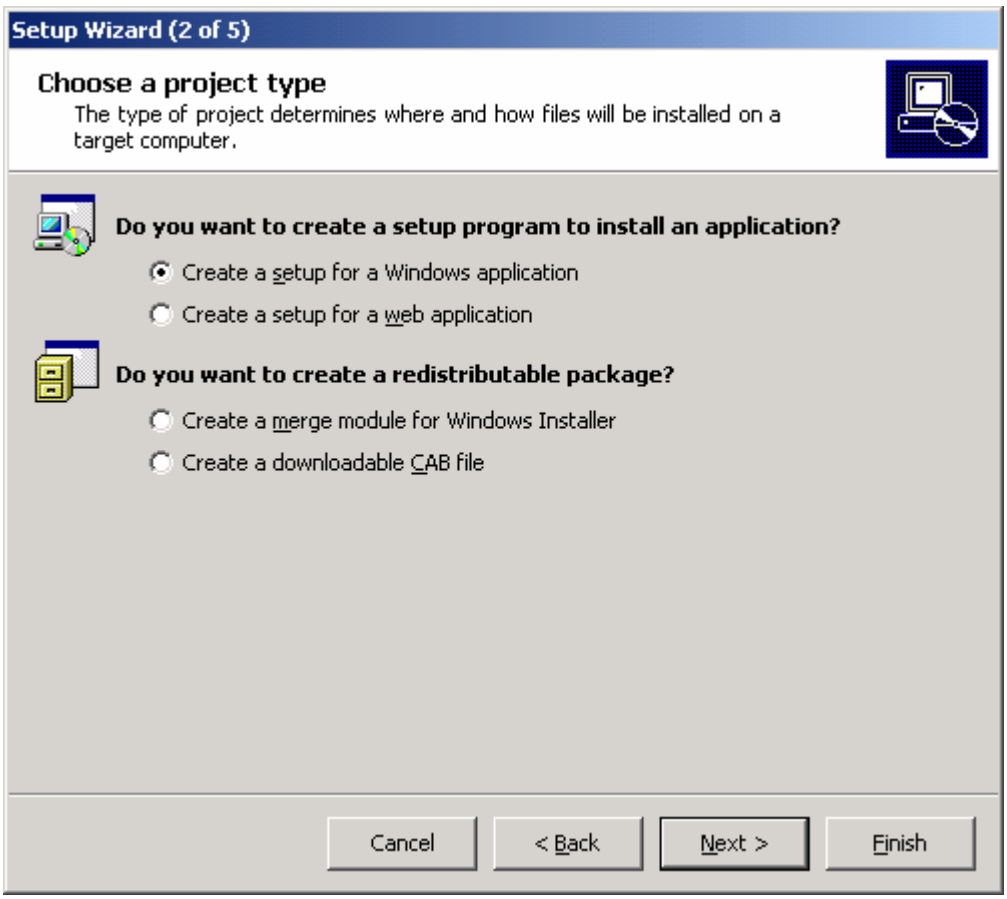
V tomto tipu si ukážeme, jak k stávajícímu projektu Visual Basicu .NET přidat distribuční jednotku. Postupujte dle následujících instrukcí:

1. Otevřete ve Visual Basicu .NET váš projekt.
2. Vyberte nabídku **New**, ukažte na položku **Add Project** a vyberte příkaz **New Project**.
3. Objeví se dialogové okno **Add New Project**. V části **Project Types** vyberte položku **Setup and Deployment Projects** a v oddílu **Templates** označte ikonu s názvem **Setup Wizard**. Pokud chcete, můžete zadat i název pro distribuční jednotku. Po všech úpravách by mělo dialogové okno vypadat jako to z obr. 2.



Obr. 2 – Přidání nového projektu do stávajícího řešení

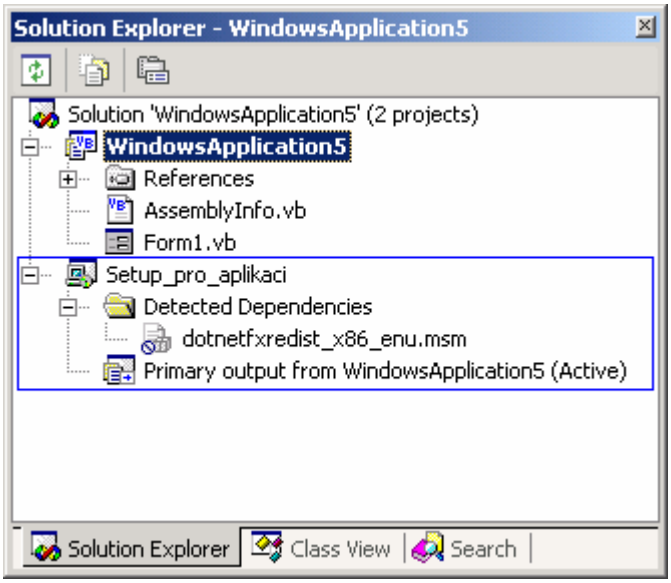
4. Po klepnutí na tlačítko OK se na obrazovce objeví průvodce (**Setup Wizard**). V prvním kroku není co provádět a proto můžete aktivovat tlačítko **Next**.
5. Ve druhém kroku vám průvodce nabídne, abyste si vybrali typ distribuční jednotky, kterou si přejete vytvořit. My zde vybereme první položku **Create a setup for a Windows application** (předpokládám, že vyvíjíte aplikaci pro Windows). Situaci demonstruje obr 3.



Obr. 3 – Bližší specifikace typu vytvářeného projektu

6. Ve třetím kroku je zapotřebí určit, které součásti chceme zahrnout do distribuční jednotky. Pro potřeby tohoto ilustračního příkladu bude stačit, když zatrhnete první položku (**Primary output from jméno_vaší_aplikace**). Po výběru klepněte na tlačítko **Next**.
7. Ve čtvrtém kroku se vás průvodce zeptá, zdali chcete do distribuční jednotky zahrnout i jiné typy souborů, například textové soubory typu ReadMe nebo soubory s nápovědou. Chcete-li přidat dodatečné soubory, klepněte na tlačítko **Add** a vyhledejte je. Pokračujte aktivací tlačítka **Next**.
8. V posledním kroku průvodce shrne všechny informace, které jste mu poskytli v předchozích krocích. Nakonec klepněte na tlačítko **Finish**.

Posléze průvodce přidá do VB .NET nový projekt s distribuční jednotkou. Tuto skutečnost můžete spatřit, když se podíváte do okna **Solution Explorer** (obr. 4).



Obr. 4 – Okno **Průzkumníka řešení** po přidání projektu s distribuční jednotkou

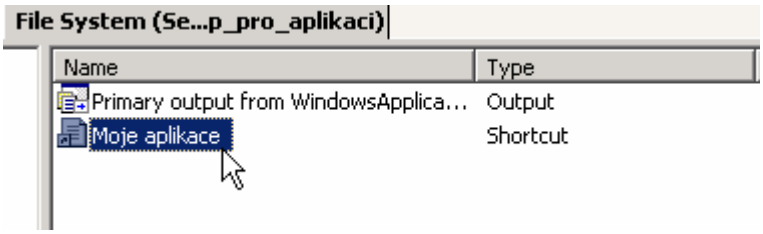
Ještě předtím, než se dostaneme k popisu okna **FileSystem**, se podíváme do okna s vlastnostmi projektu s distribuční jednotkou (jde o okno **Properties Window**). Zde můžete změnit hodnoty několika zajímavých vlastností, jako je třeba jméno autora (**Author**), popis distribuční jednotky (**Description**), či informace o výrobci (**Manufacturer**).

Jestliže v okně **Properties Window** tyto vlastnosti nevidíte, ujistěte se, že je v dialogu **Solution Explorer** vybrána položka s názvem vašeho projektu s distribuční jednotkou.

Průvodce pro vás také otevře okno **FileSystem**, které je rozděleno na dvě části. V levé části můžete vidět strukturu souborového systému tak, jak bude tato nainstalována na cílové počítačové stanici. V opozitní části se zobrazuje seznam položek, které tvoří obsah jednotlivých uzlů souborové struktury.

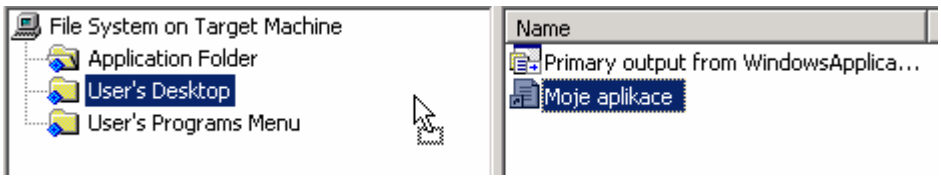
Abyste vytvořili zástupce pro aplikaci, jenž bude po instalaci aplikace na cílovém počítači umístěn na ploše, proveďte toto:

1. Ujistěte se, že v levé části okna **FileSystem** je vybrána položka **Application Folder**.
2. V pravé části okna **FileSystem** klepněte pravým tlačítkem myši na položku **Primary output from jméno_vaší_aplikace**.
3. Z kontextové nabídky vyberte první položku **Create Shortcut to Primary output from jméno_vaší_aplikace**.
4. Vytvoří se zástupce, kterého název můžete ihned přepsat třeba na **Moje aplikace** (obr. 5).



Obr. 5 – Vytvoření zástupce pro aplikaci

5. Ikonu zástupce přetáhněte na ikonu složky **User´s Desktop**, která se nachází v levé části okna **FileSystem** (obr. 6).



Obr. 6 – Přemístění zástupce do složky **User´s Desktop**

6. Zvolte nabídku **Build** a vyberte příkaz **Build Název_distribuční_jednotky**.

Podívejte se do složky vašeho projektu, kde by se měla nacházet i složka se soubory pro distribuční jednotku. Spustíte-li soubor **Setup**, bude aktivována instalační procedura. Po instalaci bude na plochu uložen soubor se zástupcem, po spuštění kterého se rozběhne samotná aplikace. Dobrá práce!

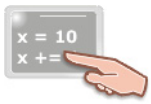
Sestavování distribučních jednotek je zcela jistě zajímavá práce, a proto se s ní střetnete také v sekci Téma měsíce v některém z příštích vydání vaší oblíbené programátorské rubriky.

Pokročilé použití operátorů

Visual Basic .NET vám dovoluje efektivnější použití hlavních operátorů ve spojení s přiřazovacím příkazem. Mezi hlavní operátory patří:

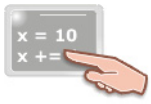
- operátor pro sčítání (+),
- operátor pro odčítání (-),
- operátor pro násobení (*),
- operátor pro dělení (/),
- operátor pro celočíselné dělení (\),
- operátor pro umocňování (^) a
- operátor pro zřetězení (&).

Pomocí pokročilého použití uvedených operátorů můžete příkaz:




```
x = x + 1
```

Zapsat jako:




```
x += 1
```

Podobná je situace i při použití dalších operátorů (tab. 1).

.

Standardní forma příkazu	Pokročilá forma příkazu
`x = x - 1`	`x -= 1`
`x = x * 1`	`x *= 1`
`x = x / 1`	`x /= 1`
`x = x \ 1`	`x \= 1`
`x = x ^ 1`	`x ^= 1`
`x = x & "text"`	`x &= "text"`

Tab. 1 – Použití zkrácených operátorů

Opět se vám tedy naskýtá příležitost pro ušetření námahy a zabezpečení efektivnějšího běhu vašich aplikací.

Integrované prostředí rubriky Visual Basic	Autor: **Ján Hanák**

Téma měsíce

Optimalizace (1. díl)

Časová náročnost (min): **60**	Začátečník	Pokročilý	Profesionál

Použitý operační systém :	Windows 2000 SP3
Hlavní vývojový nástroj :	Visual Basic 6.0 SP 5
Další vývojový software :	Žádný
Jiný software :	Žádný

Vítejte.

Z vlastní zkušenosti vím, že jednou se každý programátor dostane do situace, kdy nebude zcela spokojen s výkonem a rychlostí běhu vyvíjené aplikace. Pokud bych měl být ještě více realističtější, řekl bych, že právě myšlenky, jak zlepšit a zrychlit tu, nebo jinou část aplikace, se stávají zakrátko noční můrou programátora. Abyste ovšem nebyli ponecháni na milost a nemilost všem odvráceným aspektům programování, jsou pro vás připraveny materiály s mnoha užitečnými ukázkami, radami a tipy, takže i vaše aplikace budou tak rychlé, jak to jenom půjde. Takže, příjemné čtení a optimalizaci zdar!

Obsah

Místo optimalizace v procesu vývoje softwaru

Ještě předtím, než si vyhrneme rukávy a pustíme se do skutečné optimalizace kódu, si povězme pár slov k optimalizaci jako takové. Předem byste měli vědět, že optimalizační proces by měl být velice pečlivě a citlivě přepojen s vývojovým procesem softwarové aplikace. Jde o to, že optimalizace nepředstavuje pouze jednu fázi vývoje aplikace, je to také filozofie návrhu a vývoje softwaru. Zapomeňte na hlasy, které říkají, že nejprve je nezbytné napsat základní programový kód a ten poté optimalizovat. Lepší bude, když jakési „optimalizační myšlení" zakomponujete do struktury každé funkce, procedury a každého řádku programového kódu, jenž napíšete. Zkuste myslet na to, jak docílit efektivního běhu funkce již tehdy, když teprve začínáte psát její primární kód. Při programování (a optimalizaci obzvlášť) byste měli neustále mít v duchu obraz toho, co chcete udělat. Když víte, co je třeba udělat, můžete se ptát, jak to lze provést. Pokud jste schopni odpovědět na tyto dvě zdánlivě jednoduché otázky, můžete přikročit k otázce třetí a nejdůležitější: „Jak docílit toho, aby se rychlost provádění programového kódu vyšplhala na maximum?". Tak se, sice nepřímo, ovšem znovu, dostáváme k otázce vhodné modifikace stavby kódu.

Pokud budete mít na paměti skutečnost, že optimalizace se proplétá všemi etapami vývojového cyklu (obr. 1), získáte cennou konkurenční výhodu proti všem soupeřům a zvítězíte. A o to jde především, ne?

Obr. 1 – Optimalizace a vývojový cyklus aplikace

Optimalizace objektivní a subjektivní rychlosti

Abyste mohli lépe optimalizovat své aplikace, ujistěte se, že správně rozumíte pojmům **objektivní** a **subjektivní rychlost aplikace**.

Objektivní rychlost si můžete představit jako skutečné tempo práce funkčních modulů aplikace měřené v jistých časových jednotkách (obvykle v sekundách, nebo v milisekundách). Aby bylo možné říci, že rychlost je objektivní, musí být tato plně kvantifikovatelná. To znamená, že je potřebné tuto rychlost exaktně číselně vyjádřit. Kupříkladu si představte, že právě píšete funkci, které je jako argument předáno pole znaků, neboli textový řetězec. Vaším úkolem je napsat kód, který prohodí všechny znaky tak, že první bude posledním, druhý předposledním atd. Dejme tomu, že jste funkci napsali a nyní ji chcete otestovat. Napíšete tedy kód, ve kterém zavoláte funkci, nabídnete ji deset tisíc znaků a budete čekat na výsledek. Když budete schopni změřit dobu exekuce funkce, budete také vědět, jakou objektivní rychlostí funkce disponuje. Objektivní rychlost je vždy daná konstantní číselnou hodnotou, která udává, kolik časových jednotek uběhlo mezi spuštěním jistého procesu (např. již vzpomínané funkce) a vrácením výsledku tohoto procesu (okamžik, kdy funkce dokončila svou práci). V uvedeném příkladu byste třeba mohli dojít k informaci, že funkce vykonala svoje poslání přesně za dvě sekundy.

Jak docílit zvýšení objektivní rychlosti:

- Uvážlivě používat datové typy
- Psát efektivní cykly
- Moudře využívat grafické metody
- Optimalizovat běžně vykonávané operace (např. čtení dat ze souboru)
- Pečlivě nastavit možnosti optimalizace překladače

Na druhé straně, u **subjektivní rychlosti** není hlavním měřítkem tempa práce aplikace kvantifikace provádění jistých fragmentů programového kódu. Kdepak. Subjektivní rychlost aplikace totiž praví, jak dobře se finálnímu uživateli pracuje s vaší aplikací. Subjektivní rychlost tedy působí zejména na

vědomí a pocity uživatele a snaží se ho přesvědčit, že práce s aplikací je rychlá a flexibilní. Pokud ovšem není možné subjektivní rychlost kvantifikovat, tak jak ji máme změřit? Možná, že budete překvapeni, ale jsem přesvědčen, že nejlepším „měřítkem" subjektivní rychlosti aplikace je skutečně sám uživatel. Pokud uživatel uvidí, že aplikace se rychle spouští, pohotově reaguje na jeho podněty a bleskově překresluje okna, sdělí vám, že je s aplikací spokojen. V té chvíli si můžete být jisti, že subjektivní rychlost aplikace je vybroušena na maximum.

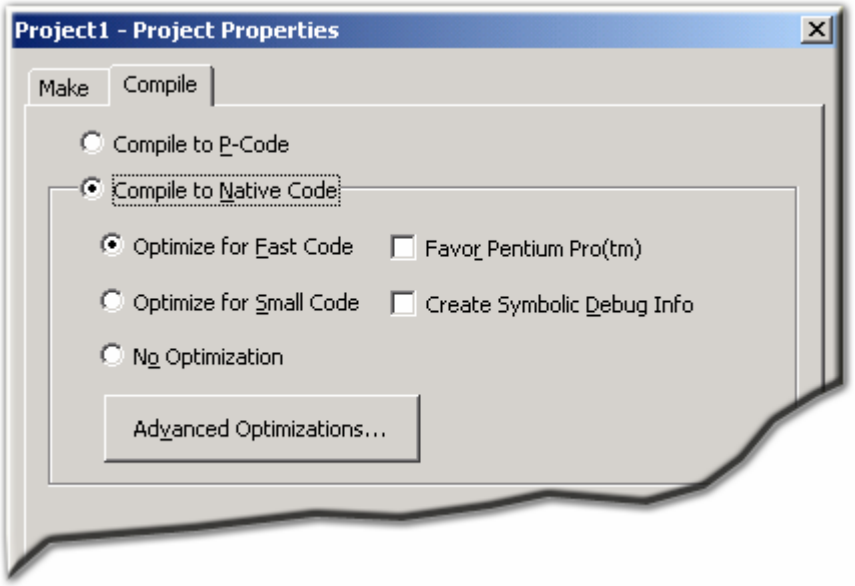
Jak docílit zvýšení subjektivní rychlosti:

- Rychlé reakce na uživatelské vstupy
- Vytříbená správa a obnova oken
- Zabezpečit nepřetržitý kontakt s uživatelem
- Při časově náročných operacích zobrazovat aktuální stav operace (např. pomocí ovládacího prvku **ProgressBar**)
- Minimalizace času potřebného na spuštění aplikace
- Při startu aplikace zobrazovat uvítací obrazovku (angl. **Splash Screen**)

Charakteristika základních optimalizačních voleb překladače

Visual Basic 6.0 disponuje značně výkonným překladačem programového kódu a dokonce vám dovoluje, abyste si upravili parametry samotné kompilace podle své chuti. Pojďme se nejprve podívat, jaké možnosti máte při kompilaci aplikace.

1. Otevřete aplikaci, kterou chcete přeložit a z nabídky **File** vyberte příkaz **Make Jméno_aplikace.exe**.
2. VB otevře okno **Make Project**. V tomto okně klepněte na tlačítko **Options**, na což se zobrazí okno **Project Properties**.
3. V okně **Project Properties** klepněte na záložku **Compile** (obr. 2).



Obr. 2 – Okno **Project Properties** s aktivní záložkou **Compile**

Na záložce **Compile** se nachází několik voleb, které si rozebereme podrobněji.

Jak si můžete všimnout na obrázku, Visual Basic vám nabízí v podstatě jenom dva hlavní režimy kompilace programového kódu aplikace. Jde o kompilaci do **P-kódu** a kompilaci do **nativního kódu**. Podívejme se, co tato záhadná slova znamenají.

Kompilace do P-kódu

V dřívějších verzích jazyka Visual Basic byla kompilace do P-kódu jedinou možností, jak získat samostatný spustitelný soubor (.EXE). P-kód (angl. **Packed Code**) je typem zhuštěného kódu, který nelze provádět přímou cestou (pomocí instrukční sady procesoru). Proto si můžete P-kód představit jako jakýsi pseudokód, který sice je kódem, protože pozůstává z platných programových instrukcí, ovšem na to, aby mohl být přímo prováděn, musí být nejprve dodatečně přeložen. Tuto dodatečnou kompilaci P-kódu má na starosti speciální exekuční nástroj, jehož kód je uložen v doprovodní dynamicky linkované knihovně (.DLL). Exekuční nástroj zabezpečí kompilaci P-kódu na strojový (nativní) kód, jemuž je procesor schopen porozumět. Proces spuštění aplikace, která byla kompilována do P-kódu, můžete vidět na obr. 3.

Obr. 3 – Schematické znázornění překladu P-kódu do nativního kódu

Zamyslíte-li se hlouběji nad stylem práce P-kódu a jeho následného překladu při spuštění aplikace, dojdete k poznání, že ona „just-in-time" kompilace přece jenom spotřebovává jistý čas navíc.

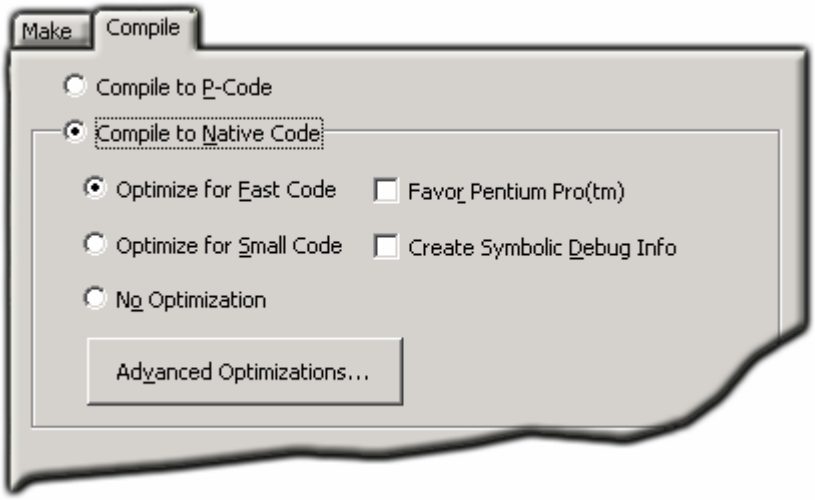
Právě tento dodatečný překlad P-kódu po spuštění aplikace „má na svědomí" skutečnost, že P-kód se, co se rychlosti týče, nemůže nikdy plně srovnávat s nativním kódem.

Na druhé straně je nutno podotknout, že právě existenci P-kódu vděčíme za pokročilé techniky při psaní programového kódu (okamžitá kontrola správnosti zapsaného kódu po stisknutí klávesy Enter) a rovněž tak při jeho odlaďování (zastavování exekuce na předem stanovených kontrolních bodech a pod.). Mimo to se technologie P-kódu také velmi podobá moderním technikám, s kterými se můžete střetnout ve světě .NET. Vzato kolem a kolem můžeme prohlásit, že existence P-kódu je jasným kladem.

Kompilace do nativního kódu

Kompilace do nativního kódu znamená, že programový kód aplikace je překladačem přeložen přímo do strojového kódu, tedy do kódu, kterému procesor rozumí a může ho přímo provádět. Jak asi tušíte, kompilací rovnou do strojového kódu se zabezpečí, že výslední (přeložený) kód aplikace bude prováděn co možná nejrychleji. Ovšem pozor! I když aplikaci podrobíte kompilaci do nativního kódu, nemusíte vždy nutně dosáhnout také navýšení výkonu! Dokonce může dojít k situaci, kdy aplikace přeložena do strojového kódu bude v rychlostním souboji poražena svým P-kódovým protějškem. Proto nelze uplatňovat přístup bezhlavé kompilace do nativního kódu, je zapotřebí efektivněji psát programové konstrukce a optimalizovat ta místa, která jsou z pohledu výkonu aplikace kritická.

Vraťme se ovšem opět do prostředí Visual Basicu. Na záložce **Compile** okna **Project Properties** si můžete po vybrání volby **Compile to Native Code** všimnout další přístupné volby (obr. 4).



Obr. 4 – Zobrazení dalších možností volby **Compile to Native Code**

Povězme si o těchto volbách něco víc:

1. Volba **Optimize for Fast Code**
   Volba **Optimize for Fast Code** je při kompilaci do nativního kódu aktivována standardně. Tato volba sděluje překladači, aby použil optimalizační nastavení pro vygenerování co možná nejrychlejšího kódu. To ve skutečnosti znamená, že kompilátor se bude snažit, aby finální kód byl co možná nejrychleji prováděn, ovšem může dojít ke zvýšení velikosti spustitelného (.EXE) souboru.

2. Volba **Optimize for Small Code**
   V situacích, kdy vám bude záležet, aby velikost spustitelného souboru nepřekročila stanovený limit, můžete použít tuto volbu. Pokud ji aktivujete, kompilátor vygeneruje kód, který nemusí být nutně nejrychlejší, ale bude zaručeně co možná nejmenší.

3. Volba **No Optimization**
   Možnost **No Optimization** můžete použít tehdy, když nebudete chtít použít žádná další optimalizační nastavení. Výsledkem aplikace uvedené volby by měl být spustitelný soubor s přijatelným výkonem a velikostí.

Další volby jsou vizuálně představovány jako zatrhávací pole, což znamená, že je můžete aktivovat současně s jednou z právě vysvětlených voleb (**Optimize for Fast Code**, **Optimize for Small Code** a **No Optimization**).

4. Volba **Favor Pentium Pro(tm)**
   Jestliže píšete aplikace, které budou běžet na počítačích s procesorem Intel Pentium Pro, můžete použít tuto volbu. Vygenerovaný programový kód bude využívat všech dovedností tohoto procesoru, ovšem při exekuci na jiných procesorových stanicích nemusí být tak výkonný.

5. Volba **Create Symbolic Debug Info**
   Pokud ovládáte jazyk C++ a ani vývojové prostředí Visual C++ vám není cizí, můžete při kompilaci projektu zatrhnout toto pole, na což překladač sestaví jak spustitelný soubor aplikace, tak i soubor s koncovkou PDB, jenž bude obsahovat symbolické ladící informace o vygenerovaném spustitelném souboru. Symbolické informace vám pomůžou s dalším odlaďováním aplikace ve vývojových nástrojích, používajících ladící informace ve stylu **CodeView**.

Příště prozkoumáme pokročilá nastavení optimalizace překladače, jež se skrývají pod tlačítkem **Advanced Optimizations**.

Integrované prostředí rubriky Visual Basic	Autor: **Ján Hanák**

Lab

Téma

Začínáme s VB .NET

Úvod do světa .NET (5. díl)

Časová náročnost (min):	Začátečník	Pokročilý	Profesionál
60	1	2	3

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

Milí čtenáři,

dnes prozkoumáme problematiku datových typů, s nimiž se můžete střetnout při programování ve Visual Basicu .NET.

Obsah

Raison d´être datových typů ve VB .NET

Jak jsme si pověděli již v minulé části seriálu, při deklaraci proměnné je potřebné určit datový typ této proměnné. V této souvislosti jste se mohli zeptat, proč se vůbec třeba zabývat s nějakými datovými typy, proč není jednoduše možné používat vždy a na všechny programové operace jenom proměnné jednoho datového typu? Vždyť by to bylo zcela jistě snazší, nebo ne? Inu, lehčí by to jistě bylo, ale podobné myšlenky musí pustit každý z hlavy, když se zamyslí nad významem slov optimální rychlost a výkonnost. Kdybyste pracovali jenom s proměnnými jednoho datového typu, zcela jistě byste se záhy začali potkávat s mnoha problémy. Především, výkonnost aplikace by byla poměrně nízká a při kritických operacích by mohlo dojít buď k neúnosnému zpomalení, nebo přímo ke krachu aplikace. Dále byste mohli zjistit, že i když neprovádíte zrovna rozsáhlé propočty, spotřebováváte stále větší a větší množství paměťového prostoru počítače. A konečně, o optimální výkonnosti a rychlosti programu by nemohlo být ani řeči.

Z uvedených důvodů je potřebné, aby programovací jazyk dokázal efektivně pracovat s rozličnými datovými typy. Jak za chvíli uvidíte, datových typů je ve Visual Basicu .NET značné množství. Kromě toho, že máte k dispozici různé datové typy, je rovněž velmi důležité, abyste se naučili, kdy ten-který datový typ použít. Právě schopnost programátora „vycítit", kdy použít ten, nebo jiný datový typ, patří mezi jeho největší přednosti. To již však poněkud zacházíme do problematiky optimalizace programového kódu, a proto si raději ukažme, s jakými datovými typy se budete moci setkat ve Visual Basicu .NET.

Přehled datových typů ve VB .NET

Celou množinu datových typů Visual Basicu .NET lze rozdělit na následující podmnožiny:

1. **Celočíselné datové typy**
2. **Datové typy pro uchovávání čísel s pohyblivou desetinnou čárkou**
3. **Ostatní datové typy**

Celočíselné datové typy

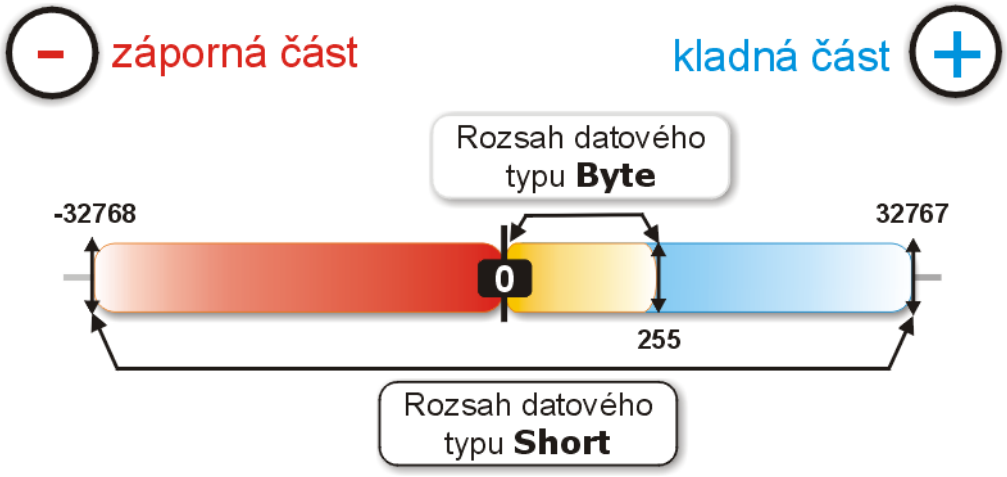
Myslím, že se zástupci této kategorie se budete střetávat asi nejčastěji. Jak je z názvu zřejmé, tato podmnožina datových typů je určena na uchovávání celočíselných hodnot. Přehled konkrétních datových typů můžete vidět v tab. 1.

Datový typ ve VB .NET	Charakteristika	Rozsah	.NET ekvivalent
Byte	8-bitové celé číslo bez znaménka	<0, 255>	System.Byte
Short	16-bitové celé číslo se znaménkem	<-32768, 32767>	System.Int16
Integer	32-bitové celé číslo se znaménkem	$<-2^{31}, 2^{31}-1>$	System.Int32
Long	64-bitové celé číslo se znaménkem	$<-2^{63}, 2^{63}-1>$	System.Int64

Tab. 1

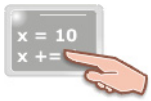
Rozsah datového typu si můžete představit jako definiční obor datového typu (přesněji řečeno, jde o definiční obor proměnné daného datového typu).

Pokud vám není jasné, co znamenají textové charakteristiky „číslo bez znaménka" a „číslo se znaménkem" uvedené v tabulce, nemějte obavy, zde je vysvětlení. Nejprve si představte jednoduchou číselnou os. Pokud rozsah požadovaného datového typu leží pouze v kladné části této pomyslné číselné osy, tak říkáme, že daný datový typ může uchovávat celá čísla bez znaménka (tedy jenom kladná čísla). Na druhé straně, když rozsah datového typu leží na obou stranách osy, pak říkáme, že jde o datový typ, jenž je schopen ukládat celá čísla také se znaménkem (tedy kladná i záporná čísla). Pro lepší ilustraci se podívejte na obr. 1.



Obr. 1 – Názorné zobrazení rozsahu vybraných datových typů pomocí pomyslné číselné osy

Pokud budete chtít deklarovat proměnnou jednoho z uvedených celočíselných datových typů, budete postupovat tak, jak jsme si ukázali již minule. Použijete tedy příkaz **Dim** s jménem proměnné, za kterým bude následovat klíčové slovo **As** a specifikace datového typu. Příklad:




```
Dim proměnná_ABC As Short
proměnná_ABC = 120
```

Datové typy pro uchovávání čísel s pohyblivou desetinnou čárkou

Ve Visual Basicu .NET můžete nalézt tři datové typy, které vám umožní pracovat s čísly s desetinnou čárkou. Jejich popis najdete v tab. 2.

Datový typ ve VB .NET	Charakteristika	Přesnost	.NET ekvivalent
Single	32-bitové číslo s pohyblivou desetinnou čárkou	7 desetinných míst	System.Single
Double	64-bitové číslo s pohyblivou desetinnou čárkou	15-16 desetinných míst	System.Double
Decimal	128-bitové číslo s pohyblivou desetinnou čárkou	28 desetinných míst	System.Decimal

Tab. 2

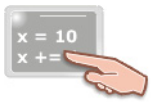
Specifikace rozsahu jednotlivých datových typů se nachází v tab. 3.

Datový typ ve VB .NET	Rozsah
Single	<+/- 1,4 x 10^{-45}, +/- 3,4 x 10^{38}>
Double	<+/- 5,0 x 10^{-324}, +/- 1,7 x 10^{308}>
Decimal	<+/- 1,0 x 10^{-28}, +/- 7,9 x 10^{28}>

Tab. 3

Datový typ **Single** se vyznačuje nejmenší přesností, a proto jej můžete efektivně používat ve chvílích, když sice budete potřebovat uchovat číslo s desetinnou čárkou, no nebudete vyžadovat přílišnou přesnost při provádění požadované operace. Ačkoli je datový typ **Decimal** nejpřesnější (může uchovat až 28 číslic za desetinnou čárkou), spotřebuje také největší množství paměťového prostoru (16 bajtů). Pokud nebudete výslovně požadovat vysokou přesnost, nebo provádět složité finanční kalkulace, není mnoho důvodů pro využití služeb tohoto datového typu. Jakousi zlatou střední cestu představuje datový typ **Double**, reprezentovaný v podobě čtyřiašedesátibitového čísla s pohyblivou desetinnou čárkou. Jestliže budete chtít upřednostnit rovnováhu mezi nabízenou přesností a spotřebou paměti, je velmi pravděpodobné, že právě datový typ **Double** se stane vaším favoritem.

Příklad použití:




```
Dim koeficient As Double
koeficient = 1000.255
```

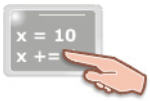
Ostatní datové typy

Kromě datových typů pro práci s celými čísly a čísly s desetinnou čárkou si musíme povědět také o dalších datových typech, které se velmi často označují i jako „nečíselné" datové typy. Jde o tyto datové typy:

1. **Boolean** (.NET ekvivalent: **System.Boolean**)
   Datový typ **Boolean** se používá pro reprezentaci dvou pravdivostních stavů proměnné: **True** a **False**. S proměnnými tohoto typu se můžete setkat například při rozhodovacích konstrukcích a cyklech (programový kód rozhodovací konstrukce se provede, jestliže je pravdivostní hodnota proměnné **True**, cyklus se může opakovat tak dlouho, dokud proměnná obsahuje hodnotu True atd.). Proměnné datového typu **Boolean** jsou uchovávány jako 16bitová čísla.

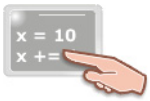
Když jsou číselné datové typy konvertovány do datového typu **Boolean**, postupuje se podle následujícího pravidla: Všechny číselné hodnoty kromě nuly (0) představují pravdivostní hodnotu **True** a samotná nula odpovídá hodnotě **False**.

Příklad aplikace:




```
Dim podmínka As Boolean
podmínka = True
```

2. **Char** (.NET ekvivalent: **System.Char**)
   Datový typ **Char** vám pomůže, budete-li chtít pracovat se symboly Unicode v rozsahu <0, 65535>. Pokud budete chtít deklarovat proměnnou **Znak** datového typu **Char** a přiřadit do ní znak „W", můžete to udělat takto:

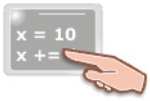



```
Dim znak As Char
znak = "W"c
```

Všimněte si prosím, že znak „W" je umístěn do dvojitých uvozovek a je k němu přidán také sufix „**c**". Když použijete tento sufix, tak Visual Basicu .NET sdělujete, aby na uvedený symbol nahlížel jako na znak datového typu **Char**.

Použití sufixu je nevyhnutné jenom při programování s volbou **Option Strict** nastavenou na hodnotu **On**. O použití této volby si povíme příště.

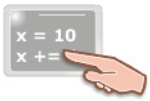
3. **String** (.NET ekvivalent: **System.String**)
   I když je práce s textovými řetězci v jiných jazycích (zejména C a C++) dosti náročná, VB .NET vás ani tentokrát nenechá na holičkách. Nabízí vám datový typ **String**, jenž vám poslouží při ukládání textových řetězců jakéhokoliv druhu. Jenom pamatujte na to, že textový řetězec musí být umístěn do dvojitých uvozovek tak, jak je to uvedeno níže:




```
Dim pozdrav As String
pozdrav = "Optimalizovanému kódu zdar!"
```

Proměnné datového typu **String** si můžete představit jako sekvenci 16bitových čísel bez znaménka v rozsahu <0, 65535>. Pokud rozvineme myšlenky, každé číslo si můžete dále představit jako symbol Unicode. Prostřednictvím analogie pak dojdeme k tomu, že datový typ **String** je posloupností více instancí datového typu **Char**.

4. **Date** (.NET ekvivalent: **System.DateTime**)
Chcete-li například zjistit, kolik dní uplynulo ode dne vašeho narození, zcela jistě budete potřebovat datový typ **Date**. Proměnné datového typu **Date** jsou schopny pojmout popis data, a to od 1. ledna roku 1 až do 31. prosince roku 9999. Tento datový typ je rovněž možné použít pro ukládaní času, od půlnoci (0:00:00) až do konce dne (23:59:59).




```
Dim datum_narozeni As Date
Dim dnesni_datum As Date
Dim rozdil As Long

datum_narozeni = #3/21/2002#
dnesni_datum = #3/21/2003#

rozdil = DateDiff(DateInterval.Day, _
datum_narozeni, dnesni_datum)

MessageBox.Show(rozdil.ToString)
```

Na uvedené ukázce programového kódu jsme si ukázali základní náležitosti, které platí při práci s datovým typem **Date**. Předně je potřebné, abyste měli na paměti, že datum, které budete ukládat do příslušné proměnné, musí být zapsáno v tomto formátu:

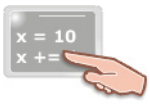
#měsíc/den/rok#

Aby bylo možné zjistit, kolik dní tvoří interval stanovený dvěma daty, můžeme zavolat funkci **DateDiff**. Funkci zadáme počáteční i koneční datum a ještě specifikujeme, aby výsledkem byl počet dní mezi zadanými daty.

5. **Object**
Datový typ **Object** je schopen pojmout hodnotu jakéhokoliv datového typu. Proměnné tohoto typu můžou obsahovat jak celá čísla, tak numerické hodnoty s pohyblivou desetinnou čárkou a rovněž tak hodnoty ostatních datových typů. Datový typ **Object** má rozsah 32 bitů.

Všechny datové typy, či už jde o hodnotové nebo referenční typy, jsou odvozeny z bázového typu **Object**. Jinak řečeno, datový typ **Object** je základem pro všechny ostatní datové typy.

Příklad použití:




```
Dim obj As Excel.Application
obj = CType(CreateObject("Excel.Application"), _
Excel.Application)
obj.Visible = True
```

Uživatelsky definované datové typy

Budete-li chtít, můžete definovat také své vlastní datové typy. Uživatelsky definované typy (UDT) se tvoří pomocí příkazu **Structure**, jenž má tento tvar:

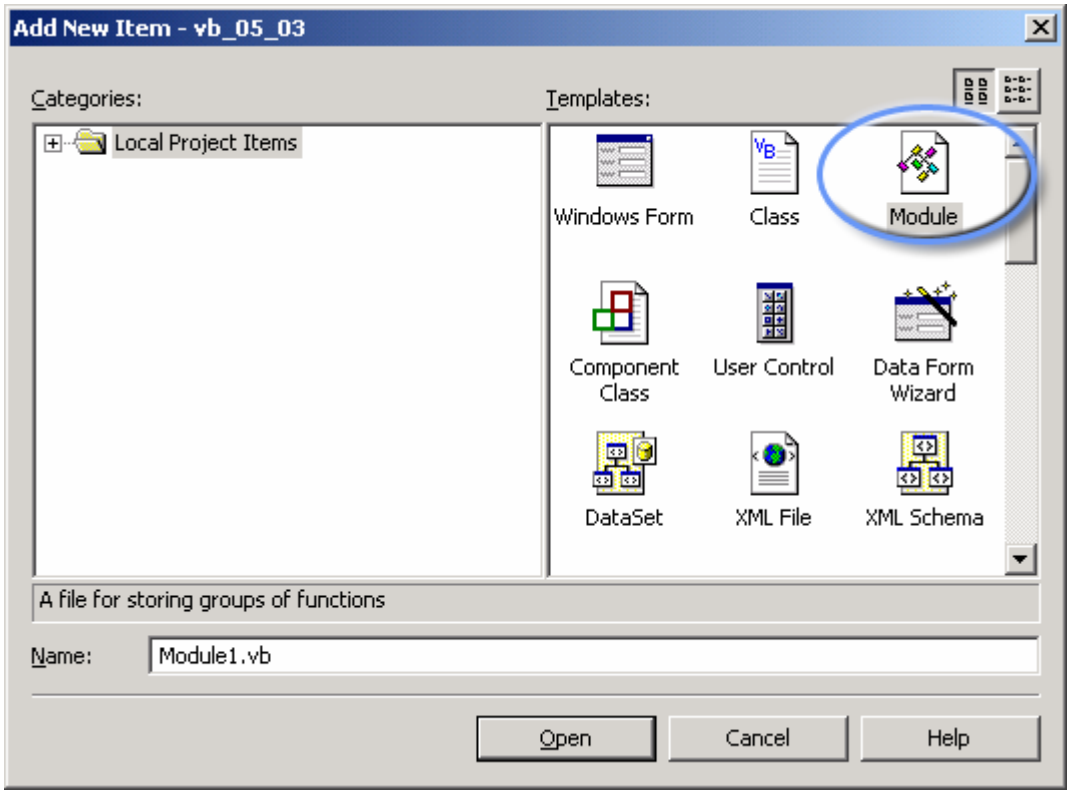
```
Structure Jméno_uživatelsky_definovaného_datového_typu
` deklarace členů struktury
End Structure
```

Struktura obvykle obsahuje několik členů. Členem se rozumí deklarovaná proměnná podporovaného datového typu. Předpokládejme, že budete chtít vytvořit uživatelsky definovaný datový typ s názvem **Student**. Už víte, že k tomu, abyste splnili tento úkol, budete potřebovat vytvořit strukturu. Dále předpokládejme, že budete chtít sledovat jméno studenta a jeho průměrný prospěch. Proto vytvoříte dva členy struktury, jeden s názvem **Jméno_studenta** (datový typ **String**) a další s názvem **Průměrný_prospěch** (datový typ **Single**).

Programový kód pro zápis charakteristiky uživatelsky definovaného datového typu musíte umístit před všechen kód třídy formuláře, nebo do samostatného modulu. V následujícím textu bude ozřejměna druhá varianta.

Po vytvoření struktury bude možné deklarovat proměnnou datového typu **Student**. Tuto proměnnou posléze inicializujeme na požadované hodnoty. Jak na to, si ukážeme v následující praktické programové ukázce. Postupujte podle vyznačených instrukcí:

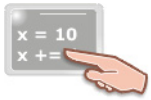
1. Spusťte Visual Basic .NET a vytvořte nový projekt.
2. Vyberte nabídku **Project** a klepněte na položku **Add New Item**.
3. V sekci **Templates** okna **Add New Item** vyberte ikonu **modulu** (obr. 2).



Obr. 2 – Výběr modulu v okně **Add New Item**

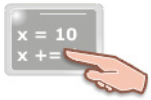
4. Aktivujte tlačítko **Open**, na což se do projektu přidá soubor s modulem (je standardně pojmenován jako **Module1.vb**).

5. Vložte do kódu modulu programový kód pro vytvoření struktury **Student**:




```
Module Module1
    Structure Student
        Dim Jméno_studenta As String
        Dim Průměrný_prospěch As Single
    End Structure
End Module
```

6. Na formulář umístěte tlačítko (nechte jej standardně pojmenované) a jeho obsluhu události **Click** vyplňte následujícím kódem:




```
Dim student1 As Student
student1.Jméno_studenta = "Jan Novák"
student1.Průměrný_prospěch = 3.2

MessageBox.Show("Jméno studenta: " & student1.Jméno_studenta & _
Chr(13) & "Průmerný prospěch: " & student1.Průměrný_prospěch, _
"Školní záznam")
```

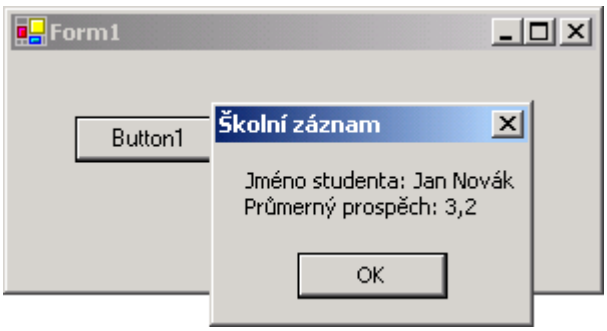
V této chvíli bych se rád přistavil při tomto fragmentu programového kódu a vysvětlil vám, jak pracuje. První důležitou věcí je samotná deklarace proměnné. Jak vidíte, proměnná **student1** je deklarována použitím uživatelsky definovaného typu **Student**. Poněkud nejasným se vám může zdát také způsob, jakým je proměnná **student1** inicializována. Při inicializaci UDT totiž platí podobná pravidla jako při inicializaci vlastností. Proto musí za názvem proměnné následovat tečkový operátor (**.**) a jméno platného členu struktury, pomocí které byla naše proměnná (**student1**) vytvořena. Členu struktury je nakonec přiřazena hodnota na základě specifikace jeho datového typu. Podobnou koncepci, jaká platila při inicializaci proměnné, použijete i ve chvíli, kdy budete chtít zpětně získat hodnoty, které členy struktury obsahují.

Jestliže chcete rozdělit programový kód na několik řádků, na konci jednoho řádku zadejte **mezeru** a znak **podtržení** (**_**). Dále stiskněte klávesu Enter a pokračujte v psaní kódu na následujícím řádku.

Pro spojení řetězce a proměnné v signatuře metody **Show** třídy **MessageBox** můžete použít symbol **zřetězení** (**&**). Simulaci aktivace klávesy Enter docílíte, když funkci **Chr** předáte ASCII kód této klávesy (jenž má hodnotu 13).

Věřím, že na první pohled může být tento postup použití UDT poněkud frustrující, no nemějte obavy, zanedlouho bude pro vás vytváření a používání uživatelsky definovaných typů hračkou.

7. Výsledek práce kódu celé praktické ukázky můžete vidět na obr. 3.



Obr. 3 – Výsledek práce UDT

Integrované prostředí rubriky Visual Basic	Autor: Ján Hanák

Téma

VB .NET

Programátorská laboratoř

Prostor pro experimentování

Časová náročnost (min):	Začátečník	Pokročilý	Profesionál
60	1	2	3

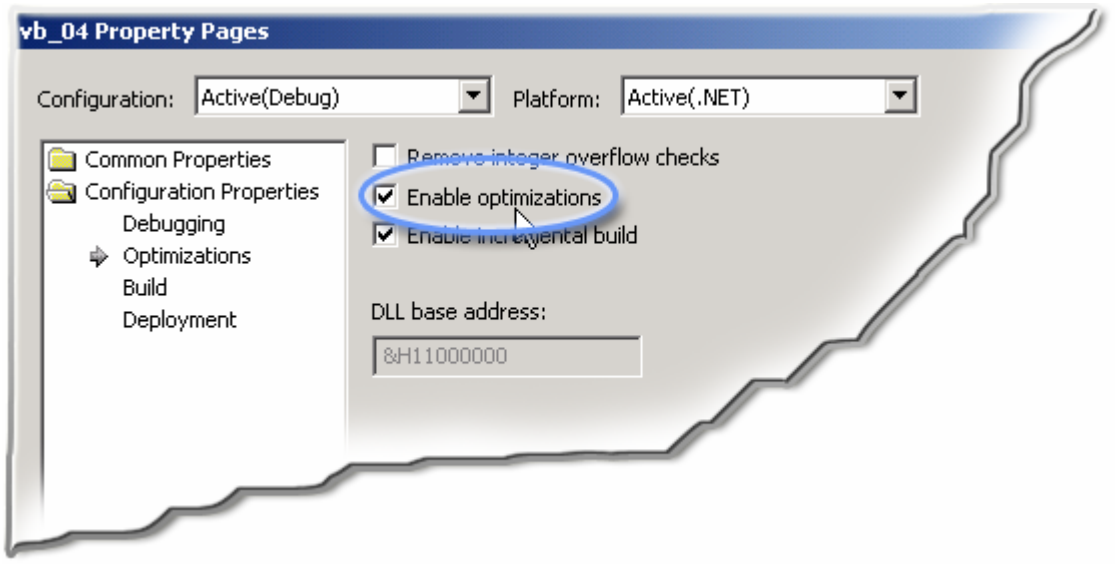
Použitý operační systém :	Windows 2000 SP3
Hlavní vývojový nástroj :	Visual Basic .NET 2002
Další vývojový software :	Žádný
Jiný software :	Žádný

VB .NET	Aktivace optimalizace kompilátora	2	0:10
VB .NET	Použití volitelných (optional) parametrů	2	0:15
VB .NET	Jak zjistit, zdali byla stisknuta klávesa Enter	1	0:10
VB .NET	Ukázka práce strukturované správy chyb	1	0:10
VB .NET	Tvorba kontextové nabídky klonováním	3	0:15

Aktivace optimalizace kompilátora

Pokud pracujete ve Visual Basicu .NET, tento typ bude pro vás zcela jistě užitečný. Každý programátor se jednou dostane do situace, kdy zatouží po rychlejším provádění svého programové kódu. Jak možná víte, čas strávený pečlivým studiem kódu a promýšlením, jak zefektivnit tu či onu proceduru nebo funkci, bývá leckdy neúnosně dlouhý. Jestliže patříte do skupiny programátorů, kteří by pro optimalizaci kódu udělali takřka všechno, vyzkoušejte následující postup:

1. Spusťte Visual Basic .NET a otevřete váš projekt.
2. V okně **Solution Explorer** klepněte pravým tlačítkem myši na název vaší aplikace. Měla by se objevit kontextová nabídka. Z nabídky vyberte poslední položku **Properties**. Za okamžik se zobrazí okno **Property Pages**.
3. Zaměřte svou pozornost na seznam, jenž se nachází v levé části dialogového okna. V seznamu klepněte na položku **Configuration Properties**.
4. Aktivovaná položka se „rozbalí" a ukáže vám svůj obsah. V této chvíli klepněte na příkaz **Optimizations**.
5. Zatrhněte pole **Enable optimizations** (obr. 1).



Obr. 1 – Zapnutí optimalizace kompilátora v okně **Property Pages**

6. Aktivujte tlačítko OK.

Jakmile v budoucnu vydáte příkaz na sestavení aplikace .NET, kompilátor použije různá optimalizační nastavení. Při své práci se bude snažit, aby vygenerovaný výstup byl kapacitně méně náročný, rychlejší a efektivnější.

Protože kompilátor při své práci může v zásadní míře změnit strukturu programového kódu na úrovni jazyka MSIL, můžete mít po sestavení aplikace se zapnutou optimalizací kompilátoru problémy s pozdějším dalším odlaďováním této aplikace. Proto se doporučuje, abyste automatickou optimalizaci kompilátoru používali jenom pro plně odladěnou aplikaci, která je generována v režimu **Release**.

Ačkoliv se jedná o automatickou optimalizaci, je možné, že v některých případech dosáhnete navýšení výkonu vaší aplikace. Neměli byste ovšem očekávat, že zapnutí uvedené optimalizační volby zcela nahradí komplexní proces pečlivé kontroly a „manuální" optimalizace programového kódu vaší aplikace.

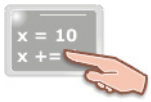
Použití volitelných (optional) parametrů

Pod pojmem volitelný parametr se rozumí parametr v signatuře funkce nebo procedury Sub, který je uveden klíčovým slovem **Optional**. Pro volitelné parametry platí následující pravidla:

1. Všem volitelným parametrům musejí být přiřazeny inicializační hodnoty.
2. Inicializační hodnota musí mít konstantní povahu (např. hodnota volitelného parametru může být inicializována numerickou konstantou).
3. Každý parametr, jenž následuje za volitelným parametrem musí být rovněž volitelný (není tedy přípustné míchat povinné a volitelné parametry v signatuře funkce či procedury).

Dobrá, tolik říká teorie a teď se podívejme, jak vypadá deklarace volitelných parametrů v praxi. Postupujte dle těchto instrukcí:

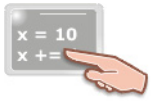
1. Zapište do okna editoru pro zápis programového kódu kód funkce **Propočet**:



```
Private Function Propočet _
(ByVal cena As Single, _
Optional ByVal množtví As Integer = 10) As Single

    MessageBox.Show((cena * množtví).ToString)
End Function
```

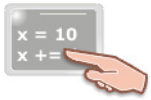
Jde o soukromou funkci, která vypočítává cenu obstarání nějaké komodity. Funkce pracuje se dvěma parametry. Prvním parametrem je cena výrobku v podobě datového typu **Single**. Druhý parametr představuje nakoupené množství výrobků a jak si můžete všimnout, jde o volitelný parametr, kterého datový typ je **Integer**. Jak již víte, volitelný parametr musí být inicializován (zde na hodnotu deseti kusů výrobků). Výsledkem práce funkce je výpočet celkové ceny obstarání, která se posléze zobrazí v dialogovém okně.

2. Zkuste přidat na formulář tlačítko a do zpracovatele události **Click** tlačítka umístěte tento řádek kódu:



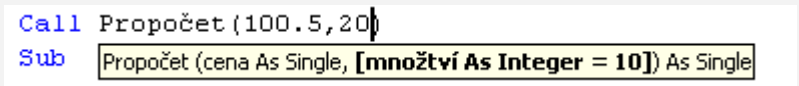
```
Call Propočet(100.5)
```

V případě, že nezadáte množství výrobků, které jste nakoupili, použije se standardní množství (10 ks). Takto by vás celý nákup stál rovných 1005 korun. Jestliže budeme abstrahovat od obchodních záležitostí a budeme se soustředit jenom na programový kód, můžeme říct, že právě toto je hlavní úkol volitelného parametru. Když totiž není zadána hodnota druhého parametru, použije se předem stanovená hodnota, což ušetří trochu námahy při psaní signatury funkce. Pokud však chcete, můžete hodnotu druhého parametru změnit, třeba na 20:



```
Call Propočet(100.5, 20)
```

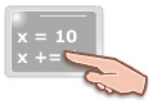
Při zápisu hodnoty druhého parametru vám opět pomůže technologie IntelliSense a poví vám, že druhý parametr je volitelný a rovněž zobrazí i jeho konstantní hodnotu. Jenom připomenu, že volitelnost druhého parametru je v popisku indikována přítomností hranatých závorek (viz obrázek).



3. I tentokrát se vypočte správná hodnota, kterou uvidíte v dialogovém okně.

Jak zjistit, zdali byla stisknuta klávesa Enter

Zjistit, zdali byla aktivována klávesa Enter, lze poměrně jednoduše pomocí následujícího kódu:



```
Private Sub TextBox1_KeyPress(ByVal sender As Object, _
ByVal e As System.Windows.Forms.KeyPressEventArgs) _
Handles TextBox1.KeyPress

    If e.KeyChar = Microsoft.VisualBasic.ChrW(13) Then
        MessageBox.Show("Byla aktivována klávesa ENTER.")
    End If

End Sub
```

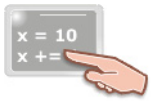
V uvedeném výpisu se předpokládá, že na formuláři se nachází textové pole s názvem **TextBox1**. A jak pracuje uvedený kód? Skutečnost, zdali byla požadovaná klávesa stisknuta či nikoliv, budeme testovat v obsluze události **KeyPress** textového pole. K události **KeyPress** dojde vždy, když uživatel stiskne klávesu, která disponuje příslušným ASCII kódem. Mezi takovéto klávesy patří alfabetické a numerické klávesy, podobně jako i některé jiné klávesy (jako je např. klávesa Enter).

Aby bylo možno rozeznat, která klávesa byla stisknuta, můžeme testovat hodnotu vlastnosti **KeyPressEventArgs.KeyChar**. Na identifikaci klávesy Enter použijeme funkci **ChrW**. Této funkci jako parametr předáme ASCII kód klávesy Enter, který je reprezentován číselnou hodnotou 13. Vždy, když uživatel zapíše do textového pole několik znaků, testujeme, zdali ASCII kód zadaného znaku neodpovídá ASCII kódu klávesy Enter. Je-li tomu tak, zobrazí se dialogové okno se zprávou, která říká, že byla zmáčknuta hledaná klávesa.

Ukázka práce strukturované správy chyb

Visual Basic .NET přichází s novinkami také v oblasti ošetřování chyb a odlaďování programů. Střetáváme se zde s tzv. strukturovanou správou chyb, podstatu které si ihned vysvětíme. Jestliže máte zkušenosti s Visual Basicem 6.0, bude pro vás lepší, když budou představeny oba systémy správy chyb (tak ve VB 6.0, jako i ve VB .NET) a vy budete moci porovnat prezentované rozdílnosti.

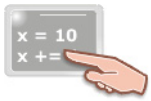
Pokud jste chtěli ve Visual Basicu 6.0 vložit do funkce nebo procedury chybovou rutinu, ve většině případů programový kód této chybové rutiny vypadal asi takto:



```
Private Sub btn1_Click()
On Error GoTo Napravit_chybu

Dim x As Integer, y As Integer
x = 1000
y = 0

Me.Caption = x \ y

Exit Sub

Napravit_chybu:
MsgBox "Tuto operaci nelze provést."
End Sub
```

V příkladě můžete pozorovat autorovu snahu o vyvolání chyby č. 11 s názvem **Division by zero** (Dělení nulou). Protože takováto operace je z matematického hlediska nepřípustná, vyskytne se chyba. Abychom mohli chybu tohoto typu ošetřit, vkládáme do programového kódu konstrukci **On Error Goto Návěstí**, kde **Návěstí** představuje tu sekci kódu, která se má provést v případě, že dojde v programu k chybě. Když se vrátíme k uvedenému kódu, uvidíme, že název **Návěstí** je **Napravit_chybu**. Můžeme tedy říci, že vždy, když dojde v této událostní proceduře k chybě, exekuce programu bude pokračovat prováděním kódu, jenž je umístěn pod názvem **Návěstí** (tedy pod řetězcem **Napravit_chybu**).

Pojďme se nyní podívat na to, jaká je situace v tomto směru ve VB .NET. Pro analogii použijeme již uvedenou podobu programového kódu (i když samozřejmě uzpůsobenou pro novou verzi jazyka) a budeme pozorovat rozdíly.



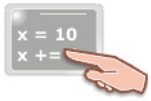

```
Private Sub btn2_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles btn2.Click
    Try
        Dim x, y As Integer
        x = 1000
        y = 0

        Me.Text = x \ y
    Catch
        MessageBox.Show("Tuto operaci nelze provést.")
    End Try
End Sub
```

Zde máte možnost vidět ukázku práce moderní strukturované správy chyb. O co vlastně jde? Zjednodušeně lze prohlásit, že všechen kód, o kterém si myslíme, že by mohl způsobit jakoukoliv chybu, umístíme do bloku **Try**. V bloku **Try** dále vytvoříme jeden (nebo i několik) bloků **Catch**. Blok **Catch** obsahuje programový kód pro ošetření vzniklé chyby. Když tedy dojde k chybě, exekuce programu se přenese do bloku **Catch** a provede se ten kód, jenž se v daném bloku nachází (přesněji dojde k zobrazení chybového hlášení pomocí třídy **MessageBox**). Celou chybovou rutinu uzavírá příkaz **End Try**.

Jenom pro zajímavost připomenu, že i v prostředí jazyka VB .NET můžete používat starší (nestrukturovanou) správu chyb prostřednictvím konstrukce **On Error Goto Návěstí**. Přestože je tento způsob ošetřování chyb stále podporován, měli byste dát přednost nové koncepci psaní chybových rutin.

Tvorba kontextové nabídky klonováním

Náplní poslední programové ukázky bude zhotovení kontextové nabídky klonováním jedné z nabídek hlavního menu programu. Přitom tak hlavní nabídku, jako i kontextuálně vázanou nabídku vytvoříme pomocí programového kódu. Podívejte se nejprve na samotný kód a pak si k němu řekneme pár slov.




```
    Dim hlavní_nabídka As New MainMenu()
    Dim nabídka1 As New MenuItem()
    nabídka1.Text = "Nabídka č.1"
    hlavní_nabídka.MenuItems.Add(nabídka1)
```

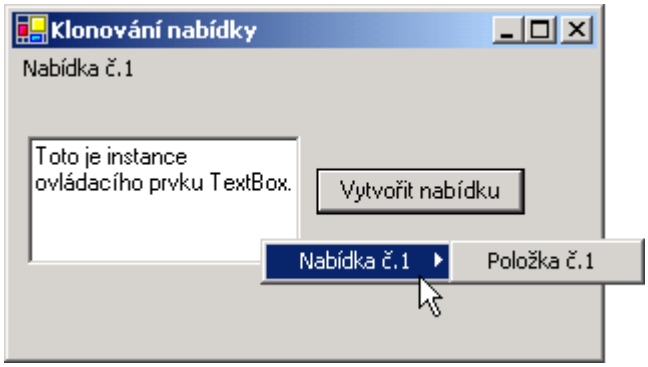
Programový kód pokračuje na následující straně

```
nabídka1.MenuItems.Add("Položka č.1")
Me.Menu = hlavní_nabídka


Dim kontextová_nabídka As New ContextMenu()
kontextová_nabídka.MenuItems.Add(nabídka1.CloneMenu())

TextBox1.ContextMenu = kontextová_nabídka
```

V první řadě je potřebné vytvořit novou hlavní nabídku (**hlavní_nabídka**). Dále vytvoříme jednu parciální nabídku (**nabídka1**), kterou posléze přidáme do hlavní nabídky. Do právě vytvořené parciální nabídky přidáme pomocí metody **Add** jednu položku. Nakonec určíme, že hlavní nabídka se má stát právoplatnou nabídkou aktivní instance formuláře.

Naše další úsilí je spojené s tvorbou kontextuálně vázané nabídky (s názvem **kontextová_nabídka**). Právě v této chvíli se dostáváme k zlatému hřebíku představení, kterým je aplikace metody **CloneMenu**. Tato metoda vytváří přesnou kopii parciální nabídky **nabídka1** (a všech součástí této nabídky). Kopie je následně přiřazena kontextové nabídce. Poslední řádek pak determinuje, že vytvořená kontextová nabídka se bude vztahovat na požadovanou instanci ovládacího prvku **TextBox**.

Pro praktické vyzkoušení práce programového kódu můžete na formulář umístit jednu instanci ovládacího prvku **Button** a rovněž jednu instanci ovládacího prvku **TextBox**. Událostní proceduru **Click** tlačítka vyplňte uvedeným kódem a spusťte testovací aplikaci. Klepněte nejdříve na tlačítko. Uvidíte, že se vytvoří hlavní nabídka, parciální nabídka a jedna položka této nabídky. Když klepnete pravým tlačítkem myši na instanci ovládacího prvku **TextBox**, objeví se kontextová nabídka, která vznikla klonováním (obr. 2).



Obr. 2 – Kontextová nabídka vytvořená klonováním

Integrované prostředí rubriky Visual Basic	Autor: **Ján Hanák**

Téma měsíce

Optimalizace (2. díl)

Časová náročnost (min): **60**	Začátečník	Pokročilý	Profesionál
			

Použitý operační systém :	Windows 2000 SP3
Hlavní vývojový nástroj :	Visual Basic 6.0 SP 5
Další vývojový software :	Žádný
Jiný software :	Žádný

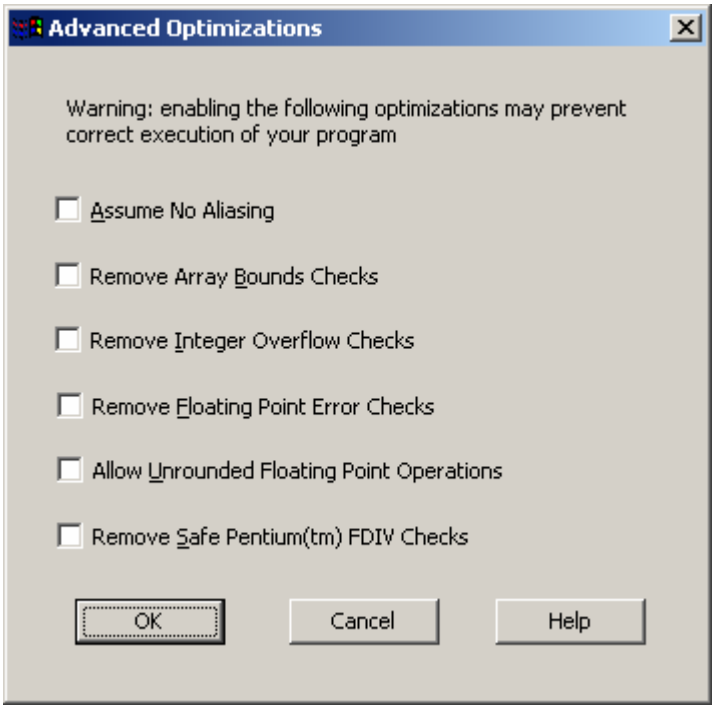
Vážení čtenáři,

úvod a jádro druhého dílu našeho miniseriálu o procesu optimalizace bude patřit přehledu pokročilých optimalizačních nastavení, jež se ukrývají pod volbou **Advanced Optimizations** na záložce **Compile** okna **Project Properties**. V závěru se společně podíváme na způsob měření tempa exekuce událostní procedury. Přeji příjemné počtení.

Obsah

Charakteristika možností pokročilých optimalizačních voleb překladače VB 6.0

Jestliže klepněte na záložce **Compile** okna **Project Properties** na tlačítko **Advanced Optimizations**, spatříte toto okno:

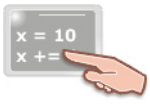


Obr. 1 – Podoba okna **Advanced Optimizations**

Jak si můžete všimnout, okno vám nabízí poměrně slušnou kolekci možností, jak ovlivnit práci kódu vaší aplikace. Ještě předtím, než přejdeme na charakteristiku jednotlivých optimalizačních voleb, bych vás chtěl upozornit na velmi důležitou skutečnost. Totiž tak, jak je možné, že aktivací vhodné volby bude vaše aplikace běžet svižněji, je rovněž možné, že právě nesprávným výběrem optimalizační volby může dojít k nečekaným chybám při provádění programového kódu, nebo, ještě hůř, ke krachu celé aplikace. Vezměte proto do úvahy tato upozornění a vždy postupujte tak, že nejprve zapnete jednu optimalizační volbu, následně spustíte a otestujete aplikaci a až poté, když všechno proběhne tak, jak má, můžete experimentovat s dalšími volbami okna **Advanced Optimizations**.

Pojďme se tedy podívat, jaké kouzelní možnosti nám okno **Advanced Optimizations** nabízí:

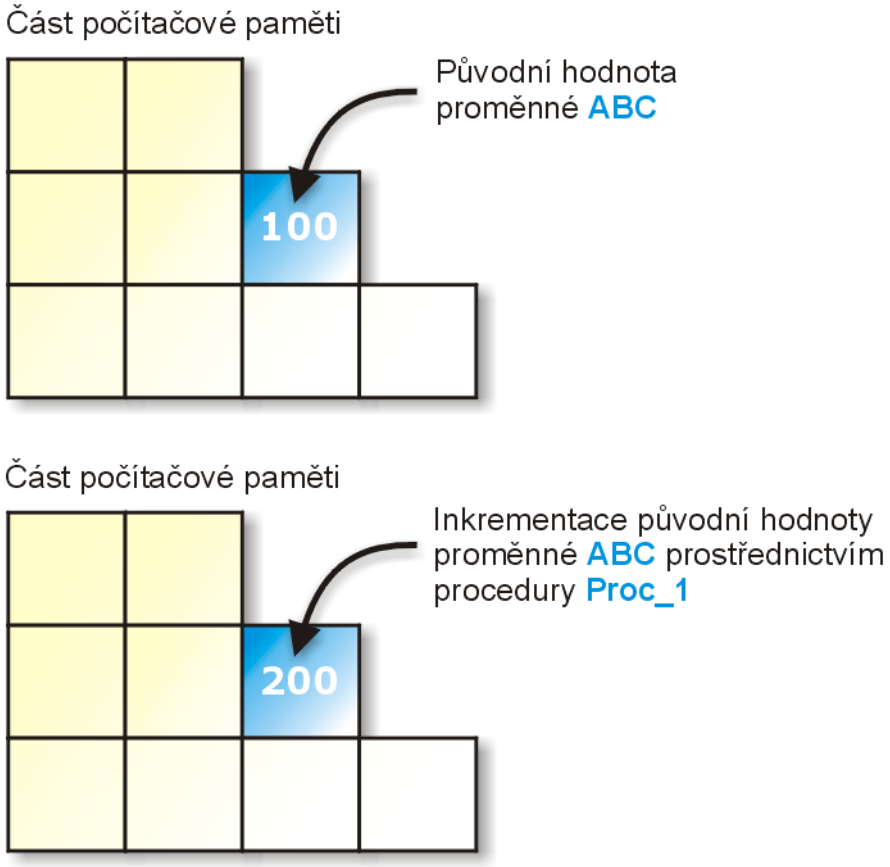
1. **Assume No Aliasing**
   Aktivací této volby oznamujete překladači, že ve vaší aplikaci nepoužíváte techniku, jejíž jméno je **aliasing**. Při aplikaci uvedené techniky dochází k tomu, že se na jedno a totéž paměťové místo (přesněji paměťovou adresu) odvoláváte stejným názvem. Typickým příkladem použití této techniky je předávání argumentů odkazem (pomocí klíčového slova **ByRef**). Techniku **aliasing** předvádí následující příklad:




```
Private Sub btnTlačítko1_Click()
Dim ABC As Integer
ABC = 100

Call Proc_1(ABC)

MsgBox ABC
End Sub

Private Sub Proc_1(ByRef hodnota As Integer)
hodnota = hodnota + 100
End Sub
```

V událostní proceduře **Click** tlačítka s názvem **btnTlačítko1** je deklarována celočíselná proměnná **ABC**, které je posléze přiřazena hodnota 100 jednotek. Co ve skutečnosti provádějí tyto dva řádky kódu? Deklarace proměnné nařizuje, aby bylo pro proměnnou vyhrazeno dostatečné paměťové místo. Přiřazovací příkaz zabezpečuje naplnění proměnné smysluplnou hodnotou. Pokud byste chtěli situaci studovat ještě podrobněji, zjistili byste, že proměnná **ABC** je umístěna na zásobník.

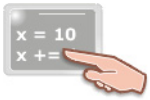
V další etapě je volána Sub procedura s názvem **Proc_1**, které je jako argument předána hodnota proměnné **ABC**. Ze zápisu signatury procedury snadno poznáte, že proceduře je předáván vstupní argument odkazem, což znamená, že proceduře není poskytnuta aktuální hodnota proměnné **ABC** (100), ale paměťová adresa (přesněji ukazatel na tuto adresu), na které je uložena hodnota proměnné **ABC**. Tímto způsobem získává procedura **Proc_1** schopnost přímého přístupu k hodnotě proměnné **ABC**, a tedy může tuto hodnotu lehce změnit, což se také o chvíli později stane. Řádek **hodnota = hodnota + 100** inkrementuje původní hodnotu proměnná **ABC**. Pokud byste chtěli být přesnější, mohli byste říct, že uvedený přiřazovací příkaz zabezpečí inkrementaci hodnoty proměnné, která se nachází na získané paměťové adrese. Tak je úkol procedury vykonán a exekuce programu se vrací zpět do událostní procedury **btnTlačítko1_Click**. Finálním bodem je zobrazení okna se zprávou prostřednictvím funkce **MsgBox**. Pokud si myslíte, že zobrazena bude numerická hodnota 200, máte pravdu. Grafické znázornění příkladu můžete vidět na obr. 2.



Obr. 2 – Proces změny hodnoty proměnné **ABC**

Podstata techniky **aliasing** tkví v možnosti odkazovat se na jednu paměťovou adresu více názvy. Je tato podmínka splněna v předchozím příkladu? Patrně ano, protože na paměťové místo s hodnotou proměnné **ABC** je možné se odkázat i z procedury **Proc_1**.

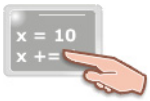
Povězme, že byste podobu této procedury pozměnili tímto způsobem:




```
Private Sub Proc_1(ByRef hodnota As Integer)
MsgBox hodnota
End Sub
```

Tak by se v prvním dialogovém okně zobrazila hodnota 100, což by byla právě aktuální hodnota proměnné **ABC**. Druhé dialogové okno by poté zobrazilo již modifikaci hodnoty proměnné. Z uvedeného lze vyvodit, že na jednu adresu v paměti je možné se odkázat jak pomocí proměnné **ABC**, tak zprostředkovaně pomocí proměnné **hodnota**.

2. **Remove Array Bounds Checks**
Volba odstraňuje implicitní kontrolu hranic rozsahu polí. Visual Basic 6.0 totiž při každé manipulaci s prvky pole zjišťuje, zdali je prvek, se kterým se právě pracuje, v povoleném intervalu, jenž byl zvolen při deklaraci pole. Jestliže vaše aplikace pracuje s poli „ve velkém", můžete zapnout tuto volbu a dosáhnout značného navýšení výkonu. Problém ovšem může nastat v situaci, kdy dojde (při zapnuté optimalizační volbě) k přístupu mimo povolené hranice pole. Takovýto pokus se může skončit až neočekávaným selháním aplikace. Dobrá rada na závěr: Budete-li chtít bezpečně procházet všemi prvky pole, použijte za tímto účelem funkce **LBound** a **UBound**. Příklad jejich použití můžete vidět v následujícím výpisu fragmentu zdrojového kódu:




```
Dim y As Integer
Dim pole() As Integer
ReDim pole(10000)

For y = LBound(pole()) To UBound(pole())
lstSeznam.AddItem (y)
Next y
```

Na prvním řádku je deklarována proměnná **y** datového typu **Integer**. Poté je deklarováno také dynamické pole (**pole()**), které je schopno pojmout zatím neurčený počet prvků typu **Integer**. Příkaz **ReDim** nastavuje vrchní hranici pole na hodnotu 10000 (pole od tohoto okamžiku obsahuje ve skutečnosti desettisíc a jeden prvek, protože jeho rozsah tvoří interval <0, 10000>). Dále chceme, aby byl obsah pole (soubor všech prvků pole) přenesen do seznamu s názvem **lstSeznam**. Protože nechceme explicitně určovat dolní a horní hranici pole, použijeme za tímto účelem spřízněné funkce **LBound** a **UBound** v cyklu **For-Next**, kterým jako argument předáme pouze název pole (**pole()**). Přidání prvků pole do seznamu zabezpečí metoda **AddItem**, které je v rámci cyklu vždy poskytnut konkrétní index prvku pole pro přidání.

Použijete-li na určení spodní a vrchní hranice pole uvedené funkce, vyhnete se jakýmkoliv potížím při použití optimalizační volby **Remove Array Bounds Checks**.

3. **Remove Integer Overflow Checks**
Zatržení této volby způsobí, že Visual Basic nebude kontrolovat, zdali byl rozsah specifického celočíselného datového typu překročen – jde o tzv. přetečení obsahů proměnných celočíselných datových typů. Při operacích s proměnnými tohoto typu si VB hlídá, aby výslední hodnota operace byla v oboru přípustných hodnot použitých datových typů. Jestliže je výslední hodnota mimo tohoto přípustného oboru, dochází k přetečení. Tuto chybu Visual Basic standardně zjistí a ohlásí, no pokud zapněte optimalizační volbu **Remove Integer Overflow Checks**, Visual Basic nebude popsanou kontrolu oboru proměnných provádět a tudíž v případě chyby neobdržíte žádné chybové hlášení, ovšem aplikace může zobrazovat nesprávné hodnoty.

4. **Remove Floating Point Error Checks**
   Budete-li chtít deaktivovat kontrolu chyb při práci s čísly s pohyblivou desetinnou čárkou, zatrhněte tuto optimalizační volbu. Podobně jako při předcházející volbě, i při operacích s proměnnými, jež uchovávají čísla s pohyblivou řádovou čárkou Visual Basic kontroluje, zdali finální hodnota operace nepřekročí přípustný rozsah použitého datového typu. Když však za běhu programu dojde k překročení stanoveného rozsahu, VB zobrazí chybové hlášení. Jestliže zatrhněte tuto volbu, VB se nebude dále zabývat implicitní kontrolou, a raději se bude věnovat jiným, podstatnějším operacím. Tato skutečnost sice může dodat vaší aplikaci potřebnou „výkonnostní injekci", ovšem zase byste měli pamatovat na to, že pokud volbu **Remove Floating Point Error Checks** zapnete a za běhu aplikace nastane chyba, nebude zobrazeno žádné varovné hlášení, no pochopitelně se můžou dostavit podivné výsledky výpočetních operací.

5. **Allow Unrounded Floating Point Operations**
   Předposlední optimalizační volba říká kompilátoru, aby při práci s čísly s pohyblivou desetinnou čárkou nezaokrouhloval tato čísla na předepsaný počet desetinných míst. Tím pádem sice přicházíte o jistou dávku přesnosti, no zvýší se rychlost prováděných operací.

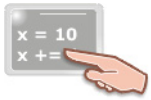
6. **Remove Safe Pentium(tm) FDIV Checks**
   Pokud nebude vaše aplikace běžet na počítačích s procesory třídy Pentium, jejichž takt je menší než 120 MHz, můžete zatrhnout tuto optimalizační volbu. Staré procesory Pentium obsahovaly chybu při dělení čísel s pohyblivou řádovou čárkou. Kompilátor Visual Basicu při generování kódu standardně přidává také instrukce, které zabraňují výskytu chyby při provádění matematických operací s čísly s pohyblivou desetinnou čárkou i na těchto starých procesorech. Když jste si naprosto jisti, že vaše aplikace nebude nikdy běžet na uvedeném typu procesoru, můžete aktivovat tuto optimalizační volbu. Výsledkem bude rychlejší exekuce programového kódu, protože operace budou vykonávány bez dodatečných korekčních instrukcí.

Měření času provádění kritických částí programu

V rámci procesu optimalizace byste měli být schopni podívat se na aplikaci, jejíž kód chcete optimalizovat, ze širšího úhlu pohledu. Velmi dobrou pomůckou je rozčlenění celé aplikace do menších logických jednotek, v prostředí kterých se pak odehrává důkladná analýza problémových sekcí. Mnoho programátorů začíná proces optimalizace velice jednoduše: Otevřou si kód aplikace a optimalizují jej tak řečeno „odshora dolů". Po kompletní optimalizaci padnou vyčerpaní na stůl a nakonec zjistí, že obětovaný čas a energie se jaksi minuly svým účinkem. Ptáte se, proč je nastíněný přístup nevhodný? Proces optimalizace aplikace musí být samozřejmě komplexní, ovšem v praxi se velmi často stává, že výkon aplikace se znatelně zvýši již tehdy, kdy je optimalizováno jenom dvacet procent celkového objemu zdrojového kódu. Právě rychlost běhu této „dvacetiprocentní" množiny kritických programových instrukcí má největší vliv na celkovou výkonnost aplikace. Jestliže je programátor natolik zkušený, aby věděl odhalit vzpomínanou kritickou množinu programových instrukcí pro optimalizaci, má již napůl vyhráno. Mezi nejčastější problémové oblasti patří často prováděné operace, jejichž charakter závisí od povahy aplikace. Když kupříkladu vyvíjíte textový procesor, vaše hlavní kritická oblast bude tvořena instrukcemi pro práci s proudy textových znaků (otevíraní, modifikace, ukládání). Jste-li vývojářem programu pro zpracování rastrové grafiky, jistě mnoho času obětujete pro to, aby překreslování oken s grafickými soubory bylo co možná nejrychlejší. Takto bychom mohli postupovat dále a uvádět nespočet podobných příkladů.

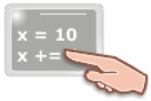
Předpokládejme v tuto chvíli, že jste již určili kritické oblasti vaší aplikace a chtěli byste změřit rychlost jejich provádění. Měření rychlosti různých úseků je ve skutečnosti základním krokem k získaní kvantitativních primárních údajů o výkonu vaší aplikace. Na nadcházející ukázce programového kódu si ukážeme, jak změřit rychlost provádění specifické části aplikace.

Povězme, že chceme přidat do seznamu 10000 položek. Tento úkol má na starosti událostní procedura **Click** tlačítka, jejíž tělo tvoří následující příkazy:



```
Dim x As Integer

For x = 1 To 10000
lstSeznam.AddItem (x)
Next x
```

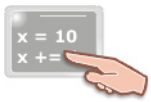
Domnívám se, že kód je pro vás dostatečně srozumitelný, a proto se nebudu jeho popisem zabývat. Místo toho vám raději ihned ukážu způsob, jak změřit, kolik trvá exekuce procedury:



```
Dim x As Integer, začátek As Single

začátek = Timer

For x = 1 To 10000
lstSeznam.AddItem (x)
Next x

Me.Caption = "Operace trvala: " & _
Format(Timer - začátek, "0.000") & _
" sekund."
```

Celý algoritmus měření tempa procedury je následovní:

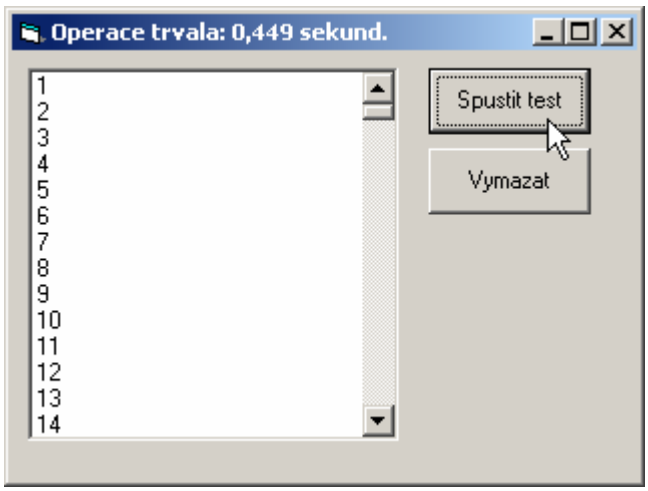
1. Do proměnné **začátek** typu **Single** je uložen počet sekund, které uplynuly od půlnoci (tuto hodnotu uchovává vlastnost **Timer** třídy **DateTime**, která je určena jenom pro čtení).
2. Za řádky programového kódu, jejichž rychlost chceme změřit, umístíme příkaz pro zobrazení vzniklého časového rozdílu od spuštění procedury (přesněji od inicializace proměnné **začátek**). Příkaz má tuto podobu:



```
Me.Caption = "Operace trvala: " & _
Format(Timer - začátek, "0.000") & _
" sekund."
```

Nejpodstatnější částí je výpočet časového rozdílu (**Timer – začátek**). Tak získáváte číselné vyjádření doby, která uplynula od inicializace počáteční proměnné. V našem příkladě je výpočet časového rozdílu ještě „uživatelsky zaobalen" do funkce **Format**. Pomocí přiměřeného nastavení druhého parametru funkce určíme, aby desetinná část zobrazené hodnoty byla tvořena třemi číslicemi. Finální hodnota je zobrazena v titulkovém pruhu okna aplikace.

Na počítači s procesorem Celeron taktovaném na 1,1 GHz trvalo naplnění seznamu přibližně 450 tisícin sekundy (obr. 3).



Obr. 3 – Výsledek měření tempa událostní procedury

Výslední hodnoty se můžou při opakovaném testu nepatrně lišit, proto můžete dodatečně implementovat kód pro provedení, dejme tomu, pěti testů a následně vypočíst průměrnou hodnotu ze všech testovacích hodnot.

Integrované prostředí rubriky Visual Basic	Autor: **Ján Hanák**

Začínáme s VB .NET

Úvod do světa .NET (6. díl)

Časová náročnost (min):	Začátečník	Pokročilý	Profesionál
90			

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

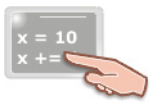
Milí čtenáři,

v šestém dílu seriálu se budeme zabývat problematikou proměnných hodnotových a referenčních datových typů. Dozvíte se, jak deklarovat více proměnných v rámci jednoho deklaračního příkazu a rovněž objevíte způsob, jakým se uskutečňuje přiřazovaní hodnot u proměnných hodnotového datového typu a přiřazování referencí u referenčních proměnných.

Obsah

Co byste ještě měli vědět o deklaraci a inicializaci proměnných hodnotových datových typů

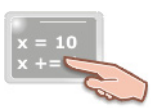
Visual Basic .NET vám dává příležitost, abyste deklaraci proměnné a přiřazení hodnoty do této proměnné spojili do jednoho celku použitím jediného příkazu, v němž se provede jak deklarace, tak i inicializace proměnné. Podívejte se na následující ukázku:




```
Dim Objednávka As Integer = 1000
```

Zde můžete vidět, jak je proměnná **Objednávka** datového typu **Integer** ihned při deklaraci naplněna numerickou hodnotou.

Když budete chtít deklarovat více proměnných stejného datového typu v rámci jediného deklaračního příkazu, můžete postupovat tak, jak je uvedeno na doprovodném výpisu zdrojového kódu:




```
Dim hodnota1, hodnota2 As Long
```

V jednom deklaračním příkazu jsou deklarovány dvě proměnné (**hodnota1** a **hodnota2**) datového typu **Long**.

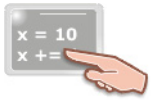
Pokud patříte mezi programátory, kteří přicházejí z VB 6.0, bude dobré, když si všimnete, že deklarace více proměnných v jednom deklaračním příkazu se od minulé verze podstatně změnila. Ve VB 6.0 by deklarace tohoto typu znamenala, že proměnná **hodnota1** je typu **Variant** a až proměnná **hodnota2** je typu **Long** (přitom abstrahujeme od skutečnosti, že rozsah datového typu **Long** ve VB .NET je jiný ve srovnaní s VB 6.0).

Pokud si prohlížíte zdrojový kód a chcete zjistit, jakého datového typu je ta-která proměnná, umístěte nad název požadované proměnné kurzor myši a chvíli posečkejte:

```
hodnota1 = x \ (y * (-z))
```

Dim hodnota1 As Long

Pro deklaraci dvou proměnných typu **Single** a dalších dvou proměnných typu **Double** můžete použít tento zápis deklaračního příkazu:

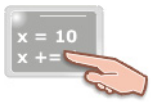



```
Dim průměr1, průměr2 As Single, výsledek1, výsledek2 As Double
```

Algoritmus přiřazování hodnot u proměnných hodnotového typu

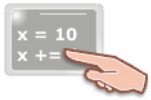
V programátorské praxi se najde nespočet případů, kdy budete muset přiřadit obsah jedné proměnné do jiné proměnné. Abyste pochopili, jak je tento proces realizován u proměnných hodnotových datových typů, rozebereme si jej podrobněji.

Předpokládejme, že se nám do rukou dostane tento útržek programového kódu:




```
Dim v1, v2 As Short
v1 = 100
v2 = v1 + 100
MessageBox.Show(v2.ToString)
```

Víte, jaká hodnota bude zobrazena v okně se zprávou? Abychom záhadu rozřešili, pojďme se podívat na to, jak se věci mají. Nuže, na prvním řádku jsou deklarovány dvě proměnné (**v1** a **v2**) typu **Short**. V dalším kroku je proměnná **v1** inicializována hodnotou 100. Pravděpodobně nejzajímavější je třetí příkaz:




```
v2 = v1 + 100
```

Co se stane? Nejprve je hodnota proměnná **v1** zvětšena o 100 jednotek (programátoři také často říkají, že hodnota proměnné je inkrementována). Jestliže před provedením inkrementace byla hodnota proměnná **v1** rovna 100 jednotkám, po zvýšení hodnoty bude současná hodnota proměnná **v1** rovna 200 jednotkám. Následně dochází k přiřazení inkrementované hodnoty proměnné **v1** do proměnné **v2**. Od nynějška tak proměnná **v2** obsahuje číselnou hodnotu 200. Tato skutečnost je záhy uživateli oznámena v okně se zprávou.

Celý algoritmus si můžete ještě jednou prohlédnout na doprovodné ilustraci (obr. 1).

Obr. 1 – Algoritmus procesu přiřazení hodnoty proměnné **v1** do proměnné **v2**

Velmi důležité je mít na paměti, že demonstrovaná situace se týká jenom proměnných hodnotových datových typů. Při přiřazování odkazů mezi proměnnými referenčních datových typů nabírá problematika zcela jiné rozměry.

Jaká je hodnota deklarované proměnné, která ještě nebyla inicializována?

Při práci s proměnnými hodnotových datových typů by vás mohla napadnout zajímavá otázka, a sice jakou hodnotu obsahuje proměnná, která již byla deklarována, no ještě nebyla inicializována. Kdybyste chtěli a deklarovali proměnnou **x** datového typu **Integer** a posléze zobrazili její hodnotu v okně se zprávou, zjistili byste, že proměnná **x** je standardně inicializována na nulovou hodnotu.

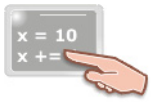
Pokud není proměnná po své deklaraci inicializována, VB .NET této proměnné přiřadí standardní inicializační hodnotu. Touto standardní inicializační hodnotou je kupříkladu u všech numerických datových typů nula, u datového typu **Boolean** je to pravdivostní hodnota **False** a u všech proměnných referenčních datových typů (jako je **Object**, **String** nebo pole) speciální hodnota **Nothing**.

Ukázka práce proměnné referenčního datového typu

Proměnné referenčních datových typů, na rozdíl od proměnných hodnotových datových typů, neuchovávají jistou hodnotu, nýbrž odkaz (referenci) na paměťové místo, kde je hodnota uložena (onou hodnotou může být i objekt, jak za chvíli uvidíte). Typickým představitelem referenčního datového typu je **třída**. Třídu si můžete představit jako šablonu pro tvorbu objektů jistého typu.

Proces deklarace a následné inicializace proměnné referenčního typu si předvedeme na bázové třídě formuláře (**Form**).

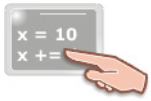
Deklarujme nejprve proměnnou **formulář**, datovým typem které bude třída **Form**:



```
Dim formulář As Form
```

Co byste si měli zcela jistě zapamatovat je skutečnost, že uvedený řádek programového kódu ještě nevytváří instanci třídy **Form** (tedy nový objekt), ale jenom říká překladači, aby vyhradil dostatečné paměťové místo pro uložení referenční proměnné **formulář**.

Nový objekt vzniká až zapsáním tohoto řádku kódu:

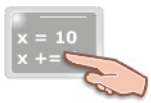


```
formulář = New Form()
```

Použitím klíčového slova **New** dochází k vytvoření nového objektu, tedy nové instance třídy **Form**. V této souvislosti je nevyhnutné rozlišovat dvě různé věci, kterými jsou proměnná **formulář** a **objekt třídy Form**. Proměnná **formulář** je proměnnou referenčního typu a z tohoto titulu je schopna uchovávat jenom odkaz, resp. referenci na „svůj" objekt. Řečeno jinými slovy, proměnná **formulář** neví nic o datech, které objekt spravuje, ona vám může poskytnout jenom přístup k paměťové adrese, na které se objekt právě nachází. Na druhé straně lze pozorovat **objekt**, tedy samotný formulář, který disponuje svými vlastnostmi a metodami a umí pružně reagovat na množinu podporovaných událostí.

Ačkoliv je možné říci, že název vzniklého objektu je **formulář**, bude dobré, když budete vědět, jak věci ve skutečnosti fungují. Textový řetězec „**formulář**" představuje název objektové (referenční) proměnné, ovšem protože k vytvořenému objektu nelze přistupovat jinak, nežli právě pomocí této proměnné referenčního typu, dochází k uvedené slovní substituci. Po zrození je nový objekt umístěn do speciálního paměťového prostoru (na hromadu) a je možné s ním dále interaktivně komunikovat.

Budete-li kupříkladu chtít, aby se vytvořený formulář objevil na obrazovce, zavolejte jeho metodu **Show**. Text v titulkovém pruhu formuláře upravíte vhodným nastavením jeho vlastnosti **Text**.



```
formulář.Show()
formulář.Text = "Toto je nový formulář."
```

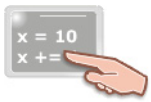
Podobu vygenerovaného formuláře ilustruje obr. 2.

Obr. 2 – Formulář, jenž vznikl z bázové třídy **Form**

Jak si můžete všimnout, zobrazený formulář má všechny potřebné součásti, které musí řádný formulář aplikace Windows mít: titulkový pruh, systémovou nabídku, okraje, aktivní plochu a také tlačítka pro minimalizaci, maximalizaci a zavření. Dokonce můžete měnit velikost a pozici formuláře na monitoru. Jak je možné, že formulář obsahuje všechny tyto prvky bez toho, abyste jejich existenci výslovně přikázali? Odpovědí na položenou otázku je definice jednoho z esenciálních pilířů objektově orientovaného programování, kterým je **dědičnost**.

Tento termín, tak populární u programátorů, kteří používají OOP, znamená, že odvozený (synovský) formulář si ponechává všechny vlastnosti svého předka, kterým je bázová třída **Form**. Také se říká, že vzniklý potomek dědí všechny požadované náležitosti svého předka. Jak jsme si již pověděli, třída slouží jako podklad pro tvorbu objektů jistého typu. Když je tedy v definici třídy výslovně uvedeno, že objekt, který z ní vznikne, má disponovat předem danými charakteristikami, zrozený objekt taký skutečně bude. Stejnou situaci využívá také nový formulář, který dostal od své bázové třídy potřebnou výbavu v podobě titulkového pruhu, systémové nabídky a dalších potřebných součástí.

Podobně jako u proměnných hodnotových typů, i při použití referenčních proměnných lze spojit kroky pro deklaraci a inicializaci do jednoho příkazu:




```
Dim formulář As New Form()
```

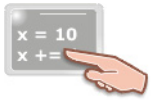
Na závěr této podkapitoly si ještě povězme, že kromě tříd můžeme do množiny referenčních datových typů zařadit také:

- Datový typ **String**
- Pole
- Delegáty
- Rozhraní

Algoritmus přiřazování referencí u proměnných referenčního typu

Námět této části bude analogický s podkapitolou, ve které jsme zkoumali způsob přiřazování hodnot u proměnných hodnotového typu. Předem vám již ale musím sdělit skutečnost, že problematika přiřazování odkazů je u referenčních proměnných podstatně odlišná, co si ostatně za chvíli ukážeme na praktickém příkladu.

Podívejme se nejdřív na zdrojový kód, jenž bude předmětem našeho zájmu:




```
Dim formulář1 As Form
formulář1 = New Form()
formulář1.Text = "Toto je nový formulář."

Dim formulář2 As Form
formulář2 = formulář1

MessageBox.Show(formulář2.Text)
```

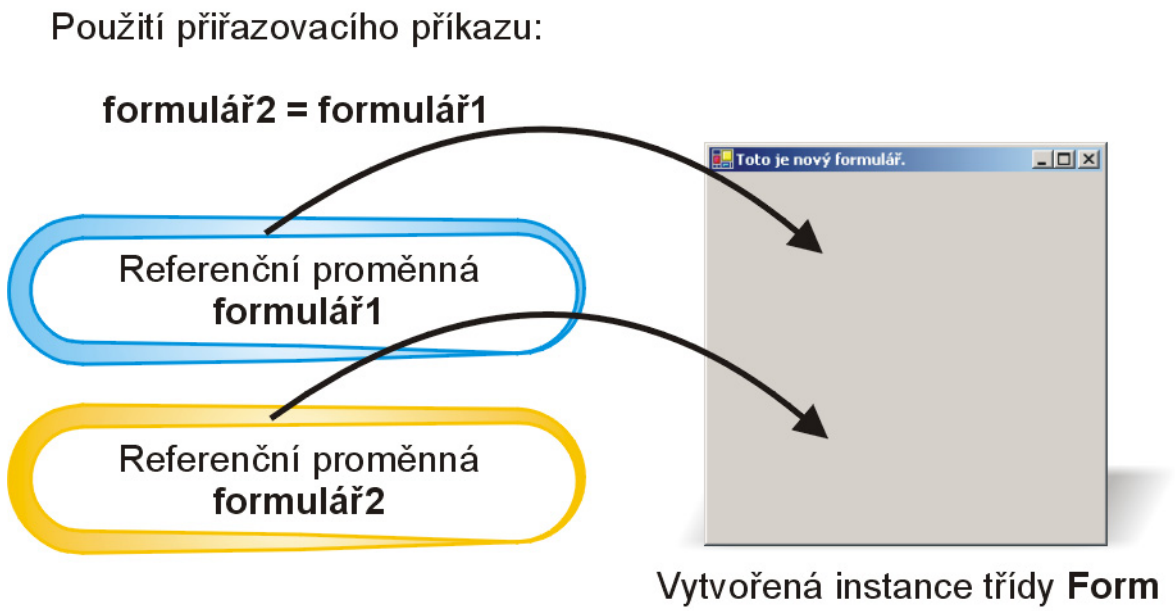
A opět se vás zeptám, zdali víte, co bude zobrazeno v okně se zprávou. Pokud jste na pochybách, věnujte prosím pozornost následujícím řádkům textu.

Smysl dvou úvodních řádků vám již nemusím blíže vysvětlovat, jednoduše dochází k deklaraci proměnné **formulář1** a vytvoření objektu třídy **Form**. Odkaz na objekt je vložen do referenční proměnné **formulář1**. Dále je modifikována vlastnost **Text** právě zrozeného objektu. V následujícím kroku je deklarována další objektová proměnná s názvem **formulář2**. Konečně se dostáváme k hlavnímu prvku programové ukázky, kterým je aplikace přiřazovacího příkazu. Ptáte se, jak probíhá přiřazení?

Vzpomeňte si na jeden z předešlých příkladů, ve kterém jste se pokoušeli přiřadit hodnotu jedné hodnotové proměnné do jiné hodnotové proměnné. Tehdy se obsah proměnné, která se nacházela na pravé straně přiřazovacího příkazu, vložil do proměnné, jež byla umístěna na levé straně přiřazovacího příkazu.

Podobná je situace i v tomto případě, jenom nesmíte zapomenout na to, že objektová proměnná **formulář1** neobsahuje žádnou hodnotu, ale referenci na vytvořený objekt. V přiřazovacím příkazu je proto odkaz na objekt, jenž uchovává proměnná **formulář1** překopírován do objektové proměnné **formulář2**. To znamená, že od této chvíle je také proměnná **formulář2** schopna přistupovat k vlastnostem vytvořeného objektu (formuláře), a tedy v dialogovém okně bude zobrazena hodnota vlastnosti **Text** tohoto objektu (tedy textový řetězec „Toto je nový formulář“).

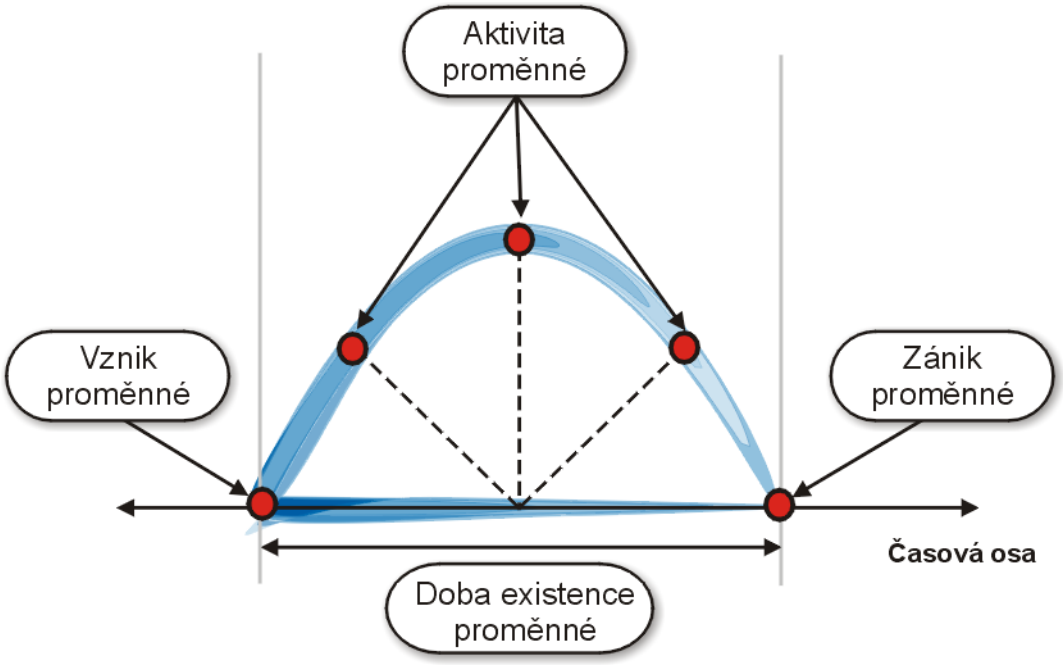
Závěrem lze poznamenat, že výsledkem aplikace přiřazovacího příkazu je stejná náplň obou objektových proměnných (obr. 3).



Obr. 3 – Po přiřazení obsahuje i objektová proměnná **formulář2** odkaz na instanci třídy **Form**

Životní cyklus proměnné

Každá proměnná, či už jde o proměnnou hodnotového nebo referenčního datového typu, má svůj vlastní životní cyklus. Životním cyklem proměnné se rozumí jistá časem ohraničená doba, po kterou proměnná existuje v paměti počítače. Graficky lze průběh „života" proměnné znázornit asi takto:



Obr. 4 – Grafická ilustrace životního cyklu proměnné

Proměnná vzniká deklarací pomocí deklaračního příkazu **Dim**. Příkaz **Dim** rezervuje v paměti počítače dostatečně velký prostor v závislosti od datového typu proměnné, aby bylo později možné uložit do proměnné hodnotu nebo referenci na objekt. Po počátečné deklaraci dochází buď ihned, nebo za nějakou dobu k naplnění proměnné hodnotou nebo odkazem, jinak řečeno, dochází k inicializaci proměnné. Jakmile je proměnná inicializována, je záhy použita k provedení výpočtu, přiřazení, nebo

jakékoliv jiné operaci v souladu s programovou logikou aplikace. V rámci svého životního cyklu může být hodnota proměnné nespočetněkrát změněna, ovšem proměnná nemůže nikdy měnit svůj datový typ. Když je proměnná jednou vytvořena pro přijímaní celých čísel v rozsahu od 0 do 255 možných hodnot, tato charakteristika je jí pevně daná a nelze ji nijak modifikovat. Ve chvíli, kdy proměnná splní svůj úkol, dochází k její destrukci. V procesu destrukce je samozřejmě zlikvidována taká hodnota, resp. reference, kterou proměnná obsahovala. Samotný proces destrukce je ovšem u proměnných hodnotových datových typů jiný, nežli je tomu u referenčních proměnných.

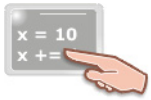
Při destrukci proměnné hodnotového typu je situace taková, jakou byste nejspíš čekali. Nejprve je zlikvidována hodnota proměnné a poté nastává destrukce proměnné. Při proměnných referenčních typů je ovšem proces destrukce specifický. Při likvidaci proměnné tohoto typu dojde k likvidaci reference, tedy odkazu, jenž proměnná obsahovala. Instance třídy (objekt), na kterou byl odkaz referenční proměnné zaměřen ale zůstává „naživu". Objekt se stává předmětem likvidace (pro **Garbage Collection**) až v okamžiku, kdy na něj nejsou navázány žádné další odkazy.

Délka životního cyklu proměnné závisí hlavně od oboru proměnné (tedy od času, v rámci kterého si proměnná udržuje svoji hodnotu). Oborem proměnné zvyčejně bývá:

- Procedura typu **Sub**
- Funkce
- Třída formuláře (případně jiná třída)
- Modul

Jestliže je proměnná deklarována uvnitř procedury typu **Sub** (např. uvnitř událostní procedury **Click** tlačítka) jejím oborem je právě tato procedura. To mimo jiné znamená, že jakmile se dokončí provádění kódu procedury (procesor přejde na řádek s příkazem **End Sub**), proměnná je podrobena likvidaci a je zničena také hodnota, kterou proměnná uchovávala. Takováto proměnná je často nazývaná i jako **lokální** proměnná, čímž se ještě více zdůrazňuje skutečnost, že působností proměnné je jenom jistá lokální oblast.

V některých případech možná budete chtít, aby proměnná nebyla po ukončení procedury či funkce zlikvidována a aby bylo možné zachovat její hodnotu. Tehdy jistě oceníte sílu klíčového slova **Static**, pomocí kterého se deklarují proměnné, které si dokáží uchovat svou hodnotu i po terminaci procedury, v níž jsou deklarovány. Uveďme si praktický příklad použití klíčového slova **Static**.




```
Private Sub btnTlačítko1_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles btnTlačítko1.Click
    Static počet_aktivací As Integer

    počet_aktivací = počet_aktivací + 1

    MessageBox.Show("Procedura byla aktivována: " & _
    počet_aktivací.ToString & "-krát.", _
    "Počet aktivací procedury Sub")
End Sub
```

Vždy, když budete chtít deklarovat proměnnou jako statickou, budete muset před její název umístit klíčové slovo **Static**.

Statickou proměnnou můžete deklarovat i s použitím příkazu **Dim**:

```
Static Dim počet_aktivací As Integer
```

Ačkoliv jde o platnou deklaraci, je poměrně zdlouhavá, a proto se velmi často statické proměnné deklarují pouze zapsáním klíčového slova **Static** před jméno proměnné.

Když budeme předpokládat, že na formuláři se nachází tlačítko s názvem **btnTlačítko1**, tak pokaždé, když uživatel klepne na tlačítko, zobrazí se počet aktivací událostní procedury **Click** tlačítka v okně se zprávou. Statická proměnná **počet_aktivací** si i po ukončení exekuce kódu událostní procedury svou hodnotu uloží a použije ji při příštím volání procedury.

Pro lepší efektivitu kódu můžete následující přiřazovací příkaz:

```
počet_aktivací = počet_aktivací + 1
```

nahradit příkazem:

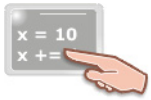
```
počet_aktivací += 1
```

Dobrá, řekli jsme si, že proměnné deklarované pomocí klíčového slova **Static** mají delší životní cyklus ve srovnání se svými „nestatickými" protějšky. Nicméně otázkou zůstává, kdy vlastně dojde k likvidaci statické proměnné. Při pokusu o odpověď se musíme ponořit hlouběji do problematiky teorie proměnných. Jestliže je totiž statická proměnná deklarována uvnitř procedury jisté třídy, délka jejího životního cyklu závisí od toho, zdali je ona procedura deklarována použitím klíčového slova **Shared** či nikoliv. Kdyby tedy procedura obsahovala ve své deklaraci také toto kouzelné slovíčko, statická proměnná by existovala po celou dobu běhu aplikace. Protože ale událostní procedura **btnTlačítko1_Click** není deklarována pomocí klíčového slova **Shared**, životní cyklus statické proměnné se kryje s životním cyklem instance třídy, uvnitř které se událostní procedura se statickou proměnnou nachází.

Na závěr ještě vzpomeňme i použití takzvaných **globálních** proměnných, kterých deklarace, obsahující klíčové slovo **Public**, se zpravidla ukládají do modulu, jehož kód je uložen v speciálním souboru. Jak již název implikuje, globální proměnné jsou „viditelné" z celé aplikace (můžete je použít v kterékoliv proceduře nebo funkci) a své hodnoty si udržují až do konce běhu aplikace.

Globální proměnné se často označují také jako **veřejné** proměnné.

Přidejte do vašeho řešení soubor s modulem a deklarujte globální proměnnou s názvem **global**:




```
Module Module1
    Public global As Integer
End Module
```

Tuto proměnnou můžete použít z kteréhokoliv místa aplikace, třeba z událostní procedury **Click** tlačítka.

Integrované prostředí rubriky Visual Basic	Autor: Ján Hanák

Programátorská laboratoř

Prostor pro experimentování

Časová náročnost (min):	Začátečník	Pokročilý	Profesionál
65			

Použitý operační systém :	Win2000 SP3, Windows XP
Hlavní vývojový nástroj :	Visual Basic .NET 2002
Další vývojový software :	Žádný
Jiný software :	Žádný

Spouštění a ukončování procesů pomocí komponenty Process

V úvodním tipu si ukážeme, jak lze z prostředí VB .NET uskutečňovat „management" jednotlivých procesů. Nejprve si vysvětlíme, co se ve skutečnosti rozumí pod pojmem **proces**, a poté přejdeme k praktické ukázce pomocí komponenty **Process**, která se nachází na panelu **Components** hlavní soupravy nástrojů (**Toolbox**).

Termínem **proces** je ve světě počítačů označována běžící aplikace. Pokud tedy uživatel poklepe na ikonu spustitelného (.EXE) souboru, dochází ke spuštění programu (data programu se načítají do paměti počítače) a program se rozběhne. V tomto okamžiku již mluvíme o procesu. Jeden proces pak může pozůstávat z jednoho, nebo i několika vláken, ovšem do této problematiky se teď nebudeme pouštět. Důležité je, abyste věděli, že výraz proces je synonymem pro aplikaci, která se právě nachází v paměti počítače, a kód které je vykonáván pomocí procesoru.

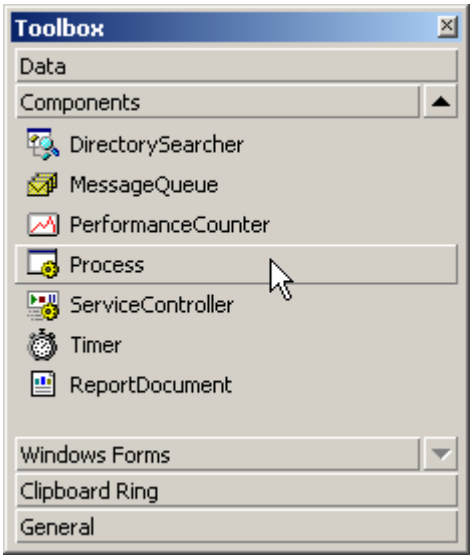
Proces ovšem není jenom teoreticky definována entita, jde o skutečný softwarový prvek, jenž disponuje mnoha speciálními charakteristikami. Aby bylo vůbec možné se na proces odkazovat, je potřebné znát jeho jméno (případně identifikační kód). Všechny procesy mají jistou prioritu, alokují jistý paměťový prostor atd.

Zajímavé je zkoumání samotné inicializace procesu, tedy okamžiku, v kterém je proces spuštěn a následně podroben přímé exekuci. Tím pomyslným iniciátorem může být buď sám uživatel (když poklepe na ikonu spustitelného souboru, nebo spuštění souboru aktivuje jiným způsobem) anebo

programátor (prostřednictvím zdrojového kódu lze startovat, spravovat a také ukončovat požadované procesy). Tak se pomalu, ale jistě dostáváme k hlavní podstatě procesů, a sice k jejich správě. VB .NET přichází s novou komponentou s názvem **Process**, která vám poskytuje tu nejsnazší cestu, jak startovat a rovněž ukončovat nové procesy.

V následující ukázce si předvedeme, jak snadné je použitím komponenty **Process** nastartovat libovolnou programovou aplikaci. Postupujte dle uvedených instrukcí:

1. Ve Visual Basicu .NET vytvořte novou aplikaci typu **Windows Application**.
2. V soupravě nástrojů (**Toolbox**) klepněte na záložku panelu **Components** (obr. 1).

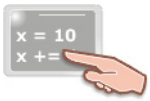


Obr. 1 – Výběr komponenty **Process** v soupravě nástrojů

3. Poklepejte na ikonu komponenty **Process**, na což se komponenta umístí na podnos komponent (obr. 2).

Obr. 2 – Automatické umístění komponenty **Process** na podnos komponent

4. Přidaná instance komponenty je standardně pojmenovaná (jako **Process1**), no pokud budete chtít, můžete její jméno změnit.
5. Přidejte na formulář jednu instanci ovládacího prvku **Button**, poklepejte na ní a do editoru pro zápis programového kódu zadejte tyto řádky kódu (komentáře psát samozřejmě nemusíte):




```
Process1.StartInfo.FileName = "calc.exe"
'Určení názvu aplikace, která se má spustit.

Process1.Start()
'Spuštění aplikace "Kalkulačka".

System.Threading.Thread.Sleep(2000)
'Uspání běhu programu VB .NET na dvě sekundy.
```

Programový kód pokračuje na následující straně

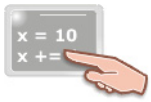
```
System.Windows.Forms.Application.DoEvents()
'Předání řízení operačnímu systému.

Process1.CloseMainWindow()
'Ukončení běhu aplikace "Kalkulačka".
```

První věcí, kterou potřebujete udělat, je určit, která aplikace se má později spustit. Z tohoto důvodu použijeme vlastnost **FileName** třídy **ProcessStartInfo**, do které přiřadíme textový řetězec, jenž reprezentuje název aplikace pro spuštění (v našem případě jde o aplikaci „Kalkulačka"). Dále pokračujeme zavoláním metody **Start**, pomocí které se zvolená aplikace rozběhne. V následující etapě je aplikace Visual Basicu .NET uspána na dvě sekundy. Toto znehybnění aplikace nám umožňuje metoda **Sleep** třídy **Thread** z jmenného prostoru **System.Threading**. Předposledním krokem je předání řízení systému a konečně ukončení spuštěného procesu použitím metody **CloseMainWindow**.

Použití metody **CloseMainWindow** je doporučováno pro řádné ukončení procesu, který má na obrazovce vizuální reprezentaci (např. aplikace „Kalkulačka" má hlavní okno, podobně i ostatní součásti grafického uživatelského rozhraní). Další výhodou metody **CloseMainWindow** je to, že ve fázi ukončování procesu předá řízení samotnému procesu, který tak má možnost se bezproblémově ukončit. Kdybyste např. spustili textový editor Word a následně ukončili jeho činnost, řízení procesu při jeho terminaci by bylo svěřeno přímo tomuto procesu, tedy aplikaci Word, která by se vás mohla kupříkladu zeptat, zdali nechcete uložit změny, které byly v aktivním dokumentu provedeny. Pro úplnost je ovšem potřebné dodat, že procesy lze ukončit i jiným způsobem, který spočívá v použití metody **Kill**. Aplikace této metody je vhodná při finalizaci procesů, které nemají grafickou reprezentaci. Metoda **Kill** se ovšem od metody **CloseMainWindow** liší zásadním znakem – když ji použijete, přinutíte předmětný proces, aby se ukončil, a to i za cenu ztráty dat. Metoda **Kill** tedy poskytuje možnost, jak abnormálně ukončit práci procesu, proto ji používejte jen v opravdu odůvodněných případech.

Aktivace poštovního klienta z aplikace VB .NET

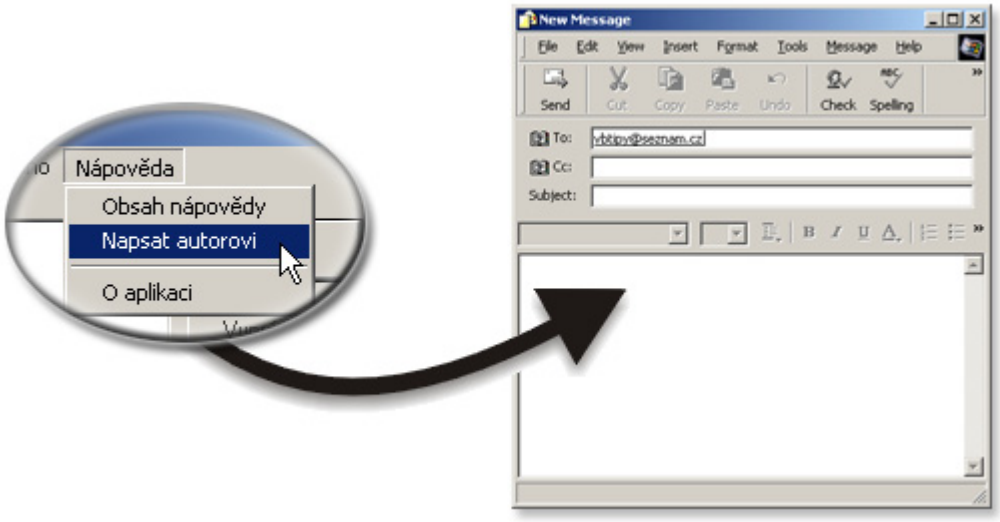
Po přečtení tohoto triku pro vás již nebude problém aktivovat z aplikace VB .NET poštovního klienta a zobrazit okno s novou zprávou elektronické pošty. Pro vyřešení stanoveného úkolu vám bude stačit jeden řádek programového kódu:




```
Process.Start("mailto:vbtipy@seznam.cz")
```

Jednoduše použijeme metodu **Start** třídy **Process** pro nastartování nového procesu. Uvedené metodě předáme textový řetězec, který specifikuje adresu elektronické pošty. VB .NET je natolik předvídavý, že pochopí, co se od něj žádá, i když jsme nezadali název programu, který slouží jako poštovní klient (ve skutečnosti si Visual Basic .NET tuto informaci zjistí prozíravě sám).

Příkaz pro aktivaci poštovního klienta a zobrazení okna s novou zprávou elektronické pošty můžete vhodně umístit do nabídky **Nápověda** ve vaší aplikaci (obr. 3).



Obr. 3 – Ukázka propojení nabídky **Nápověda** s aktivací poštovního klienta

Ukázka vizuálního dědění formulářů pomocí utility Inheritance Picker

Jestliže potřebujete použít jeden a tentýž formulář jako vizuální a funkční předlohu pro jiné, odvozené formuláře, jistě přijdete rychle na chuť programovacímu procesu, jehož jméno je dědění. Po pravdě řečeno, pojem dědění je poněkud rozsáhlejší, nám ovšem bude stačit, když se prozatím podíváme na takzvané vizuální dědění formulářů prostřednictvím chytré utility, která se jmenuje **Inheritance Picker**. A jak celý proces dědění formulářů probíhá?

Především je zapotřebí vytvořit základní formulář, kterému se někdy říká také bázový formulář, neboli formulář, jenž je postaven na bázové třídě. Takový bázový formulář vytvoříte přesně tak, jak to zvyčejně děláváte u jakéhokoliv jiného formuláře. Jednoduše na formulář umístíte všechny potřebné instance ovládacích prvků a komponent, které řádně pojmenuje a vhodným způsobem také upravíte jejich vlastnosti. Když máte bázový formulář hotový, řešení zkompilujete. A větší část práce je tím pádem vykonána.

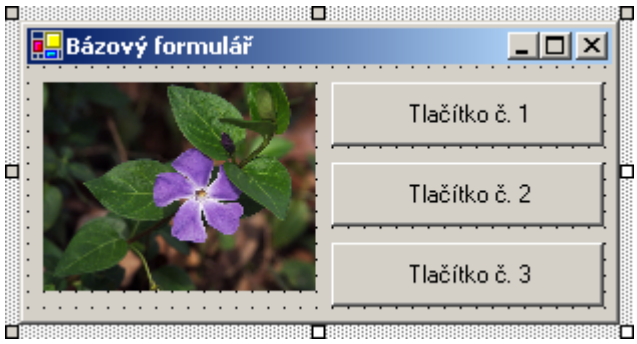
Nyní předpokládejme, že budete chtít do projektu vložit formulář, který bude potomkem formuláře, jenž byl vytvořen předtím. Tento nový formulář je odvozeným formulářem a v hantýrce objektově orientovaného programování se říká, že odvozený formulář dědí všechny vlastnosti, metody a veškerou funkcionalitu z rodičovského (bázového) formuláře. Dostáváme tak přesnou kopii bázového formuláře. Nejlepší na tom všem je ale to, že okamžitě získáváte nový formulář bez veškeré další práce (není tedy nutné, abyste do řešení manuálně přidávali další formulář a ten poté pracně vyplňovali instancemi ovládacích prvků). Odvozený formulář ovšem není nedotknutelný, právě naopak! Můžete dále rozšiřovat jeho vizuální i programovou funkcionalitu, podle toho, co budete právě potřebovat.

Přesto ovšem nezapomínejte, že předmětem tohoto cvičení je ukázka vizuální dědičnosti formuláře (právě pomocí utility **Inheritance Picker**). Dědění je samozřejmě možné také z prostředí programového kódu, no tento proces je poněkud komplikovanější a teď se ním nebudeme zabývat.

V následující ukázce si předvedeme, jakým stylem se implementuje vizuální dědění formuláře v praxi. Zde je postup:

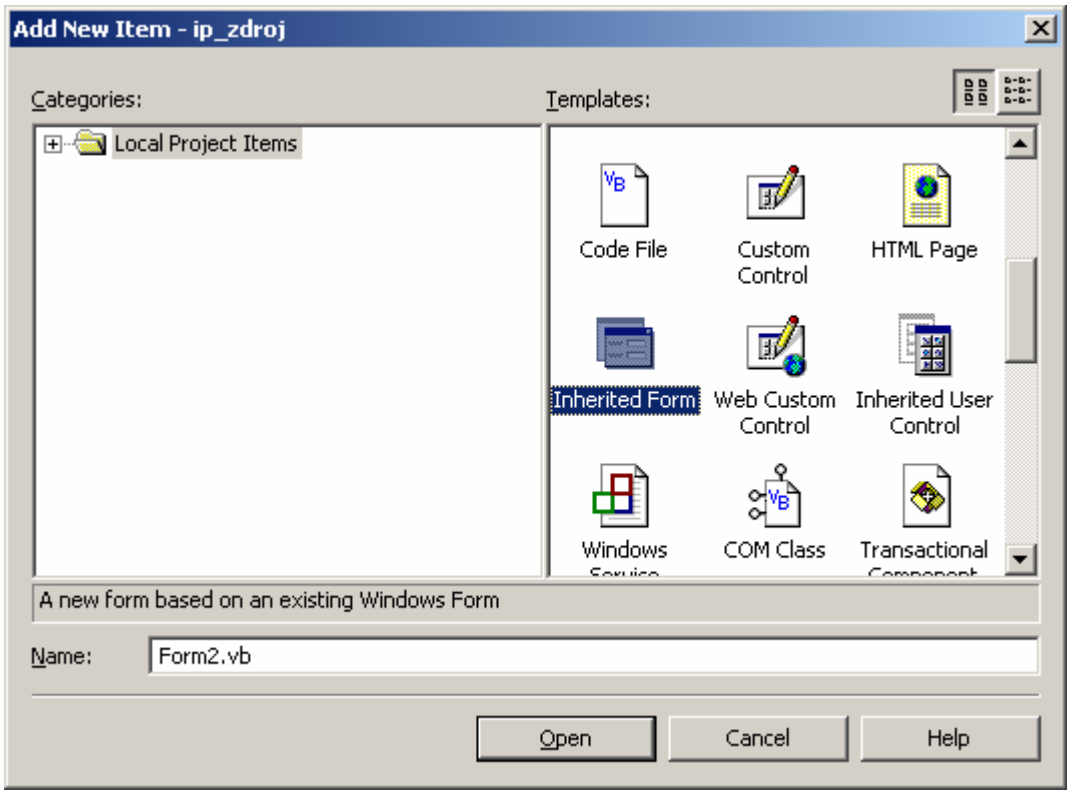
1. Jestliže jste tak ještě neučinili, spusťte Visual Basic .NET a vytvořte novou aplikaci typu **Windows Application**.
2. V této chvíli se dohodneme, že formulář **Form1** se od nynějška stane bázovým formulářem, který tak bude tvořit základní vizuální kostru pro odvozené formuláře, neboli formuláře vzniklé děděním. Budete-li chtít, můžete upravit formulář do vámi požadované podoby (např. můžete vytvořit bázový formulář pro dialogové okno s možnostmi, nebo okno s informacemi

o aplikaci). Bázový formulář, který jsem připravil pro potřeby této studie můžete vidět na obr. 4.



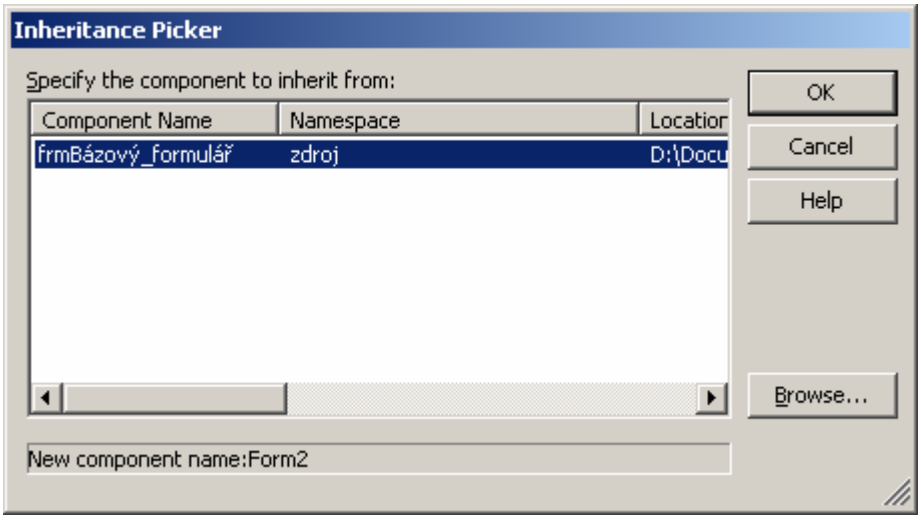
Obr. 4 – Příklad bázového formuláře

3. Pokud jste s úpravami spokojeni, vyberte nabídku **Build** a klepněte na položku **Build Solution**.
4. Vyberte menu **Project** a zvolte příkaz **Add Inherited Form**. V okně **Add New Item** se ujistěte, že v sekci **Templates** je aktivní ikona **Inherited Form** (obr. 5).



Obr. 5 – Výběr typu formuláře, jenž vznikne děděním

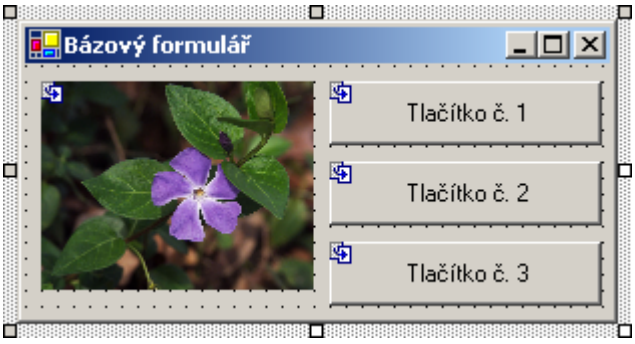
5. Klepněte na tlačítko **Open**.
6. Na obrazovce se objeví dialogové okno utility **Inheritance Picker**, ve kterém budete muset určit formulář, ze kterého se má dědit. Protože budeme chtít dědit z již připraveného bázového formuláře, vybereme jeho jméno se seznamu (obr. 6).



Obr. 6 – Výběr rodičovského formuláře v okně **Inheritance Picker**

7. Stiskněte tlačítko OK.

Inheritance Picker vloží do vašeho řešení nový (odvozený) formulář, standardně pod jménem **Form2**. Jak můžete vidět na obr. 7, synovský formulář je od svého předka takřka na nerozeznání. Skutečnost, že jde o formulář, jenž vznikl děděním vizuálně podtrhuje jenom malá ikonka se šipkou, která je nerozlučně spjata s každou instancí vložených ovládacích prvků.



Obr. 7 – Odvozený formulář, jenž vznikl děděním z bázového formuláře

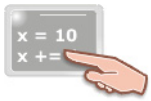
Synovský formulář lze dále upravovat podle potřeb, můžete měnit jeho vzhled či plnit událostní procedury potřebným programovým kódem.

Implementace zpracovatele události za běhu aplikace

Přepokládejme, že se dostanete do situace, kdy budete chtít za běhu programu přidat na formulář instanci ovládacího prvku **Button** a této instanci přiřadit zpracovatele události **Click**. Přesněji, bude nevyhnutné, abyste vytvořili instanci třídy **Button**, tuto instanci přidali do kolekce instancí ovládacích prvků formuláře a posléze zabezpečili, aby se po klepnutí na vytvořenou instanci ukončila právě spuštěná testovací aplikace. Hlavním problémem, jak by se na první pohled mohlo zdát, není vytvoření instance ani její umístění na formulář, dokonce ani ukončení právě běžícího procesu by nemuselo být také složité. Tím skutečným problémem je způsob, jak uskutečnit, aby se po klepnutí na vytvořenou instanci třídy **Button** spustila příslušná metoda, resp. procedura typu Sub. Hlavním cílem je proto najít přepojení mezi událostí **Click** a zpracovatelem této události (tedy procedurou Sub, která bude vyvolána jako reakce na vzniklou událost). Toto přepojení nám pomůže vybudovat příkaz **AddHandler** ve spolupráci s operátorem **AddressOf**.

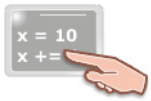
Vysvětleme si celou situaci na praktickém příkladě:

1. Přidejte na formulář instanci ovládacího prvku **Button** a do zpracovatele události **Click** této instance vložte tyto řádky programového kódu:



```
Dim tlačítko As Button
tlačítko = New Button()

tlačítko.Location = New Point(20, 80)
tlačítko.Text = "Tlačítko"
Me.Controls.Add(tlačítko)

AddHandler tlačítko.Click, AddressOf Zpracovatel_Click
```

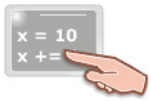
Prvním krokem je deklarace referenční proměnné s názvem **tlačítko**. Druhým krokem je posléze samotná instantace, tedy vytvoření nového objektu (nové instance třídy **Button**). Následuje určení pozice, na které se bude objekt (tlačítko) nacházet. K tomu potřebujeme vlastnost **Location**, do které přiřadíme x-ovou a y-ovou souřadnici levého horního rohu tlačítka. Přiřazením libovolného textového řetězce do vlastnosti **Text** určíme popisek tlačítka. Dále zavoláme metodu **Add** třídy **Control.ControlCollection**, které předáme název vytvořeného objektu (tlačítka). Zaměřme v této chvíli pozornost na poslední řádek:



```
AddHandler tlačítko.Click, AddressOf Zpracovatel_Click
```

Zde můžete vidět použití příkazu **AddHandler**, pomocí kterého je možné přepojit událost (**tlačítko.Click**) se zpracovatelem této událostí (**Zpracovatel_Click**). To tedy znamená, že vždy, když dojde k události **Click** objektu **tlačítko**, bude tato událost ošetřena prostřednictvím procedury Sub, jejíž název je **Zpracovatel_Click**. Aby mohl program zdárně najít zpracovatele události, je nutné poskytnout jeho adresu pomocí operátoru **AddressOf**.

2. Přesuňte se v editoru pro zápis zdrojového kódu za příkaz **End Sub** událostní procedury **Button1_Click** (jde o událostní proceduru tlačítka, které jste na formulář umístili v režimu návrhu aplikace). Stiskněte klávesu Enter, čím přikážete vložit prázdný řádek a zadejte kód zpracovatele události **Click** objektu **tlačítko**:



```
Private Sub Zpracovatel_Click(ByVal sender As System.Object, _
ByVal e As EventArgs)

    If MessageBox.Show("Přejete si ukončit tento proces?", _
    "Ukončit proces?", MessageBoxButtons.YesNoCancel, _
     MessageBoxIcon.Question) = DialogResult.Yes Then
        Process.GetCurrentProcess.CloseMainWindow()
    End If

End Sub
```

V kódu zpracovatele události je nejdříve zobrazeno dialogové okno s otázkou, zdali si uživatel opravdu přeje ukončení právě aktivního procesu (tedy spuštěné aplikace). Následně je testována návratová hodnota metody **Show** třídy **MessageBox**, která představuje hodnotu tlačítka, které uživatel stiskl. V případě, že uživatel vyjádřil kladnou odpověď na položenou otázku, je aktivní proces ukončen pomocí metody **CloseMainWindow**.

3. Stiskněte klávesu **F5** a aplikaci vyzkoušejte. Jestliže vše proběhlo v pořádku, po klepnutí na tlačítko **Button1** se vytvoří další tlačítko s názvem **Tlačítko**. Když klepnete na toto tlačítko, zobrazí se okno se zprávou. Po kladné odpovědi na položenou otázku je aplikace ukončena.

Použití vizuálních stylů Windows XP v aplikacích Visual Basicu .NET

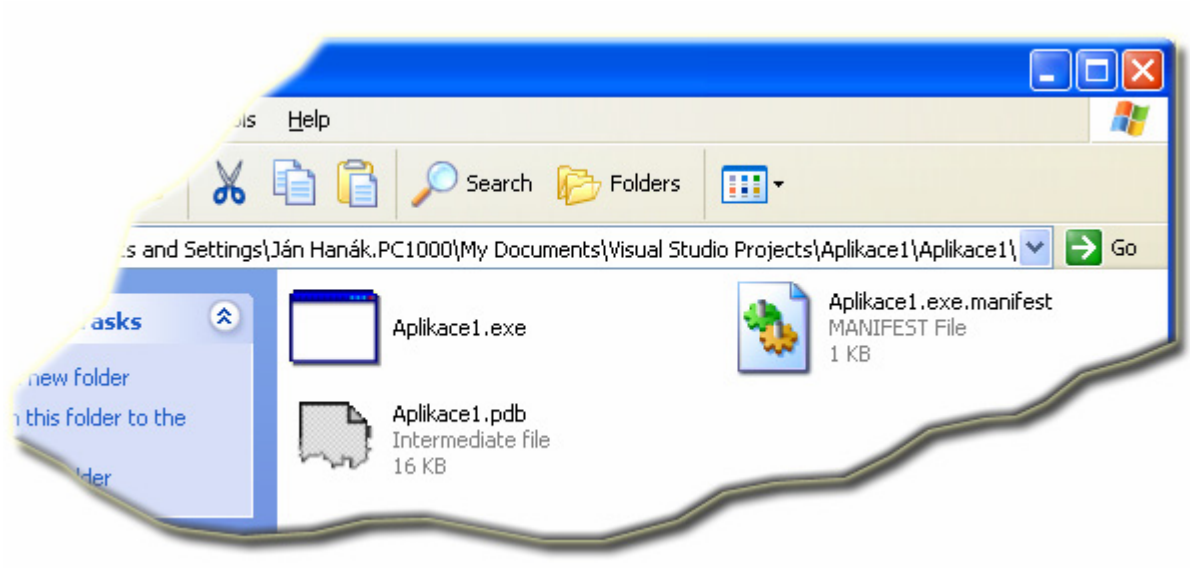
Pokud byste rádi věděli, jak změnit vizuální podobu vaší aplikace tak, aby ta šla s dobou a využívala dovedností grafického rozhraní operačního systému Windows XP, jste na správném místě. V tomto tipu si ukážeme, jak začlenit podporu pro nový grafický styl do aplikace vyvíjené ve VB .NET.

V tomto tipu se předpokládá, že pracujete s Visual Basicem .NET pod operačním systémem Windows XP.

Aby vaše aplikace mohla odpovídat modernímu vizuálnímu stylu, jenž je známý právě z Windows XP, musí obsahovat **XML manifest**, který bude definovat, jak se mají jednotlivé instance ovládacích prvků vykreslit na formuláři. Manifest má následující strukturu:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
<assemblyIdentity
version="1.0.0.0"
processorArchitecture="X86"
name="Nazev aplikace"
type="win32"
/>

<description>Popis aplikace.</description>
<dependency>
<dependentAssembly>
<assemblyIdentity
type="win32"
name="Microsoft.Windows.Common-Controls"
version="6.0.0.0"
processorArchitecture="X86"
publicKeyToken="6595b64144ccf1df"
language="*"
/>

</dependentAssembly>
</dependency>
</assembly>
```

Kód manifestu zkopírujte do Poznámkového bloku a uložte pod názvem **Jméno_aplikace.exe.manifest**. Kdybyste kupříkladu vyvíjeli aplikaci s názvem **Aplikace1**, název souboru s manifestem by byl **Aplikace1.exe.manifest**. Dále je potřebné, abyste soubor s manifestem přemístili do složky, ve které se nachází spustitelný soubor vaší aplikace (standardně je to složka **Bin**). Po uložení souboru s manifestem by měl obsah složky **Bin** vypadat přibližně takto:

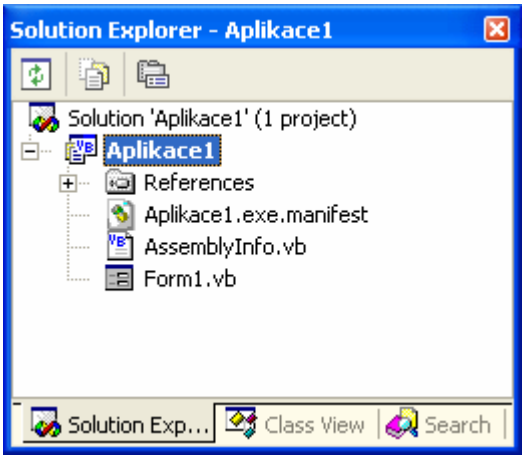


Obr. 8 – Soubor s manifestem ve složce **Bin**

Když jste se vším uvedeným hotovi, otevřete ve VB .NET řešení s kódem vaší aplikace.

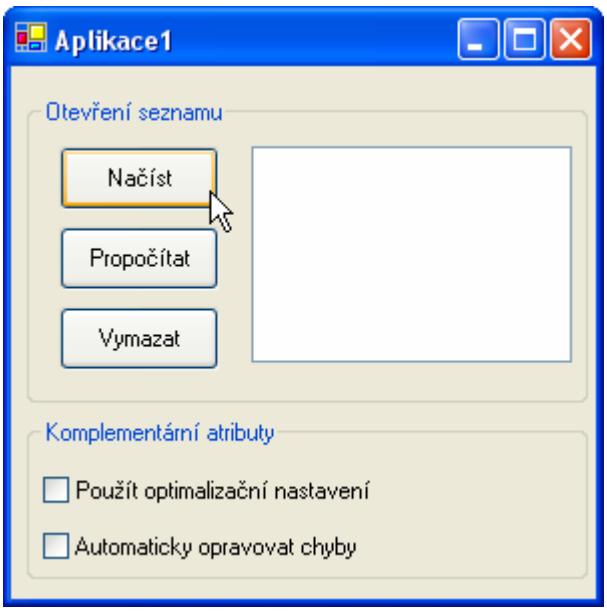
Pokračujte podle následujících kroků:

1. Nastavte vlastnost **FlatStyle** u všech instancí ovládacích prvků, které podporují tuto vlastnost na hodnotu **System**. Tím Visual Basicu přikážete, aby překreslování instancí ovládacích prvků přenechal na operační systém (Windows XP).
2. V okně **Průzkumníka řešení** (**Solution Explorer**) klepněte pravým tlačítkem myši na název vaší aplikace, v objevivší se kontextové nabídce ukažte na položku **Add** a poté klepněte na položku **Add Existing Item**.
3. Vyhledejte soubor s manifestem a předejte jej do projektu. Po přidání souboru s manifestem by mělo okno **Průzkumníka řešení** vypadat asi takto:



Obr. 9 – Okno **Průzkumníka řešení** po přidání souboru s manifestem

4. Spusťte aplikaci (klávesa **F5**) a vychutnejte si použití vizuálních stylů Windows XP (obr. 10).



Obr. 10 – Ukázka použití vizuálních stylů operačního systému Windows XP v aplikaci VB .NET

Integrované prostředí rubriky Visual Basic	Autor: **Ján Hanák**

Téma měsíce

Optimalizace (3. díl)

Časová náročnost (min): **70**	Začátečník	Pokročilý	Profesionál

Použitý operační systém :	Windows 2000 SP3
Hlavní vývojový nástroj :	Visual Basic 6.0 SP 5
Další vývojový software :	Žádný
Jiný software :	Žádný

Milí čtenáři,

dnešním dílem zakončíme problematiku optimalizace zdrojového kódu ve Visual Basicu 6.0. V předešlých částech jste se dověděli, jaké možnosti optimalizačních nastavení vám Visual Basic verze 6.0 nabízí a rovněž tak víte, jak změřit rychlost provádění jistého úseku programového kódu. Třetí a závěreční část bude věnována praktickým ukázkám optimalizace kódu. Budeme zkoumat, jaký dopad na výkon kódu budou mít námi použité techniky optimalizace.

Obsah

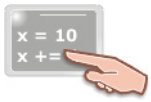
Experiment č. 1: Záhady použití datových typů

Pokud máme mluvit o komplexní optimalizaci programového kódu, musíme vycházet ze základní programovací entity, kterou je proměnná. Kdyby existoval jenom jeden datový typ, který by byl použitelný na všechny programovací operace, bylo by možné do něj uložit takřka cokoliv a jeho použití by nijak neovlivnilo rychlost provádění programových instrukcí, nebyla by otázka použití vhodných datových typů proměnných ani zdaleka tak důležitá. I když je představa ideálního datového typu jakkoliv lákavá, při praktických pokusech záhy zjistíme, že o nějakém „ideálním" stavu si můžeme nechat jenom zdát.

Visual Basic 6.0 poskytuje značné množství rozličných datových typů, které dovedou uchovávat různé hodnoty. Problémem pochopitelně není deklarace proměnné kýženého datového typu. Kdepak. Daleko složitější je vypracování si jisté intuice, která vám bude říkat, jaký nejvhodnější datový typ byste měli zvolit v závislosti od řešeného problému. Bohužel otázka optimální volby datového typu pro danou operaci je někdy natolik složitá, že ani pokročilí programátoři často nevědí, zdali ta jejich volba byla opravdu optimální.

První dobrá rada zní: V každém případě deklarujte proměnné pomocí klíčového slova **As** a specifikace vhodného datového typu. Visual Basic sice umožňuje deklaraci proměnných bez udání platného datového typu, ovšem v tomto případě více ztratíte, nežli získáte. Jestliže při deklaraci proměnné nebudete specifikovat její datový typ, bude ji automaticky přiřazen datový typ **Variant**. Typ **Variant** je jediný typ, který může uchovávat hodnoty jiných datových typů. Na druhé straně je to také „žrout paměti", protože každá instance proměnné tohoto datového typu zabere 128 bitů (16 bajtů) paměti.

Generické proměnné, což je název pro proměnné, které nemají explicitně determinován specifický datový typ, jsou ovšem v mnoha případech užitečné (například, když budete chtít vytvořit instanci jisté třídy za běhu programu, no v době psaní kódu nebudete znát specifický typ třídy, použití generické proměnné může vyřešit váš programátorský problém). Naopak, vždy, když je to možné, byste měli explicitně určit specifický typ proměnné. Budete-li chtít spustit instanci aplikace Microsoft Word a přidat do ní jeden dokument, budete deklarovat objektovou proměnnou pomocí specifického typu třídy **Word.Application**:



```
Dim Objektová_Proměnná As Word.Application
Set Objektová_Proměnná = New Word.Application

With Objektová_Proměnná
     .Documents.Add
     .Visible = True
End With
```

Použití specifického typu objektové proměnné v uvedeném příkladě demonstruje také mechanizmus, jemuž se říká **early-binding**, neboli **časné vázání** (vzniklým instancím třídy se pak říká objekty s časnou vazbou). Použití časného vázání je velmi užitečné také pro vás, jako programátory. Ve chvíli, kdy uvedete specifický typ objektové proměnné, bude Visual Basic vědět, jaké metody a vlastnosti daná třída obsahuje a tyto vám zpřístupní prostřednictvím technologie IntelliSense při použití tečkového operátoru.

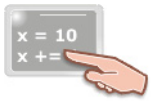
Aby uvedený fragment programového kódu pracoval spolehlivě, musíte do vaší aplikace zavést odkaz na objektové knihovny: Microsoft Office 10.0 Object Library a Microsoft Word 10.0 Object Library. V IDE Visual Basicu vyberte nabídku Project a klepněte na položku References. V seznamu pak vyhledejte uvedené knihovny a zahrňte je do projektu.

Poznámka: Objektové knihovny verze 10.0 přináleží softwaru Microsoft Office XP (2002). Pokud máte na svém počítači nainstalovanou jinou verzi, použijte samozřejmě tu vaší.

V následující sekci se podíváme na to, jaký vliv na výkon provádění programového cyklu bude mít zvolený datový typ předmětných proměnných.

Měření tempa programového kódu při použití proměnných různých datových typů

Nyní si představíme obsahy dvou událostních procedur dvou tlačítek. V obou případech budeme provádět matematické výpočty, ovšem pokaždé s jinou sadou datových typů proměnných.
V událostní proceduře **Click** prvního tlačítka budou deklarovány proměnné datového typu **Long**:

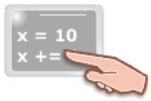


```
Dim lngČítač As Long, x1 As Long, x2 As Long
Dim začátek As Single

začátek = Timer()

For lngČítač = 1 To 200000
Randomize
x1 = Rnd() * 10000 + Rnd() * 500
```

Programový kód pokračuje na následující straně

```
x2 = x1 ^ 2
Next lngČítač

MsgBox "Cyklus s proměnnými typu Long byl vykonán za " & _
Format(Timer - začátek, "0.000")
```

V událostní proceduře **Click** druhého tlačítka budou zase použity proměnné datového typu **Variant**:



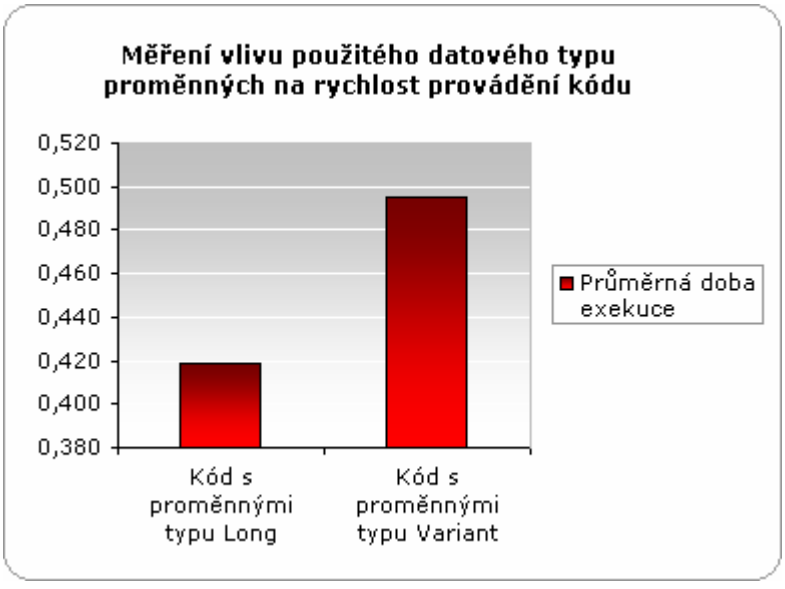

```
Dim varČítač As Variant, x1 As Variant, x2 As Variant
Dim začátek As Single

začátek = Timer()

For varČítač = 1 To 200000
Randomize
x1 = Rnd() * 10000 + Rnd() * 500
x2 = x1 ^ 2
Next varČítač

MsgBox "Cyklus s proměnnými typu Variant byl vykonán za " & _
Format(Timer - začátek, "0.000")
```

Jednotlivá měření jsou uvedena v tab. 1.

Tab. 1

Číslo měření	Proměnné datového typu Long	Proměnné datového typu Variant
1.	0, 418 s	0, 484 s
2.	0, 418 s	0, 480 s
3.	0, 422 s	0, 520 s
Průměr	**0, 419 s**	**0, 495 s**

Po provedení testů můžeme prohlásit, že kód s proměnnými datového typu **Long** byl proveden v průměru o přibližně 76 tisícin vteřiny rychleji ve srovnání se svým „variantním" protějškem (obr. 1).



Obr. 1 – Působení různých datových typů na rychlost provádění programového kódu

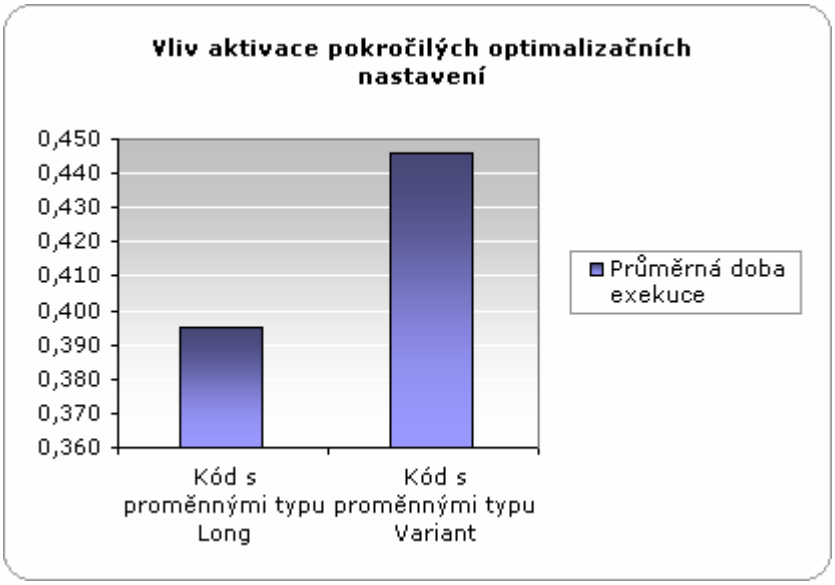
Pokračujme aktivací pokročilých optimalizačních nastavení, které nám nabízí samotný Visual Basic. Postupujte takto:

1. Vyberte nabídku **File** a klepněte na položku **Make Jméno_Aplikace.exe**.
2. V okně **Make Project** klikněte na tlačítko **Options**.
3. V dialogovém okně **Project Properties** aktivujte záložku **Compile**.
4. Zde se ujistěte, že jsou vybrány volby **Compile to Native Code** a **Optimize for Fast Code**.
5. Klepněte na tlačítko **Advanced Optimizations** a v stejnojmenném dialogovém okně zatrhněte všechny položky pokročilých optimalizačních nastavení.
6. Vygenerujte samostatně spustitelný (.EXE) soubor.

Po opětovně provedených testech jsem získal nové údaje, které jsou zobrazeny v tab. 2.

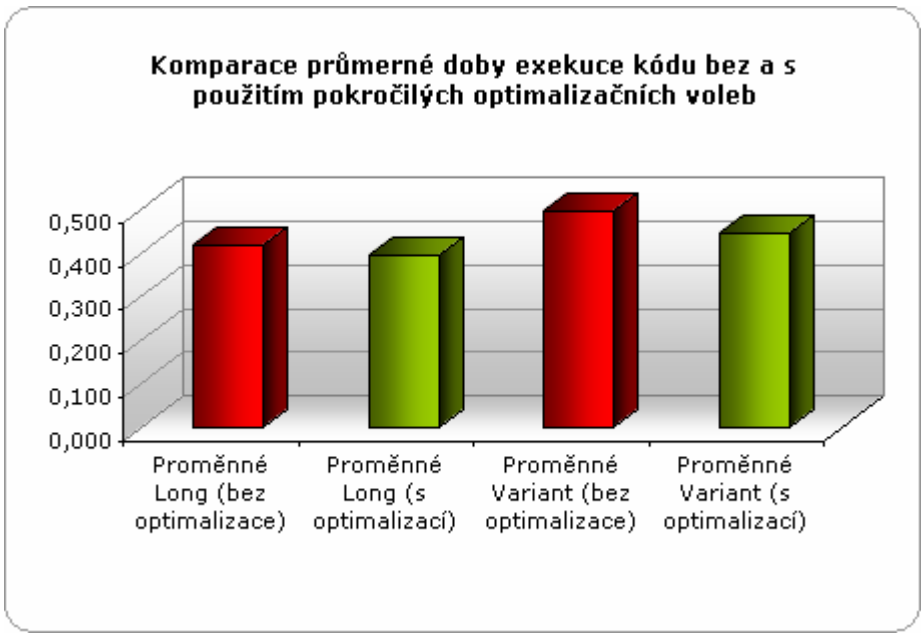
Tab. 2	**Měření výkonu programového kódu s použitím pokročilých optimalizačních nastavení**	
Číslo měření	**Proměnné datového typu Long**	**Proměnné datového typu Variant**
1.	0, 391 s	0, 449 s
2.	0, 391 s	0, 449 s
3.	0, 402 s	0, 441 s
Průměr	**0, 395 s**	**0, 446 s**

Z tabulky je jasně patrné, že exekuce programového kódu obou událostních procedur byla opět rychlejší. Výkon programového kódu s proměnnými datového typu **Long** byl lepší v průměru o čtyřiadvacet tisícin vteřiny. Zrychlila se rovněž exekuce programového kódu s proměnnými datového typu **Variant**, a to v průměru o 49 tisícin vteřiny. Ze vzájemného rychlostního porovnání událostních procedur opět vítězí ta, v těle které jsou deklarovány proměnné datového typu **Long**. Výkonnostní náskok činí přibližně 51 tisícin sekundy (obr. 2).



Obr. 2 – Větší rychlost exekuce kódu pomocí aktivace pokročilých optimalizačních nastavení

Jak je vidět, zapnutí optimalizačních voleb zabezpečilo ještě další navýšení výkonu programového kódu. Komplexní ukázku porovnání rychlosti kódu bez a s použitím optimalizačních nastavení můžete vidět na obr. 3.

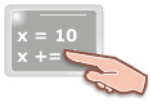


Obr. 3 – Ukázka vlivu optimalizace na rychlost prováděného kódu

Experiment č. 2: Kritické omezení rychlosti práce ovládacích prvků

Pokud pracujete často s mnoha ovládacími prvky, možná, že jste již narazili na problém, kterému se v programátorské hantýrce říká „kritické omezení rychlosti". Každý ovládací prvek je samostatnou entitou, která je založena na základních principech objektově orientovaného programování. Jedním z principů je také zapouzdření programového kódu ovládacího prvku, který není v žádném případě přístupný svému okolí (klientské aplikaci). Na jedné straně je existence zapouzdření jasnou výhodou pro autora ovládacího prvku, protože ten má jistotu, že k samotnému jádru prvku se nikdo „zvenku" nedostane. Na druhé straně ovšem vyvstává otázka, jak dokonale zvládnul autor optimalizaci programového kódu svého ovládacího prvku. Je práce s ovládacím prvkem flexibilní nebo ne? A právě teď se dostáváme k vysvětlení již předestřeného syndromu kritické omezení rychlosti ze strany ovládacího prvku. Jakmile jednou v programovém kódu dovolíte ovládacímu prvku, aby převzal chod programu do svých rukou, nemáte žádnou šanci na jakoukoliv optimalizaci jeho činnosti. Uveďme si malý příklad:

1. Ve VB 6 založte nový projekt typu **Standard EXE**.
2. Na formulář přidejte instanci ovládacího prvku **CommandButton** a také instanci ovládacího prvku **ListBox**.
3. Do událostní procedury **Click** tlačítka přidejte tento kód:




```
Dim x As Integer, čas As Single

čas = Timer()

For x = 1 To 30000
List1.AddItem x
Next x

MsgBox "Naplnění seznamu trvalo " & Format(Timer - čas, "0.000")
```

Povězme, že procesor bude potřebovat přibližně jednu sekundu na to, aby do seznamu přidal 30 tisíc čísel. I kdybyste zapnuli všechna optimalizační nastavení, nedosáhli byste žádného signifikantního nárůstu rychlosti kódu. Tento okamžik je označen jako kritické omezení rychlosti ze

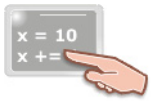
strany ovládacího prvku. Hlavním „viníkem" je metoda **AddItem** ovládacího prvku **ListBox**. Jak jsme si již pověděli, když předáme řízení kódu do rukou metody **AddItem**, nemůžeme dále optimalizovat rychlost přidávání položek do seznamu, protože touto kompetencí disponuje jenom samotný ovládací prvek **ListBox**.

Ačkoliv nemůžeme v tomto případě zabezpečit zvýšení absolutní (resp. objektivní) rychlosti programového kódu, můžeme během doby práce metody **AddItem** zaměstnat uživatelovu pozornost a informovat ho o tom, že aplikace právě dokončuje jeden ze svých pracovních úkolů. Na rozptýlení uživatele se výborně hodí ovládací prvek **ProgressBar** (ukazatel průběhu).

Instanci ovládacího prvku přidáte na formulář následovně:

1. Vyberte nabídku **Project** a klepněte na položku **Components**.
2. Ujistěte se, že je vybrána záložka **Controls** a v seznamu vyhledejte položku **Microsoft Windows Common Controls 6.0 (SP4)**.
3. Zatrhněte vybranou položku a aktivujte tlačítko **OK**. Do soupravy nástrojů (**ToolBox**) se přidá několik ovládacích prvků, mezi jinými také **ProgressBar**.
4. Poklepejte na ikonu ovládacího prvku, čímž přidáte jeho instanci na formulář.
5. Vlastnost **Align** prvku **ProgressBar** upravte na hodnotu **2 – vbAlignBottom**. Tím přikážete, aby byl ukazatel průběhu umístěný na spodní straně formuláře. Podle potřeby můžete rovněž upravit velikost vložené instance ovládacího prvku **ProgressBar**.

Zdrojový kód v proceduře **CommandButton1_Click** modifikujte podle vzoru, jenž je zobrazen níže:




```
Dim x As Integer, čas As Single

čas = Timer()

Form1.MousePointer = vbHourglass
DoEvents

Dim h As Object
Set h = ProgressBar1

With h
    .Min = 1
    .Max = 30000
End With

For x = 1 To 30000
    List1.AddItem x
    h.Value = x
Next x

h.Value = h.Min

Form1.MousePointer = vbDefault
```

Při studiu kódu si můžete všimnout několik zajímavých programovacích prvků:

- Aby bylo na první pohled jasné, že aplikace nepřestala reagovat, ale stále je aktivní, změníme vizuální podobu kurzoru myši na přesýpací hodiny (**Form1.MousePointer = vbHourglass**). Tak jsme uživateli naznačili, že aplikace má právě něco důležitého na práci.
- Rychlejšího provádění kódu dosáhneme nepřímým přístupem k vlastnosti **Value** ovládacího prvku **ProgressBar1**. Za tímto účelem deklarujeme generickou proměnnou **h**, do které posléze uložíme referenci na instanci ovládacího prvku **ProgressBar1**.
- Pokud chceme nastavit několik vlastností ovládacího prvků najednou, nebudeme je nastavovat postupně se zadáním celého jména ovládacího prvku, ale využijeme chytrou

programovací konstrukci s názvem **With-End With**. V rámci ní nastavíme minimální (**Min**) a maximální (**Max**) přípustnou hodnotu intervalu prvku **ProgressBar1**.
- V těle cyklu **For-Next** kromě zavolání metody **AddItem** ovládacího prvku **List1**, aktualizujeme také hodnotu ukazatele průběhu.
- Po přidání všech položek do seznamu je hodnota vlastnosti **Value** ovládacího prvku **ProgressBar1** nastavena na minimum (**Min**).
- Nakonec je zpětně změněna také podoba kurzoru myši (**Form1.MousePointer = vbDefault**).

Čas, který bude potřebný pro vykreslení a aktualizaci ukazatele průběhu bude, přes všechny pokusy o optimalizaci zdrojového kódu, o něco delší, nežli původně zjištěná hodnota. Vskutku, také ovládání ovládacího prvku **ProgressBar** něco stojí. Pro dosažení optimálního výsledku je zapotřebí najít rovnováhu mezi objektivní a subjektivní rychlosti aplikace.

Jestliže chcete, můžete uvedenou situaci vyřešit ještě jinak:

1. Nemusíte použít ovládací prvek **ProgressBar**, ale jenom změňte kurzor myši na přesýpací hodiny.

2. Vytvořte speciální formulář, který zobrazíte uživateli během načítání položek do seznamu (aby byl formulář řádně překreslen, po jeho zobrazení zavolejte funkci **DoEvents**). Formulář bude uživateli říkat, že aplikace právě provádí důležitou operaci, dokončení které si vyžaduje dodatečný čas.

3. Načítejte položky do seznamu předem, například při startu aplikace.

Pokuste se načíst položky v menších množstvech. Tedy nebudete načítat všech 30 tisíc položek najednou, ale vytvoříte menší skupiny položek (třeba po pěti tisících), které poté načtete rychleji.

Integrované prostředí rubriky Visual Basic	Autor: **Ján Hanák**

Lab

Téma

Začínáme s VB .NET

Úvod do světa .NET (7. díl)

Časová náročnost (min):	Začátečník	Pokročilý	Profesionál
95	1	2	3

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

Vážení programátoři,

nabídka červencového dílu seriálu bude vskutku bohatá. Nejprve se podíváme na některá pravidla, která byste měli dodržovat při pojmenovávání proměnných jak hodnotových, tak i referenčních datových typů. Dále vzpomeneme využití komentářů a jejich smysl při dokumentaci zdrojového kódu. Ve finále se můžete těšit na vysvětlení pojmů funkce, argument a parametr. Rovněž dojde i na ozřejmění dvou základních způsobů předávání argumentů. Příjemné počtení.

Obsah

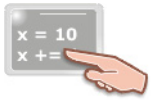
Zásady pro pojmenovávání proměnných

I když by se mohlo na první pohled zdát, že problematika správného pojmenování proměnných je, ve srovnání s jinými programátorskými oblastmi, poněkud méněcenná, není tomu tak. Dokonce lze říci, že pokud programátor ovládne ty správné návyky v určování názvosloví proměnných, získá nejenom značný přehled o situaci ve svém zdrojovém kódu, ale bude rovněž dostatečně připraven pro profesionální práci v programátorském týmu.

Na začátek si uveďme několik dobrých rad, které můžete při pojmenovávání proměnných vhodně využít.

1. Jméno proměnné **nesmí začínat číslicí**, protože Visual Basic .NET tuto skutečnost přímo zakazuje. Mimochodem, podobné pravidlo zastávají také jiné programovací jazyky. Na druhé straně třeba připomenout, že i když se číslice nemůže nacházet na samém počátku jména proměnné, na jiné pozici v názvu není její přítomnost zakázána.

Příklady:

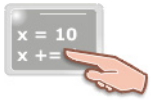


```
Dim 123Objednávka as Short
'Nepovolené pojmenování proměnné.

Dim Objednávka123 As Short
'Přípustné pojmenování proměnné.
```

2. Jménem proměnné nesmí být žádné klíčové slovo Visual Basicu .NET. I kdyby deklarace takovéto proměnné proběhla bez problémů, záhy po spuštění programu byste obdrželi nesprávné výsledky a další chyby.

3. Jméno proměnné **může začínat písmenem** (malým nebo velkým) a také **znakem podtržení** (_).

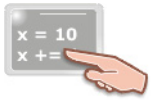
   Příklady:



```
Dim Název As String
Dim výpočet As Double
Dim _Proměnná As Integer
```

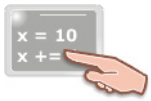
4. Jméno proměnné by mělo být co možná nejvýstižnější a rovněž tak i nejkratší. Vždy se snažte přiřadit proměnné název, který tvoří co nejméně znaků, resp. volte vždy nejkratší kombinaci znaků a číslic. I když ve skutečnosti Visual Basic .NET nijak nestanovuje maximální délku proměnné, zkuste se vyhnout názvům delším nežli 32 znaků.

5. Při studiu cizího programového kódu se můžete střetnout s několika způsoby pojmenovávání proměnných. Poměrně často je název proměnné složen ze dvou (nebo i více) slov, z kterých každé začíná velkým písmenem abecedy:



```
Dim PočetVýrobků As Integer
Dim KoeficientInflace As Single
```

   Druhá varianta spočívá v častém použití znaku podtržení (_):

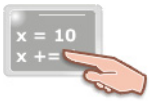


```
Dim Obsah_kruhu As Integer
'nebo také
Dim Obsah_Kruhu As Integer
```

   Ovšem vezměte prosím v potaz, že z posledních dvou příkladů pojmenování proměnné může být v jedné lokální oblasti (proceduře, funkci, metodě) použit jenom jeden z nich. Protože Visual Basic .NET nepatří mezi programovací jazyky, které by rozlišovali malá a velká písmena, považoval by deklaraci druhé proměnné za chybnou (jde o chybu tzv. duplicitní deklarace proměnné).

6. Ve světě programování byla otázka vhodného pojmenování proměnných vždy předmětem zájmu mnoha vědců a odborníků. Jedním z průkopníků byl také Američan maďarského původu Charles Simonyi, který vytvořil takzvanou maďarskou notaci. Hlavní smysl této

notace tkví v přidání speciálního prefixu před jménu proměnné. Jednoduše, před jméno každé proměnné byla umístěna předpona, která identifikovala datový typ té-které proměnné. Ukažme si příklady:



```
Dim intDeterminant As Integer
Dim strSlogan As String
```

Prefix býval zvyčejně třípísmenný a spolehlivě charakterizoval příslušnost proměnné k vybranému datovému typu. V uvedených příkladech prefix **int** říká, že proměnná je datového typu **Integer** a předpona **str** zase identifikuje proměnnou datového typu **String**.

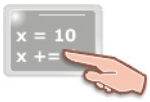
Maďarská notace byla velice populární především při programování v jazycích C a C++, později se ovšem také dostala i do prostředí Visual Basicu. Mnoho programátorů se s ní sžilo natolik, že ji používají dodnes.

7. Nakonec jsem si nechal jedno staré, ovšem léty prověřené pravidlo: V podstatě nezáleží na tom, jaký způsob pojmenovávání proměnných si vyberete. Důležité je, abyste vybraný styl pojmenovávání proměnných povýšili na standard, který budete v každém případě dodržovat.

Komentáře a jejich smysl v procesu dokumentace zdrojového kódu

Visual Basic .NET poskytuje programátorům již hodně dlouho možnost zařadit do kódu také řádky ryze informačního charakteru, kterým se říká komentáře. Ve výpisech programových kódů jsem už komentáře někdy používal, takže jste se s nimi již mohli střetnout. Komentář můžete do vašeho kódu zařadit velmi jednoduše:

1. Vložte do kódu **znak apostrofu** (') a zapište informační text:



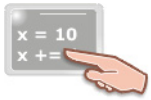
```
Dim CenaVýrobku As Single
'Právě byla deklarována proměnná datového typu Single.
```

Vše, co se objeví za znakem apostrofu považuje VB .NET za komentář. Komentář poznáte lehce podle jeho zelené barvy.

Informační text komentáře může být uveden nejen znakem apostrofu, ale také textovým řetězcem **REM**. Ve chvílích, kdy nebudete chtít hledat správnou klávesu pro vložení apostrofu, vám možná zapsání formulky REM ušetří drahocenný čas.

Pokud by vám standardní barva komentáře nevyhovovala, můžete ji změnit. Více informací si můžete přečíst v kapitole **Jak změnit standardní barvu komentářů**.

Při práci s komentáři si musíte pamatovat, že systém komentářů je založen na řádkovém schématu. To znamená, že po vložení apostrofu je celý řádek vyhrazen jenom pro komentář. Když do takového řádku vložíte programový kód, tento nebude proveden, protože Visual Basic .NET bude přepokládat, že jde o komentář. Komentář se samozřejmě může rozkládat také na více řádcích. V tomto případě se ale musí na začátku každého řádku nacházet znak apostrofu. Podívejte se na následující ukázku:




```
MessageBox.Show("Tento text se zobrazí v okně se zprávou.")
'Metoda Show
'třídy MessageBox
'zabezpečí zobrazení okna se zprávou.
```

Velmi důležitá je skutečnost, že při kompilaci zdrojového kódu aplikace komentáře zbytečně nezdržují práci překladače, nakolik tento je ignoruje. Není tedy pravdou tvrzení, že čím víc komentářů v kódu používáte, tím je váš kód prováděn pomaleji.

Implementace komentářů do programového kódu vaší aplikace je velmi důležitá. Pokud budete schopni vhodnou formou dokumentovat činnost vašeho kódu, vytvoříte si pevnou základnu pro pozdější modifikaci a rozšiřování funkčnosti aplikace. Přitom bude dobré, když budete komentovat jak jednoduché sekvence, tak i složité vnořené partie programového kódu. Usilí, které vynaložíte na tvorbu komentářů, se vám vrátí i s pomyslnými úroky ve chvíli, když se k předmětnému kódu vrátíte po půlroce, nebo i delší době.

Značné množství programátorů začátečníků propadá dojmu, že kódu, který napíšou dnes, budou rozumět kdykoliv. Když se tito programátoři po delší době vrátí k vlastnímu zdrojovému kódu, jenom s hrůzou budou nahlížet na komplikované programovací konstrukce, na pochopení kterých budou potřebovat dodatečný čas. Kdyby ovšem svůj kód pečlivě dokumentovali, tento dodatečný čas mohli využít daleko smysluplněji.

Význam komentářů ještě vzrůstá při studiu cizího zdrojového kódu. Věřte, že přijít na to, jak myslel jiný programátor při psaní té-které funkce či procedury, je někdy podstatně těžší, jako napsání své vlastní procedury nebo funkce.

Pokud vyvíjíte uživatelský ovládací prvek nebo komponentu, kterou pak poskytnete jiným vývojářům bezplatně, nebo ji budete nabízet na trhu softwarových součástí, měli byste dokumentační systém rozšířit o kompletní elektronickou dokumentaci ve formě nápovědy. V případě ovládacího prvku by nápověda měla obsahovat popis všech tříd (bázových i abstraktních), rozhraní, metod a vlastností.

Návrh ovládacích prvků by měl vyhovovat základním principům objektově orientovaného programování (OOP). Jedním z těchto principů je takzvané **zapouzdření** (**encapsulation**). Zapouzdření říká, že klientský kód by měl s ovládacím prvkem komunikovat jenom prostřednictvím jeho rozhraní (tedy voláním příslušných metod a využíváním vstavěných vlastností). Samotné jádro implementace ovládacího prvku by ale nikdy nemělo být přístupné klientské aplikaci. Z tohoto důvodu se cizí programátor nikdy nedostane ke komentářům zdrojového kódu ovládacího prvku.

Jak změnit standardní barvu komentářů

Nejste-li spokojeni se zelenou barvou komentářů, připravil jsem pro vás postup, jak tuto barvu modifikovat a změnit podle vašich požadavků. Postupujte podle následujících instrukcí:

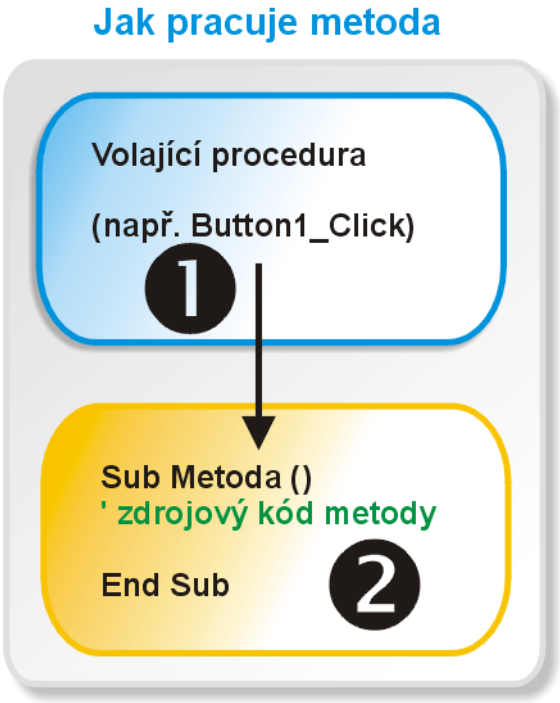
1. Vyberte nabídku **Tools** a klepněte na položku **Options**.
2. V objevivším se dialogovém okně klepněte na složku **Environment** a vyberte položku **Fonts and Colors**.
3. V seznamu **Display items** vyhledejte položku **Comment**.
4. Z otevíracího seznamu **Item foreground** vyberte barvu pro komentáře.

Jestliže vás ani nabídka barev otevíracího seznamu **Item foreground** nepotěšila, klepněte na tlačítko **Custom** a vyberte si vlastní barvu z barevného spektra RGB.

Charakteristika pojmu funkce

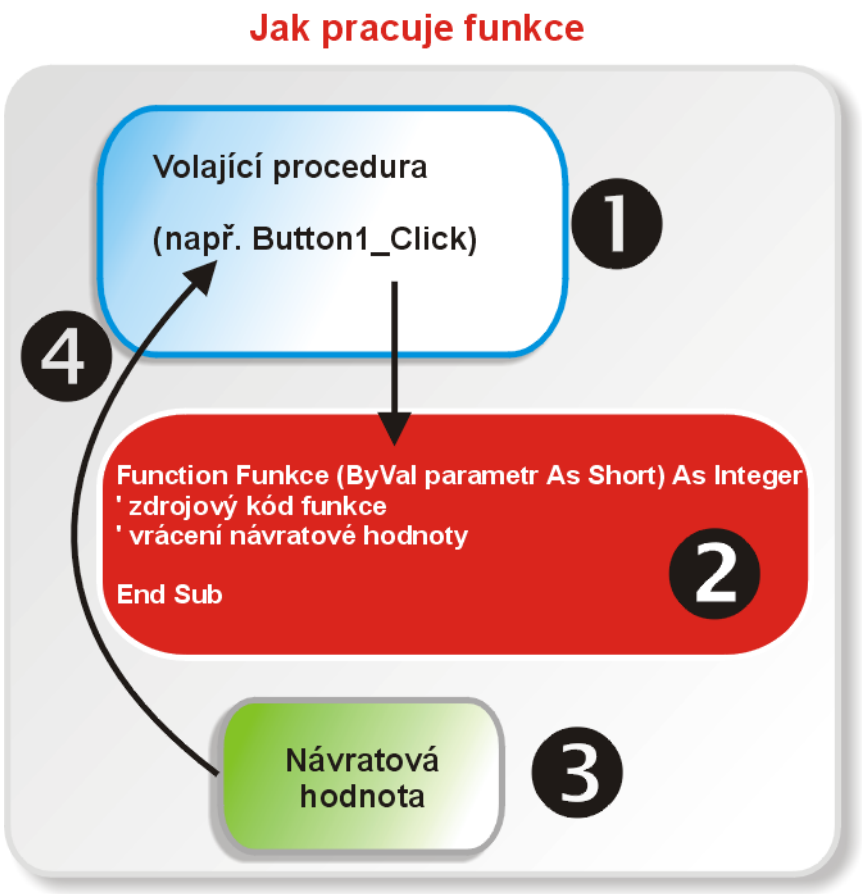
Věřím, že funkcím přijdete rychle na chuť, protože pod pojmem funkce si můžete představit metodu, která vrací jistou hodnotu. Právě schopnost vracet hodnotu dělá z obyčejné metody pravou a nefalšovanou funkci. Hodnota, kterou funkce vrací, se nazývá návratová hodnota funkce. Hlavní rozdíl mezi metodou a funkcí je následující:

- Volání metody se uskutečňuje z volající procedury (touto volající procedurou může být třeba událostní procedura **Button1_Click**). Po zavolání metody je proveden programový kód, jenž se nachází v jejím těle (tedy mezi příkazy **Sub** a **End Sub**). Metoda přitom nevrací žádnou návratovou hodnotu. Úkolem metody je jenom exekuce jistého bloku kódu a po splnění tohoto úkolu je provádění metody ukončeno (metoda je ukončena ve chvíli, kdy procesor přejde na řádek s příkazem **End Sub**). Situaci vizuálně demonstruje obr. 1.



Obr. 1 – Činnost práce metody

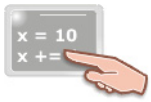
- Volání funkce se také uskutečňuje z volající procedury (třeba **Button1_Click**). Po zavolání funkce je, podobně jako u metody, proveden jistý fragment programového kódu. Jak jsme si již řekli, funkce vrací návratovou hodnotu. Tato hodnota je nejprve vypočtena a poté předána zpět volající proceduře. Samotná volající procedura posléze naloží s návratovou hodnotou funkce podle svých potřeb (např. ji použije v další sérii výpočtů). Situaci vizuálně znázorňuje obr. 2.



Obr. 2 – Činnost práce funkce

Deklarace funkce ve VB .NET

Nejdříve se podívejme na ukázkovou deklaraci funkce, a poté si povíme, z čeho se deklarační příkaz funkce skládá:




```
Public Function Moje_Funkce(ByVal číslo As Short) As Integer

    Dim Druhá_Mocnina As Integer
    Druhá_Mocnina = číslo * číslo
    Return Druhá_Mocnina

End Function
```

Deklarační příkaz pro vytvoření funkce pozůstává z těchto hlavních součástí:

- **Modifikátor přístupu**
  Modifikátor přístupu určuje úroveň, na které bude funkce přístupná. Jinými slovy, pomocí modifikátoru přístupu můžete určit, zdali bude funkce veřejná (**Public**), a tak přístupná z celého projektu, nebo naopak soukromá (**Private**). Soukromá funkce je přístupná pouze v rámci vyššího programového bloku, v kterém je deklarována (tímto blokem může být například třída). Kromě modifikátorů **Public** a **Private** existují ještě další (**Protected**, **Friend** a **Protected Friend**), ovšem těmi se v této chvíli nebudeme zabývat.

- **Příkaz Function**
  Tento příkaz musí následovat ihned za specifikací modifikátoru přístupu. Příkaz říká Visual Basicu, že budete chtít vytvořit funkci. Protějškem příkazu **Function** je příkaz **End Function**. Tyto dva příkazy ohraničují takzvané tělo funkce. Tělem funkce se proto rozumí všechny řádky programového kódu, které se nacházejí mezi příkazy **Function** a **End Function**.

- **Název funkce**
  Název funkce představuje uživatelsky přívětivé pojmenování funkce (v našem případě je názvem funkce řetězec **Moje_Funkce**). Zvolený název funkce budete používat vždy, když budete chtít funkci zavolat.

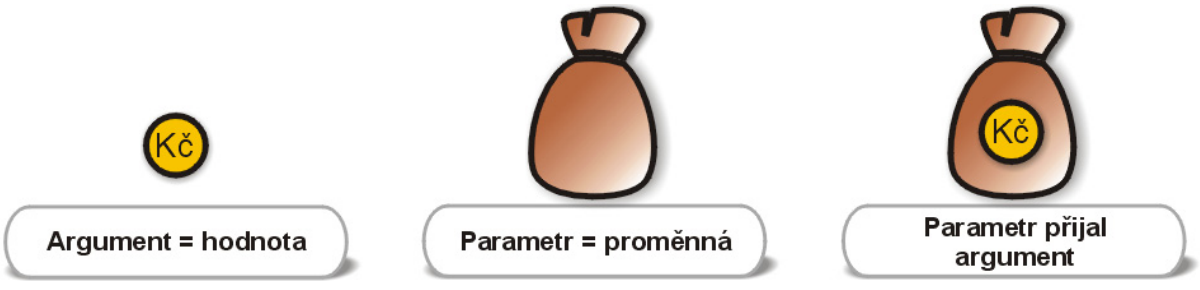
- **Signatura funkce**
  Pod pojmem signatura funkce se rozumí seznam parametrů funkce, který je umístěn v kulatých závorkách. Jestliže signatura funkce obsahuje víc jak jeden parametr, jsou parametry od sebe odděleny čárkami. Signatura funkce nemusí také obsahovat žádný parametr, ovšem i v tomto případě je použití kulatých závorek povinné. V našem příkladě tvoří signaturu funkce jenom jeden parametr datového typu **Short** s názvem **číslo**. Bližší pohled na parametry a argumenty můžete najít v kapitole Charakteristika pojmů argument a parametr.

- **Návratová hodnota funkce**
  Jak již víte, funkce po vykonání své práce vrací návratovou hodnotu. V deklaračním příkazu je potřebné determinovat datový typ návratové hodnoty funkce. Určení návratové hodnoty funkce se uskuteční zapsáním klíčového slova **As** za signaturu funkce a přidáním názvu platného datového typu. Z deklarace naší funkce je zřejmé, že funkce bude vracet celočíselnou hodnotu typu **Integer**. Vrácení hodnoty má na starosti příkaz **Return**. Vše, co se nachází za tímto příkazem tvoří návratovou hodnotu funkce (v našem případě jde o hodnotu proměnné **Druhá_Mocnina**).

Naše funkce (**Moje_Funkce**) pracuje v několika krocích. Abyste ovšem byli schopni opravdu pochopit skutečnou činnost funkce, musíme si nejprve ozřejmit pojmy argument a parametr.

Charakteristika pojmů argument a parametr

Pojmy **argument** a **parametr** jsou ve světě programování velice důležité, a proto bude dobré, když budete jejich studiu věnovat více času. **Argumentem** se rozumí **hodnota**, která je poskytnuta funkci jako jakási „vstupní surovina". Aby ovšem funkce mohla tuto hodnotu přijmout a uchovat, musí disponovat **parametrem**, neboli **proměnnou**, do které se přijímaná hodnota uloží. Vztah mezi argumentem a parametrem je znázorněn na obr. 3.



Obr. 3 – Ilustrace vztahu argumentu a parametru

Charakteristika způsobu předávání argumentů

Visual Basic .NET podporuje dvě varianty předávání argumentů:

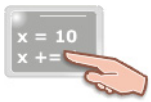
- **Předávání argumentů hodnotou (pomocí klíčového slova ByVal)**
  Při předávání argumentu hodnotou je volané funkci poskytnuta jenom kopie skutečného argumentu. Funkce tak pracuje pouze s kopii skutečného argumentu a i když argument změní, tato změna bude pouze lokální a nijak neovlivní skutečný argument. Jestliže budete chtít, aby byl argument funkci předán hodnotou, před název parametru, jenž bude přijímat tento argument, dodejte klíčové slovo **ByVal**. Použití tohoto klíčového slova při předávání argumentů hodnotou ovšem není povinné, protože VB .NET implicitně předává všechny argumenty hodnotou. Když tedy před jméno přijímacího parametru nepřidáte klíčové slovo **ByVal**, nic se neděje, stejně bude argument předáván hodnotou. Zapsání klíčového slova **ByVal** ale zpřehledňuje kód a vizuálně vás informuje o způsobu předávání argumentů.

- **Předávání argumentů odkazem (pomocí klíčového slova ByRef)**
  Na druhé straně se můžete střetnout také s předáváním argumentů odkazem. V tomto případě je volané funkci poskytnut skutečný argument. Přesněji, funkci je dodán ukazatel na paměťovou adresu, na které se skutečný argument nachází. Když funkce získá přístup ke skutečné hodnotě, může ji také lehce modifikovat podle svých požadavků. Když budete chtít, aby byl parametru funkce předán argument odkazem, před jméno parametru přidejte klíčové slovo **ByRef**. Předávání argumentů odkazem je velmi užitečná technika, která se využívá také při některých pokročilých programovacích postupech.

Praktická ukázka použití funkce

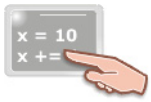
V této praktické studii si ukážeme, jak volat funkci z volající procedury. Postupujte takto:

1. Vytvořte novou aplikaci pro Windows (**Windows Application**).
2. Na formulář přidejte jedno tlačítko, které nechte implicitně pojmenované.
3. Poklepejte na tlačítko, čím zobrazíte editor pro zápis kódu. Za příkaz **End Sub** událostní procedury **Button1_Click** vložte prázdný řádek a zadejte následující programový kód pro funkci **Moje_Funkce**:




```
Public Function Moje_Funkce(ByVal číslo As Short) As Integer
    Dim Druhá_Mocnina As Integer
    Druhá_Mocnina = číslo * číslo
    Return Druhá_Mocnina
End Function
```

4. Do událostní procedury **Button1_Click** vložte tento řádek kódu:




```
MessageBox.Show(Moje_Funkce(100).ToString)
```

Jednotlivé části deklarace funkce **Moje_Funkce** jsme si již objasnili. V této chvíli se zaměřte na parametr funkce s názvem **číslo**. Jak vidíte, před jménem parametru je uvedeno klíčové slovo **ByVal**, které indikuje, že argument, který bude parametr přijímat, bude předáván hodnotou. Parametr bude tedy obsahovat jenom kopii skutečné hodnoty. A jakáže hodnota bude parametru předána? Bude to hodnota, která je poskytnuta funkci při jejím zavolání z volající procedury. Když se podíváte na kód událostní procedury **Click** tlačítka **Button1**, uvidíte, že funkci je poskytnuta celočíselná hodnota 100.

Pojďme ovšem dál a podívejme se blíže na algoritmus práce funkce.

Poté, co je funkci dodána hodnota 100, je kopie této hodnoty (resp. tohoto argumentu) umístěna do parametru **číslo**. Protože je parametr deklarován jako proměnná datového typu **Short**, jehož rozsah tvoří uzavřený interval <-32,768, 32,767>, je vše v pořádku a kopie argumentu může být úspěšně uložena do proměnné **číslo**. Kdybyste ovšem překročili povolený rozsah datového typu, program by nepracoval správně a střetli byste se s chybou za běhu programu.

V okamžiku, kdy parametr funkce obsahuje kopii skutečné hodnoty, je tato kopie použita k další operaci, kterou je vypočtení její druhé mocniny. Výsledek součinu je posléze uložen do lokální proměnné s názvem **Druhá_Mocnina**. Ve finálním kroku je vypočtená druhá mocnina argumentu vrácena pomocí příkazu **Return** volající událostní proceduře **Button1_Click**. Vrácená hodnota je nakonec zobrazena v okně se zprávou, které je zobrazeno prostřednictvím metody **Show** třídy **MessageBox**.

Integrované prostředí rubriky Visual Basic	Autor: Ján Hanák

Téma

VB .NET

Programátorská laboratoř

Prostor pro experimentování

Časová náročnost (min):	Začátečník	Pokročilý	Profesionál
70	1	2	3

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

VB .NET	Ukázka míchání programovacích jazyků platformy .NET (Visual Basic .NET a Visual C# .NET)	3	0:45
VB .NET	Implementace vlastnosti, která je jenom pro čtení	2	0:20
VB .NET	Zobrazení čísel řádků v editoru pro zápis kódu	1	0:05

Ukázka míchání programovacích jazyků platformy .NET (Visual Basic .NET a Visual C# .NET)

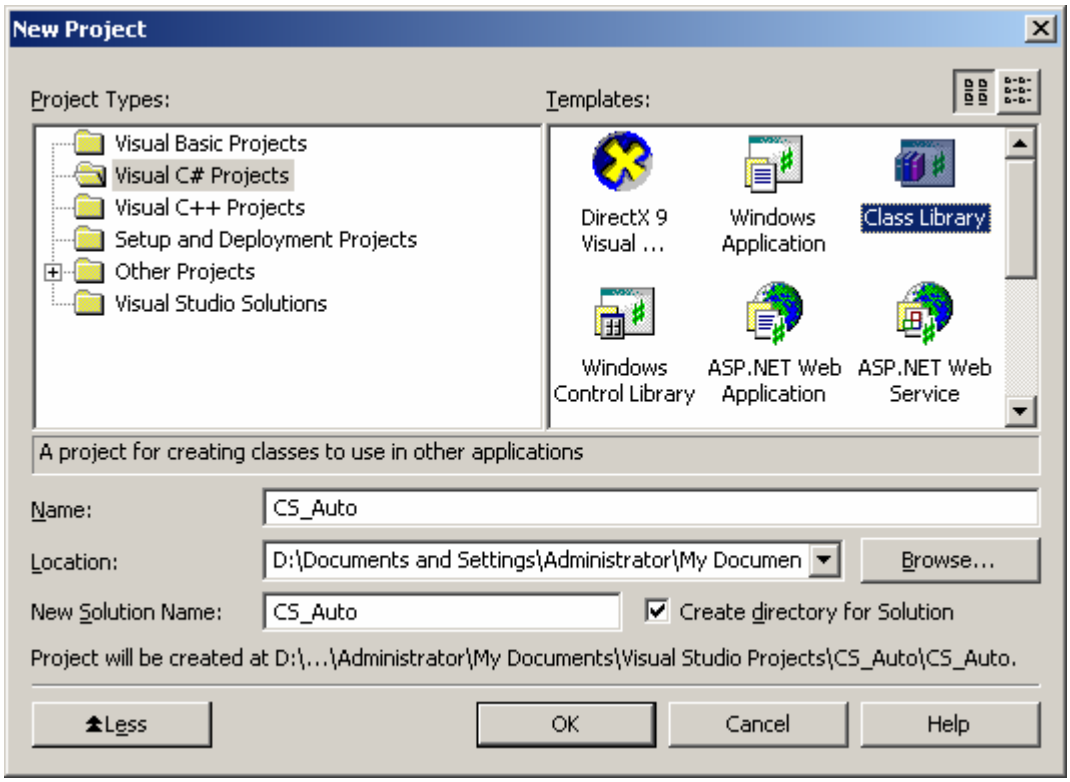
Skalní příznivci naší programovací rubriky si jistě vzpomenou, že problematice míchaní programovacích jazyků jsme se již věnovali. Tehdy bylo naším společným cílem vytvoření dynamicky linkované knihovny (DLL) v jazyku Visual C++ a posléze její využití v prostředí Visual Basicu. V tomto tipu si na mušku vezmeme dalšího zástupce programovacích jazyků platformy .NET, kterým je zcela nový a plně objektově orientovaný Visual C# .NET.

C# představuje další evoluční článek ve vývoji programovacích jazyků, základem kterých byl populární, i když pro mnoho programátorů značně obtížný jazyk C a posléze také C++. Ti z vás, kteří někdy pracovali s Visual C++ a jeho IDE, jistě uznají, že největším problémem, zejména v začátcích, byla absence plně vizuálního prostředí pro návrh aplikací ve stylu Windows. Dobrou zprávou je, že při programování ve Visual C# .NET již nebudete odkázáni jenom na psaní programovacích smyček, ale můžete využít stejné IDE, na jaké jste zvyklí z VB .NET. Ve všeobecnosti by bylo možné prohlásit, že jazyk C# je velmi vhodným kandidátem na to, abyste se pokusili porozumět alespoň základům práce s ním. Nejenom, že vám nabízí všechno „dobré" ze světa C a C++, ale přidává mnohé prvky a programové konstrukce, které jsou vám, jako profesionálním programátorům ve VB .NET, v mnoha ohledech známé.

V následující případové studii si ukážeme, jak lze spojit sílu Visual Basicu a C#. Naším hlavním zájmem bude vytvoření třídy v prostředí C#. Tato třída bude obsahovat jednu metodu, kterou později zavoláme z Visual Basicu .NET. Pusťme se tedy do práce.

Tvorba projektu typu Class Library ve Visual C#

1. Spusťte Visual C#, vyberte nabídku **File**, ukažte na **New** a klepněte na položku **Project**.
2. V dialogovém okně **New Project** se ujistěte, že v sekci **Project Types** je vybrán adresář **Visual C# Projects**. V rámečku **Templates** zvolte položku s ikonou **Class Library** (obr. 1). Projekt s třídou jsem pojmenoval jako **CS_Auto** a rovněž tak jsem aktivoval i volbu **Create directory for Solution**.



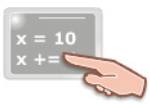
Obr. 1 – Výběr typu projektu **Class Library** ve Visual C# .NET

3. Jestliže klepněte na tlačítko **OK**, C# vytvoří projekt s názvem **CS_Auto**, do kterého přidá třídu s názvem **Class1**.

Pokud se podíváte do okna **Solution Explorer**, uvidíte, že kód třídy **Class1** je uložen v souboru s názvem **Class1.cs**.

Bližší pohled na automaticky vygenerovaný kód projektu

Podívejme se v této chvíli na všechen automaticky sestavený kód a povězme si, co dělá:




```
using System;

namespace CS_Auto
{
      /// <summary>
      /// Summary description for Class1.
      /// </summary>
```

Programový kód pokračuje na následující straně

```
        public class Class1
        {
              public Class1()
              {

                    //
                    // TODO: Add constructor logic here
                    //
              }
        }
  }
```

Při studiu programového kódu jazyka Visual C# uvidíte, že některé příkazy a klíčová slova jsou psána malými písmeny a jiné partie kódu kombinují malá i velká písmena. Je potřebné, abyste měli na paměti skutečnost, že jazyk C# striktně rozlišuje mezi malými a velkými písmeny abecedy (jazyk C# je tedy „case-sensitive", podobně jako třeba C nebo C++).

Na úvodním řádku je použit příkaz **using**, který je protějškem příkazu **Imports** z VB .NET. Zavádí do programového kódu třídy odkaz na jmenný prostor .NET Frameworku s názvem **System**. Podobně také příkaz **Imports** se ve VB .NET používá na to, aby bylo možné odkazovat se na další třídy, metody a vlastnosti daného jmenného prostoru bez nutnosti uvádění plně kvalifikovaných názvů. Ve C# se příkaz **using** musí nacházet ještě před definicí jakýchkoliv dalších jmenných prostorů.

Podobně jako v jazycích C a C++, tak i v C# je řádek programového kódu ukončen středníkem, tedy ne stiskem klávesy Enter, jak je tomu ve Visual Basicu .NET.

Výpis kódu pokračuje definicí jmenného prostoru pomocí klíčového slova **namespace**. Jak si můžete všimnout, jméno projektu (**CS_Auto**), které jste zadali při vytváření projektu, je současně názvem jmenného prostoru. Kód samotné třídy (zatím standardně pojmenované jako **Class1**) i konstruktoru se nachází ve vyhrazeném bloku, jenž je ohraničen klíčovým slovem **namespace** a příslušnými složenými závorkami, které definují hranice tohoto programového bloku.

Značka **<Summary>** patří mezi několik dokumentačních značek, které podporují implementaci XML dokumentace programového kódu jazyka C#.

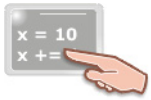
Konečně se dostáváme k deklaraci samotné třídy **Class1**. Třída je deklarována s přídomkem **public**, co znamená, že jde o veřejnou třídu, která bude dostupná v celém projektu. Přesnější by bylo vyjádření, že instance této veřejné třídy, tedy samotné objekty, bude možné tvořit z jakéhokoliv místa v projektu. Standardní název třídy změňte na **Automobil**.

V bloku třídy je deklarován také konstruktor, který má ve Visual C# stejný název jako samotná třída. Konstruktorem se rozumí metoda, která je primárně aktivována v okamžiku vytváření instance třídy. Konstruktor je velmi užitečný, protože prostřednictvím něho lze vhodně inicializovat datové členy třídy ještě předtím, než se tyto použijí. Jak v prostředí C#, tak i ve VB .NET lze vytvářet několik variant konstruktorů se stejným názvem a odlišnou signaturou. Uvedenému procesu se poté říká přetížení konstruktoru.

Protože třída a její konstruktor musí mít stejné názvy, upravte i název konstruktoru na **Automobil**.

Přidání metody do vytvořené třídy

Do třídy **Automobil** přidáme pro zjednodušení jenom jednu veřejnou metodu s názvem **VypočístSpotřebu**, úkolem které bude zjištění výše spotřeby automobilu. Ukažme si ovšem nejprve výpis modifikovaného programového kódu projektu a poté si vysvětlíme, jak ve skutečnosti pracuje:




```
using System;

namespace CS_Auto
{
	/// <summary>
	/// Summary description for Class1.
	/// </summary>
	public class Automobil
	{
		public Automobil()
		{
			//
			// TODO: Add constructor logic here
			//
		}

		public int VypočístSpotřebu(int PočetKilometrů,
			short SpotřebaNaKilometr)
		{
			return PočetKilometrů*SpotřebaNaKilometr;
		}
	}
}
```

Kód metody **VypočístSpotřebu** se nachází pod kódem konstruktoru třídy. Metoda je definována jako veřejná a pracuje se dvěma parametry:

- **PočetKilometrů** v podobě datového typu **int**
- **SpotřebaNaKilometr** v podobě datového typu **short** (budeme zjednodušeně předpokládat, že výše spotřeby bude vždy udávána v celočíselné hodnotě)

V těle metody je umístěn jenom jeden řádek kódu, který vypočte celkovou spotřebu automobilu a tuto vypočtenou hodnotu následně vrátí volající proceduře (volající proceduru vytvoříme za chvíli ve VB .NET).

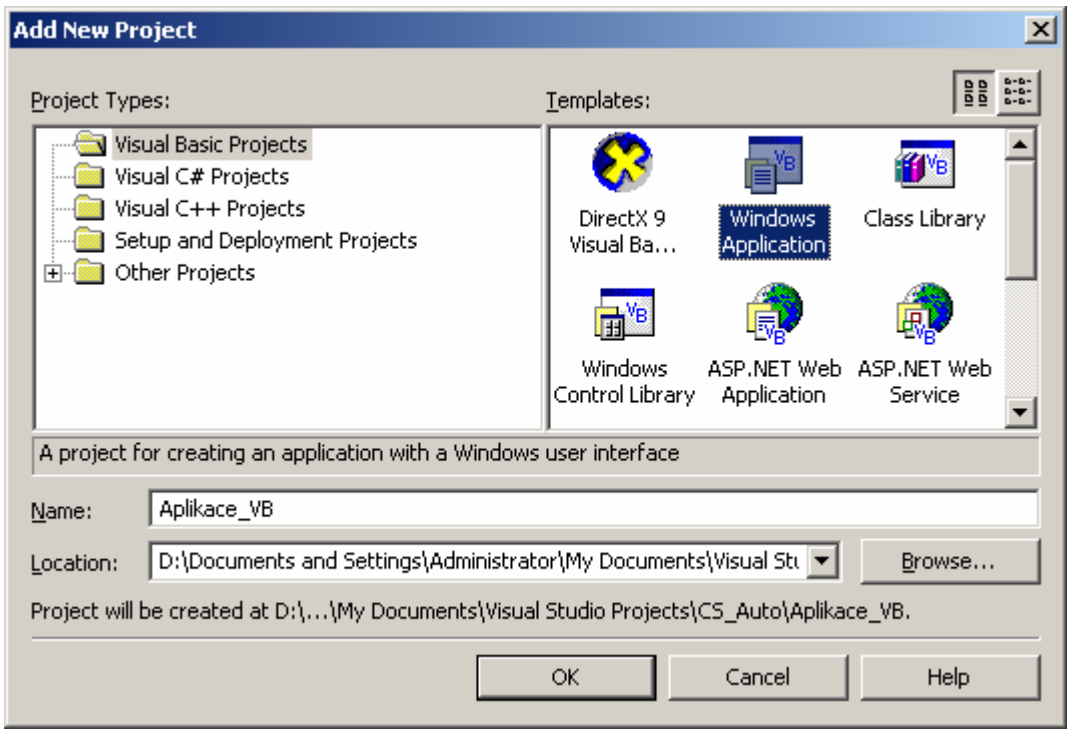
Po napsání programového kódu je načase přeložit projekt. Vyberte nabídku **Build** a aktivujte příkaz **Build Solution**.

Jestliže vše proběhlo úspěšně, můžete přistoupit k dalšímu kroku, kterým je přidání projektu VB .NET do stávajícího řešení a nadvázání komunikace s právě vytvořenou třídou.

Přidání projektu VB .NET do stávajícího řešení

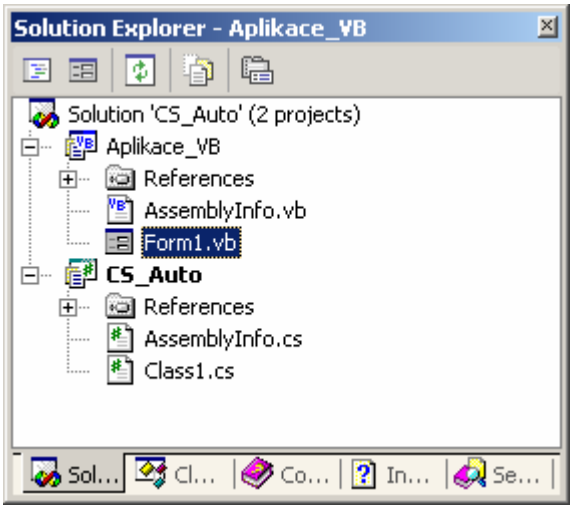
Postupujte podle instrukcí:

1. Otevřete nabídku **File**, ukažte na položku **Add Project** a vyberte příkaz **New Project**.
2. V sekci **Project Types** vyberte složku s názvem **Visual Basic Projects**. V rámečku **Templates** vyberte ikonu standardní aplikace pro Windows (**Windows Application**). Přidávaný projekt také vhodně pojmenujte.



Obr. 2 – Přidání projektu Visual Basicu do řešení

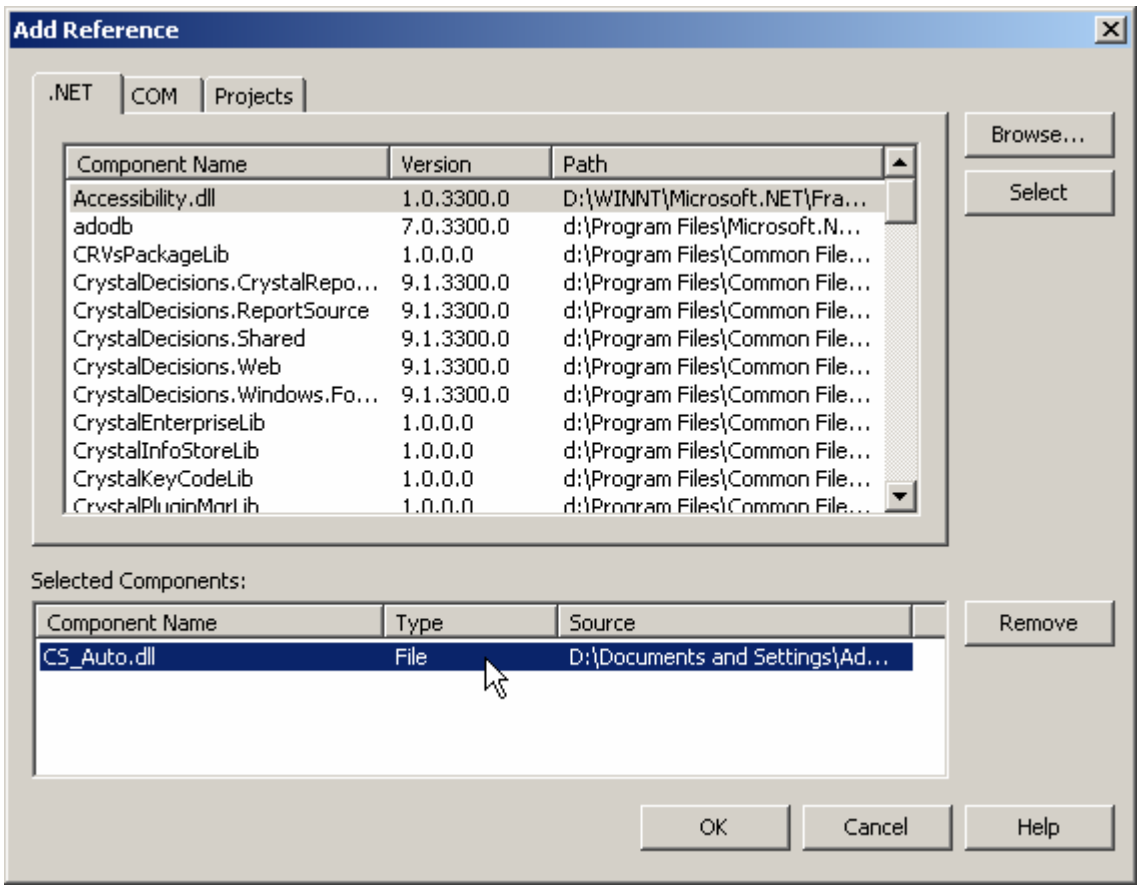
3. Obrazová struktura okna **Solution Explorer** by měla vypadat jako ta na obr. 3.



Obr. 3 – Podoba okna **Solution Explorer** po přidání projektu VB .NET

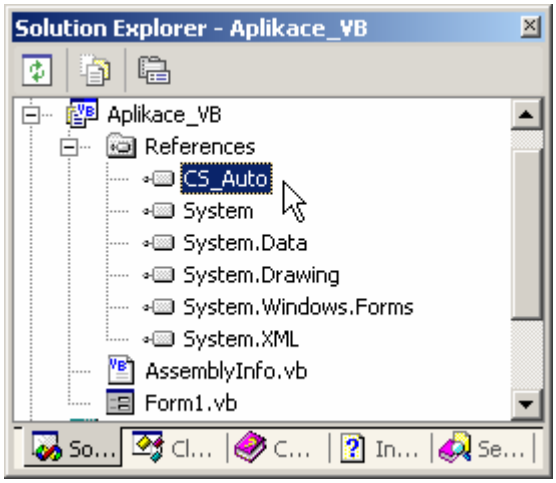
4. V okně **Solution Explorer** klepněte pravým tlačítkem myši na položku **References** projektu VB .NET a zvolte příkaz **Add Reference**.

5. V zobrazeném dialogovém okně se ujistěte, že je vybrána záložka **.NET**. Klepněte na tlačítko **Browse** a vyhledejte soubor DLL s kódem třídy, kterou jste vytvořili předtím v C#. Po vyhledání souboru a přidání odkazu na něj by mělo okno vypadat jako to na obr. 4.



Obr. 4 – Přidání reference na knihovnu DLL s kódem třídy **Automobil**

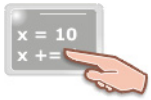
6. Nakonec klikněte na tlačítko **OK**. V tomto okamžiku se přidá odkaz na knihovnu DLL do projektu Visual Basicu. Tuto situaci můžete pozorovat v okně **Solution Explorer** (obr. 5).



Obr. 5 – Jmenný prostor **CS_Auto** v projektu VB .NET

7. V okně **Solution Explorer** ještě na chvíli zůstaneme. Aby bylo za chvíli možné spustit projekt VB .NET a otestovat chování třídy, musíte Visual Basicu povědět, aby jako startovací projekt nastavil právě projekt typu **Windows Application**. To uděláte tak, že na název projektu VB .NET klepněte pravým tlačítkem myši a z kontextuální nabídky vyberete příkaz **Set as StartUp Project**.

8. Pokračujte tím, že na formulář přidáte jednu instanci ovládacího prvku **Button**. Na vytvořenou instanci poklepejte, na což se vám zpřístupní okno editoru pro zápis programového kódu. Do těla událostní procedury **Button1_Click** zadejte tento fragment zdrojového kódu:



```
Dim ObjektAuto As New CS_Auto.Automobil()

MessageBox.Show("Peněžní vyjádření spotřeby PHM vašeho auta činí " & _
ObjektAuto.VypočístSpotřebu(100, 27)).ToString & " Kč.", _
"Ukázka míchání programovacích jazyků")
```

9. Spusťte řešení (klávesa **F5**) a klepněte na tlačítko **Button1**. Výslední efekt činnosti kódu můžete vidět na obr. 6.

Obr. 6 – Míchání programovacích jazyků v praxi

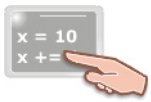
Implementace vlastnosti, která je jenom pro čtení

Při objektově orientovaném návrhu je někdy zapotřebí vytvořit vlastnost objektu, která bude pro vnější svět jenom pro čtení. Jestliže je vlastnost určena pouze pro čtení, programový kód klientské aplikace může pouze číst hodnotu dané vlastnosti, není ovšem schopen hodnotu vlastnosti jakkoliv měnit. Jak vytvořit vlastnost jenom pro čtení, si ukážeme v tomto tipu.

Postupujte takto:

1. Vytvořte novou aplikaci pro Windows (**Windows Application**).
2. Přidejte do aplikace soubor pro třídu. Soubor s třídou zařadíte do aplikace tak, že vyberete nabídku **Project** a aktivujete položku **Add Class**.

Předpokládejme, že budete chtít vytvořit třídu s názvem **Počítač**, která bude obsahovat vlastnost **TaktCPU**. Tato vlastnost bude určena jenom pro čtení. Programový kód, jenž předvádí tuto modelovou situaci je uveden níže:



```
Public Class Počítač
    Private ReadOnly m_TaktCPU As Integer
    Public ReadOnly Property TaktCPU() As Integer
        Get
            Return m_TaktCPU
        End Get
    End Property

    Public Sub New()
        m_TaktCPU = 1700
    End Sub
End Class
```

Každá vlastnost, která je určena jenom pro čtení, musí mít ve své deklaraci uvedeno klíčové slovo **ReadOnly**. Přesně stejná je situace i u vlastnosti **TaktCPU**. V těle vlastnosti **TaktCPU** je vytvořen jenom blok **Get-End Get**, který vrací hodnotu soukromého datového členu **m_TaktCPU**. Právě tento soukromý datový člen uchovává inicializační hodnotu, která reprezentuje takt procesoru v MHz a

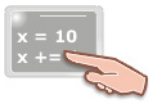
podobně jako samotná vlastnost i on musí mít v deklaračním příkazu uvedeno klíčové slovo **ReadOnly**.

Dobrá, doposud jsme si řekli, že vlastnost **TaktCPU** je určena jenom pro čtení a z tohoto důvodu nemá klientský kód oprávnění k modifikaci této vlastnosti. Jak ale potom nastavíme počáteční hodnotu této vlastnosti? Předem by bylo třeba říci, že samotná vlastnost neobsahuje hodnotu. Vlastnost je jenom programovací konstrukce, která umožňuje přistupovat k jisté hodnotě. Tato hodnota je ovšem uložena v soukromém datovém členu, resp. v proměnné datového typu **Integer** s názvem **m_TaktCPU**.

Vhodným okamžikem pro počáteční inicializaci datového členu **m_TaktCPU** je konstruktor třídy **Počítač**. Konstruktor ve VB .NET má vždy podobu veřejné procedury **Sub New**, která je standardně bez jakýchkoliv parametrů (pochopitelně, je možné vytvořit také konstruktor s parametry). V kódu konstruktoru dochází k inicializaci datového členu **m_TaktCPU** na hodnotu 1700. Vždy, když dojde k zrození instance této třídy, bude spuštěn kód konstruktoru, který inicializuje hodnotu soukromého datového členu. Po vytvoření instance třídy bude možné k hodnotě soukromého datového členu přistupovat pomocí vlastnosti **TaktCPU**.

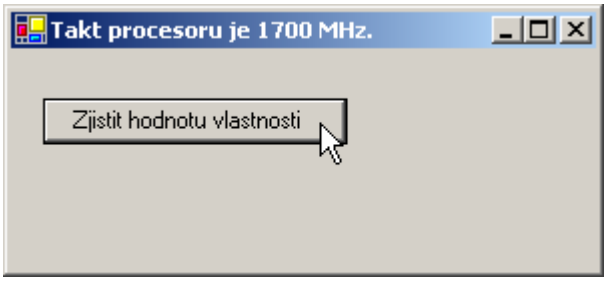
Předveďme si praktickou ukázku:

1. Přidejte na formulář instanci ovládacího prvku **Button** a programový kód jeho událostní procedury **Click** doplňte tímto způsobem:




```
Private Sub Button1_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles Button1.Click

    Dim MojePC As Počítač
    MojePC = New Počítač()
    Me.Text = "Takt procesoru je " & _
    MojePC.TaktCPU & " MHz."

End Sub
```

2. Po spuštění aplikace se hodnota vlastnosti **TaktCPU** objektu **MojePC** objeví v titulkovém pruhu okna (obr. 7).



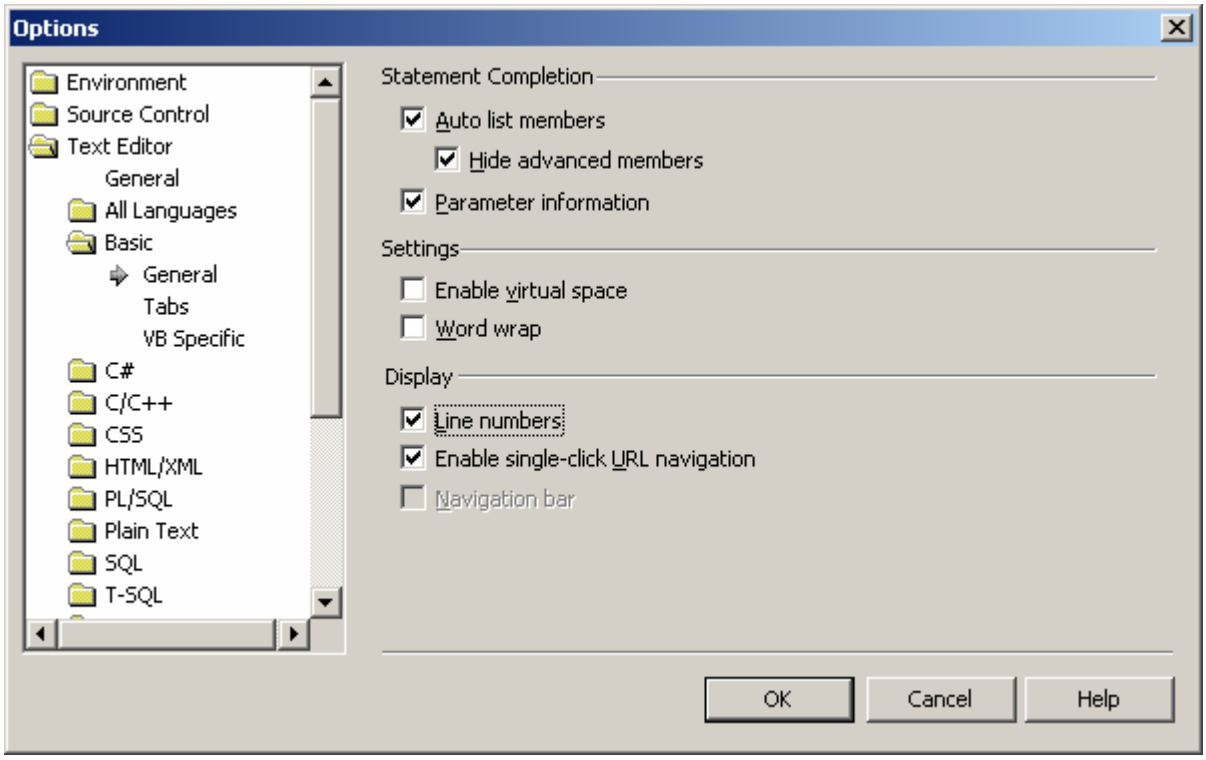
Obr. 7 – Zjištění hodnoty vlastnosti, která je určena jenom pro čtení

Zobrazení čísel řádků v editoru pro zápis kódu

Z vlastní zkušenosti vím, že některým programátorům se s zdrojovým kódem pracuje lépe v případě, kdy jsou všechny řádky kódu číselně označené. Visual Basic .NET samozřejmě podporuje možnost zobrazování čísel řádků programového kódu. Volba, která tuto schopnost IDE aktivuje je ovšem poněkud dobře ukryta. Jak se k ní dopracovat, si ukážeme právě teď:

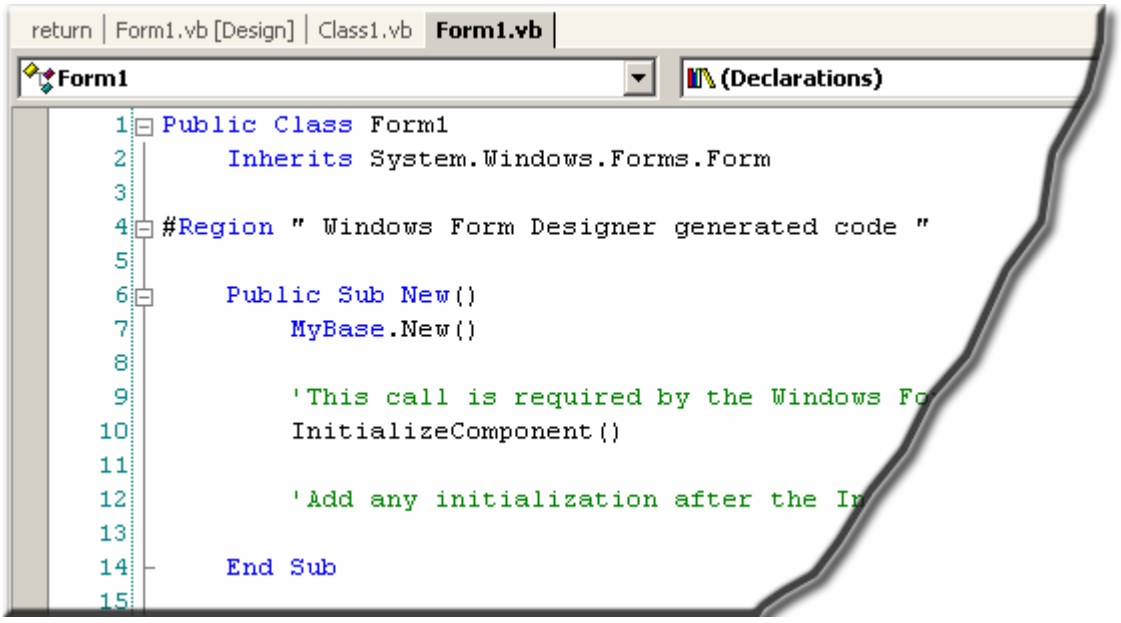
1. Vyberte menu **Tools** a klepněte na poslední položku **Options**.

2. V stromové struktuře složek v levé části okna **Options** klepněte na složku s názvem **Text Editor**.
3. Dále klepněte na podsložku **Basic** a ujistěte se, že je vybrána položka **General**.
4. V sekci **Display** zatrhněte volbu **Line numbers** (obr. 8) a aktivujte tlačítko **OK**.



Obr. 8 – Aktivace zobrazování čísel řádků v editoru pro zápis zdrojového kódu

Od této chvíle budou čísla řádků vaším nerozlučným pomocníkem při psaní programového kódu (obr. 9).



Obr. 9 – Čísla řádků v editoru pro zápis zdrojového kódu

Integrované prostředí rubriky Visual Basic	Autor: **Ján Hanák**

Téma měsíce

Distribuční jednotky (1. díl)

Časová náročnost (min):	Začátečník	Pokročilý	Profesionál
80			

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

Milí čtenáři,

zcela určitě není pochyb o tom, že proces rozmísťování aplikací, resp. tvorby pokročilých distribučních jednotek aplikací, má ve vývoji softwaru své nezastupitelné místo. I když aplikace spolehlivě plní všechny své funkce, právě od způsobu rozmísťování a posléze samotné instalace závisí skutečnost, zdali bude mít u finálního uživatele úspěch, či nikoliv. Jestliže pozorně sledujete naší programátorskou rubriku o Visual Basicu, jistě víte, že problematice tvorby jednoduché distribuční jednotky jsme se již věnovali. V dnešní lekci ovšem půjdeme mnohem dál a ukážeme si, jaké pokročilé možnosti nám Visual Basic .NET nabízí právě v oblasti tvorby a následné konfigurace instalačních projektů.

Obsah

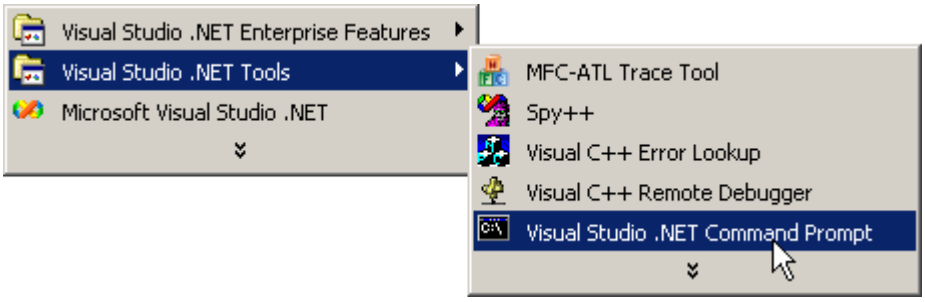
Rychlý začátek aneb příkaz XCOPY v praxi

Ne, nebojte se, nebudeme si demonstrovat použití tohoto příkazu v prostředí operačního systému MS-DOS, i když počátky vzniku příkazu **XCOPY** sahají do právě zmíněného prostředí. Skutečností ovšem je, že stejnojmenný příkaz se dostal také do příkazové řádky Visual Studia .NET. Význam a aplikace příkazu ve verzi .NET naproti všemu zůstává podobná: Pomocí příkazu **XCOPY** můžete snadno uskutečnit kopírovací proces, při kterém dojde k vytvoření kopie zdrojových dat (tedy dat, které se nacházejí v jisté složce, případně také ve vnořených podsložkách). Přestože jde o nejjednodušší cestu, jak přemístit zdrojové údaje do složky cílového počítače, je tato cesta plně funkční.

Uveďme si ihned malý příklad, abyste viděli, jak lze příkaz **XCOPY** použít v praxi. Povězme, že máme všechny potřebné soubory, které tvoří komplexní skeleton aplikace, sdruženy v jedné složce s názvem **Moje_Aplikace** na disku C. Aby bylo možné překopírovat obsah uvedené složky například do stejnojmenné složky, ovšem na disk D, udělejte toto:

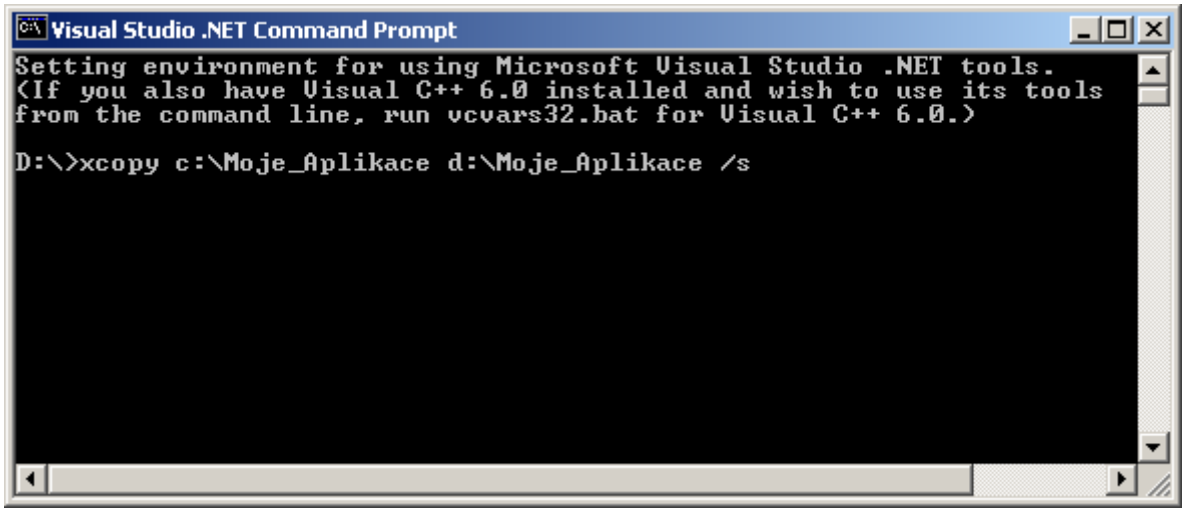
1. Vyberte nabídku **Start**, pokračujte přes menu **Programs** a ukažte na položku **Microsoft Visual Studio .NET**.

2. Vyberte nabídku **Visual Studio .NET Tools**.
3. Klepněte na položku s názvem **Visual Studio .NET Command Prompt** (obr. 1).



Obr. 1 – Spuštění příkazové řádky Visual Studia .NET

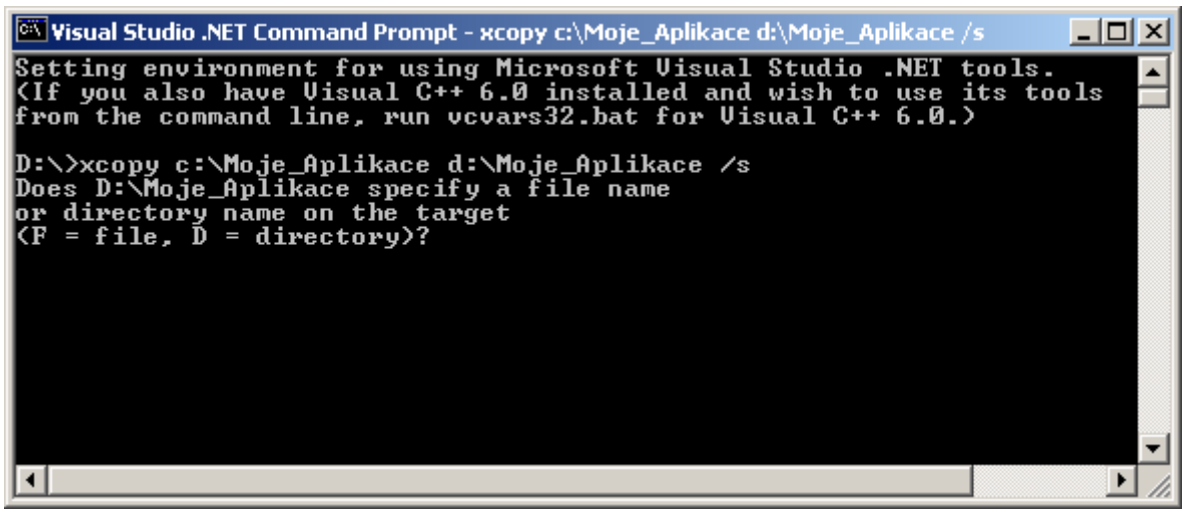
4. Okamžitě se na obrazovce zobrazí dialogové okno s příkazovou řádkou. Zde zadejte tuto podobu příkazu **XCOPY** (obr. 2):



Obr. 2 – Zadání příkazu **XCOPY** v příkazové řádce

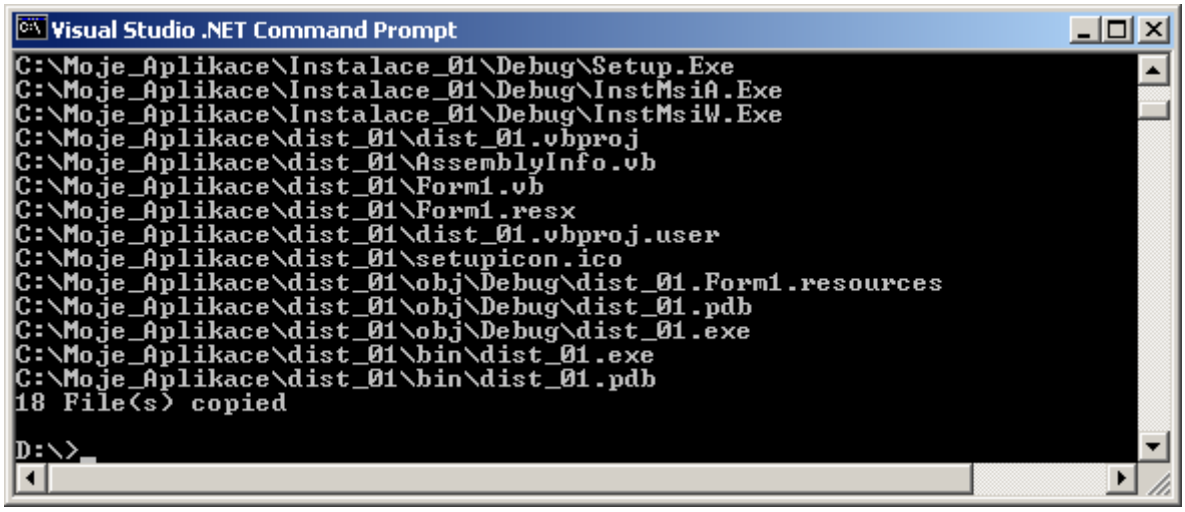
Vysvětleme si podrobněji celý zápis příkazu. Za jménem příkazu je specifikována cesta ke zdrojové složce, resp. k složce, která obsahuje všechny potřebné zdrojové soubory. Aby příkaz mohl správě pracovat, je ovšem nutné dodat i cestu k cílové složce (v našem případě má tato cesta podobu d:\Moje_Aplikace). Nakonec je zadán také příznak **/s**, jenž zabezpečí, aby byly přemístěny všechny podřízené složky hlavní složky.

5. Jestliže jste příkaz správně zapsali, stiskněte klávesu Enter. V tuto chvíli se vás příkaz zeptá na formu cílových dat, přesněji, zdali půjde o jednoduchý soubor nebo o složku se soubory (obr. 3).



Obr. 3 – Určení formy uložení cílových dat

6. Protože chceme, aby byla vytvořena složka a do ní následně překopírovány všechny potřebné soubory, stiskněme klávesu s písmenem D. Poté se obsah zdrojové složky překopíruje do cílové složky na disku D. Příkaz **XCOPY** vás po ukončení své práce také informuje o počtu přemístěných souborů (obr. 4).



Obr. 4 – Po ukončení kopírovacího procesu je zobrazena informace o počtu přemístěných souborů

Kromě příznaku **/s** disponuje příkaz **XCOPY** také spoustou jiných a velmi užitečných modifikátorů. Pokud si je chcete prohlédnout, zadejte do příkazové řádky příkaz **HELP XCOPY**.

Pravděpodobně budete souhlasit, když řeknu, že použití příkazu **XCOPY** je opravdu triviální. I když se jedná bezesporu o snad nejrychlejší metodu rozmísťování aplikací, je důležité mít na paměti několik skutečností, které se s tímto příkazem přímo vážou. Tak především, příkaz **XCOPY** můžete použít jenom na počítači s nainstalovaným prostředím .NET Framework. Tento požadavek je patrně nejvíc palčivý, ovšem není možné jej jakkoliv obejít, a proto je nutné ho respektovat. Druhý nárok souvisí se souborovou strukturou zdrojové složky (pokud tedy budete chtít kopírovat obsah složky a ne samotný soubor pochopitelně). Zdrojová složka musí obsahovat všechny potřebné soubory, které bude aplikace během své činnosti vyžadovat. Do množiny přípustných souborů lze zařadit spustitelné (.EXE) soubory, soubory s kódem dynamicky linkovaných knihoven (.DLL), zdrojové soubory a jakékoliv další nevyhnutné soubory. Z uvedeného jasně vyplývá, že distribuce aplikací pomocí příkazu **XCOPY** je vhodná jenom pro aplikace, které si všechna potřebná data „vezou s sebou“. Použití příkazu **XCOPY** není proto jistě namístě v případě, kdy je aplikace závislá od řady

externích a dynamicky se vyvíjejících faktorů (např. dostupnost databáze nebo sdílených komponent, o nichž v okamžiku použití příkazu **XCOPY** nevíme říci, zdali budou v budoucnu k dispozici a pod.).

Tvorba testovací aplikace a základního instalačního projektu

Dále budeme pokračovat vytvořením jednoduché testovací aplikace pro Windows, která bude otevírat a zobrazovat grafické obrázky ve formátu PNG. Po vytvoření nezbytné aplikační logiky přidáme do vygenerovaného řešení instalační projekt a ukážeme si, jaké možnosti rozšíření a pokročilé konfigurace nám nabízí IDE Visual Basicu .NET.

Zde je postup:

1. Jestliže jste tak ještě neučinili, spusťte Visual Basic .NET a přikažte vytvoření aplikace pro Windows (**Windows Application**). Testovací aplikaci pojmenujte jako **ProhlížečObrázků**.

2. Nastavte tyto vlastnosti implicitně vytvořeného formuláře:

Vlastnost	Hodnota
Name	frmProhlížeč
FormBorderStyle	FixedSingle
StartPosition	CenterScreen
Text	Prohlížeč obrázků (verze 1.0)

Protože jste změnili hodnotu vlastnosti **Name** formuláře, bude potřebné, abyste modifikovali také spouštěcí formulář. V okně **Solution Explorer** klepněte na název testovací aplikace pravým tlačítkem a z kontextové nabídky vyberte položku **Properties**. V dialogovém okně **Property Pages** se ujistěte, že je otevřena složka **Common Properties** a aktivována položka **General**. Poté z otevíracího seznamu **Startup object** vyberte položku se změněným názvem formuláře (**frmProhlížeč**).

3. Přidejte na formulář instanci ovládacího prvku **PictureBox**. Vlastnosti instance upravte takto:

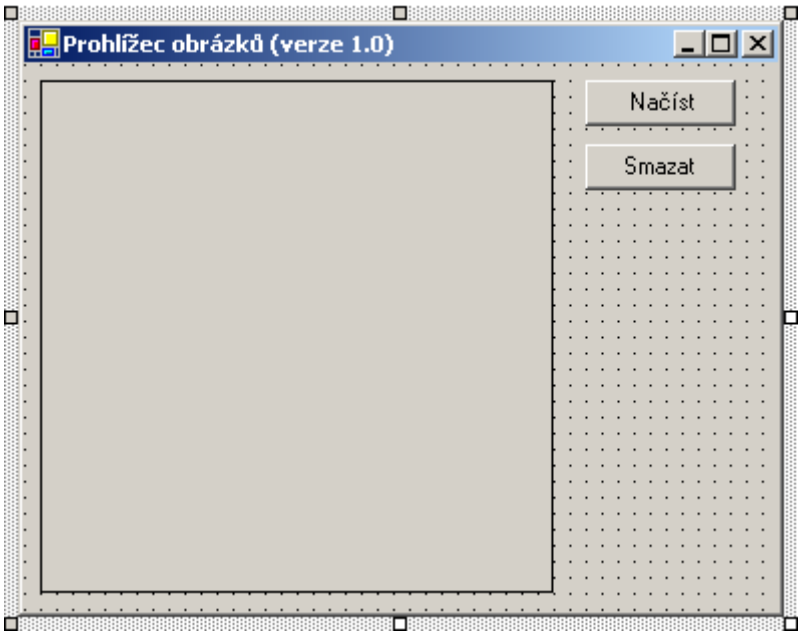
Vlastnost	Hodnota
BorderStyle	FixedSingle
Name	pctProhlížeč
SizeMode	StretchImage

Vlastnost **SizeMode** ovládacího prvku **PictureBox** determinuje způsob zobrazení obrázků po jejich načtení. Hodnota **StretchImage** říká, aby byl obrázek po načtení roztažen na celou plochu ovládacího prvku. Je patrné, že takto dojde k jisté degradaci obrazu, nakolik formát obrázku se musí přizpůsobit formátu zobrazovací plochy ovládacího prvku. Protože ale chceme vytvořit pouze jednoduchou testovací aplikaci, která nám bude složit jenom jako „podklad" pro tvorbu distribuční jednotky, budeme od této a podobných skutečností abstrahovat.

4. Umístěte na formulář dvě instance ovládacího prvku **Button** a jejich vlastnosti upravte takto:

Pořadí instancí	Vlastnost	Hodnota
1.	**Name**	btnNačíst
	Text	Načíst
2.	**Name**	btnSmazat
	Text	Smazat

Nyní by váš formulář mohl vypadat jako ten na obr. 5.



Obr. 5 – Vzhled testovací aplikace v režimu návrhu

Implementace aplikační logiky

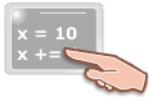
V této kapitole se zaměříme na napsání aplikační logiky. Protože nechceme vyvíjet profesionální aplikaci pro zpracování obrázků různých grafických formátů, bude potřebné ošetřit jenom dvě události:

1. Načtení obrázku na plochu ovládacího prvku
2. Smazání plochy ovládacího prvku

Ještě předtím, než budeme moci načíst obrázek na plochu ovládacího prvku, musíme vědět, jaký obrázek se má načíst. Řečeno jinými slovy, budeme potřebovat cestu k vybranému souboru, například v podobně c:\MojeAplikace\Obrázky\Obrázek1.PNG. Cestu k souboru můžeme získat prostřednictvím komponenty, která se nazývá **OpenFileDialog**.

Přicházíte-li z Visual Basicu verze 6, zanedlouho zjistíte, že komponenta **OpenFileDialog** je co do funkčnosti a stylu práce velmi podobná komponentě **CommonDialog**.

Vyhledejte proto ikonu této komponenty v soupravě nástrojů (**Toolbox**) a poklepejte na ní. Vzápětí se instance komponenty (standardně pojmenovaná jako **OpenFileDialog1**) přidá na podnos komponent (**Component Tray**). Poklepejte na tlačítko s textem **Načíst** a vyplňte jeho událostní proceduru **Click** níže uvedeným kódem:



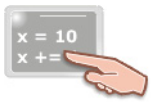

```
With OpenFileDialog1
    .Filter = "Grafické soubory (*.PNG)|*.PNG"
    .ShowDialog()
End With
```

Programový kód pokračuje na následující straně

```
If Len(OpenFileDialog1.FileName) <> 0 Then
    Dim obr As New Bitmap(OpenFileDialog1.FileName)
    pctProhlížeč.Image = obr
Else
End If
```

V bloku **With-End With** je nastavena vlastnost **Filter** instance komponenty. Tato vlastnost určuje, jaké typy souborů se budou zobrazovat v dialogovém okně pro vyhledání grafických souborů. Dále je volána metoda **ShowDialog**, která má na starosti zobrazení již vzpomínaného dialogového okna. V dalším kroku je průběh aplikace přenesen na uživatele, který má možnost procházet všechny disky a jejich složky a nalézt ten grafický soubor, jenž chce otevřít. Když uživatel označí kýžený soubor a klepne na tlačítko **Otevřít**, je cesta k tomuto souboru uložena do vlastnosti **FileName** komponenty **OpenFileDialog1**. Když se návratová hodnota funkce **Len** nerovná nule, znamená to, že uživatel vybral jistý grafický soubor. Dále je vytvořen objekt **obr** třídy **Bitmap** na základě cesty k souboru (textový řetězec určující cestu k souboru představuje argument, jenž je předán konstruktoru třídy **Bitmap**). Odkaz na zrozený grafický objekt je uložen do vlastnosti **Image** ovládacího prvku **PictureBox1**, na což se data grafického souboru zobrazí na ploše tohoto prvku.

Událostní proceduru **Click** tlačítka **btnSmazat** upravte takto:




```
pctProhlížeč.Image = Nothing
OpenFileDialog1.FileName = ""
```

Obrázek z plochy ovládacího prvku **PictureBox1** odstraníme tak, že do vlastnosti **Image** tohoto prvku přiřadíme hodnotu **Nothing**. Poté tedy vlastnost **Image** nebude obsahovat odkaz na žádný objekt. Druhým krokem je „reset" vlastnosti **FileName** komponenty **OpenFileDialog1**.

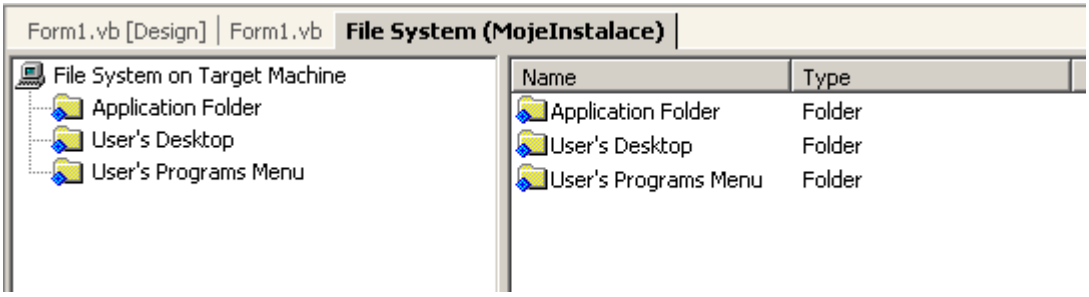
Budete-li chtít, můžete projekt sestavit a vyzkoušet jeho funkčnost. Po vybraní souboru s obrázkem by se tento měl objevit na ploše ovládacího prvku. Výborně, přidání aplikační logiky jste zvládli, můžeme popojet zase o krok dál.

Zařazení instalačního projektu

V tomto okamžiku přidáme do stávajícího řešení instalační projekt a zadáme základní informace pro sestavení plně funkční distribuční jednotky. Postupujte podle těchto instrukcí:

1. Otevřete nabídku **File**, ukažte na položku **Add Project** a klepněte na příkaz **New Project**.
2. V dialogovém okně **Add New Project** vyberte složku **Setup and Deployment Projects** a klepněte na ikonu s názvem **Setup Wizard**.
3. Projekt pojmenujte jako **MojeInstalace**.
4. Objeví se průvodce, který vám pomůže se sestavením instalačního projektu. Ve druhém kroku se ujistěte, že je vybrána možnost **Create a setup for a Windows application**.
5. Ve třetím kroku zatrhněte volbu **Primary output from ProhlížečObrázků**. Pokračujte až do konce, přičemž nemusíte do projektu pro instalaci přidávat žádné dodatečné soubory, na které budete dotázáni ve 4. kroku průvodce.

Kostra instalačního projektu je tímto pádem úspěšně vytvořena. Na obrazovce počítače byste měli v tuto chvíli vidět otevřené okno editoru s názvem **File System** (obr. 6).



Obr. 6 – Otevřený editor **File System**

Charakteristika editoru File System

Jak si můžete na obrázku všimnout, editor **File System** je rozdělen do dvou polí. V levém poli je zobrazena struktura adresářů na cílovém počítači. Zde můžete vidět tři složky:

- **Application Folder** - reprezentuje složku, ve které budou nainstalovány základní soubory aplikace.
- **User´s Desktop** – zobrazuje plochu cílového počítače.
- **User´s Programs Menu** – je představitelkou obsahu nabídky **Programs** v nabídce **Start** cílového počítače.

Klepnete-li na složku **Application Folder**, v pravém poli se objeví obsah této složky (nachází se zde jenom jedna položka, která reprezentuje spustitelný soubor testovací aplikace). Klepněte na tento soubor pravým tlačítkem myši a z kontextového menu vyberte příkaz **Create Shortcut to Primary output from ProhlížečObrázků (Active)**. Visual Basic .NET vytvoří zástupce, kterého pojmenujte jako **Zástupce aplikace**. Zástupce poté můžete přetáhnout do jedné ze složek **User´s Desktop** a **User´s Programs Menu**, jež se nacházejí v levém poli.

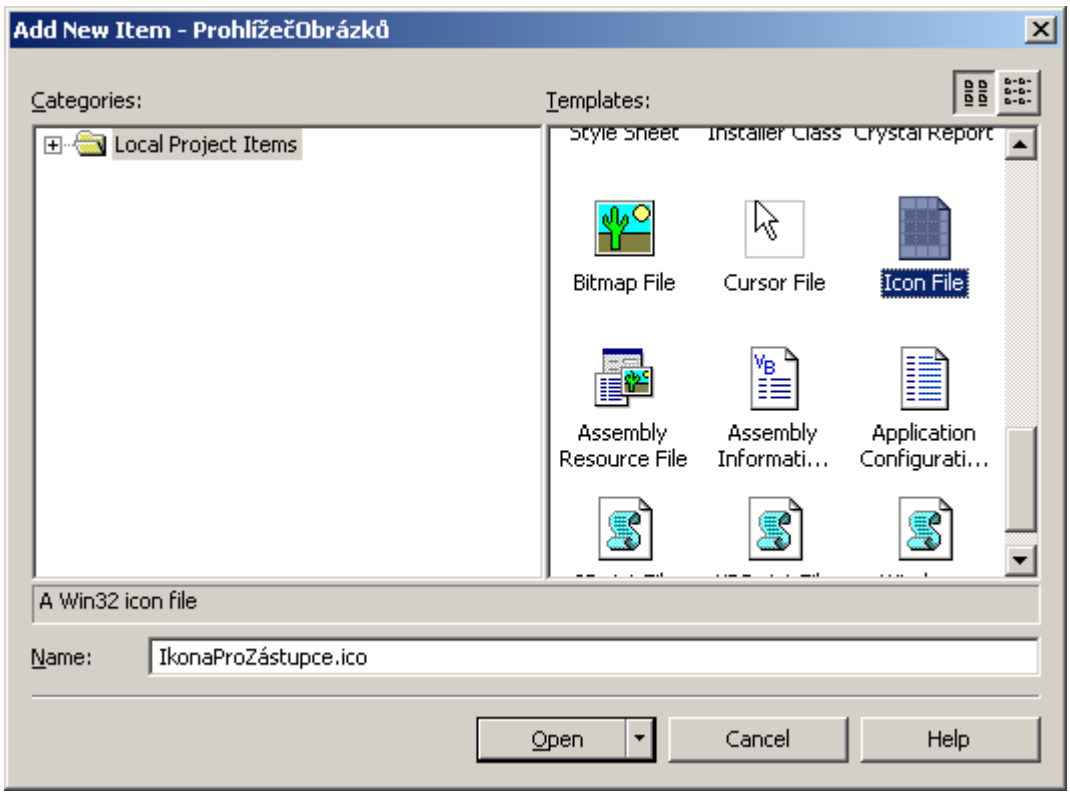
Pokud budete chtít umístit zástupce do obou složek, budete muset vytvořit ještě jednoho zástupce, a to podobným způsobe, jakým jste vytvořili prvního zástupce (tedy klepnutím pravým tlačítkem myši na ikonu **Primary output from ProhlížečObrázků (Active)** a zvolením příkazu **Create Shortcut to ...**).

Nejste-li spokojeni se vzhledem standardní ikony pro zástupce, můžete ji změnit. Jak na to si ukážeme v další kapitole.

Tvorba uživatelské ikony pro zástupce aplikace

V okně **Solution Explorer** klepněte na název testovací aplikace pravým tlačítkem myši a z kontextuální nabídky vyberte položku **Add** a poté **Add New Item**.

V dialogovém okně **Add New Item** vyhledejte položku, která představuje soubor s ikonou podle specifikací Win32 (obr. 7). Vytvářený soubor s ikonou pojmenujte jako **IkonaProZástupce**.



Obr. 7 – Přidání souboru s ikonou do projektu testovací aplikace

Soubor s ikonou je okamžitě otevřen v IDE, takže můžete neprodleně začít s návrhem vzhledu ikony. K dispozici jsou vám podpůrné nástroje pro výběr a modifikaci různých grafických elementů. Předpokládejme, že jste již ikonu nakreslili. Dále pokračujte takto:

1. Uložte soubor s ikonou klepnutím na ikonu diskety na panelu s tlačítky.
2. Přepněte se zpět do editoru **File System**.
3. Klepněte na složku, do které jste umístili položku se zástupcem.
4. Označte položku se zástupcem a v okně **Properties Window** vyhledejte vlastnost **Icon**.
5. Ze seznamu vyberte položku **Browse**.
6. V dialogovém okně **Icon** klikněte opět na tlačítko **Browse**.
7. Objeví se okno **Select Item in Project**. Zde poklepejte na složku **Application Folder**.
8. Pokračujte aktivací tlačítka **Add File**.
9. Vyhledejte váš soubor s ikonou a přidejte jej do instalačního projektu.
10. Potvrďte vykonané změny a zavřete všechna dialogová okna.

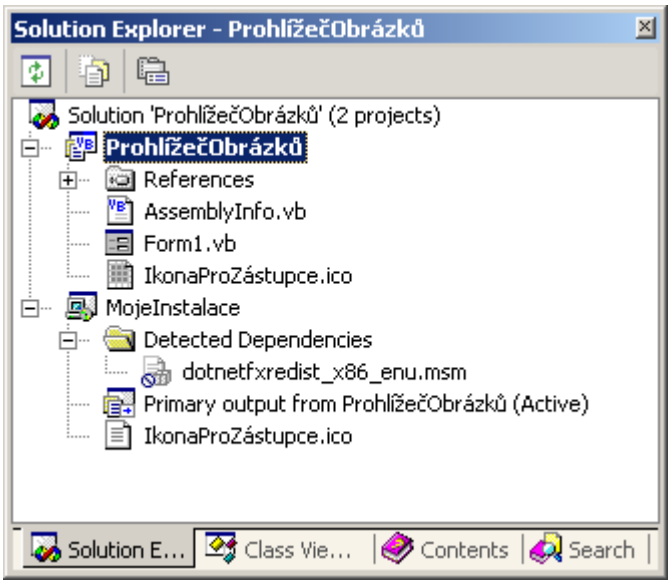
Proveďte kompilaci instalačního projektu (**Build** > **Build MojeInstalace**) a vygenerujte platné instalační soubory.

Pokud je to nutné, můžete uskutečnit také kompilaci testovací aplikace.

Spusťte instalaci klepnutím na soubor **Setup.exe** a testovací aplikaci nainstalujte. Uvidíte, že po ukončení instalace se na ploše objeví vámi navržená ikona zástupce aplikace.

Dialogové okno Solution Explorer pod drobnohledem

Podívejme se blíže na okno **Solution Explorer** (obr. 8):



Obr. 8 – Okno **Solution Explorer** pod drobnohledem

Samotnou testovací aplikaci si teď příliš nevšímejte, raději zaměřte svou pozornost na instalační projekt s názvem **MojeInstalace**. V stromové struktuře instalačního projektu se nachází uzel **Detected Dependencies**. Ten obsahuje reference na externí součástí, od kterých je instalační projekt více či méně závislý. V našem případě se zde nachází jenom jeden odkaz (**dotnetfxredinst_x86_enu.msm**), který „ukazuje" na instalační soubory .NET Frameworku. Tyto soubory jsou implicitně vyčleněny z instalačního projektu, nakolik jsou značně náročné na diskovou kapacitu. Jestliže ovšem nebude mít jistotu, že na cílovém počítači je .NET Framework nainstalován, měli byste tuto referenci do instalačního projektu zahrnout. V tom případě ale buďte připraveni na skutečnost, že se markantně zvýši kapacitní náročnost instalačních souborů.

Soubory .NET Frameworku přidáte do projektu takto:

1. Klepněte na položku **dotnetfxredinst_x86_enu.msm** pravým tlačítkem myši.
2. V kontextové nabídce zrušte zatržení u položky **Exclude**.

Kromě uzlu **Detected Dependencies** můžeme v rámci instalačního projektu pozorovat další dvě položky; jedna z nich patří přímo spustitelnému (.EXE) souboru aplikace a druhá reprezentuje uživatelskou ikonu pro zástupce aplikace.

Ukázka základních konfiguračních možností instalačního projektu

Na závěr si ještě probereme některé zajímavé vlastnosti, které můžete v zájmu zlepšení uživatelské přívětivosti a atraktivnosti instalačního projektu změnit. Seznam všech vlastností se zpřístupní v okně **Properties Window** po klepnutí na ikonu instalačního projektu v okně **Solution Explorer**.

Přehled některých užitečných vlastností i s komentářem můžete nalézt v následující tabulce:

Název vlastnosti	Charakteristika
AddRemoveProgramsIcon	Specifikuje ikonu, která se bude zobrazovat v dialogovém okně pro přidání resp. odebrání programových součástí.
Author	Sdružuje informace o autorovi aplikace.
Description	Nabízí vhodný prostor pro stručný popis práce a účelu aplikace.
Manufacturer	Podává informace o výrobci aplikace. Zde uvedené informace jsou ve většině případů totožné s informace uvedenými ve vlastnosti **Author**.
ManufacturerURL	Determinuje adresu webových stránek výrobce.
ProductName	Obsahuje jméno softwarového produktu.
RemovePreviousVersion	Nařizuje, zdali má instalátor při instalaci odstranit předcházející verzi softwarového produktu (jestliže je tato verze detekována).
SupportPhone	Specifikuje telefonní číslo technické podpory softwaru.
SupportURL	Obsahuje webovou adresu, na které můžou uživatelé najít dodatečné informace technického charakteru, nebo zodpovězené nejčastěji kladené dotazy.
Title	Textový řetězec, jenž zobrazuje instalátor.

Integrované prostředí rubriky Visual Basic	Autor: **Ján Hanák**

Lab

Téma

Začínáme s VB .NET

Úvod do světa .NET (8. díl)

Časová náročnost (min):	Začátečník	Pokročilý	Profesionál
95	1	2	3

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

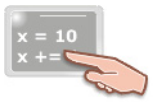
Vážení čtenáři,

již osmým dílem pokračuje váš oblíbený seriál o představování programovacího jazyka Visual Basic .NET. Nabídka dnešní části bude vskutku výtečná. Nejprve se podíváme na praktickou ukázkou práce dvou způsobů předávání argumentů, a poté se budeme soustředit na pochopení konstant a enumeračních typů. Závěr bude patřit prezentaci možností voleb **Option Explicit** a **Option Strict**.

Obsah

Praktická ukázka práce dvou variant předávání argumentů

Pokud jste absolvovali naše předchozí sezení, už víte, co se ukrývá pod pojmy argument a parametr. Rovněž tak víte, co je to funkce a jak ji použít. Jak jsme si řekli, argumenty lze předávat příslušným parametrům buď odkazem nebo hodnotou. Jenom pro rychlé zopakování si připomeňme, že při předávání argumentů hodnotou je volané proceduře nebo funkci předána pouze kopie skutečného argumentu, a tak, i když se bude tato procedura snažit sebevíc, nikdy se jí nepodaří změnit hodnotu skutečného argumentu. Na druhé straně, při předávání argumentů odkazem je situace zcela jiná. V tomto případě je totiž volané proceduře předán ukazatel na paměťové místo, na kterém se skutečný argument nachází. Protože procedura „ví", kde je uložena hodnota, je schopna tuto hodnotu snadno modifikovat. Dobrá, teorii jsme probrali a teď se společně vrhněme na praktickou demonstraci. Celou situaci si ukážeme na níže uvedeném programovém kódu třídy formuláře **Form1**:




```
Public Class Form1
    Inherits System.Windows.Forms.Form

Windows Form Designer generated code
```

Programový kód pokračuje na následující straně

```
    Private Sub Button1_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Button1.Click
        Dim a, b As Integer
        a = 100
        b = 150
        Součin(a, b)
        MessageBox.Show("Hodnota proměnné a: " & a.ToString & _
        vbCrLf & "Hodnota proměnné b: " & b.ToString)
    End Sub

    Private Sub Součin(ByVal parametr1 As Integer, _
    ByRef parametr2 As Integer)
        parametr1 = parametr1 * parametr2
        parametr2 = parametr1 * parametr2
    End Sub
End Class
```

Z uvedeného fragmentu zdrojového kódu je zřejmé, že předmětem našeho zkoumání bude událostní procedura **Click** tlačítka **Button1** a soukromá (**Private**) procedura **Součin**, která vypočítává součin argumentů svých parametrů. Abyste mohli pečlivě sledovat tok programu, probereme si všechny řádky po jednotlivých krocích.

Dobrodružství v událostní proceduře Button1_Click

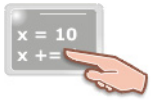
V těle událostní procedury jsou deklarovány dvě proměnné (**a**, **b**) datového typu **Integer**. Dále jsou obě proměnné inicializovány prostřednictvím přiřazovacího příkazu. Do proměnné **a** je uložena hodnota 100 a do proměnné **b** hodnota 150. Ve třetím kroku je zavolána procedura **Součin**, které jsou jako argumenty poskytnuty hodnoty proměnných **a** a **b**. V tomto okamžiku se tok programu přemísťuje do procedury **Součin**.

Dobrodružství v proceduře Součin

Procedura **Součin** je soukromá, což znamená, že jí nelze použít jinde než právě v třídě formuláře **Form1** (říkáme také, že viditelnost procedury je omezena na obor třídy formuláře). Velmi důležitým prvkem, který hraje roli později při vykonávání programového kódu v těle procedury, je způsob jakým jsou proceduře **Součin** předány vstupné argumenty. Aby procedura vůbec mohla být schopna nějaké argumenty přijmout, musí mít za tímto účelem deklarovány speciální proměnné, jimž se říká parametry (nebo také formální parametry). Jak si můžete ze zápisu signatury procedury všimnout, procedury má dva parametry, pojmenované jako **parametr1** a **parametr2**. Před prvním parametrem se nachází klíčové slovo **ByVal**, které specifikuje, že tomuto parametru bude předán argument hodnotou. Naproti tomu, název druhého parametru předchází klíčové slovo **ByRef**, které oznamuje překladači, že tento parametr očekává argument, jenž bude předán odkazem.

Když je ve volající událostní proceduře **Button1_Click** volána procedura **Součin**, jsou ji předány dvě hodnoty. Jak se s těmito hodnotami vypořádá procedura **Součin**? První argument je předán hodnotou, což znamená, že proceduře je poskytnuta pouze kopie skutečné hodnoty. Ve skutečnosti je tedy hodnota proměnné **a** (100) zkopírována a následně poskytnuta proceduře **Součin**. Procedura tak nebude pracovat se skutečnou hodnotou, nýbrž právě s touto kopii. Radikálně jiná je ovšem situace při druhém argumentu, ten je totiž předáván odkazem. Tedy, hodnota proměnné **b** (150) je přímo poskytnuta proceduře **Součin** (přesněji je uložena do formálního parametru s názvem **parametr2**).

V samotném těle procedury se nacházejí dva přiřazovací příkazy:



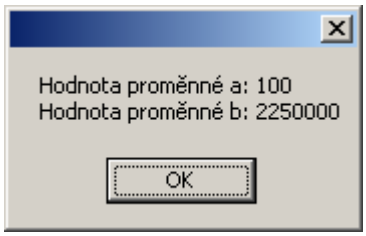

```
parametr1 = parametr1 * parametr2
parametr2 = parametr1 * parametr2
```

Co dělají? Vypočítávají součin svých argumentů. V prvním řádku je vypočten součin hodnot 100 a 150, a do proměnné **parametr1** je tedy uložena hodnota 15000. Ve druhém řádku je vypočten součin čísel 15000 (což je hodnota proměnné **parametr1**) a 150. Výsledkem je číslo 2250000, které je posléze uloženo do proměnné **parametr2**.

Otázkou ovšem zůstává, jak se změny, provedené na argumentech v proceduře **Součin** odrazily na podobě skutečných argumentů.

Vyřešení záhady v událostní proceduře Button1_Click

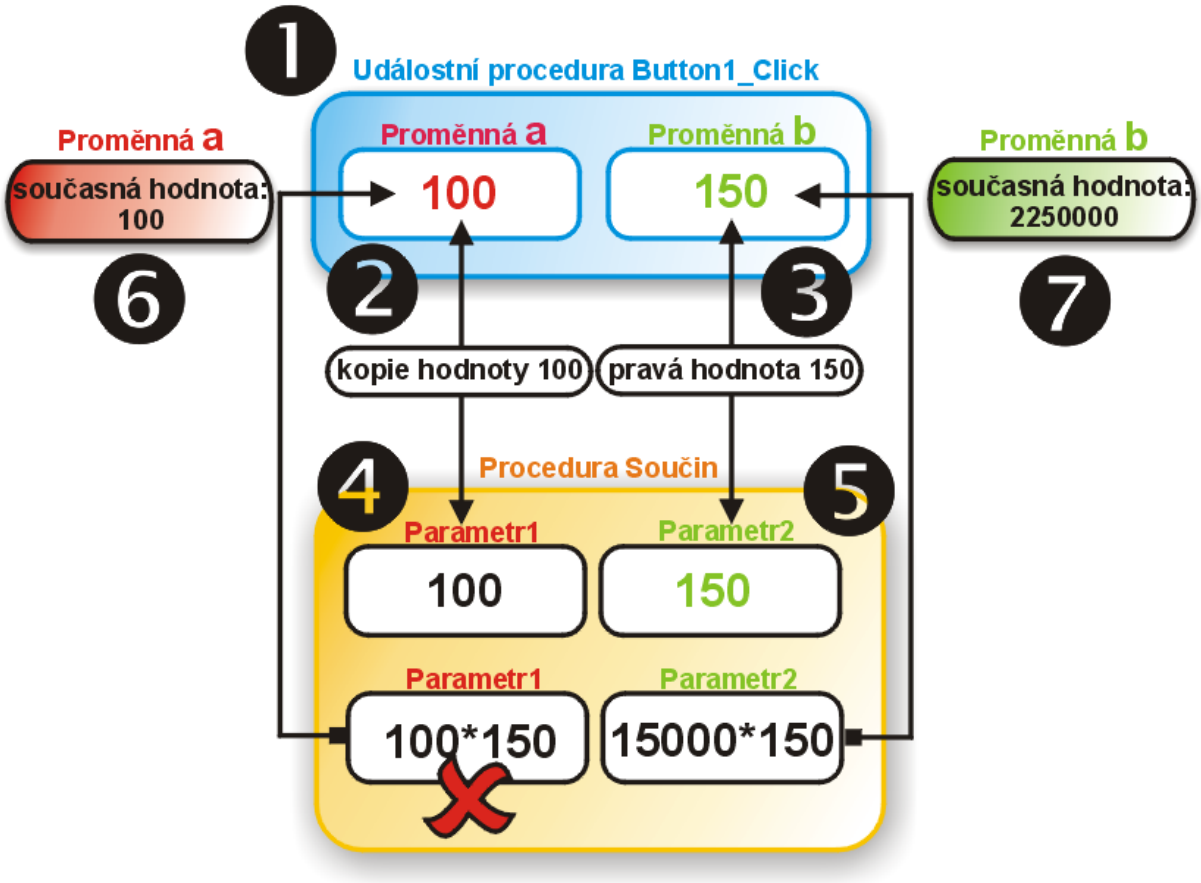
Celou pravdu nám zprostředkuje až metoda **Show** třídy **MessageBox**, která zobrazí dialogové okno se současnými hodnotami proměnných **a** a **b**. Podívejme se nejprve na výsledek a pak si k němu řekneme několik slov (obr. 1).



Obr. 1 – Skutečné hodnoty proměnných po exekuci procedury **Součin**

Ano, jak se můžete z obrázku přesvědčit, hodnota proměnné **a** je rovna 100 jednotkám. Je to pochopitelné, protože průběh procedury **Součin** neměl na hodnotu této proměnné žádný vliv (a ani nemohl mít, protože procedura **Součin** nepracovala se skutečnou hodnotou proměnné **a**, ale jenom s kopii této skutečné hodnoty). Hodnota proměnné **b** je přesně stejná, jakou jsme vypočítali při druhém přiřazovacím příkazu v proceduře **Součin**. Hodnota (argument) 150 byl proceduře **Součin** předán odkazem, byl tedy předán ukazatel na tuto hodnotu. Procedura **Součin** poté tuto skutečnou hodnotu změnila na 2250000. Právě tato hodnota je zobrazena v dialogovém okně se zprávou.

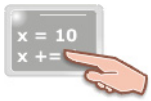
Komplexní mechanizmus graficky znázorňuje obr. 2.



Obr. 2 – Algoritmus práce předávání argumentů různými způsoby

Konstanty

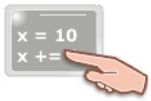
Konstanty jsou, jednoduše řečeno, proměnné, které nemění svou hodnotu. Dobrá, uznávám, že předchozí tvrzení je přinejmenším podivné, ovšem ve skutečnosti pravdivé. Konstanty jsou proměnné, do kterých ovšem můžete uložit jenom jednu hodnotu. Tato hodnota musí být do konstanty uložena ještě v režimu návrhu aplikace (teda ne až za běhu aplikace, jak je to obvyklé u proměnných). Konstantu deklarujeme pomocí příkazu **Const** takto:




```
Private Sub Button2_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles Button2.Click

    Const MojeKonstanta As Short = 1200

End Sub
```

Za příkazem **Const** následuje jméno pro konstantu, specifikace datového typu a inicializační hodnota. Konstanty jsou typické právě přiřazením oné inicializační hodnoty, bez této hodnoty by totiž nešlo o konstantu. V uvedeném výpisu zdrojového kódu se nachází deklarace konstanty uvnitř událostní procedury **Button2_Click**. Nachází-li se deklarace konstanty v proceduře **Sub**, ve funkci (**Function**) nebo v definici vlastnosti (**Property**), je tato konstanta použitelná jenom v oboru procedury, resp. funkce v rámci které je deklarována (obor konstanty je tak poměrně malý). V praxi se obor konstant zvětšuje, a to dosti radikálním způsobem. Konstanty jsou většinou deklarovány v oboru třídy nebo modulu, aby mohly být přístupné značné množině procedur a funkcí.

Použití konstanty je velmi jednoduché a téměř totožné s použitím libovolné proměnné. Zásadní rozdíl tkví v tom, že hodnotu konstanty můžete jenom číst, zatímco hodnotu proměnné můžete také modifikovat. Malý příklad za všechny:



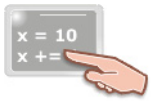
```
Private Sub Button2_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles Button2.Click
    Const Zrychlení As Single = 9.81
    Dim m As Short, F As Integer
    m = 100
    F = m * Zrychlení
    MessageBox.Show("Výsledek: " & F & " N.")
End Sub
```

Zařazení konstant do programového kódu má mnohé výhody. Konstanty nejenže zpřehledňují zdrojový kód a zlepšují jeho čitelnost, ale také vám poskytují možnost rychlé změny hodnoty konstanty (v režimu návrhu aplikace) a tato změna se okamžitě projeví na všech místech, kde se zápis konstanty vyskytuje.

Kromě celočíselných konstant a konstant obsahujících čísla s plovoucí desetinnou čárkou, můžete deklarovat také konstanty řetězcové (datový typ **String**) a znakové (datový typ **Char**).

Enumerační typy

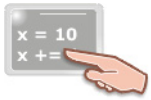
Enumerační, nebo také výčtové typy, si můžete představit jako sadu několika konstant, která tvoří jednotný celek. Jeden enumerační typ může obsahovat množství členů neboli konstant. Samotný člen má své vlastní jméno a je mu také přiřazena inicializační hodnota. Všechny enumerační typy jsou deklarovány pomocí příkazu **Enum**. Na rozdíl od konstant, enumerační typy nesmí být deklarovány uvnitř procedury, zato jejich deklarace může být uvedena na úrovni třídy, modulu nebo struktury. Jak vypadá deklarace výčtového typu si ukážeme právě teď:



```
Module Module1
    Public Enum Měsíce
        Leden = 1
        Únor = 2
        Březen = 3
        Duben = 4
        Květen = 5
        Červen = 6
    End Enum
End Module
```

Uvedený deklarační příkaz nařizuje vytvoření výčtového typu s názvem **Měsíce**. Jednotlivé členy enumerace se nacházejí v těle příkazu. Jak si můžete všimnout, náš výčtový typ obsahuje šest členů pro stejný počet měsíců v roce. Každý člen je inicializován na jistou hodnotu.

Použití enumeračního typu je opravdu jednoduché:



```
Dim a As Byte
a = Měsíce.Duben
Me.Text = a
```

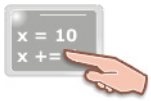
Zde je deklarována proměnná a datového typu **Byte** a následně je do této proměnné přiřazena hodnota členu **Duben** výčtového typu **Měsíce**. Modifikovaná hodnota proměnné **a** je zobrazena v titulkovém pruhu hlavního okna aplikace.

Jestliže zadáte za jménem výčtového typu tečkový operátor, IntelliSense vám nabídne seznam platných členů daného enumeračního typu:

```
Private Sub Button3_Click
    Dim a As Byte
    a = Měsíce.
    Me.Text
End Sub
```

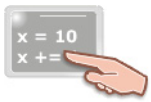
Březen
Duben
Květen
Leden
Únor
Červen

Implicitně jsou členy výčtových typů deklarovány s použitím datového typu **Integer** (tak je to i v našem případě). Jestliže chcete, můžete výslovně určit i jiný podporovaný datový typ (**Byte**, **Short** a **Long**) enumerace a všech jejich členů, například takto:



```
Public Enum Barvy As Byte
    Červená = 10
    Modrá = 11
    Zelená = 12
End Enum
```

Pokud explicitně nezadáte inicializační hodnoty členů enumerace, budou jim přirazeny výchozí inicializační hodnoty, začínající od nuly:



```
Public Enum Barvy As Byte
    Červená
    Modrá
    Zelená
End Enum
```

V tomto případě mají enumerační členy tyto hodnoty:

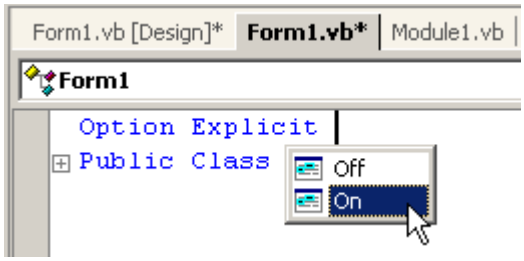
- Červená = 0
- Modrá = 1
- Zelená = 2

Charakteristika voleb Option Explicit a Option Strict

Programátoři, kteří přicházejí z Visual Basicu verze 6, jistě vědí, co znamená volba **Option Explicit** a za jakým účelem se také používá. Pro ty z vás, kteří neměli tu čest pracovat s nižší verzí jazyka Visual Basic, jsou určeny následující řádky.

Volba **Option Explicit** určuje, zdali budete muset ve svém zdrojovém kódu explicitně deklarovat všechny proměnné či nikoliv. Jestliže je volba nastavena na hodnotu **On**, Visual Basic .NET bude zjišťovat, jestli jste deklarovali všechny proměnné pomocí deklaračního příkazu **Dim**. V opačném případě, je-li hodnota volby **Off**, což je implicitní nastavení, není nutné, abyste všechny proměnné přímo deklarovali. Nedeklarované proměnné mají přesto svůj datový typ, je jím datový typ **Object**.

Zápis volby **Option Explicit** (a podobně také volby **Option Strict**) se musí ve Visual Basicu .NET nacházet před veškerým zdrojovým kódem daného souboru. Tedy v souboru **Form1.vb**, jenž obsahuje kód třídy formuláře, musí být zápis volby umístěn na prvním řádku (ještě před příkazem **Imports**, nebo před příkazem pro deklaraci třídy). Tento moment zachycuje obr. 3.



Obr. 3 – Nastavení volby **Option Explicit** na hodnotu **On**

Volba **Option Strict** má s volbou **Option Explicit** několik společných formálních rysů. Tak především, pro zápis volby platí stejná pravidla, jak tomu bylo u volby **Option Explicit**. Nastavování hodnoty volby je rovněž velmi podobné. Zde ovšem všechny podobnosti končí, protože volba **Option Strict** má zcela jiný význam nežli volba **Option Explicit**. Je-li volba **Option Strict** nastavena na hodnotu **On**, jsou zakázané všechny konverze datových typů, při kterých by mohlo dojít ke ztrátě dat. Visual Basic .NET standardně provádí tzv. implicitní konverze, a to při potřebě konverze hodnot jednoho datového typu do jiného. Tyto konverze se realizují přímo, bez jakéhokoliv přičinění ze strany uživatele (programátora). Někdy ovšem můžou tyto konverze způsobit ztrátu dat, čemuž má zabránit právě použití volby **Option Strict**. Když je volba zapnutá, implicitní konverze hodnot datových typů jsou povoleny jen v tom případě, jestliže jde o typově bezpečnou situaci. Pokud budete chtít provést konverzi, kterou volba **Option Strict** nepovoluje, můžete použít tzv. explicitní konverzní mechanizmus, v rámci kterého určíte způsob konverze hodnot. O konverzních mechanizmech si povíme příště.

Integrované prostředí rubriky Visual Basic	Autor: Ján Hanák

Téma

VB .NET

Programátorská laboratoř

Prostor pro experimentování

Časová náročnost (min):	Začátečník	Pokročilý	Profesionál
65	1	2	3

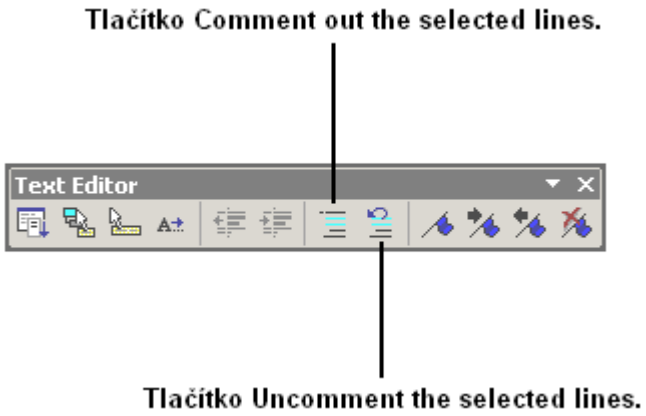
Použitý operační systém :	Windows 2000 SP3
Hlavní vývojový nástroj :	Visual Basic .NET 2002
Další vývojový software :	Žádný
Jiný software :	Žádný

VB .NET	Vizuální způsob přidávání komentářů	1	0:10
VB .NET	Ozřejmení práce konstruktoru a destruktoru	3	0:35
VB .NET	Tvorba přetíženého konstruktoru	3	0:20

Vizuální způsob přidávání komentářů

Smysl použití komentářů jsme si již vysvětlovali a tedy není ani nejmenších pochyb o tom, zdali byste je měli ve svém zdrojovém kódu používat. V tomto tipu si ovšem ukážeme, jak přidat komentář, a to plně vizuální cestou, tedy bez toho, abyste museli zapsat symbol apostrofu, nebo textový řetězec REM. Postupujte podle níže uvedených instrukcí:

1. Otevřete váš projektový soubor ve Visual Basicu .NET.
2. Přidejte do kódu komentáře a prozatím ignorujte skutečnost, že je editor zvýrazňuje vlnovkou modré barvy.
3. Ujistěte se, že je zobrazen panel s tlačítky s názvem **Text Editor**. Není-li tomu tak, klepněte pravým tlačítkem myši na standardním panelu s tlačítky a aktivujte položku **Text Editor**. Na panelu nás budou zajímat především dvě tlačítka: **Comment out the selected lines** – pro přidání komentářů a **Uncomment the selected lines** – pro odstraňování již vytvořených komentářů (obr. 1).



Obr. 1 – Podoba panelu **Text Editor**

4. Vyberte řádky s komentáři do bloku a klepněte na tlačítko **Comment out the selected lines**. Vybrané řádky se rázem promění na komentáře (bude před ně umístěn symbol apostrofu a barva textu bude zelená).

Zde představena technika najde uplatnění zejména při tzv. hromadném přidávání komentářů, ovšem využít ji můžete i v situaci, kdy budete chtít zamezit provádění jistého fragmentu zdrojového kódu aplikace a nebude se vám chtít přidávat symbol apostrofu na každý řádek s kódem. Jednoduše vyberte všechen kód do bloku a klepněte na vzpomínané tlačítko pro jeho proměnu na komentář. Budete-li chtít komentáře odstranit, postupujte analogicky, jenom po vybrání bloku textu klepněte na tlačítko **Uncomment the selected lines**.

Ozřejmení práce konstruktoru a destruktoru

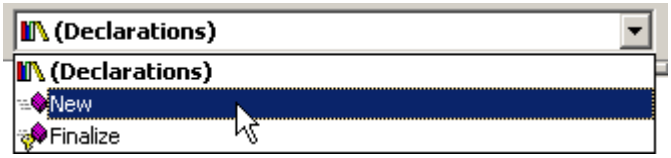
Pojmy konstruktor a destruktor jsou pro programátory ve Visual Basicu .NET nové, a proto bude dobré, když si je představíme. Konstruktor si můžete představit jako první metodu, která je podrobena exekuci v okamžiku vytvoření objektu. Ano, použití konstruktoru a posléze také destruktoru spadá do problematiky objektově orientovaného programování, a proto předpokládám, že vám tato oblast programování není úplně cizí.

Konstruktor je aktivován vždy při zrození objektu jako platné instance třídy. Tento okamžik je velmi vhodný pro jisté počáteční práce, jako je například inicializace datových členů instance třídy. Konstruktor nikdy nevrací hodnotu a může být přetížen (to znamená, že můžete vytvořit několik variant konstruktoru se stejným jménem, ovšem jinou sadou parametrů).

Ve Visual Basicu .NET má konstruktor vždy podobu procedury typ **Sub** s názvem **New**. Procedura může být bezparametrická, nebo může také obsahovat jeden nebo i několik parametrů. Na nastávající ukázce si budeme demonstrovat použití konstruktoru v praxi.

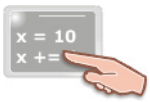
Postupujte takto:

1. Vytvořte novou aplikaci pro Windows (**Windows Application**).
2. Otevřete nabídku **Project** a klepněte na položku **Add Class**.
3. Přidejte do projektu soubor s kódem pro třídu (soubor můžete nechat implicitně pojmenovaný).
4. Název třídy změňte na **Třída1**.
5. Nyní otevřete otevírací seznam, jenž ukrývá metody vytvořené třídy. Otevírací seznam s metodami třídy se nachází na pravé straně okna editoru pro zápis programového kódu. Po otevření klepněte na metodu s názvem **New** (obr. 2).



Obr. 2 – Výběr metody **New**

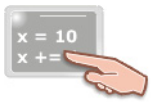
6. Do kódu třídy se automaticky přidá kód metody **New**, neboli konstruktoru této třídy:



```
Public Class Třída1
    Public Sub New()

    End Sub
End Class
```

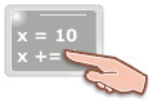
Standardně je konstruktor definován jako metoda s veřejným přístupem a prázdným seznamem parametrů. Zásadní význam má zde právě ona „veřejnost" konstruktoru, která dovoluje, aby byl kód konstruktoru vykonán a instance třídy zrozena. Jestliže by byl konstruktor definován s modifikátorem **Private**, nebylo by možné vytvářet instance dané třídy.

Abyste se přesvědčili, že konstruktor odvádí svou práci při tvorbě objektu, napište do procedury **New** tento řádek kódu:



```
MessageBox.Show("Toto je zpráva z konstruktoru.")
```

Po vytvoření objektu bude proveden kód konstruktoru, a tedy na obrazovce počítače uvidíte text této zprávy. Přidejte na formulář jednu instanci ovládacího prvku **Button** a do těla její událostní procedury **Click** zapište kód pro vytvoření instance třídy **Třída1**:



```
Dim Objekt_01 As New Třída1()
```

Jak jistě víte, instance třídy se vytváří použitím klíčového slova **New** a zadáním specifikace třídy. Po spuštění aplikace a klepnutí na tlačítko bude zobrazeno okno se zprávou, které dokazuje, že konstruktor odvádí svou práci dokonale. Ovšem uvedený příklad je vskutku triviální a jeho praktické využití takřka minimální. Proto se pojďme podívat na poněkud složitější uplatnění konstruktoru.

V dalším příkladu si ukážeme, jak použít konstruktor na vytvoření nové instance formuláře a jak přidat na plochu této nové instance instanci ovládacího prvku **Button**. Rovněž tak nám tělo konstruktoru poslouží na inicializaci hodnot všech potřebných vlastností a na aktivaci žádaných metod. Zajímavostí bude skutečnost, že půjde o parametrický konstruktor. Parametr konstruktoru bude přijímat textový řetězec, jenž bude později použit jako popisek u vytvořené instance tlačítka. Upravte kód třídy podle tohoto vzoru:

```
Public Class Třída1
    Inherits System.Windows.Forms.Form
```

Programový kód pokračuje na následující straně

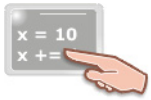
```
    Public Sub New(ByVal PopisekTlačítka As String)
        Dim Formulář As New Form()
        Dim Tlačítko As New Button()
        With Tlačítko
            .Location = New Point(10, 30)
            .Height = 100
            .Width = 200
            .Text = PopisekTlačítka
        End With

        With Formulář
            .Text = "Konstruktorem vytvořený formulář."
            .MinimizeBox = True
            .MaximizeBox = False
            .Controls.Add(Tlačítko)
            .Show()
        End With
    End Sub
End Class
```

Aby bylo možné v kódu konstruktoru vytvářet instance formuláře a tlačítka, je nutné použít příkaz **Inherits**, jenž říká, že naše třída bude dědit charakteristiky z třídy **Form**, která se nachází ve jmenném prostoru **System.Windows.Forms**. Dále si všimněte zejména signaturu konstruktoru. Je zde deklarován parametr datového typu **String** s názvem **PopisekTlačítka**, kterému bude argument předán hodnotou, což indikuje použití klíčového slova **ByVal**.

V těle konstruktoru se děje mnoho věcí:

- jsou vytvořeny instance tříd **Form** a **Button**,
- jsou specifikovány požadované vlastnosti vytvořené instance třídy **Button** (**Location**, **Height**, **Width** a **Text**),
- jsou upraveny některé vlastnosti vytvořené instance třídy **Form** (**Text**, **MinimizeBox**, **MaximizeBox**),
- instance ovládacího prvku **Button** je přidána do kolekce instancí ovládacích prvků instance formuláře,
- je zobrazena instance formuláře i s instancí ovládacího prvku **Button**.

Přemístěte se do událostní procedury **Button1_Click** tlačítka, které se nachází na formuláři. Zde zadejte kód pro vytvoření objektu s parametrickým konstruktorem:

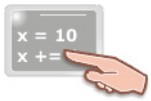



```
Dim Objekt_01 As New Třída1("Toto je moje tlačítko.")
```

Po spuštění aplikace a klepnutí na tlačítko se zobrazí nové dialogové okno formuláře, které bude obsahovat jedno tlačítko (obr. 3).

Obr. 3 – Ukázka práce konstruktoru v akci

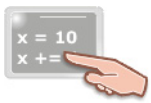
Destruktor si zase můžete představit jako poslední metodu, která je aktivována těsně před kompletní terminaci instance třídy. Destruktor je tedy jakýmsi protějškem konstruktoru. Destruktor se ve Visual Basicu .NET nazývá finalizer a je reprezentován procedurou typu **Sub** s názvem **Finalize**. Tato procedura má následující podobu:



```
Protected Overrides Sub Finalize()
    MyBase.Finalize()
End Sub
```

Jak vidíte, procedura je deklarována s modifikátory **Protected** a **Overrides**. Modifikátor **Protected** určuje, že procedura je přístupna pouze z třídy, ve které je deklarována a také z jakékoliv odvozené třídy z dané třídy. Použití modifikátoru **Overrides** nařizuje, aby se před terminací objektu použila procedura dané třídy a ne stejnojmenná procedura, která se nachází v bázové třídě. V těle destruktoru se nachází jenom jeden řádek, který zabezpečuje volání metody **Finalize** bázové třídy, což naznačuje použití klíčového slova **MyBase**. Protože je management paměti v kompetenci **Garbage Collection**, nelze předem odhadnout, kdy k exekuci finalizeru dojde (jde o tzv. nedeterministickou finalizaci instancí tříd).

Budete-li chtít zjistit, kdy se kód finalizeru provádí, upravte metodu **Finalize** tímto způsobem:

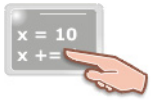


```
Protected Overrides Sub Finalize()
    MyBase.Finalize()
    MessageBox.Show("Právě se provádí kód destruktoru třídy.")
End Sub
```

Spusťte aplikaci, klepněte na tlačítko a zobrazte nový formulář. Poté zavřete nový a také standardní formulář. Ve chvíli, kdy bude procesor zpracovávat kód finalizeru, bude zobrazeno okno se zprávou.

Tvorba přetíženého konstruktoru

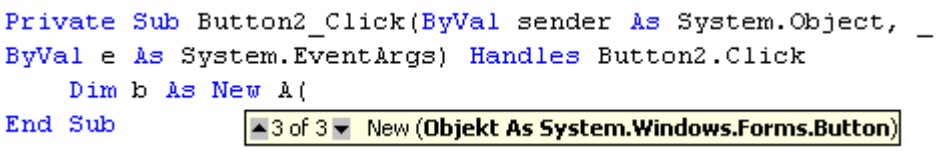
Konstruktoru třídy se říká, že je přetížený, jestliže existuje několik alternativ konstruktoru, které sice mají stejné jméno, ovšem liší se seznamem parametrů. Při vytváření objektu je potom jenom na programátorovi, který konstruktor se rozhodne použít. Programový kód třídy s přetíženým konstruktorem může mít třeba tuto podobu:




```
Public Class A

    Public Sub New(ByVal TextovýŘetězec As String)
        MessageBox.Show("Počet znaků v řetězci: " & _
        Len(TextovýŘetězec))
    End Sub

    Public Sub New(ByVal Koeficient As Double)
        Dim x As Double
        x = (Koeficient * 1.27) / 0.05
        MessageBox.Show("Hodnota výpočtu s použitím koeficientu: " & _
        x.ToString)
    End Sub

    Public Sub New(ByVal Objekt As Button)
        Dim frm1 As New Form()
        frm1.Controls.Add(Objekt)
        Objekt.Text = "Tlačítko1"
        frm1.Show()
    End Sub
End Class
```

V našem případě jsou deklarovány tři varianty konstruktoru třídy **A**. První varianta pracuje s parametrem typu **String**, jenž nese název **TextovýŘetězec**. Je-li tomuto konstruktoru předán textový řetězec, vypočte se počet znaků v řetězci prostřednictvím funkce **Len** (funkce **Len** vrací počet znaků v řetězci, přičemž za znak je považována také mezera). Druhá alternativa přijímá argument v podobě čísla s pohyblivou desetinnou čárkou. Vypočte hodnotu koeficientu, kterou zobrazení v okně se zprávou. Snad nejpozoruhodnější je třetí podoba konstruktoru, parametrem které je instance ovládacího prvku **Button**. Když je parametru předán název instance ovládacího prvku **Button**, je vytvořena instance třídy **Form** s názvem **frm1**. Následně je instance ovládacího prvku **Button** přidána do kolekce instancí ovládacích prvků formuláře **frm1**. Taktéž je modifikován textový řetězec, jenž se bude zobrazovat na vytvořené instanci ovládacího prvku **Button**.

Při deklaraci objektové proměnné vám technologie IntelliSense nabídne k použití všechny varianty konstruktoru (obr. 4).



Obr. 4 – Přetížený konstruktor s technologie IntelliSense

Integrované prostředí rubriky Visual Basic	Autor: **Ján Hanák**

Téma měsíce

Distribuční jednotky (2. díl)

Časová náročnost (min):	Začátečník	Pokročilý	Profesionál
60			

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

Vážení čtenáři,

po minulém úvodním dílu do problematiky tvorby distribučních jednotek aplikací je nejvyšší čas pustit se do poněkud neprobádaných končin. V dnešní části uvidíte, jaké pokročilé možnosti nastavení ukrývá konfigurace instalačního projektu a začneme také s charakteristikou vestavěných editorů, s jejichž pomocí si můžete instalaci vaší aplikace přizpůsobit zcela k obrazu svému.

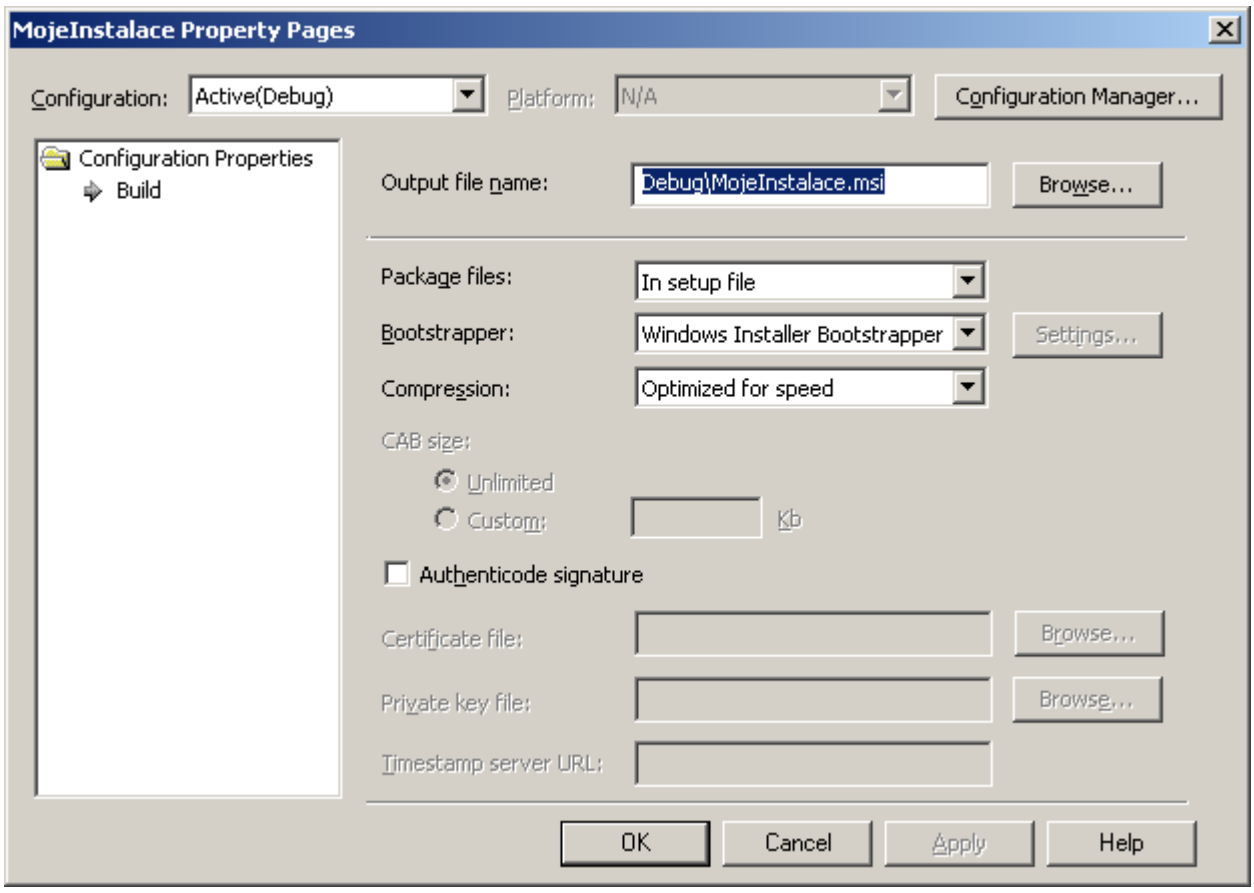
Obsah

Pokročilá nastavení instalačního projektu

Možnosti modifikace a editace instalačního projektu jsou daleko rozsáhlejší, než jsme si minule ukázali. Integrované prostředí vám dovoluje nastavit takřka všechno tak, jak potřebujete. Podívejme se tedy, jak na to:

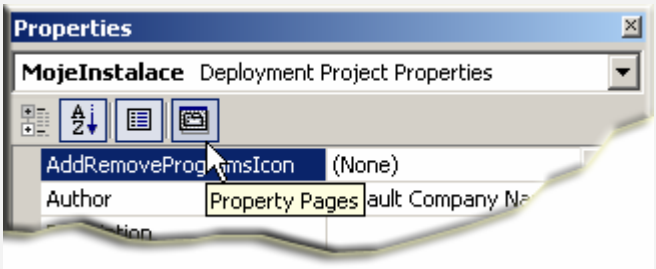
V následujícím textu se předpokládá, že jste probrali látku, která byla náplní prvního dílu problematiky, pojednávajícího o distribučních jednotkách aplikací a máte také připravený ukázkový instalační projekt. Z tohoto důvodu bude zde vysvětlovaná tematika navazovat na již známé informace.

1. V okně **Solution Explorer** klepněte na složenou položku **MojeInstalace** pravým tlačítkem myši a z kontextuální nabídky vyberte příkaz **Properties**. Zobrazí se dialogové okno **MojeInstalace Property Pages**, které nabízí množství nastavení, týkajících se sestavení instalačního projektu (obr. 1).



Obr. 1 – Dialogové okno **MojeInstalace Property Pages**

Dialogové okno **Property Pages** můžete zobrazit i tak, že v okně **Solution Explorer** vyberete položku **MojeInstalace** a v okně **Properties Window** klepnete na tlačítko **Property Pages**:



Proberme si tato nastavení podrobněji.

Textové pole Output file name

V tomto textovém poli je zobrazeno jméno výstupního souboru, ve kterém budou uložena instalační data. Instalační soubor má třípísmennou příponu .MSI a ve skutečnosti jde o soubor instalační utility Windows Installer. Kromě samotného jména instalačního souboru zde můžete vidět i složku, ve které je tento soubor uložen. Pokud se projekt nachází v režimu ladění (**Debug**), je jako aktivní vybrán právě tento režim a instalační soubor bude uložen ve složce **Debug**, která se nachází ve složce **MojeInstalace**. Budete-li chtít změnit umístění a název výstupního instalačního souboru, můžete přepsat obsah textového pole, nebo klepnout na tlačítko **Browse**.

Otevírací seznam Package files

Nastavení možností seznamu **Package files** určuje, jak budou projektové soubory testovací aplikace zakomponovány do instalačního projektu. V seznamu máte na výběr tyto možnosti:

- **As loose uncompressed files**
- **In setup file**
- **In cabinet file(s)**

Standardně je vybrána položka **In setup file**, která indikuje, že všechny potřebné soubory budou uloženy a zkomprimovány do jednoho instalačního souboru. Tato možnost je ideální, protože poskytuje velmi dobrý poměr mezi stupněm komprese a složitostí správy instalačního souboru. Jestliže nechcete, aby byly projektové soubory zkomprimovány, ani jakkoliv upravovány, můžete vybrat první volbu **As loose uncompressed files**. V tomto případě budou jednoduše vytvořené kopie všech projektových souborů, které budou umístěné do stejné složky jako instalační soubor (.MSI). Poslední varianta ukrývá možnost **In cabinet file(s)**. Tu můžete výhodně použít tehdy, potřebujete-li přesně specifikovat velikost výsledních CAB souborů. Budete-li chtít distribuovat vaší aplikaci pomocí disket, vyberete asi tuto možnost. Důležitá zpráva: Zvolíte-li možnost **In cabinet files(s)**, zpřístupní se přepínací pole **CAB size**, ve kterém můžete explicitně stanovit velikost jednotlivých souborů s CAB archivy.

Otevírací seznam Bootstrapper

Konfigurační položky, které se nacházejí v tomto seznamu, determinují, zdali má být společně s hlavním instalačním souborem vygenerována i menší instalační utilita, které se v originálu říká **Bootstrapper**. Za tímto názvem se skrývá aplikace, která zjistí, zdali se na cílovém počítači nachází požadovaná verze exekučního prostředí pro aplikaci Windows Installer. Pokud cílový počítač nemá vhodnou verzi běhového prostředí pro aplikaci Windows Installer, **Bootstrapper** ji nainstaluje a nakonec spustí primární instalační soubor. **Bootstrapper** instaluje aplikaci Windows Installer ve verzi 1.5, takže pokud se na cílovém počítači nachází starší verze tohoto softwaru, bude přepsána právě verzí 1.5.

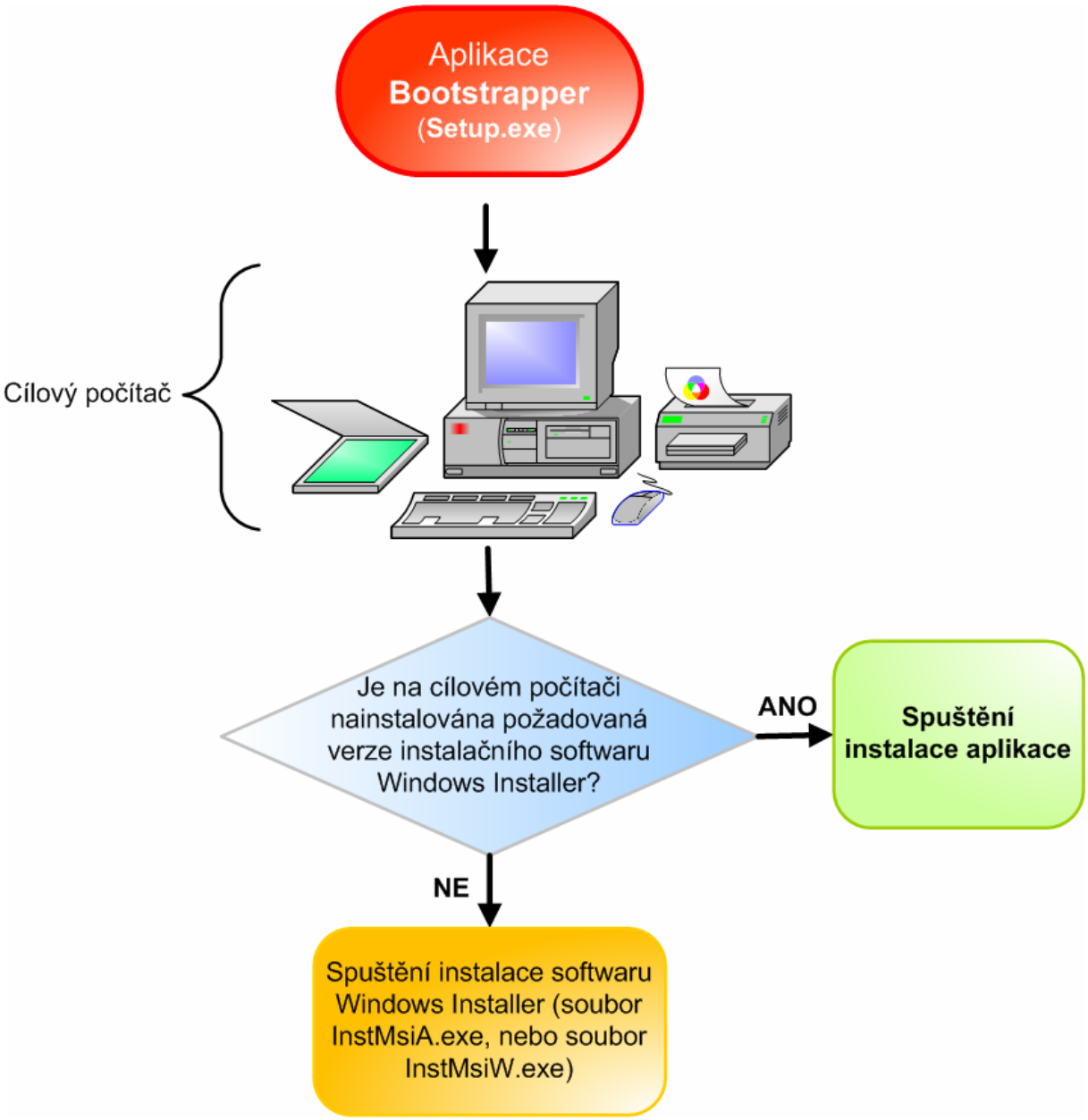
Standardně je ze seznamu **Bootstrapper** vybrána položka **Windows Installer Bootstrapper**, která zařazuje aplikaci pro testování a případnou instalaci souborů pro Windows Installer do hlavního instalačního souboru. Měli byste vědět, že selekce této volby způsobí vygenerování dalších (dodatečných) souborů, popis kterých můžete najít v tab. 1.

Soubor	Charakteristika
Setup.exe	Startovací bod instalace. Pokud je soubor spuštěn, pokusí se zjistit, zdali se na cílovém počítači nachází potřebná verze instalačního softwaru Windows Installer. Jestliže počítač nedisponuje požadovanou verzí této instalační aplikace, je spuštěn soubor InstMsiA.exe nebo soubor InstMsiW.exe, podle typu operačního systému cílového počítače. Oba soubory instalují verzi 1.5 aplikace Windows Installer.
InstMsiA.exe	Instaluje Windows Installer 1.5 na počítačích s operačním systémem Windows 95/98.
InstMsiW.exe	Instaluje Windows Installer 1.5 na počítačích s operačním systémem Windows NT a Windows 2000.
Setup.ini	Tento soubor obsahuje název hlavního instalačního souboru (.MSI). Hlavní instalační soubor bude spuštěn z procesu Setup.exe ve chvíli, kdy bude zjištěna přítomnost požadované verze softwaru Windows Installer (případně poté, co bude potřebná verze nainstalována).

Tab. 1

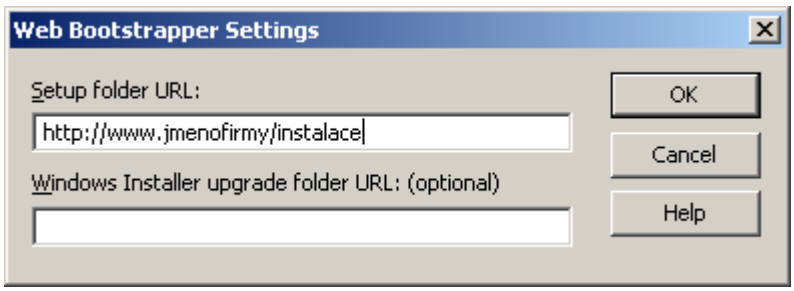
Operační systém Windows XP již nativně obsahuje Windows Installer verze 1.5, takže pokud plánujete instalaci pouze pro tento systém, můžete zařazení aplikace **Bootstrapper** vynechat.

Grafickou interpretaci procesu práce aplikace **Bootstrapper** můžete vidět na obr. 2.



Obr. 2 – Aplikace **Bootstrapper** v akci

Jestliže si nepřejete, aby byla do instalačního projektu zahrnuta aplikace **Bootstrapper**, vyberte ze seznamu **Bootstrapper** volbu **None**. V tomto případě ovšem budete muset zajistit, že se na cílovém počítači budou nacházet potřebné soubory exekučního prostředí pro běh instalace vaší aplikace. Chcete-li provádět instalaci aplikace z webového serveru, můžete zvolit třetí možnost s názvem **Web Bootstrapper**. Jakmile vyberete tuto položku, objeví se dialogové okno **Web Bootstrapper Settings** (obr. 3).



Obr. 3 – Dialogové okno **Web Bootstrapper Settings**

V tomto okně jsou umístěna dvě textová pole:

- **Setup folder URL** – do tohoto textového pole zapište URL adresu, na které se nacházejí instalační soubory vaší aplikace. Může jít o určení složky na webovém serveru v síti Internet (jak je uvedeno na obr. 3), nebo o určení složky v rámci sítě Intranet. V každém případě se však ujistěte, že uvedená adresa směruje na složku a ne na hlavní instalační soubor (.MSI).
- **Windows Installer upgrade folder URL (optional)** – toto textové pole slouží na zapsání URL adresy, na které se nacházejí soubory pro instalaci softwaru Windows Installer. Podobně jako v předchozím případě, i zde je potřebné určit adresu, která je nasměrována spíše na složku než na nějaký konkrétní soubor.

Ponecháte-li textové pole **Windows Installer upgrade folder URL (optional)** prázdné, předpokládá se, že soubory pro instalaci, resp. aktualizaci softwaru Windows Installer se nacházejí na URL adrese, která byla zapsána do pole **Setup folder URL**.

Otevírací seznam Compression

Jak asi tušíte, seznam **Compression** nabízí volby pro kompresi finálního instalačního souboru (za předpokladu, že je ze seznamu **Package files** vybrána volba **In setup file**), nebo pro kompresi jednoho či několika CAB souborů (je-li vybrána volba **In cabinet file(s)**). Implicitní nastavení je **Optimized for speed**, což znamená, že kompilátor se bude snažit generovat instrukce instalačního souboru tak, aby bylo provádění kódu co možná nejrychlejší. Protože kompilátor v tomto případě přiřazuje vyšší prioritu právě rychlosti instalace, bude sestavený instalační soubor poněkud větší. Jestliže pro vás hraje důležitější roli právě kapacitní náročnost instalačního souboru, můžete zvolit volbu **Optimized for size**. Uděláte-li tak, dáte kompilátoru příkaz, aby se soustředil raději na minimalizaci velikosti instalačního souboru než na rychlost, se kterou je vykonáván instalační kód. A konečně, pokud nebudete chtít vůbec žádnou kompresi, je zde volba **None**.

Otevírací seznam **Compression** není přístupný, jestliže jste v seznamu **Package files** vybrali možnost **As loose uncompressed files**. Použití této volby zamezí začlenění jakýchkoliv kompresních algoritmů, protože budou vytvářeny kopie originálních (zdrojových) souborů vaší aplikace.

Přepínací pole CAB size

Pole **CAB size** je aktivní, jenom když je ze seznamu **Package files** vybrána volba **In cabinet file(s)**. Při určování velikosti CAB souboru, resp. souborů jsou vám k dispozici dva přepínače:

- **Unlimited** – zvolíte-li tento přepínač, bude vygenerován jenom jeden CAB soubor, v němž budou uložena veškerá data vaší aplikace.
- **Custom** – při aktivaci tohoto přepínače se okamžitě zpřístupní také textové pole, do kterého můžete zapsat přesnou hodnotu velikosti každého CAB souboru (tato hodnota je udávána v kilobajtech, KB).

Pokud plánujete distribuovat kód vaší aplikace prostřednictvím disket, zadejte do textového pole přepínače **Custom** hodnotu 1440. Kompilátor na základě této informace pro vás připraví několik CAB souborů přibližně uvedené velikosti. Následně každý CAB soubor zkopírujte v příslušném pořadí na připravené diskety.

Podobný postup jako u disket můžete zvolit také třeba u ZIP médií, nebo u disků CD-R/RW. Potřebnou velikost si však v každém případě musíte pečlivě vypočítat, abyste optimálně vyplnili prostor zvoleného distribučního média.

Zatrhávací pole Authenticode signature

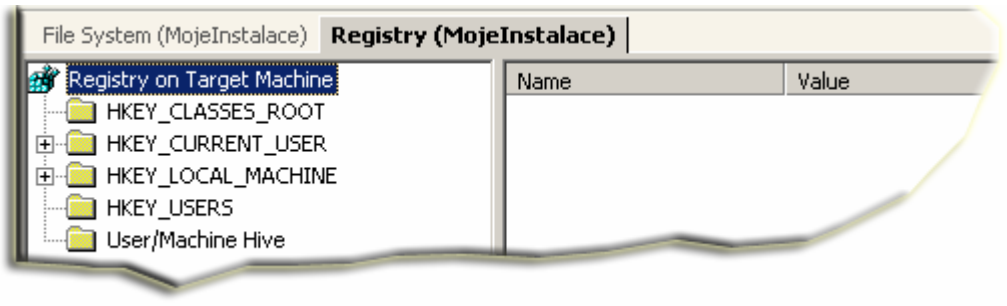
Píšete-li profesionální aplikaci, budete zřejmě také chtít, aby byla nainstalována profesionálním způsobem. Onen punc profesionality můžete instalačnímu projektu dodat specifikací certifikačního souboru a privátního klíče. Tak bude váš instalační soubor digitálně podepsán a vaši uživatelé si můžou být jisti, že instalační soubor pochází právě od vás. Chcete-li váš instalační soubor opatřit uvedenými prvky, zatrhněte pole **Authenticode signature** a vyhledejte soubory s certifikátem (pole **Certificate file**) a soukromým klíčem (**Private key file**). Volitelně lze také specifikovat URL adresu serveru, který byl použit pro přidělení digitálního certifikátu (**Timestamp server URL**).

Charakteristika editoru Registry

Editor **Registry** představuje velmi užitečnou pomůcku, která vám dovolí v režimu návrhu instalačního projektu naplánovat vytváření klíčů a hodnot v registrech operačního systému Windows. Navržené klíče a odpovídající hodnoty budou do registrů zapsány při instalaci aplikace. I když bylo možné tento úkol zvládnout i v předchozí verzi Visual Basicu, šlo o poněkud náročnou práci (bylo zapotřebí napsat všechen kód ručně). Pokud jste tedy pracovali s VB 6, můžete na programování vlastních akcí s registry při instalaci aplikace díky bohu zapomenout. IDE vám nabízí komfort, jenž s sebou přináší vizuální práce s registrem!

Na následujících řádcích si předvedeme, jak vytvořit registrový klíč a jak mu přiřadit textovou hodnotu. Postupujte takto:

1. V okně **Solution Explorer** klepněte pravým tlačítkem myši na instalační projekt s názvem **MojeInstalace**.
2. Jakmile se objeví kontextuální nabídka, ukažte na nabídku **View** a klepněte na položku **Registry Editor**.
3. Podobu otevřeného editoru můžete vidět na obr. 4. V levém poli se nachází virtuální podoba registrů Windows na cílovém počítači. V pravém poli stojí dva sloupce: **Name** pro deklarování jména registrové proměnné a **Value** pro nastavení hodnoty této proměnné.



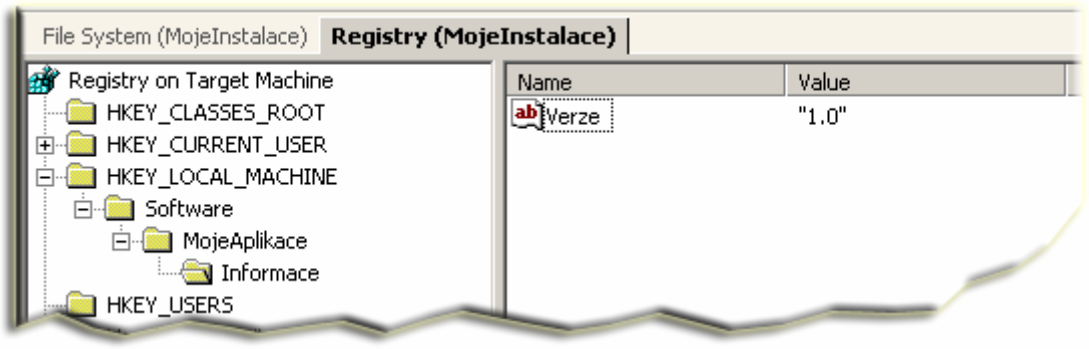
Obr. 4 – Editor **Registry**

4. Klepněte na symbol plus (+) uzlu **HKEY_LOCAL_MACHINE**.
5. Klepněte na symbol plus (+) uzlu **Software**.
6. Klepněte na klíč [**Manufacturer**].

7. V okně **Properties Window** změňte hodnotu vlastnosti **Name** vybraného registrového klíče na **MojeAplikace**.

Název klíče můžete změnit i tak, že na klíč klepnete pravým tlačítkem myši a z kontextové nabídky vyberete příkaz **Rename**. Dále stačí už jenom zapsat nový název pro klíč.

8. Z nabídky **Action** vyberte položku **New** a klikněte na položku **Key**. IDE vloží do editoru podklíč s implicitním názvem **New Key #1**.
9. Změňte název vytvořeného podklíče na **Informace**.
10. Přesvědčete se, že je stále vybrán podklíč **Informace**. Opět rozviňte nabídku **Action → New**, ovšem tentokrát aktivujte volbu **String Value**. Do sloupce **Name**, jenž se nachází v levém poli okna editoru, byla přidána textová proměnná (je standardně pojmenována jako **New Value #1**).
11. Klepněte na název proměnné pravým tlačítkem myši, vyberte příkaz **Rename** a změňte název proměnné na **Verze**.
12. Nyní proměnné přiřadíme hodnotu. V okně **Properties Window** vepište do vlastnosti **Value** hodnotu 1.0. Tato hodnota říká, že na cílovém počítači bude nainstalována první verze vaší aplikace. Podobu editoru **Registry** po uskutečněných změnách můžete vidět na obr. 5.



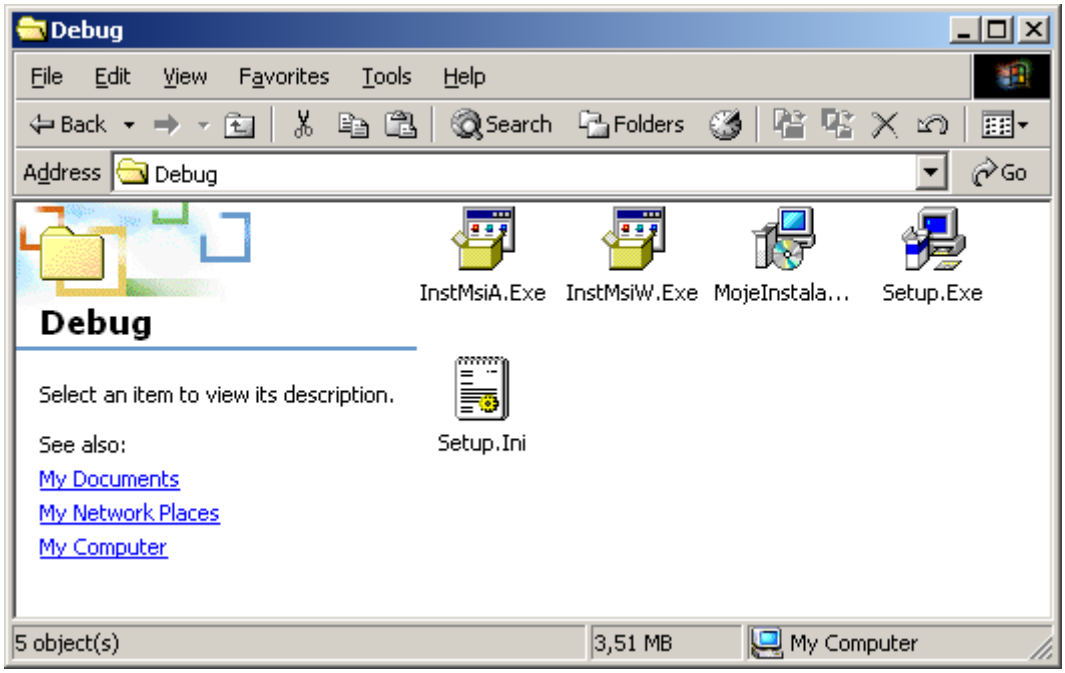
Obr. 5 – Finální podoba editoru **Registry**

13. V okně **Solution Explorer** klepněte na položku **MojeAplikace** pravým tlačítkem myši a klepněte na příkaz **Build**.

Integrované prostředí vygeneruje výstupní instalační soubory, které budou uloženy do složky **Debug** adresáře **MojeInstalace**. Obraz složky **Debug** je zobrazen na obr. 6.

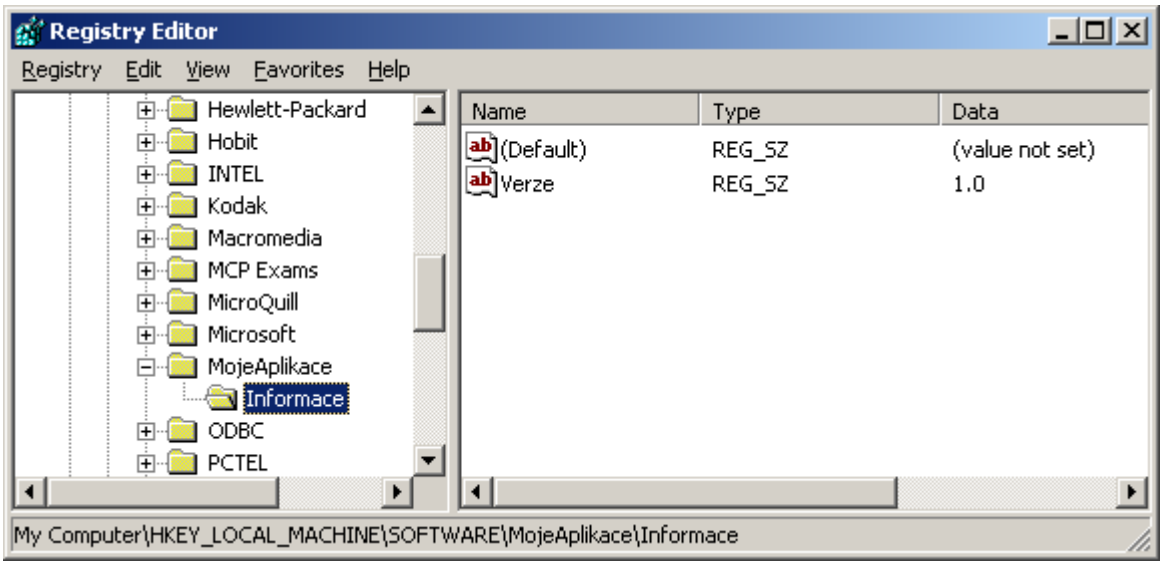
Při sestavování ukázkové instalace bylo použito implicitní nastavení okna **MojeInstalace Property Pages**:

- Output file name: Debug\MojeInstalace.MSI
- Package files: In setup file
- Bootstrapper: Windows Installer Bootstrapper
- Compression: Optimized for speed



Obr. 6 – Obraz složky **Debug**

Instalaci zahájíte poklepáním na aplikaci **Setup.exe**. Okamžitě bude zjištěna přítomnost nevyhnutných souborů pro Windows Installer a v případě potřeby budou potřebné soubory automaticky nainstalovány. Když instalační program zjistí, že cílový počítač disponuje použitelnou verzí softwaru Windows Installer, bude spuštěna instalace samotné aplikace. Následujte pokyny průvodce až do finální etapy. Instalátor zabezpečí správné nainstalování aplikace, přidá ikonu zástupce na plochu a vykoná zápisy do registrů systému Windows. Podíváte-li se do větvě HKEY_LOCAL_MACHINE\Software, uvidíte zde položku s názvem **MojeAplikace** (obr. 7).



Obr. 7 – Informace, které do registrů Windows přidal instalátor

Aplikaci **Registry Editor** spustíte takto:

- Vyberte nabídku **Start** a klepněte na položku **Run** (Spustit).
- Do pole pro zadání názvu programu vepište **regedit**.

Integrované prostředí rubriky Visual Basic	Autor: **Ján Hanák**

Lab

Téma

Začínáme s VB .NET

Úvod do světa .NET (9. díl)

Časová náročnost (min):	Začátečník	Pokročilý	Profesionál
45	1	2	3

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

Milí čtenáři,

v deváté lekci našeho seriálu si povíme něco o konverzních mechanizmech, které řídí konverzi hodnot proměnných různých datových typů při rozmanitých příležitostech, například při použití přiřazovacího příkazu, nebo při práci s parametry procedury. Dozvíte se, jak pracují implicitní a explicitní konverze a jaké funkce použít při konverzi hodnot proměnných různých datových typů.

Obsah

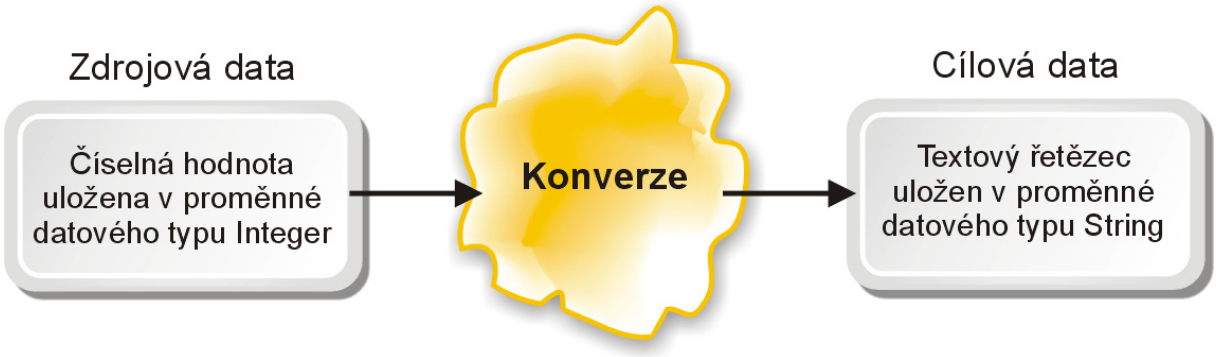
Filosofie konverzních mechanizmů

Máme-li mluvit o konverzních mechanizmech, musíme si ze všeho nejdříve vysvětlit jejich smysl, neboli skutečnost, proč je vůbec potřebné nějaké konverze používat. Každý programovací jazyk obsahuje jistou množinu vestavěných datových typů, které můžou uchovávat různorodé hodnoty. Nejinak je tomu i ve Visual Basicu .NET. V předešlých dílech seriálu jste mohli vidět přehled zástupců mnoha datových typů, se kterými jsme pracovali v programovém kódu. Již víte, že takzvané celočíselné datové typy mají možnost pracovat s celými čísly, a že třeba proměnné datového typu **String** dokáží uložit velmi dlouhý textový řetězec. Ano, dosud nebylo zapotřebí jakkoliv se zmiňovat o konverzích mezi proměnnými jednotlivých datových typů, protože tyto proměnné působily více-méně osamoceně. Pokud ovšem budeme chtít, aby proměnné začaly mezi sebou mluvit, záhy se potřeba konverze hodnot dostaví sama.

Pod pojmem konverze se ve všeobecnosti rozumí jistá proměna zdrojových dat na cílová data. Pracuje-li s grafickým softwarem, je možné, že často převádíte rastrové obrázky z jednoho formátu do jiného. Povězme, že máte za úkol převést obrázek, jenž je uložen ve formátu BMP, do podoby souboru JPG. Přesně toto je názorný příklad konverze. Jak tato konverze probíhá? Vskutku jednoduše, protože vše za vás vyřeší váš program pro editaci grafických předloh. Program jednoduše vezme vstupní data (rastrový obrázek ve formátu BMP) a tato data konvertuje do požadované cílové podoby (formát JPG). Tuto situaci přibližuje obr. 1.

Obr. 1 – Názorná ukázka konverze jednoho grafického formátu do jiného

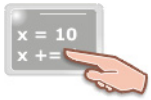
Situace, podobné té předchozí, se vyskytují i v programování. Zde ovšem nepůjde přímo o změnu formátu grafického obrázku, nýbrž o změnu hodnot proměnných různých datových typů. Budeme-li postupovat analogicky, můžeme tuto situaci opět graficky znázornit (obr. 2).



Obr. 2 – Jedna z podob konverzního procesu v programování

V uvedeném schématu je ilustrována konverze hodnoty proměnné celočíselného datového typu do proměnné datového typu **String** (mimochodem, jde o jeden z nejběžnějších typů konverzí).

Zajímavější je ovšem pohled na tento příklad po jednotlivých částech. Povězme, že máme deklarovanou proměnnou **A** datového typu **Integer**, ve které je uložena jistá numerická hodnota (řekněme 100). Hodnota 100 představuje naše zdrojová data, která vstupují do konverzního mechanizmu. Jak jsme si již řekli, výsledkem práce konverzního mechanizmu jsou jistá cílová data. V našem případě mají tato data podobu textového řetězce, jenž obsahuje znaky „1", „0" a „0". Textový řetězec je uschován v proměnné **B** datového typu **String**. Z číselné hodnoty 100 se tedy stane řetězec „100", který můžeme zobrazit třeba v titulkovém pruhu hlavního okna aplikace. Nyní se podívejme, jak tento modelový příklad znázornit v programovém kódu Visual Basicu .NET:




```
Dim A As Integer = 100
Dim B As String
B = A
Me.Text = B
```

Nejpozoruhodnější je bezpochyby třetí řádek s přiřazovacím příkazem **B = A**. Algoritmus přiřazovacího příkazu má přibližně tuto podobu:

- Visual Basic porovná datové typy obou proměnných a zjistí, že proměnné **A** a **B** nejsou stejného datového typu.

- Protože proměnné nejsou zástupci shodného datového typu, bude potřebné uskutečnit konverzi hodnoty proměnné, která se nachází na pravé straně přiřazovacího příkazu (proměnná **A**) do podoby hodnoty datového typu proměnné, která stojí na levé straně přiřazovacího příkazu (proměnná **B**). Hodnota proměnná **A** tedy vystupuje v úloze zdrojových dat.
- Visual Basic vezme celočíselnou hodnotu proměnné **A** a převede ji do podoby textového řetězce. Textový řetězec, neboli hodnota datového typu **String** představuje cílová data, která mají být výsledkem práce konverzního mechanizmu.
- Textový řetězec je ve finálním kroku přiřazen do proměnné **B**.

Všimněte si, že konverze byla v uvedeném příkladu provedena zcela automaticky a bez jakéhokoliv zásahu ze strany programátora (všechnu práci vykonal Visual Basic sám). Konverzím, které se uskutečňují automaticky se říká implicitní konverze. Opakem implicitních konverzí jsou konverze explicitní, při kterých musí programátor určit předmět a typ kýžené konverze. V dalších částech se podíváme na implicitní a explicitní konverze zblízka.

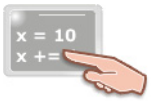
Implicitní konverze

Implicitní konverze jsou realizovány automaticky vždy, když konverze může být provedena bez toho, aby došlo ke ztrátě dat. Visual Basic je zvyčejně natolik chytrý, že ví, kdy implicitní konverze provést k plné spokojenosti programátora. Některé varianty, které přináší implicitní konverzní mechanizmus, jsou shrnuty v tab. 1.

Zdrojový datový typ	Cílový datový typ
Byte	Short, Integer, Long, Single, Double, Decimal
Short	Integer, Long, Single, Double, Decimal
Integer	Long, Single, Double, Decimal
Long	Single, Double, Decimal
Single	Double
Char	Integer, Long, Single, Double, Decimal

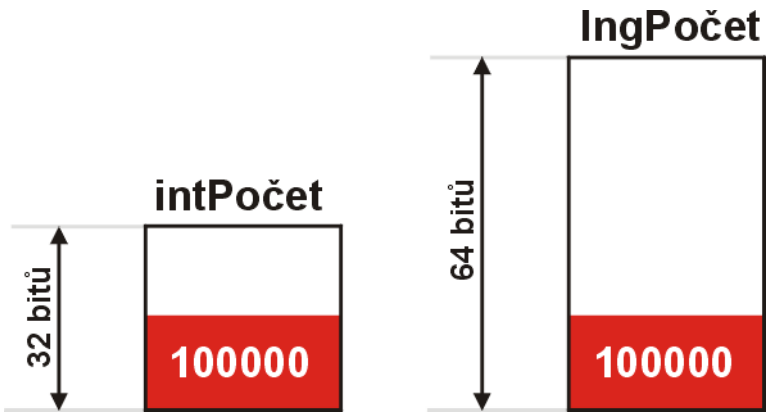
Tab. 1 – Implicitní konverze

Ukažme si další příklad, v němž se uplatňuje mechanizmus implicitní konverze:




```
Dim intPočet As Integer, lngPočet As Long
intPočet = 100000
lngPočet = intPočet
MessageBox.Show("Hodnota proměnné lngPočet je " & lngPočet)
```

Zde jsou deklarované dvě proměnné: Proměnná **intPočet** typu **Integer** a proměnná **lngPočet** typu **Long**. Do proměnné **intPočet** je přiřazena hodnota 100000. Následně je hodnota integerovské proměnné přiřazena do proměnné **lngPočet**. Jelikož obě proměnné jsou deklarované použitím různých datových typů, dochází v tomto případě k exekuci konverzního mechanizmu. Při bližším pohledu by bylo možné říci, že jde o rozšiřující konverzi (**widening conversion**), protože proměnné datového typu **Long** zabírají více místa v paměti a rovněž tak mají daleko větší rozsah ve srovnání s proměnnými datového typu **Integer**. Zatímco proměnná **intPočet** obsazuje v paměti 32 bitů, u proměnné **lngPočet** je to až 64 bitů (obr. 3).

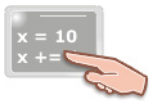


Obr. 3 – Anatomie proměnných **intPočet** a **lngPočet**

Z uvedeného jasně vyplývá, že při konverzi hodnot proměnných datového typu **Integer** do proměnných datového typu **Long** nemůže nikdy dojít k žádné ztrátě dat, a proto je tato konverze zcela bezpečná.

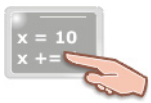
V posledním kroku je zobrazena hodnota proměnné **lngPočet** v dialogovém okně se zprávou. Ačkoliv to možná není na první pohled patrné, i v tomto příkaze dochází ke konverzi hodnoty. Za všechno může operátor zřetězení (**&**), jenž převede hodnotu proměnné **lngPočet** na hodnotu datového typu **String**. Takto vzniklý textový řetězec je dále spojen s textovým řetězcem, jenž se nachází nalevo od operátoru **&**. Spojením, neboli zřetězením těchto dvou textových řetězců vzniká nový řetězec, který je následně zobrazen v okně se zprávou.

Kromě rozšiřujících konverzí existují i opozitní konverze, při kterých dochází ke konverzi hodnot do datového typu s menším rozsahem (**narrowing conversions**). Posuďte tento fragment zdrojového kódu:



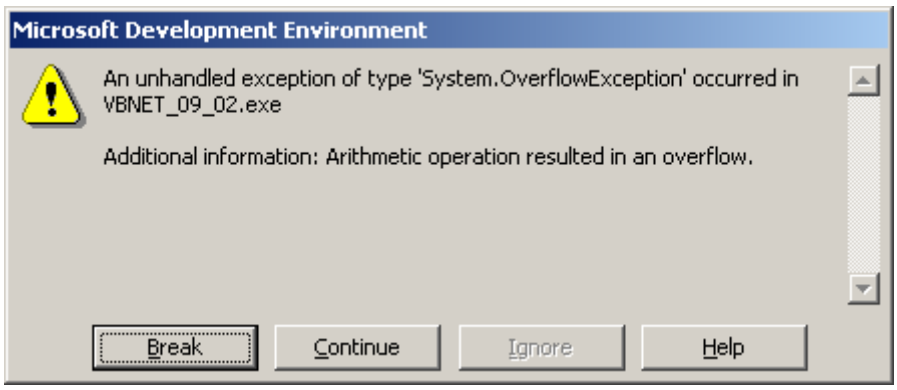

```
Dim x As Byte, y As Short
y = 200
x = y
MessageBox.Show("Hodnota proměnné x je " & x)
```

Máme zde deklarovanou proměnnou **x** datového typu **Byte** a proměnnou **y** datového typu **Short**. Do proměnné **y** je přiřazena hodnota 200. Dále je hodnota proměnné **y** přiřazena do proměnné **x**. Samotnému přiřazení předchází konverze hodnoty proměnné **y** do hodnoty datového typu **Byte** (jde o datový typ cílové proměnné, která se nachází na levé straně přiřazovacího příkazu). Všimněte si prosím, že i když je rozsah proměnné **y** větší než rozsah proměnné **x**, proběhne toto přiřazení bez problémů. Důvodem je skutečnost, že proměnná **y** obsahuje hodnotu (200), která se nachází rovněž v intervalu přípustných hodnot pro datový typ **Byte** cílové proměnné **x**. Co se ale stane, když do proměnné **y** přiřadíme třeba hodnotu 1000?




```
Dim x As Byte, y As Short
y = 1000
x = y
MessageBox.Show("Hodnota proměnné x je " & x)
```

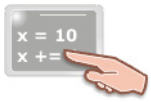
V tomto případě bude vygenerována chyba za běhu programu (obr. 4).



Obr. 4 – Chyba přetečení

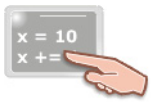
Jak se můžeme dozvědět z chybového hlášení, došlo k výjimce **System.OverflowException**. Visual Basic nám sděluje, že při realizaci přiřazovacího příkazu došlo k překročení hranic datového typu **Byte** proměnné **x** (jde o takzvanou chybu přetečení rozsahu datového typu) Protože proměnná datového typu **Byte** může pojmout pouze 256 hodnot z intervalu <0, 255>, není jednoduše možné do této proměnné uložit hodnotu 1000. Chybu můžeme vyřešit v zásadě dvěma způsoby:

1. Změníme datový typ proměnné **x** například na typ **Integer**. Když tak uděláme, bude realizována rozšiřující konverze a nemusíme se obávat žádné ztráty dat.
2. Budeme dávat pozor na to, aby do proměnné **x** nebyly ukládány nepřípustné hodnoty. V tomto případě můžeme použít techniku testování podmínky prostřednictvím příkazu **If** a vznik chyby tak předjímat:




```
Dim x As Byte, y As Short
y = 1000
If y >= 0 And y <= 255 Then
    x = y
    MessageBox.Show("Hodnota proměnné x je " & x)
Else
    MessageBox.Show("Hodnota " & y & _
    " nemůže být uložena do proměnné x.")
End If
```

I když jsme se ještě v našem seriálu nevěnovali řízení toku programu pomocí rozhodovací programovací konstrukce **If**, nastává vhodná chvíle na alespoň malou ukázku. Rozhodovací konstrukce **If** testuje jistou podmínku a v případě, že je tato podmínka splněna (jinak řečeno, výraz v podmínce je vyhodnocen na pravdivostní hodnotu **True**), je proveden kód, jenž se nachází za klíčovým slovem **Then**. Není-li podmínka splněna, je proveden programový kód, jenž následuje za návěstím **Else**. Celá konstrukce **If** je uzavřena příkazem **End If**. Velmi důležitý je výraz, který se testuje. V našem případě vypadá testovací výraz následovně:




```
y >= 0 And y <= 255
```

Tento výraz není jednoduchý, nýbrž kombinovaný. Provádí testování proměnné **y** pomocí logického operátoru **AND**. Výraz je pravdivý, jestliže je proměnná **y** rovna nebo větší než nula a současně menší nebo rovna hodnotě 255. Je-li výraz vyhodnocen na pravdivostní hodnotu **True**, je podmínka splněna a proveden programový kód, jenž následuje na klíčovým slovem **Then**. Řečeno jinak, obsahuje-li proměnná **y** přípustnou hodnotu, kterou lze přiřadit do proměnné **x**, přiřazovací příkaz bude vykonán. Jinak bude zobrazeno dialogové okno se zprávou, že přiřazení nelze uskutečnit.

Prezentovaná technika zamezí vzniku chyby za běhu programu a nebude zobrazeno ani žádné chybové hlášení. Někteří programátoři tuto techniku znají pod názvem defenzivní programování, jehož cílem je předvídání výskytu chyby a v případě vzniku chyby její okamžité ošetření.

Explicitní konverze

Explicitní konverze jsou nevyhnutné tehdy, když hodnoty proměnných různých datových typů nemůžou být implicitně konvertovány. Ve skutečnosti nabírá potřeba explicitních konverzí na významu v případě, že při programování používáte volbu **Option Strict** nastavenou na hodnotu **On**. Volba **Option Strict**:

- zakazuje všechny konverze, při kterých by mohlo dojít ke ztrátě dat,
- povoluje jenom rozšiřující implicitní konverze,
- zakazuje realizaci konverze mezi numerickými datovými typy a datovým typem **String**,
- zakazuje vytváření objektů s pozdní vazbou (**late-binding**).

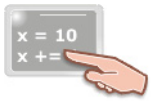
Vytváření objektů s pozdní vazbou (**late-binding**) probereme někdy příště, kdy se budeme učit o objektově orientovaném programování (OOP).

Pokud budete chtít konvertovat hodnoty proměnných rozličných datových typů, budete muset použít funkci **CType**, případně jiné konverzní funkce, které jsou uvedeny v tab. 2.

Název konverzní funkce	Návratová hodnota konverzní funkce
CBool	Boolean
CByte	Byte
CChar	Char
CDate	Date
CDbl	Double
CDec	Decimal
CInt	Integer
CLng	Long
CObj	Object
CShort	Short
CSng	Single
CStr	String

Tab. 2 – Konverzní funkce

Ukažme si názorný příklad (přitom předpokládáme, že je aktivována volba **Option Strict**):



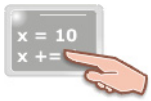

```
Dim intVýsledek As Integer, strŘetězec As String
intVýsledek = Math.Max(5000, 10000)
strŘetězec = CType(intVýsledek, String)
Me.Text = strŘetězec
```

Ve výpisu zdrojového kódu jsou deklarované dvě proměnné, **intVýsledek** datového typu **Integer** a **strŘetězec** datového typu **String**. Do proměnné **intVýsledek** je uložena hodnota, kterou vrací metoda **Max** třídy **Math**. Tato metoda přijímá dvě celočíselné hodnoty a vrací tu z nich, která je větší. Metoda

vrací hodnotu v podobě celého čísla datového typu **Integer**. Náš hlavní úkol spočívá v uložení získané návratové hodnoty do proměnné **strŘetězec**, která je zástupcem datového typu **String**. Jak jsme si již pověděli, aktivovaná volba **Option Strict** nedovoluje Visual Basicu provést implicitní konverzi celočíselné hodnoty do podoby textového řetězce. Proto musíme použít funkci **CType** v uvedeném tvaru.

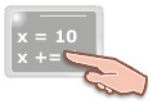
Funkce **CType** pracuje se dvěmi parametry. Prvním z nich je název vstupní proměnné (**intVýsledek**), která vstupuje do konverzního procesu. Druhým parametrem je název datového typu (**String**), do něhož má být vstupní proměnná konvertována. Výsledkem práce funkce je textový řetězec, jenž je zobrazen v titulkovém pruhu okna formuláře.

Analogicky by bylo možné předcházející programovou ukázku přepsat použitím konverzní funkce **CStr**:



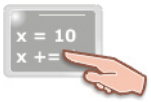

```
Dim intVýsledek As Integer, strŘetězec As String
intVýsledek = Math.Max(5000, 10000)
strŘetězec = CStr(intVýsledek)
Me.Text = strŘetězec
```

Další příklad vypíše do titulkového pruhu aplikace textový řetězec „True“:




```
Dim a As Short, b As Boolean
a = 100
b = CBool(a)
Me.Text = CStr(b)
```

Konverze typu celočíselná hodnota → pravdivostní hodnota datového typu **Boolean** jsou možné a platí, že jakákoliv nenulová hodnota reprezentuje hodnotu **True**, zatímco nula je konvertována na pravdivostní hodnotu **False**. Pokud byste chtěli přiřadit hodnotu proměnné typu **Boolean** do celočíselné proměnné typu **Integer**, pravdivostní hodnotu **True** bude reprezentovat 1 a hodnotu **False** nula. Tuto konverzi představuje následující fragment zdrojového kódu:

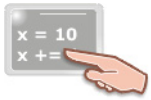



```
Dim k As Integer, l As Boolean
l = False
k = CType(l, Integer)
Me.Text = CStr(k)
```

Protože proměnná **l** obsahuje pravdivostní hodnotu **False**, bude po konverzi v titulkovém pruhu okna aplikace zobrazena nulová hodnota.

Hodnoty proměnných datového typu String a jejich konverze

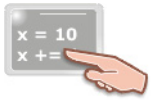
Hodnoty proměnných datového typu **String** lze konvertovat do hodnot proměnných numerických datových typů, samozřejmě za předpokladu, že obsah textového řetězce, jenž je uložen v proměnné typu **String**, může být převeden do numerické podoby. Prostudujte si tento příklad:



```
Dim strText As String
Dim intČíslo As Integer, intOperace As Integer
strText = "1440"
intČíslo = CType(strText, Integer)
intOperace = (intČíslo * 2) \ 100 + 5
Me.Text = CType(intOperace, String)
```

Zde je hodnota „1440" proměnné **strText** převedena do podoby celého čísla pomocí konverzní funkce **CType**. Takto získaná numerická hodnota je dále použita v matematické operaci. Nakonec je hodnota proměnné **intOperace** opět konvertována do hodnoty typu **String** a uložena do vlastnosti **Text** aktuální instance formuláře. Popravdě, výpočetní vzorec se vám může zdát poněkud složitý, a proto si jej probereme podrobněji.

Matematický výraz:



```
intOperace = (intČíslo * 2) \ 100 + 5
```

Má následující algoritmus výpočtu:

1. Vypočte se součin hodnoty proměnné **intČíslo** a čísla 2.
2. Součin vypočtený v předchozím kroku je vydělen dvěmi, ovšem pozor. Operátor zpětné lomítko (****) provádí celočíselné dělení. Výsledkem tohoto typu dělení je celé číslo (v případě, že výsledek disponuje desetinnou částí, je tato ořezána).
3. K získané hodnotě je nakonec připočteno pět jednotek.
4. Vypočtená hodnota je přiřazena do proměnné **intOperace**, která se nachází na levé straně přiřazovacího příkazu.

   Přesný průběh výpočtu můžete sledovat na následujícím schématu (na šedém pozadí jsou znázorněné operace s nejvyšší prioritou):

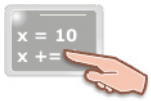
   intOperace = (1440 * 2) \ 100 + 5

   intOperace = 2880 \ 100 + 5

   intOperace = 28 + 5

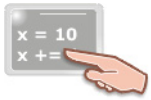
   intOperace = 33

Jak jsme si již ukázali, můžete hodnotu celočíselné proměnné konvertovat i pomocí operátoru zřetězení (**&**).Tento program umístí do titulkového pruhu textový řetězec „1 + 1 se rovná 2".



```
Dim strText As String = "1 + 1 se rovná "
Dim intHodnota As Integer = 2
Me.Text = strText & intHodnota
```

Pro konverzi textového řetězce do podoby čísla s pohyblivou desetinnou čárkou můžete použít funkci **Val**. Parametr této funkce přijímá argument ve tvaru textového řetězce a vrací odpovídající hodnotu v podobě čísla datového typu **Double**.




```
Dim strText As String = "1200.250"
Dim dblHodnota As Double
dblHodnota = Val(strText)
Debug.WriteLine(dblHodnota)
```

Funkce **Val** převede hodnotu textového řetězce na desetinné číslo, které je následně zobrazeno v okně **Output**. Metoda **WriteLine** třídy **Debug** se postará o to, aby se hodnota proměnné **dblHodnota** objevila v okně **Output**.

Zobrazování hodnot proměnných pomocí metody **WriteLine** třídy **Debug** je velmi užitečné zejména tehdy, když máte aktivovanou volbu **Option Strict** a potřebujete okamžitě zjistit hodnotu nějaké proměnné, ale nechcete ji konvertovat pomocí funkce **CType**.

Integrované prostředí rubriky Visual Basic	Autor: Ján Hanák

Programátorská laboratoř

Prostor pro experimentování

Časová náročnost (min):	Začátečník	Pokročilý	Profesionál
85	1	2	3

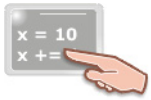
Použitý operační systém :	Windows 2000 SP3
Hlavní vývojový nástroj :	Visual Basic .NET 2002
Další vývojový software :	Žádný
Jiný software :	Žádný

Tvorba abstraktní třídy

Jednou z významných vlastností, které spadají do oblasti objektově orientovaného programování ve Visual Basicu .NET, je tvorba abstraktních tříd. Hned na začátek si ovšem musíme říci, že psaní abstraktních tříd nepatří zrovna mezi běžné úkoly programátora ve VB .NET. Na druhou stranu, pochopení principu, na němž spočívají základy tematiky abstraktních tříd, vás katapultuje do vyšších úrovní hry, které se říká profesionální programování.

Abstraktní třídy jsou třídy, které nejsou určeny pro přímou tvorbou objektů jistého typu. Znamená to, že abstraktní třída nemůže sloužit jako „továrna na objekty". Ve skutečnosti zastává abstraktní třída pozici bázové třídy, ze které mohou jiné třídy dědit veškerou funkcionalitu. Ovšem výraz, že jiné třídy můžou dědit z abstraktní třídy, je poněkud slabý. Skutečnost je taková, že jiné třídy musí dědit z abstraktní třídy. Tato vskutku fundamentální vlastnost abstraktní třídy je umocněna použitím klíčového slova **MustInherit** v deklaraci abstraktní třídy. I když abstraktní třída může obsahovat definice členských funkcí a procedur, obvykle jsou v abstraktní třídy uvedeny pouze deklarace těchto entit. Když vycházíme ze skutečnosti, že abstraktní třídou nelze přímo použít pro tvorbu objektů, je toto chování pochopitelné. V praxi se zvyčejně postupuje tak, že do abstraktní třídy se umístí pouze deklarace členů třídy, a konkrétní implementace těchto členů je následně ponechána na odvozené třídě. Jestliže již netrpělivě čekáte na ukázku abstraktní třídy, nebudu vás dále napínat. V následující případové studii si ukážeme, jak zhotovit jednoduchou abstraktní třídu. Postupujte takto:

1. Spusťte Visual Basic .NET a přikažte vytvoření aplikace pro Windows (**Windows Application**).
2. Do projektu přidejte soubor pro programový kód třídy. Vyberte nabídku **Project** a klepněte na položku **Add Class** (třídu můžete nechat implicitně pojmenovanou).
3. Do právě vytvořeného souboru třídy zapište kód abstraktní třídy:

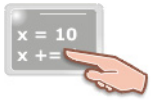


```
Public MustInherit Class A
    Public MustOverride Sub VytvořitFormulář()
End Class
```

Jak jsme si již řekli, v deklaraci abstraktní třídy se musí nacházet klíčové slovo **MustInherit**, které indikuje, že tato třída je určena pouze pro tvorbu z ní odvozených podtříd. V těle abstraktní třídy se nachází deklarace jednoho abstraktního členu této třídy. Přesněji jde o deklaraci procedury Sub s názvem **VytvořitFormulář**. Jak můžete vidět, v deklaraci členské procedury se nachází klíčové slovo **MustOverride**, které říká, že ještě předtím, než bude tato procedura použita, musí být překryta v kódu odvozené třídy. Při deklaraci členské procedury v abstraktní třídě je zapotřebí určit jenom indikátor přístupu (**Public**), klíčové slovo **MustOverride** a jméno procedury (**VytvořitFormulář**), které je následováno závorkami (v tomto případě prázdnými). Pokud byste chtěli, nic vám nebrání v tom, abyste přidali vlastní parametry s příslušnými datovými typy a návratovou hodnotu (v případě deklarace funkce).

Proceduru typu Sub můžete snadno „předělat" na funkci nahrazení klíčového slova Sub v deklaraci klíčovým slovem **Function**. Pokud se rozhodnete implementovat funkci, nezapomeňte také řádně specifikovat její návratovou hodnotu.

4. Přidejte kód pro odvozenou třídu, která dědí z abstraktní třídy:

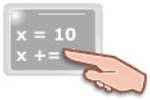


```
Public Class B
    Inherits A
    Public Overrides Sub VytvořitFormulář()
        Dim frm As New Form()
        With frm
            .Text = "Formulář č. 1"
            .Show()
        End With
    End Sub
End Class
```

Skutečnost, že třída **B** je odvozená do abstraktní třídy **A**, demonstruje příkaz **Inherits**. Protože abstraktní třída nabízí jenom deklaraci členské procedury, leží povinnost její implementace na bedrech odvozené třídy. Odvozená třída **B** se tedy této úlohy musí chtě-nechtě zhostit, a proto překryje proceduru z abstraktní třídy **A**. Aby bylo na první pohled jasné, že procedura překrývá proceduru z abstraktní třídy, je do deklarace překryté procedury vloženo klíčové slovo **Overrides**. V těle překryté procedury je vytvořena nová instance třídy **Form**, které je přiřazen text, jenž se bude zobrazovat v titulkovém pruhu okna instance a tato instance je posléze zobrazena.

Možná se divíte, jak může být vytvořen objekt třídy **Form**, když ani jedna ze tříd nedědí z třídy **System.Windows.Forms.Form**. Fígl spočívá v tom, že vaše odvozená třída ví o třídě **Form**, a také ví, kde se táto třída nachází. Řekne jí to totiž Visual Basic .NET, protože odkaz na vzpomínanou třídu se nachází v sekci odkazů (**References**) v okně **Solution Explorer**.

5. Přidejte na formulář jednu instanci ovládacího prvku **Button** a její událostní proceduru **Click** vyplňte tímto programovým kódem:



```
Dim ObjektováProměnná As New B()
ObjektováProměnná.VytvořitFormulář()
```

Jestliže spustíte aplikaci a klepněte na tlačítko, zobrazí se nový formulář s předem určeným textem v titulkovém pruhu.

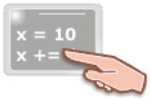
Tvorba a implementace rozhraní

Ve světě, v němž vládnou objekty, je velmi důležité, aby se tyto objekty dokázali mezi sebou domluvit. Aby ovšem bylo možné uskutečnit vůbec nějakou konverzaci mezi objekty, musí nejprve existovat jistý protokol, neboli rozhraní, které umožní přistupovat k metodám a vlastnostem objektů. Pod pojmem rozhraní si tedy můžete představit jistou smlouvu, nebo dohodu, která říká, jak bude uskutečňován komunikační proces mezi objekty. Jestliže jste používali nižší verze Visual Basicu, mohli jste rozhraní používat, ovšem jejich tvorba byla svěřena jiným programovacím jazykům (především Visual C++). S příchodem .NET verze jazyka Visual Basic se situace radikálně změnila, a proto se i my můžeme těšit na programovaní rozhraní.

Z technického hlediska je rozhraní ryzí informační protokol, který má jisté charakteristiky společné s tvorbou abstraktní třídy. Rozhraní se deklaruje příkazem **Interface** a v těle tohoto příkazu se uvádějí prototypy potřebných procedur, funkcí, vlastností a událostí, které má rozhraní podporovat. Prototypem se rozumí pouze specifikace členu rozhraní, jeho jména, signatury a návratové hodnoty (jestliže je zapotřebí). Je velmi důležité si uvědomit, že definice rozhraní pozůstává jenom z uvedených prototypů členů. Veškerá implementace členů rozhraní je (opět podobně jako u abstraktních tříd) přenesena do třídy, která bude uvedené rozhraní implementovat. Zjednodušeně by se dalo říct, že rozhraní definuje pouze to, „co se má udělat" a implementační třída zase nařizuje „jak se to má udělat".

Aby ovšem nezůstalo jen u teorie, pokusíme se sestavit rozhraní, poté jej implementovat a využít ve třídě. Zde jsou všechny potřebné instrukce:

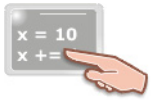
1. Spusťte Visual Basic .NET a vytvořte aplikace pro Windows (**Windows Application**).
2. Do projektu přidejte soubor pro kód třídy (nabídka **Project**, položka **Add Class**).
3. Do třídy začleňte kód pro rozhraní:



```
Public Interface IRozhraní
    Sub SpustitProgram(ByVal JménoProgramu As String)
    ReadOnly Property ZjistitIDProcesu()
End Interface
```

Rozhraní je definováno mezi příkazy **Interface** a **End Interface**. Příkazu **Interface** předchází určení modifikátoru přístupu (**Public**). Rozhraní je v našem případě veřejné, je tedy přístupné v rámci celého projektu. Za příkazem **Interface** následuje jméno rozhraní a podle konvence začíná toto jméno na písmeno „I". V těle rozhraní se nacházejí deklarace členů rozhraní, které mají stejnou úroveň přístupu jako celé rozhraní (**Public**). Můžete zde vidět jednu Sub proceduru s názvem **SpustitProgram**, která pracuje s jedním parametrem typu **String** s názvem **JménoProgramu**. Dalším členem je vlastnost **ZjistitIDProcesu**, která je určena jenom pro čtení, což indikuje použití klíčového slova **ReadOnly**.

4. Pokračujte zapsáním programového kódu třídy, která bude implementovat členy rozhraní:

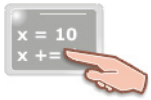


```
Public Class C
    Implements IRozhraní

    Dim m_Proces_ID As Integer

    Public Sub SpustitProgram(ByVal JménoProgramu As String) _
    Implements IRozhraní.SpustitProgram
        Dim proc As New Process()
        With proc
            .StartInfo.FileName = JménoProgramu
            .Start()
        End With
        m_Proces_ID = proc.Id
    End Sub

    Public ReadOnly Property ZjistitIDProcesu() _
    Implements IRozhraní.ZjistitIDProcesu
        Get
            Return m_Proces_ID
        End Get
    End Property
End Class
```

Zde vidíte výpis kódu pro třídu **C**. Ihned za deklaračním příkazem třídy se nachází příkaz **Implements** s názvem rozhraní pro implementaci. Příkaz **Implements** determinuje, že třída **C** bude podporovat rozhraní **IRozhraní** a je nutné, aby třída definovala všechny členy rozhraní. To třída **C** také na následujících řádcích dělá. Nejdřív je plně implementována Sub procedura **SpustitProgram**. Všimněte si prosím, že za signaturou procedury **SpustitProgram** je opět uveden příkaz **Implements**, za kterým je uveden člen rozhraní, který je prostřednictvím procedury implementován. V rámci procedury je vytvořena instance třídy **Process** s názvem **proc**. Jméno programu, jenž se bude spouštět je uloženo do vlastnosti **Filename**. Běh kódu pokračuje zavoláním metody **Start**, která zabezpečí spuštění specifikovaného souboru.

Implementována je i vlastnost **ZjistitIDProcesu**, která vrací identifikační číslo spuštěného procesu. Opět si všimněte, že je potřebné zapsat příkaz **Implements** za jméno vlastnosti a určit odpovídající protějšek z definice rozhraní.

5. Přidejte na formulář instanci ovládacího prvku **Button** a do jeho událostní procedury **Click** napište tento kód:



```
Dim x As New C()
x.SpustitProgram("Notepad.exe")
Me.Text = "ID procesu: " & x.ZjistitIDProcesu
```

Tento kód vytváří objektovou proměnnou **x**, která obsahuje referenci na instanci třídy **C**. Ve druhém kroku je zavolána metoda **SpustitProgram**, které je předán název programu, jenž se má spustit. Poslední příkaz zobrazuje identifikační kód právě spuštěného programu (procesu) v titulkovém pruhu testovací aplikace.

Překrytí metody v odvozené třídě

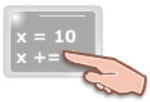
V tomto tipu si ukážeme, jak překrýt metodu bázové třídy v odvozené třídě. Začneme však zvolna a vysvětlíme si, co se pod pojmem překrývání metod ve skutečnosti rozumí. Proces překrývání metod není ve světě programování nijak nový, ostatně mnohé jazyky, například C++, jej používají již

nějaký ten pátek. Hlavní myšlenka je takováto: Pomocí překrývání metod můžete jednoduše nahradit metodu bázové třídy svou vlastní metodou, která může mít zcela jinou implementaci. Jak je jistě patrné, proces překrývání metod je velice úzce spjat s procesem dědění. Představte si následující příklad: Máte dvě třídy, jednu bázovou a druhou odvozenou z bázové třídy.
S funkcionalitou bázové třídy jste vcelku spokojeni, ovšem přáli byste si, aby několik metod bázové třídy disponovalo novější stavbou programového kódu, a zajišťovalo lepší funkcionalitu. Pokud se někdy dostanete do podobné situace, určitě přivítáte pomocnou ruku, kterou vám mechanizmus překrývání metod může nabídnout. Výsledkem bude jednoduše překrytí metody bázové třídy (programový kód překryté metody bude zcela jiný a bude tak poskytovat modifikovanou funkcionalitu).

Při překrývání metody ovšem nesmíte zapomenout na skutečnost, že nová implementace metody musí mít stejný modifikátor přístupu, název a rovněž tak i stejnou signaturu jako metoda, kterou se chystáte překrýt.

Překrývání metody v odvozené třídě si budeme demonstrovat na následující programové ukázce.

1. Spusťte Visual Basic .NET a vytvořte aplikaci pro Windows (**Windows Application**).
2. Pokračujte přidáním souboru pro programový kód třídy (nabídka **Project** a příkaz **Add Class**).
3. Do souboru třídy vložte následující kód pro definici bázové a odvozené třídy:




```
Public Class BázováTřída
    Public Overridable Sub ZobrazitČas()
        MessageBox.Show("Právě teď je " & _
        DateTime.Now.TimeOfDay.Hours & _
        " hodin.")
    End Sub
End Class

Public Class OdvozenáTřída
    Inherits BázováTřída
    Public Overrides Sub ZobrazitČas()
        Form.ActiveForm.Text = "Právě teď je " & _
        DateTime.Now.TimeOfDay.Hours & " hodin."
    End Sub
End Class
```

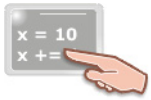
Předmětem našeho zájmu je procedura **ZobrazitČas**. Tato procedura je v bázové třídě definována s klíčovým slovem **Overridable**, které z procedury dělá vhodného kandidáta pro pozdější překrytí. Mimochodem, metoda **ZobrazitČas** v bázové třídě je zodpovědná za zobrazení dialogového okna s počtem uplynulých hodin dne.

Třída s názvem **OdvozenáTřída** dědí svou funkcionalitu z bázové třídy. I v této třídě se nachází kód s metodou **ZobrazitČas**, ovšem jak si můžete všimnout, tato metoda:

- je deklarována s použitím klíčového slova **Overrides**,
- provádí jinou činnost nežli její předchůdkyně v bázové třídě (zobrazuje počet hodin v titulkovém pruhu aktivního formuláře),
- má stejný modifikátor přístupu (**Public**), název (**ZobrazitČas**) i signaturu.

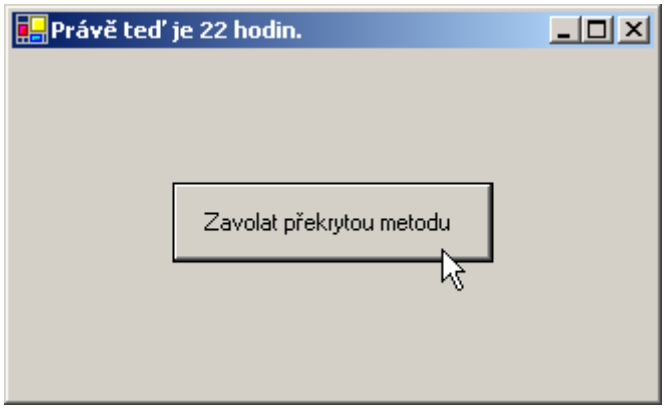
Můžeme prohlásit, že metoda **ZobrazitČas** v odvozené třídě překrývá metodu **ZobrazitČas** v bázové třídě.

4. Na formulář umístěte jednu instanci ovládacího prvku **Button** a do události **Click** instance přidejte tento kód:



```
Dim obj As New OdvozenáTřída()
obj.ZobrazitČas()
```

Výsledek můžete vidět na obr. 1.

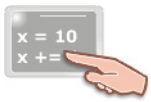


Obr. 1 – Zavolání překryté metody z odvozené třídy

Rotace rastrového obrázku

Programátoři grafických aplikací se zcela určitě nezaobejdou bez techniky rotování rastrových obrázků. Dobrou zprávou je, že platforma .NET poskytuje množství tříd, které vám mohou při programování různých grafických efektů v mnohém pomoci. My se nyní podíváme na to, jak rotovat jednoduchým rastrovým obrázkem, jenž je uložen ve formátu GIF. Budeme postupovat takto:

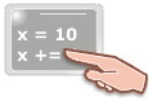
1. Na formulář umístěte instanci ovládacího prvku **PictureBox**, kterou nechejte implicitně pojmenovanou (**PictureBox1**).
2. Na plochu formuláře nakreslete dvě instance ovládacího prvku **Button**, které pojmenujte jako **btnNačístObrázek** a **btnRotovat**.
3. Vlastnost **Text** prvního tlačítka změňte na „Načíst obrázek".
4. Vlastnost **Text** druhého tlačítka změňte na „Rotovat".
5. Do událostní procedury **Click** tlačítka **btnNačístObrázek** vepište tento fragment programového kódu:



```
With PictureBox1
     .BorderStyle = BorderStyle.FixedSingle
     .SizeMode = PictureBoxSizeMode.CenterImage
     .Image = Image.FromFile("D:\obr1.gif")
End With
```

Aby měl ovládací prvek **PictureBox1** vizuálně ohraničenou plochu, upravíme jeho vlastnost **BorderStyle** na hodnotu **FixedSingle**. Dále je nastavena vlastnost **SizeMode** na hodnotu **CenterImage**, což znamená, že obrázek, jenž bude vykreslen na ploše tohoto ovládacího prvku, bude okamžitě vycentrován horizontálně i vertikálně. V další etapě je do vlastnosti **Image** ovládacího prvku **PictureBox1** umístěna reference na objekt **Image**, který je tvořen rastrovým obrázkem načteným z předem připraveného grafického souboru.

6. Do událostní procedury **Click** tlačítka **btnRotovat** zadejte tento zdrojový kód:

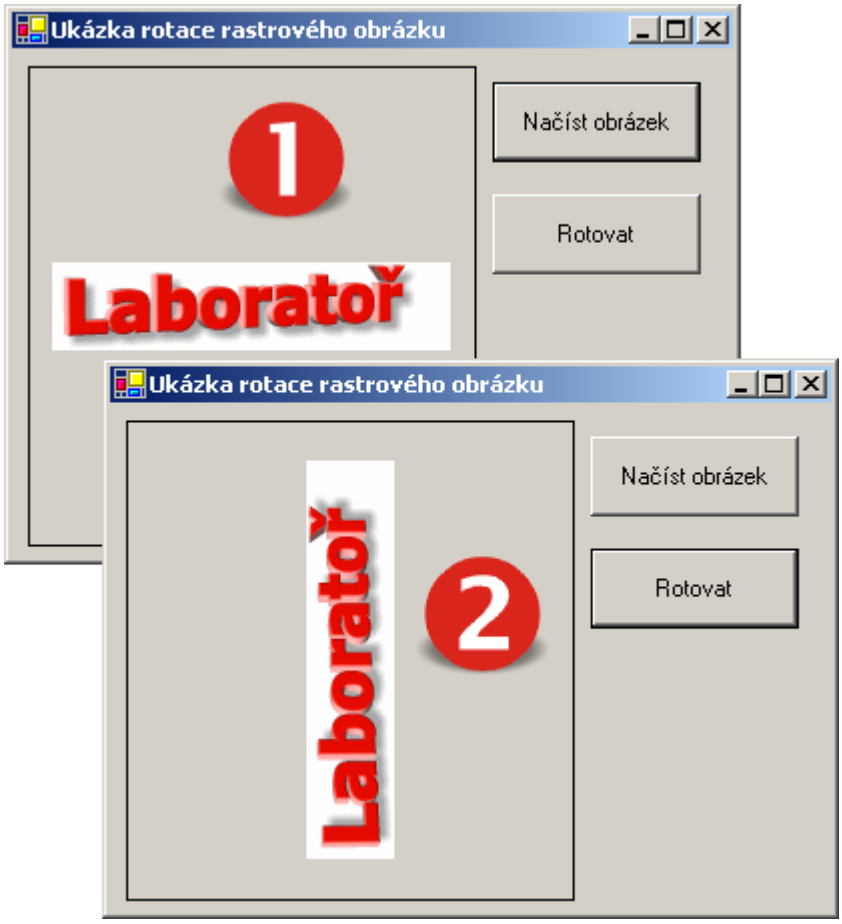



```
Dim g As Image = PictureBox1.Image
g.RotateFlip(RotateFlipType.Rotate270FlipNone)
PictureBox1.Refresh()
```

Zde jsou realizovány tyto úkony:

- je vytvořena objektová proměnná **g**, do které je uložena reference na objekt s rastrovým obrázkem (tato reference je získána z vlastnosti **Image** ovládacího prvku **PictureBox1**),
- je použita metoda **RotateFlip**, která provede rotaci obrázku o 270 stupňů,
- je překreslena plocha ovládacího prvku **PictureBox1** použitím metody **Refresh**.

Načtený obrázek a jeho modifikovanou verzi můžete vidět na obr. 2.



Obr. 2 – Rotace rastrového obrázku

Integrované prostředí rubriky Visual Basic	Autor: **Ján Hanák**

Téma měsíce

Distribuční jednotky (3. díl)

Časová náročnost (min): **100**	Začátečník	Pokročilý	Profesionál
			

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

Vážení čtenáři,

v poslední části seriálu o tvorbě distribučních jednotek aplikací si ukážeme další postupy a triky, které vám pomohou sestavit naprosto profesionální instalační systém. Dozvíte se, jak asociovat soubory s uživatelskou příponou s aplikací ihned při instalaci a jak pracovat s vestavěným editorem **File Types**.

Obsah

Teoretický úvod do problematiky vytváření asociace mezi aplikací a podporovaným typem souborů

V této kapitole si budeme demonstrovat, jak realizovat zaregistrování asociace mezi aplikací a soubory, které budou používány touto aplikací. Jistě jste se již střetli s mnoha aplikacemi, které interně používají svůj vlastní formát pro ukládání a otevírání dokumentů. Vezměme si třeba textový procesor Word společnosti Microsoft. Nativním souborovým formátem pro ukládání, otevírání a modifikování textových dokumentů je u této aplikace formát doc. Každý soubor, jenž disponuje prvotní třípísmennou extenzí doc, je tedy z pohledu aplikace Word textovým dokumentem. Vizuálně je podoba textového dokumentu doc umocněna také použitím vhodné grafické ikony, která uživatelům připomíná, o jaký soubor se ve skutečnosti jedná. Pokud poklepete na tuto ikonu, spustí se aplikace Word. Ovšem zamýšleli jste se někdy, proč je tomu tak?

Odpovědí na uvedenou otázku je vytvoření asociace mezi aplikací a soubory, které tato aplikace dokáže přečíst, pozměnit a uložit. Jestliže je takováto asociace vytvořena, je tím výslovně určeno, že daná aplikace bude spouštět a spravovat podporovaný typ souborů. Pokud se vrátíme k úvodnímu příkladu s aplikací Word, zde je jasně pozorovatelná asociace mezi aplikací a soubory s příponou doc.

Všechny informace, které se týkají vytvořené asociace, jsou uloženy v registrech operačního systému Windows. Tyto informace jsou pro operační systém až životně důležité, protože je to právě on, kdo musí zajistit všechno potřebné pro bezproblémovou komunikaci mezi aplikací a asociovanými soubory.

Chcete-li si prohlédnou extenze asociovaných souborů na vašem počítači, spusťte aplikaci Registry Editor a rozviňte uzel s názvem HKEY_CLASSES_ROOT. Jak si můžete všimnout, nachází se zde opravdu značné množství asociovaných souborů.

Jestliže uživatel poklepe na soubor s příponou doc, operační systém vyhledá ve svých registrech tento typ přípony a zjistí, s jakou aplikací je daný typ souboru asociován. Poté, co operační systém obdrží veškeré požadované informace, je aktivována cílová aplikace (v tomto případě Word) a je jí poskytnuto jméno souboru, jenž byl aktivován uživatelem (kromě jména souboru mohou být aplikaci nabídnuty i další argumenty příkazového řádku, které dodávají aplikaci další potřebné informace). V následující etapě je řízení předáno aplikaci Word, která už sama zajistí otevření souboru a načtení jeho obsahu do příslušného aplikačního okna.

Jedna aplikace může spravovat i více než jeden typ souborů. Ve skutečnosti se můžete střetnout s mnoha aplikacemi, které jsou schopné pracovat s několika typy souborů. I již několikrát zmiňovaná aplikace Word si rozumí s mnoha typy souborů: Kromě nativního formátu doc jsou to například soubory rtf, klasické textové soubory (txt) a mnohé další.

Jisté problémy nastávají v případě, když je jeden typ souboru asociován s několika aplikacemi. Tato situace velmi často nastává, když má uživatel na počítači nainstalované různé programy pro tvorbu grafických předloh. Tak se stane, že třeba grafický soubor png umějí zpracovat dva nebo tři programy. Ovšem, který grafický program se po poklepání na ikonu souboru spustí? Opět jedna otázka, pro rozřešení které si budeme muset nasadit klobouk detektiva a vzít lupu do ruky. Jak jsme si již pověděli, při instalaci většiny aplikací se do registrů operačního systému Windows ukládají údaje o asociovaných souborech (podle těchto informací je operační systém schopen určit asociaci mezi aktivovaným souborem a cílovou aplikací). Dobrá, tento postup funguje skvěle při instalaci první aplikace, ale co se stane, když budeme chtít nainstalovat další? Snadno může dojít k přepsání asociačních informací, což znamená, že od této chvíle bude po poklepání na ikonu souboru PNG spouštěna druhá (posledně nainstalovaná) aplikace a ne aplikace původní. Programátoři s detektivní povahou však zcela jistě naleznou způsob, jak vhodně upravit registry systému tak, aby vše pracovalo dle jejich požadavků.

Abyste viděli, jak nám může Visual Studio .NET pomoci při navazování asociace mezi aplikací a uživatelským typem souboru, budeme se věnovat následujícím záležitostem:

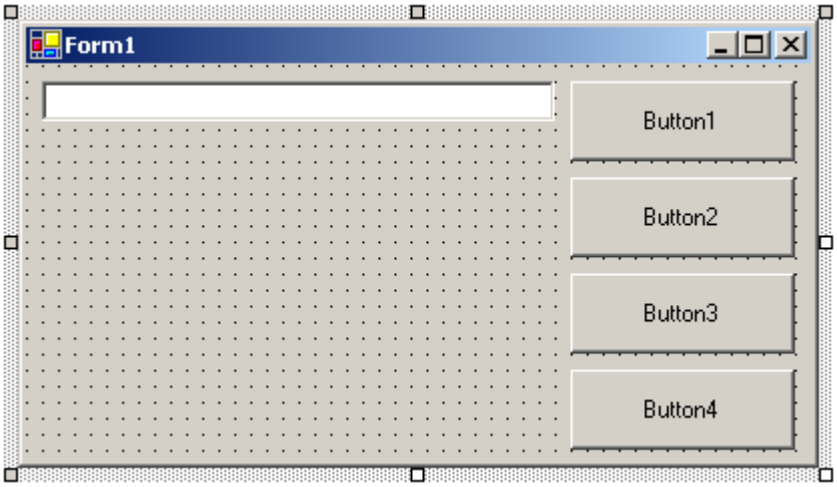
- Vytvoříme testovací aplikaci, která bude otevírat a ukládat soubory ve vlastním formátu (konkrétně se bude jednat o soubory s extenzí abc).
- Vytvoříme instalační projekt, jenž bude zabezpečovat registraci asociace mezi aplikací a soubory abc v operačním systému.
- Soubory abc opatříme vlastní ikonou.
- Do testovací aplikace zařadíme proceduru **Sub Main**, která bude obsahovat kód pro načtení obsahu souboru abc poté, co na ikonu souboru poklepe uživatel.

Vytváření testovací aplikace

Naše testovací aplikace bude umět číst a ukládat data do vlastního interního typu souboru. Soubory aplikace budou mít extenzi abc. Pusťme se tedy do práce:

1. Spusťte Visual Studio .NET, vyberte nabídku **File**, ukažte na nabídku **New** a klepněte na položku **Project**. V rámečku **Project Types** vyberte položku **Visual Basic Projects** a v rámečku **Templates** zvolte projekt aplikace pro Windows (**Windows Application**). Vytvářenou aplikaci pojmenujte jako **AplikaceABC**.

2. Na formulář přidejte jednu instanci ovládacího prvku **TextBox** a čtyři instance ovládacího prvku **Button** v podobě znázorněné na obr. 1.



Obr. 1 – Vzhled formuláře po přidání instancí ovládacích prvků

3. Vlastnosti instance **TextBox** upravte takto:

Vlastnost	Hodnota
Name	txtPole
Multiline	True
ScrollBars	Vertical
TabIndex	4
Text	Prázdna hodnota

Změňte také velikost instance ovládacího prvku **TextBox** tak, aby instance zabírala vhodnou plochu formuláře.

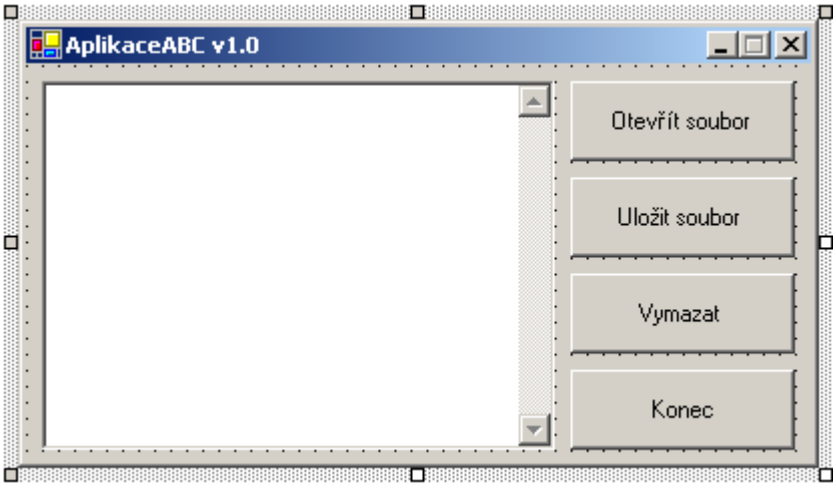
4. Vlastnosti instancí ovládacího prvku **Button** modifikujte takto:

Název instance	Vlastnost	Hodnota
Button1	**Name**	btnOtevřít
	TabIndex	0
	Text	Otevřít soubor
Button2	**Name**	btnUložit
	TabIndex	1
	Text	Uložit soubor
Button3	**Name**	btnVymazat
	TabIndex	2
	Text	Vymazat soubor
Button4	**Name**	btnKonec
	TabIndex	3
	Text	Konec

5. Vlastnosti formuláře **Form1** změňte také:

Vlastnost	Hodnota
MaximizeBox	False
StartPosition	CenterScreen
Text	AplikaceABC v1.0

Po provedených úpravách by měl vzhled formuláře a jeho prvků odpovídat obr. 2.

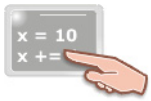


Obr. 2 – Instance ovládacích prvků po modifikaci jejich vlastností

Výborně, návrh formuláře jsme zvládli. Teď začneme s psaním programového kódu pro obsluhu událostních procedur **Click** jednotlivých tlačítek.

Vytváření zpracovatele události btnOtevřít_Click

Do zpracovatele události **btnOtevřít_Click** vepište tento fragment zdrojového kódu:




```
Dim obj_OpenFileDialog1 As New OpenFileDialog()
obj_OpenFileDialog1.Filter = "Soubory ABC (*.abc)|*.abc"

If (obj_OpenFileDialog1.ShowDialog()) = DialogResult.OK Then
    If Len(obj_OpenFileDialog1.FileName) <> 0 Then

        Dim f As Integer = FreeFile()
        Dim text As String

        FileOpen(f, obj_OpenFileDialog1.FileName, _
        OpenMode.Input)

        Do While Not EOF(f)
            text = text & LineInput(f) & vbCrLf
        Loop

        txtPole.Text = text
        FileClose(f)
    End If
End If
```

Protože budeme chtít otevírat soubor s příponou abc, bude dobré, když uživateli nabídneme snadnou cestu k vyhledání souboru a zjištění jeho jména. V tomto směru nám zcela jistě pomůže komponenta s názvem **OpenFileDialog**, pomocí které zobrazíme standardní dialog operačního systému Windows pro otevření souboru. Kdysi jsme si předvedli, jak přidat instanci komponenty na formulář v době návrhu aplikace, ovšem v této ukázce je instance komponenty vytvořená až za běhu programu. Poté, co je instance třídy **OpenFileDialog** vytvořena, můžeme použít její vlastnost **Filter**, a výslovně tak určit, jaký typ souborů se bude v dialogu zobrazovat. Samotný dialog zobrazíme zavoláním metody **ShowDialog** vytvořené instance. Jestliže uživatel klepne v dialogu na tlačítko OK, je metodou **ShowDialog** vrácena návratová hodnota **DialogResult.OK**.

Enumerace **DialogResult** je velmi užitečná v případě, kdy potřebujeme zjistit, na jaké tlačítko v dialogu uživatel klepnul. Mezi členy této enumerace patří: **Abort**, **Cancel**, **Ignore**, **No**, **None**, **OK**, **Retry** a **Yes**.

Interní hodnota členu **OK** enumerace **DialogResult** představuje v tomto případě celé číslo 1, proto by bylo možné nahradit zápis **DialogResult.OK** zápisem konstantní hodnoty 1.

Jestliže byl v dialogu vybrán jistý soubor (tuto skutečnost zjistíme změřením délky textového řetězce, jenž získáme z vlastnosti **FileName** instance **obj_OpenFileDialog1**), můžeme přejít k samotnému procesu otevření souboru a načtení jeho obsahu do textového pole. Protože je pochopení tohoto procesu zvláště důležité, ukážeme si jej po jednotlivých krocích:

1. Nejdříve je potřebné zavolat funkci **FreeFile** a prostřednictvím ní získat dostupné manipulační číslo souboru. Manipulační číslo slouží pro potřeby jednoznačné identifikace cílového souboru a v našem případě je uloženo do celočíselné proměnné **f**.

2. Pokračujeme deklarací textové proměnné **text**, do které bude později uložen celý obsah otevřeného souboru.

3. Pro otevření souboru použijeme funkci **FileOpen**. Aby funkce spolehlivě pracovala, musíme vyplnit její tři formální parametry:

   - manipulační číslo souboru (toto získáme z proměnné **f**),
   - jméno souboru (je uloženo ve vlastnosti **FileName** instance **obj_OpenFileDialog1**),
   - mód, ve kterém je soubor otevírán (**OpenMode.Input**).

   Když Visual Basic .NET otevírá soubor, musí vědět, jak má k otevřenému souboru přistupovat. Z tohoto důvodu je nutné určit příslušný přístupový mód. V naší ukázce je uveden mód **Input**, který říká, že budeme chtít data načítat. Jak za chvíli uvidíte, pro uložení dat do souboru použijeme mód **Output**.

4. Jestliže je soubor otevřen, již nám nic nebrání v tom, abychom jeho obsah uložili do proměnné **text** typu **String**. Jak si můžete všimnout, načítání dat probíhá v cyklu, který se ukončí, až bude dosáhnut konec souboru. Data ze souboru jsou načítána sekvenčně řádek po řádku (načtení každého řádku má na starosti funkce **LineInput**). Protože ale funkce **LineInput** nepřidává za každý načtený řádek znaky **Chr(13)**, resp. **Chr(13) + Chr(10)**, musíme tyto znaky přidat my pomocí konstanty **vbCrLf**. Každý řádek je po svém načtení uložen do proměnné **text** a po ukončení běhu cyklu je obsah této proměnné uložen do vlastnosti **Text** textového pole **txtPole**. Finálním bodem je uzavření souboru pomocí funkce **FileClose**. Aby funkce **FileClose** spolehlivě našla otevřený soubor, poskytneme jí manipulační číslo tohoto souboru.

Vytváření zpracovatele události btnUložit_Click

Kód, jenž se nachází v událostní proceduře **btnUložit_Click**, má tuto podobu:


```
Dim obj_SaveFileDialog1 As New SaveFileDialog()
obj_SaveFileDialog1.Filter = "Soubory ABC (*.abc)|*.abc"

If (obj_SaveFileDialog1.ShowDialog()) = DialogResult.OK Then
    If Len(obj_SaveFileDialog1.FileName) <> 0 Then
```

Programový kód pokračuje na následující straně

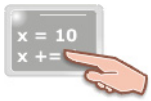
```
            Dim f As Integer = FreeFile()
            Dim text As String
            FileOpen(f, obj_SaveFileDialog1.FileName, OpenMode.Output)
            PrintLine(f, txtPole.Text)
            FileClose(f)
        End If
    End If
```

Struktura uvedeného zdrojového kódu je do značné míry podobná kódu, který jsme zadávali do zpracovatele události **btnOtevřít_Click**. Zaměřme se proto jenom na odlišné prvky:

1. Aby bylo možné zobrazit dialog pro uložení souboru, vytvoříme novou instanci třídy **SaveFileDialog**.
2. Nový soubor, v němž budou uložena zapsaná data, otevřeme v módu **OpenMode.Output**.
3. Obsah textového pole uložíme do souboru pomocí funkce **PrintLine**, které poskytneme manipulační číslo souboru a vlastnost **Text** textového pole **txtPole**.

Vytváření zpracovatele události btnVymazat_Click

Klepne-li uživatel na tlačítko s návěstím **Vymazat**, aplikace se podívá na obsah textového pole. Jestliže se v textovém poli nachází nějaký text, bude zobrazeno dialogové okno s výzvou, zdali si uživatel opravdu přeje vymazat obsah textového pole. Tuto charakteristiku implementuje následující výpis programového kódu:

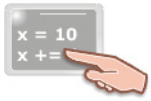



```
If Len(txtPole.Text) <> 0 Then
    If (MessageBox.Show("V textovém poli se nachází nějaký text." _
    & vbCrLf & "Chcete ho opravdu vymazat?", _
    "Vymazání textového pole", MessageBoxButtons.YesNo, _
     MessageBoxIcon.Exclamation, _
     MessageBoxDefaultButton.Button2)) = DialogResult.Yes Then
        txtPole.Text = ""
        txtPole.Focus()
    Else
    End If
End If
```

Obsah textového pole vymažeme velice jednoduše – stačí, když do vlastnosti **Text** pole umístíme prázdný řetězec (""). Zavoláním metody **Focus** zabezpečíme, že zaměření (focus) se přenese do právě „vyčištěného" textového pole, takže uživatel může začít zadávat nový text.

Vytváření zpracovatele události btnKonec_Click

Jestliže je aktivováno tlačítko pro ukončení aplikace, musíme zjistit, zdali se v textovém poli nenachází nějaký fragment textu. Je-li tomu tak, zobrazíme dialogové okno, které uživatele vyzve k uložení obsahu textového pole do souboru. Nebude-li uživatel chtít uložit text, klepne na tlačítko **No** (resp. **Ne**) a aplikace se ukončí zavoláním metody **Close** aktuální instance formuláře (**Me**). V této souvislosti ovšem musíme myslet také na skutečnost, kdy uživatel na tlačítko **Konec** klepne nechtěně, a proto zařazujeme i tlačítko pro zavření dialogu (**Cancel**, resp. **Zrušit**).




```
If Len(txtPole.Text) <> 0 Then
```

Programový kód pokračuje na následující straně

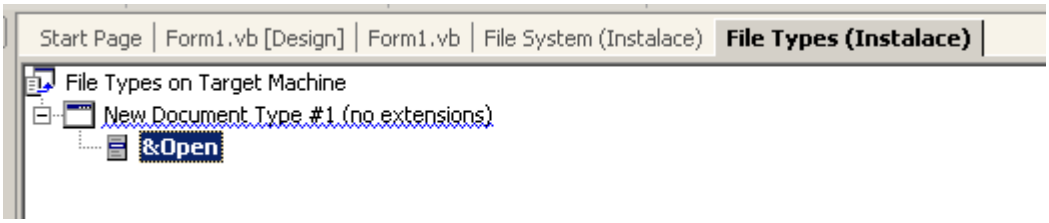
```
        Dim odpověď As Integer = MessageBox.Show( _
        "Obsah textového pole byl změněn." & vbCrLf & _
        "Chcete provedené změny uložit?", _
        "Uložit soubor", MessageBoxButtons.YesNoCancel, _
         MessageBoxIcon.Question, MessageBoxDefaultButton.Button1)

        If odpověď = DialogResult.Yes Then
            Call btnUložit_Click(Nothing, Nothing)
            Me.Close()
        ElseIf odpověď = DialogResult.No Then
            Me.Close()
        Else
        End If

    Else
        Me.Close()
    End If
```

Vytváření instalačního projektu

Jestliže máte připravenou testovací aplikaci, můžete se vrhnout na sestavení instalačního projektu.

Postupujte podle následujících instrukcí:

1. Vyberte nabídku **File**, ukažte na položku **Add Project** a klepněte na **New Project**.
2. V rámečku **Project Types** klepněte na položku **Setup and Deployment Project**.
3. Z rámečku **Templates** vyberte ikonu **Setup Wizard**.
4. Zadejte název pro instalační projekt a klepněte na tlačítko OK.
5. Následujte pokyny průvodce a sestavte kostru instalačního projektu.
6. V okně **Solution Explorer** klepněte pravým tlačítkem myši na název instalačního projektu. Když se objeví kontextová nabídka, ukažte na položku **View** a klepněte na subpoložku **File Types**. Integrované prostředí Visual Studia otevře editor **File Types**, který nám pomůže při vytváření asociace mezi aplikací s uživatelskými typy souborů.
7. V editoru klepněte pravým tlačítkem myši na položku **File Types on Target Machine** a z kontextové nabídky vyberte příkaz **Add File Type**.
8. Všimněte si, že pod položku **File Types on Taget Machine** se přidal uzel s názvem **New Document Type #1 (no extensions)** a pod uzlem se nachází akce s názvem **Open** (obr. 3).



Obr. 3 – Editor **File Types** po přidání nového typu souboru

9. Ujistěte se, že je vybrána položka **New Document Type #1 (no extensions)**. Přemístěte se do okna **Properties Window** a hodnotu vlastnosti **Name** nahraďte textovým řetězcem **Soubor ABC**.
10. Klepněte na název vlastnosti **Command** a následně klepněte na tlačítko se třemi tečkami (...). Zobrazí se dialogové okno s titulkem **Select Item in Project**. Poklepejte na položku **Application Folder** a vyberte položku **Primary output from AplikaceABC (Active)**.

Hodnotu vlastnosti **Command** tvoří název aplikace, která bude asociována s jistým typem souborů (v našem případě se soubory s extenzí abc). To znamená, že jestliže uživatel poklepe na soubor s příponou abc, bude spuštěna naše aplikace s názvem AplikaceABC. Aby ovšem mohla aplikace zpracovat data aktivovaného souboru, musíme ji speciálně upravit. Více se dozvíte v části **Implementace procedury Sub Main do testovací aplikace** dále v tomto textu.

11. Klepněte na tlačítko OK.
12. Do vlastnosti **Extensions** vložte textový řetězec **abc** pro určení souborové extenze. Přitom není nutné, abyste před název extenze uváděli také tečku (**.**).

Položka **Open**, která se nachází pod názvem typu souboru, představuje standardní akci, která se uskuteční poté, co uživatel poklepe na ikonu souboru v průzkumníkovi. Tato akce je rovněž nabídnuta uživateli poté, co uživatel klepne na ikonu souboru pravým tlačítkem myši.

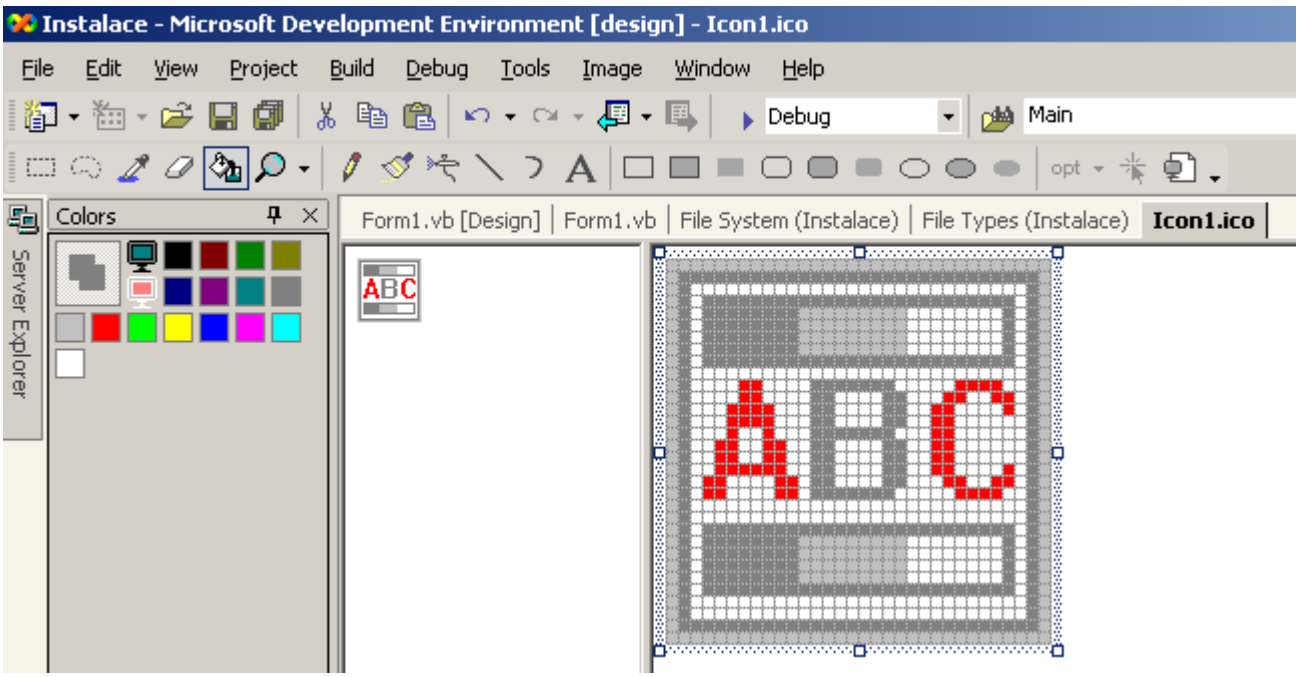
Vytváření vlastní ikony pro uživatelský typ souborů

V grafickém uživatelském prostředí operačního systému Windows má každý známý typ souboru svoji grafickou reprezentaci v podobě ikony. Protože ikona, její styl a barevné schéma umožňují uživatelům lépe a hlavně rychleji identifikovat typ cílového souboru, ukážeme si, jak náš uživatelský typ souboru opatřit grafickou ikonou. Budete-li následovat níže uvedené kroky, zcela jistě neminete cíl:

1. Vyberte nabídku **File**, ukažte na položku **New** a klepněte na položku **File**.

Stejného účinku můžete dosáhnout aplikací klávesové zkratky CTRL+SHIFT+N.

2. Ujistěte se, že v rámečku **Categories** je vybrána položka **General** a z rámečku **Templates** vyberte položku **Icon File**.
3. Pokud chcete, můžete vytvářenou ikonu pojmenovat, nebo ji nechejte implicitně pojmenovanou.
4. Klepněte na tlačítko **Open**. IDE otevře soubor s ikonou a nabídne vám, abyste podobu ikony upravili podle svých přání. I když se o správné tvorbě ikon vedou dlouhé diskuse, my si vystačíme s ikonou, která je zobrazena na obr. 4.



Obr. 4 – Vlastní ikona pro uživatelský typ souborů

5. Když jste s podobou ikony spokojeni, můžete ji uložit (**File>Save Icon1.ico**).
6. Vraťte se zpět do editoru **File Types**. V okně **Properties Window** klepněte na název vlastnosti **Icon**, aktivujte tlačítko s šipkou (▼) a vyberte položku **Browse**.
7. Uvidíte dialogové okno **Icon**. Když klepnete na tlačítko **Browse**, objeví se „staré známé" dialogové okno **Select Item In Project**.
8. Poklepejte na položku **Application Folder**.
9. Klikněte na tlačítko **Add File** a vyhledejte soubor s ikonou.
10. Otevřená dialogová okna pozavírejte klepnutím na příslušná tlačítka OK.

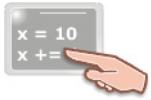
Implementace procedury Sub Main do testovací aplikace

Jak jsme si již řekli, když uživatel poklepe na soubor s extenzí abc, spustí se naše aplikace, která načte obsah souboru do textového pole. Doposud jsme si však nic nepověděli o implementačních detailech této techniky. Ovšem právě teď nastala ta správná chvíle. Na následujících řádcích si ukážeme, jak naprogramovat aplikaci tak, aby byla schopna provádět to, co od ní požadujeme.

První důležitou informací je, že programový kód, jenž bude odpovědný za načtení obsahu aktivovaného souboru, musí být uložen v proceduře **Sub Main**. Tato procedura není ve Visual Basicu implicitně přítomná, a proto ji musíme začlenit do kódu manuálně. Zde je postup:

1. V okně **Solution Explorer** klepněte pravým tlačítkem myši na název aplikace (**AplikaceABC**), ukažte na položku **Add** a klepněte na položku **Add Module**. Soubor s kódem modulu (**Module1.vb**) můžete nechat standardně pojmenovaný a stiskněte tlačítko **Open**.

2. Do modulu zadejte tento programový kód:




```
Public Sub Main(ByVal Argumenty() As String)

    If Argumenty.Length <> 0 Then
        Dim frm As New Form1()
```

Programový kód pokračuje na následující straně

```
            Dim Soubor As String = Argumenty(0)
            Dim f1 As Integer = FreeFile()
            FileOpen(f1, Soubor, OpenMode.Input)

            Dim text2 As String
            Do While Not EOF(f1)
                text2 = text2 & LineInput(f1) & vbCrLf
                frm.txtPole.Text = text2
            Loop

            FileClose(f1)
            Application.Run(frm)
        Else
            Application.Run(New Form1())
        End If
    End Sub
```

Toto je kód procedury **Sub Main**. Je velmi důležité si uvědomit, že jakmile do aplikace začleníme kód procedury **Sub Main**, stává se tato procedura vstupním bodem aplikace. To znamená, že po spuštění aplikace bude vyhledaná právě tato procedura a bude realizován kód, který procedura obsahuje. Při návrhu kódu, jenž bude tvořit kostru procedury **Sub Main**, musíme vzít v potaz skutečnost, že kód procedury musí spolehlivě ošetřovat dva stavy:

- **Standardní spuštění aplikace**, kdy je aplikace spuštěna poklepáním na spustitelný soubor (.exe), nebo je aktivován zástupce aplikace na ploše (jestliže existuje).

- **Podmíněné spuštění aplikace**. Při podmíněném spuštění aplikace není aplikace aktivována explicitně, nýbrž je aktivován soubor s extenzí, která je asociována s aplikací. Aplikaci je předána plně kvalifikovaná cesta k souboru, takže aplikace přesně ví, kde se cílový soubor nachází. Kromě plné cesty k souboru mohou být aplikaci předány také další informace (v podobě argumentů příkazového řádků).

O tom, který z uvedených stavů nastal, rozhoduje v proceduře **Sub Main** konstrukce **If**. V této rozhodovací konstrukci je testována vlastnost **Length** pole **Argumenty**. Jestliže je hodnota vlastnosti odlišná od nuly, můžeme se domnívat, že nastal druhý stav (podmíněné spuštění aplikace), v opačném případě půjde o standardní spuštění aplikace.

Jak si můžete všimnout, procedura **Sub Main** přijímá dynamické pole argumentů typu **String**. I když zde deklarujeme pole argumentů, nás bude zajímat především první prvek tohoto pole (**Argumenty (0)**), kterým je plná cesta k spuštěnému souboru (žádné další informace nebudeme v naší ukázce potřebovat). Tato cesta je uložena do proměnné **Soubor** datového typu **String**. Dalším neméně podstatným programovým prvkem je vytvoření instance třídy formuláře **Form1** (odkaz na tuto instanci je uložen do referenční proměnné **frm**). Proces otevření souboru jsme si již vysvětlili, takže víte, jak probíhá. Jakmile jsou data ze souboru načtena, dochází k uzavření souboru a spuštění samotné aplikace. Spuštění aplikace docílíme zavoláním metody **Run** třídy **Application**, které předáme odkaz na vytvořenou instanci třídy **Form1**.

Při standardním spuštění aplikace nebudou aplikaci předané žádné informace, a proto se provede programový kód, jenž se nachází za návěstím **Else** rozhodovací konstrukce **If**. Za uvedeným návěstím je umístěn jenom jeden řádek kódu, který spouští aplikaci a vytváří novou instanci třídy **Form1**.

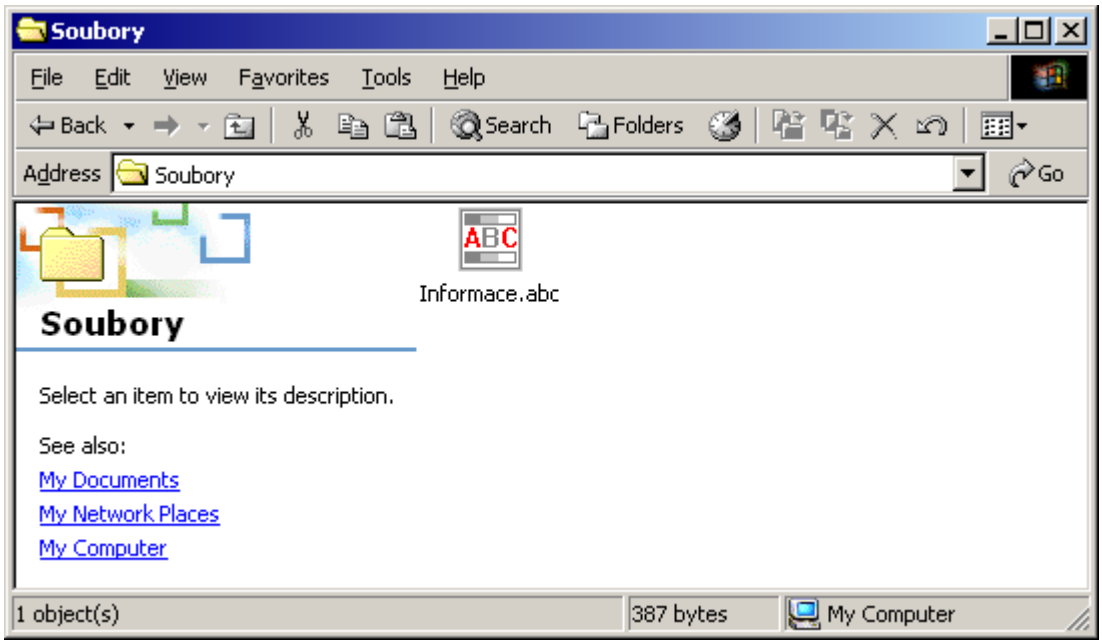
Poté, co zařadíme kód procedury **Sub Main** do testovací aplikace, musíme o této skutečnosti podat zprávu i Visual Basicu:

1. V okně **Solution Explorer** klikněte pravým tlačítkem myši na název aplikace (**AplikaceABC**) a z kontextové nabídky vyberte volbu **Properties**.
2. V okně **Property Pages** vyhledejte otevírací seznam **Startup object** a vyberte z něj položku **Sub Main**.
3. Klepněte na tlačítko OK.

Nakonec proveďte sestavení testovací aplikace a instalačního projektu.

Testování aplikace v reálném prostředí

Vyhledejte složku s instalačními soubory a poklepejte na ikonu souboru **Setup.exe**. Spustí se instalátor, který vás provede všemi potřebnými kroky instalace. Po ukončení instalace otevřete složku s nainstalovanými soubory aplikace a spusťte aplikaci s názvem **AplikaceABC**. Do textového pole zadejte nějaký text a klepněte na tlačítko **Uložit soubor**. Zadejte jméno souboru a uložte jej. V průzkumníku vyhledejte složku s uloženým souborem a všimněte si ikonu, kterou je soubor opatřen (obr. 5).



Obr. 5 – Ikona vytvořeného souboru s příponou abc

Ukončete aplikaci klepnutím na tlačítko **Konec** a na výzvu pro uložení souboru odpovězte záporně. V průzkumníku poklepejte na ikonu uloženého souboru. Uvidíte, že se opravdu spustí naše aplikace a do textového pole se načte obsah uloženého souboru. Skvělá práce!

Návrhy na vylepšení testovací aplikace

Ukázková aplikace **AplikaceABC** je, mírně řečeno, velmi jednoduchá, a proto byste možná uvítali pár postřehů neboli nápadů na její vylepšení. Budete-li chtít, můžete si vyzkoušet své programátorské dovednosti a modifikovat programovou strukturu tak, aby zahrnovala některé z uvedených vylepšení:

- Aplikace by v první řadě potřebovala zlepšit svoji aplikační logiku. Například při zjišťování, zdali byly v otevřeném souboru provedeny změny, je určitě lepší zařadit globální booleovskou proměnnou, která bude složit jako jednoduchý přepínač. Tato proměnná by mohla být implicitně inicializovaná na hodnotu **False** a jestliže uživatel učiní zásahy do obsahu souboru, hodnota proměnné by se změnila na **True**.

- Aby uživatel věděl, s jakým souborem pracuje, mohla by se cesta k právě editovanému souboru nacházet v titulkovém pruhu okna aplikace. Popřemýšlejte, zdali by bylo vhodné zobrazovat plně kvalifikovanou cestu k souboru, nebo jenom samotný název souboru.

- Při standardním spuštění aplikace by neměly být aktivní tlačítka **Uložit soubor** a **Vymazat**, protože když se aplikace spustí, je zobrazeno pouze prázdné textové pole, a tudíž není nic k ukládání nebo mazání.

- Pokud by aplikace chtěla být k uživateli přívětivá, zcela jistě by si měla „pamatovat" jména posledně otevřených souborů (tuto vlastnost dnes můžete najít v drtivé většině softwarových aplikací). Aplikace si přitom nemusí vést seznam všech otevřených souborů, uživatel bude jistě spokojen, když bude moci prohlížet seznam deseti posledně otevřených souborů.

- Seznam posledně otevřených souborů by měl být po ukončení aplikace zapsán do registrů operačního systému Windows a při dalším spuštění aplikace by mělo dojít k jeho automatickému načtení.

- Pokud by aplikace sloužila na tvorbu důvěrných dokumentů, bylo by vhodné obsah dokumentu ještě před jeho uložením podrobit šifrovacímu procesu. Samotný text můžete zašifrovat pomocí různých technik.

- Jsem přesvědčen, že potenciálních vylepšení aplikace **AplikaceABC** by se našlo ještě daleko víc. Zkuste na ně přijít sami!

Integrované prostředí rubriky Visual Basic	Autor: **Ján Hanák**

Lab

Téma

Začínáme s VB .NET

Úvod do světa .NET (10. díl)

Časová náročnost (min):	Začátečník	Pokročilý	Profesionál
60	1	2	3

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

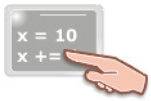
Vážení čtenáři,

v desáté lekci si ukážeme, jak Visual Basic .NET přistupuje k přetypování proměnných hodnotového typu na proměnné odkazového typu. Tento proces se označuje jako **boxing** hodnotové proměnné. Kromě boxingu si předvedeme také opačný proces – **unboxing**. Po zvládnutí těchto technik se budeme soustředit na úvod do operátorů ve Visual Basicu .NET. Přeji vám příjemné počtení.

Obsah

Proces přetypování proměnných hodnotového typu na proměnné odkazového typu (boxing)

Proces, při kterém dochází ke konverzi proměnné hodnotového datového typu do podoby proměnné odkazového typu, se nazývá **boxing** (český překlad není jednotný, a proto budeme raději používat původní anglický termín). Ve Visual Basicu .NET je **boxing** realizován implicitně, tedy bez jakéhokoliv dalšího zásahu ze strany programátora. Jak probíhá proces konverze si předvedeme na následujícím příkladu:



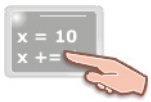

```
Dim x As Integer
Dim y As Object
Randomize()
x = CInt(100 * Rnd() + 1)
y = x
Me.Text = CStr(y)
```

Výpis zdrojového kódu představuje dvě proměnné: proměnnou **x** datového typu **Integer** a proměnnou **y** generického datového typu **Object**. Již víte, že proměnné typu **Integer** mohou nabývat platné celočíselné hodnoty. Proměnné generického datového typu **Object** jsou ve srovnání s celočíselnými proměnnými zcela jistě univerzálnější. Do těchto proměnných totiž můžete uložit cokoliv, tedy celočíselné hodnoty, desetinná čísla, text, znaky a jiné hodnoty. Je to proto, že datový typ **Object** je schopen pojmout obsah proměnné libovolného datového typu.

Tvrzení, že proměnné datového typu **Object** mohou uchovávat obsah proměnných libovolného datového typu, je po technické stránce poněkud nepřesné. Hodnoty proměnných datového typu **Object** jsou totiž ve skutečnosti ukládány jako 32bitové adresy, které ukazují na cílové objekty. Mezi cílové objekty můžeme zařadit jednak instance referenčních datových typů (jako jsou pole, třídy, rozhraní a datový typ **String**) a také zástupce hodnotových typů (celočíselné typy, typy s pohyblivou desetinnou čárkou, struktury, výčtové typy a také datové typy **Boolean**, **Char** a **Date**). Řečeno jinými slovy, v generické proměnné typu **Object** je uložen odkaz na objekt, který vznikl ze třídy, nebo byl zrozen v procesu, jemuž se říká **boxing**.

Velmi důležitou informací je, že všechny třídy a datové typy v prostředí .NET Frameworku, jsou odvozeny od bázové systémové třídy **System.Object**. Dalším podstatným faktem je, že datový typ **Object** je instancí bázové třídy **System.Object**. Když tedy spojíme uvedené teze do jedné věty, můžeme říci, že existuje jedna supertřída **System.Object**, od které jsou odvozeny všechny datové typy a třídy, které budeme při programování používat.

Pomocí příkazu **Randomize** inicializujeme generátor náhodných čísel a poté voláme funkci **Rnd**, abychom získali náhodné číslo z intervalu <1, 101>. Získané náhodné číslo je ještě před svým uložením do proměnné **x** převedeno do podoby celého čísla použitím konverzní funkce **CInt** (použití správné konverzní funkce je důležité zejména v případě, kdy máte aktivovanou volbu **Option Strict**). Nás bude ovšem nejvíce zajímat zejména proces průběhu následujícího přiřazovacího příkazu:



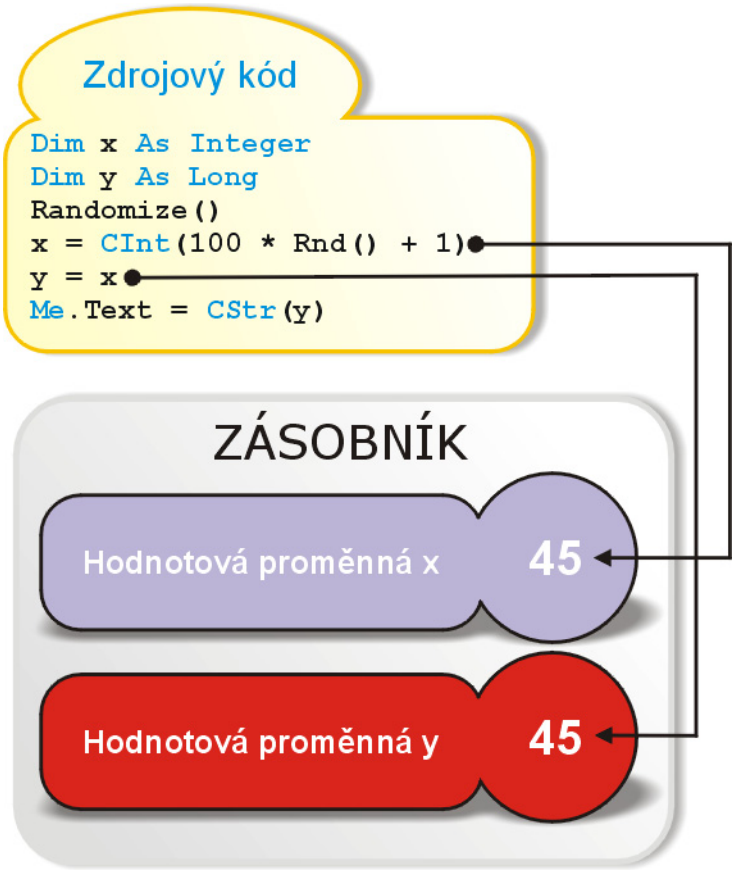

```
y = x
```

I když se to možná na první pohled nezdá, uvedený přiřazovací příkaz předvádí techniku přetypování hodnotové proměnné (**x** typu **Integer**) na odkazovou proměnnou (**y** typu **Object**). V rámci tohoto procesu jsou realizovány tyto operace:

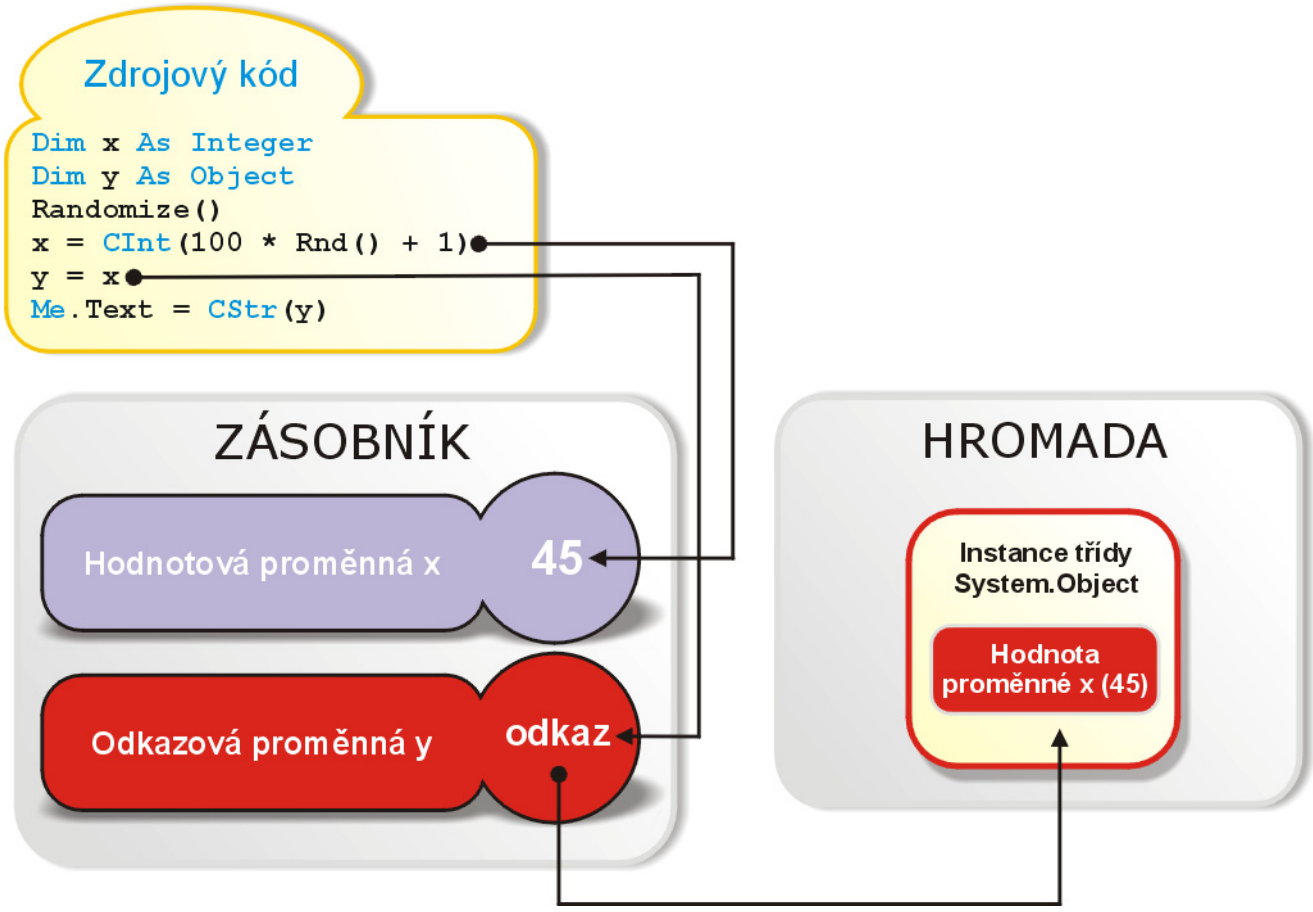
1. Je vytvořena instance třídy **System.Object**, a tato je uložena na řízené hromadě (**managed heap**).
2. Do zásobníku (**stack**) je přidána objektová referenční proměnná **y**, do které je umístěn odkaz na vytvořenou instanci třídy **System.Object**.
3. Je vytvořena kopie hodnoty proměnné **x**, která je poskytnuta vytvořené instanci třídy **System.Object**. Tato instance tedy obsahuje stejnou hodnotu, jaká je uložena v proměnné **x**.

Tyto tři kroky v zjednodušeném ponětí předvádějí techniku boxingu. Jak jsme si již řekli, při boxingu jde o přetypování hodnotové proměnné na referenční proměnnou. Při bližším pohledu ovšem zjistíme, že situace je daleko komplikovanější, protože je nutné vytvořit objekt na řízené hromadě a datovou položku objektu naplnit zkopírovanou hodnotou hodnotové proměnné (proměnné **x** v našem případě). Když převedeme boxing hodnotové proměnné, získáme objekt boxovaného typu.

Porovnejme přiřazovací příkaz s boxingem s běžným přiřazovacím příkazem, při kterém dochází k přiřazení hodnoty jedné hodnotové proměnné do jiné hodnotové proměnné (obr. 1).

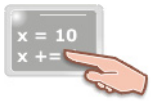


Obr. 1 – Běžný přiřazovací příkaz



Obr. 2 – Přiřazovací příkaz s boxingem

Boxing je v tomto případě uskutečňován implicitně, což znamená, že o všechny operace s ním spojené se postará Visual Basic sám. Boxing však můžeme vyvolat i explicitně:




```
Dim x As Integer
Dim y As Object
Randomize()
x = CInt(100 * Rnd() + 1)
y = CType(x, Object)
Me.Text = CStr(y)
```

Pokud budeme chtít, můžeme použít konverzní funkci **CType**, a tak názorně ukázat, že půjde o konverzi mezi hodnotovým a referenčním typem proměnné. Přestože je tento zápis lépe čitelný, není zapotřebí jej uvádět, protože Visual Basic provádí boxing implicitně.

Proces přetypování proměnných odkazového typu na proměnné hodnotového typu (unboxing)

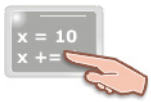
Protějškem boxingu je **unboxing** (opět budeme používat původní anglický termín). Dedukcí můžeme přijít ke skutečnosti, že unboxing představuje zpětné přetypování odkazové proměnné. Výsledkem unboxingu je tedy původní hodnotová proměnná.

Při realizaci unboxingu jsou prováděny tyto operace:

1. Je vyhledána požadovaná instance třídy **System.Object** a je podrobena testu, zdali skutečně obsahuje boxovanou hodnotu hodnotové proměnné. Z uvedeného tedy vyplývá, že unboxing může být proveden jenom nad takou hodnotovou proměnnou, která byla předtím podrobena boxingu.

2. Boxovaná hodnota, která je uložena společně s instancí, je zkopírována do specifikované hodnotové proměnné.

Techniku unboxingu si předvedeme na vylepšeném úvodním příkladu:




```
Dim x As Integer
Dim y As Object
Dim a As Integer
Randomize()
Dim f As Integer = FreeFile()
FileOpen(f, "c:\seznam.txt", OpenMode.Output) 'Otevření souboru.
For a = 1 To 10 'Zápis cyklu For.
    x = CInt(100 * Rnd() + 1)
    y = x 'Zde je realizován boxing ...
    x = CType(y, Integer) '... a zde zase unboxing.
    If a >= 1 And a < 10 Then 'Rozhodovací konstrukce.
        Print(f, CStr(y) & ", ")
    Else
        Print(f, CStr(y) & ".")
    End If
Next a
FileClose(f) 'Uzavření otevřeného souboru.
```

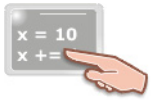
Pokud se vám zdá, že nevíte, co tento fragment programového kódu dělá, nepropadejte panice, zde je vysvětlení. Hlavním cílem tohoto krátkého programu je zjištění 10 náhodných čísel z intervalu <1,

101>. Ono anoncované vylepšení spočívá v tom, že seznam deseti vybraných čísel bude uložen do textového souboru (tento textový soubor má název seznam.txt a je uložen na disku C). Aby bylo možné jakákoliv data uložit do souboru, musí být tento nejprve vytvořen. Vytvoření a okamžité otevření souboru pro zápis má na starosti funkce **FileOpen**, která pracuje v našem případě se třemi parametry, jimiž jsou:

- Manipulační číslo souboru (toto číslo je vráceno funkcí **FreeFile** a uloženo do proměnné **f** datového typu **Integer**). Manipulační číslo souboru slouží na jednoznačnou identifikaci vytvářeného, resp. otevřeného souboru.
- Plná cesta k souboru, která je opatřena dvojitými uvozovkami. Jak si můžete všimnout, cesta pozůstává z určení cílového disku, názvu souboru a typu přípony (.txt).
- Mód, ve kterém má být soubor otevřen. V tomto případě je určen člen **Output** enumerace **OpenMode**, jenž specifikuje, že soubor bude otevřen pro zápis dat.

Jestliže je volání funkce **FileOpen** úspěšné, je vytvořen specifikovaný soubor, který je otevřen pro zápis údajů.

Ačkoliv jsme ještě neprobírali cykly, věřím, že malá ukázka jejich práce vám přinese příjemné osvěžení. V naší ukázce je použit cyklus **For-Next**. Zjednodušeně řečeno, cyklus **For-Next** provádí několikrát příkaz, nebo blok příkazů, které se nacházejí mezi klíčovými slovy **For** a **Next**. Počet opakování cyklu **For** je determinován řídící proměnnou **a**. Tato proměnná nabývá hodnot 1 až 10, přičemž maximální hodnota určuje počet průchodů cyklem. V těle cyklu je náhodně nalezeno číslo z požadovaného intervalu, které je přiřazeno do proměnné **x**. Následně dochází k přetypování hodnotové proměnné **x**, výsledkem kterého je vytvoření instance třídy **System.Object** s boxovanou hodnotou proměnné **x**. Příkaz:




```
x = CType(y, Integer)
```

provádí **unboxing** boxované hodnotové proměnné (boxovaná hodnota instance je zkopírována do hodnotové proměnné **x**).

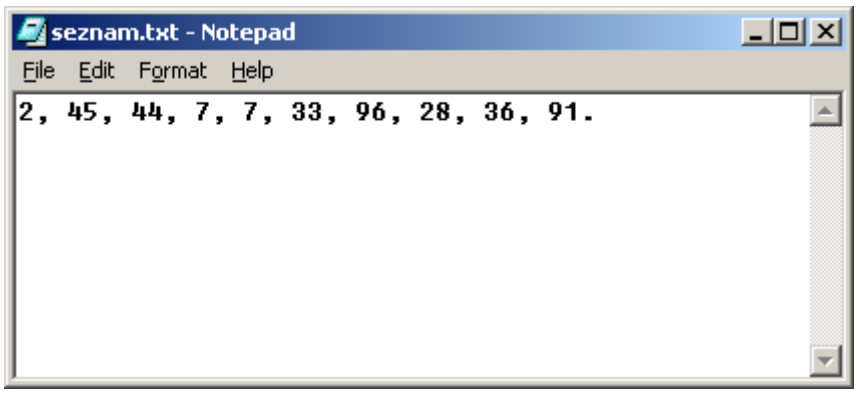
Zápis dat do souboru zabezpečuje funkce **Print**, která do souboru zapisuje řádek za řádkem (tento styl zápisu je označován jako sekvenční zápis dat). Funkci **Print** je předáno manipulační číslo souboru, do něhož má být nasměrován výstup a textový řetězec, jenž bude zapsán v jednom okamžiku.

Rozhodovací konstrukce **If** řídí tok zápisu dat, přičemž mezi jednotlivá čísla umísťuje čárky a za posledním číslem vkládá tečku.

Jestliže je soubor naplněn vybranými čísly, je uzavřen prostřednictvím funkce **FileClose** (funkci je předáno manipulační číslo souboru).

Zavření souboru po všech zamýšlených operacích je velmi, ale opravdu velmi důležité. Pokud by totiž soubor nebyl zavřen, mohlo by dojít k různým chybám (např. kdybyste se pokusili o jeho opětovné otevření, obdrželi byste chybové hlášení kompilátoru **System.IO.IOException**).

Pokud uvedený fragment zdrojového kódu zkopírujete do událostní procedury **Click** tlačítka, spustíte aplikaci a klepnete na tlačítko, na disku C bude vytvořen soubor s deseti náhodně vybranými čísly z intervalu <1, 101> (obr. 3).



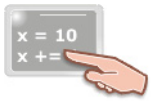
Obr. 3 – Textový soubor s náhodně vygenerovanými čísly

Mějte na paměti, že přílišné používání boxingu a zpětného unboxingu je spojeno s vyššími nároky aplikace na výpočetní výkon. Při boxingu se totiž vytváří instance třídy **System.Object**, do které je zkopírována hodnota hodnotové proměnné. Samotné vytvoření instance a následné kopírování hodnoty je ovšem pomalejší nežli přímé kopírování hodnot mezi dvěmi hodnotovými proměnnými (uloženými v zásobníku).

Operátory

Visual Basic .NET nabízí programátorům skutečně značný počet operátorů, které provádějí nejrůznější operace. Operátorem se ve světě programování rozumí prvek, který je aplikován na jeden, nebo i několik operandů a uskutečňuje s těmito operandy jistou, předem specifikovanou činnost. Komplexní celek, jenž je složen z operandů a operátorů, se nazývá výraz. Pro programátory je v mnoha případech důležitá zejména pravdivostní hodnota vyhodnocovaného výrazu (buď hodnota **True** nebo **False**), která se později může stát vstupní hodnotou pro potřeby rozhodovací konstrukce **If** nebo cyklů.

Na začátek si ukažme jednoduchý příklad použití operátoru a dvou operandů:



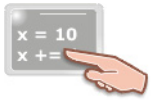

```
Dim číslo1, číslo2 As Integer
Dim Součin As Integer
Součin = číslo1 * číslo2
```

Zde můžete vidět příklad aritmetického operátoru pro násobení (*), jenž pracuje s dvěma operandy (operandy jsou celočíselné proměnné **číslo1** a **číslo2**). Operátor pro násobení je typem takzvaného binárního operátoru, protože pracuje se dvěma operandy. Práce operátoru je velice jednoduchá a probíhá přibližně podle následujícího algoritmu:

1. Je získána hodnota levého operandu (proměnné **číslo1**).
2. Je získána hodnota pravého operandu (proměnné **číslo2**).
3. Je vypočten součin hodnot obou operandů, který je současně výsledkem práce operátoru pro násobení.

Tím ovšem výčet operátorů příkladu nekončí, protože mezi operátory zařazujeme také operátor přiřazení (=). Smysl práce přiřazovacího operátoru jsme si vysvětlili již dávno. Jenom pro připomenutí si povězme, že operátor pro přiřazení přiřazuje hodnotu výrazu, jenž se nachází na pravé straně přiřazovacího příkazu do proměnné, která stojí na levé straně přiřazovacího příkazu.

Kromě binárních operátorů poznáme také operátory unární. Tyto pracují pouze s jedním operandem:




```
Dim a, b As Boolean, c As Short = 2
a = Not b
c = -c
Me.Text = "Proměnná a: " & a & " Proměnná c: " & c
```

V tomto výpisu programového kódu můžete spatřit dva unární operátory. Prvním z nich je logický operátor **Not**, který provádí logickou negaci svého operandu (tímto operandem je proměnná **b** datového typu **Boolean**). Proměnné datového typu **Boolean** jsou implicitně inicializované na pravdivostní hodnotu **False**. Použijeme-li operátor pro logickou negaci, převedeme původní pravdivostní hodnotu proměnné na její protějšek (tedy na hodnotu **True**). Druhým z prezentovaných unárních operátorů je takzvané unární mínus (**-**). Tento operátor uskutečňuje negaci hodnoty, která je uložena v celočíselné proměnné **c**. Proměnná **c** je při své deklaraci inicializována hodnotou 2, a po negaci této hodnoty bude platná hodnota proměnné **c** -2.

Mnoho začínajících programátorů si plete operátor unární mínus s aritmetickým operátorem pro odčítání. Zdrojem chybného rozlišení operátorů je pravděpodobně jejich vzájemná vizuální podoba. Mějte prosím na paměti, že operátor unární mínus je unárním operátorem, a vystačí si tedy s jenom jedním operandem. Na druhé straně, operátor pro odečítání je binárním operátorem, podobně jako operátor pro násobení (aby mohl pracovat, potřebuje dva operátory).

Kromě unárních a binárních operátorů existují rovněž ternární operátory. Ternární operátory ke své činnosti potřebují tři operandy a ve světě programování jsou poměrně vzácné, ve Visual Basicu se s nimi dokonce ani nestřetnete. Ternární operátor se nachází například v jazycích C a C++. Patříte-li mezi zvídavé programátory, můžete si prohlédnout ukázku použití ternárního operátoru **?:** v **Managed Extensions for C++**.

Pro zvídavé programátory: Použití ternárního operátoru **?:** v Managed Extensions for C++

Managed Extensions for C++ představují rozšíření programovacího jazyka Visual C++ .NET pro generování řízeného kódu pro Common Language Runtime a .NET Framework. V Managed Extensions for C++ můžeme použít ternární operátor **?:**. Tento operátor pracuje s třemi operandy a jak za chvíli uvidíte, je schopen vhodně substituovat jednoduchou rozhodovací konstrukci **If**. Operátor **?:** má tuto generickou podobu:

logický výraz ? výraz1 : výraz2 ;

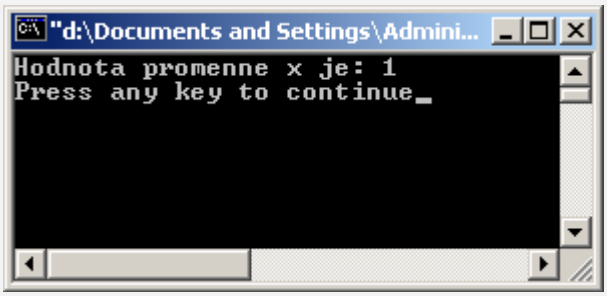
Prvním operandem je výraz, jehož hodnotu lze implicitně konvertovat do podoby datového typu **bool** (datový typ **bool** je v Managed Extensions for C++ ekvivalentem systémového typu **System.Boolean**). Jestliže je zkonvertovaná hodnota logického výrazu rovna pravdivostní hodnotě **True**, bude vykonán programový kód druhého operandu (**výraz1**), v opačném případě bude vykonán kód třetího operandu (**výraz2**).

A nyní je již nejvyšší čas pro praktickou ukázku ternárního operátoru:

```
int _tmain(void)
{
    int a = 5, x;
       a > 10 ? x = 0 : x = 1;
       Console::Write(S"Hodnota promenne x je: ");
       Console::WriteLine(x);
    return 0;
}
```

Tento výpis zdrojového kódu předvádí funkci **_tmain**, která je implicitním vstupním bodem aplikace typu Managed C++ Application. V těle funkce jsou deklarované dvě proměnné datového typu **int** (datový typ **int** je ekvivalentem systémového datového typu **System.Int32**). Proměnná **a** je okamžitě po deklaraci inicializována na hodnotu 5, ovšem proměnná **x** není explicitně inicializována (je jí přiřazena implicitní inicializační hodnota). Všimněte si zejména použití ternárního operátoru **?:**. Jestliže je logický výraz **a > 10** pravdivý, do proměnné **x** bude uložena nulová hodnota, jinak se proměnná **x** naplní hodnotou 1. V našem

případě je pravdivostní hodnota logického výrazu **False**, protože do proměnné **a** jsme přiřadili hodnotu 5, která není větší než hodnota 10. Výsledkem práce operátoru je tedy uložení hodnoty 1 do proměnné **x** (viz obrázek).



Integrované prostředí rubriky Visual Basic	Autor: Ján Hanák

Téma

VB .NET

Programátorská laboratoř

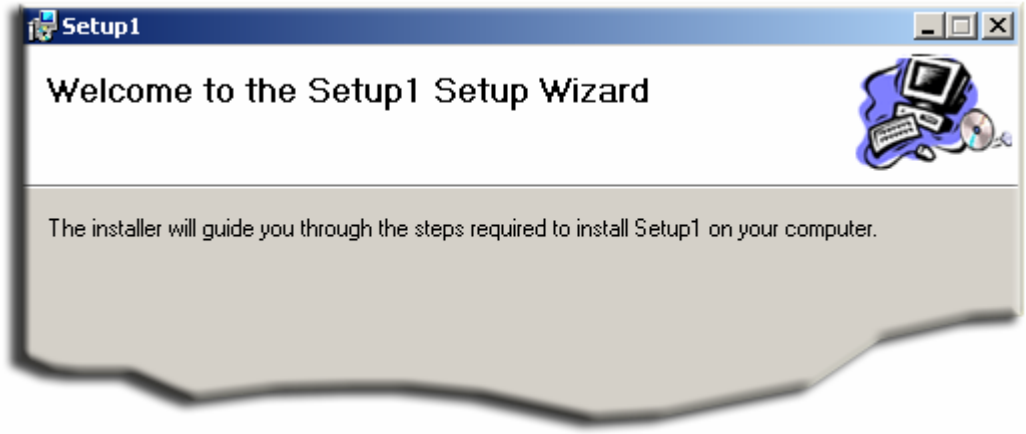
Prostor pro experimentování

Časová náročnost (min):	Začátečník	Pokročilý	Profesionál
85	1	2	3

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

Vlastní obrázek při instalaci aplikace

Zdaleka ne všichni programátoři vědí, že integrované vývojové prostředí Visual Studia .NET jim umožňuje opatřit instalaci aplikace vlastní grafikou. Jestliže jste již experimentovali s tvorbou distribučních jednotek aplikací, zcela jistě jste si všimli, že v horní části dialogového okna instalátoru se nachází grafický obrázek (obr. 1).



Obr. 1 – Dialogové okno instalátoru s obrázkem v horní části

Dobrou zprávou je, že můžete standardně dodávaný obrázek nahradit svým vlastním obrázkem. Abyste dosáhli co možná nejlepších výsledků, pamatujte na následující rady:

- Obrázek musí být uložen v souboru bmp nebo jpeg. Protože není omezena barevná hloubka obrázku, můžete používat bohaté a syté barvy v 16 a 32-bitovém provedení.

- Velikost regionu obrázku by měla být 500x70 pixelů. Obrázek by tedy měl být 500 pixelů široký a 70 pixelů vysoký. Tyto hodnoty jsou doporučované, a proto by bylo více než vhodné, kdybyste je dodržovali. Můžete sice použít i jinou velikost regionu obrázku, ovšem v tomto případě bude obrázek přizpůsoben výchozím hodnotám (dojde k jeho zkreslení).

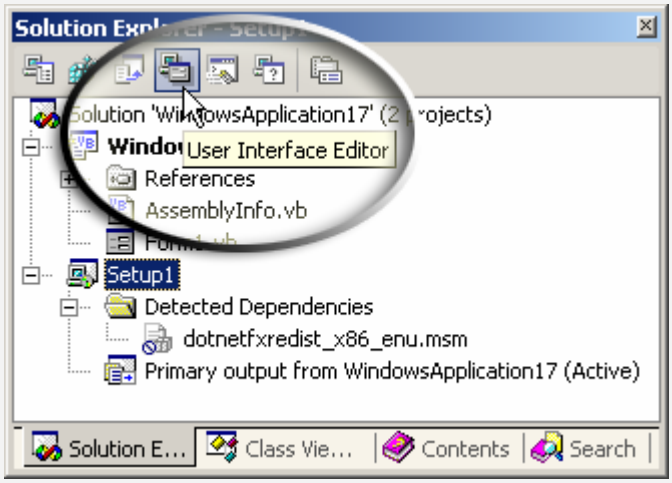
- Jak si můžete všimnout z obr. 1, velkou část plochy obrázku pokrývá bílá barva a jenom v pravé části obrázku se nachází grafika. Toto schéma rozložení grafiky v obrázku je odvozeno od potřeby zobrazení textu instalátoru. Tento text je poměrně dlouhý a jelikož je jeho barva černá, je nejlépe viditelná právě na bílém pozadí. I když je možné zaplnit celou plochu obrázku grafikou, není takovýto postup doporučován. Nejlepší proto bude, když svoji vlastní grafiku umístíte do pravé části obrázkového regionu. Pokud opravdu potřebuje grafikou pokrýt velkou část obrázkového regionu, zkuste zvolit jemné a takřka průhledné barevné ztvárnění tak, aby byl text instalátoru dobře čitelný.

Obrázek o rozměrech 500x70 pixelů si můžete připravit ve vašem oblíbeném programu pro tvorbu grafických předloh.

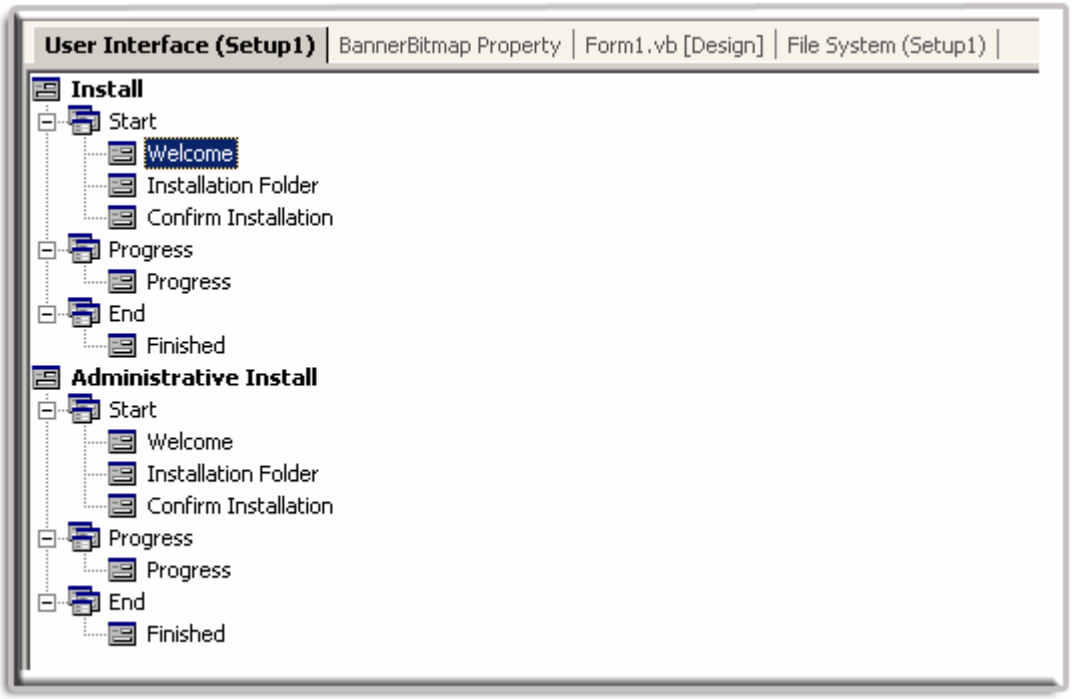
V následujícím postupu se předpokládá, že jste vytvořili instalační projekt, který jste přidali do řešení vaší aplikace.

1. V okně **Solution Explorer** klepněte pravým tlačítkem myši na název instalačního projektu.
2. Když se objeví kontextová nabídka, ukažte na položku **View** a poté klepněte na příkaz **User Interface**. Integrované prostředí Visual Studia .NET otevře editor **User Interface**.

Editor **User Interface** můžete zobrazit i rychleji. Stačí, když v okně **Solution Explorer** vyberete jméno instalačního projektu a klepněte na ikonu **User Interface**, která se nachází v horní části okna (viz obrázek).



3. Podobu editoru **User Interface** můžete vidět na obr. 2.



Obr. 2 – Editor **User Interface**

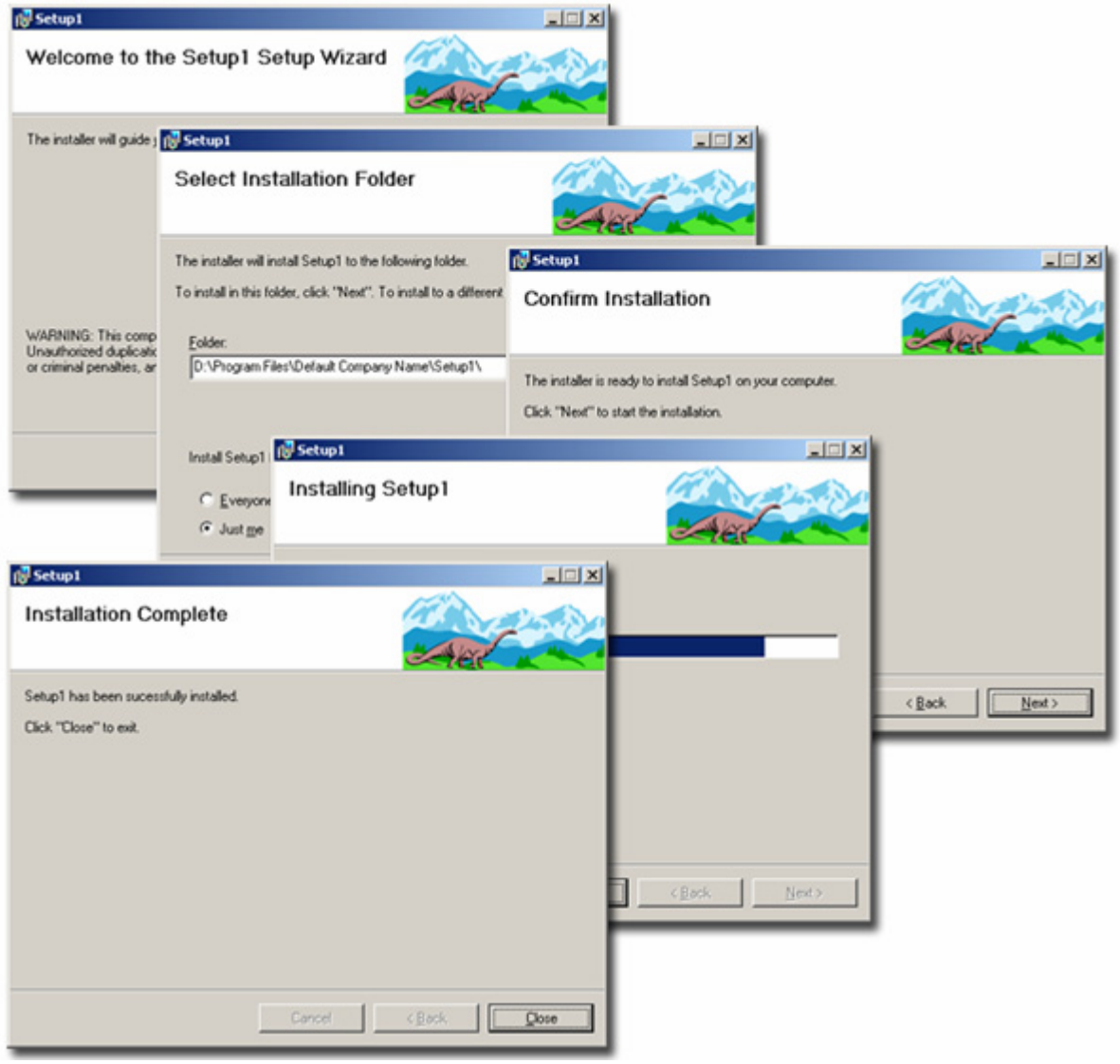
Editor **User Interface** umožňuje provést modifikaci grafického uživatelského rozhraní instalačního procesu. V okně editoru se nacházejí dvě stromové struktury pro dva odlišné typy instalací. Jednou z nich je standardní instalace (pojmenovaná jako **Install**) a tou druhou je administrativní instalace (pojmenovaná jako **Administrative Install**). Všimněte si, že oba typy instalací jsou tvořeny ze tří hlavních instalačních etap:

- Zahájení instalace (uzel **Start**)
- Jádro instalace (uzel **Progress**)
- Ukončení instalace (uzel **End**)

Každá etapa obsahuje jeden nebo i několik kroků, které jsou postupně uskutečňované v okamžiku, kdy dochází k vykonání kódu příslušné etapy. Tyto kroky jsou ve stromové struktuře editoru znázorněny speciálními ikonkami a jsou podřazeny etapě, do které patří.

4. Klepněte na položku **Welcome**, která se nachází pod uzlem **Start** standardní instalace (**Install**).
5. V okně **Properties Window** klepněte na název vlastnosti **BannerBitmap**. Dále klikněte na tlačítko se šipkou (▼) a ze seznamu vyberte položku **Browse**.
6. V dialogovém okně **Select Item in Project** poklepejte na položku s názvem **Application Folder**.
7. Stiskněte tlačítko **Add File** a vyhledejte váš grafický soubor s obrázkem pro instalaci.
8. Dialogové okno **Select Item in Project** zavřete klepnutím na tlačítko OK.
9. Tímto jste přiřadili obrázek pouze prvnímu kroku instalačního procesu. To znamená, že obrázek bude zobrazen pouze v prvním kroku instalátoru. To ovšem nestačí, my potřebujeme, aby byl obrázek viditelný po celou dobu instalace. Proto je zapotřebí správně nastavit vlastnost **BannerBitmap** také u všech ostatních kroků instalace (přesněji se jedná o kroky **Installation Folder**, **Confirm Installation**, **Progress** a **Finished**).
10. Klepněte na ikonu **Installation Folder**, která se nachází pod uzlem **Start**.
11. V okně **Properties Windows** klepněte na vlastnost **BannerBitmap**, dále klepněte na šipku (▼) a zvolte položku **Browse**.
12. V dialogovém okně **Select Item in Project** poklepejte na položku **Application Folder**.
13. Označte název grafického souboru a klepněte na tlačítko OK.
14. Opakujte kroky 10 až 13 pro nastavení vlastnosti **BannerBitmap** u zbývajících kroků instalátoru (**Confirm Installation**, **Progress** a **Finished**).

15. V okně **Solution Explorer** klepněte na název instalačního projektu a z kontextového menu vyberte příkaz **Build**.
16. Po sestavení instalačního projektu klepněte na jeho název ještě jednou pravým tlačítkem myši a zvolte položku **Install**. Tím se zahájí spuštění instalátoru. Všimněte si, že všechny kroky instalátoru disponují novým grafickým ztvárněním (obr. 3).



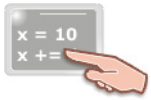
Obr. 3 – Obrázek, jenž je viditelný během celého průběhu instalačního procesu

Zastínění metody bázové třídy

Visual Basic .NET nám kromě překrytí metody bázové třídy nabízí taktéž možnost zastíněný této metody. Zatímco při překrytí metody bázové třídy musí mít metoda odvozené třídy stejný název, signaturu, návratovou hodnotu a modifikátor přístupu, při zastínění stačí, když metoda odvozené třídy má jenom stejný název jako metoda bázové třídy, kterou hodláme zastínit. To tedy znamená, že můžeme vybudovat metodu, která se liší následujícími aspekty:

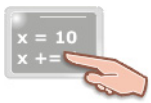
- signaturou (seznamem parametrů),
- typem návratové hodnoty,
- modifikátorem přístupu (např. **Friend**).

Následující fragment programového kódu ukazuje, jak zastínit metodu bázové třídy (je uveden celý obsah souboru **Class1.vb**):



```
Option Strict On
Public Class Kostka
    Public Sub HoditKostkou()
        MessageBox.Show("Byla zavolána metoda HoditKostkou" _
        & " bázové třídy.")
    End Sub
End Class

Public Class Kostka2
    Inherits Kostka
    Friend Shadows Sub HoditKostkou(ByVal PočetHodů As Byte)
        If PočetHodů <= 0 Or PočetHodů >= 256 Then
            MessageBox.Show("Zadali jste nepřípustnou hodnotu.", _
            "Chyba při zadání počtu hodů", MessageBoxButtons.OK, _
             MessageBoxIcon.Warning)
        Else
            Dim i As Byte
            For i = 1 To PočetHodů
                Randomize()
                Dim Hodnota As Byte
                Hodnota = CByte(Int((6 * Rnd()) + 1))
                MessageBox.Show("Bylo hozeno číslo " & Hodnota & ".", _
                "Zpráva o výsledku")
            Next i
        End If
    End Sub
End Class
```

V kódu můžete vidět deklarace dvou tříd: třídy **Kostka** a třídy **Kostka2**. Třída **Kostka** obsahuje jednu veřejnou Sub proceduru s názvem **HoditKostkou**. Jak jste si již asi domysleli, budeme simulovat házení kostkou. Metoda **HoditKostkou** bázové třídy ovšem neimplementuje žádnou logiku pro simulaci hodu – namísto toho jenom zobrazuje dialogové okno se zprávou. Programový kód, který opravdu simuluje hru s kostkou, se nachází v proceduře **HoditKostkou** odvozené třídy s názvem **Kostka2**. Velmi dobře se podívejte na deklaraci procedury **HoditKostkou** v odvozené třídě:

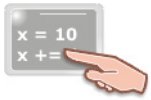


```
Friend Shadows Sub HoditKostkou(ByVal PočetHodů As Byte)
```

Klíčové slovo **Shadows** v deklaraci naznačuje, že tato procedura zastiňuje proceduru **HoditKostkou** bázové třídy. Všimněte si, že deklarace procedury **HoditKostkou** v odvozené třídě se od své kolegyně z bázové třídy liší:

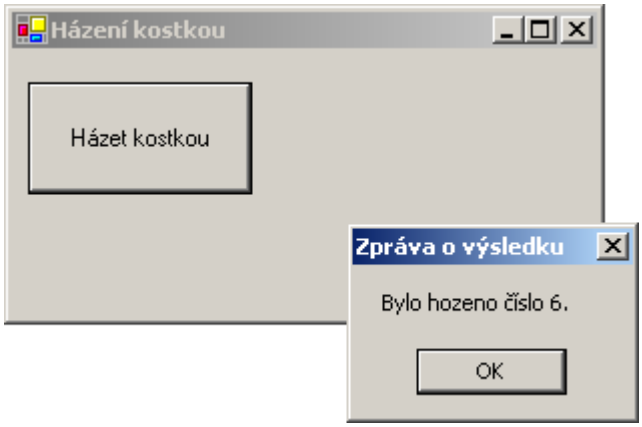
- **Modifikátorem přístupu** (Zde je použit modifikátor **Friend**, což znamená, že třída a její členy budou viditelné pouze z dané assembly).
- **Signaturou** (Procedura pracuje s jedním parametrem typu **Byte**, který určuje, kolikrát se má kostkou házet).

Budete-li chtít otestovat, zdali procedura odvozené třídy spolehlivě zastiňuje proceduru bázové třídy, přidejte na formulář jednu instanci ovládacího prvku **Button** a její událostní proceduru **Button1_Click** vyplňte následujícím kódem:




```
Dim obj As Kostka2
obj = New Kostka2()
obj.HoditKostkou(3)
```

Spusťte aplikace a klepněte na tlačítko. Protože jsme zadali, aby počítač házel kostkou třikrát, budou zobrazena tři dialogová okna se zprávou o každém hodu (obr. 4).



Obr. 4 – Zastínění metody bázové třídy v praxi

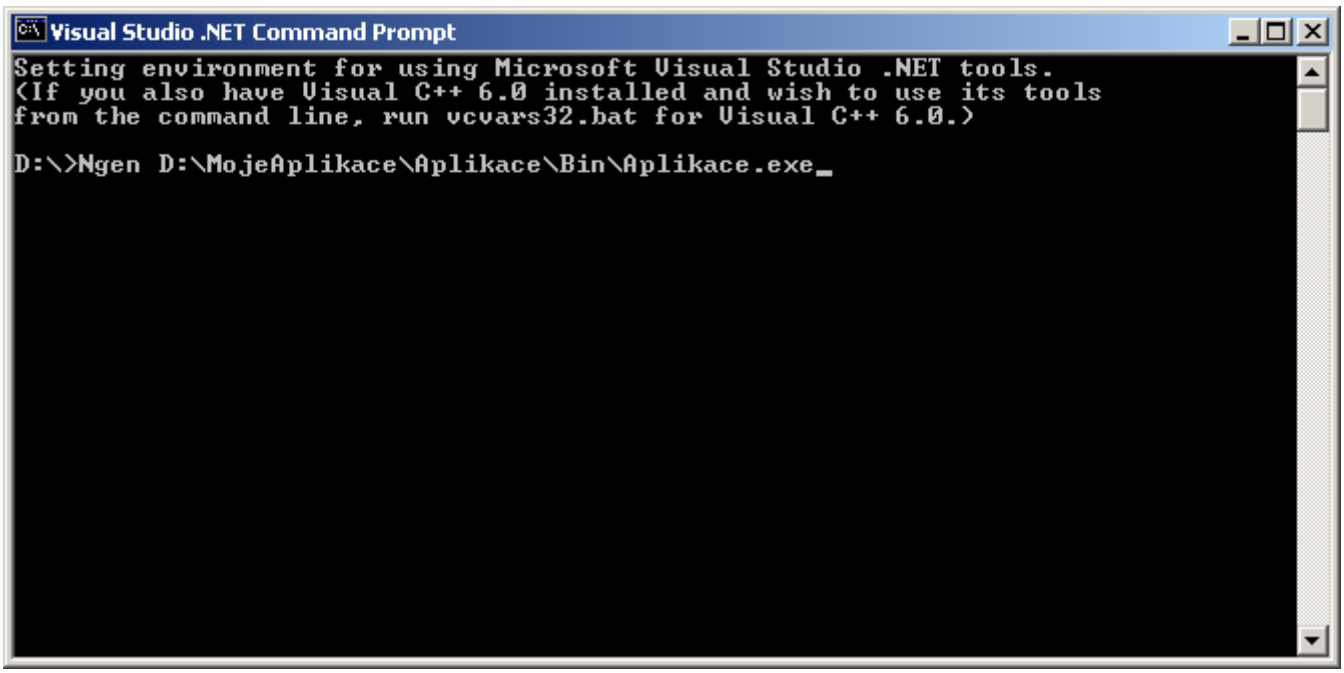
Kompilace MSIL kódu assembly do strojového kódu pomocí utility Native Image Generator (Ngen)

Výsledkem práce kompilátoru a linkeru je v prostředí .NET platformy samostatný celek s názvem assembly. Assembly, kromě jiného, obsahuje také kód přeložené aplikace, ovšem nikoli v podobě strojového kódu, nýbrž v podobě **MSIL** kódu. Po spuštění assembly jsou potřebné instrukce jazyka **MSIL** přeloženy do nativního kódu, jemuž rozumí instrukční sada procesoru (překlad **MSIL** kódu je realizován na požádání prostřednictvím **Just-In-Time** kompilátoru). V mnoha běžných situacích je uvedený proces překladu aplikace vyhovující a plně uspokojuje také požadavky na konstantní výkon aplikace. V jistých kritických okamžicích je ovšem zapotřebí dosáhnout maximální rychlost provádění kódu aplikace tak, že **MSIL** kód assembly přeložíme přímo do nativního kódu procesoru.

Kompilaci **MSIL** kódu assembly do podoby strojového kódu můžeme uskutečnit pomocí utility **Native Image Generator**. Tato utilita se spouští z příkazového řádku Visual Studia .NET a není tedy začleněna přímo do integrovaného vývojového prostředí. Výsledkem činnosti programu **Native Image Generator** je strojový kód assembly, jenž může být ihned po spuštění podroben přímé exekuci. Vygenerovaný soubor s nativním obrazem assembly je také instalován do **Native Image Cache**.

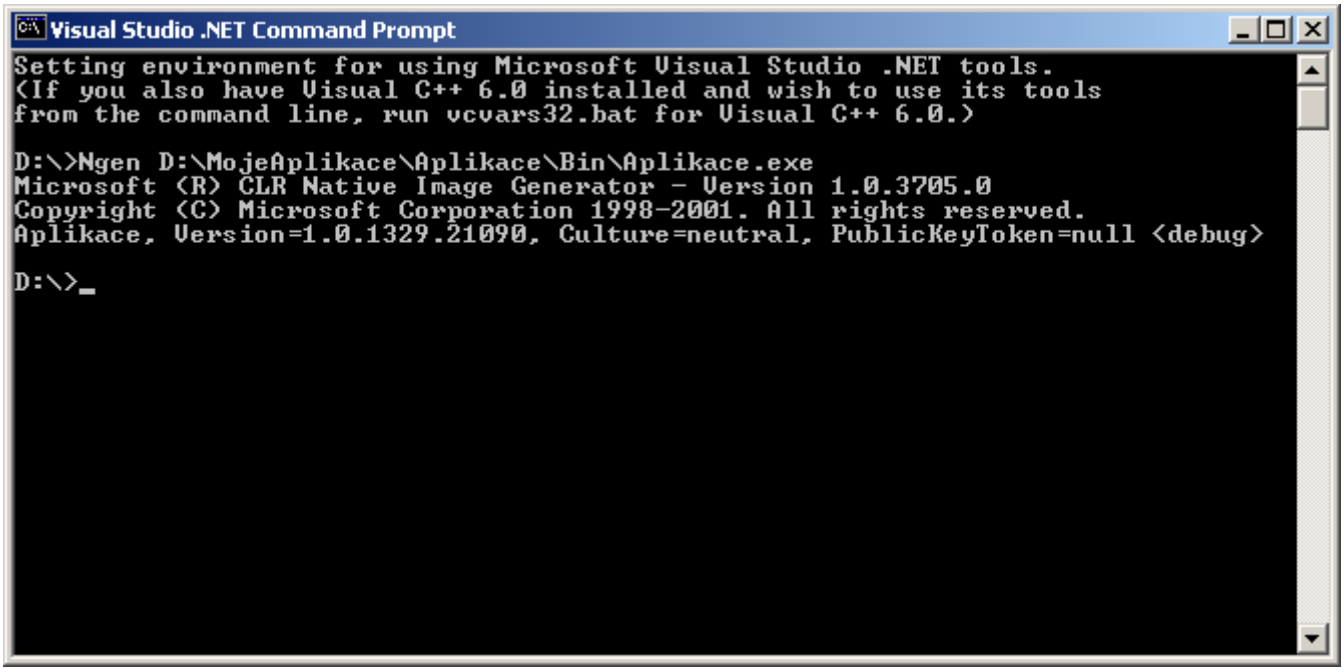
Pro kompilaci MSIL kódu assembly do strojového kódu udělejte toto:

1. Ve Visual Studiu .NET vyberte nabídku **Build** a vyberte příkaz **Build Solution**, pomocí kterého sestavíte assembly vašeho řešení.
2. Minimalizujte okno Visual Studia .NET.
3. Klepněte na nabídku **Start**, ukažte na nabídku **Programs** a vyhledejte nabídku **Visual Studio .NET**.
4. Pokračujte vybráním nabídky **Visual Studio. NET Tools** a konečně klikněte na položku **Visual Studio .NET Command Prompt**.
5. Po aktivaci položky se spustí příkazový řádek Visual Studia .NET. Po spuštění bude zobrazen pouze název kořenového disku (např. D:\). Zadejte název kompilačního programu (**Ngen**) a cestu k vaší assembly (obr. 5).



Obr. 5 – Použití utility **Native Image Generator**

6. Ujistěte se, že cesta k assembly, kterou jste zadali, je opravdu správná. Poté stiskněte klávesu Enter, čímž se spustí utilita **Native Image Generator**, která převede **MSIL** kód assembly do nativního kódu procesoru (obr. 6).



Obr. 6 – Výsledek práce utility **Native Image Generator**

7. Uzavřete okno **Visual Studia .NET Command Prompt**.
8. Vyhledejte vaší assembly a spusťte ji. Měli byste zaznamenat citelné zvýšení rychlosti, s jakou se assembly spouští.

Na závěr si vás dovolím upozornit na některé důležité aspekty práce utility **Ngen**:

- Při kompilaci assembly pomocí utility **Ngen** se může stát, že některé metody nebude umět tato utilita zpracovat. Tyto metody budou následně vyloučeny z kompilace a nebudou tedy převedeny do nativního kódu (budou uloženy v **MSIL** kódu). Jestliže budou někdy v průběhu

exekuce aplikace tyto metody volány, bude aktivován **JIT** kompilátor, který přeloží **MSIL** kód metod do podoby nativního kódu procesoru.

- Když používáte program **Native Image Generator** pro sestavení nativního obrazu kódu assembly, výsledek práce programu bude ovlivněn následujícími faktory:
    - Typem CPU zdrojového počítače (přesněji typem instrukční sady CPU).
    - Verzí operačního systému Windows.
    - Formou zdrojové assembly a také všech assembly, na které se zdrojová assembly odkazuje.
    - Bezpečnostními nařízeními.
- .NET Framework verze 1.0 nepodporuje automatickou tvorbu a odstranění nativních obrazů assembly, které se získají použitím utility **Ngen**. Proto je zapotřebí realizovat tyto operace manuálně.

Určení pořadí, v jakém ovládací prvky získávají zaměření (focus)

Tento tip pomůže zejména začínajícím programátorům, avšak ušetří trochu práce také jejich zkušenějším kolegům. Pokud je váš formulář v režimu návrhu aplikace přeplněn množstvím ovládacích prvků, může být určování pořadí, v jakém tyto ovládací prvky získávají zaměření (focus) poněkud obtížné. My si ukážeme, jak tuto nezáživní práci zvládnout s noblesou, která je vlastní integrovanému vývojovému prostředí Visual Studia .NET.

Je-li váš formulář opravdu plný instancí ovládacích prvků, možná byste se měli zamyslet nad jeho reorganizací. Promyslete si, zda skutečně potřebujete tolik instancí ovládacích prvků. Funkčně podobné ovládací prvky můžete seskupit do rámečku (ovládací prvek **GroupBox**).

Jestliže jste se ještě nestřetli s pojmem zaměření, nebo s jeho anglickým ekvivalentem focus, zde je vysvětlení. Každý ovládací prvek, který umístíte na formulář, může získat zaměření, když je uživatelem označen. Označení prvku většinou nenastává samoúčelně, nýbrž jde ruku v ruce s jinou událostí (např. s klepnutím na ovládací prvek). Skutečnost, že ovládací prvek získal zaměření, je i vizuálně umocněna (např. na ploše tlačítka se objeví tečkovaný obdélníkový region (viz obrázek)):

Ne všechny ovládací prvky ovšem mají schopnost získat zaměření (např. ovládací prvek **PictureBox** touto schopností nedisponuje). Visual Basic .NET vám také umožňuje zakázat, aby ovládací prvek implicitně přijímal zaměření (nastavením vlastnosti **TabStop** ovládacího prvku na hodnotu **False**).

Přecházet po ovládacích prvcích na formuláři můžete použitím tabulátoru. Stisknete-li klávesu Tab, zaměření se přenese z aktivního ovládacího prvku na následující ovládací prvek.

Při návrhu vzhledu formuláře Visual Basic sleduje pořadí, v jakém umísťujete jednotlivé ovládací prvky na plochu formuláře. Podle tohoto pořadí potom Visual Basic vyplňuje hodnoty vlastností **TabIndex** všech ovládacích prvků. Ovládací prvek, jenž byl na formulář umístěn jako první, má nulovou hodnotu vlastnosti **TabIndex** a každý později umístěný ovládací prvek pak disponuje hodnotou o jednotku větší.

Z uvedeného vyplývá, že když si nejprve důkladně promyslíte návrh formuláře a až poté začnete s „kreslením" ovládacích prvků, neměli byste narazit na žádné potíže. Problémy ovšem nastávají

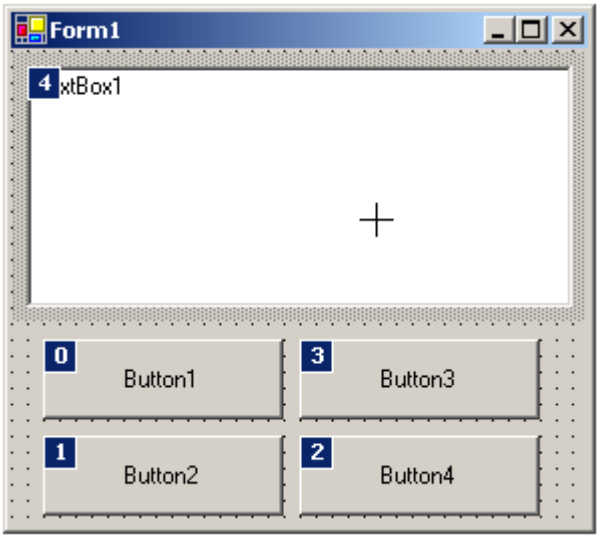
tehdy, kdy začnete měnit pozici ovládacích prvků a ještě k tomu přidáváte na formulář nové ovládací prvky.

Pořadí, v jakém ovládací prvky získávají zaměření, možno nastavit třemi způsoby:

1. Manuální modifikací vlastnosti **TabIndex** u všech ovládacích prvků, u kterých je to zapotřebí.
2. Nastavením vlastnosti **TabIndex** ovládacích prvků programově v událostní proceduře **Load** formuláře.
3. Použitím nové funkce IDE s názvem **Tab Order**.

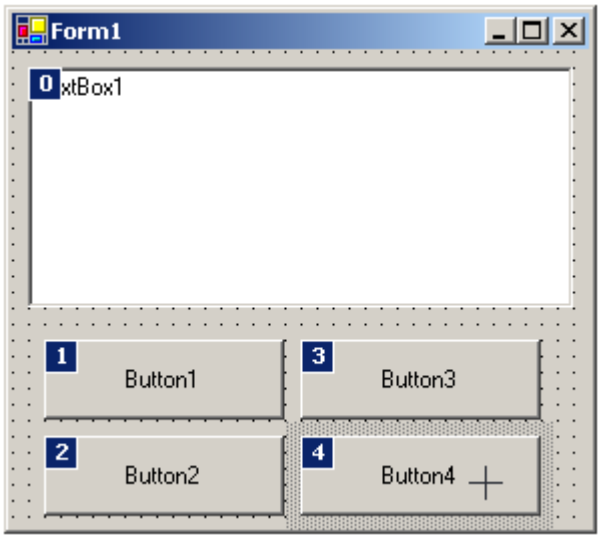
My si předvedeme třetí možnost, kterou je použití funkce **Tab Order** integrovaného vývojového prostředí. Postupujte takto:

1. Přesvědčete se, že aplikace se nachází v režimu návrhu.
2. Vyberte nabídku **View** a klepněte na položku **Tab Order**.
3. Kurzor myši se změní na křížek a každý ovládací prvek bude identifikován číslem, které se zobrazí v jeho levém horním rohu (obr. 7).



Obr. 7 – Aktivace funkce **Tab Order**

4. Nyní jednoduše klepněte na ten ovládací prvek, který má získat zaměření jako první.
5. Dále klepněte na prvek, který bude získávat zaměření jako druhý a tak dále, až kým neaktivujete všechny zbývající ovládací prvky. Výsledek ukázkového příkladu můžete vidět na obr. 8.



Obr. 8 – Nové pořadí zaměření ovládacích prvků formuláře

6. Abyste zrušili působení funkce **Tab Order**, vyberte nabídku **View** a klepněte na položku **Tab Order**.

Integrované prostředí rubriky Visual Basic	Autor: **Ján Hanák**

VB .NET

Lab

Téma měsíce

Programování **vícevláknových** aplikací

Časová náročnost (min):	Začátečník	Pokročilý	Profesionál
65	1	2	3

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

Milí čtenáři,

32bitové prostředí operačního systému Windows s sebou přineslo několik podstatných a zásadních změn, a to nejen z uživatelského, nýbrž i z programovacího pohledu. Na přední příčku v řebříčku „top" změn se řadí také plná podpora multitaskingu (paralelního zpracování většího množství úloh) a multithreadingu (podpora exekuce vícevláknových aplikací). Jak naprogramovat vícevláknovou aplikaci se dozvíte právě v tomto vydání sekce Téma měsíce.

Obsah

Teoretický úvod do problematiky procesů a vláken

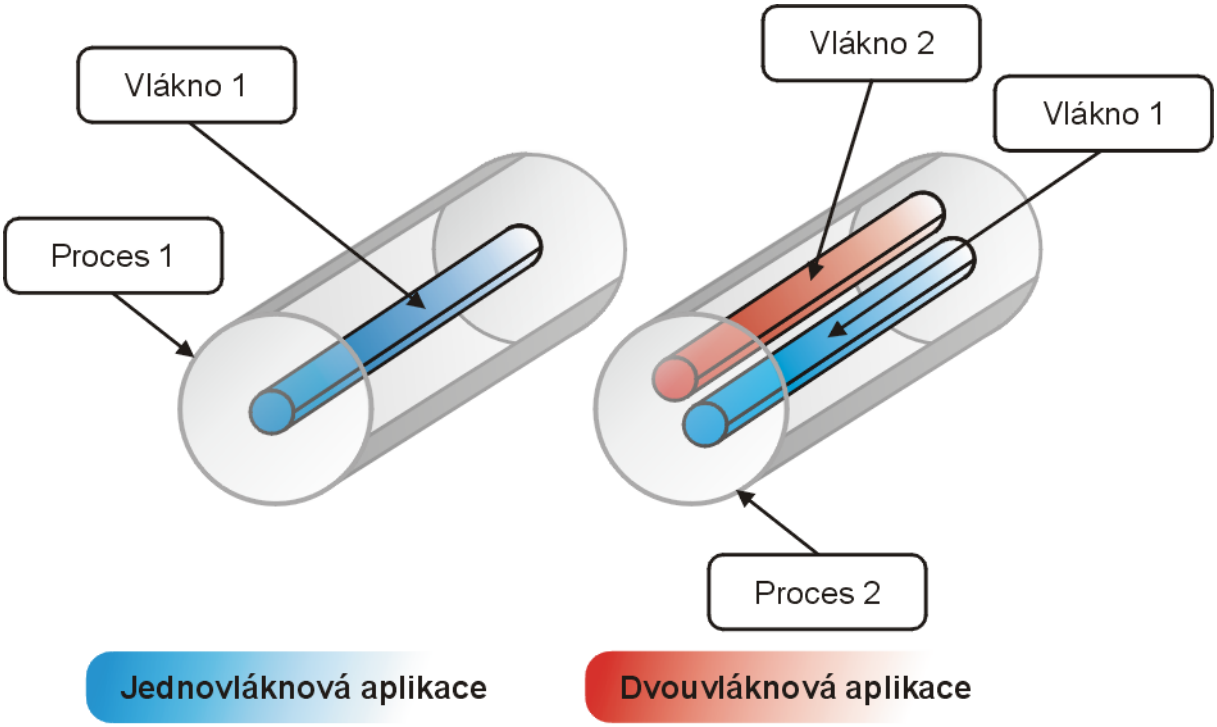
Abychom mohli mluvit o programování vícevláknových aplikací na patřičné úrovni, musíme si nejprve povědět pár slov k „vícevláknové" terminologii. Začneme od definice pojmu aplikace, který je vám pravděpodobně nejbližší. Aplikace je definována jako soubor programového kódu, dat a dalších interních nebo externích zdrojů, které společně vytvářejí jeden celek, jenž řeší jistou smysluplnou úlohu. Pojem aplikace se někdy volně zaměňuje s termínem počítačový program.

I když je ve většině případů tato substituce možná, není zcela přesná. A to z toho důvodu, že programem může být také jenom několik řádků programového kódu, které ač řeší jistý úkol, nemusí být nutně samostatným celkem, jenž je předložen před uživatele. Můžeme však říct, že aplikace je již hotový, odladění počítačový program, jenž se stává předmětem prodeje na softwarovém trhu.

V .NET prostředí je kód aplikace uložen v jedné, nebo i několika assembly. Assembly je pak uložena v přenositelném spustitelném souboru **PE** (**Portable Executable**). Jestliže je soubor aktivován (poklepáním na jeho ikonu), potřebný programový kód v podobě jazyka **MSIL** bude přeložen **JIT** kompilátorem, umístěn do paměti počítače a podroben přímé exekuci. V tomto okamžiku říkáme, že aplikace běží. V programování se běžící aplikace označuje jako **proces**. Pro potřeby procesu je alokován jistý paměťový prostor, ve kterém se daný proces nachází. 32bitový operační systém Windows je, jak již bylo řečeno, plně vybaven pro správu více procesů, resp. úkolů. Jak je to možné? Velmi zjednodušeně můžeme prohlásit, že každému procesu operační systém přičlení stanovený časový interval, v rámci kterého je procesor (CPU) počítače plně zaměstnán prováděním kódu daného procesu. Po uplynutí vymezeného intervalu se operační systém „přepne" do jiného

procesu a jistou dobu vykonává kód tohoto procesu. Tímto způsobem operační systém spravuje všechny procesy, neboli spuštěné aplikace. Je důležité si uvědomit, že práce s procesy je kompletně v rukou operačního systému, nikoliv jiných procesů.

Každý proces je tvořen přinejmenším jedním vláknem (**thread**). Vlákno lze definovat jako samostatnou jednotku exekuce programového kódu procesu. Každé vlákno má své identifikační číslo (ID), jméno a prioritu. Zatímco jméno vlákna tvoří textový řetězec, jenž slouží na uživatelskou identifikaci vlákna, identifikační číslo vlákno přesně charakterizuje z pohledu operačního systému. Priorita vlákna určuje, jak intenzivně je zapotřebí programový kód daného vlákna provádět. Zvyčejně mají vlákna normální prioritu, avšak v případě potřeby je možné zvýšit, nebo naopak snížit prioritu vlákna na příslušnou úroveň. Jestliže proces obsahuje jenom jedno vlákno, mluvíme o jednovláknové aplikaci (**single threaded application**). Pokud jste vyvíjeli aplikace v předchozí verzi Visual Basicu, nejspíš jste pracovali pouze s jednovláknovými aplikacemi. Visual Basic .NET ovšem nabízí programování také vícevláknových aplikací (**multithreaded applications**). Vícevláknová aplikace je aplikace, která používá dvě a více vláken. Komparaci jedno a vícevláknové aplikace můžete vidět na obr. 1.



Obr. 1 – Jedno a vícevláknová aplikace

Vlákno, které je na obrázku pojmenováno jako „Vlákno 1" je základním vláknem procesu, což znamená, že toto vlákno je vytvořeno implicitně běhovým prostředím aplikace. Naproti tomu vlákno s názvem „Vlákno 2" je pracovním vláknem, které vytvořil programátor pro své potřeby. Ačkoliv obrázek představuje jenom dvouvláknovou aplikaci, proces může obsahovat i větší množství vláken. Příliš mnoho vláken ovšem není žádoucí, protože by snadno mohly nastat potíže se synchronizací jednotlivých vláken (pomineme-li poněkud větší paměťovou náročnost aplikace).

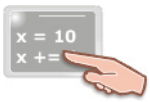
Každému vláknu v procesu je přiřazen jistý časový interval, během kterého CPU počítače realizuje programové instrukce daného vlákna. Jakmile tento interval uplyne, CPU okamžitě začne zpracovávat kód dalšího vlákna. Jelikož je časový interval realizace kódu vlákna velmi malý, vizuálně to vypadá, jakoby obě vlákna pracovala současně.

Vytváříme první vícevláknovou aplikaci

V následující programové ukázce si předvedeme, jak sestrojit jednoduchou dvouvláknovou aplikaci. Postupujte podle níže uvedených instrukcí:

1. Spusťte Visual Basic .NET a vytvořte standardní aplikace pro Windows (**Windows Application**).

2. Na formulář přidejte jednu instanci ovládacího prvku **Button**, kterou pojmenujte jako **btnVytvořit_vlákno** (libovolně můžete také modifikovat vlastnost **Text** vytvořené instance). Na vytvořenou instanci poklepejte, čímž nařídíte Visual Basicu, aby vygeneroval kostru událostní procedury **Click** instance. Do obslužné procedury zadejte tento programový kód:

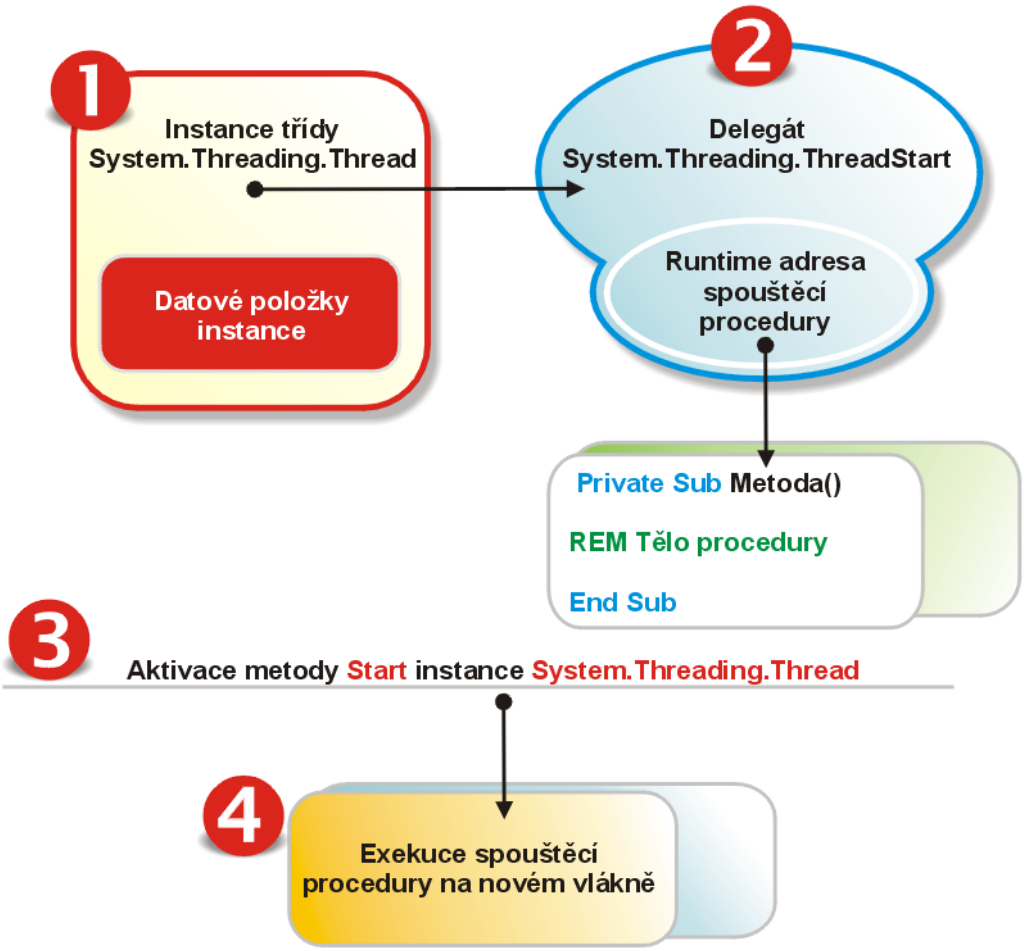



```
Dim NovéVlákno As New Threading.Thread(AddressOf Metoda)
NovéVlákno.Name = "MojeVlákno"
NovéVlákno.Start()
```

Pokaždé, když budeme chtít vytvořit nové vlákno, použijeme třídu **Thread** z jmenného prostoru **System.Threading**. Z tohoto pohledu je tedy zřejmé, že k vytvořenému vláknu budeme ve skutečnosti přistupovat prostřednictvím instance třídy **Thread**. Instance této třídy se ovšem rodí poněkud zvláštním způsobem. Podívejme se na něj blíže. Především je zapotřebí obsloužit parametrický konstruktor třídy **Thread**, kterému je nutno předat (pomocí operátoru **AddressOf**) runtime adresu procedury, která bude prováděna na nově vytvořeném vlákně. Ve skutečnosti je však celý proces poněkud komplikovanější: Ačkoliv je tato skutečnost zastíněna, je vytvořen delegát **ThreadStart**, jemuž je svěřena runtime adresa spouštěcí procedury. I když vytvoříme nové vlákno (resp. instanci třídy **Thread**), neznamená to ještě, že jsme také toto vlákno spustili.

V této souvislosti je nutno podotknout, že při práci s vlákny je zapotřebí důsledně rozlišovat dva pojmy, a sice vytvoření vlákna a jeho spuštění. Vlákno je vytvořeno jakmile je zrozena instance třídy **Thread**. Takto vytvořené vlákno má příznak **Unstarted**, což znamená, že ještě nebylo spuštěno. Aby mohlo být vlákno spuštěno, musí být zavolána metoda **Start** instance třídy **Thread**.

Spuštění nového vlákna zabezpečíme zavoláním metody **Start** instance třídy **Thread**. Jakmile je aktivována metoda **Start**, je zavolána spouštěcí procedura, která bude prováděna na novém vlákně. Proces tvorby a spuštění nového vlákna je znázorněn na obr. 2.



Obr. 2 – Proces vytvoření a spuštění vlákna

Pro zvídavé programátory: Vytvoření instance třídy **Thread** pod drobnohledem

Ještě jednou se podívejme na výše uvedený zdrojový kód pro vytvoření instance třídy **Thread**:

```
Dim NovéVlákno As New Threading.Thread(AddressOf Metoda)
NovéVlákno.Name = "MojeVlákno"
NovéVlákno.Start()
```

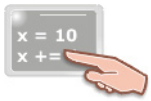
I když je tento kód zcela správný, není v něm vidět role delegáta **ThreadStart**. Obměníme-li kód následovně, vyjdou skryté skutečnosti na povrch:

```
Dim NovéVlákno As New Threading.Thread _
(New Threading.ThreadStart(AddressOf Metoda))
NovéVlákno.Name = "MojeVlákno"
NovéVlákno.Start()
```

Tento kód jasně demonstruje způsob, pomocí něhož je instanci třídy **Thread** předán delegát **ThreadState**, kterému je prostřednictvím operátoru **AddressOf** poskytnuta runtime adresa spouštěcí procedury (s názvem **Metoda**), která bude realizována na nově vytvořeném vlákně. Ačkoliv je tento zápis kódu pro vytvoření vlákna znatelně delší, umožňuje nám pohled do zákulisí procesu vytváření vlákna. V mnoha programech jej však zcela jistě neuvidíte, protože Visual Basic přidává uvedené rozšíření kódu implicitně, a tedy není nutné, abyste tuto „prodlouženou" verzi kódu používali ve svých aplikacích.

V našem případě je vytvořena instance třídy **Thread** s názvem **NovéVlákno**. Konstruktoru třídy je předána adresa procedury **Metoda**, tato procedura se tedy bude provádět na druhém (vytvořeném) vlákně. Ještě předtím, než je vlákno spuštěno, je upravena vlastnost **Name** instance **MojeVlákno** – nové vlákno tak bude disponovat uživatelsky přívětivým jménem.

3. Jestliže jste událostní proceduru **Click** instance tlačítka vyplnili uvedeným zdrojovým kódem, obdrželi jste zcela jistě připomínku, že procedura **Metoda** není deklarována. Proto je nutné kód této procedury zahrnout do kódu třídy formuláře **Form1**. Protože ukázková procedura **Metoda** využívá referenční proměnné a konstanty s oborem třídy, je uveden výpis kompletního programového kódu, jenž se nachází v souboru **Form1.vb**:




```
Option Strict On
Public Class Form1
    Inherits System.Windows.Forms.Form

    Dim frm As Form
    Dim lblText1 As Label
    Dim lbltext2 As Label
    Dim Progres As ProgressBar

    Const PROGRES_MIN As Integer = 1
    Const PROGRES_MAX As Integer = 1000

    Private Sub Button1_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Button1.Click
        Dim NovéVlákno As New Threading.Thread(AddressOf Metoda)
        NovéVlákno.Name = "MojeVlákno"
        NovéVlákno.Start()
    End Sub

    Private Sub Metoda()
        frm = New Form()
        lblText1 = New Label()
        lbltext2 = New Label()
        Progres = New ProgressBar()
        With frm
            .ControlBox = False
            .FormBorderStyle = FormBorderStyle.FixedDialog
            .Text = "Probíhá výpočet náhodných čísel..."
        End With

        With lblText1
            .Location = New Point(0, 0)
            .Size = New Size(frm.Width, frm.Height \ 4)
            .Text = "Právě probíhá generování náhodných čísel"
            .Font = New Font("Verdana", 18, FontStyle.Bold, _
            GraphicsUnit.Pixel)
            .TextAlign = ContentAlignment.MiddleCenter
        End With

        With lbltext2
            .Location = New Point(0, 75)
            .Size = New Size(frm.Width, frm.Height \ 6)
            lbltext2.Text = "Číselný průběh generování čísel:"
            .Font = New Font("Verdana", 11, FontStyle.Bold, _
            GraphicsUnit.Pixel)
            .TextAlign = ContentAlignment.MiddleCenter
        End With
```

Programový kód pokračuje na následující straně

```
            With Progres
                .Dock = DockStyle.Bottom
                .Height = frm.Height \ 6
                .Minimum = PROGRES_MIN
                .Maximum = PROGRES_MAX
            End With

            With frm.Controls
                .Add(Progres)
                .Add(lblText1)
                .Add(lbltext2)
            End With


            AddHandler frm.Load, AddressOf frm_Load
            frm.Show()

            Dim f As Integer = FreeFile()
            Randomize()
            Dim a, x As Integer
            FileOpen(f, "d:\data.txt", OpenMode.Output)
            For a = PROGRES_MIN To PROGRES_MAX
                x = CInt(Int(1000 * Rnd()) + 1))
                Progres.Value = a
                lbltext2.Text = a & " / " & PROGRES_MAX
                If a < PROGRES_MAX Then
                    Print(f, x & ", ")
                Else
                    Print(f, x & ".")
                End If
                Application.DoEvents()
            Next a
            FileClose(f)
            frm.Close()

        End Sub

        Private Sub frm_Load(ByVal sender As Object, ByVal e As EventArgs)
            frm.Top = (Screen.PrimaryScreen.Bounds.Height - frm.Height) \ 2
            frm.Left = (Screen.PrimaryScreen.Bounds.Width - frm.Width) \ 2
        End Sub
    End Class
```

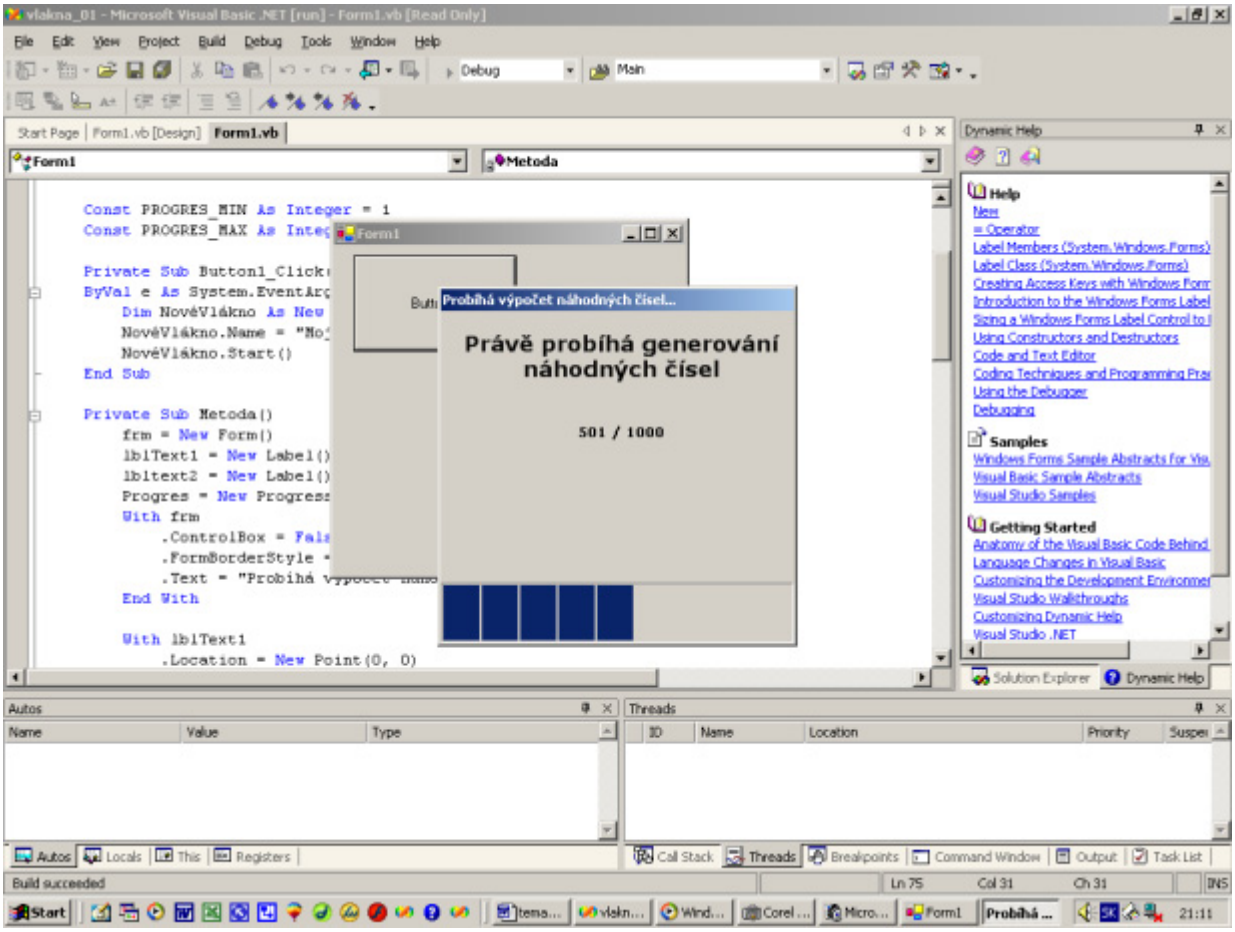
Na vytvořeném vlákně bude prováděn kód procedury **Metoda**, který realizuje následující činnosti:

- Dochází k vytvoření instance třídy **Form**. Na plochu instance třídy **Form** jsou přidány instance tříd **Label** a **ProgressBar**, přičemž je vždy upravena pozice vytvořených instancí a modifikovány jsou také další požadované vlastnosti.
- Aby mohla být instance třídy **Form** vycentrována za běhu programu, je aplikován příkaz **AddHandler** s použitím operátoru **AddressOf**. Příkaz **AddHandler** vytváří zpracovatele jisté události (v našem případě události **Load** instance třídy **Form**) a operátor **AddressOf** ukazuje na paměťové místo se vstupním bodem obslužní procedury (**frm_Load**) pro událost **Load**.
- V těle procedury je vygenerováno 1000 náhodných čísel z intervalu <1, 1000>. Inicializaci generátoru náhodných čísel na základě systémového času zabezpečí příkaz **Randomize**. Všechna získaná čísla jsou následně zapsána do textového souboru s názvem data.txt. Tento soubor je vytvořen na disku d, ovšem v případě, že se na vašem počítači nenachází logický nebo fyzický disk s uvedeným jménem, budete muset název disku změnit podle vašich pracovních podmínek. Mezi všechna vygenerovaná čísla jsou vloženy čárky a za posledním číslem je umístěna tečka.
- Průběh generování náhodných čísel a jejich zápis do souboru demonstruje ukazatel průběhu (neboli instance třídy **ProgressBar**).

- Metoda **DoEvents** třídy **Application** umožní řádné překreslení instance třídy **Form** a také všech dalších instancí, které byly na plochu formuláře přidány.
- Zcela originálně je vyřešeno vycentrování instance třídy **Form** v horizontálním i vertikálním směru. Předem je totiž připravena událostní procedura **frm_Load**, programový kód které bude proveden po zavolání metody **Show** instance třídy **Form** (s názvem **frm**). Protože je nutné instanci třídy **Form** umístit do středu obrazovky, použijeme vhodnou modifikaci vlastností **Top** a **Left**. Událostní procedura však bude realizována, jenom pokud vytvoříme systémového delegáta, kterému pomocí operátoru **AddressOf** předáme adresu této procedury.

Testujeme první vícevláknovou aplikaci

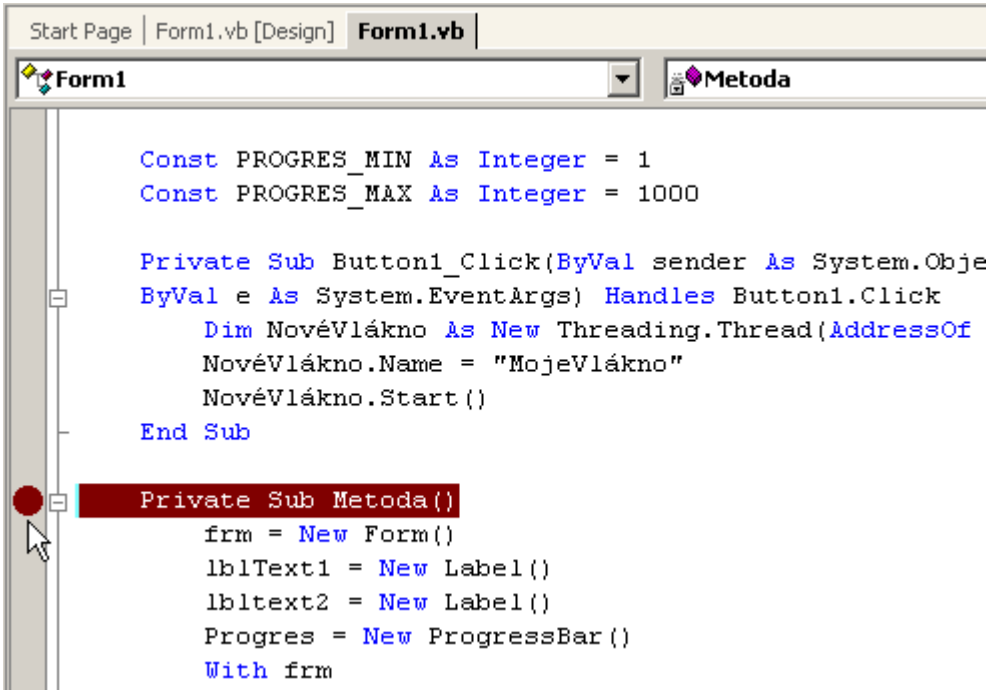
Jste-li s programováním hotovi, můžete aplikaci sestavit (**Build ➔ Build Solution**). Jestliže aplikaci spustíte, uvidíte, že svoji práce odvádí skvěle (obr. 2).



Obr. 2 – Vícevláknová aplikace v akci

Jak se ovšem dovíme, že generování náhodných čísel, jako i další kód, jenž je uložen v proceduře **Metoda** skutečně běží na nově vytvořeném vlákně? Inu, asi takto:

1. Pokud se váš projekt ještě nachází v režimu běhu, tak uzavřete dialogové okno aplikace, čímž se dostanete do režimu návrhu.
2. V programovém kódu třídy **Form1** vyhledejte začátek procedury **Metoda**.
3. Abychom viděli, zdali je již vytvořené nové vlákno, umístíme na začátek procedury **Metoda** programovou zarážku (breakpoint). To uděláte tak, že nalevo od názvu procedury klepnete na plochu šedého panelu (obr. 3).



Obr. 3 – Vložení zarážky do programového kódu

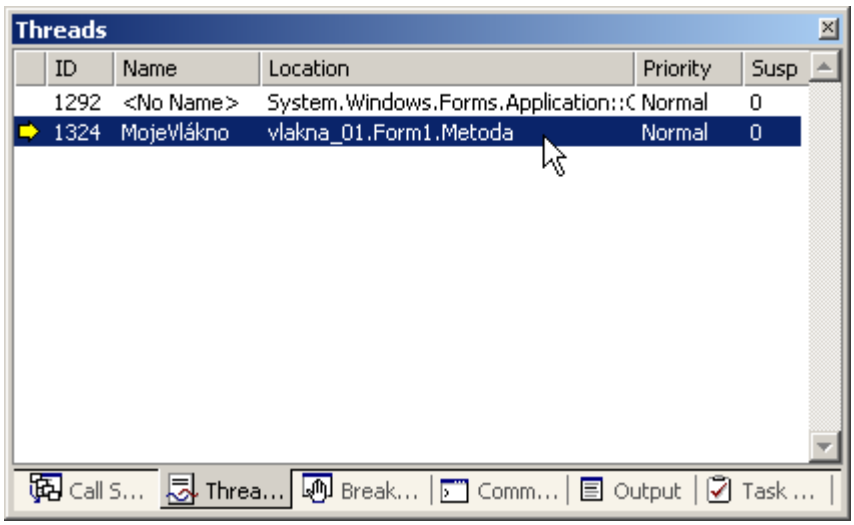
O správnosti umístění zarážky vás ujistí červený puntík, jenž se objeví v levém panelu. Kromě červeného puntíku si můžete také všimnout červený pruh, kterým je zvýrazněn řádek programového kódu, na němž je zarážka umístěná.

Jestliže na nějaký řádek programového kódu umístíme zarážku, dáme tím debuggeru Visual Basicu příkaz, aby při přechodu na řádek se zarážkou zastavil běh aplikace a zpřístupnil dodatečné možnosti pro odladění kódu aplikace.

Zařazení programové zarážky je výhodné především proto, že jakmile debugger Visual Basicu „skočí" na řádek se zarážkou, bude běh aplikace pozastaven a my budeme moci prozkoumat vnitřní programové jádro aplikace. Pozastavenou aplikaci lze samozřejmě dále rozběhnout, nebo ji můžete ukončit.

4. Spusťte aplikaci a klepněte na tlačítko pro vytvoření nového vlákna. Běh aplikace se zastaví a vy budete přeneseni do režimu odladění aplikace.
5. Seznam vláken je uveden v dialogu **Threads**. Pokud tento dialog nevidíte, vyberte nabídku **Debug**, ukažte na subnabídku **Windows** a klepněte na položku **Threads**. Zobrazí se dialog **Threads**, v němž jsou uvedena všechna aktivovaná vlákna (obr. 4).

Dialogové okno **Threads** můžete zobrazit také pomocí klávesové zkratky CTRL+ALT+H.



Obr. 4 – Dialogové okno **Threads**

6. Všimněte si, že v okamžiku přerušení běhu aplikace již bylo vytvořeno nové vlákno. Toto vlákno má název **MojeVlákno**, identifikační číslo 1324 a normální prioritu.
7. Pokud chcete, aby aplikace pokračovala v běhu, vyberte nabídku **Debug** a klepněte na položku **Continue**. V opačném případě můžete zvolit příkaz **Stop Debugging** (rovněž z nabídky **Debug**).
8. Vytvořenou programovou zarážku odstraníte klepnutím na červení puntík v levém panelu.

Operace s vlákny

Jakmile je vlákno vytvořeno, můžete s ním provádět několik operací. Rámec potenciálních akcí pro práci s vlákny zastřešuje třída **Thread** se svými členskými metodami a vlastnostmi. V následující tabulce se nachází přehled některých klíčových metod a vlastností pro práci s vlákny, o nichž si myslím, že by mohly být pro vás užitečné.

Název	Typ členu	Charakteristika
Abort	Metoda	Metoda se používá pro zrušení vlákna. Po zavolání metody je ve vybraném vláknu generována výjimka **ThreadAbortException**, na což je spuštěn proces rušení vlákna.
CurrentThread	Vlastnost	Jde o sdílenou (shared) vlastnost, což znamená, že jde o vlastnost samotné třídy **Thread** a nikoliv instance této třídy. Použijete-li tuto vlastnost, získáte přístup k vláknu, jehož programový kód je právě prováděn.
Name	Vlastnost	Pokud chcete vlákno pojmenovat, případně chcete-li jméno vlákna získat, můžete použít tuto vlastnost. Hodnotou vlastnosti je textový řetězec, jenž představuje uživatelské jméno pro požadované vlákno.
Priority	Vlastnost	Vlastnost určuje prioritu vlákna. Každé vlákno se může nacházet v několika prioritních stavech: • **Highest** (Nejvyšší priorita) • **AboveNormal** (Vyšší nežli normální priorita) • **Normal** (Standardní priorita) • **BelowNormal** (Nižší nežli normální priorita) • **Lowest** (Nejnižší priorita) Hodnotou vlastnosti je jeden z členů enumerace **ThreadPriority** (**AboveNormal**, **BelowNormal**, **Highest**, **Lowest**, **Normal**).
ResetAbort	Metoda	Jde o sdílenou (shared) metodu, která likviduje požadavek na zrušení (**Abort**) vlákna.

Název	Typ členu	Charakteristika
Resume	Metoda	Metodu lze aplikovat na vlákno, které bylo uvedeno do stavu nečinnosti použitím metody **Suspend**. Metoda zabezpečí probuzení vlákna a exekuci jeho programového kódu.
Sleep	Metoda	Jde o sdílenou (shared) a také přetíženou metodu, která se vyskytuje ve dvou exemplářích. Obě přetížené varianty metody zamezí provádění programového kódu aktivního vlákna na určitý časový interval (tento je zvyčejně měřen v milisekundách).
Start	Metoda	Zavolání metody způsobí rozběhnutí programového kódu na daném vlákně. Vláknu, resp. instanci třídy **Thread**, je po aktivaci metody **Start** přiřazen stav **ThreadState.Running**, což znamená, že vlákno je aktivní (je realizován kód vlákna). Jestliže bylo vlákno jednou ukončeno, nelze jej opět aktivovat prostřednictvím opětovného volání metody **Start**.
Suspend	Metoda	Metoda **Suspend** je podobně jako metoda **Sleep** používaná na „uvedení vlákna do stavu spánku". Mezi těmito dvěma metodami ovšem existuje několik podstatných rozdílů. Především, metoda **Sleep** je sdílená (shared), což znamená, že ji možno aplikovat jenom na vlákno, jehož programový kód se právě uskutečňuje. Metoda **Suspend** je naproti tomu metodou instance a nikoliv třídy, z čehož vyplývá, že tuto metodu můžete použít na jakoukoliv instanci třídy **Thread**. Druhým odlišným znakem je skutečnost, že jestliže je použita metoda **Sleep**, je předem definovaný časový interval (v milisekundách), v rámci něhož bude vlákno spát. Na druhé straně, bude-li zavolána metoda **Suspend**, nelze předem určit, kdy má dojít k uspání vlákna. Ve skutečnosti dochází k uspání vlákna v okamžiku, kdy je dosažen tzv. bezpečný bod. Určení existence bezpečného bodu má na starosti Common Language Runtime (CLR) pomocí služby Garbage Collection. A konečně, poslední odlišnost spočívá v opětovné aktivaci deaktivovaného vlákna. Bude-li vlákno uspáno pomocí metody **Suspend**, lze jej ihned probudit zavoláním metody **Resume**. Uspíte-li však vlákno metodou **Sleep**, neexistuje žádný způsob, jak jej probudit předtím, nežli vyprší stanovená doba vlákna pro nečinnost.
ThreadState	Vlastnost	Každé vlákno se může nacházet v určitém počtu stavů. Skutečnosti o stavu vlákna zprostředkovává právě vlastnost **ThreadState**. Jde o vlastnost pouze pro čtení, což znamená, že hodnotu vlastnosti lze přečíst, ovšem není možné ji jakkoliv modifikovat (toto chování je vzhledem k charakteru vláken snadno pochopitelné). Hodnotou vlastnosti je jeden z členů enumerace **ThreadState**.

Tab. 1 – Charakteristika užitečných vlastností a metod třídy **Thread**

Integrované prostředí rubriky Visual Basic	Autor: **Ján Hanák**

Začínáme s VB .NET

Úvod do světa .NET (11. díl)

Časová náročnost (min):	Začátečník	Pokročilý	Profesionál
45			

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

Vážení čtenáři,

v jedenácté lekci budeme pokračovat v problematice použití operátorů ve Visual Basicu .NET. Nejprve si představíme klasifikaci operátorů v přehledné formě, abyste získali základní přehled o všech kategoriích operátorů, s nimiž se můžete v nové verzi programovacího jazyka setkat. Poté přejdeme na bližší výklad aritmetických a porovnávacích operátorů. Přeji vám příjemné počtení.

Obsah

Klasifikace operátorů

Visual Basic .NET obsahuje značné množství operátorů, které můžete použít při různých příležitostech. Abyste měli ihned ze začátku základní přehled o operátorech, rozdělíme si je na několik homogenních skupin (tab. 1).

Kategorie operátorů	Charakteristika
Aritmetické operátory	Mezi aritmetické operátory patří tyto: • operátor pro sčítaní (**+**) • operátor pro odčítaní (**-**) • operátor pro násobení (*****) • operátor pro dělení (**/**) • operátor pro celočíselné dělení (****) • operátor zbytku po dělení (**Mod**) • operátor pro umocňování (**^**) Aritmetické operátory se používají při běžných matematických operacích s celými čísly, nebo s čísly s desetinnou částí.

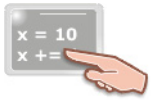
Porovnávací operátory	Porovnávací operátory se využívají při porovnávání hodnot dvou, nebo i více výrazů. Mezi porovnávací operátory zařazujeme tyto: • operátor rovnosti (**=**) • operátor nerovnosti (**<>**) • operátor menší než (**<**) • operátor větší než (**>**) • operátor menší nebo rovno (**<=**) • operátor větší nebo rovno (**>=**) • operátor **Is** • operátor **Like**
Přiřazovací operátory	Skupina přiřazovacích operátorů se ve Visual Basicu .NET značně rozrostla v důsledku zavedení tzv. zkrácených tvarů operátorů. Do této kategorie patří následující operátory: • operátor přiřazení (**=**) • zkrácený operátor pro sčítaní a přiřazení (**+=**) • zkrácený operátor pro odečítaní a přiřazení (**-=**) • zkrácený operátor pro násobení a přiřazení (***=**) • zkrácený operátor pro dělení a přiřazení (**/=**) • zkrácený operátor pro celočíselné dělení a přiřazení (**\=**) • zkrácený operátor pro umocňování a přiřazení (**^=**) • zkrácený operátor pro zřetězení a přiřazení (**&=**)
Logické operátory	Logické operátory se vkládají do logických výrazů, přičemž pomocí operátorů lze získat pravdivostní hodnotu daného logického výrazu. Tato pravdivostní hodnota může posléze posloužit pro řízení jiných programových konstrukcí (např. rozhodovací konstrukce **If** či cyklů). Mezi logické operátory patří tyto: • operátor **And** • operátor **AndAlso** • operátor **Or** • operátor **OrElse** • operátor **Not** • operátor **Xor**
Operátory pro zřetězení	Operátory této kategorie se uplatňují při zřetězení dvou, nebo i několika operandů (tyto operandy mohou být řetězcového typu (**String**), nebo i jiného typu). Mezi operátory pro zřetězení patří: • operátor pro zřetězení (**&**) • operátor pro zřetězení (**+**)
Bitové operátory	Bitové operátory se používají při provádění operací nad jednotlivými bity hodnot celočíselných datových typů. Do skupiny bitových operátorů patří následující zástupci: • bitový operátor **And** • bitový operátor **Or** • bitový operátor **Not** • bitový operátor **Xor** Jak si můžete všimnout, bitové operátory mají stejné názvy jako některé logické operátory. Ve skutečnosti ovšem existuje pouze jedna verze těchto operátorů, ovšem způsob práce operátorů je determinován na základě typu dodaných operandů. Operátorům, kterých činnost závisí od určitého kontextu (v tomto případě od typu operandů) se říká přetížené operátory. Přetížené operátory jsou užitečnou programovací technikou, protože operátory „vědí“, co mají provést se svými operandy.

Speciální operátory	Visual Basic .NET má v rukávu rovněž dvě esa, kterými jsou dva speciální operátory: • operátor **AddressOf** • operátor **GetType** Operátor **AddressOf** umožňuje přístup k runtime adrese procedury nebo funkce. Operátor **AddressOf** se používá např. při tvorbě delegátů, nebo při programování vícevláknových aplikací. Operátor **GetType** vrací typ vstupního objektu.

Tab. 1 – Klasifikace operátorů

Aritmetické operátory

Aritmetické operátory můžete použít, kdykoliv budete potřebovat realizovat jistou aritmetickou operaci. Pravděpodobně nejčastěji se budete setkávat se základními aritmetickými operátory, mezi které patří operátory pro sčítaní, odčítání, násobení a dělení. Všechny tyto operátory pracují tak, jak byste čekali: Vezmou hodnoty svých operandů a provedou příslušnou matematickou operaci. Zde můžete vidět několik příkladů:

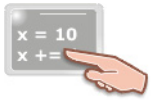



```
Dim a, b, c, d As Integer
a = 11 + 88
b = 111 - 55
c = b * 10
d = CInt(c / 3)
Me.Text = "a=" & a & " b=" & b & " c=" & c & " d=" & d
```

V uvedeném fragmentu zdrojového kódu se střetáváme s proměnnými **a**, **b**, **c** a **d** datového typu **Integer**. V dalších krocích jsou proměnné inicializované vypočtenými hodnotami. Tyto hodnoty představují výsledek práce aritmetických operátorů. Kdybyste tento kód umístili do událostní procedury **Click** tlačítka a spustili byste projekt, v titulkovém pruhu aktuální instance formuláře by se objevili pozměněné hodnoty použitých proměnných: **a**=99, **b**=56, **c**=560 a **d**=187. I když jsou výsledné hodnoty proměnných **a**, **b** a **c** vskutku pochopitelné, jisté pochybnosti byste mohli mít u proměnné **d**. Hodnota proměnné **d** je totiž 187, a to i přesto, že podíl čísel 560 a 3 je 186,6667. Jak je to možné? Tato skutečnost je způsobena použitím konverzní funkce **CInt**, která převede návratovou hodnotu operátoru pro dělení do podoby celého čísla. Připomínám, že konverzní funkci **CInt** je nutné použít v případě, kdy pracujete s aktivovanou volbou **Option Strict On**. Když operátor **/** dokončí svou práci, vrací zvyčejně výslední hodnotu v podobě desetinného čísla s dvojitou přesností (datový typ **Double**). Návratovou hodnotu operátoru převezme konverzní funkce **CInt**, která přetypuje předanou hodnotu na hodnotu datového typu **Integer**. Jelikož hodnoty typu **Integer** nemohou uchovávat desetinnou část čísla, je číslo zaokrouhleno. Tedy z přesné hodnoty 186,6667 se stane hodnota 187.

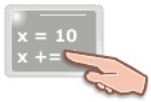
Pokud při programování nepoužíváte volbu **Option Strict On**, nemusíte konverzní funkci **CInt** volat explicitně. Visual Basic převede přetypování implicitně, ovšem výsledná hodnota uložená v proměnné **d** bude stejná (187).

Budete-li chtít získat přesnou hodnotu podílu, můžete použít tento programový kód:



```
Dim a, b, c As Integer
Dim d As Double
a = 11 + 88
b = 111 - 55
c = b * 10
d = c / 3
Me.Text = "a=" & a & " b=" & b & " c=" & c & " d=" & d
```

Zcela jistě zajímavé bude použití operátoru pro celočíselné dělení (****):



```
Dim x As Short = 100
Dim y As Short = 26
Me.Text = "Výsledek celočíselného dělení je " & x \ y
```

Operátor pro celočíselné dělení (****) vydělí hodnotu levého operandu hodnotou pravého operandu a vrátí podíl v podobě celočíselní hodnoty (v našem případě jde o hodnotu 3). Jelikož po „zavolání" operátoru obdržíme celočíselnou hodnotu, není zapotřebí použít žádnou konverzní funkci. Kromě toho, celočíselné dělení je výpočetně méně náročné jako přesné dělení, při kterém je výsledkem číslo s pohyblivou desetinnou čárkou.

Správné použití operátoru pro celočíselné dělení může v jistých situacích znatelně ovlivnit rychlost vykonávání programového kódu. Nepotřebujete-li přílišnou přesnost, dalo by se říci, že operátor **** je pro vás „to pravé".

Mnozí programátoři operátor pro celočíselné dělení neznají. Když jej proto použijete ve vhodné chvíli, můžete ukázat, jak znalí programátoři jste. Jestliže se budete ucházet o místo programátora ve Visual Basicu a ukážete znalost operátoru pro celočíselné dělení, vaše hodnota v očích vašeho budoucího zaměstnavatele zcela jistě stoupne. Z tohoto pohledu lze použití operátoru **** přirovnat pomyslné třešničce na dortu.

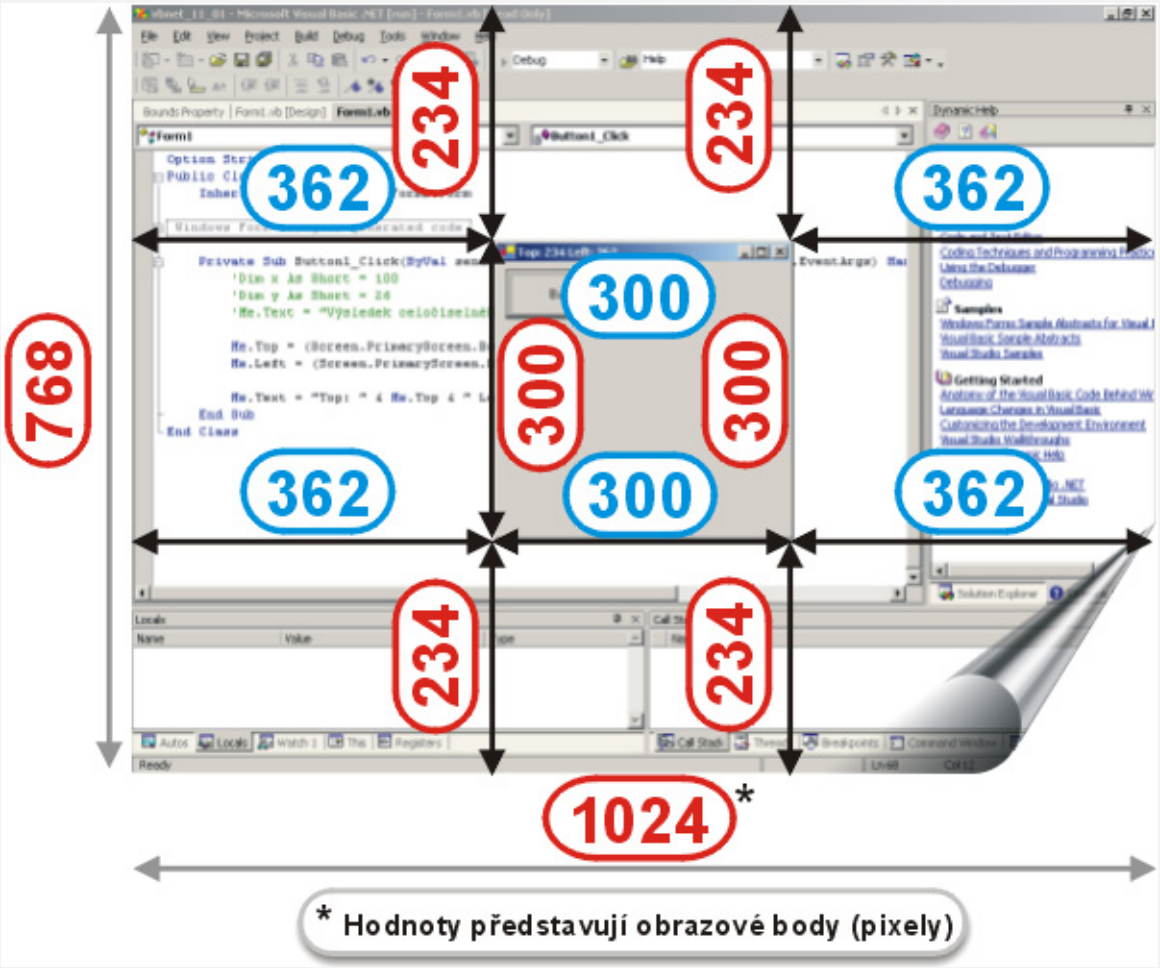
Pro zvídavé programátory: Použití operátoru \ při centrování formuláře

Formulář (instance třídy **Form1**) lze umístit do středu obrazovky pomocí následujících řádků zdrojového kódu:

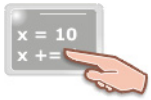
```
Me.Top = (Screen.PrimaryScreen.Bounds.Height - Me.Height) \ 2
Me.Left = (Screen.PrimaryScreen.Bounds.Width - Me.Width) \ 2
```

Při centrování formuláře použijeme třídu **Screen**. Třída **Screen** však nedisponuje veřejným konstruktorem, a proto nelze přímo vytvořit její instanci. Instance třídy **Screen** se vytvoří v okamžiku, kdy zavoláme veřejné metody nebo vlastnosti této třídy. V našem případě voláme vlastnost **PrimaryScreen**, která vrátí instanci třídy **Screen**. Vlastnost **PrimaryScreen** je sdílená (**shared**) a určena jenom pro čtení (**readonly**). Když zavoláme vlastnost **PrimaryScreen**, zjistíme primární zobrazovací jednotku (pokud počítač disponuje pouze jednou zobrazovací jednotkou, je vrácena tato, v opačném případě je vrácena ta zobrazovací jednotka, která je nakonfigurována jako hlavní). Dále voláme vlastnost **Bounds**, která vrátí instanci třídy **Rectangle** (také vlastnost **Bounds** je určena jenom pro čtení). Tato instance představuje obdélníkový region, jenž reprezentuje viditelnou plochu primární zobrazovací jednotky. Výšku, resp. šířku obdélníkového regionu získáme prostřednictvím vlastností **Height** a **Width**. Aby byl formulář vystředěn, je nutné od výšky, resp. šířky obdélníkového regionu odečíst výšku, resp. šířku formuláře a následně získané hodnoty vydělit dvěmi pomocí operátoru pro celočíselné dělení (****).

Předpokládejme, že používáte 17palcový monitor s rozlišením 1024x768 obrazových bodů. Spustíte-li uvedený kód, bude vlastnost **Top** formuláře rovna hodnotě 234 a vlastnost **Left** zase hodnotě 362. Při tomto výpočtu vycházíme ze skutečnosti, že rozměry formuláře mají implicitní velikost 300x300 obrazových bodů (viz obrázek).



Budete-li chtít získat zbytek po dělení, použijte operátor **Mod**. Kupříkladu následující kód vytiskne do titulku okna hlášení „Zbytek po dělení je 3“:

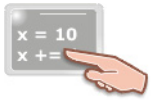



```
Dim m, n As Byte
m = 23
n = 4
Me.Text = "Zbytek po dělení je " & m Mod n
```

Podívejme se, proč je tomu tak. Algoritmus práce operátoru **Mod** probíhá přibližně v těchto krocích:

1. Operand **m** je vydělen operandem **n**, tedy **23 / 4 = 5,75**.
2. Desetinná část podílu je ořezána, získáváme tedy celé číslo **5**.
3. Modifikovaná hodnota podílu je vynásobená operandem **n**, tedy **5 * 4 = 20**.
4. Od operandu **m** je odečten vypočtený součin, tedy **23 - 20 = 3**.

Aby byl náš výklad aritmetických operátorů úplný, ukažme si ještě příklad operátoru pro umocňování:



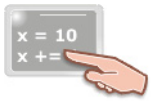
```
Dim k As Integer = 2, l As Integer = 8
Me.Text = CStr(k ^ l)
```

Všeobecní forma použití operátoru pro umocňování je:

Číselná hodnota ^ exponent

V našem případě je vypočtena osmá mocnina čísla 2, která je rovna hodnotě 256. Operátor **^** před samotným výpočtem převede všechny operandy do datového typu **Double** a na takto upravených operandech je posléze provedena operace umocnění. Návratová hodnota operandu je opět datového typu **Double**, takže používáte-li volbu **Option Strict On**, budete muset v případě potřeby uskutečnit explicitní přetypování získané hodnoty.

Exponentem přitom může být jakýkoliv numerický výraz:



```
Dim k As Integer = 2, l As Integer = 8
Me.Text = CStr(k ^ (l * 2 - 6))
```

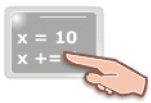
Tento programový kód vypočítá desátou mocninu čísla 2, což je 1024.

Porovnávací operátory

Porovnávací operátory porovnávají výrazy nebo části výrazů a na základě výsledků porovnání vracejí jistou pravdivostní hodnotu (buď hodnotu **True** nebo **False**). Více informací nabízí tab. 2.

Porovnávací operátor	Operandy	Výsledkem je pravdivostní hodnota True	Výsledkem je pravdivostní hodnota False
Operátor rovnosti (=)	a, b	a = b	a <> b
Operátor nerovnosti (<>)	a, b	a <> b	a = b
Operátor menší než (<)	a, b	a < b	a >= b
Operátor větší než (>)	a, b	a > b	a <= b
Operátor menší nebo rovno (<=)	a, b	a <= b	a > b
Operátor větší nebo rovno (>=)	a, b	a >= b	a < b

Ukázku použití porovnávacích operátorů přináší další fragment zdrojového kódu:



```
Dim q, v As Short
Randomize()
q = CShort(Int((10 * Rnd()) + 1))
If q >= 1 And q <= 5 Then
    For v = 1 To q
        Debug.WriteLine("Bylo vygenerováno číslo " & q)
    Next v
End If
```

V tomto kódu jsou deklarované dvě proměnné (**q**, **v**) datového typu **Short**. Příkaz **Randomize** inicializuje generátor náhodných čísel, čili je patrné, že se budeme pokoušet o získání náhodných čísel z jistého číselného intervalu. Tento interval je tvořen čísly 1 až 10, přičemž pro vygenerování náhodných čísel používáme funkci **Rnd**. Funkce **Rnd** je v našem případě vnořena do funkce **Int** (pokud byste se chtěli dozvědět o generování náhodných čísel více, nepřehlédněte ostrůvek pro zvídavé programátory dále v tomto textu). Získané náhodné číslo je přetypováno do datového typu **Short** a přiřazeno do proměnné **q**. Pokud hodnota proměnné **q** z intervalu <1,5> je aktivován cyklus **For-Next**, který do okna **Output** vypíše uvedenou zprávu (zpráva bude vypsaná n-krát, kde n je obdržené náhodné číslo z intervalu <1,5>).

Pro zvídavé programátory: Generování náhodných čísel pod drobnohledem

Pro generování náhodných čísel se zvyčejně používá funkce **Rnd** společně s příkazem **Randomize**. Není-li příkaz **Randomize** následován numerickou hodnotou, která by inicializovala generátor náhodných čísel, je generátor inicializován na základě hodnoty získané prostřednictvím systémového času (přesněji jde o návratovou hodnotu funkce **Timer**). Příkaz **Randomize** nemusí být vůbec aplikován; v tomto případě je funkce **Rnd** inicializována posledně vygenerovaným číslem.

Funkce **Rnd** vrací hodnotu, která může být větší nebo rovna než nula, ovšem je vždy menší než 1. Takto získaná hodnota ovšem není zvyčejně použitelná, a proto je nějak smysluplně upravena. Vygenerovaná hodnota tak může být třeba vynásobena jistým číslem (nejčastěji mocninou čísla 10). Číslo, které bylo vytvořeno pomocí funkce **Rnd**, je uloženo ve tvaru datového typu **Single** (jde tedy o číslo s desetinnou částí). Protože při generování náhodných čísel se ve většině experimentů nepoužívá právě ona desetinná část čísla, bývá tato odstraněna. Existují dva způsoby, pomocí kterých lze „zlikvidovat" desetinnou část náhodného čísla:

1. Zaokrouhlení náhodného čísla
2. Ořezání desetinné části náhodného čísla

K zaokrouhlení náhodného čísla podle známých matematických pravidel dochází při následujících příležitostech:

1. Při přiřazení náhodného čísla do celočíselné proměnné. V tomto okamžiku je implicitně realizována konverze náhodného čísla z datového typu **Single** do cílového celočíselného datového typu.
2. Při explicitně realizované konverzi použitím vhodné konverzní funkce, např. **CInt** nebo **CShort**.

Ořezaní desetinné části náhodného čísla lze uskutečnit zavoláním funkce **Int** nebo funkce **Fix**. Obě funkce odstraňují desetinnou část náhodného čísla, ovšem při práci se zápornými čísly se chovají poněkud odlišně (tato odlišnost nás ale nemusí zajímat, protože my pracujeme pouze s kladnými čísly).

Budete-li chtít získat náhodné celé číslo z jistého intervalu, použijte tuto magickou formulku:

Proměnná = **Int** ((Horní hranice intervalu – Dolní hranice intervalu + 1) * **Rnd()** + Dolní hranice intervalu)

Pro interval <0, 100> bude programový kód vypadat takto:

```
Proměnná = CInt(Int((100 - 1 + 1) * Rnd()) + 1)
```

Aby kód vypadal uhlazeně, upravíme jej takto:

```
Proměnná = CInt(Int(100 * Rnd()) + 1)
```

Algoritmus generování náhodného čísla má přibližně tuto podobu:

1. Funkce **Rnd** vrátí náhodně vybranou numerickou hodnotu typu **Single** z intervalu <0, 1). Povězme, že funkce **Rnd** vybere hodnotu 0.234.
2. Hodnota 0.234 je vynásobena hodnotou 100, čímž získáváme mezihodnotu 23.4.
3. Funkce **Int** odstraní desetinnou část mezihodnoty, na což se z čísla s desetinnou částí stane celé číslo 23.
4. K hodnotě 23 je připočtena hodnota dolní hranice zvoleného intervalu (v našem případě 1). Výsledkem je tedy hodnota 24.

Integrované prostředí rubriky Visual Basic	Autor: Ján Hanák

Téma

VB .NET

Programátorská laboratoř

Prostor pro experimentování

Časová náročnost (min):	Začátečník	Pokročilý	Profesionál
75	1	2	3

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

VB .NET	Tvorba zapečetěné třídy	3	0:35
VB .NET	Přístup k registrům operačního systému Windows	2	0:20
VB .NET	Vytváření vlastního delegáta	3	0:20

Tvorba zapečetěné třídy

V jistých situacích je pro programátora nežádoucí, aby třída, kterou vytvořil svými vlastními silami, byla zdrojem pro další odvozené třídy. Jestliže programátor použije v deklaraci třídy klíčové slovo **NotInheritable**, výslovně nařizuje, že daná třída nemůže sloužit jako bázová třída. Třídě s uvedenými charakteristikami se říká zapečetěná třída. Pokud porovnáme standardní třídu a její zapečetěnou kolegyni, můžeme pozorovat, že primárním rozdílem mezi těmito dvěma třídami je jejich chování při dědění.

Pod pojmem „standardní třída" budeme v následujícím textu rozumět třídu, v deklaraci které se nenachází klíčové slovo **NotInheritable**, jakožto ani jiné modifikátory, které jakýmkoliv způsob ovlivňují chování třídy.

Zatímco od standardní třídy můžeme odvodit tolik podtříd, kolik budeme chtít, zapečetěná třída touto dovedností nedisponuje. Ovšem obě třídy mohou vytvářet své instance, v tomto směru se zapečetěná třída chová úplně stejně jako její standardní protějšek.

I když není tvorba zapečetěné třídy zcela běžnou záležitostí, existuje několik pádných důvodů pro označení třídy klíčovým slovem **NotInheritable**:

- Programátor do zapečetěné třídy implementoval všechnu požadovanou funkcionalitu a již není zapotřebí, aby byla tato třída doplněna o další schopnosti v odvozených třídách.
- Programátor nechce, aby se třída, kterou vytvořil, stala bázovou třídou pro jiné podtřídy.

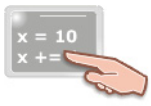
Zapečetěné třídy jsou zpravidla třídy, jejichž stavba plně pokrývá vyřešení jisté problematiky. Jednoduše řečeno, zapečetěná třída obsahuje všechny prvky (vlastnosti, metody a datové členy), které budou instance této třídy využívat. Zapečetěná třída je tak daleko objemnější a robustnější

jako jakákoliv jiná třída. Tato skutečnost vyplývá právě z faktu, že zapečetěná třída si „veze všechno s sebou“. Abyste viděli hlavní rozdíly mezi návrhem a tvorbou standardní a zapečetěné třídy, uveďme si některé podstatné znaky (tab. 1).

Komparační znaky	Typ třídy	
	Standardní třída	**Zapečetěná třída**
Proces návrhu	Při návrhu standardní třídy je ve většině případů brán do úvahy požadavek na pozdější rozšiřitelnost třídy pomocí dědičnosti, a proto se standardní třída navrhuje jako bázová třída. To znamená, že do bázové třídy je zařazena pouze všeobecná, resp. základní funkcionalita, ovšem veškeré konkrétní nebo specifické rysy jsou implementovány až v odvozených třídách.	Proces návrhu zapečetěné třídy musí pozůstávat z pečlivé analýzy všech aspektů funkcionality této třídy. Protože charakter zapečetěné třídy nemůže být později rozšířen v podtřídách, musí zapečetěná třída obsahovat všechny členy, které budou instance této třídy ke své práci potřebovat.
Dědičnost	Odvozené třídy vznikají na základě techniky jednoduché dědičnosti, která je přímo podporována platformou .NET (přesněji společnou jazykovou specifikací – Common Language Specification). Standardní třída není v procesu dědění nijak omezena, a proto je možné vytvářet libovolné množství podtříd.	Zapečetěná třída dědičnost nepodporuje, což znamená, že nemůže sloužit jako bázová třída pro tvorbu odvozených tříd.
Rozšíření	Rozšíření standardní třídy může v zásadě probíhat ve dvou směrech: • rozšíří se přímo programový kód standardní třídy • rozšíření se uskuteční upravením programového kódu odvozených tříd	Rozšíření zapečetěné třídy se musí uskutečnit prostřednictvím začlenění dodatečného programového kódu do samotné zapečetěné třídy.
Vytváření instancí	Plná podpora vytváření instancí třídy.*	Plná podpora vytváření instancí třídy.*
	* Za předpokladu, že programátor nezačlenil do kódu třídy privátní konstruktor.	* Za předpokladu, že programátor nezačlenil do kódu třídy privátní konstruktor.

Tab. 1 – Komparace standardní a zapečetěné třídy

Na následujících řádcích je uveden výpis zdrojového kódu zapečetěné třídy:




```
Option Strict On
Public NotInheritable Class ZapečetěnáTřída
    Private m_Počet As Byte

    Public Property ZjistitPočetFormulářů() As Byte
        Get
            Return m_Počet
        End Get
        Set(ByVal Value As Byte)
            m_Počet = Value
        End Set
    End Property
```

Programový kód pokračuje na následující straně

```
    Public Sub VytvořitFormuláře(ByVal Počet As Byte)
        m_Počet = Počet

        If Počet <= 0 Or Počet > 10 Then
            MessageBox.Show("Bylo zadáno nevhodné číslo (" & Počet _
            & ") pro počet formulářů." & vbCrLf & _
            "Nyní bude zobrazen dialog, v němž budete moci zadat " & _
            "správné číslo pro počet formulářů.", "Zpráva o chybě", _
            MessageBoxButtons.OK, MessageBoxIcon.Error)
            Dim odp As Byte = CByte(InputBox _
            ("Zadejte číslo pro počet formulářů (<1, 10>).", _
            "Určení počtu formulářů", "1"))
            Počet = odp
            GoTo Tvorba_Formulářů
        Else
            Tvorba_Formulářů:
            Dim a As Byte
            Dim frm(Počet - 1) As Form
            For a = 0 To CByte(Počet - 1)
                frm(a) = New Form()
                With frm(a)
                    .Text = "Formulář č. " & a + 1
                    .Show()
                End With
                Application.DoEvents()
            Next
        End If
    End Sub
End Class
```

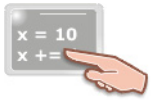
Ukázková zapečetěná třída slouží na vytváření polí instancí třídy **Form**. Jak si můžete všimnout, v těle třídy se nachází veřejná metoda **VytvořitFormuláře**, která odvádí veškerou potřebnou práci. Počet vytvořených formulářů je uložen do soukromého datového členu s názvem **m_Počet**. Hodnotu členu **m_Počet** lze získat zavoláním veřejné vlastnosti **ZjistitPočetFormulářů**. Věřím, že jako profesionálním programátorům je vám kód zapečetěné třídy víceméně jasný, ovšem chtěl bych vás upozornit na tyto skutečnosti:

1. Kód metody **VytvořitFormuláře** testuje hodnotu formálního parametru **Počet** (typu **Byte**). Jestliže je hodnota tohoto parametru jiná než jsou povolené hodnoty z intervalu <1, 10>, zobrazí se varovní hlášení. Hlášení informuje programátora, že byla zadána nepřípustná celočíselná hodnota. Jakmile bude zavřeno okno s varovným hlášením, bude zavolána funkce **InputBox**, která zobrazí dialogové okno s textovým polem pro zadání správné hodnoty pro počet formulářů, které se mají vytvořit. Zadaná hodnota bude explicitně konvertována do typu **Byte** a uložena do proměnné **odp**. Následně bude proveden příkaz **GoTo**, který přeskočí na specifikované návěstí (**Tvorba_Formulářů**).

Všimněte si, že v kódu není proveden test návratové hodnoty funkce **InputBox** (jednoduše předpokládáme, že uživatel zadá správnou hodnotu z ohraničeného celočíselného intervalu). Pokud bude tento kód používat v reálných aplikacích, ujistěte se, že jste provedli také test návratové hodnoty funkce **InputBox**. A to z toho důvodu, že i když se uživatel již jednou zmýlil, není vůbec vyloučeno, že tak udělá i podruhé.

2. Kód, jenž následuje za návěstím **Tvorba_Formulářů**, inicializuje pole **frm()** na hodnotu o jednotku menší nežli je hodnota parametru **Počet**. Zde je důležité zdůraznit, že Visual Basic .NET přistupuje k deklaraci polí jinak než jeho předchůdce. Dolní hranice pole je po novém vždy rovna nule, zatímco horní hranice není prakticky omezená. Při deklaraci pole je potřebné zadat hodnotu, která reprezentuje horní hranici pole (naproti tomu, v některých jiných jazycích platformy .NET se při deklaraci pole udává hodnota, jež představuje počet prvků pole – typicky v Managed Extensions for C++). Tak je připraveno pole, jehož prvky budou schopny uložit odkazy na instance třídy **Form**. V cyklu jsou všechny prvky pole naplněny příslušnými odkazy na vytvořené instance třídy **Form**.

K použití instance zapečetěné třídy nám bude stačit několik řádků programového kódu:




```
Dim x As New ZapečetěnáTřída()
x.VytvořitFormuláře(3)
Me.Text = "Počet vytvořených formulářů: " & _
CStr(x.ZjistitPočetFormulářů)
```

Přístup k registrům operačního systému Windows

Snad všichni programátoři a vývojáři chtějí, aby byly jejich aplikace co možná nejvíc uživatelsky přívětivé. Uživatelská přívětivost je značně rozsáhlý pojem, jenž v sobě absorbuje velké množství doporučených standardů, které říkají, jak má aplikace vypadat a jak se má chovat. Dodržováním těchto standardů tak mohou programátoři zabezpečit, že uživatelům se bude s aplikací pracovat pohodlně, rychle a bez jakýchkoliv potíží. Jedním z požadavků, které jsou na reálnou aplikaci kladeny, je schopnost aplikace pamatovat si potřebné informace o stavech, v nichž se během své činnosti ocitla. Tak si aplikace může pamatovat seznam posledně otevřených souborů, přesnou pozici svého dialogového okna, nebo časový interval pro automatické uložení editovaného dokumentu. Na uložení a posléze načítaní všech uvedených informací lze s výhodou využít registrů operačního systému.

Ve Visual Basicu .NET můžeme aplikovat dvě koncepce přístupu k registrům:

1. Starou koncepci, která je známá již z Visual Basicu 6 a při které se používají vestavěné funkce pro práci s registry (jde o funkce **SaveSetting**, **GetSetting**, **DeleteSetting** a **GetAllSettings**). Nevýhodou této koncepce je to, že všechny informace lze uložit jenom do předem určené sekce registrů (přesněji jde o sekci s názvem **HKEY_CURRENT_USER\Software\VB and VBA Program Settings**). Pro uložení informací do jiných sekcí registrů je nutné zavolat adekvátní API funkce, ovšem jedná se o značně náročnou záležitost. Použijete-li novou koncepci, nemusíte si s tímto omezením dále lámat hlavu, protože nové třídy rámce .NET Framework vám umožňují provádět zápis do libovolných oblastí registrů.

2. Novou a unikátní koncepci, kterou nabízí Visual Basic .NET. Pro přístup k registrům použijeme třídy za tímto účelem vytvořené (přesněji jde o třídy **Registry** a **RegistryKey**). Ačkoliv je použití nové koncepce poněkud komplikovanější, můžete údaje o své aplikaci ukládat do rozličných sekcí registrů, čímž je okamžitě likvidováno omezení staré koncepce.

Abyste viděli, jak obě koncepce pracují, ukážeme si jejich použití v praxi. Pokaždé provedeme vytvoření registrového klíče, jemuž přiřadíme příslušnou hodnotu a tuto hodnotu naplníme předem určenými daty.

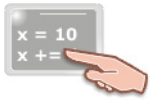
Při práci s registry je zapotřebí odlišovat dva pojmy: jméno hodnoty a data hodnoty. Zatímco jméno hodnoty představuje textový řetězec, který hodnotu uživatelsky deklaruje, data hodnoty determinují hodnotu po datové stránce. Jednoduše řečeno, data hodnoty jsou vlastnictvím hodnoty, a jsou to právě data, které jsou předmětem vstupně-výstupních operací při práci s registry. Mezi data hodnoty mohou patřit čísla, textové řetězce, nebo specifické konstanty.

Registry operačního systému jsou velmi důležitou entitou, a to jak ve vztahu k nainstalovaným aplikacím, tak i vůči samotnému operačnímu systému. Před zásahem do registrů proveďte vždy nejprve jejich zálohu. Při zásazích do registrů se ujistěte, že nemodifikujete údaje, které byly zapsány jinou aplikací a přímo upravujte jenom ty informace, které jste do registrů sami zapsali.

Demonstrace staré koncepce přístupu k registrům

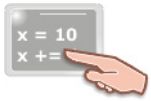
Postupujte podle následujících instrukcí:

1. Spusťte Visual Basic .NET a vytvořte novou aplikaci pro Windows (**Windows Application**).
2. Na formulář přidejte dvě instance ovládacího prvku **Button**, přičemž jednu pojmenujte jako **btnZápisHodnot** a druhou zase **btnČteníHodnot**. Rovněž vhodně upravte vlastnost **Text** obou instancí.
3. Událostní proceduru **Click** instance **btnZápisHodnot** upravte takto:



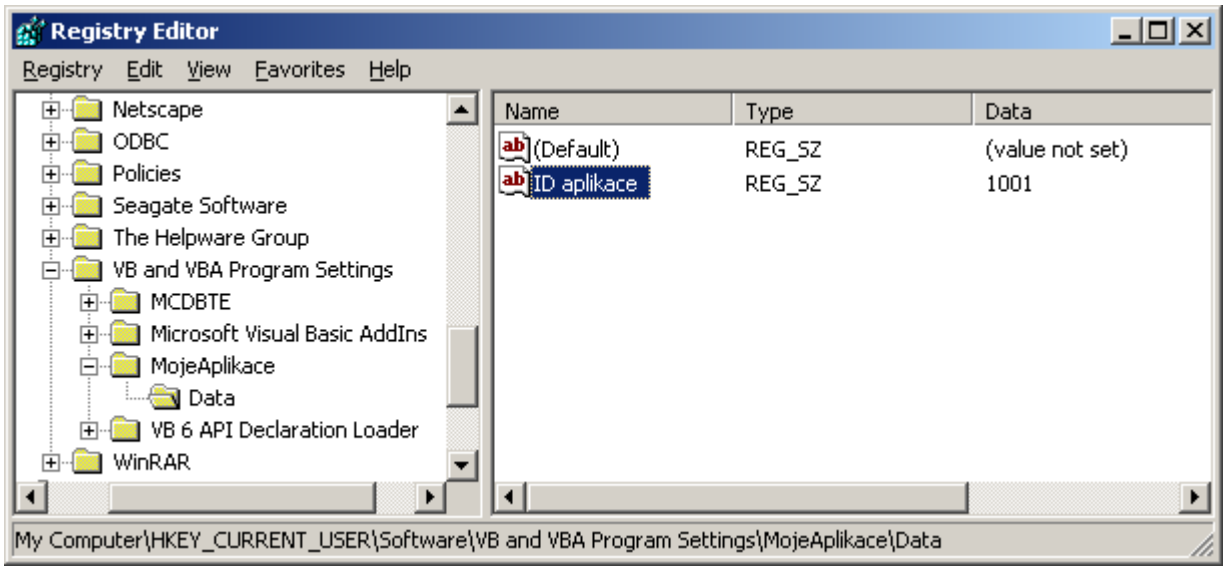

```
Private Sub btnZápisHodnot_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles btnZápisHodnot.Click
    SaveSetting("MojeAplikace", "Data", "ID aplikace", "1001")
End Sub
```

4. Událostní proceduru **Click** instance **btnČteníHodnot** modifikujte tímto způsobem:




```
Private Sub btnČteníHodnot_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles btnČteníHodnot.Click
    MessageBox.Show("ID aplikace je: " & _
    GetSetting("MojeAplikace", "Data", "ID aplikace"))
End Sub
```

5. Spusťte aplikaci (F5) a klepněte nejprve na tlačítko pro zápis hodnot do registrů a poté aktivujte druhé tlačítko. Měli byste obdržet informace o datech hodnoty klíče **\MojeAplikace\Data\ID aplikace**. Podíváte-li se do registrů, uvidíte, že byl vytvořen příslušný klíč a byla mu přiřazena hodnota (obr. 1).

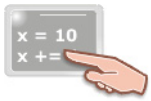


Obr. 1 – Použití funkcí **SaveSetting** a **GetSetting** pro přístup k hodnotě uložené v registrech

Demonstrace nové koncepce přístupu k registrům

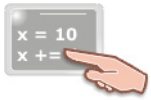
Jak jsme si již řekli, při použití této koncepce se vyhneme použití funkcí typu **SaveSetting** a **GetSetting**. Místo nich totiž povoláme k práci specializované třídy **Registry** a **RegistryKey** z jmenného prostoru **Microsoft.Win32**. Postupujte takto:

1. Rozšiřte stávající kolekci instancí ovládacího prvku **Button** o další dvě, které pojmenujte jako **btnZápisHodnot2** a **btnČteníHodnot2** (opět upravte také hodnoty vlastnosti **Text** obou instancí).
2. Před kód třídy formuláře (implicitně **Form1**) vložte následující řádek pro vložení jmenného prostoru **Microsoft.Win32**:



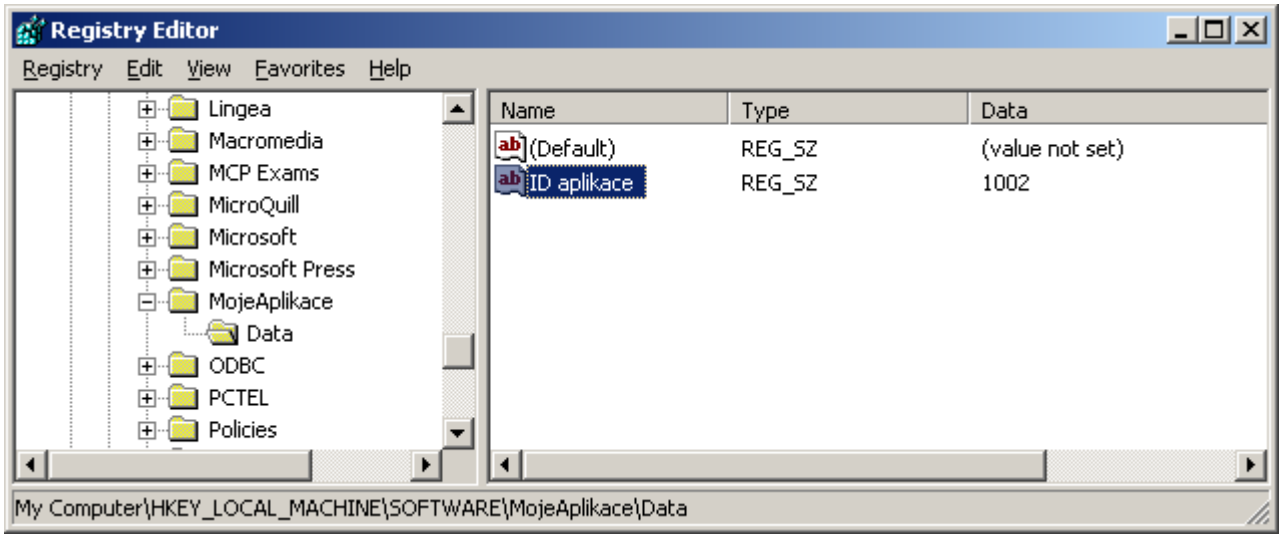

```
Imports Microsoft.Win32
```

3. Událostní proceduru **Click** instance **btnZápisHodnot2** pozměňte následovně:




```
Private Sub btnZápisHodnot2_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles btnZápisHodnot2.Click
    Dim reg As RegistryKey
    reg = Registry.LocalMachine.CreateSubKey("Software\MojeAplikace\Data")
    reg.SetValue("ID aplikace", "1002")
End Sub
```

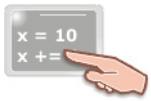
Pohled do registrů přibližuje obr. 2.



Obr. 2 – Použití tříd **Registry** a **RegistryKey** pro přístup k hodnotě uložené v registrech

I když opět vytváříme klíč se stejným názvem, v tomto případě je klíč uložen do jiné sekce registrů (přesněji do **HKEY_LOCAL_MACHINE\Software**). Abychom získali přístup do této sekce registrů, musíme použít třídu **Registry** a její veřejný sdílený datový člen **LocalMachine**. Pro vytvoření nového klíče použijeme metodu **CreateSubKey**, které předáme doplňkovou cestu ke klíči v uvedené podobě (plná cesta k našemu klíči vypadá takto: **HKEY_LOCAL_MACHINE\Software\MojeAplikace\Data**). Návratovou hodnotou metody **CreateSubKey** je instance třídy **RegistryKey**, přičemž odkaz na tuto instanci je uložen do proměnné **reg**. Dobrá, v této chvíli máme vytvořen klíč, ovšem náš úkol ještě není splněn. Abychom klíči přiřadili hodnotu a hodnotu asociovali s platnými daty, zavoláme metodu **SetValue**, která nabídneme jméno hodnoty ve tvaru datového typu **String** (**"ID aplikace"**) a data hodnoty (**1002**).

Budete-li chtít data hodnoty načíst, upravte událostní proceduru **btnČteníHodnot2_Click** tak, jak je uvedeno níže:



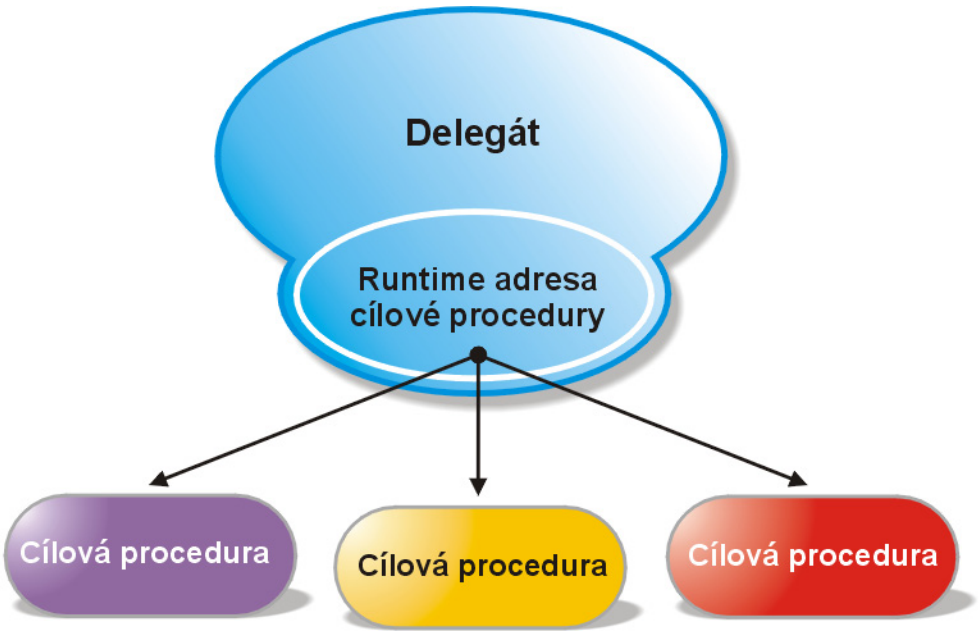

```
Private Sub btnČteníHodnot2_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles btnČteníHodnot2.Click
    Dim reg2 As RegistryKey
    reg2 = Registry.LocalMachine.OpenSubKey("Software\MojeAplikace\Data", False)
    MessageBox.Show("Hodnota zapsaná v registrech: " & _
    CStr(reg2.GetValue("ID aplikace")))
End Sub
```

Abychom otevřeli požadovaný registrový klíč, zavoláme metodu **OpenSubKey**. V našem případě je použita přetížená varianta metody, která pracuje se dvěma parametry: prvním je název klíče a druhým pak booleovská hodnota, která určuje, zdali je možné hodnotu klíče modifikovat. K datům hodnoty se dostaneme pomocí metody **GetValue**, která poskytneme jméno hodnoty, data které chceme obdržet. Metoda **GetValue** vrací data hodnoty (v podobě datového typu **Object**), která jsou v naší ukázce přetypována do podoby datového typu **String** a zobrazena v dialogu.

Vytváření vlastního delegáta

Přecházíte-li na programování pro platformu .NET, zcela jistě si zanedlouho všimnete speciálního programovacího aparátu, který se doposud v programování nevyskytoval. Ano, touto novinkou jsou delegáti, neboli objekty, které usnadňují přístup k metodám a funkcím jiných objektů za běhu programu. O delegátech se často mluví jako o typově bezpečných funkčních ukazatelích. Pokud patříte mezi programátory v C/C++, mohli byste namítat, že funkční ukazatele jsou tady již nějaký ten pátek, tak jakážto novinka. Ovšem, v jistém smyslu máte pravdu, avšak třeba zdůraznit, že delegáty nabízejí daleko větší funkcionalitu nežli opravdové funkční ukazatele. Kromě toho, s delegáty můžete pracovat v mnoha .NET jazycích, např. v C# či Managed Extensions for C++.

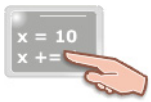
V dnešní ukázce si předvedeme, jak sestrojit vlastního delegáta. Ještě předtím se však podívejme na základní způsob práce delegáta. Delegát je objekt, jenž může za běhu aplikace získat runtime adresu požadované procedury a posléze tuto proceduru aktivovat. Ve skutečnosti ovšem může delegát aktivovat jakoukoliv proceduru, která má stejný seznam parametrů a stejnou návratovou hodnotu jako samotný delegát. Delegát tak nemusí nutně volat jednu a tu samou proceduru, neboť je schopen aktivovat libovolnou proceduru, která vyhovuje požadovaným podmínkám. Rozhodnutí, kterou proceduru zavolat, lze vyřešit až za běhu aplikace.



Obr. 3 – Schematické znázornění práce delegáta

A nyní si ukažme, jak přistoupit k tvorbě delegáta:

1. Spusťte Visual Basic .NET a vytvořte aplikaci pro Windows (**Windows Application**).
2. Na plochu formuláře klepněte pravým tlačítkem myši a z kontextové nabídky vyberte položku **View Code**.
3. Programový kód třídy **Form1** upravte podle níže uvedeného vzoru:




```
Option Strict On
'Vložené odkazy na potřebné jmenné prostory.
Imports System.Drawing
Imports System.Drawing.Drawing2D

Public Class Form1
    Inherits System.Windows.Forms.Form

    'Deklarace delegáta.
    Private Delegate Function Delegát() As Boolean

    '****** Deklarace první cílové funkce. ******
    'Funkce vytváří instanci třídy ListBox, do které budou uložena
    'náhodně vygenerovaná čísla.
    Private Function MojeFunkce() As Boolean
        Try
            'Vytvoření instance třídy ListBox a její umístění
            'do kolekce ovládacích prvků formuláře.
            Dim lstSeznam As New ListBox()
            With lstSeznam
                .Location = New Point(0, 0)
                .Size = New Size(Me.Width \ 2, Me.Height \ 2)
            End With
            Me.Controls.Add(lstSeznam)
```

Programový kód pokračuje na následující straně

```
            'Výpočet druhé mocniny čísel z intervalu <0, 9>.
            Dim a As Byte
            a = 0
            Do While a < 10
                Dim x As Short
                x = CShort(a ^ 2)
                lstSeznam.Items.Add(x)
                a += CByte(1)
            Loop
            'Zachycení jakékoliv výjimky, která by mohla být generována.
        Catch e As Exception
            'Dojde-li k chybě, funkce je ukončena a vrací hodnotu False.
            Return False
        End Try
        'Při úspěšném ukončení funkce je vrácena hodnota True.
        Return True
    End Function

    '****** Deklarace druhé cílové funkce. ******
    'Funkce vyplňuje plochu formuláře gradientní výplní.
    Private Function MojeKreslícíFunkce() As Boolean
        Try
            'Vytvoření instance třídy Rectangle
            Dim rec As New Rectangle(New Point(0, 0), _
                    New Size(Me.Width, Me.Height))
            'Vytvoření grafického štětce pomocí třídy LinearGradientBrush
            Dim br As New LinearGradientBrush(rec, Color.Gold, _
            Color.Magenta, LinearGradientMode.ForwardDiagonal)
            'Vytvoření grafického objektu
            Dim g As Graphics = Me.CreateGraphics()
            'Kreslení gradientní výplně.
            g.FillRectangle(br, rec)
        Catch e As Exception
            'Dojde-li k chybě, funkce je ukončena a vrací hodnotu False.
            Return False
        End Try
        'Při úspěšném ukončení funkce je vrácena hodnota True.
        Return True
    End Function

    Private Sub Form1_Click(ByVal sender As Object, _
    ByVal e As System.EventArgs) Handles MyBase.Click
        'Inicializace generátoru náhodných čísel na bázi systémového času
        Randomize()
        'Deklarace proměnné, ve které bude uloženo získané
        'náhodné číslo
        Dim sHodnota As Short
        'Vytvoření referenční proměnné, která bude uchovávat odkaz
        'na později vytvořenou instanci delegáta
        Dim InstanceDelegáta As Delegát
        'Generování náhodného čísla z intervalu <1,10>
        sHodnota = CShort(Int(10 * Rnd()) + 1)
        'Jestliže je vygenerované číslo z intervalu <1,5> tak...
        If sHodnota >= 1 And sHodnota <= 5 Then
            'Vytvoření instance delegáta, která obsahuje
            'runtime adresu funkce MojeFunkce
            InstanceDelegáta = New Delegát(AddressOf MojeFunkce)
            'jinak...
        Else
            'Vytvoření instance delegáta, která obsahuje runtime
            'adresu funkce MojeKreslícíFunkce
            InstanceDelegáta = New Delegát(AddressOf MojeKreslícíFunkce)
```

Programový kód pokračuje na následující straně

```
        End If
        'Test návratové hodnoty cílové funkce
        If InstanceDelegáta.Invoke() Then
            Me.Text = "Funkce byla úspěšně zavolána."
        Else
            Me.Text = "Při práci funkce došlo k potížím."
        End If
    End Sub
End Class
```

Bonusový materiál pro čtenáře rubriky Visual Basic	Autor: **Ján Hanák**

Speciál pro programátory

Visual Basic .NET a Managed Extensions for C++

Časová náročnost (min):	Začátečník	Pokročilý	Profesionál
65	1	2	3

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

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++.

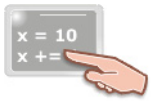
Z uvedeného výčtu je zřejmé, že Managed Extensions for C++ jsou rozšířeními jazyka C++ pro vytváření aplikací pro **CLR** a .NET Framework. Na druhé straně ovšem nelze Managed Extensions for C++ chápat jako jenom pouhé rozšíření, protože jde vskutku o nový programovací jazyk s mnoha novými konstrukcemi 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 směřují 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;
```

Programový kód pokračuje na následující straně

```
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);

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

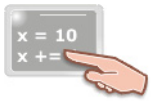
Tento programový kód ukazuje třídu **Kresleni**, která obsahuje jednu veřejnou členskou funkci s názvem **ZacitKresleni**. V deklaračním příkazu třídy se nachází klíčové slovo **__gc**, které naznačuje, že jde o řízenou (**garbage-collected**) třídu. Instance řízené třídy budou po vytvoření umístěné 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:

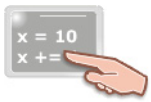



```
Public Class Podtřída
    Inherits KnihovnaDLL.Kresleni
End Class
```

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.

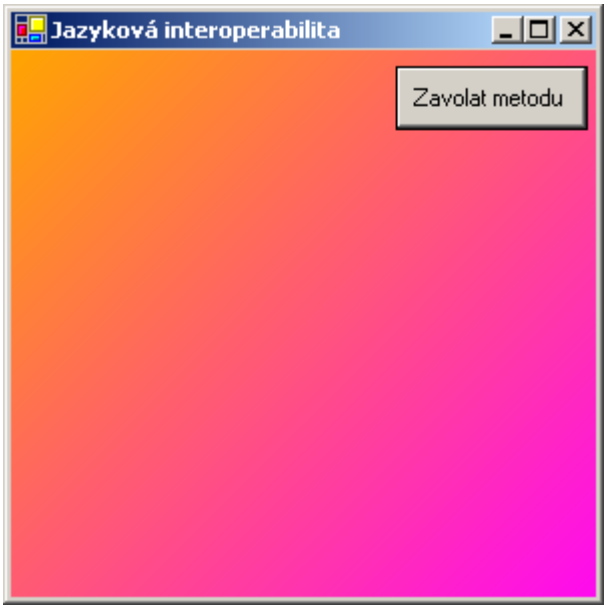
Testování jazykové interoperability

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

Integrované prostředí rubriky Visual Basic	Autor: **Ján Hanák**

Téma měsíce

Inteligentní **značky** (Smart Tags)

Časová náročnost (min): **125**	Začátečník	Pokročilý	Profesionál
			

Použitý operační systém :	Windows 2000 SP3
Hlavní vývojový nástroj :	Visual Basic 6.0 SP 5
Další vývojový software :	Microsoft Office XP
Jiný software :	Smart Tags SDK v.1.1

Vážení čtenáři,

vánoční vydání sekce Téma měsíce bude vskutku jedinečné, protože se budeme věnovat oblasti, která je pro mnoho programátorů velkou neznámou, a sice vývojem vlastních inteligentních značek pro kancelářské aplikace z balíku Office XP. Poznáte funkční jádro, které pohání technologii inteligentních značek a připravíte si svou vlastní dynamicky linkovanou knihovnu (ActiveX DLL) s kódem vaší inteligentní značky. Přeji vám příjemné počtení.

Obsah

Inteligentní značky na první pohled

Inteligentní značky představují moderní technologii, která se poprvé objevila v softwarovém balíku kancelářských aplikací Office XP od společnosti Microsoft. Původně se inteligentní značky jmenují Smart Tags, ovšem my budeme pracovat s již celkem vžitým českým ekvivalentem. Pokud jste tedy někdy měli tu čest pracovat s (prozatím) poslední verzí balíku Office, pravděpodobně jste se již s inteligentními značkami setkali. Pokud ne, vydejte se s námi na krátkou seznamovací cestu, v rámci které uvidíte, proč jsou inteligentní značky tak důležitou a rovněž vzrušující technologií.

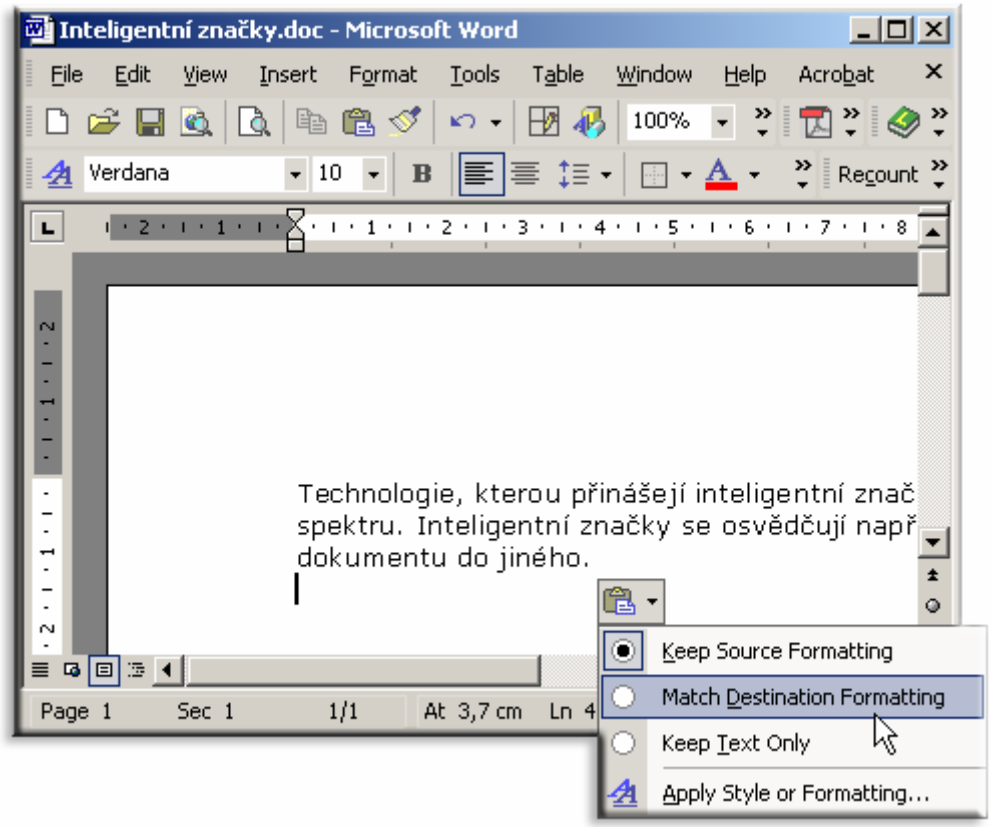
Hlavním krokem, který vedl vývojáře z Redmondu k vytvoření a zařazení inteligentních značek do populárního softwaru bylo, jak jinak, usnadnění práce finálních uživatelů s množstvem různých typů zpracovávaných dat. Vezměme si třeba průměrného uživatele počítače, jenž ovládá na přijatelné úrovni práci s operačním systémem, textovým procesorem a tabulkovým kalkulátorem. Tento uživatel nejspíš často pracuje s textovými dokumenty a tabulkovými sešity. Je také více než pravděpodobné, že mnoho údajů, které se nacházejí v uživatelových tabulkách, má jistou souvislost s textovými daty, které uživatel používá při psaní dokumentů v textovém procesoru. A dokonce je možné, že některé typy dat z textových dokumentů a tabulek mohou sloužit pro potřeby časového

managementu v aplikaci Outlook. Je-li tomu tak, pročpak neusnadníme uživateli jeho práci a nezabudujeme do aplikací technologii, která bude vstupní data analyzovat a v případě potřeby je kombinovat, navzájem formátovat, nebo provádět nad těmito daty jiné požadované operace? Ano, takto patrně vypadala počáteční myšlenka členů vývojového týmu pro stavbu inteligentních značek.

Jestliže uživatel zapíše do textového dokumentu jméno svého přítele nebo obchodního partnera, aplikace by měla být natolik inteligentní, aby zadaný textový řetězec poznala a nabídla uživateli několik akcí, které lze s předmětným řetězcem uskutečnit. V tomto případě by mohlo jít o zaslání elektronické zprávy dané osobě, nebo o přidání jména osoby do elektronického diáře uživatele.

Z pohledu uživatele je nejpodstatnější zejména fakt, že obě operace lze realizovat pouhým klepnutím myší a není nutné manuálně kopírovat data, spouštět další aplikace a data opětovně vkládat.

Technologie, kterou přinášejí inteligentní značky, je využitelná v daleko širším aplikačním spektru. Inteligentní značky se osvědčují například při kopírování části textu z jednoho dokumentu do jiného. Po vložení textu se objeví inteligentní značka, která ukazuje, že je možné s vložený textem něco provést, třeba zladit jeho formátování s formátování cílového dokumentu, do něhož byl text vložen (obr. 1).



Obr. 1 – Inteligentní značka v praxi

Automatické opravy textu jsou rovněž reprezentovány prostřednictvím podporované inteligentní značky. Ihned ze začátku je ovšem dobré mít na paměti, že s inteligentními značkami se můžete střetnou jenom v Office XP (2002), předchozí verze touto technologií nedisponují.

Ačkoliv je využití inteligentních značek nejintenzivnější právě v aplikacích Office XP (Word XP, Excel XP a Outlook XP), ve skutečnosti se s inteligentními značkami můžete setkat také v HTML dokumentech.

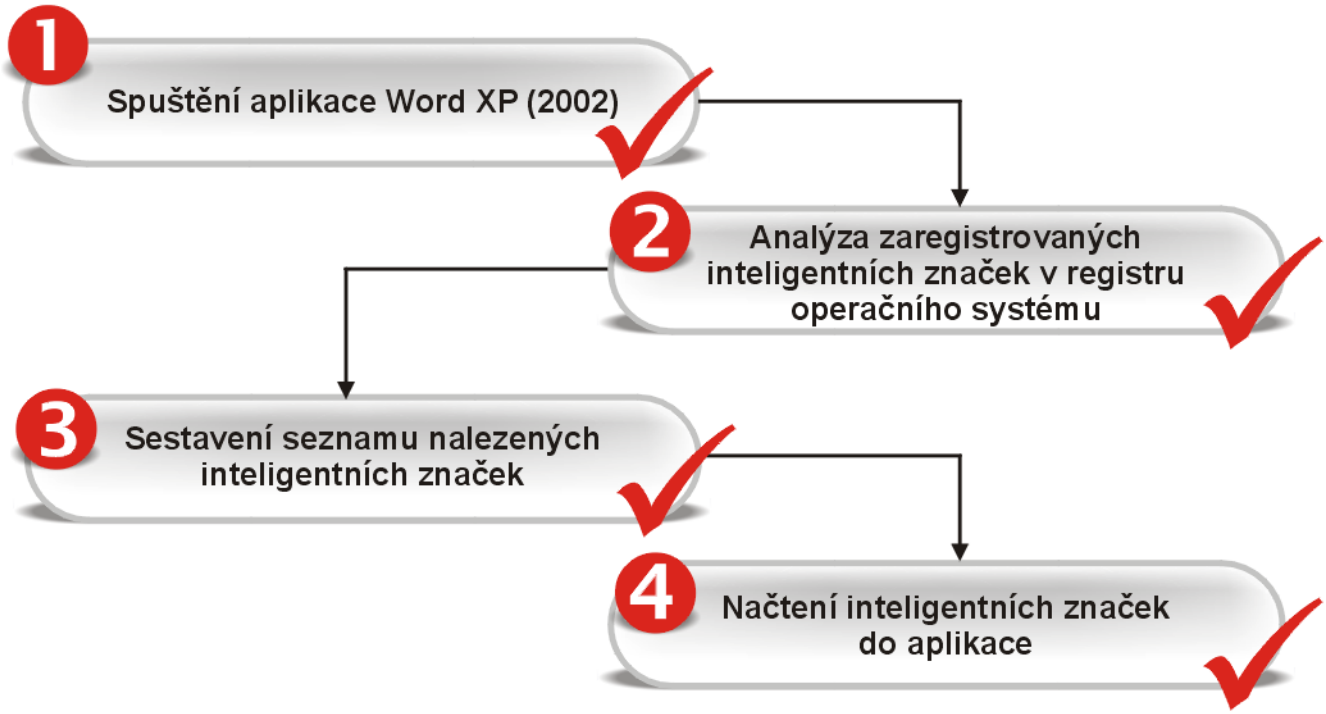
Každá aplikace z balíku Office XP (2002) obsahuje jistou sadu zabudovaných inteligentních značek, které se automaticky aktivují v případě potřeby (o technickém pohledu na způsob práce inteligentních značek budeme mluvit za chvíli). Kromě inteligentních značek, které jsou implicitně vestavěné do aplikací Office XP, existují také další inteligentní značky, a to buď přímo od společnosti Microsoft, nebo od jiných softwarových firem. Další možností, jak získat nové inteligentní značky, je jejich samostatné naprogramování a zaregistrování v operačním systému. A právě tuto alternativu si předvedeme podrobněji na dalších řádcích tohoto dokumentu.

Spolupráce inteligentních značek a aplikací Office XP

Všechny inteligentní značky, se kterými se můžete setkat v aplikacích Word XP a Excel XP, musejí být před svým použitím platně zaregistrovány v operačním systému. Tato registrace je mimořádně důležitá, protože jenom registrované inteligentní značky jsou schopné obě aplikace rozeznat a pracovat s nimi. Popojeďme však o kousek dál a povězme si, jak s inteligentními značkami pracuje například aplikace Word XP. Po každém spuštění aplikace Word XP je realizováno zjištění, jaké typy inteligentních značek jsou zaregistrované v operačním systému. Poté aplikace sestaví seznam nalezených inteligentních značek a provede jejich načtení.

Zjištění zaregistrovaných inteligentních značek a jejich implementace do aplikace trvá pouze několik sekund.

Jakmile uživatel zapíše do dokumentu textový řetězec, technologie inteligentních značek podrobí řetězec analýze a zjistí, zdali se jedná o „citlivý" řetězec. Pokud je řetězec citlivý, znamená to, že byl rozpoznán jako typ dat, nad kterým operuje jistá inteligentní značka. Citlivý textový řetězec je opatřen tečkovaným podtržením fialové barvy. Když uživatel najede kurzorem myši na citlivý textový řetězec, objeví se tlačítko se šipkou. Ukáže-li uživatel na objevivší se tlačítko, bude zobrazena kontextová nabídka se seznamem akcí, které lze nad vstupním typem dat (tedy nad textovým řetězcem) provést. Po zvolení požadované akce je tato provedena, přičemž výsledek akce je rozličný a závisí od typu inteligentní značky. Některé inteligentní značky mohou měnit formátování textového řetězce, jiné zase spouštět externí aplikace či internetové prohlížeče.

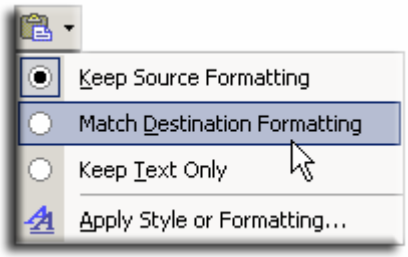


Obr. 2 – Spolupráce inteligentních značek a aplikace Word XP

Architektura inteligentních značek

Technologie inteligentních značek je postavena na vlastním aplikačním programovým rozhraním (API), které vyhovuje specifikacím standardu COM (Component Object Model). API inteligentních značek je uloženo v typové knihovně s názvem **Microsoft Smart Tags 1.0** (tato knihovna je dodávána společně s Office XP). Aplikační programové rozhraní obsahuje dvě důležitá rozhraní, která budeme při vývoji vlastní inteligentní značky rozhodně potřebovat. Jde o rozhraní **ISmartTagRecognizer** a **ISmartTagAction**. Ještě předtím, než se na obě rozhraní podíváme blíže, si musíme ozřejmit dva pojmy, jimiž jsou identifikátor (**recognizer**) a akce (**action**). Znalost obou termínů je nezbytně nutná pro pochopení způsobu práce inteligentních značek.

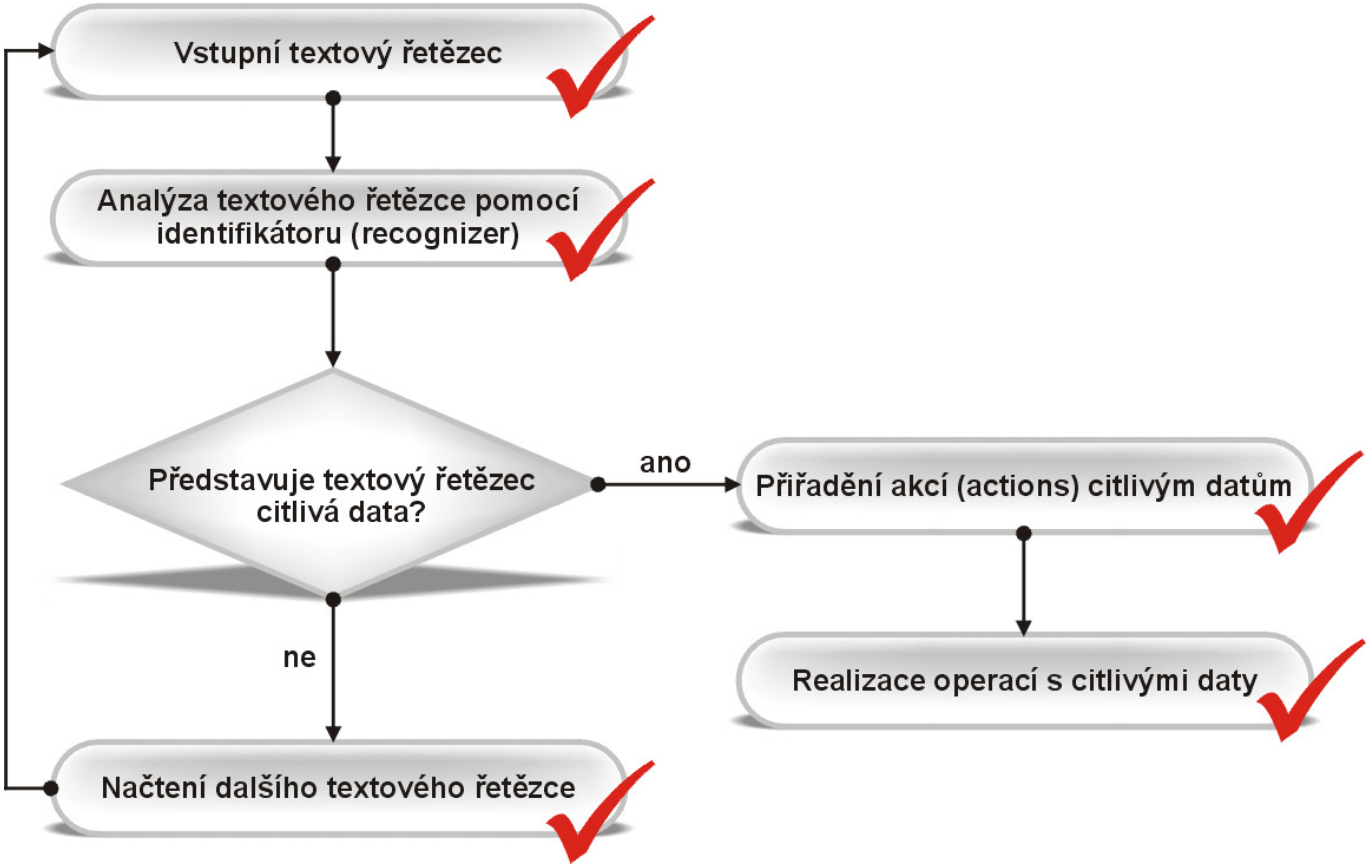
Identifikátor (**recognizer**) představuje programový kód, pomocí něhož cílová aplikace rozezná citlivá data. Citlivá data jsou data, která mohou být pomocí dané inteligentní značky podrobena jistým programovým operacím. Každá inteligentní značka obsahuje uzavřenou množinu dat, kterou dokáže identifikovat. Identifikace dat je realizována právě pomocí identifikátoru. Identifikátor sleduje, s jakými daty uživatel pracuje. V případě aplikace Word XP jsou těmito daty textové řetězce. Každý zadaný textový řetězec je prozkoumán prostřednictvím identifikátorů všech podporovaných inteligentních značek. Pokud zadaný textový řetězec vyhovuje specifikacím identifikátoru jedné z inteligentních značek, je tento textový řetězec rozeznán a z pohledu API označen jako citlivý. Skutečnost, že máme do činění s citlivým textovým řetězcem, je vizuálně umocněna tečkovaným podtržením tohoto řetězce. Výsledním produktem práce identifikátoru je zjištění citlivých dat. Jsou-li k dispozici citlivá data, je možné nad těmito daty uskutečnit jisté programové operace, kterým se v prostředí API inteligentních značek říká akce (**actions**). Akce představují operace, které lze s citlivými daty provést. Vizuálně jsou všechny akce seskupeny do jednotné kontextové nabídky (obr. 3).



Obr. 3 – Akce inteligentní značky

Pokud uživatel klepne na některou z položek kontextové nabídky, spustí se příslušná akce. Jak si můžete všimnout, jedna inteligentní značka může disponovat i několika akcemi, přičemž počet podporovaných akcí není omezen a je jenom na programátorovi, kolik akcí do vlastní inteligentní značky implementuje.

Proces práce identifikátoru a akcí inteligentních značek můžete vidět na obr. 4.



Obr. 4 – Proces práce identifikátoru a akcí inteligentních značek

Oběma vysvětleným termínům odpovídají již zmíněná rozhraní: Kód pro implementaci identifikátoru (**recognizer**) obsahuje rozhraní **ISmartTagRecognizer**, zatímco pro akce (**actions**) inteligentních značek je k dispozici rozhraní **ISmartTagAction**. Budete-li chtít vytvořit vlastní inteligentní značku, budete muset implementovat několik vlastností a metod obou rozhraní.

Visual Basic 6.0 není jediným prostředím, ve kterém lze vytvářet dynamicky linkovanou knihovnu (ActiveX DLL) s kódem inteligentní značky. Za tímto účelem můžete použít také další programovací jazyky, které jsou otevřeny specifikaci COM, například Visual C++.

Vytváříme inteligentní značku

Konečně se dostáváme k samotnému procesu vývoje inteligentní značky. Vytváření inteligentní značky bude demonstrovat postup, jenž se nachází na následujících řádcích. Naše inteligentní značka bude vyhledávat v textovém dokumentu aplikace Word XP specifické počítačové výrazy, jako např. CPU, AGP a další. Pokud bude v dokumentu nalezen jeden z podporovaných výrazů, bude zobrazeno akční tlačítko s kontextuální nabídkou. V nabídce bude umístěna jedna akce, resp. položka, po aktivaci které bude spuštěn internetový prohlížeč s adresou elektronické počítačové encyklopedie (v našem případě vymyšlené). Nuže, pusťme se do práce:

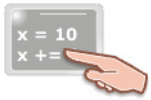
1. Spusťte Visual Basic 6.0 a vytvořte aplikaci typu **ActiveX DLL**.
2. Projekt přejmenujte na **MojeZnačka**.
3. Třídu **Class1** přejmenujte na **CIdentifikátor**.
4. Ujistěte se, že v sekci **Declarations** třídy **CIdentifikátor** je zapsán řetězec **Option Explicit**, který zabezpečuje vynucení deklarace proměnných.
5. Vyberte nabídku **Project** a klepněte na položku **References**. V seznamu **Available References** vyhledejte položku **Microsoft Smart Tags 1.0 Type Library** a zatrhněte ji. Poté stiskněte

tlačítko OK. Tímto jste přidali do vašeho projektu odkaz na vzpomínanou typovou knihovnu, v níž se nacházejí rozhraní **ISmartTagRecognizer** a **ISmartTagAction**.

Nyní začneme s implementací požadovaných vlastností a metod rozhraní **ISmartTagRecognizer**. Všechen kód budete umísťovat přímo do třídy **CIdentifikátor**.

Implementace rozhraní ISmartTagRecognizer

1. Ihned za příkaz **Option Explicit** vložte tento řádek kódu:



```
Implements ISmartTagRecognizer
```

Příkaz **Implements** říká, že budeme chtít implementovat vlastnosti a metody rozhraní **ISmartTagRecognizer**.

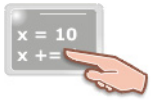
2. Ze všeho nejdříve začneme s implementací vlastností **ProgID**, **Name** a **Desc**. Popis vlastností je uveden v tab. 1.

Název vlastnosti	Charakteristika
ProgID	Jednoznačný programový identifikátor třídy, která implementuje rozhraní **ISmartTagRecognizer**. Tento identifikátor je jazykově nezávislý a nabízí způsob, jak se odkazovat na DLL s kódem třídy prostřednictvím objektových modelů cílových aplikací.
Name	Jméno pro identifikátor. Toto jméno se zobrazuje na záložce **Smart Tags** dialogového okna **AutoCorrect Options** v aplikacích Word XP a Excel XP. (Nabídka **AutoCorrect Options** je dostupná z nabídky **Tools**).
Desc	Stručná charakteristika druhu činnosti identifikátoru inteligentní značky.

Tab. 1 – Charakteristika vlastností **ProgID**, **Name** s **Desc**

Vlastnosti **ProgID**, **Name** a **Desc** jsou určeny jenom pro čtení (**read-only**).

3. Kód vlastností **ProgID**, **Name** a **Desc** je následovní:



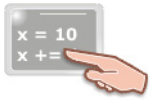
```
Private Property Get ISmartTagRecognizer_ProgId() As String
ISmartTagRecognizer_ProgId = "MojeZnačka.Identifikátor"
End Property

Private Property Get ISmartTagRecognizer_Name _
(ByVal LocaleID As Long) As String
ISmartTagRecognizer_Name = "Moje inteligentní značka"
End Property

Private Property Get ISmartTagRecognizer_Desc _
(ByVal LocaleID As Long) As String
ISmartTagRecognizer_Desc = "Jednoduchý příklad použití" _
& "inteligentní značky."
End Property
```

4. Pokračujeme implementací vlastností **SmartTagCount**, **SmartTagName** a **SmartTagDownloadURL**. Popis vlastností se nachází v tab. 2.

Název vlastnosti	Charakteristika
SmartTagCount	Vlastnost určuje počet inteligentních značek, které je identifikátor schopen rozeznat.
SmartTagName	Jedinečné pojmenovaní pro typy inteligentních značek, které podporuje identifikátor.
SmartTagDownloadURL	Určení adresy, ze které lze stáhnout nové akce pro danou inteligentní značku.

Tab. 2 – Charakteristika vlastností **SmartTagCount**, **SmartTagName** s **SmartTagDownloadURL**

5. Programový kód vlastností **SmartTagCount**, **SmartTagName** a **SmartTagDownloadURL** má tuto podobu:

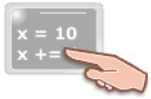


```
Private Property Get ISmartTagRecognizer_SmartTagCount() As Long
ISmartTagRecognizer_SmartTagCount = 1
End Property

Private Property Get ISmartTagRecognizer_SmartTagName _
(ByVal SmartTagID As Long) As String
If SmartTagID = 1 Then
    ISmartTagRecognizer_SmartTagName = _
    "značka_01#MojeZnačka"
End If
End Property

Private Property Get ISmartTagRecognizer_SmartTagDownloadURL _
(ByVal SmartTagID As Long) As String
ISmartTagRecognizer_SmartTagDownloadURL = ""
End Property
```

Náš identifikátor bude pracovat pouze s jednou inteligentní značkou (**SmartTagCount = 1**), název které je **MojeZnačka**. Plně kvalifikovaný název inteligentní značky je složen ze dvou celků: jména jmenného prostoru inteligentní značky a samotného názvu inteligentní značky. V našem případě je jmenný prostor determinován textovým řetězcem **značka_01**. Za jmenným prostorem následuje symbol mřížky (**#**), za kterým je uvedeno pojmenování konkrétní inteligentní značky (**MojeZnačka**). Kompletní název inteligentní značky je proto "**značka_01#MojeZnačka**". Jelikož nepředpokládáme, že bude existovat webová stránka, z níž bude možné získat nové informace pro inteligentní značku, vlastnost **SmartTagDownloadURL** je naprogramována tak, aby vracela prázdný řetězec.

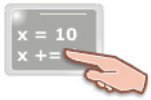
V dalším kroku se zaměříme na sestavení seznamu textových řetězců, které bude identifikátor identifikovat jako citlivá data. Tento úkol splníme tak, že vytvoříme pole textových řetězců, které posléze naplníme požadovanými textovými řetězci. Bude-li identifikátorem rozeznán jakýkoliv z těchto řetězců v textovém dokumentu, bude opatřen tečkovaným podtržením a akčním tlačítkem pro zobrazení podporovaných akcí inteligentní značky.

6. Za příkaz **Implements ISmartTagRecognizer** vložte tyto dva řádky programového kódu:



```
Dim Rětězce(5) As String
Dim PočetRětězců As Integer
```

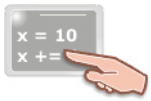
7. Deklarované pole **Rětězce** inicializujeme v událostní proceduře **Class_Initialize**:



```
Private Sub Class_Initialize()
    Rětězce(1) = "pc"
    Rětězce(2) = "cpu"
    Rětězce(3) = "gpu"
    Rětězce(4) = "agp"
    Rětězce(5) = "ht"
    PočetRětězců = 5
End Sub
```

Všechny textové řetězce zapisujte použitím malých písmen abecedy. Tato forma psaní textových řetězců je v tomto případě důležitá proto, aby bylo možné provádět vyhledávání textových řetězců s důrazem na rozlišování malých a velkých písmen abecedy. Algoritmus pro vyhledávání textových řetězců se nachází v následujícím fragmentu zdrojového kódu.

8. Posledním krokem bude implementace metody **Recognize** rozhraní **ISmartTagRecognizer**. V těle této metody je uložen kód, jenž bude zjišťovat, zdali se vstupní textový řetězec nachází v poli s podporovanými textovými řetězci. Připomeňme, že je realizováno „case-sensitive" vyhledávání.



```
Public Sub ISmartTagRecognizer_Recognize(ByVal Text As String, _
ByVal DataType As SmartTagLib.IF_TYPE, ByVal LocaleID As Long, _
ByVal RecognizerSite As SmartTagLib.ISmartTagRecognizerSite)
    Dim x As Integer
    Dim Index As Integer, VelikostŘetězce As Integer
    Dim Pb As SmartTagLib.ISmartTagProperties
    Text = LCase(Text)
    For x = 1 To PočetRětězců
        Index = InStr(Text, Rětězce(x))
        VelikostŘetězce = Len(Rětězce(x))
        While (Index > 0)
            Set Pb = RecognizerSite.GetNewPropertyBag
            RecognizerSite.CommitSmartTag _
            "značka_01#MojeZnačka", Index, VelikostŘetězce, Pb
            Index = InStr(Index + VelikostŘetězce, Text, Rětězce(x))
        Wend
    Next x
End Sub
```

Implementace členů rozhraní ISmartTagAction

Abychom mohli inteligentní značce přiřadit také odpovídající akce, musíme rovněž implementovat členy rozhraní **ISmartTagAction**. Implementace těchto členů se bude odehrávat v další třídě, a proto do stávajícího projektu přidejte novou třídu a pojmenujte ji **CAkce**. Dále pokračujte následovně:

1. Podobně jako třída **CIdentifikátor**, i třída **CAkce** musí obsahovat příkaz **Implements** s určením rozhraní, jehož členy se chystáme implementovat. Proto ihned za příkaz **Option Explicit** vložte tento řádek zdrojového kódu:

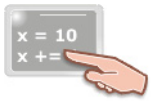
```
Implements ISmartTagAction
```

2. Do nově vytvořené třídy **CAkce** zařadíme programový kód pro vlastnosti **ProgID**, **Name** a **Desc**. Význam vlastností je vysvětlen v tab. 3.

Název vlastnosti	Charakteristika
ProgID	Jednoznačný identifikátor třídy, která implementuje rozhraní **ISmartTagAction**.
Name	Stručná charakteristika konkrétní implementace rozhraní **ISmartTagAction**.
Desc	Delší a podrobnější popis práce implementovaného rozhraní **ISmartTagAction**.

Tab. 3 – Charakteristika vlastností **ProgID**, **Name** a **Desc** rozhraní **ISmartTagAction**

3. Programový kód pro implementaci vlastností:



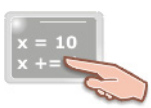

```
Private Property Get ISmartTagAction_ProgId() As String
    ISmartTagAction_ProgId = "MojeZnačka.Akce"
End Property

Private Property Get ISmartTagAction_Name _
(ByVal LocaleID As Long) As String
  ISmartTagAction_Name = "Moje inteligentní značka"
End Property

Private Property Get ISmartTagAction_Desc _
(ByVal LocaleID As Long) As String
  ISmartTagAction_Desc = "Provádí akce pro inteligentní značku."
End Property
```

4. Dále budeme implementovat vlastnosti **SmartTagCount**, **SmartTagName** a **SmartTagCaption**. Charakteristiku uvedených vlastností najdete v tab. 4.

Název vlastnosti	Charakteristika
SmartTagCount	Determinuje počet typů inteligentních značek, které může rozpoznat identifikátor. V naší ukázce pracujeme pouze s jedním typem inteligentní značky, a proto bude tato vlastnost vracet celočíselnou hodnotu 1.
SmartTagName	Jde o pojmenování podporovaného typu inteligentní značky v již známém formátu, jenž je složen z jmenného prostoru, znaku mřížky (#) a samotného názvu typu značky.
SmartTagCaption	Textový řetězec, jenž se bude objevovat v kontextové nabídce akcí inteligentní značky.

Tab. 4 – Popis vlastností **SmartTagCount**, **SmartTagName** a **SmartTagCaption**

5. Kód pro implementaci vlastností **SmartTagCount**, **SmartTagName** a **SmartTagCaption** vypadá takto:




```
Private Property Get ISmartTagAction_SmartTagCount() As Long
   ISmartTagAction_SmartTagCount = 1
End Property
```

Programový kód pokračuje na následující straně

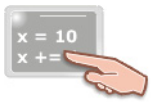
```
Private Property Get ISmartTagAction_SmartTagName _
(ByVal SmartTagID As Long) As String
    If (SmartTagID = 1) Then
    ISmartTagAction_SmartTagName = "značka_01#MojeZnačka"
  End If
End Property

Private Property Get ISmartTagAction_SmartTagCaption _
(ByVal SmartTagID As Long, ByVal LocaleID As Long) As String
  ISmartTagAction_SmartTagCaption = "MojeZnačka"
End Property
```

6. Pokračujeme implementací vlastností **VerbCount**, **VerbID**, **VerbCaptionFromID** a **VerbNameFromID**. Více informací o těchto vlastnostech nabízí tab. 5.

Název vlastnosti	Charakteristika
VerbCount	Říká, kolik akcí podporuje implementovaný typ inteligentní značky.
VerbID	Vlastnost vrací jedinečný identifikátor (ID) cílové akce inteligentní značky.
VerbCaptionFromID	Specifikuje uživatelsky přívětivý textový řetězec, který popisuje název akce a jenž se objeví v kontextové nabídce s dostupnými akcemi.
VerbNameFromID	Textový identifikátor akce, který je používán interně cílovou aplikací.

Tab. 5 – Bližší pohled na vlastnosti **VerbCount**, **VerbID**, **VerbCaptionFromID** a **VerbNameFromID**

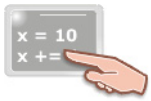
7. Technická stránka implementace vlastností je následovní:




```
Private Property Get ISmartTagAction_VerbCount _
(ByVal SmartTagName As String) As Long
    If (SmartTagName = "značka_01#MojeZnačka") Then
    ISmartTagAction_VerbCount = 1
  End If
End Property

Private Property Get ISmartTagAction_VerbID _
(ByVal SmartTagName As String, ByVal VerbIndex As Long) As Long
  ISmartTagAction_VerbID = VerbIndex
End Property

Private Property Get ISmartTagAction_VerbCaptionFromID _
(ByVal VerbID As Long, ByVal ApplicationName As String, _
ByVal LocaleID As Long) As String
  If (VerbID = 1) Then
    ISmartTagAction_VerbCaptionFromID = "Otevřít internetový prohlížeč"
  End If
End Property

Private Property Get ISmartTagAction_VerbNameFromID _
(ByVal VerbID As Long) As String
   If (VerbID = 1) Then
        ISmartTagAction_VerbNameFromID = "otevritProhlizec"
   End If
End Property
```

8. Finálním krokem je implementace metody **InvokeVerb**, která má na starosti exekuci akce v okamžiku, kdy uživatel klikne na název této akce v kontextové nabídce akcí. Jenom na připomenutí, uživatelsky přívětivý název akce vrací vlastnost **VerbCaptionFromID**. Název naší

akce je **Otevřít internetový prohlížeč**, a proto do těla metody **InvokeVerb** zapíšeme kód pro aktivaci prohlížeče:




```
Public Sub ISmartTagAction_InvokeVerb _
(ByVal VerbID As Long, ByVal ApplicationName As String, _
ByVal Target As Object, _
ByVal Properties As SmartTagLib.ISmartTagProperties, _
ByVal Text As String, ByVal XML As String)
    Dim Prohlížeč
    If (VerbID = 1) Then
        Set Prohlížeč = CreateObject("InternetExplorer.Application")
        Prohlížeč.Navigate2 ("www.pocitacovaencyklopedie.cz")
        Prohlížeč.Visible = True
    End If
End Sub
```

Kompilace kódu inteligentní značky

Kompilaci projektu s kódem inteligentní značky uskutečníte výběrem příkazu **Make MojeZnačka.dll** z nabídky **File**. Vydáte-li tento pokyn, bude projektu zkompilován, přičemž výsledkem kompilace bude knihovna ActiveX DLL s názvem **MojeZnačka.dll**. Visual Basic 6.0 vygeneruje také jedinečné identifikátory (**CLSID**) pro identifikátor (**recognizer**) a akci (**action**) inteligentní značky. Tyto identifikátory jsou zapsány do registru operačního systému. Aby Visual Basic 6.0 generoval vždy stejné identifikátory pro námi vytvořenou inteligentní značku, ujistěte se, že používáte volbu **Project Compatibility**. Vyberte nabídku **Project** a klepněte na položku **Project Properties**. V dialogovém okně označte záložku **Component** a podívejte se do rámečku **Version Compatibility**. Není-li nastavena volba **Project Compatibility**, nastavte ji. Tak docílíte toho, že Visual Basic 6.0 bude pro vytvořenou inteligentní značku generovat stále stejné **CLSID** identifikátory (a to i tehdy, uskutečníte-li v programovém kódu projektu nějaké změny). Nakonec uložte projekt a zavřete integrované vývojové prostředí Visual Basicu.

Registrace inteligentní značky v operačním systému

Předtím, než se vrhneme na testování funkčnosti naší nové inteligentní značky, musíme realizovat registraci značky v operačním systému. Proces registrace je složen ze dvou parciálních procesů:

- Registrace akce (**action**) inteligentní značky
- Registrace identifikátoru (**recognizer**) inteligentní značky

Registrace akce (action) inteligentní značky

1. Spusťte aplikaci **Regedit**.
2. V sekci **HKEY_CLASSES_ROOT** vyhledejte klíč s názvem **MojeZnačka.CAkce**.
3. Když klepněte na symbol plus (+), jenž se nachází nalevo od názvu klíče (**MojeZnačka.CAkce**), klíč se rozvine a zpřístupní se vám podklíč **Clsid**.
4. Označte podklíč **Clsid**. V pravém poli se objeví položka s názvem **(Default)**.
5. Klepněte na položku **(Default)** pravým tlačítkem myši a z kontextové nabídky vyberte příkaz **Modify**.
6. V dialogovém okně **Edit String** si všimněte textové pole **Value data**. Číselně-řetězcová konstanta CLSID je implicitně vybrána do bloku. Klepněte na tuto konstantu pravým tlačítkem myši a vyberte příkaz **Copy**.
7. Dialogové okno **Edit String** uzavřete klepnutím na tlačítko **Cancel**.

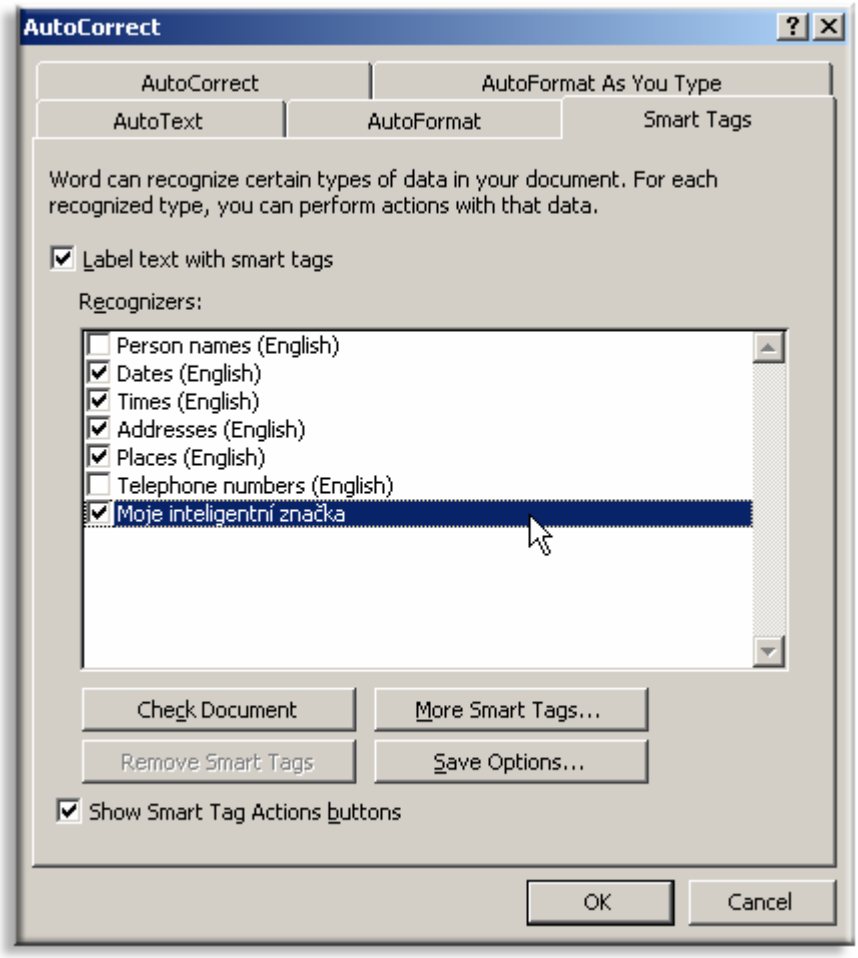
8. V sekci **HKEY_CURRENT_USER\Software\Microsoft\Office\Common\Smart Tag** vyhledejte klíč s názvem **Actions**.
9. Klikněte na symbol plus (+), jenž se nachází nalevo od klíče **Actions**. Klíč **Actions** se otevře a nabídne vám seznam vnořených podklíčů.
10. Klepněte na klíč **Actions** pravým tlačítkem myši, z kontextového menu vyberte nabídku **New** a klepněte na položku **Key**.
11. Okamžitě bude vytvořen nový klíč, jehož název bude vybrán do bloku. Klepněte na text v bloku pravým tlačítkem myši a z kontextové nabídky vyberte příkaz **Paste**. Název klíče bude vyplněn jedinečným identifikátorem (**CLSID**), jenž přináleží hodnotě klíče **MojeZnačka.CAkce** ze sekce **HKEY_CLASSES_ROOT**.

Registrace identifikátoru (recognizer) inteligentní značky

1. V sekci **HKEY_CLASSES_ROOT** vyhledejte klíč s názvem **MojeZnačka.CIdentifikátor**.
2. Zobrazte podklíč **Clsid** a hodnotu položky **(Default)** zkopírujte pomocí systémové schránky.
3. V sekci **HKEY_CURRENT_USER\Software\Microsoft\Office\Common\Smart Tag** najděte klíč s názvem **Recognizers**.
4. Zobrazte všechny podklíče klíče **Recognizers**.
5. Klepněte na klíč **Recognizers** pravým tlačítkem myši a přidejte nový podklíč (**New→Key**).
6. Do názvu klíče vložte zkopírovanou hodnotu položky **(Default)** podklíče **Clsid** klíče **MojeZnačka.CIdentifikátor**.

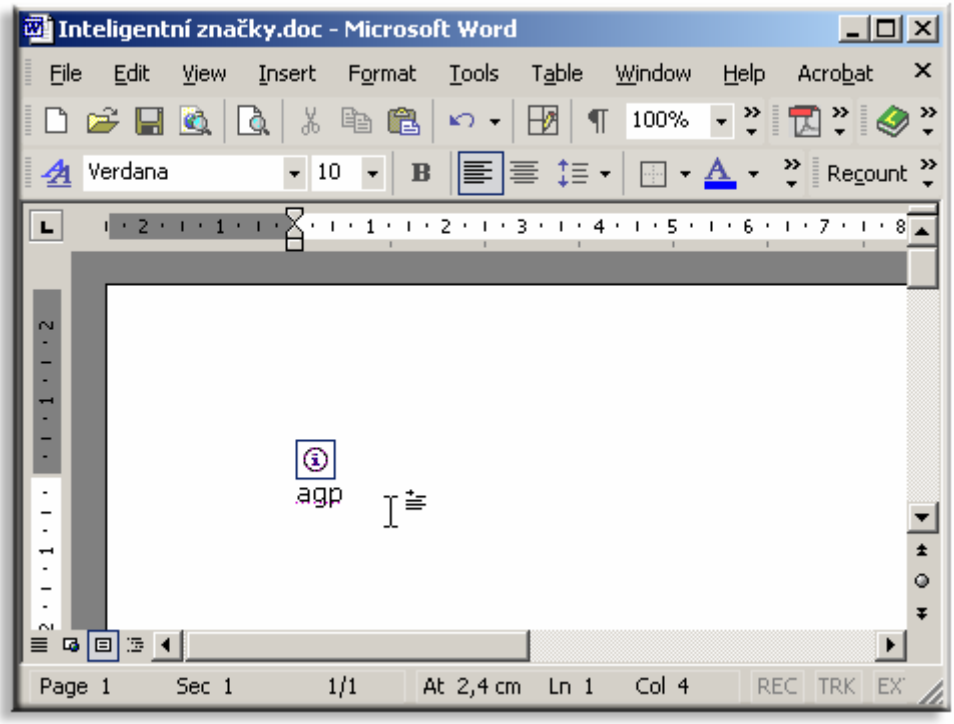
Testování inteligentní značky

Ihned poté, co byla inteligentní značka úspěšně zaregistrována, můžete přikročit k testovací fázi procesu. Spusťte aplikaci Microsoft Word XP (2002), vyberte nabídku **Tools** a klepněte na položku **AutoCorrect Options**. V dialogovém okně vyberte záložku **Smart Tags** (obr. 5).



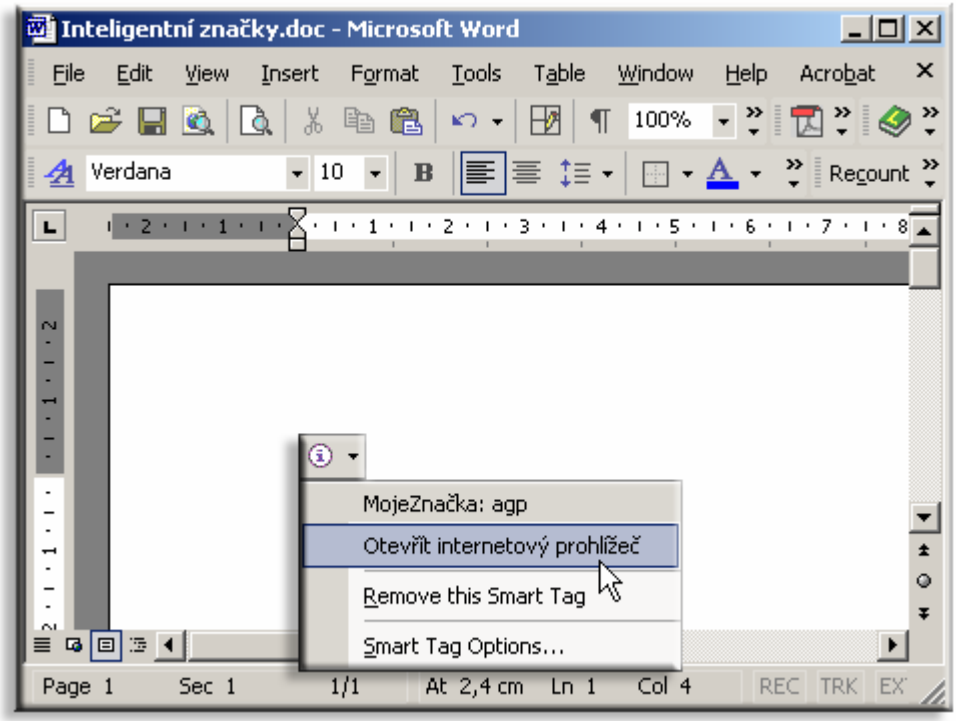
Obr. 5 – Dialogové okno **AutoCorrect** s aktivovanou záložkou **Smart Tags**

V seznamu **Recognizers** by se měla nacházet naše inteligentní značka (**Moje inteligentní značka**). Pokud se v seznamu položka s názvem naší značky neobjevuje, pravděpodobně inteligentní značka nebyla úspěšně zaregistrovaná v operačním systému. Je-li tomu tak, vraťte se do části, jež pojednává o registraci inteligentní značky. Pokud seznam obsahuje naší inteligentní značku, je vše v naprostém pořádku, a tedy můžeme začat s testováním značky. Zavřete dialog **AutoCorrect** a napište do dokumentu textový řetězec „agp" (uvozovky můžete vynechat) a stiskněte mezerník nebo klávesu Enter. Textový řetězec bude opatřen tečkovaným podtržením fialové barvy, ovšem je možné, že toto formátování neuvidíte, protože Word XP textový řetězec „agp" nezná, a proto jej bude formátovat pomocí červené vlnovky. Tak se stane, že červená vlnovky jednoduše „překryje" tečkované podtržení. Jestliže najedete na citlivý textový řetězec kurzorem myši, objeví se akční tlačítko (obr. 6).



Obr. 6 – Citlivý text a akční tlačítko, symbolizující přítomnost inteligentní značky

Akční tlačítko symbolizuje, že text pod ním byl rozeznán identifikátorem naší inteligentní značky. Najeďte na toto tlačítko myší a klepněte na něj. Uvidíte kontextovou nabídku s akcemi inteligentní značky (v našem případě se objeví pouze jedna akce s názvem **Otevřít internetový prohlížeč**). Situaci znázorňuje obr. 7.



Obr. 7 – Inteligentní značka v akci

Pro spuštění internetového prohlížeče klepněte na příslušnou akci inteligentní značky.

Integrované prostředí rubriky Visual Basic	Autor: **Ján Hanák**

Začínáme s VB .NET

Úvod do světa .NET (12. díl)

Časová náročnost (min):	Začátečník	Pokročilý	Profesionál
40			

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

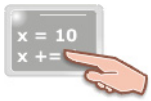
Vážení čtenáři,

dvanáctá lekce o programování ve Visual Basicu .NET pro začátečníky již na vás netrpělivě čeká. Náplní jsou i nadále operátory: Nejprve si vysvětlíme speciální porovnávací operátory **Is** a **Like**, a poté plynule přejdeme na další kategorie operátorů. Přeji vám příjemné počtení.

Obsah

Porovnávací operátor Is

Operátor **Is** se používá pro zjištění, zdali obsahují dvě různé referenční proměnné ukazatele na stejný objekt. Jestliže je výsledek porovnání kladný, operátor vrací hodnotu **True**, v opačném případě **False**. Budete-li například chtít zjistit, zdali dvě referenční proměnné obsahují odkaz na aktuální instanci formuláře, můžete použít následující fragment programového kódu:




```
Dim refA, refB As Form
refA = Me
refB = refA
Me.Text = CType((refB Is refA), String)
```

Obě odkazové proměnné (**refA**, **refB**) mohou uchovávat reference na instance třídy **Form**. Do proměnné **refA** je uložen odkaz na aktuální instanci třídy **Form** (pomocí klíčového slova **Me**). Přiřazovací příkaz způsobí kopírování reference, která je uložena v proměnné **refA**, do proměnné **refB**.

Připomeňme si, jak pracuje operátor přiřazení při hodnotových a referenčních proměnných:

1. Jestliže použijeme přiřazovací operátor na dvě hodnotové proměnné, bude hodnota proměnné, která je umístěna napravo od přiřazovacího operátoru, zkopírována do proměnné, která se nachází na levé straně přiřazovacího operátoru. Při tomto typu kopírovacího procesu dochází ke kopírování hodnoty jedné proměnné do druhé proměnné. Zkopírovaná hodnota je přitom zcela nezávislá od svého originálu.

2. Operátor pro přiřazení se ve společnosti referenčních proměnných chová zcela jinak. Jak jsme si již řekli, odkazové proměnné jsou naplněné referencemi, které ukazují zvyčejně na objekty, tedy instance tříd. Předmětem kopírování se v tomto případě stává hodnota referenční proměnné, tedy samotná reference. Detailněji řečeno, reference, která je uložena v proměnné, jež se nachází na pravé straně přiřazovacího příkazu, bude zkopírována do odkazové proměnné, nacházející se na levé straně operátoru pro přiřazení. Výsledkem celého procesu je skutečnost, že obě referenční proměnné obsahují stejné reference na jednu (cílovou) instanci třídy. Změníme-li stav objektu pomocí jedné referenční proměnné, tato změna se projeví i při použití druhé odkazové proměnné. Zde je důležité si uvědomit, že ačkoliv používáme dvě různé referenční proměnné, obě dvě jsou nasměrované na stejný objekt.

Pokud se rovnají reference obou odkazových proměnných, bude vrácena operátorem **Is** hodnota **True** (jako v tomto případě). Používáte-li při programování volbu **Option Strict** nastavenou na hodnotu **True**, budete rovněž muset explicitně přetypovat návratovou hodnotu operátoru **Is** z typu **Boolean** na typ **String** (například pomocí konverzní funkce **CType**, jak je uvedeno ve výše představené ukázce kódu).

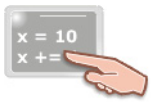
Porovnávací operátor Like

Operátor **Like** se používá pro speciální porovnání textových řetězců, resp. porovnání textových znaků v textových řetězcích. Generická podoba použití operátoru **Like** je tato:

Výsledek = TextovýŘetězec Like šablona

Výsledkem je pravdivostní hodnota, která identifikuje, zdali **TextovýŘetězec** splňuje kritéria **šablony**. Pokud je porovnání vyhovující, je do proměnné **Výsledek** uložena hodnota **True**, jinak bude proměnná **Výsledek** obsahovat hodnotu **False**.

Prostřednictvím operátoru **Like** můžeme například zkoumat, zdali se požadovaný znak nachází v jisté skupině znaků:




```
Dim textA As String = "C"
Dim textB As String = "[A-F]"
Me.Text = CStr(textA Like textB)
```

Operátor **Like** testuje, jestli se znak C nachází v posloupnosti znaků A až F. Ano, tato podmínka je splněna, a proto se v titulku dialogového okna objeví hodnota **True**. Šablonou mohou být různé zástupné znaky, přičemž jejich popis můžete najít v dokumentaci k Visual Basicu .NET. Použití operátoru **Like** se váže na způsob, jakým Visual Basic .NET přistupuje k porovnávaní textových řetězců. Práce s řetězci je založená na volbě **Option Compare**.

Budete-li experimentovat s nastaveními volby **Option Compare**, mějte na paměti, že kód volby musí být umístěn před veškerý kód třídy, resp. modulu (podobně, jako tomu je u voleb **Option Strict** a **Option Explicit**).

Tato volba může nabývat pouze dvou hodnotu:

1. **Option Compare Binary** – determinuje, že výsledek porovnávání řetězců je založen na binární charakteristice jednotlivých znaků. Znamená to, že při porovnávaní znaků textového řetězce se Visual Basic .NET řídí binární podobou těchto řetězců. Operační systémy Windows provádějí řazení znaků pomocí kódové stránky. Jestliže je použita volba **Option Compare Binary**, velké písmena abecedy jsou řazena před své odpovídající malé protějšky (při porovnání tedy sehrává roli velikost znaků). Například písmeno A je řazeno před písmeno a. Jestliže neuvedete jinak, je při porovnávání řetězců implicitně použita volba **Option Compare Binary**.

2. **Option Compare Text** – nařizuje porovnání, v rámci kterého Visual Basic .NET nenahlíží na velikost znaků, ovšem systém porovnávání je založen na místní konfiguraci operačního systému. Navážeme-li na předchozí příklad s varianty písmena A, můžeme prohlásit, že při použití volby **Option Compare Text** se velké písmeno rovná své malé alternativě, tedy A = a.

Přiřazovací operátory

Přiřazovací operátory provádějí přesně to, co vyplývá z jejich názvu, tedy přiřazují jisté hodnoty, nejčastěji z jedné proměnné do jiné proměnné. Obecná podoba použití přiřazovacích operátorů je následovní:

Proměnná = hodnota

Proměnnou může být jakákoliv platně deklarovaná proměnná, ovšem je třeba brát v úvahu datový typ této proměnné. Datový typ proměnné je užitečný pro vymezení intervalu cílových hodnot, které lze do proměnné umístit. Pochopitelně, specifikace datového typu proměnné by se měla odvíjet od charakteru hodnot, které budeme chtít do proměnné uložit. Proměnné typu **Integer** dokáží ukládat celá čísla, zatímco typ **String** je výhodné použít při práci s textovými řetězci. Ve skutečnosti je Visual Basic .NET poněkud méně náchylný na nesoulad datových typů proměnných a typů hodnot, které lze do těchto proměnných uložit. Pokud se hodnota, kterou chcete do proměnné uložit, neshoduje s datovým typem proměnné, Visual Basic .NET se bude snažit tento nesoulad „zahladit", a to pomocí konverze typu hodnoty. Máme-li deklarovanou proměnnou typu **String** a uložíme do ní čelnou hodnotu, nebude hlášena žádná chyba ani v režimu návrhu aplikace, ani při jejím běhu. Visual Basic .NET je natolik inteligentní, že nezbytné konverzní operace provede automaticky sám. Připomeňme, že implicitní konverze jsou realizovány jenom v případě, jestliže nemůže při konverzním procesu dojít ke ztrátě dat a jestliže nepoužíváte při programování volbu **Option Strict** nastavenou na hodnotu **True**. V opačném případě se ke slovu dostávají explicitní konverze.

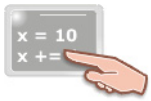
Visual Basic .NET nabízí spoustu přiřazovacích operátorů, přičemž novinkou jsou zkrácené tvary některých operátorů. V tab. 1 můžete vidět ukázky použití přiřazovacích operátorů.

Přiřazovací operátor	**Ukázka použití**	**Ekvivalentní zápis**
Operátor přiřazení (=)	Dim x As Short = 20	Dim x As Short = 20
Zkrácený operátor pro sčítání a přiřazení (+=)	x += 1	x = x + 1
Zkrácený operátor pro odečítání a přiřazení (-=)	x -= 1	x = x – 1
Zkrácený operátor pro násobení a přiřazení (*=)	x *= y	x = x * y

Přiřazovací operátor	Ukázka použití	Ekvivalentní zápis
Zkrácený operátor pro dělení a přiřazení (/=)	x /= z	x = x / z
Zkrácený operátor pro celočíselné dělení a přiřazení (\=)	a \= b	a = a \ b
Zkrácený operátor pro umocňování a přiřazení (^=)	a ^= b	a = a ^ b
Zkrácený operátor pro zřetězení a přiřazení (&=)	a &= b	a = a & b

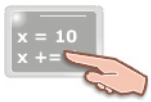
Tab. 1 – Přehled přiřazovacích operátorů

Podívejme se teď na způsob práce přiřazovacích operátorů blíže. Vezměme si třeba zkrácený operátor pro sčítání a přiřazení a prozkoumejme, jak ve skutečnosti pracuje:




```
Dim a1 As Short, a2 As Integer
a1 = Math.Abs(CShort((-2) * 2))
a2 = 10
a2 += a1
Me.Text = CType(a2, String)
```

Kód pracuje s dvěma proměnnými: **a1** typu **Short** a **a2** typu **Integer**. Do proměnné **a1** je přiřazena absolutní hodnota výrazu **CShort((-2)*2))**, tedy 4. Běh kódu pokračuje naplněním proměnné **a2** celočíselnou hodnotou 10. My se však budeme soustředit až na samotnou aplikaci zkráceného přiřazovacího operátoru +=:




```
a2 += a1
```

Přiřazovací operátor vykonává svoji práci podle tohoto algoritmu:

1. Zjistí hodnotu proměnné, která se nachází na pravé straně přiřazovacího příkazu (**a1**). Inicializovaná proměnná **a1** obsahuje hodnotu 4 a právě tato hodnota bude získána.
2. Hodnota proměnné **a1** bude přičtena k proměnné, která se nachází na levé straně přiřazovacího příkazu (**a2**).
3. Součet hodnot proměnných **a1** a **a2** bude přiřazen do proměnné, stojící na levé straně přiřazovacího příkazu (**a2**). Výsledkem činnosti operátoru pro sčítání a přiřazení je tedy uložení hodnoty 14 do proměnné **a2**.

Protože proměnné **a1** a **a2** jsou zástupci různých datových typů, bude ještě před přiřazením finální hodnoty provedeno přetypování hodnoty proměnné **a1** do podoby typu **Integer**.

Logické operátory

Logické operátory se používají pro nalezení pravdivostní hodnoty logických výrazů (logický výraz je posloupnost logických operandů a operátorů). Logickým výrazem je každý výraz, jehož hodnotu lze vyjádřit pomocí logických hodnot **True** nebo **False**. Logické operátory společně s logickými výrazy se v programování velmi často používají v případech, kdy je potřebné zjistit, zdali je jistá podmínka

platná či nikoliv. Přestože je oblast působnosti logických operátorů značně rozsáhlá, nejvíc se s nimi budete setkávat nejspíš při rozhodovacích konstrukcích **If** a při cyklech.

Visual Basic .NET nabízí následující logické operátory: **And**, **Or**, **Xor**, **Not**, **AndAlso** a **OrElse**.

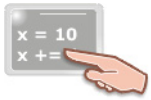
Přehled logických operátorů **And**, **Or**, **Xor** a **Not**, společně s ukázkou jejich aplikace na testovací operandy, přináší tab. 2.

Logický operátor	Operand1	Operand2	Použití logického operátoru	Finální pravdivostní hodnota
And	**A = True**	**B = True**	**A And B**	**True**
	A = True	**B = False**		**False**
	A = False	**B = True**		**False**
	A = False	**B = False**		**False**
Or	**A = True**	**B = True**	**A Or B**	**True**
	A = True	**B = False**		**True**
	A = False	**B = True**		**True**
	A = False	**B = False**		**False**
Xor	**A = True**	**B = True**	**A Xor B**	**False**
	A = True	**B = False**		**True**
	A = False	**B = True**		**True**
	A = False	**B = False**		**False**
Not	**A = True**		**Not A**	**False**
		B = False	**Not B**	**True**

Vysvětlení aplikace logických operátorů And, Or, Xor a Not	
And	Operátor **And** vrací hodnotu **True** právě tehdy, obsahují-li oba operandy hodnotu **True**. V opačném případě je vrácená hodnota **False**.
Or	Operátor **Or** vrací hodnotu **True**, je-li alespoň jeden z operandu pravdivý (**True**). To znamená, že hodnota **True** je vrácená vždy, jenom ne tehdy, kdy oba operandy obsahují hodnotu **False**.
Xor	Operátor **Xor** vrací hodnotu **True**, pokud právě jeden operand obsahuje hodnotu **True**. Tato hodnota (**True**) je vrácena v případě, kdy oba operandy disponují různou pravdivostní hodnotou.
Not	Operátor **Not** provádí logickou negaci, tedy převrácení původní pravdivostní hodnoty operandu. Je-li operand pravdivý (**True**), po aplikace operátoru **Not** bude jeho hodnota **False**. Podobně probíhá i zpětný proces (negace již znegované pravdivostní hodnoty).

Tab. 2 – Přehled aplikace logických operátorů **And**, **Or**, **Xor** a **Not**

Nyní se podívejme na programovou ukázku použití logických operátorů **And**, **Or**, **Xor** a **Not**:




```
Dim x1, x2, x3, x4 As Boolean
x1 = True
x2 = Not x1
x3 = x1 And x2
x4 = (x1 Or x2) Xor x3
Me.Text = CType(x4, String)
```

Tento výpis programového kódu vypíše do titulku aktuální instance třídy **Form** textový řetězec **True**. Jak Visual Basic .NET dospěl k takovému výsledku? Inu, asi takto:

Krok	**Komentář k práci programového kódu**	**Aktuální hodnoty proměnných**			
		x1	**x2**	**x3**	**x4**
1.	Do proměnné **x1** je uložena hodnota **True**.	**True**	False *	False *	False *
2.	Hodnota proměnné **x1** je znegována pomocí operátoru **Not**, takže do proměnné **x2** je uložena hodnota **False**.	True	**False**	False *	False *
3.	Na proměnné **x1** a **x2** je aplikován logický operátor **And**. Jelikož pravdivostní hodnota obou proměnných není **True**, operátor vrací hodnotu **False**.	True	False	**False**	False *
4.	Mezi hodnotami proměnných **x1** a **x2** je realizovaná logická disjunkce pomocí operátoru **Or**. Jelikož proměnná **x1** obsahuje hodnotu **True**, operátor vrací hodnotu **True**. Na získanou hodnotu **True** a hodnotu proměnné **x3** (**False**) je použit operátor **Xor**. Tento operátor vrací hodnotu **True** tehdy, obsahuje-li právě jeden operand hodnotu **True**. Tato podmínka je splněna, a proto je do proměnné **x4** uložena pravdivostní hodnota **True**.	True	False	False	**True**

Legenda	
*	Proměnná je implicitně inicializovaná na hodnotu **False**.
	Finální hodnota zkoumané proměnné v daném kroku algoritmu

Integrované prostředí rubriky Visual Basic	Autor: Ján Hanák

Téma

VB .NET

Programátorská laboratoř

Prostor pro experimentování

Časová náročnost (min):	Začátečník	Pokročilý	Profesionál
70	1	2	3

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

VB .NET	Použití sdílených datových členů, vlastností a metod	3	0:30
VB .NET	Vytváření miniatur obrázků (thumbnails)	2	0:20
VB .NET	Deklarování a generování událostí	3	0:20

Použití sdílených datových členů, vlastností a metod

Zájemci o studium objektově orientovaného programování ve Visual Basicu .NET často narážejí na potíže ve chvíli, kdy se jich zeptáte, jaký je rozdíl mezi instančními datovými členy, vlastnostmi a metodami a jejich sdílenými protějšky. Zkušenější jedinci sice dokáží vysvětlit význam a použití instančních datových položek na přijatelné úrovni, ovšem oříškem takřka vždycky bývá charakteristika sdílených datových členů. Abychom si udělali v nastíněné situaci pořádek, zaměříme se na popis jak instančních datových položek, tak i jejich sdílených kolegů.

Jistě víte, že když použijete správnou syntaxi příkazu s klíčovým slovem **New** a názvem třídy, vytvoříte instanci dané třídy. Tato nově vytvořená instance je uložena do generace 0 řízené hromady počítače.

Řízená hromada je oblast počítačové paměti, do které jsou ukládány takzvané řízené instance tříd. Řízená hromada je kontrolována prostřednictvím společného běhového prostředí (**Common Language Runtime**, **CLR**). **CLR** za účelem kontroly a správy řízené hromady využívá služeb automatické správy paměti s názvem **Garbage Collection**. Ačkoliv nebudeme zacházet příliš do hloubky, měli byste vědět, že instance, neboli objekty, jsou řazeny do tří generací (generace 0, 1 a 2). Nejmladší, resp. posledně vytvořené objekty, jsou vždy umísťovány do nulté generace.

Většina instancí obsahuje svá vlastní data, která jsou uložena v privátních datových členech každé instance. Instance se svými daty pracují pomocí programovacích konstrukcí, mezi něž patří vlastnosti a metody. Každá vytvořená instance obsahuje svou vlastní kopii všech implementovaných datových členů, což znamená, že pro každou instanci a její datové položky je v paměti vytvořen separátní prostor, ve kterém jsou uložena citlivá data dané instance. Tímto způsobem mohou různé instance existovat a pracovat nezávisle na sobě. Datové členy, které patří instanci, se nazývají instanční datové členy. Analogicky, programovací konstrukce, které přistupují k instančním datovým

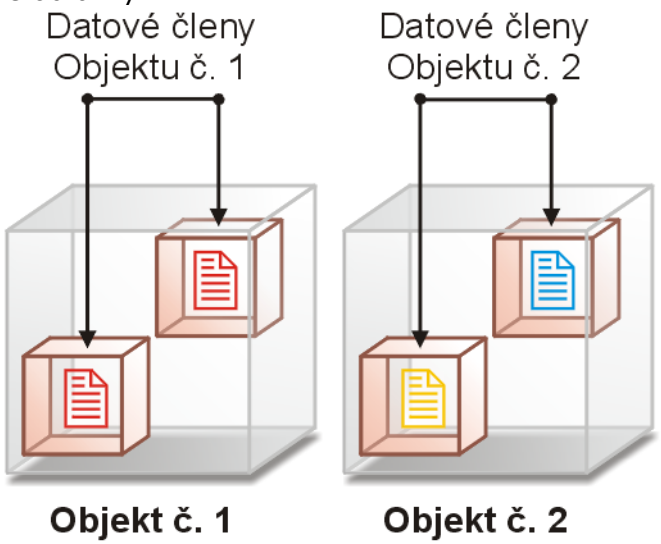
členům, tedy vlastnosti a metody, jsou známé jako instanční vlastnosti a metody. Pro instanční datové členy, vlastnosti a metody je charakteristický jejich vztah k dané instanci. Jednoduše, instanční metody mohou provádět operace jenom nad daty konkrétní instance. Na druhé straně existují sdílené datové členy a také sdílené vlastnosti a metody.

V některých programovacích jazycích, třeba v C# a v řízeném C++, jsou datové členy tohoto typu označovány jako statické. Pokud tedy někdy uslyšíte své kolegy pracující v C# mluvit o statických metodách, budete vědět, že jejich diskuse se točí kolem sdílených metod.

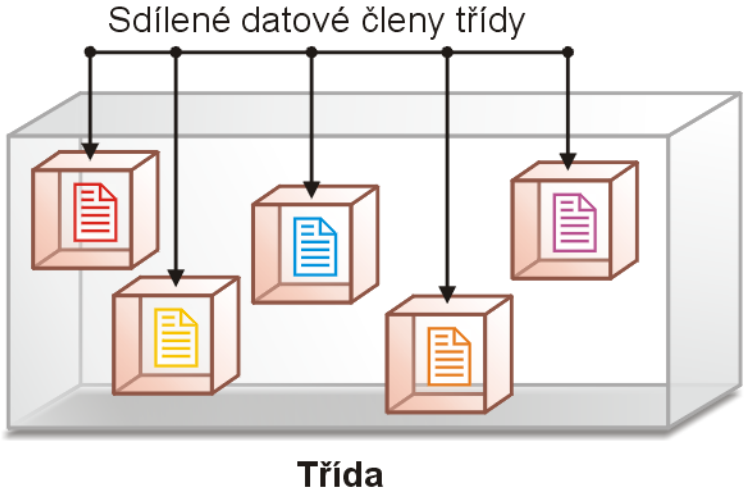
Sdílené datové členy se vztahují na třídu, spíše než na jednotlivé instance třídy. Sdílené datové členy jsou ve Visual Basicu .NET deklarovány pomocí klíčového slova **Shared**. Pro všechny sdílené datové členy je na řízené hromadě vytvořen souvislý prostor bajtů, v němž jsou tyto sdílené datové členy uloženy. Aby bylo možné k sdíleným datovým členům přistupovat, musejí existovat také sdílené vlastnosti a metody. Tyto entity operují se sdílenými členy, tedy daty třídy.

Je důležité pochopit, že sdílené vlastnosti a metody mohou pracovat jenom se sdílenými datovými členy třídy, nikoliv s datovými členy instancí dané třídy.

Abyste lépe pochopili rozdíl mezi instančními datovými členy a sdílenými datovými členy, prostudujte si níže uvedené obrázky.



Obr. 1 – Grafická ilustrace dvou objektů a jejich datových členů

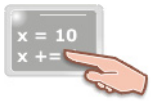


Obr. 2 – Grafická ilustrace třídy a sdílených datových členů

Na následujících řádcích si předvedeme dvě programové ukázky, v nichž uvidíte použití sdílených datových členů, vlastností a metod.

Ukázka č. 1 – Zjištění počtu alokovaných objektů pomocí sdílených členů
V první ukázce uvidíte, jak nám mohou sdílené členy pomoci při zjišťování aktuálního stavu vytvořených objektů. Postupujte následovně:

1. Spusťte Visual Basic .NET a vytvořte standardní aplikaci pro Windows (**Windows Application**).
2. Přidejte do projektu soubor s kódem třídy (**Project → Add Class**).
3. Implicitně vytvořený programový kód třídy upravte podle uvedeného vzoru:



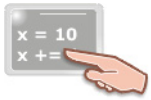

```
Public Class TřídaA
    Private Shared PočetObjektů As Integer

    Sub New()
        PočetObjektů += 1
    End Sub

    Protected Overrides Sub Finalize()
        PočetObjektů -= 1
    End Sub

    Public Shared ReadOnly Property ZjistitPočetObjektů() As Integer
        Get
            Return PočetObjektů
        End Get
    End Property
End Class
```

Jedná se o třídu s názvem **TřídaA**. V těle třídy se nachází jeden sdílený datový člen typu **Integer** s názvem **PočetObjektů**. Všimněte si, že před názvem datového členů je umístěno klíčové slovo **Shared**. Tím je výslovně naznačeno, že pracujeme se sdíleným datovým členem, jenž se vztahuje ke třídě jako celku.

Sdílený datový člen **PočetObjektů** je deklarován rovněž použitím klíčového slova **Private**. Jde tedy o soukromý sdílený datový člen, což je vhodné, protože takto dodržujeme princip skrývání dat. Princip skrývání dat představuje jeden ze základních pilířů koncepce OOP. Jednoduše řečeno, význam tohoto principu spočívá v tom, že nikdy neposkytneme klientskému kódu přímý přístup k datovému členu třídy. Místo toho hodnotu příslušného datového členu zjistíme pomocí implementované veřejné sdílené vlastnosti.

Filozofie ukázky je zcela přímočará: Vždy, když dojde k vytvoření instance třídy **TřídaA**, bude v konstruktoru třídy inkrementován sdílený datový člen **PočetObjektů**. Datový člen **PočetObjektů** je implicitně inicializován na nulovou hodnotu. Po vytvoření každé instance bude hodnota sdíleného členu **PočetObjektů** zvýšena o jednotku. Před likvidací instance je proveden kód, jenž je umístěn v chráněné a překryté Sub proceduře s názvem **Finalize**. Jelikož je daná instance určena k likvidaci, je nutné v proceduře **Finalize** dekrementovat hodnotu sdíleného datového členu **PočetObjektů**. Aktuální hodnotu sdíleného datového členu **PočetObjektů** nám na požádání nabídne sdílená vlastnost **ZjistitPočetObjektů**. Návratová hodnota sdílené vlastnosti je typu **Integer** a odpovídá skutečnému počtu instancí dané třídy, které se ještě nacházejí na řízené hromadě.

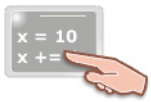
4. Na formulář přidejte jednu instanci ovládacího prvku **Button** a také jednu instanci komponenty **Timer**.
5. Ujistěte se, že vlastnost **Enabled** instance komponenty **Timer** je nastavena na hodnotu **False** a že vlastnost **Interval** této instance obsahuje hodnotu 100 (milisekund).
6. Zpracovatele události **Click** instance ovládacího prvku **Button** upravte následovně:



```
Private Sub Button1_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles Button1.Click
    Timer1.Enabled = True
    Dim i As Integer
    Dim Objekt As Object
    For i = 1 To 1000
        Objekt = New TřídaA()
    Next
End Sub
```

Tento kód vytvoří 1000 instancí třídy **TřídaA**.

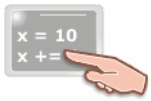
7. Obsluhu události **Tick** instance komponenty **Timer** vyplňte tímto kódem:



```
Private Sub Timer1_Tick(ByVal sender As System.Object, _
ByVal e As System.EventArgs)
    Me.Text = "Počet objektů: " & TřídaA.ZjistitPočetObjektů
End Sub
```

Spusťte testovací aplikaci a klepněte na tlačítko. V titulkovém pruhu dialogového okna se objeví zpráva o vytvoření 1000 objektů. Jak jsme si již řekli, všechny objekty jsou uloženy do generace 0 řízené hromady. Objekty však po svém vytvoření nejsou nijak využívány, a proto se za krátkou dobu stanou soustem pro automatickou správu paměti. Aktuální počet alokovaných objektů se zobrazuje v titulkovém pruhu dialogu přibližně každých 100 milisekund. Jelikož je generace 0 řízené hromady vyhrazena pro dočasné objekty, za krátkou dobu budou všechny vytvořené instance z paměti odstraněny (titulek dialogového okna bude ukazovat nulovou hodnotu).

Ukázka č. 2 – Použití sdílené funkce třídy pro přidání instance třídy Button na aktivní formulář
Zajisté jste se již někdy střetli s třídou, která přímo neumožňovala tvorbu svých instancí. U takovéto třídy tedy není možné použít známou syntaxi s klauzulí **New**. Ačkoliv třída neposkytuje možnost přímého vytváření instancí, přesto může být užitečná. Ano, klíčem k úspěchu je zařazení sdílené funkce (resp. metody). Vložte do stávajícího projektu Visual Basicu .NET další třídu a její podobu upravte takto:



```
Public Class TřídaB
    Public Shared Function PřidatTlačítko _
    (ByVal Jméno As String, ByVal Návěstí As String, _
    ByVal Šířka As Integer, ByVal Výška As Integer, _
    ByVal Pozice As Point) As Boolean
        Try
            Dim Tlačítko As New Button()
            With Tlačítko
                .Name = Jméno
                .Text = Návěstí
                .Width = Šířka
                .Height = Výška
                .Location = Pozice
            End With
            Form.ActiveForm.Controls.Add(Tlačítko)
        Catch x As System.Exception
```

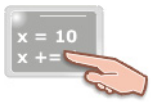
Programový kód pokračuje na následující straně

```
                Throw New System.Exception("Vytváření tlačítka se nezdařilo.")
            End Try
            Return True
        End Function
    End Class
```

Předmětem našeho zájmu je sdílená funkce **PřidatTlačítko**. Signatura této sdílené funkce je značka rozsáhlá, pozůstává s množství parametrů, které specifikují:

- Název tlačítka (**Jméno As String**)
- Textový řetězec, jenž se bude objevovat na tlačítku (**Návěstí As String**)
- Šířka regionu tlačítka (**Šířka As Integer**)
- Výška regionu tlačítka (**Výška As Integer**)
- Pozice tlačítka uvnitř kontejnerového ovládacího prvku, tedy formuláře (**Pozice As Point**)

Přidejte na formulář další instanci ovládacího prvku **Button** a do zpracovatele události **Click** této instance umístěte tento programový kód:

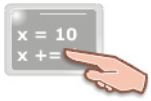



```
Private Sub Button2_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles Button2.Click
    Me.Text = CStr(TřídaB.PřidatTlačítko _
    ("btnTlačítko1", "Tlačítko", 100, 60, New Point(10, 200)))
End Sub
```

Kód volá sdílenou metodu **PřidatTlačítko** třídy s názvem **TřídaB**. Výsledkem práce sdílené metody je vytvoření nové instance třídy **Button** a začlenění této instance do kolekce instancí aktuálního formuláře (tedy formuláře, jehož programový kód je právě podroben exekuci). V našem případě bude levý roh regionu instance třídy **Button** umístěn na bodě se souřadnicemi (10, 200), instance bude mít šířku 100 pixelů a výšku 60 pixelů. Proběhnou-li všechny operace úspěšně, sdílená funkce vrátí hodnotu **True**, která bude zobrazena v titulkovém pruhu aktuálního formuláře.

Vytváření miniatur obrázků (thumbnails)

Grafický subsystém platformy .NET Framework s názvem GDI+ vám poskytuje relativně snadnou cestu pro vytváření grafické miniatury (thumbnail) jakéhokoliv obrázku, jehož data jsou uložena v jednom z nativně podporovaných grafických souborů. Budete-li chtít rychle vytvořit grafickou zmenšeninu vámi požadovaného obrázku, použijte následující programový kód:



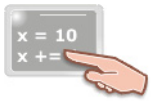

```
Dim obr As Image = Image.FromFile("c:\obr1.bmp")
Dim th As Image = obr.GetThumbnailImage(200, 150, Nothing, IntPtr.Zero)
Dim g As Graphics = Me.CreateGraphics
g.DrawImage(th, 10, 10, th.Width, th.Height)
```

Uvedený kód pracuje podle tohoto algoritmu:

1. Začínáme deklarací referenční proměnné **obr**, která může uchovávat referenci na instanci třídy **Image**.
2. Jestliže předáme sdílené metodě **FromFile** třídy **Image** plně kvalifikovanou cestu k souboru s grafickými daty, tato metoda zabezpečí vytvoření instance třídy **Image** na řízené hromadě. Vytvořená instance bude představovat grafický objekt, jehož data byla načtena ze specifikovaného souboru.
3. Pokračujeme deklarací další referenční proměnné **th**. Podobně jako proměnná **obr**, také proměnné **th** může obsahovat ukazatel na instanci třídy **Image**. Rozdíl mezi oběma

proměnnými je však v tom, že zatímco proměnná **obr** obsahuje referenci na původní grafický objekt, do proměnné **th** bude za chvíli uložen odkaz na novou instanci třídy **Image**. Tato nová instance bude obsahovat grafickou miniaturu původního obrázku.

4. Klíčem k úspěchu je nepopiratelně aktivace metody **GetThumbnailImage**, jejíž generická podoba je následovní:




```
Public Function GetThumbnailImage( _
   ByVal thumbWidth As Integer, _
   ByVal thumbHeight As Integer, _
   ByVal callback As Image.GetThumbnailImageAbort, _
   ByVal callbackData As IntPtr _
) As Image
```

Popis parametrů je uveden v tab. 1.

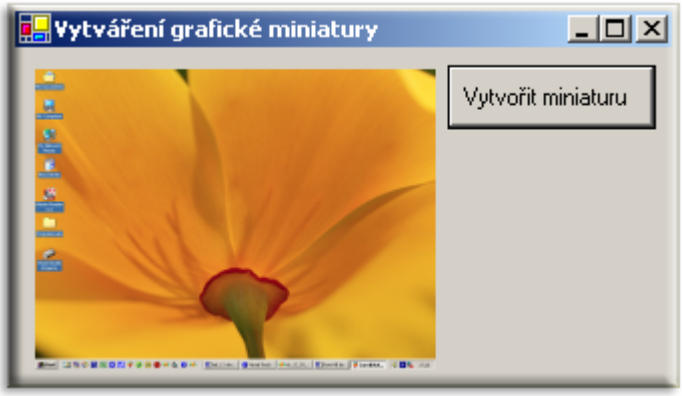
Parametr	Charakteristika
thumbWidth	Šířka miniatury grafického obrázku měřená v pixelech.
thumbHeight	Výška miniatury grafického obrázku měřená v pixelech.
	 Pokud jsou hodnoty parametrů **thumbWidth** a **thumbHeight** nulové, jsou rozměry grafické miniatury určeny na základě implicitních nastavení systému.
callback	Delegát **Image.GetThumbnailAbort**. Ve verzi GDI+ 1.0 se tento delegát nevyužívá. Ačkoliv oficiální dokumentace říká, že je nevyhnutné delegáta vytvořit, nemusí tomu být nutně tak. V praxi totiž stačí, když do parametru **callback** uložíte hodnotu **Nothing**.
callbackData	Parametr musí obsahovat hodnotu sdíleného členu **Zero** struktury **IntPtr**. Sdílený datový člen **Zero** reprezentuje ukazatel, jenž byl inicializován na nulovou hodnotu.

Tab. 1 – Charakteristika parametrů metody **GetThumbnailImage**

5. Pokud je volání metody úspěšné, finálním výsledkem je instance třídy **Image**. Tato instance představuje grafickou miniaturu, tedy modifikovaný původní obrázek.

Některé grafické formáty dovolují kromě samotného grafického obrazce uložit v jednom souboru také miniaturu tohoto grafického obrazce. Pokud metoda zjistí, že soubor disponuje i grafickou miniaturou, je načtena tato miniatura. V opačném případě je zmenšen původní grafický obrazec pomocí transformačních algoritmů.

6. Grafický objekt aktuální instance formuláře si připravíme pomocí metody **CreateGraphics** a posléze použijeme metodu **DrawImage**, která zabezpečí vykreslení grafické miniatury (obr. 3).



Obr. 3 – Vytvoření grafické miniatury použitím metody **GetThumbnailImage**

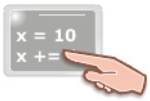
Deklarování a generování událostí

Visual Basic .NET ruku v ruce s platformou .NET nabízí vývojářům množství tříd, ovládacích prvků a komponent. Instance těchto entit žijí ve světě, jenž je řízen modelem založeným na událostech. Ano, programování řízené událostmi je dnes v kurzu, a i proto se můžete takřka na každém kroku střetnout s událostmi.

Událost můžeme definovat jakou souhru faktorů, která nastane v jistém časovém okamžiku. Událost tak říká, že nastala jistá situace. Programové entity, jako například objekty, musí vědět, jak se mají chovat v případě, kdy nastane jistá událost. Zvyčejně objekt neví, jak na vzniklou situaci smysluplně reagovat, a proto jej to musíme naučit. Tato výuka je založená na implementaci zpracovatelů příslušných událostí. Mezi událostí a jejím zpracovatelem je zřejmý vzájemný vztah: Je to událost sama, kdo iniciuje činnost zpracovatele události, přičemž však žádný zpracovatel není aktivován do okamžiku, kdy nenastane jistá událost. V tomto tipu si ukážeme, jak pracovat s událostmi a jejími zpracovateli v programovém kódu. Uvidíte, jak se událost deklaruje a také, jak ji lze vygenerovat.

Postupujte následovně:

1. Vytvořte nový projekt Visual Basicu .NET pro standardní aplikaci Windows (**Windows Application**).
2. Na formulář přidejte instanci ovládacího prvku **Button** a na vytvořenou instanci poklepejte.
3. První etapou v životním cyklu události je její deklarace. Pro deklaraci události je k dispozici příkaz **Event**:

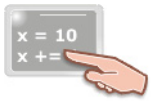



```
Public Event Událost(ByVal hwnd As Form)
```

Zde vidíte kompletní ukázku deklarace události. Před klíčovým slovem **Event** se nachází modifikátor přístupu události, který je v našem případě **Public**. Za klíčovým slovem **Event** se nachází název události a je jenom na vás, jak svoji událost pojmenujete (měli byste ovšem dodržovat pravidla pro pojmenovávání programových jednotek). Za názvem události následuje signatura, teda seznam parametrů, s kterým bude událost pracovat. Přesněji řečeno, s těmito parametry nebude pracovat přímo událost, ale její zpracovatel, jak sami za chvíli uvidíte. Parametry mohou být předávány hodnotou, nebo odkazem (zde je zvolen první způsob). Parametrická proměnná **hwnd** je schopna uchovávat ukazatel na instanci třídy **Form**.

4. Jakmile je událost deklarována, můžeme zařadit do kódu také zpracovatele této události. Je důležité vědět, že každá událost musí disponovat svým zpracovatelem. Ve dvojici událost - zpracovatel je právě zpracovatel tím aktivním prvkem, který iniciuje exekuci programových

operací v okamžiku, kdy dojde ke generování události. Kód zpracovatele události může mít třeba tuto podobu:

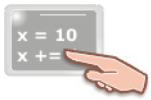


```
Public Sub Zpracovatel(ByVal hwnd As Form)
    Dim f As Integer = FreeFile()
    Try
        FileOpen(f, "d:\info.txt", OpenMode.Output, OpenAccess.Write)
        Print(f, "Handle okna: " & hwnd.Handle.ToString & _
        vbCrLf & "Titulek okna: " & hwnd.Text & _
        vbCrLf & "Neprůhlednost (%): " & (hwnd.Opacity) * 100 & _
        vbCrLf & "Výška okna (pixely): " & hwnd.Height & _
        vbCrLf & "Šířka okna (pixely): " & hwnd.Width)
        Me.Text = "Soubor s informacemi byl úspěšně vytvořen."
        FileClose(f)
    Catch x As Exception
        MessageBox.Show("Při vytváření souboru došlo k chybě.", _
        "Zpráva o chybě", MessageBoxButtons.OK, MessageBoxIcon.Error)
        Me.Text = "Soubor s informacemi nebylo možné vytvořit."
    End Try
End Sub
```

Zpracovatel události musí mít stejnou signaturu jako samotná událost.

Po formální stránce je zpracovatel události Sub procedurou, což znamená, že zpracovatel události nemůže nikdy vracet žádnou návratovou hodnotu. Věřím, že výpis kódu je vám jasný na první pohled – jednoduše jsou zjištěny informace o cílové instanci třídy **Form** a tyto informace jsou vzápětí uloženy do textového souboru na logický oddíl pevného disku.

5. Po deklaraci události a implementaci jejího zpracovatele je načase vytvořit mezi těmito dvěma entitami přepojení. Toto přepojení je velmi důležité proto, aby událost mohla najít svého zpracovatele. Z pohledu tvorby zdrojového kódu je přepojení mezi událostí a jejím zpracovatele realizováno prostřednictvím příkazu **AddHandler**:

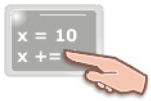


```
Private Sub Form1_Load(ByVal sender As Object, _
ByVal e As System.EventArgs) Handles MyBase.Load

    AddHandler Událost, AddressOf Zpracovatel

End Sub
```

Jelikož není možné vytvořit přepojení mezi událostí a jejím zpracovatelem ve fázi psaní kódu, musí být toto přepojení vytvořeno až za běhu programu. Z tohoto důvodu se kód příkazu **AddHandler** nachází v událostní proceduře **Form1_Load**.

Tento řádek kódu naznačuje, že při startu aplikace bude vytvořen vztah mezí událostí a jejím zpracovatelem. Ve skutečnosti je situace komplikovanější, protože Visual Basic .NET automaticky vytvoří delegáta, který obsahuje adresu zpracovatele události. Pokud byste programovali například v C#, museli byste příslušného delegáta explicitně deklarovat, ovšem Visual Basic .NET tuto operaci provádí za vás, a tím minimalizuje vaše programátorské úsilí.

6. Nakonec zbývá už jenom vygenerovat událost. Událost může být generována takřka kdykoliv, závisí tedy jenom na programátorovi, kdy bude chtít událost generovat. Generování událostí je realizováno použitím příkazu **RaiseEvent**:




```
Private Sub Button1_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles Button1.Click

    RaiseEvent Událost(Me)

End Sub
```

Za příkazem **RaiseEvent** následuje název události, kterou chceme generovat. Pokud událost pracuje s parametry, je nutné tyto patřičně naplnit. Jak si můžete všimnout, událost bude generována v okamžiku, kdy uživatel klepne na tlačítko **Button1** (události mohou být samozřejmě vyvolávány také při mnoha dalších příležitostech).