Vysoká škola báňská – Fakulta bezpečnostního inženýrství
Katedra požární ochrany a ochrany obyvatelstva

Ing. Pavel Šenovský

Programovací jazyky

Ostrava 2009

Obsah

Úvod 4
1 NET framework 6
2 První program – AHOJ 8
  2.1 Co je to projekt 8
  2.2 Nastavení vlastností 10
  2.3 Kompilace a kam se program uložil 11
3 Jednoduchá kalkulačka 14
  3.1 Fungování jednoduché kalkulačky 14
  3.2 Událostmi řízené programování 15
  3.3 Typová bezpečnost 17
  3.4 Ladění programu 18
  3.5 Ošetření výjimek 20
4 Jednoduchý textový editor 22
  4.1 Fungování textového editoru 22
  4.2 Nastavení formulářů 22
  4.3 Třídy instance a IO operace 23
5 Analýza textového souboru 30
  5.1 Výběr podle prvního písmene 30
  5.2 Rozdělování řetězce 36
  5.3 CSV soubor 37
6 Trojúhelník – úvod do tvorby tříd 39

Úvod

Tato skripta jsou určena pro výuku předmětu Programovací jazyky oboru Bezpečnostní plánování a jako takové jsou zaměřeny především na výsledek, tedy nikoliv na „informatickou čistotu“.

Výukové texty jsou orientovány spíše problémově. Tedy stanovíme problém a tento problém se pokusíme vyřešit a k tomuto účelu si vysvětlíme teorii, kterou pro jeho vyřešení budeme potřebovat.

Jednotlivé kapitoly jsou doprovázeny příklady. Zdrojové kódy těchto příkladů naleznete v modulu tohoto předmětu na http://prometheus.vsb.cz.

Způsob notace je podobný jako v jiných skriptech, která jsem napsal.

Průvodce studiem
Slouží pro seznámení studentů s látkou, která bude v kapitole probírána.

Čas nutný ke studiu
Představuje odhad doby, který budete potřebovat ke prostudování celé kapitoly. Jedná pouze o orientační odhad, neznepokojujte se proto, pokud Vám studium bude trvat o něco déle nebo budete hotovi rychleji.

Vysvětlení, definice, poznámka
U této ikony najdete vysvětlující text, poznámku k probíranému tématu, která problém uvede do širších souvislostí, popřípadě důležitou definice.

Kontrolní otázky
Na závěr každé kapitoly je zařazeno několik otázek, které prověří, zda jste problematice kapitoly dostatečně porozuměli. Pokud nebudete vědět odpověď na některou otázku, je to signál pro Vás, abyste se ke kapitole vrátili.

Chvilka oddechu
Text označený touto ikonkou neberte příliš vážně, je tam pro Vaše pobavení.

Příklad
Tímto způsobem je označený příklad.

Přeji Vám, abyste čas, který strávíte s tímto textem, byl co možná nejpříjemnější, a abyste jej nepovažovali za ztracený.

Na závěr úvodu si dovolím malé, ale důležité upozornění.

Upozornění

Pro pochopení probírané problematiky je nezbytně nutné, abyste průběžně své poznatky zkoušeli prakticky implementovat ve zvoleném vývojovém prostředí – **pouze průběžná a dlouhodobá** (ne nutně dlouhá) **práce může v programování vést k úspěchu**.

Takže přípravu neponechávejte na poslední chvíli – **TENTO PŘEDMĚT NEOKECÁTE!**

Autor.

1 NET framework

Průvodce studiem

V této kapitole se seznámíme s technologií NET.

Čas nutný ke studiu

Studium této kapitoly můžete zvládnout během 15 minut.

NET framework je skupina knihoven, které tvoří rozhraní mezi službami běžného operačního systému a uživatelem. Jedná se tedy o další abstrakční vrstvu, která zajišťuje:

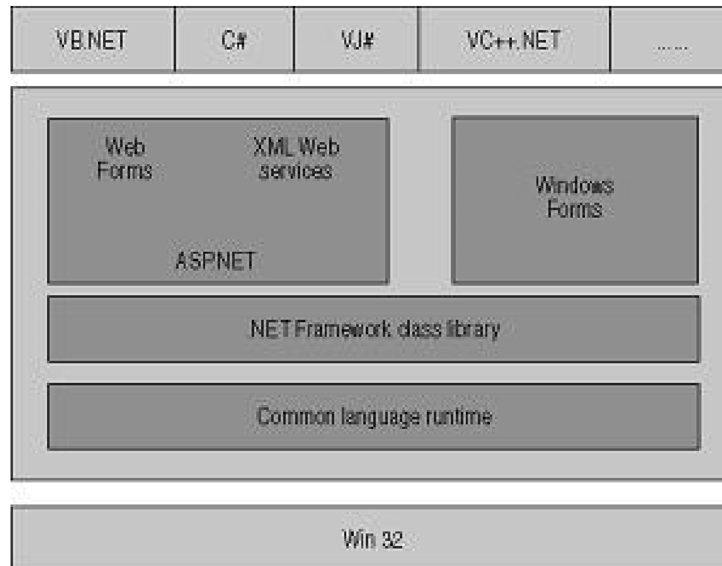
- Automatické řízení přístupu do paměti (nemožnost chyb typu buffer overflow)
- Možnost použít jakýkoliv programovací jazyk podporující NET framework s možnostmi jejich vzájemné spolupráce
- Překlad JIT na koncovém počítači zajišťující vysoký pracovní výkon prostředí

Historicky byla technologie NET koncipována jako odpověď na úspěch programovacího jazyka JAVA, který od 90. let vyvíjí firma SUN. NET technologie jsou oproti JAVA mladší a proto jsou navrženy trochu jiným způsobem, tak aby Microsoft neopakoval chyby, které byly přijaty v rámci vývoje JAVY.

Hlavní rozdíly oproti JAVA:

- Možnost použít více programovacích jazyků
- Menší roztříštěnost API frameworku
- Oficiálně podporována možnost provozovat pouze na operačním systému Windows

Architekturu NET framework si můžeme schematicky znázornit na obr. 1.



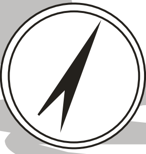
Obr. 1: Architektura NET framework

Podobně jako v případě JAVY se program nejprve zkompiluje do určitého mezistavu. V případě JAVy to byl tzv. byte kód, v případě NET jazyků hovoříme o MSIL (Micorosft Intermediate Language). Tento kód se při prvním spuštění programu dopřeloží do pro daný koncový počítač optimální formy.

To znamená dvě věci:

- První spuštění programu trvá déle, protože se provádí překlad z MSIL do binární formy
- Pro provoz aplikací NET, musí být na koncovém počítači nainstalován .NET Framework Runtime

2 První program – AHOJ



Průvodce studiem

Nastal čas, abychom sestavili náš první program, a samozřejmě nemůžeme začít ničím jiným než programem AHOJ.

Po prostudování kapitoly budete umět

- vytvořit formulář a základní prvky na něm
- změnit vlastnosti prvků na formuláři

Čas nutný ke studiu

Studium této kapitoly můžete zvládnout během 30 - 45 minut. Do této doby však není započten čas nutný na instalaci vývojového prostředí, délka v takovém případě závisí na rychlosti připojení k Internetu a rychlost samotného počítače během instalace.

2.1 Co je to projekt

Zkusme si udělat první program. K tomuto účelu použijeme Microsoft Visual Studio C# Express 2008 (dále VS). Tento program pro začátek neprovede nic jiného než zobrazení jednoduchého formuláře s vykresleným pozdravem AHOJ.

Upozornění

Pokud zatím nemáte instalováno vývojové prostředí, tak teď je ten správný čas pro instalaci.

Pokud máte operační systém Windows XP nebo novější pak můžete instalovat MS Visual Studio C# Express, které můžete používat bezplatně a to i pro komerční účely. Stahovat můžete z http://www.microsoft.com/Express/

Pokud používáte starší Windows 2000, můžete použít open source prostředí IC SharpDevelop ve verzi 1.1 (velmi stará), kterou můžete stáhnout a následně používat zdarma z http://www.icsharpcode.net/OpenSource/SD/Download/.

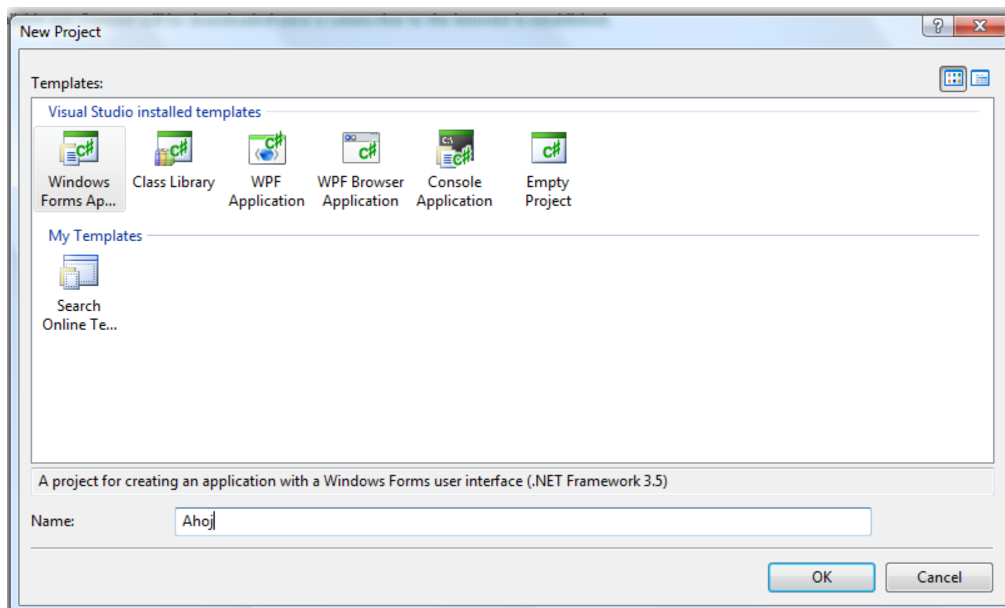
Pokud používáte operační systém Linux, stáhněte si z repozitáře Vaší distribuce prostředí MonoDevelop.

Tato skripta budou předpokládat použití MS Visual Studio C# Express.

Vytvoření takového programu je jednoduché, všechnu práci totiž za nás provede samotné vývojové prostředí. Postupovat budeme tak, že ve Visual Studiu vytvoříme nový projekt typu Windows Application. Vývojové prostředí

samo vytvoří základ naší aplikace včetně hlavního formuláře nazvaného Form1 (viz. Obr. 2).

Ve skutečnosti se toho „skrytě, na pozadí“ provede více. Vytvoří se nové řešení (solution) a v tomto řešení se vytvoří projekt a v rámci tohoto projektu se teprve vytvoří náš formulář. Projekt si je přitom potřeba představit jako funkční celek, třeba spustitelný soubor exe nebo dll knihovna. Projektů může být přitom v rámci řešení obsaženo více. Řešení nám tedy sdružuje projekty, které spolu souvisejí, například hlavní soubor a další projekty pro podpůrné knihovny nutné pro běh hlavního programu.

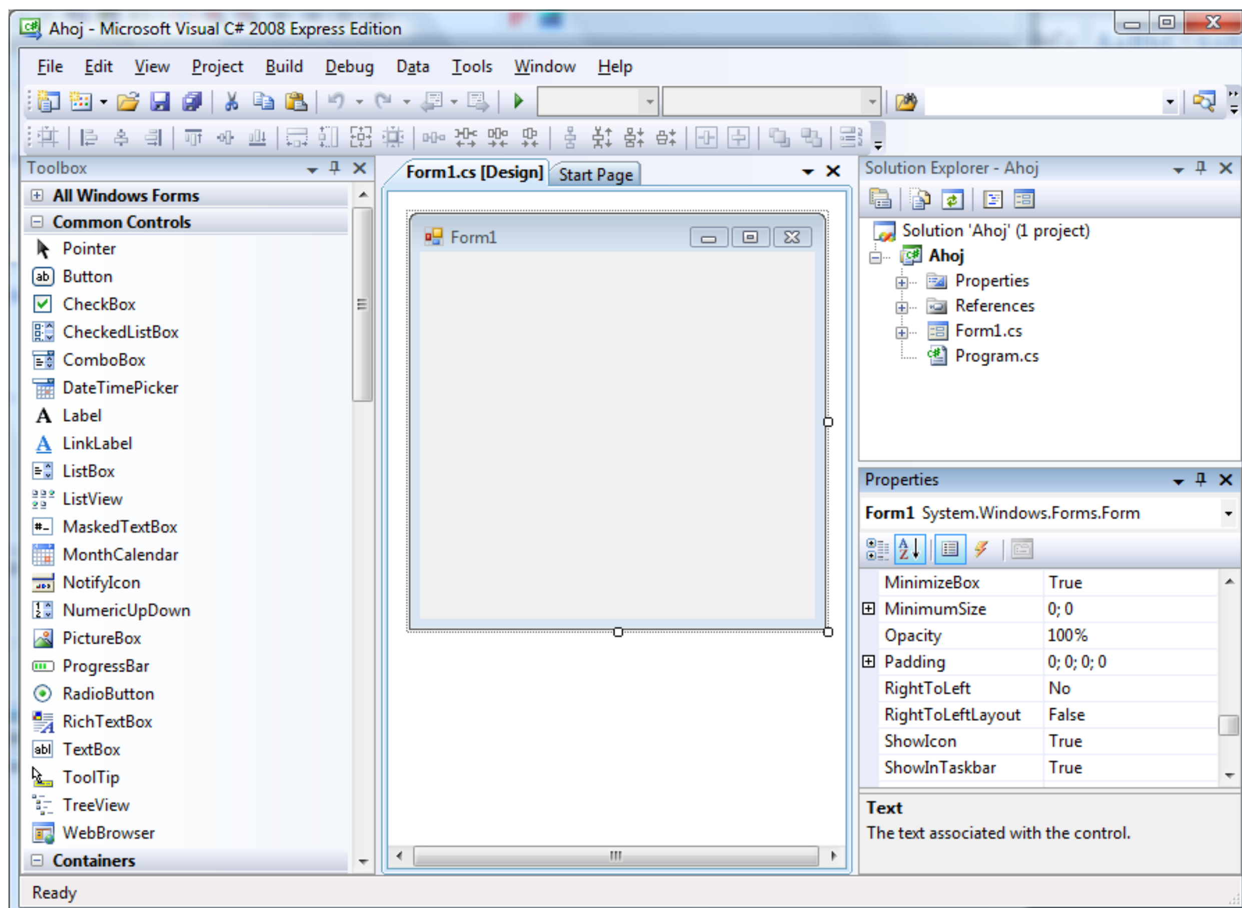


Obr. 2: základní projekt typu Windows Application

V rámci naší práce budeme ale obvykle v rámci jednoho řešení vést pouze jeden projekt. Podívejme se na prostředí Visual Studia a náš projekt, tak jak byl vytvořen (viz. obr. 3).

Visual studio může zobrazovat spoustu různorodých informací v dokovacích oknech. Všechna tato okna je možné vypnout/zapnout v menu View. Pokud tedy nebudete mít některé z oken, o kterých budeme za chvíli mluvit zapnuté, můžete si jej zobrazit právě přes toto menu.

Obrazovka Visual Studia se skládá z několika částí. Na pravé straně máme *průzkumníka řešením (solution explorer)*, který nám umožňuje procházet jednotlivé soubory, které jsou dostupné v rámci projektu. Po vytvoření nového „prázdného“ projektu, máme k dispozici pouze jeden základní formulář nazvaný Form1, který je otevřen a připraven pro práci na vlastním panelu uprostřed obrazovky, a soubor Program.cs, který obsahuje pokyny pro spuštění programu – říká, že se má spustit Form1.



Obr. 3: Visual Studio- základní rozhraní

2.2 Nastavení vlastností

Okno *properties* (na obr. vpravo dole) zobrazuje vlastnosti právě zvoleného objektu na formuláři. V našem případě máme zvolený samotný formulář, a proto se zobrazují právě jeho vlastnosti. Tyto vlastnosti můžeme měnit buďto nyní „staticky“ změnou nastavení v tomto dialogovém okně nebo „dynamicky“ z programového kódu.

- Pro formulář jsou nejdůležitější následující vlastnosti:
- Name – jméno pod kterým můžeme daný objekt ovládat
- BackColor – barva pozadí formuláře
- BackgroudImage – tapeta, která se má použít na pozadí formuláře
- BackgroundImageLayout – stanovuje jakým způsobem se má tapeta použít, v podstatě to funguje podobně jako tapeta Vašeho operačního systému, také ji můžete zobrazit jako dlaždice (tile), roztáhnout (stretch) apod.
- Enabled – zda je formulář povolen (true/false).
- Icon – jaká ikona pro formuláž se má použít
- MaximizeBox a MinimizeBox – určí, zda se má zobrazit maximalizační a minimalizační tlačítko formuláře.
- Text – nastaví, jaký text se má zobrazit v titulku formuláře

- WindowState – nastavuje počáteční stav formuláře po spuštění (normální/minimalizovaný/maximalizovaný)

My budeme chtít, aby náš program zobrazil hlášení Ahoj. Titulek Form1 by nás přitom nepochybně rušil, proto jej změníme – do vlastnosti formuláře Text vepíšeme nový titulek „Můj první program“.

Na formulář nyní dáme statický text „Ahoj“. Tento text není možné vložit na formulář přímo, musíme k tomuto účelu použít objekt typu Label. Objekty na formuláři vytváříme tak, že zvolíme v dokovacím okně *Toolbox* (na obr. 3 vlevo) daný objekt (klikneme na něj) a potom klikneme na formulář do místa, kam jej hodláme vytvořit.

Všechny takto vytvořené objekty se implicitně pojmenovávají stejně typem objektu a pořadím, ve kterém byl vytvořen v rámci formuláře, v našem případě se proto popisek pojmenuje Label1.

Důležité vlastnosti popisku:

- Name – jméno, pod kterým můžeme přistupovat k popisku z programového kódu
- BackColor a ForeColor – barva pozadí a popředí (barva fontu) popisku
- Font – jaký font se má použít (řez, velikost písma apod.)
- Text – jaký text se má v popisku zobrazit
- Visible – zda se má popisek vůbec zobrazit.

My nastavíme vlastnost Text popisku na „AHOJ“. Tím pádem je náš první prográmek hotov. Jak jej ale spustíme.

2.3 Kompilace a kam se program uložil

Abychom mohli program spustit, musíme jej nejprve přeložit (kompilovat). VS rozlišuje dvě formy kompilace a to kompilace a spuštění programu pro ladění (menu Debug -> Start debugging, klávesová zkratka F5) a přeložení programu (menu Build -> Build solution, klávesová zkratka F6).

Po přeložení se vytvoří spustitelné soubory v místě Vašeho projektu, obvykle:

Windows 2000 a XP
C:\Documents and Settings\uživatel\dokumenty\Visual Studion XXXX\projects\jméno projektu\bin\

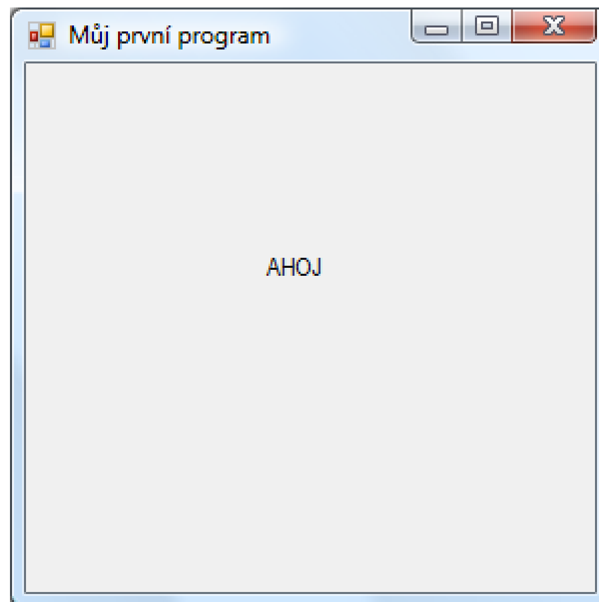
Windows Vista
C:\users\uživatel\dokumenty\Visual Studion XXXX\projects\jméno projektu\bin\

Spustitelné soubory se vytvoří ve složkách debug a release. Při prvním ukládání projektu můžete samozřejmě zvolit i jiné umístnění.

V případě kompilace se vyplatí zapamatovat si klávesové zkratky, protože činnosti překladu budeme provádět poměrně často. Spusťme tedy náš program (F5). Výsledek je znázorněn na obr. 4.

Mnoho činnosti s tímto programem neuděláme – můžeme měnit velikost formuláře, minimalizovat maximalizovat a ukončit běh programu. Běh programu můžeme ukončit buď zavřením formuláře (kliknutím na křížek, klávesovou zkratkou ALT+F4) nebo vysláním pokynu VS, aby ukončilo proces ladění pomocí klávesové zkratky SHIFT+F5.

Pokud spustíme svůj program přímo z jeho umístnění (viz předchozí strana), pak můžeme ukončit běh programu pouze tak, jak je to běžné u ostatních programů (kliknutím na křížek, klávesovou zkratkou ALT+F4), klávesová zkratka SHIFT+F5 bude fungovat pouze v případě, že jsme spustili program pro ladění z VS.



Obr. 4: Program Ahoj

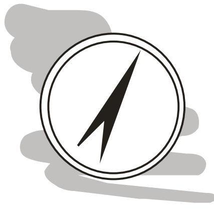
Kontrolní úkoly:

1) Změňte barvu pozadí formuláře na černou a barvu fontu popisku na bílou.
2) Zakažte minimalizaci a maximalizaci formuláře.
3) Přidejte nový popisek obsahující vaše jméno a datum vytvoření tohoto formuláře.
4) Spusťte svůj program přímo (tedy ne v debug režimu).

Zdrojové kódy příkladu:

Zdrojové kódy programu jsou dostupné v modulu Programovací jazyky na prometheus.vsb.cz v sekci příkladů program AHOJ. Zdrojáky v sobě neobsahují zapracované kontrolní úkoly.

3 Jednoduchá kalkulačka



Průvodce studiem

Nyní zkusíme trošku složitější příklad, který už bude potřebovat nějaký programový kód.

Po prostudování kapitoly budete umět

- Oživit formulář
- Vytvářet a řídit události
- Přetypovávat a zpracovávat proměnné
- Zpracovávat výjimky

Čas nutný ke studiu

Pro prostudování této kapitoly budete potřebovat asi dvě hodiny.

3.1 Fungování jednoduché kalkulačky

Předchozí příklad, byl hezký, jednoduchý, bohužel ale také nic nedělal. V tomto příkladě se pokusíme zrealizovat náš první program, ve kterém budeme skutečně programovat.

Zkusme navrhnout jednoduchou kalkulačku. Po ní budeme chtít v rámci zjednodušení, aby vzala dvě čísla a sečetla je. Práce programátora spočívá v řešení třech úloh:

1) Návrh grafického uživatelského rozhraní (GUI)
2) Řešení mechaniky výpočtů (programování řešení problému)
3) Oživení GUI (napojení GUI k výpočtům)

Základem úspěšného řešení problému programem je schopnost programátora vyřešit problém „ručně". Tedy situace, kdy programátor ví, jak problém řešit. Na základě této znalosti, programátor bude vědět, jaké vstupní údaje bude potřebovat a podle toho navrhne GUI i logiku programu.

V našem případě potřebujeme dostat dvě čísla od uživatele, tlačítko, které spustí výpočet, a budeme potřebovat místo, kam vypíšeme výsledek. Vstup od uživatele můžeme získat použitím prvku typu textové pole (TextBox). TextBox funguje podobným způsobem jako Label, TextBox ale umožňuje, aby uživatel text v něm obsažený měnil.

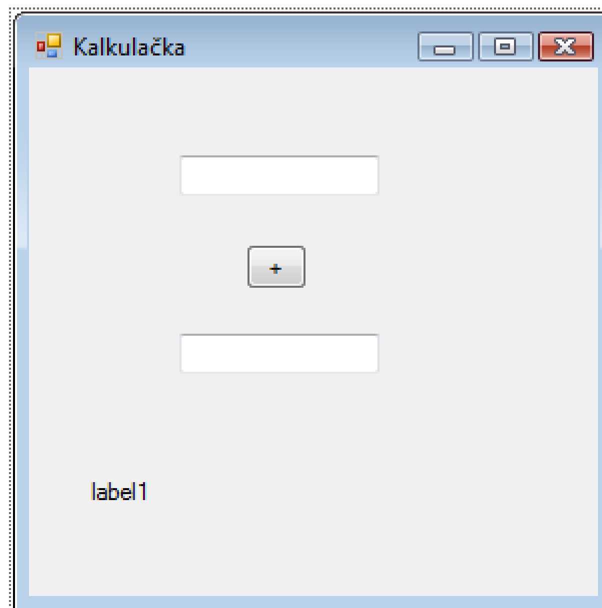
Pro vypsání výsledku můžeme použít nám již dobře známý Label. Jako spouštěč pro operaci součtu můžeme použít tlačítko (Button).

Uvedené prvky si rozmístníme na formuláři a změníme jejich vlastnosti dle potřeby. Celkovou představu o rozložení si můžete vytvořit na základě obr. 5.

Změníme některé vlastnosti prvků, tak aby lépe odpovídaly našim potřebám:

- Vlastnost Text formuláře změníme na „Kalkulačka“
- Vlastnost Text tlačítka změníme na „+“
- Zmenšíme tlačítko na formuláři do čtvercové podoby (prostým tažením)
- Změníme jména (vlastnost „(Name)“) u obou textových polí a popisku na cislo1, cislo2 a vysledek.

Změny v názvech provádíme z toho důvodu, že k těmto ovládacím prvkům budeme potřebovat přistoupit z programového kódu. Jméno TextBox1 nám neříká, co od tohoto prvku očekáváme, v okamžiku, kdy takto pojmenovaných prvků budeme na formuláři mít více (třeba TextBox1 – 10) můžeme jednoduše při programovaní udělat chybu a brát jiné než zamýšlené hodnoty (z jiného textového pole). Změnou názvu se tomuto problému elegantně vyhneme. Takové názvy se navíc dobře pamatují.



Obr. 5: Rozložení prvků na formuláři kalkulačky

Jelikož součet dvou čísel je extrémně jednoduchý, nemusíme příliš rozpitvávat způsob fungování této části programu, můžeme se vrhnout rovnou na jeho realizaci.

3.2 Událostmi řízené programování

Při definování způsobu chování programované aplikace používáme tzv. událostmi řízené programování. Když zobrazíme formulář, provede se událost Load formuláře, když klikneme na tlačítko, provede se událost Click tlačítka apod. Definujeme tedy, co se má provést když nastane nějaká událost. Jelikož

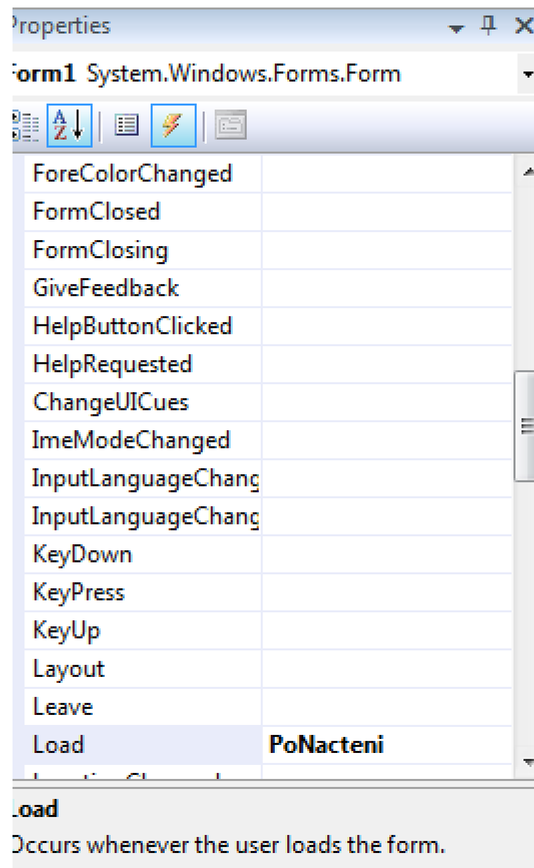
možných událostí je pro každý prvek velké množství nebude mít definice úplně každého smysl – vyplňujeme pouze takové události, které nás z hlediska požadované funkčnosti programu zajímají.

Nás bude zajímat především událost po kliknutí tlačítka, k tomu nás bude zajímat i událost Load formuláře, můžeme ji totiž použít pro smazání ošklivého nápisu label1 v našem popisku. Jistě tento výmaz bychom mohli provést smazáním nápisu ve vlastnosti Text tohoto popisku, jenže pak by popisek přestal být v režimu návrhu viditelný a nám by se s ním v případě potřeby špatně manipulovalo.

Událost můžeme nadefinovat dvojím způsobem: buď dvojklikem na daném prvku – tím se nadefinuje nejčastěji používaná událost daného prvku, nebo přepnutím se do režimu událostí na panelu vlastností (kliknutím na ikonu blesku) a vepsáním jména, pod kterým chceme tuto událost vést. Toto jméno přitom musí být unikátní v rámci daného formuláře.

Já jsem použil druhý způsob a pojmenoval událost *PoNacteni*. Dvojitým kliknutím na jménu události se přepneme do zdrojového kódu dané události. V rámci něho chceme smazat obsah popisku. Jelikož popisek jsme nazvali vysledek, bude kód události vypadat následovně.

```
private void PoNacteni(object sender, EventArgs e)
{
        vysledek.Text = "";
}
```



Obr. 6: Okno událostí formuláře

Postupujeme tedy stejně jako pro manuálním nastavení vlastnosti text. K přístupu však využijeme tečkovací konvenci: jméno_prvku.jméno_vlastnosti. Každý příkaz v C# musí být zakončen středníkem.

Stejným způsobem budeme postupovat i u definice události po kliknutí tlačítka. Tuto událost si můžeme nazvat třeba Plus. Kód této procedury bude následující.

```
private void Plus(object sender, EventArgs e)
{
        vysledek.Text = cislo1.Text + cislo2.Text;
}
```

Tento kód můžeme pokusně spustit (F5). Do jednotlivých textových polí (nazvali jsme je cislo1 a cislo2) vložíme čísla, klikneme na tlačítko plus a uvidíme součet, jenom to nebude součet, který bychom očekávali. Když sečteme číslo 50 a 150 bude v tomto případě výsledek 50150. Čím je to způsobeno?

3.3 Typová bezpečnost

Všimněte si, že ve všech předchozích případech jsme volali vlastnost Text ať už u textových polí nebo popisku. Protože C# je tzv. *typově bezpečné* předpokládá, že vlastnost Text obsahuje vždy textový řetězec a to i když do něj

vložíme nějaká čísla. Budeme muset tyto vstupní údaje tzv. *přetypovat*, tedy změnit datový typ z textového řetězce (string) na desetinné číslo (double).

K účelu přetypování použijeme dvě proměnné typu double, třeba c1 a c2. Přetypování provedeme voláním funkce Convert.ToDouble(co_chceme_převést). Místo funkce ToDouble bychom mohli podle potřeby použít i jiné funkce jako je ToString, ToFloat apod., v našem případě potřebujeme převést textový řetězec na číslo, proto volíme Convert.ToDouble.

Již víme, že vlastnost Text musí obsahovat textový řetězec, jenomže my jsme přetypováním získali číslo – součet budeme proto muset opět převést zpět na textový řetězec. Můžeme k tomu použít funkci Convert.ToString(co_chceme_převést), nebo můžeme tuto funkci zavolat přímo z proměnné součtu. Převod na textový řetězec se totiž provádí tak často, že tato možnost je automaticky dostupná pro každou proměnnou. Podívejme se níže, jak by takto upravený kód vypadal.

```
private void Plus(object sender, EventArgs e)
{
   double c1 = Convert.ToDouble(cislo1.Text);
   double c2 = Convert.ToDouble(cislo2.Text);
   double soucet = c1 + c2;
   vysledek.Text = soucet.ToString();
}
```

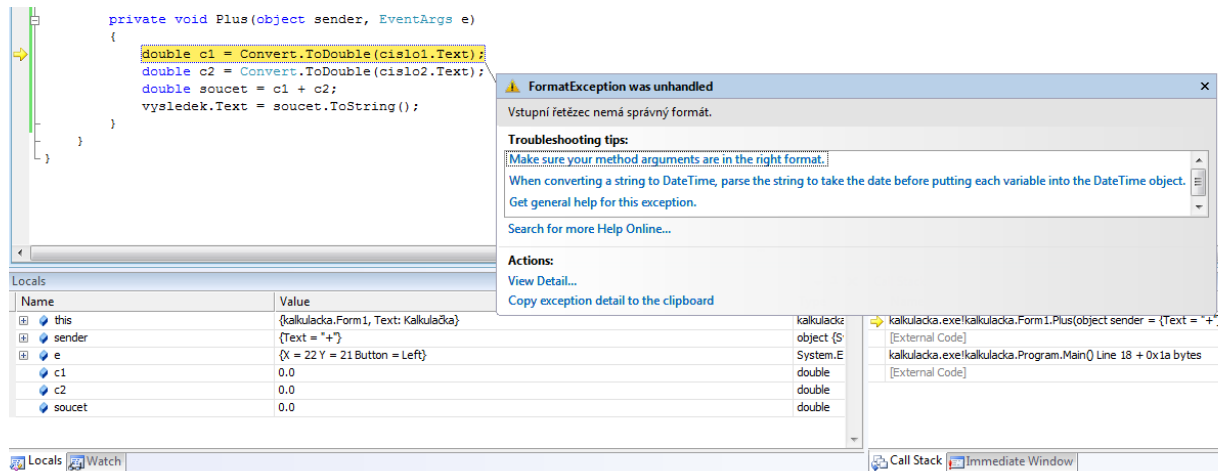
Protože už víme, že C# je typově bezpečné (nemůžeme mísit jednotlivé typy proměnných), je jasné, že pro každou proměnnou musíme stanovit datový typ. Datový typ se stanovuje vypsáním jeho jména před jméno proměnné. Základní datové typy jsou následující:

- int – 32 bitové číslo
- string – textový řetězec
- float – desetinné číslo s jednoduchou přesností
- double – desetinné číslo s dvojitou přesností

Vyzkoušejme funkčnost takto upraveného programu. Do textových polí nyní můžete zadávat celá čísla a čísla desetinná (oddělovač desetinných míst je v standardním nastavení českých Windows desetinná čárka). Alespoň v prvním pokusu si na čísla dejte pozor tak, ať vidíte, že nyní se čísla skutečně sčítají, místo aby se spojovaly jako řetězce… chybu se pokusíme vyvolat až za chvíli.

3.4 Ladění programu

A chvíle pro chybu nastává nyní: vložme do jednoho z (nebo obou) polí něco jiného než číslo, např. text. V okamžiku, kdy klikneme na tlačítko pro sčítání, program se na chvíli vážně zamyslí a následně vyhodí výjimku. Uvidíte obrázek podobný obrázku 7.



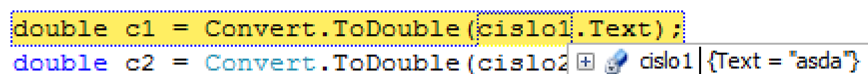
Obr. 7: Naše první výjimka

Výjimka nastane vždy, když se program střetne s něčím neočekávaným. Takovou výjimku můžeme (měli bychom) ošetřit, tedy ji nějak zpracovat. V případě, že program spustíme standardní cestou, tedy ne z prostředí VS, nezpracovaná výjimka vždy skončí pádem celého programu. Zpracovat výjimku je tedy také v našem zájmu.

Protože jsme se setkali s výjimkou v okamžiku, kdy náš program byl spuštěn přes VS, přepne se VS do režimu ladění. Tento režim je přizpůsoben k tomu, abychom mohli velmi pohodlně odhalit zdroj chyb.

V okamžiku kdy nastala výjimka, zastavilo automaticky VS běh programu (funguje stejně jako třeba pauza na DVD přehrávači) a můžeme zkoumat stav jednotlivých proměnných v tomto okamžiku. Žlutě je přitom zvýrazněn řádek, kde k přerušení běhu došlo s vysvětlením výjimky, která nastala.

Ve spodním okraji je okno Locals obsahující informace o stavu lokálních proměnných právě vykonávané funkce. Informace o obsahu proměnných a aktivních prvků můžeme získat také tak, že kurzorem myši najedeme na proměnnou nebo prvek, který nás zajímá, a chvíli počkáme. Objeví se bublinková nápověda s obsahem této proměnné.

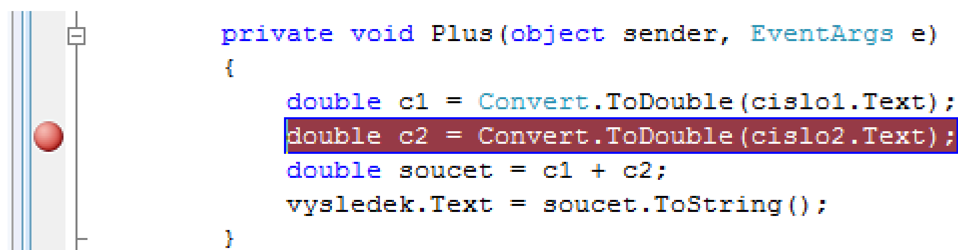


Obr. 8: Zjištění obsahu prvku nebo proměnné

Tímto způsobem můžete rychle a snadno zjistit, jaký je obsah jednotlivých proměnných a snáze tak odhalit příčinu všech problémů. Běh programu přitom můžeme tzv. krokovat – tedy provádět program po jednotlivých řádcích pomocí příkazů Debug -> Step into (klávesová zkratka F11) nebo Debug -> Step over (klávesová zkratka F10).

Step into provede „vstup“ do řádku – tedy pokusí se krokovat i funkce obsažené na tomto řádku, zatímco step over provede provedení celého řádku.

Pokud chceme přerušit běh programu v určitém místě, učiníme tak vytvořením tzv. místa přerušení (break point). V místě, kde má dojít k přerušení, IDE vykreslí červenou tečku a červeně se označí i celý řádek, viz. obr. 9.



Obr. 9: Nastavení break point

Zapínání a vypínání bodů přerušení provádíme kliknutím na šedý okraj vlevo od řádku, kde má k přerušení dojít. Na obr. 9 je znázorněno červenou tečkou.

Pokračovat v běhu programu můžeme opět pomocí menu Debug - > Continue (klávesová zkratka F5) popřípadě můžeme úplně přerušit běh programu pomocí menu Debug - > Stop debugging (klávesová zkratka shift + F5).

3.5 Ošetření výjimek

V životním cyklu aplikace může dojít k situaci, kterou aplikace neočekává a není schopna se s ní bez pomoci vypořádat. V našem případě přesně tato situace nastala v okamžiku, kdy jsme do jednoho z polí naší kalkulačky zadali textový řetězec. Poté, co se program pokusil provést přetypování obsahu pole na desetinné číslo – neuspěl – vyhodila se výjimka.

Neošetřená výjimka v aplikaci způsobí pád aplikace. V případě, že aplikaci spouštíme v režimu pro ladění nám aplikace spadne do IDE se zvýrazněním místa, kde k vyhození výjimky došlo. V případě, že program spouštíme z průzkumníka (obecně mimo prostření Visual studia) a bude jeho běh ukončen úplně.

V každém případě nám program zabrání v další práci s ním. Tomu by bylo dobré zabránit – jinými slovy tuto výjimku ošetřit. Ošetření provedeme zavedením konstrukce try … catch, viz. následující výpis.

```
private void Plus(object sender, EventArgs e)
{
        double c1, c2;
        try
    {
        c1 = Convert.ToDouble(cislo1.Text);
        c2 = Convert.ToDouble(cislo2.Text);
    }catch{
        MessageBox.Show("V textových polích mohou být obsaženy pouze číselné údaje.");
        return;
    }
```

```
    double soucet = c1 + c2;
    vysledek.Text = soucet.ToString();
}
```

Pomocí konstrukce try provedeme pokus o provedení příkazů nacházejících se uvnitř této konstrukce. Pokud dojde při zpracování těchto příkazů k vyhození výjimky, bude tato výjimka zpracována v sekci catch. Pokud catch použijeme, tak jako ve výše uvedeném příkladě bude v této sekci zpracována jakákoliv výjimka. Kromě tohoto způsobu, ale můžeme zpracovávat různé typy výjimek různě – to ale alespoň prozatím vynecháme.

V našem případě dojde při zpracování výjimky k tomu, že vykreslíme okno pomocí metody show objektu MessageBox, které bude obsahovat chybové hlášení, a hned potom ukončíme běh procedury sčítání pomocí příkazu return. Pouze pokud nedojde k vyhození výjimky, provede se součet čísel a vykreslení výsledku na obrazovku.

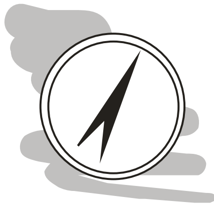
Kontrolní úkoly

1) Doplňte podporu pro další matematické operace (odečítání, násobení a dělení).
2) Upravte program tak, aby po provedení operace se výsledek této operace překopíroval do prvního textového pole a druhé pole vynulujte.
3) Zrealizujte jednoduchý převodní program, pro převod váhy v kilogramech na gramy.

Zdrojové kódy příkladu:

Zdrojové kódy programu jsou dostupné v modulu Programovací jazyky na prometheus.vsb.cz v sekci příkladů program Kalkulačka. Zdrojáky v sobě neobsahují zapracované kontrolní úkoly.

4 Jednoduchý textový editor



Průvodce studiem

Budeme pokračovat o něco složitějším příkladem. Tentokráte se pokusíme vytvořit jednoduchý, ale plně funkční textový editor.

Po prostudování kapitoly budete umět

- Vytvořit a zprovoznit menu
- Načítat a ukládat textové soubory

Čas nutný ke studiu

Pro prostudování této kapitoly budete potřebovat asi dvě hodiny.

4.1 Fungování textového editoru

Textový editor, by nám měl umožňovat, abychom vkládali a upravovali a editovali textové soubory. Náš jednoduchý editor bude schopen zpracovávat pouze čistě textové soubory (tedy soubory bez formátování).

Budeme chtít, aby náš textový editor obsahoval:

- Textové pole, které se bude roztahovat dle velikosti aktivního okna
- Menu, které nám umožní otevírat, ukládat soubory a ukončit program, to vše s použitím klávesových zkratek
- Otevírání souboru by mělo probíhat pomocí specializovaného dialogového okna.

Pusťme se do toho.

4.2 Nastavení formulářů

Vytvoříme si novou aplikaci typu WindowsApplication a v ní budeme mít jeden formulář. U tohoto formuláře nastavíme vlastnost text na „Můj textový editor“.

Vzhled celý můžeme v IDE Visual studio pohodlně naklikat a požadovanou funkčnost získáme prostým nastavením vlastností jednotlivých objektů. Začněme s menu. Menu si v našem projektu vytvoříme pomocí nástroje MenuStrip. Po jeho vytvoření na formuláři se nám objeví proužek s nápisem „Type here“.

Pro definici menu musíme:

1) jednotlivé položky vepsat
2) nastavit jejich vlastnosti (jako jsou klávesové zkratky apod.)

3) oživit menu (vytvořit události ovládající jednotlivé položky menu)

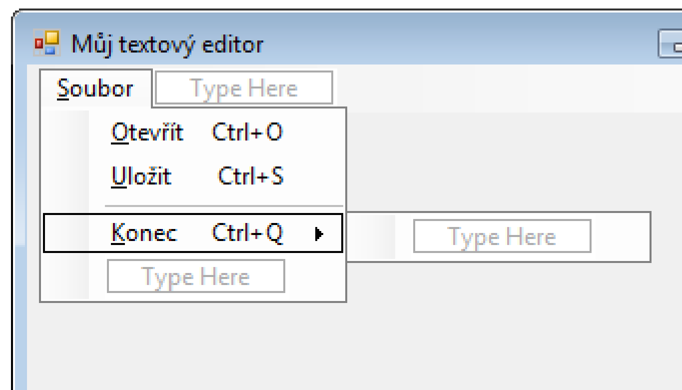
Položky menu velmi jednoduše vypíšeme – prostě klikneme na Type here a píšeme položku – Visual studio zároveň umožní vytvořit další položku menu popřípadě podmenu.

V klasickém menu máme obvykle jedno písmeno podtrženo – toto písmeno je potom použitelné pro navigaci v menu pomocí klávesnice. Menu si tak můžeme otevřít v běhu programu obvykle stisknutím klávesy ALT a výběrem písmene. Menu Soubor -> Otevřít tak může být přístupné přes stisk kláves ALT, S a O.

Visual studio umožňuje takovou funkčnost jednoduše definovat a to přímo v popisu položky menu a to s použitím znaku & označujícího aktivní písmeno. Menu Soubor -> Otevřít tak definujeme jako &Soubor -> &Otevřít.

Klávesové zkratky položky menu nastavíme pro jednotlivé položky menu nastavením vlastnosti ShortcutKeys.

Po spuštění programu bude toto menu funkční okamžitě ve smyslu, že v něm budeme moci procházet, kromě toho samozřejmě nebude dělat nic.



Obr. 10: Definice menu

4.3 Třídy instance a IO operace

Událost po kliknutí na položku menu vytvoříme jednoduše tak, že na položce menu dvakrát klikneme. IDE se předpne do režimu editace zdrojového kódu.

Začněme něčím jednodušším – menu Konec. Při kliknutí na tuto položku menu budeme chtít, aby se formulář uzavřel. K tomu využijeme volání metody Dispose() formuláře. Zdrojový kód bude vypadat následovně:

```
private void konecToolStripMenuItem_Click(object sender, EventArgs e)
{
        this.Dispose();
}
```

Všimněte si klíčového slova this – to je velmi důležité. Nyní se totiž v zásadě poprvé „natvrdo“ setkáváme s objektovými charakteristikami programovacího jazyka C#.

Pozor následuje teoretická část

Následující část musíte nezbytně pochopit! Pokud chcete alespoň trochu „stíhat“ musíte nezbytně alespoň tušit co je to **třída** a **instance třídy**.

Pokud se nepovede ani při tom nejvyšším úsilí neváhejte přijít na konzultaci – **pochopení je naprosto klíčové**.

Třída kam se podíváš

Prakticky cokoliv s čím pracujeme v C# je třídou. Podívejme se na to, co je to třída. **Třída** je objekt, který má určité vlastnosti (properties) a je schopen určitých činností (obsahuje metody).

Například formulář není ničím jiným než třídou. Její vlastnosti jsme si zvykli nastavovat pomocí grafického rozhraní. Také tlačítko nebo textové pole jsou ve skutečnosti třídy se svými vlastnostmi a metodami.

Když nastavujeme vlastnosti formulářů, tlačítek a podobně tak ale ve skutečnosti nepracujeme s třídami, ale instancemi třídy. Třídu je tedy potřeba chápat jako určitou šablonu, návod. Tedy např. všechny formuláře budou mít stejnou strukturu vlastností (titulek, barva pozadí apod.), jednotlivé instance formulářů se ale budou lišit hodnotami, které tyto vlastnosti budou nabývat (např. různé titulky apod.).

Zatím používáme pouze třídy, které pro nás připravil Microsoft, později budeme vytvářet i vlastní třídy.

Třída a instance třídy

K třídě Město vymyslete různé vlastnosti a vymyslete také instance třídy.

Tím se oklikou dostáváme k tomu, co je a co dělá příkaz this.Dispose(). Klíčové slovo this se vztahuje k instanci třídy, ze které je volán. V našem případě se tedy vztahuje k instanci formuláře, se kterým pracujeme. Metoda Dispose() provede, destrukci této třídy, čímž dojde k uzavření formuláře.

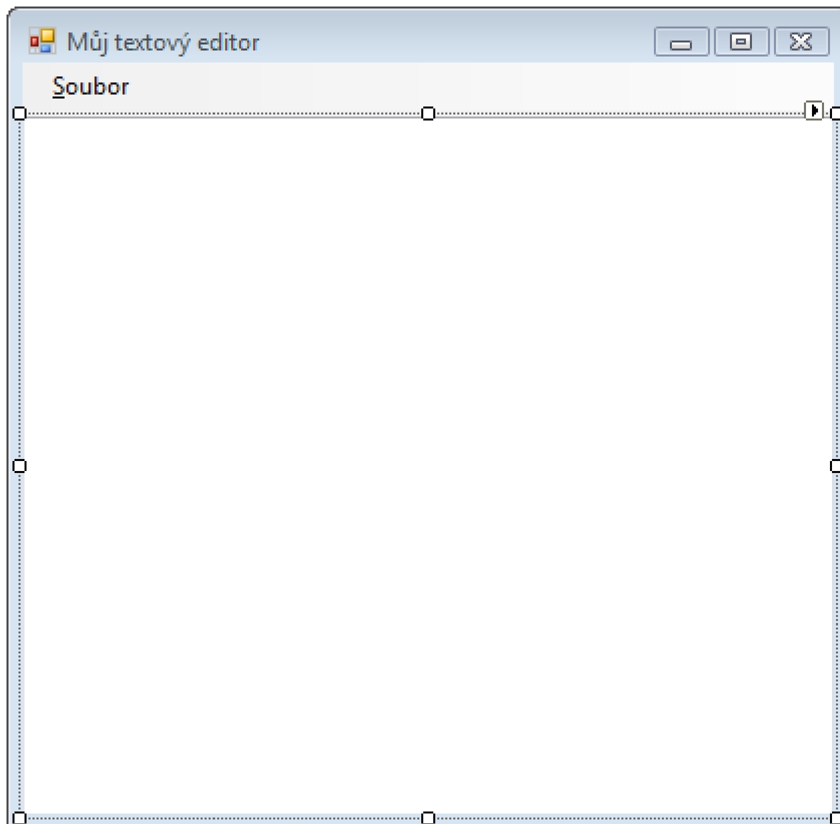
To, že formulář funguje, jak má, můžeme ověřit spuštěním našeho programu (F5). Samozřejmě, kromě uzavírání zatím formulář nebude nic umět.

Třída a instance třídy (řešení)

Město může mít následující vlastnosti: počet obyvatel, primátor, souřadnice GPS, apod. Instancí třídy město může být například Ostrava nebo Praha apod.

Pro zprovoznění textového editoru budeme potřebovat nějaké místo, kam budeme vypisovat text. K tomuto účelu budeme potřebovat textové pole. Toto pole je ale implicitně malé – pouze jeden řádek. Budeme také potřebovat, aby se i v případě změny velikosti formuláře roztáhlo textové pole po celé ploše formuláře.

Tyto požadavky zajistíme nastavením vlastností vytvořeného textového pole. Nejprve nastavíme vlastnost *Multiline* na true. To nám zajistí, že textové pole bude moci obsahovat libovolné množství řádků. Následně roztáhneme textové pole do celé šířky a délky formuláře tak jako na obr. 11.

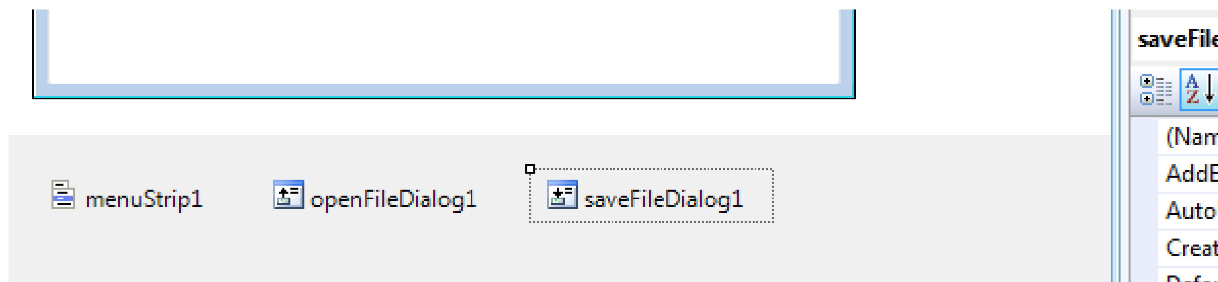


Obr. 11: Textové pole přes celý formulář

Konečně nastavíme vlastnost anchor (kotva) na všechny strany (z top, left na top, left, right a bottom). Můžeme opět spustit program a zkusit, jak změna velikosti funguje.

Teď budeme potřebovat zprovoznit trochu složitější věci – samotné načítání a ukládání souborů. Zjednodušíme si situaci a budeme předpokládat pouze jednoduché textové soubory (TXT) bez formátování.

Operace otevírání a ukládání souborů jsou v aplikacích poměrně časté, proto Microsoft připravil nástroje, které nám tyto činnosti velmi zjednoduší. V nástrojích jsou proto obsaženy nástroje OpenFileDialog a SaveFileDialog. Postupně na ně klikněte a klikněte do formuláře. Všimněte si, že na rozdíl od tlačítek nebo textových polí, nevytváří se instance prvku na formuláři, ale umístní se dolů mimo formulář, tedy podobně jako prvek pro tvorbu menu.



Obr. 12: OpenFileDialog a SaveFileDialog

OpenFileDialog budeme ovládat z události po kliknutí menu Otevřít, zatímci SaveFileDialog budeme ovládat z menu Uložit.

Při použití bychom měli nastavit základní vlastnosti prvků. Začněme s OpenFileDialog.

```
private void otevřítToolStripMenuItem_Click(object sender, EventArgs e)
{
        openFileDialog1.Title = "Otevřít soubor";
        openFileDialog1.Filter = "Textové soubory (*.txt)|*.txt|Všechny soubory
        (*.*)|*.*";
        openFileDialog1.ShowDialog();
        StreamReader sr = new StreamReader(openFileDialog1.FileName);
        textBox1.Text = sr.ReadToEnd();
        sr.Close();
}
```

Tyto vlastnosti tentokrát nastavíme přímo z kódu. Vlastnost Title nastaví titulek dialogového okna. Vlastnost Filter nastaví, jaké soubory se nám budou pro otevření nabízet. Formát filtru je popisek | *.něco | jiný popisek | *.neco2 | atd. Počet filtrů není omezen.

Pomocí metody ShowDialod() potom dialogové okno zobrazíme, aby uživatel vybral soubor. Až uživatel vybere soubor a klikne tlačítko OK, cesta k souboru se uloží do vlastnosti FileName OpenFileDialogu a my ji můžeme dále zpracovávat.

K dalšímu zpracování budeme potřebovat funkce dostupné ze jmenného prostroru IO (input output – vstupní a výstupní operace).

Jmenné prostory

Jmenné prostory jsou v podstatě adresy, na kterých se nachází funkce a objekty podobného charakteru (vzájemně související).

Některé jmenné prostory nám Visual Studio zavádí do zdrojového kódu automaticky. Najdeme je vždy úplně nahoře, na začátku zdrojového kódu. Jmenné prostory se vždy vztahují pouze pro daný soubor.

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

Pokud budeme chtít použít funkce a objekty z jiných jmenných prostorů (a přesně tohle zrovna hodláme udělat), budeme je muset do výčtu výše doplit, nebo tyto funkce volat i s celou jejich adresou. Pro operace IO se jedná o jmenný prostor System.IO. Do výčtu výše proto doplníme.

```
using System.IO;
```

Do používaných jmenných prostorů dáváme jenom takové prostory, které budeme používat intenzivněji.

Načtení textového souboru provedeme pomocí StreamReaderu. Jedná se o objekt, který slouží pro čtení textových proudů. Ve zdrojovém kódu si všimněte, že používáme klíčové slovo new, konkrétně `new StreamReader(openFileDialog1.FileName);`. Tímto volánám (new) vytváříme novou instanci objektu StreamReader. Jako parametr zadávámé parametry tzv. konstruktoru objektu.

Konstruktor

Konstruktor je speciální funkce třídy, která se vyvolává při vytváření instance třídy. Slouží pro prvotní nastavení třídy tak, aby mohla vykonávat své funkce.

Pak už stačí použít jen funkci ReadToEnd(), která přečte celý soubor a výsledek vrátí jako textový řetězec. My tento řetězec vložíme do textového pole textBox1.

Konečně, pokud něco otevřeme, je potřeba to také uzavřít, k tomuto účelu si zavoláme metodu Close().

Všimněte si, že pokud bychom nepřidali do seznamu jmenných prostorů System.IO, museli bychom řádek StreamReaderu přepsat následovně:

```
System.IO.StreamReader sr = new System.IO.StreamReader(openFileDialog1.FileName);
```

Analogicky k tomu bychom vytvořili ovladač události po kliknutí na menu Uložit. Jedinou změnou bude použití StreamWriteru místo StreamReaderu.

```
private void uložitToolStripMenuItem_Click(object sender, EventArgs e)
{
    saveFileDialog1.Title = "Uložit soubor";
    saveFileDialog1.Filter = "Textové soubory (*.txt)|*.txt|Všechny soubory (*.*)|*.*";
    saveFileDialog1.ShowDialog();
    StreamWriter sw = new StreamWriter(saveFileDialog1.FileName);
    sw.Write(textBox1.Text);
    sw.Close();
}
```

Spusťte program a ověřte, že skutečně funguje.

Z hlediska funkčnosti programu ovšem má k dokonalosti daleko – není totiž vůbec žádný problém způsobit pád, nebo jiné nežádoucí chování programu. Prvním problémem je zhroucení programu pokud místo tlačítka OK klikneme na tlačítko Storno – není soubor, který bychom chtěli otevřít nebo do něj uložit. Navíc v případě, že otevíráme soubory opakovaně, tak se provede otevření předchozího souboru (a zahození případných změn, které jste mezitím provedli).

Je tedy potřeba provést kontrolo toho, jak jsme dialogy otevření nebo uložení souboru ukončili. To provedeme pomocí jednoduché podmínky.

```
if (openFileDialog1.ShowDialog() != DialogResult.OK) return;
```

resp.

```
if (saveFileDialog1.ShowDialog() != DialogResult.OK) return;
```

Porovnáváme výsledek dialogového okna s konstantou DialogResult.OK. V případě, že je stisknuto cokoliv jiného než OK (!=), ukončíme běh události (return).

Tímto způsobem můžeme zapsat pouze podmínky na jeden řádek. Obecně pro více řádků můžeme zapsat následovně:

```
if (podmínka)
{
        ... pokud podmínka vyhodnocena jako pravda
}
else
{
        ... pokud podmínka vyhodnocena jako nepravda
}
```

V rámci podmínek můžeme používat následující operátory porovnání:

== je rovno
!= není rovno
> je větší
< je menší
>= je větší nebo rovno
<= je menší nebo rovno

Složitější podmínky můžeme tvořit pomocí logických spojek

&& and
|| or

A vyhodnocování můžeme ovlivnit pomocí závorek.
Celý upravený zdrojový kód pak bude vypadat následovně:

```
private void otevřítToolStripMenuItem_Click(object sender, EventArgs e)
{
        openFileDialog1.Title = "Otevřít soubor";
        openFileDialog1.Filter = "Textové soubory (*.txt)|*.txt|Všechny soubory
        (*.*)|*.*";
        if (openFileDialog1.ShowDialog() == DialogResult.OK) return;
        StreamReader sr = new StreamReader(openFileDialog1.FileName);
        textBox1.Text = sr.ReadToEnd();
        sr.Close();
}

private void uložitToolStripMenuItem_Click(object sender, EventArgs e)
{
        saveFileDialog1.Title = "Uložit soubor";
        saveFileDialog1.Filter = "Textové soubory (*.txt)|*.txt|Všechny soubory (*.*)|*.*";
        if(saveFileDialog1.ShowDialog() == DialogResult.OK) return;
        StreamWriter sw = new StreamWriter(saveFileDialog1.FileName);
        sw.Write(textBox1.Text);
        sw.Close();
}
```

Kontrolní úkoly:

Přidejte do menu položku Nový, která smaže text v okně stávajícího soboru.

Zdrojové kódy příkladu:

Zdrojové kódy programu jsou dostupné v modulu Programovací jazyky na prometheus.vsb.cz v sekci příkladů program Textový editor. Zdrojáky v sobě neobsahují zapracované kontrolní úkoly.

5 Analýza textového souboru

Průvodce studiem

Jedním z nejčastějších úkolů, které v programování řešíme je načítání a zpracovávání textových souborů. To je také jedním ze základních důvodů, proč byl do inženýrského studia zaveden předmět Programovací jazyky.

Po prostudování kapitoly budete umět

- Zpracovat data z textového souboru

Čas nutný ke studiu

Pro prostudování této kapitoly budete potřebovat asi dvě hodiny.

Práci s textovými řetězci si vyzkoušíme na několika příkladech. Všechny tyto příklady budou zpracovávat stejný datový soubor a to konkrétně soubor ERPG.txt, který obsahuje seznam nebezpečných látek a určité úrovně zájmových koncentrací těchto látek.

Nejprve zkusíme ze seznamu vybrat látky začínající na určité písmeno. Potom zkusíme řádky látek rozdělit podle významových sloupců a přehodit jejich sloupce. Jako třetí úkol si dáme vytvoření plnohodnotného souboru CSV z textového souboru.

5.1 Výběr podle prvního písmene

Prvním úkolem, jehož splnění nás čeká je výběr látek podle prvního písmene. K splnění tohoto úkolu rozšíříme program, který jsme si připravili v minulé kapitole.

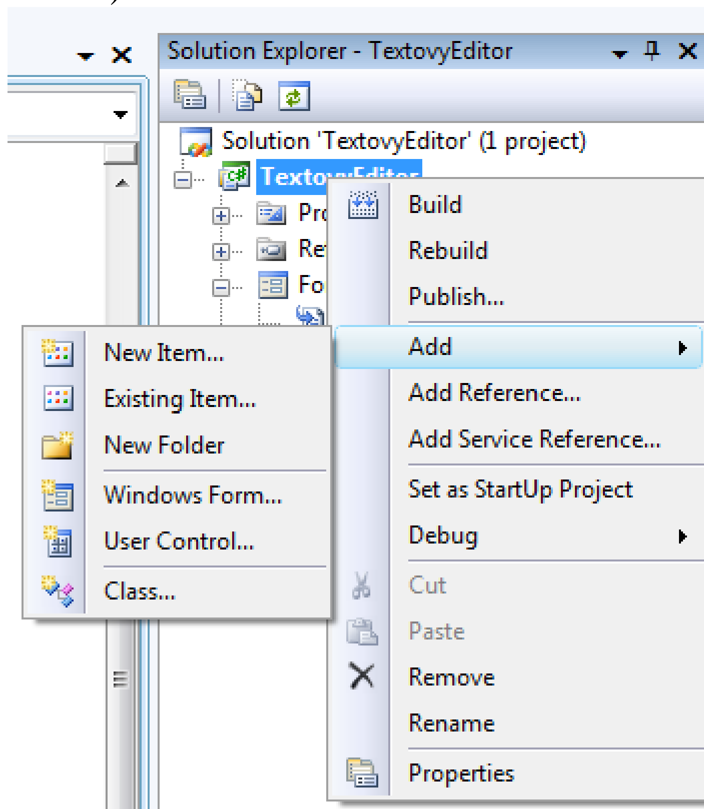
Rozšíříme menu o další položku nazvanou ERPG a tu rozvineme do submenu, kam budeme vkládat jednotlivé položky spouštějící úkoly z této kapitoly. Začneme položkou nazvanou Filtrace dle písmene.

Podívejme se na to, co by měl náš program v tomto ohledu dělat. Měl by načíst textový soubor, tento soubor vy však měl být zpracováván po řádcích, tedy po jednotlivých látkách. Při zpracování řádku se podíváme na jeho první písmeno, a pokud odpovídá zvolenému písmenu, řádek přidáme do textového pole.

Změněné požadavky na funkčnost s sebou nesou požadavky na změnu algoritmu, který použijeme pro zpracování textového souboru.

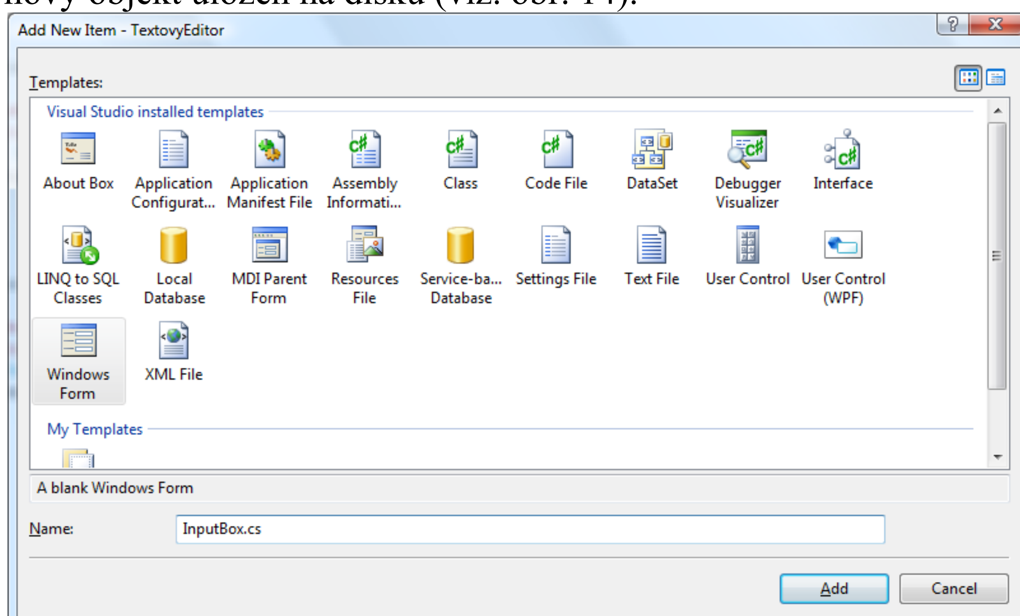
Ovladač dialogového okna *Otevřít soubor* zůstane stejný, avšak je potřeba navíc získat písmeno, vyřešit načítání po řádcích a vyhodnocování počátečního písmene řádku.

Zadávání písmene řádku vyřešíme pomocí samostatného formuláře, který bude vypadat podobně jako na obr. 15. Nový formulář vytvoříme z kontextového menu projektu kliknutím na volbu Add a Windows Form (viz. obr. 13).



Obr. 13: Vytvoření nového formuláře

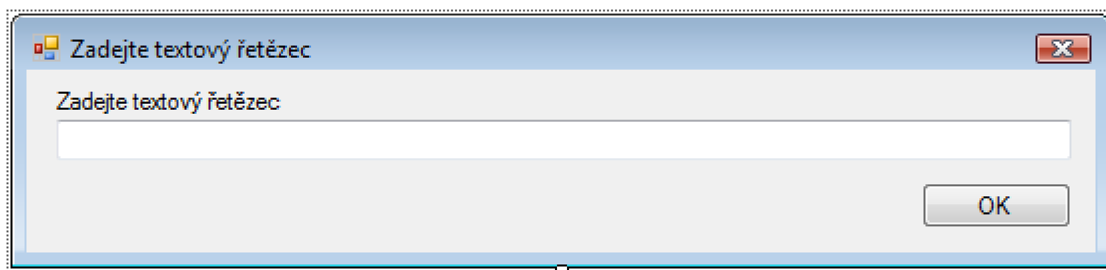
Objeví se dialogové okno, které nám umožní nadefinovat si typ nově vytvářeného objektu a také nám umožní definovat jméno, pod kterým bude tento nový objekt uložen na disku (viz. obr. 14).



Obr. 14: Nový objekt

Dialogové okno z obr. 14 je potřeba chápat jako výběr určité šablony. Vybereme šablonu typu objektu, který hodláme vytvořit a Visual Studio provede základní nastavení, čímž nám jako programátorům velmi usnadní práci. My si vybereme šablonu Windows Form (formulář Windows) a pojmenuje ji třeba InputBox.cs.

Samotný formulář navrhneme graficky tak, jak už jsme si zvykli.



Obr. 15: Formulář pro zadání textového řetězce

Pro formulář byl použit popisek, textové pole, tlačítko a nastavení pár vlastností. Možná vás napadne jak docílit schování minimalizačního a maximalizačního tlačítka. Obě jsme zakázali nastavením vlastností formuláře na false a to konkrétně vlastností MaximizeBox a MinimizeBox. Konečně zabráníme veškerým změnám velikosti pomocí nastavení vlastnosti formuláře FormBorderStyle na FixedDialog.

Formulář oživíme – budeme chtít, aby po kliknutí na tlačítko OK se přeneslo to, co je napsáno v textovém poli do nějaké proměnné, kterou použijeme pro další zpracování v našem programu.

Možná Vás napadne, proč to dělat tak složitě? Odpovědí je kvůli **zapouzdření** (encapsulation).

Pozor následuje teoretická část

Následující část musíte nezbytně pochopit! Pokud chcete alespoň trochu „stíhat" musíte nezbytně alespoň tušit co je to **zapouzdření**.

Pokud se nepovede ani při tom nejvyšším úsilí neváhejte přijít na konzultaci – **pochopení je naprosto klíčové**.

Už jsme si říkali, že v C# pracujeme prakticky výhradně s objekty. Víme, že objekty jsou celistvé a pracujeme s nimi obvykle pomocí odkazů, které nazýváme instancemi.

Zapouzdření není nic jiného, než filosofický přístup k objektům – to co je vně objektu nezajímá, jak přesně objekt pracuje, ale jaké služby a vlastnosti poskytuje. Mimo objekt tedy vidíme co, ale ne jak objekt pracuje. Tedy pro přístup z vnějšku definujeme rozhraní, které bude z vnějšku přístupné.

Podívejme se na pseudokód demonstrující viditelnost proměnných a funkcí:

```
public class trida1
{
   public int trida_prom1;
    private int trida_prom2;

    public int funkce1()
    {
        int fce_prom1 = this.trida_prom1;
        if (this.trida_prom1 == 1)
        {
                int fce_prom3;
        }
   }

   private int funkce2()
    {
        int fce_prom2 = this.trida_prom2;
        if (this.trida_prom1 == 1)
        {
                int fce_prom4;
        }
   }
}
```

V předchozím pokusném kódu máme definované dvě proměnné trida_prom1 a trida_prom2 celočíselného typu deklarované jedna jako veřejná a jedna jako privátní proměnná. Z hlediska vnitřního fungování třídy, je veřejnost nebo privátnost proměnných a funkcí (uvnitř) třídy irelevantní.

Proto funkce 1 i 2 „vidí“ obě proměnné (tedy trida_prom1 i trida_prom2). Zevnitř funkcí k těmto proměnným přistupujeme pomocí klíčového slova this. Proměnné, které jsou ale deklarovány uvnitř funkcí, mimo funkce viditelné nejsou. Proto ve funkci 2 nemůžeme zpracovávat proměnné deklarované ve funkci 1.

Podobně pokud nadeklarujeme proměnnou uvnitř podmínky nebo třeba cyklu, bude tato proměnné viditelná právě jen v této podmínce nebo cyklu a nebude viditelná ani ve zbytku funkce.

Pokud tedy priváte a public neovlivňují viditelnost proměnných a funkcí uvnitř tříd – co dělají? Ovlivňují, co bude viditelné mimo třídu.

Private je tak viditelné pouze uvnitř dané třídy. Public je viditelné i mimo ni.

Pro splnění našeho úkolu si proto vytvoříme veřejnou proměnnou typu string, můžeme ji nazvat PredanyText a do této proměnné uložíme to, co jsme napsali do textového pole.

Zdrojový kód našeho příkladu tak bude vypadat následovně.

```
public partial class InputBox : Form
```

```
{
        public string PredanyText;

        public InputBox()
        {
                InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
                this.PredanyText = textBox1.Text;
                this.Hide();
        }
}
```

Všimněte si, že na konci události po kliknutí, volám metodu Hide() místo metody Dispose(). To je dáno tím, že potřebuji uchovat instanci dialogového okna v paměti, dokud něco neudělám s proměnnou (jinak by to nemělo smysl). Z tohoto důvodu dialogové okno pouze skryji, místo toho, abych jej zrušil.

Z hlavního programu spustíme formulář následovně:

```
InputBox inp = new InputBox();
inp.ShowDialog();
if (inp.PredanyText.Length != 1)
{
        MessageBox.Show("Je potřeba zadat jediné písmeno");
        return;
}
```

Tedy vytvoříme instanci formuláře InputBox, který jsme před chvílí navrhli a pomocí metody ShowDialog() jej zobrazíme. Metoda ShowDialog zobrazuje formulář v režimu tzv. modálního okna. To znamená, že uživatel bude muset vyplnit formulář předtím, než bude moci pokračovat v práci s původním formulářem.

Oproti tomu metoda Show() zobrazí normální okno a uživatele bude moci pracovat s oběma formuláři zároveň. Teoreticky je možné zobrazit neomezené množství formulářů, prakticky jsme ale omezeni pamětí a především pracovní plochou a naší omezenou schopností se na ní vyznat.

V druhé části programu výše provádím testování délky předaného řetězce. Využívám k tomu vlastnosti řetězce Length, tedy délka. Pokud tato délka není rovna 1, tedy nejedná se o písmeno, vypíšeme chybové hlášení a ukončíme načítání.

Načítání souboru po řádcích provedeme pomocí cyklu while. Jeho syntaxe pro náš problém bude následující.

```
string radek = "";
while ((radek = sr.ReadLine()) != null)
{

}
```

Podmínka, která se vyhodnocuje je trošku složitější v její první části (`radek = sr.ReadLine()`) načteme řádek souboru a porovnáme jej s hodnotou null (tedy nic). Pokud při načítání dojdeme na konec souboru při každém dalším pokusu o načtení řádku (ReadLine) bude vrácena právě hodnota null – a v tomto okamžiku ukončíme běh cyklu.

Pro vyhodnocení prvního písmene využijeme toho, že textový řetězec (string) je možno chápat jako pole znaků (char). Podmínka pro vyhodnocování prvního písmene proto bude vypadat následovně:

```
if (radek[0] == Convert.ToChar(inp.PredanyText))
```

Tedy podíváme se na první písmeno řádku (s indexem 0) a porovnáme jej se zadaným písmenem. Textový řetězec v tomto případě musíme přetypovat na znak. Alternativně bychom mohli použít také první znak zadaného řetězce. Podmínka by pak vypadala následovně:

```
if (radek[0] == inp.PredanyText[0])
```

Funkce v obou případech bude samozřejmě stejná.

Zdrojový kód se vším všudy, bude vypadat následovně:

```
private void filtracePísmenoToolStripMenuItem_Click(object sender, EventArgs e)
{
	openFileDialog1.Title = "Otevřít soubor";
	openFileDialog1.Filter = "Textové soubory (*.txt)|*.txt|Všechny soubory
(*.*)|*.*";
	InputBox inp = new InputBox();
	inp.ShowDialog();
	if (inp.PredanyText.Length != 1)
	{
		MessageBox.Show("Je potřeba zadat jediné písmeno");
		return;
	}
	if (openFileDialog1.ShowDialog() != DialogResult.OK) return;
	StreamReader sr = new StreamReader(openFileDialog1.FileName);
	string radek = "";
	string vysledek = "";
	while ((radek = sr.ReadLine()) != null)
	{
		if (radek[0] == Convert.ToChar(inp.PredanyText))
		{
			vysledek += radek + "\r\n";
		}
	}
	sr.Close();
	textBox1.Text = vysledek;
}
```

Jediný řádek, který jsme si zatím nevysvětlili je `vysledek += radek + "\r\n";`. Operátor += znamená to stejné jako vysledek = vysledek + radek + … Zápis +=, ale také -= je doporučován používat z hlediska optimalizace překladače C# (ale

také C++). Řádek tedy nedělá nic jiného, než že v případě, že řádek začíná na požadované písmeno, celý řádek připojí do proměnné vysledek.

Na konec řádku přidáváme znaky „\r\n“. Jak jste si zajisté všimli při zkoušení programu, tyto znaky se do textového souboru nevykreslují, jedná se totiž o tzv. escape sekvence. \r\n jsou dva znaky používané v systému Windows pro signalizaci konce řádku.

Přehled všech používaných escape sekvencí je obsažen v následující tabulce.

Tab. 1: Escape characters

Escape character	Význam
\n	Nový řádek
\r	Carriage return
\r\n	Kombinace obou konců řádek
\“	Uvozovky
****	Zpětné lomítko „\“
\t	tabulátor

Soubor s ERPG informacemi

Soubor, se kterým pracujeme, najdete na http://prometheus.vsb.cz.

5.2 Rozdělování řetězce

Jako základ pro práci použijeme opět to, co jsme vytvořili před chvílí. Na rozdíl od předchozího příkladu musíme znát formát textového souboru. Když se na něj podíváme (pro úryvek viz. níže) zjistíme, že jednotlivé sloupce jsou zde rozděleny pomocí znaku tabulátoru.

```
Acetaldehyde (75-07-0)  10 ppm      200 ppm     1000 ppm
Acetic Acid (64-19-7)   5 ppm 35 ppm      250 ppm
Acetic Anhydride (108-24-7)   0.5 ppm     15 ppm      100 ppm
Acrolein (107-02-8)     0.05 ppm    0.15 ppm    1.5 ppm
Acrylic Acid (79-1 0-7) 2 ppm 50 ppm      750 ppm
Acrylonitrile (107-13-1)      10 ppm      35 ppm      75 ppm
Allyl Chloride (107-05-1)     3 ppm 40 ppm      300 ppm
```

Máme zde jméno s CAS číslem a jednotlivé koncentrace ERPG 1 – 3. Řekněme, že budeme chtít prohodit ze cvičebních důvodů pořadí sloupců ERPG. Výsledný textový soubor tedy bude mít následující podobu: jméno a CAS, ERPG 3, 2, 1.

Vypustíme tedy načítání písmene a změníme práci algoritmu v rámci cyklu načítajícího textový soubor. Úprava není složitá, proto vypíši zdrojový kód celé události a teprve poté jej doplním výkladem.

```
private void rozděleníŘetězceToolStripMenuItem_Click(object sender, EventArgs e)
{
        openFileDialog1.Title = "Otevřít soubor";
        openFileDialog1.Filter = "Textové soubory (*.txt)|*.txt|Všechny soubory
(*.*)|*.*";
        if (openFileDialog1.ShowDialog() != DialogResult.OK) return;
        StreamReader sr = new StreamReader(openFileDialog1.FileName);
        string radek = "";
        string vysledek = "";
        string[] pole_r;
        while ((radek = sr.ReadLine()) != null)
        {
                pole_r = radek.Split('\t');
                vysledek += pole_r[0] + "\t" + pole_r[3] + "\t" + pole_r[2] + "\t" +
        pole_r[1] + "\r\n";
        }
        sr.Close();
        textBox1.Text = vysledek;
}
```

Před začátkem cyklu definuji pole řetězců nazvané pole_r. To, že se jedná o pole, indikuje přítomnost hranatých závorek [] za určením datového typu. Jelikož toto pole budeme naplňovat funkcí, není potřeba, abychom stanovovali počet políček pole.

V samotném cyklu provedu naplnění pole pomocí funkce split, kterou volám nad řádkem souboru, který zpracovávám. Jako parametr této funkce zadáváme písmeno, které chceme použít pro rozdělení řetězce. V našem případě je to znak tabulátoru, pro jehož zadání použijeme escape sekvenci \t.

Všimněte si také, že tento parametr je ohraničen apostrofy a nikoliv uvozovkami – uvozovky totiž ohraničují textové řetězce, zatímco apostrofy mohou ohraničovat textové řetězce a znaky.

Samotné sestavení výsledku je už jednoduché, pokud je struktura souboru následující (včetně indexů při rozdělení podle tabulátoru):

Jméno (0) – ERPG1 (1) – ERPG2 (2) – ERPG3 (3)

A já chci mít

Jméno (0) – ERPG3 (3) – ERPG2 (2) – ERPG(1)

Nezbývá než tyto indexy použít pro pole.

5.3 CSV soubor

Třetím úkolem, který jsme si pro tuto kapitolu vytyčili je vytvoření CSV souboru, který můžeme otevřít třeba v MS Excel. CSV je zkratka pro Comma Separated Values, tedy soubor ve kterém jsou jednotlivé sloupce odděleny pomocí čárek. Ve skutečnosti to nemusí být a také často nejsou čárky. My použijeme středníky.

Tento úkol budě opět aplikaci toho předchozího. Budeme tedy opět používat funkci split. Tentokrát ale dvakrát, protože navíc ke jménu a hodnotám ERPG budeme chtít osamostatnit také CAS číslo.

Zdrojový kód řešení bude vypadat následovně:

```
private void cSVSouborToolStripMenuItem_Click(object sender, EventArgs e)
{
	openFileDialog1.Title = "Otevřít soubor";
	openFileDialog1.Filter = "Textové soubory (*.txt)|*.txt|Všechny soubory
(*.*)|*.*";
	if (openFileDialog1.ShowDialog() != DialogResult.OK) return;
	StreamReader sr = new StreamReader(openFileDialog1.FileName);
	string radek = "";
	string vysledek = "jméno;CAS;ERPG-1;ERPG-2;ERPG-3\r\n";
	string[] pole_r;
	string[] jmeno_cas;
	while ((radek = sr.ReadLine()) != null)
	{
		pole_r = radek.Split('\t');
		jmeno_cas = pole_r[0].Split('(');
		jmeno_cas[0] = jmeno_cas[0].Trim();
		jmeno_cas[1] = jmeno_cas[1].Replace(")", "");
		vysledek += jmeno_cas[0] + ";" + jmeno_cas[1] + ";" + pole_r[1]+ ";" +
	pole_r[2] + ";" + pole_r[3] + "\r\n";
	}
	sr.Close();
	textBox1.Text = vysledek;
}
```

Jak vidno oproti předchozímu případu jsme přidali hlavičku tabulky. Jméno a CAS číslo osamostatňujeme pomocí rozdělení řetězce podle znaku „(„ a na jednu půlku pouštíme metodu Trim() jejímž úkolem je odebrat případné mezery na začátku a konci řetězce (existují varianty TrimEnd a TrimStart, které odstraňují mezery pouze na začátku respektive na konci řetězce). Používáme také funkci Replace, která umožňuje zaměnit určitý znak za jiný.

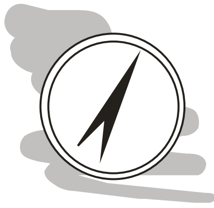
Kontrolní úkoly:

Zkuste příklady modifikovat navrhnout vlastní tabulku ve formě CSV souboru a zpracujte ji v programu (třeba prohoďte v rámci programu sloupce).

Zdrojové kódy příkladu:

Zdrojové kódy programu jsou dostupné v modulu Programovací jazyky na prometheus.vsb.cz v sekci příkladů program Hrátky s textem. Zdrojáky v sobě neobsahují zapracované kontrolní úkoly.

6 Trojúhelník – úvod do tvorby tříd



Průvodce studiem

V této kapitole začneme tvořit vlastní třídy. Budeme pracovat s něčím, co je nám známo – s trojúhelníkem. Budeme počítat jeho vlastnosti jako je délka a obsah.

Budeme se zabývat jeho konstrukcí.

Po prostudování kapitoly budete umět

- Tvořit vlastní třídy
- Psát komentáře pro dokumentaci tříd

Čas nutný ke studiu

Pro prostudování této kapitoly budete potřebovat asi dvě hodiny.

Kromě GUI (grafického uživatelského rozhraní) , budeme tentokrát potřebovat vystavět solidní strukturu tříd, které budou vykonávat činnosti, které budeme potřebovat.

Pro zjednodušení budeme pracovat s pouze 2D souřadnicemi trojúhelníku. O trojúhelníku můžeme prohlásit, že se skládá z hran a o těch můžeme prohlásit, že jsou definovány ohraničujícími body. Budeme tedy potřebovat celkem tři třídy: Bod, Úsečka a Trojúhelník.

Začněme specifikací vlastností třídy Bod. Třída bod bude mít dvě vlastnosti – x-ovou a y-ovou souřadnici. Tyto vlastnosti nastavíme pomocí konstruktoru třídy.

```
/// <summary>
/// veřejná třída reprezentující bod v 2D prostoru
/// </summary>
public class Bod
{
        #region promenne
        private double x, y;
        #endregion

        #region vlastnosti
        /// <summary>
        /// X-ová souřadnice bodu
        /// </summary>
        public double X
        {
                set{x = value; }
                get{return x; }
        }

        /// <summary>
        /// Y-ová souřadnice bodu
```

```
        /// </summary>
        public double Y
        {
                set{y = value; }
                get{return y; }
        }
        #endregion

        /// <summary>
        /// Veřejný konstruktor bodu. Jako vstupní parametry konstruktoru jsou
        /// považovány X, Y a Z-ová souřadnice bodu (uvažujeme ve 3D).
        /// </summary>
        /// <param name="x">X-ová souřadnice bodu</param>
        /// <param name="y">Y-ová souřadnice bodu</param>
        public Bod(double x, double y)
        {
                X = x;
                Y = y;
        }
}
```

Všimněte si způsobu, jakým je třída sestavena. Nejprve jsou definovány privátní proměnné třídy typu double (desetinné číslo – dvojitá přesnost). Ty jsou ohraničeny direktivou region. Direktiva region má za úkol zpřehlednit kód ve vývojovém prostředí – tyto regiony je totiž možné sbalit, tak aby nám nepřekážely v další práci.

Následuje definice vlastností třídy. Tu definujeme následovně:

```
        public double X
        {
                set{x = value; }
                get{return x; }
        }
```

Deklarace je stejná jako u proměnné, všimněte si ale, že název vlastnosti je velkým písmem a pro název proměnné jsem použil písmo malé. Využívám tak schopnosti C# rozlišovat velikost písmen a proto X != x.

Možná vás napadne otázka, čím se liší veřejná vlastnost třídy od veřejné proměnné třídy. Rozdíl je ve způsobu zpracování. Proměnná má vždy nějako hodnotu a tato hodnota je stejná pro třídu, ve které byla proměnná deklarována i mimo třídu. Vlastnost proti tomu je pouze rozhraní, kterým třída komunikuje s okolím. Tedy na vnějšek můžeme deklarovat, že souřadnice X bude v metrech, ale vnitřně může třída pracovat se souřadnicí v milimetrech:

```
                set{x = value * 1000; }
```

Tedy mechanismus vlastností spadá do oblasti nástrojů podporujících zapouzdření. Z vnějšku nás nezajímá, jak třída uvnitř pracuje - využíváme pouze zveřejněné rozhraní.

Každá vlastnost má také část set a get. Set se spouští pokaždé, když nastavujeme vlastnost, např.:

```
Bod b = new Bod(0, 0);
b.X = 15;
```

Vytvořil jsem instanci třídy Bod nazvanou b o souřadnicích [0, 0] a souřadnici X měním na 15. Výsledkem bude bod o souřadnicích [15,0]. Při nastavování vlastnosti se v třídě vyvolá část set dané vlastnosti, v našem případě: `set{x = value; }`. Tedy do proměnné x se předá nastavená hodnota vlastnosti. Podstatné je, že vnitřně v rámci třídy jsou vlastnosti ukládány do proměnných.

Část get se vyvolá, pokud naopak chceme získat hodnotu dané vlastnosti.

Jelikož vlastnost je dostupná z vnějšku třídy, je nezbytně nutné, abychom věděli přesně, co a jak máme použít. Z tohoto důvodu je nezbytně nutné dokumentovat to, co píšeme. Visual Studio má vestavěné nástroje, které nám v tomto ohledu pomůžou.

Před každou vlastnost, funkci nebo třídu můžeme vložit dokumentační komentář vložením tří za sebou jdoucích lomítek „///“. Po vložení /// Visual Studio vygeneruje šablonu dokumentačního komentáře, kterou prostě vyplníme. Tyto komentáře by měly být stručné a výstižné. Tyto komentáře budou kromě dokumentační činnosti přímo v kódu dané třídy sloužit i jako nápověda v okamžiku, kdy budeme chtít takto dokumentovanou třídu použít odjinud.

Třída Bod je velmi jednoduchá a proto vypisujeme většinou pouze <sumary> tedy shrnutí třídy nebo vlastnosti. Výjimkou je konstruktor třídy. Ten má parametry, které by měly být taktéž dokumentovány (direktiva param).

Zkusme rozebrat další objekt Vektor (úsečka, chcete-li). Zdrojový kód může vypadat následovně:

Mimochodem můžeme používat i jiné druhy komentářů. Dvě lomítka označí jako komentář zbytek řádku, a více řádkový komentář je označen /* na začátku a */ na konci.

```
/// <summary>
/// vektor obsahuje naprosto stejné datové složky jako bod. Liší se pouze
/// konstruktorem
/// </summary>
public class vektor
{
        #region promenne
        private double x, y, delka;
        #endregion

        #region vlastnosti
        /// <summary>
        /// X-ová souřadnice vektoru
        /// </summary>
        public double X
        {
```

```
        set
        {
                x = value;
                this.Delka = this.DelkaVektoru();
        }
        get{return x; }
}

/// <summary>
/// Y-ová souřadnice vektoru
/// </summary>
public double Y
{
        set
        {
                y = value;
                this.Delka = this.DelkaVektoru();
        }
        get{return y; }
}

/// <summary>
/// délka vektoru
/// </summary>
public double Delka
{
        set{delka = value; }
        get{return delka; }
}
#endregion

/// <summary>
/// Vektor je definován zadáním jeho jednotlivých datových složek
/// </summary>
/// <param name="x">datová složka X</param>
/// <param name="y">datová složka Y</param>
public vektor(double x, double y)
{
        X = x;
        Y = y;
        this.Delka = this.DelkaVektoru();
}

/// <summary>
/// Vektor je definován dvěma body
/// </summary>
/// <param name="p1">bod 1 vektoru</param>
/// <param name="p2">bod 2 vektoru</param>
public vektor(Bod p1, Bod p2)
{
        X = p2.X - p1.X;
        Y = p2.Y - p1.Y;
        this.Delka = this.DelkaVektoru();
}

/// <summary>
/// Provede výpočet délky vektoru na základě souřadnic vektoru nastavených
/// v kontruktoru třídy
/// </summary>
/// <returns>délku vektoru</returns>
```

```
        public double DelkaVektoru()
        {
                return Math.Sqrt(this.X * this.X + this.Y * this.Y);
        }

        /// <summary>
        /// Funkce provede normalizaci vektoru (každou složku podělí délkou vektoru
        /// </summary>
        public void normalizujVektor()
        {
                double d = this.Delka;
                this.X = this.X / d;
                this.Y = this.Y / d;
        }
}
```

Jak vidíte, zdrojový kód třídy vektor je poněkud delší a také složitější. Vlastnosti jsou zde tři – souřadnice X a Y směrnice vektoru a délka vektoru. Všimněte si, že vlastnosti souřadnic jsou využity k tomu, aby při změně souřadnic směrnice byla přepočítána délka vektoru.

Schopnosti objektu zpracovávat různé druhy parametrů v rámci stejně pojmenovaných funkcí říkáme **polymorfizmus**. Z vnejšku se totiž jeví, že je doustupná pouze jeden konstruktor (v tomto případě, jinak může být i funkce). Teprve na základě zadaných parametrů se rozhodne, která funkce se pro splnění úkolu použije.

Všimněte si také, že třída nemá jeden, ale hned dva konstruktory, které umožné definovat vektor buďto na základě znalosti souřadnic směrnice vektoru nebo na základě znalosti dvou bodů ohraničujících vektor.

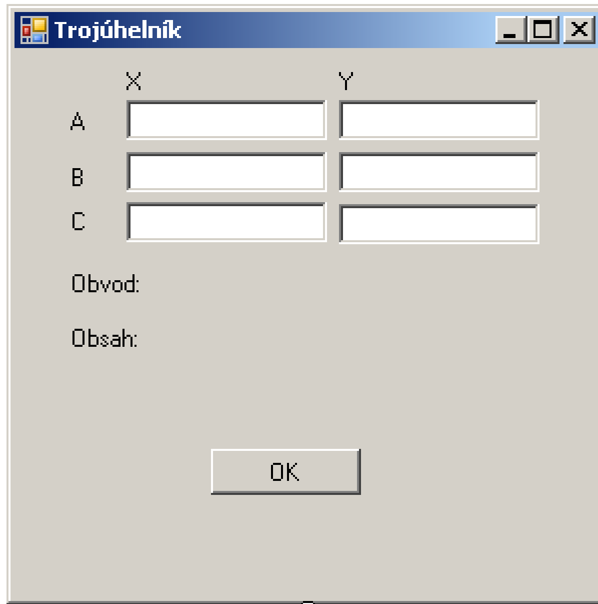
V třídě máme také dvě metody a to DelkaVektoru a normalizujVektor. Pro funkci nebo metodu vždy musíme nastavit návratovou hodnotu. V případě vélky vektoru to bude desetinné číslo (double), ale v případě normalizace vektoru chceme pouze transformovat stav třídy a proto nechceme, aby tato metoda něco vracela. Z tohoto důvodu nastavujeme typ návratové hodnoty na void (tedy prázdný).

V případě, že je návratová hodnota jiného typu než void, musí všechny cesty v rámci dané metody vracet nějakou hodnotu. V našem případě se délka vektoru počítá pouze podle jednoho vzorce – proto máme pouze jednu cestu, ale teoreticky bychom mohli určité hodnoty vyhodnocovat v podmínkách a používat vzorce různé.

```
/// <summary>
///  třída reprezentuje trojúhelník v prostoru
/// </summary>
public class Trojuhelnik
{
        private Bod b1, b2, b3;
        private vektor v1, v2, v3;

        /// <summary>
        /// první bod trojúhelníku
```

```
        /// </summary>
        public Bod A
        {
                set { b1 = value; }
                get { return b1; }
        }

        /// <summary>
        /// druhý bod trojúhelníku
        /// </summary>
        public Bod B
        {
                set { b2 = value; }
                get { return b2; }
        }

        /// <summary>
        /// třetí bod trojúhelníku
        /// </summary>
        public Bod C
        {
                set { b3 = value; }
                get { return b3; }
        }

        /// <summary>
        /// veřejný konstruktor provádějící konstrukci trojúhelníku na základě tří bodů
        /// </summary>
        /// <param name="a">bod A</param>
        /// <param name="b">bod B</param>
        /// <param name="c">bod C</param>
        public Trojuhelnik(Bod a, Bod b, Bod c)
        {
                this.A = a;
                this.B = b;
                this.C = c;
        }

        /// <summary>
        /// provede výpočet obvodu trojúhelníku
        /// </summary>
        /// <returns></returns>
        public double ObvodTrojuhelniku()
        {
                v1 = new vektor(A, B);
                v2 = new vektor(B, C);
                v3 = new vektor(C, A);
                return v1.Delka + v2.Delka + v3.Delka;
        }

        /// <summary>
        /// obsah trojúhelníku vypočtený pomocí Horneovy věty
        /// </summary>
        /// <returns>obsah trojúhelníku</returns>
        public double ObsahTrojuhelniku()
        {
                double s = ObvodTrojuhelniku() / 2;
                return Math.Sqrt(s * (s - v1.Delka) * (s - v2.Delka) * (s - v3.Delka));
        }
}
```

Pak už nám stačí zpracovat jednoduché GUI a budeme mít program hotový. Rozhraní by mohlo vypadat podobně jako na obr. 16.



Obr. 16: Návrh GUI pro výpočet obvodu a obsahu trojúhelníku

Zdrojový kód události po kliknutí na tlačítko OK by mohl vypadat následovně:

```
private void button1_Click(object sender, EventArgs e)
{
        Bod b1, b2, b3;
        try
        {
                b1 = new Bod(Convert.ToDouble(textBox1.Text),
        Convert.ToDouble(textBox2.Text));
                b2 = new Bod(Convert.ToDouble(textBox3.Text),
        Convert.ToDouble(textBox4.Text));
                b3 = new Bod(Convert.ToDouble(textBox5.Text),
        Convert.ToDouble(textBox6.Text));
        }
        catch
        {
                MessageBox.Show("zadávejte desetinná čísla!");
                return;
        }
        Trojuhelnik t = new Trojuhelnik(b1, b2, b3);
        label6.Text = "Obvod: " + t.ObvodTrojuhelniku();
        label7.Text = "Obsah: " + t.ObsahTrojuhelniku();
      }
```

Možná Vás napadne otázka, zda by nebylo jednodušší vypsat veškerý kód přímo do ovladačů událostí. Pro tento jednoduchý příklad by tomu tak možná skutečně bylo, ale i pro trochu složitější už ne.

Hlavní výhodou objektově orientovaného programování je to, že úkoly, které se snažíme v programu splnit můžeme rozdělit do úkolů menších, samostatných, ve kterých je menší pravděpodobnost, že uděláme chybu. Tyto samostatné objekty můžeme opakovaně využívat pro plnění úkolů složítějších. Vytváříme tak celé hierarchie objektů, které jsou schopny řešit i složité úkoly avšak zároveň zůstávají relativně přehledné.

Kontrolní úkoly:

Příklad, se kterým jsme pracovali, upravte tak, aby pracoval v 3D.

Zdrojové kódy příkladu:

Zdrojové kódy programu jsou dostupné v modulu Programovací jazyky na prometheus.vsb.cz v sekci příkladů program Trojúhelník. Zdrojáky v sobě neobsahují zapracované kontrolní úkoly.