Kapitola 1

Tvorba Windows Application

1.1 Základní myšlenky

Pokud chceme vytvořit Windows Application v C#, objeví se před námi okno rozdělené do tří sloupců. V pravém sloupci je *Solution Explorer*, v levém sloupci je *Toolbox* a v prostředním sloupci vidíme editor. Pokud trochu předběhneme, může se jednat buďto o editor textový nebo obrázkový. Postupujme opět příkladem a předpokládejme, že jsme vytvořili projekt jménem `Pokus1`, který je typu *Windows Application*.

Jelikož v editoru textovém se již vyznáme, věnujme se chvíli editoru obrázkovému. Do toho se dostaneme výběrem souboru *Form1.cs* v *Solution Exploreru*. Jelikož cílem textu je vytvořit čtenáři představu o tom, jak se taková aplikace vytváří a nechceme se zaměřovat na žádná pokročilejší témata (k čemuž je ideální referenční příručka popisující všechny atributy a metody jednotlivých tříd), nebudeme se zabývat tím, co všechno nám přesně nabízí *Toolbox* v levé části obrazovky. Vystačíme se základními znalostmi toho, že *Button* je tlačítko, *CheckBox* je zaškrtávací okénko (jímž odpovídáme na otázku typu Ano/Ne), *Label* je popisek (kus textu), významný pro nás bude ještě *Progress Bar*, což je proužek ukazující, v jaké fázi čekání na něco jsme (kupříkladu při instalaci, kdy máme třeba hodinu čekat a nejsme si jisti, zda výpočet nezatuhl) a hlavně *TextBox* (tedy textové okénko, do kterého může program nebo uživatel zapisovat text). V *Designeru* si navrhneme vzhled okna, který budeme po aplikaci požadovat.

Zde předpokládáme, že čtenář již ví, co to jsou objekty, že mají vlastnosti (či atributy) a metody. Nyní je důležité si uvědomit, že Windows Application se vytváří jako *program řízený událostmi, tedy se program nemusí* agilně *starat,* co se mohlo stát, *ale je* naopak *upozorňován* a pouze na to musí reagovat. Základním stavem je tedy stav nejpřirozenější, kdy program čeká, až uživatel

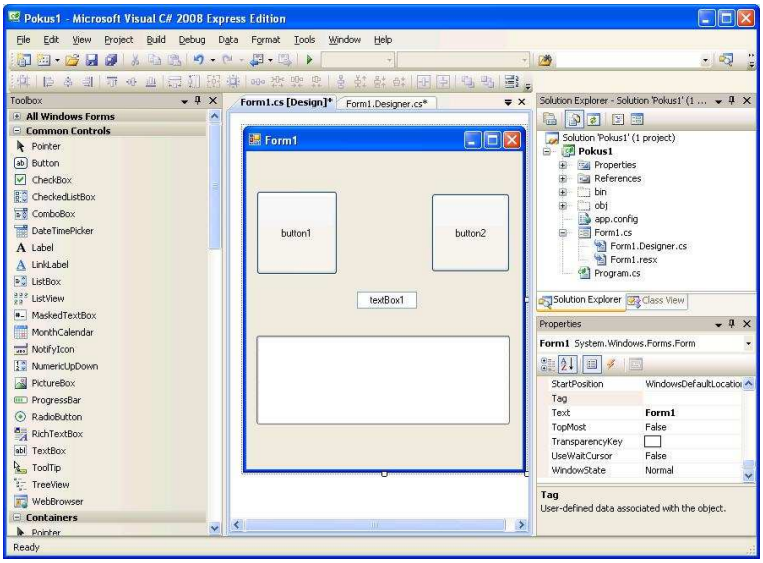
začne něco chtít. V takové situaci jsme v konsolové aplikaci čekali zpravidla, až uživatel stiskne klávesu nebo něco napíše. V aplikaci řízené událostmi za nás čeká program a ukazuje uživateli okno s různými tlačítky, menu, textovými políčky, checkboxy a vším možným. Pokud se uživatel rozhodne kupříkladu stisknout tlačítko, systém nám toto oznámí zavoláním metody `click` u příslušného tlačítka. Na nás potom je, abychom zajistili odpovídající reakci. Reakce na stisknutí tlačítka tedy bude definována v metodě *click* pro příslušné tlačítko. Jako reakci můžeme například chtít něco vypsat do vhodného textového políčka. To uděláme tak, že dotyčnému textovému políčku změníme atribut `Text`. Práci tedy máme s překladačem důvtipně rozdělenu tak, že aplikační interface nás bombarduje voláním metod jednotlivých objektů (podle událostí, které nastaly) a porůznu nám poskytuje podrobnosti v atributech vybraných objektů. My na to obecně reagujeme voláními dalších metod, nebo aspoň nastavením atributů nějakých objektů. Přitom platí, že je-li nějak hodnota atributu jistého ovládacího prvku změněna, musí se toto reflektovat ve vzhledu okna vůči uživateli. Takže například uživatel může zapsat do textového políčka a my si z textového políčka můžeme tuto hodnotu přečíst (atribut `Text`). Na druhé straně když my (jakožto program) zapíšeme pod atribut `Text` v textovém políčku, objeví se tento náš názor v dotyčném textovém políčku uživateli. Pomocí atributů jednotlivých prvků můžeme nastavovat různé jejich vlastnosti (například kdo do textového políčka může zapisovat – zda jen my, nebo i uživatel, což je ovládáno boolským atributem `ReadOnly`).

1.1.1 Zatloukání hřebíku

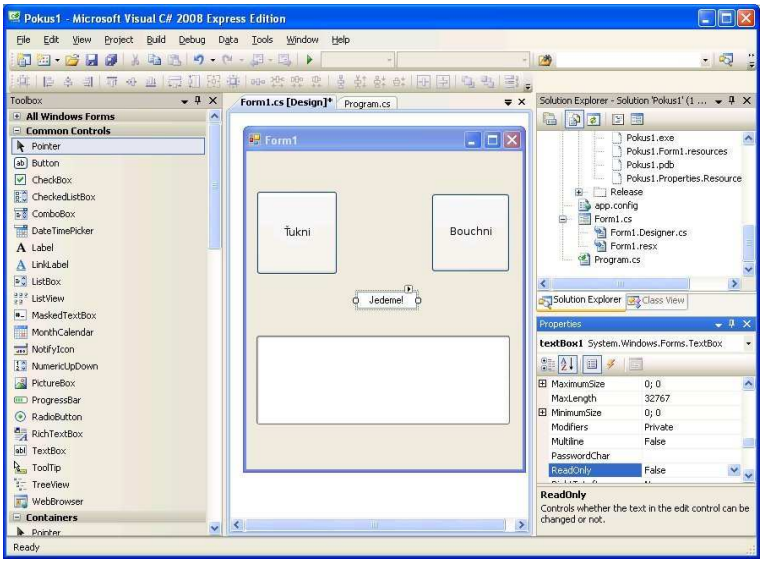
Jako první (neužitečný leč instruktivní) příklad si udělejme aplikaci nazvanou *zatloukání hřebíku*. Zatloukáme-li hřebík, obvykle vezmeme kladivo a tím buďto zlehka ťukneme (aby to neodnesly prsty) nebo pořádně bacíme. Pro účely jednoduchosti od držení hřebíku a možnosti zranění odhlédneme a soustředíme se na to, že při ťuknutí hřebík zajede do zdi asi o 10 procent, kdežto při pořádném bouchnutí asi o pětinu. Tedy vyrobíme okno se dvěma tlačítky (řídícími ťukání resp. bouchání kladivem), jedním *progress barem* (ukazujícím, o kolik již hřebík zajel do zdi) a jedním textovým oknem komentujícím situaci. Program nebude dělat nic jiného, než že po stisknutí jednoho tlačítka *progress bar* poskočí o deset procent dále a po stisku druhého tlačítka o dvacet procent. Nakreslíme tedy formulář jako je na obrázku 1.1.

Jelikož se nám původní generické popisky prvků nelíbí, v okně *Properties* (v dolní polovině pravé třetiny) přenastavujeme tlačítkům 1 a 2 a textovému poli atribut `Text` na jiná slova viz obrázek 1.2.

Nyní máme popsáno, jak má vypadat formulář našeho okna. Teď ještě



Obrázek 1.1: Příklad projektu – na formuláři jsou dvě tlačítka, textové políčko a progress bar.



Obrázek 1.2: Oproti minulému obrázku jsme přejmenovali prvky.

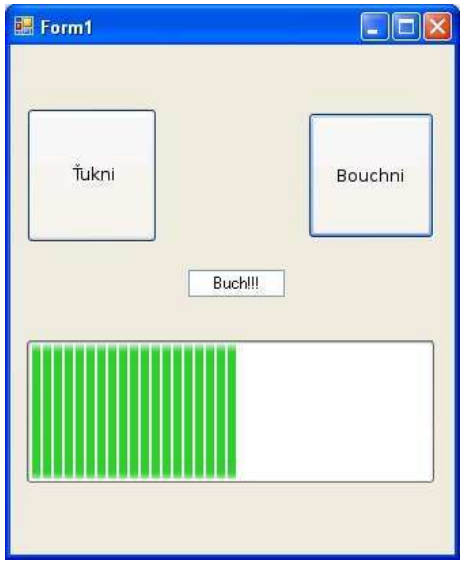
zbývá naprogramovat příslušné reakce, tedy říct, co se má stát, pokud je stisknuto jedno, resp. druhé tlačítko. Za tím účelem v okně *Properties* (vpravo dole) clickneme na ikonu *Events* vyznačenou žlutým blesčíkem. Najdeme si událost `click`. Nyní můžeme stanovit jméno funkce (handleru) ovládající tuto událost. Pokud nemáme v tomto ohledu zvláštní přání, můžeme si dvojclicknutím na položku *click* v okně *Events* nechat vygenerovat handler s kanonickým jménem sestávajícím z názvu objektu, podtržítka a názvu události. V našem případě se tedy automaticky vygenerovaná metoda bude jmenovat `button1_Click` a po dvojclicknutí na jméno metody se nám v prostředním sloupci místo formuláře rovnou objeví textový editor s předepsanou funkcí `button1_Click`, kterou chceme doplnit. Řekli jsme si, že chceme posunout výpočet o deset procent, tedy nastavit na *progress baru* o deset procent více a zkontrolovat, zda nejsme na aspoň sto procentech výpočtu (a v takovém případě program ukončit) a ještě chceme (z dlouhé chvíle) do textového políčka clicknutí na tlačítko okomentovat slovem „ťuk". Toho dosáhneme tímto kódem:

```
private void button1_Click(object sender, EventArgs e)
{
    progressBar1.Increment(10); // Posuň progressBar1 o 10
    textBox1.Text = "ťuk!"; // Do textBox1 napiš "ťuk!"
//    check(); //zavolej funkci check()
}
```

Tím jsme pořídili téměř vše – tedy vše, kromě testu, zda výpočet ještě neukončit. Jelikož totéž budeme chtít udělat i po stisku druhého tlačítka, napíšeme si na to funkci `check`, jejíž tělo umístíme někam před definici funkce `button1_Click()`, tedy zdrojový text modifikujeme takto:

```
void check()
{
    if (progressBar1.Value >= 100)
        Application.Exit();
}

private void button1_Click(object sender, EventArgs e)
{
    progressBar1.Increment(10); // Posun progressBar1 o 10
    textBox1.Text = "ťuk!"; // Do textBox1 napiš "ťuk!"
    check(); //zavolej funkci check()
}
```



Obrázek 1.3: Ukázka běžící aplikace.

Funkce `check` (která nebere žádné argumenty a ani nic nevrací) se podívá, zda `progressBar1` nemá hodnotu aspoň sto (což je implicitní hodnota `progressBar1.Maximum`). Pokud ano, zavolá funkci `Application.Exit`, která podle očekávání ukončí aplikaci.

Nyní budeme chtít přidat obsluhu události click i pro `button2`. Jedna z možností jak toho dosáhnout je vybrat v prostředním sloupci Studia tab `Form1.cs[Design]` a ve formuláři clicknout na `button2`, v okně *Properties* dvojclicknout na `button2_Click` a v textovém editoru definovat:

```
private void button2_Click(object sender, EventArgs e)
{
    progressBar1.Increment(20);
    textBox1.Text = "Buch!!!";
    check();
}
```

Nyní můžeme aplikaci zkompilovat (*Build* → *Build Solution* nebo F6), spustit (*Debug* → *Start without debugging* nebo Ctrl+F5), spustit s debugováním (*Debug* → *Start* či F5 a následně krokovat). Objeví se nám okno podobné tomu v *Builderu*, které nám umožní ťukat na dvě tlačítka a které bude vypadat (po chvíli snažení s myší) asi tak, jak vidíme na obrázku 1.3.

Až progress bar dojede na konec (nebo za konec), okno se samo zavře.

1.1.2 Cvičení

1. V uvedeném příkladu si hrajte s atributy jednotlivých objektů a pozorujte, jaké změny v aplikaci uděláte. Pokud nastavíte *progress baru* styl *Marquee*, program přestane fungovat, protože *progress baru* stylu *Marquee* nemá smysl volat metodu `Increment`!

2. Vyrobte „jednoduchou" kalkulačku, tedy okno se čtyřmi textovými poli a jedním tlačítkem. Do prvních tří textových polí zadáte číslo, operátor a číslo, operátor může být +, -, *, nebo /, stisknete tlačítko a program do čtvrtého textového pole vypíše výsledek aritmetického výrazu. Bude-li v políčku operátoru znak **x**, program se ukončí. Zajistěte, aby uživatel nemohl zapisovat do políčka na výsledek.
   Pozor na dělení nulou!

3. * Vyrobte „normální" kalkulačku, tedy kalkulačku s displayem a tlačítky na číslice a na operátory (viz Programy → Příslušenství → Kalkulačka) a samozřejmě implementujte její funkčnost.

4. ** Implementujte v příkladu 3 přepínač „Vědecká kalkulačka/Normální kalkulačka", který bude zobrazovat (resp. skrývat) tlačítka vybraných matematických funkcí.

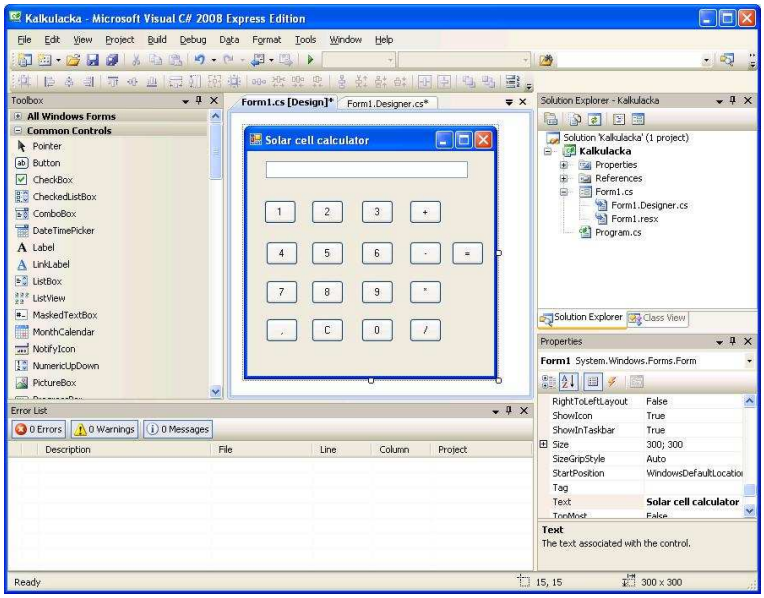
Návod:

- V úloze 3 bude asi nejhorší definovat pro každé tlačítko speciální handler události `click`. Definujeme proto jen jeden handler, který použijeme pro všechna číselná tlačítka a které tlačítko akci vyvolalo, zjistíme podle argumentu `sender`.

- V úloze 4 doporučujeme naklikat všechna tlačítka a řídit jejich zobrazování atributem `Visible` jednotlivých tlačítek.

1.2 Další aspekty a ovládací prvky C#

V této sekci si ukážeme další prvky a možnosti. Ne náhodou se budeme věnovat příkladům zmíněným jako cvičení k minulé sekci. Jako první si vyřešíme úlohu s obyčejnou kalkulačkou. Na to navážeme naznačením, jak jsme si představovali řešení vědecké kalkulačky. Na závěr si vyřešíme mírné rozšíření úlohy s jednoduchou kalkulačkou.



Obrázek 1.4: Formulář obyčejné kalkulačky.

1.2.1 Obyčejná kalkulačka

Každého asi napadlo, že úloha s obyčejnou kalkulačkou měla být zejména zdlouhavá, pokud jde o klikání formuláře. Tlačítka můžeme naklikat hrubou silou, nebo vyrobit uvnitř programu (obojí má své nevýhody, nevýhodou prvního je ztráta času nad klikáním, nevýhoda druhého je podobná, člověk umísťuje tlačítka bez kontaktu s formulářem). Každopádně pokud klikáme více než dvě tlačítka, je třeba dát pozor, zda jsme na nějaké nezapomněli (například já jsem zapomněl na nulu a její absenci jsem zpozoroval až po rozšíření na vědeckou kalkulačku, proto můj formulář má trochu avantgardní layout). Na jednoduché kalkulačce se zřejmě objeví číslice *1 – 9*, operace *sčítání, odčítání, násobení* a *dělení, rovnítko, desetinná čárka*, tlačítko *C* (které vrátí výpočet do stavu, v jakém byl na začátku) a ti bystřejší nezapomenou ještě číslici *0*. Tlačítka můžeme na formuláři kopírovat (Ctrl+C a Ctrl+V), což má tu výhodu, že tlačítka budou aspoň stejně velká. Přitom můžeme s výhodou využít toho, že atributy se kopírují (metody bohužel ne). Naklikáme tedy formulář na způsob obrázku 1.4. Již víme, že tlačítka pojmenujeme pomocí atributu `Text`, stejně přejmenujeme i celé okno.

Nyní je na čase se zamyslet a navrhnout způsob, kterým se neudřeme a dosáhneme cíle. Asi je jasné, že mluvím o návrhu (implementaci) metod `click` pro jednotlivá tlačítka. Tlačítka *C* a *desetinná čárka* mají velmi specifickou funkci, pro ty tedy napíšeme ovladače zvlášť, ale číslice (kterých je

deset) dělají vlastně všechny to samé, podobně operace +, −, ∗, / a = dělají to samé, bylo by tudíž hloupé psát pro každé tlačítko zvláštní handler. Pak bychom (pro čísla, kde situace nejvíc bije do očí) měli deset funkcí, které by se lišily jen v jedné konstantě.

Takhle tedy ne, svého času si vážíme. Napíšeme proto pro všechny číslovky společný handler a pojmenujeme jej například `cislo`. Handler se totiž volá se dvěma argumenty, jak snadno zjistíme pohledem na Studiem předpřipravený prototyp:

```
private void cislo(object sender, EventArgs e)
```

Od druhého argumentu (`EventArgs`) v tento moment odhlédneme, pro nás bude zajímavý argument `sender`. Ten odkazuje k objektu, kterému právě obsluhujeme událost. Funkce `cislo` tedy bude obsluhovat událost `click` deseti tlačítkům. Podle obsahu proměnné `sender` zjistíme, kdo nás vyvolal. Povšimněme si ovšem, že `sender` je typu `object`, což nevěští mnoho dobrého, jelikož jde o velmi obecný typ a nemůžeme tak spekulovat na jeho metody a atributy. My ovšem budeme opatrní a budeme vždy volat funkci `cislo` z objektu typu `Button`. Můžeme si proto vynutit přetypování sendera na tento již podstatně užitečnější typ a využívat veškerých jeho atributů. K tomu účelu definujeme na počátku funkce `cislo` proměnnou `pom` a takto zinicializujeme:

```
Button pom = (Button)sender;
```

nebo

```
Button pom = sender as Button;
```

Druhá varianta je výhodnější v tom, co se stane, pokud `sender` nelze na typ `Button` přetypovat (v prvním případě program spadne, ve druhém dostaneme `null`).

Nyní zbývá poměrně snadná úloha určit za pomoci proměnné `pom`, které tlačítko nás zavolalo. K tomu můžeme využít kupříkladu proměnné `Tag`, která nám padne do oka. Ponastavujeme tedy všem tlačítkům proměnnou `Tag` na hodnotu shodnou s atributem `Text` (ponechme stranou, že bychom mohli použít přímo proměnnou `Text`, jak tomu uděláme při dalších příležitostech, jednou jsme při psaní udělali (špatné) rozhodnutí a pokusem o jeho nápravu bychom nadělali víc škody než užitku, tedy zavlekli bychom do textu množství chyb – změnu udělám až v další verzi textu).

Atribut `Tag` je příjemný tím, že zatímco atribut `Text` by měl dávat uživateli smysl, atribut `Tag` si nastavujeme „pro sebe", tedy si do něj můžeme dát hodnoty podstatně příjemnější, než dáváme do atributu `Text`. Můžeme tedy nyní poměřovat `pom.Tag` se známými údaji. Jedná se ovšem o objekt, který musíme zkonvertovat nejlépe na `String`, budeme se tedy dovolávat hodnoty funkce `pom.Tag.ToString()`. Tuto hodnotu můžeme buďto poměřovat na rovnost řetězců, nebo zkonvertovat na číslo pomocí `Convert.ToDouble()`. Jelikož se jedná o čísla, s výhodou využijeme té druhé možnosti.

Zbývá ještě drobnost, leč zásadní: Jakým způsobem budeme reprezentovat „stav" kalkulačky? Globální proměnné nemáme, proto navrhuji vyrobit třídu, která bude obsahovat všechny důležité proměnné a instanci této třídy si vyrobit při startu programu. Třídu definujeme kupříkladu na konec souboru `Form1.cs` (ne před třídu `Form1`, to by nás Studio prohnalo, že třída `Form1` není první). Třídu pojmenujeme třeba `Data` a definujeme takto:

```
public class Data
{
    public int poslop,cislic;
    public double minule,soucasne;
    public bool tecka;//byla uz tecka?
    public Data()
    {
        soucasne=minule = poslop = cislic = 0;
        tecka = false;
    }
}
```

Proměnná `soucasne` bude určovat číslo, které právě načítáme (nebo jsme načetli jako poslední), proměnná `minule` bude udávat číslo, které jsme načetli jako předposlední, nebo které vyplynulo jako výsledek nějaké operace, proměnná `tecka` určuje, jestli už byla desetinná čárka. Proměnná `cislic` určuje počet již prošlých desetinných číslic (pokud `tecka==true`). V `poslop` zakódujeme poslední zadanou operaci.

Vyhodnocování čísel je jasné (budeme jen upravovat obsah proměnné `soucasne`. Dokud nepřijde desetinná čárka, dosavadní hodnotu vynásobíme deseti a přičteme hodnotu `pom.Tag`. Pokud již desetinná čárka prošla, přičteme k `soucasne` hodnotu posunutou správným způsobem za desetinnou čárku.

Složitější bude situace s prováděním operací. Trochu předběhneme a řekneme si, že *přijde-li operátor,* vždycky *provedeme předchozí operaci* a operátor, který právě přišel, si jen poznamenáme a při příchodu dalšího operátoru bude operátor „minulý" a jeho kód tak najdeme v proměnné `poslop`. Abychom se nemuseli zdržovat detaily, handler události `click` u desetinné čárky jen nastaví proměnnou `tecka` na `true` a nechá vypsat do `textBox1` hodnotu `soucasne.ToString()+","`. Desetinnou tečku samotnou bychom sice vypisovat nemuseli, ale pak bychom se divili, proč se nám za číslem objeví desetinná čárka až po stisku první desetinné číslice.

Handler tlačítka *C* pak neudělá nic jiného, než totéž, čím instanci třídy `Data` inicializujeme, tedy povšimněte si, že při popsaném způsobu údržby

kalkulačky je začátek ekvivalentní tomu, kdybychom na počátku naťukali *"0" "+"*.

Jednotlivé operace ($+, -, *, /$ a $=$) zakódujeme do malých integerů takto: Nula bude kódem sčítání (proto v konstruktoru třídy `Data` vše inicializujeme na nulu a proto, že desetinná tečka ještě neprošla, inicializujeme `tecka` na `false`). Odčítání bude mít kód 1, a tak dále. K uložení těchto kódů použijeme atribut `Tag`.

Jak jsem se již zmínil, abychom ve třídě `Form1` získali proměnnou jménem `data` typu `Data` a to řádně zinicializovanou, přidáme na začátek definice třídy `Form1` definici proměnné `data` a v konstruktoru třídy `Form1` proměnnou `data` zinicializujeme asi takto:

```
namespace Kalkulacka
{   public partial class Form1 : Form
    {
        Data data; //Objekt obsahující natukane udaje
        public Form1()
        {
            data = new Data(); // Vyrobime nove datove uloziste
            InitializeComponent();
        }
```

Okomentované řádky jsou přidané, ostatní jsou původní (beze změn). Pokračujeme tedy již dlouho toužebně očekávaným handlerem `cislo` řídícím kliknutí na číselné tlačítko:

```
private void cislo(object sender, EventArgs e)
{
    Button pom = (Button)sender;// Sender je vzdy typu Button,
                                // jinak se tady zacnou dit veci
    switch(data.tecka)
    {   case true: //Pridavame za desetinnou carku
            data.soucasne = data.soucasne +
                      Convert.ToDouble(pom.Tag.ToString()) /
                      Math.Pow(10,++data.cislic);
        break;
        case false: //Pridavame pred desetinnou carku
            data.soucasne=10*data.soucasne+
                      Convert.ToDouble(pom.Tag.ToString()));
        break;
    };
    textBox1.Text = Convert.ToString(data.soucasne);
```

```
    //Vypiseme novy obsah "data.soucasne".
}
```

Vidíme, že ovladač události pro deset tlačítek jsme pořídili na 14 řádek včetně jednoho prázdného.

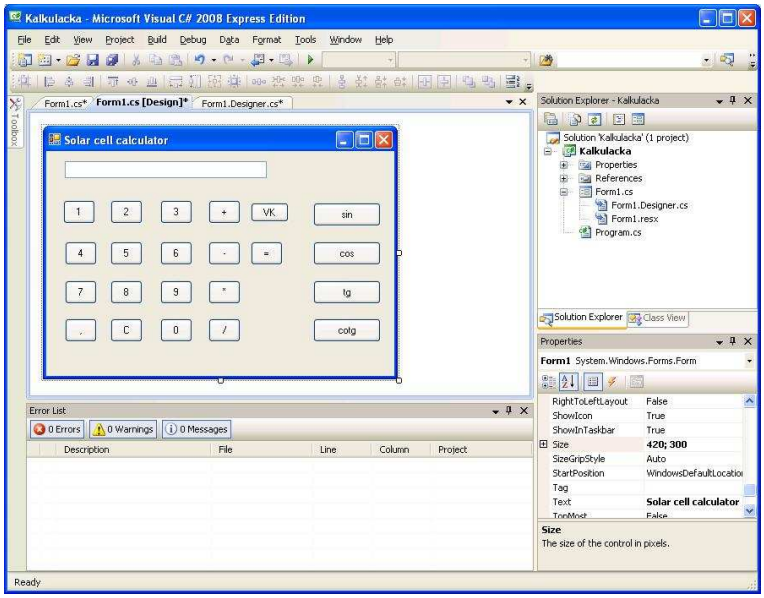
Handlery pro tlačítka `,` a `C` jsou naprosto nezajímavé, pokud se ale někdo již zvládl v našem příkladu ztratit, mohou vypadat třeba takto:

```
private void clear(object sender,EventArgs e)//Reset kalkulacky
{
    data.soucasne = data.minule = data.poslop = data.cislic =0;
    data.tecka = false;
    textBox1.Text = Convert.ToString(data.minule);
}

private void puntik(object sender,EventArgs e)//Desetinna carka
{
    data.tecka = true;
    textBox1.Text = Convert.ToString(data.soucasne) + ',';
}
```

Druhou (a poslední) zajímavou částí příkladu je handler události `click` pro tlačítka popisující jednotlivé aritmetické operace. Jelikož aritmetická operace, kterou budeme při stisku „operačního tlačítka" naprosto nezávisí na tom, jakou operaci jsme „objednali" teď, ale co jsme chtěli dělat minule, bylo by úplnou hloupostí definovat pro každou operaci handler zvlášť, uděláme tedy podobný trik, tentokrát opět použijeme hodnotu v atributu `Tag` a tento atribut nastavíme tak, aby tlačítko pro sčítání mělo `Tag` nula, tlačítko pro odčítání jedna, násobení dvě, dělení tři a rovnítko čtyři. Funkci obsluhující všechny aritmetické operace pojmenujeme kupříkladu `operace` a můžeme ji implementovat třeba takto:

```
private void operace(object sender, EventArgs e)
{   Button b=(Button)sender; //Opet zjistime, kdo nas vola
    switch(data.poslop) //Co udelame zavisi na minule operaci!!
    {   //0 je kod scitani:
        case 0: data.minule+=data.soucasne;
            break;
        case 1: data.minule-=data.soucasne;
            break;
        case 2: data.minule*=data.soucasne;
            break;
```



Obrázek 1.5: Formulář vědecké kalkulačky. V pravé části funkce *sinus, cosinus, tangens* a *cotangens*. Jejich buttony mají atribut `Visible` nastavený na `false`. Tlačítko `VK` ve svém handleru události `click` atribut `Visible` goniometrickým funkcím přenastaví. Zde je ale demonstrováno, jak změnit velikost formuláře. Buďto zatáhneme za okraj, nebo formuláři nastavíme atribut `Size`.

```
        case 3: data.minule/=data.soucasne;
            break;
        case 4: data.minule=data.soucasne;
            break;
    }
    data.soucasne=data.cislic=0; //Zahajujeme nové číslo...
    data.tecka = false; //...které je zatím celé.
    data.poslop=Convert.ToInt32(b.Tag);
                //Zapamatujeme si současnou operaci na příště
    textBox1.Text = Convert.ToString(data.minule);
                //Vypíšeme výsledek na display
}
```

Důležité je nepojmenovat (resp. nepokusit se pojmenovat) handler `operator`, protože to už je v C# klíčové slovo!

Diskuse

Předvedené řešení je jednou z možností, jak problém řešit. Samozřejmě by bylo možno provést mnoho modifikací. Například třídu `Data` nebylo nutné tvořit, jednotlivé prvky jsme mohli vložit přímo do třídy `Form`. Anebo jsme mohli tuto třídu udělat statickou se všemi atributy statickými a netvořit její instanci. Pak bychom získali konstrukci blízkou globálním proměnným, které nám v C# mohou (a také budou) chybět.

Taktéž jsme operace (sčítání, odčítání…) nemuseli indexovat celými čísly, mohli jsme je indexovat znakem (určujícím přímo dotyčnou operaci).

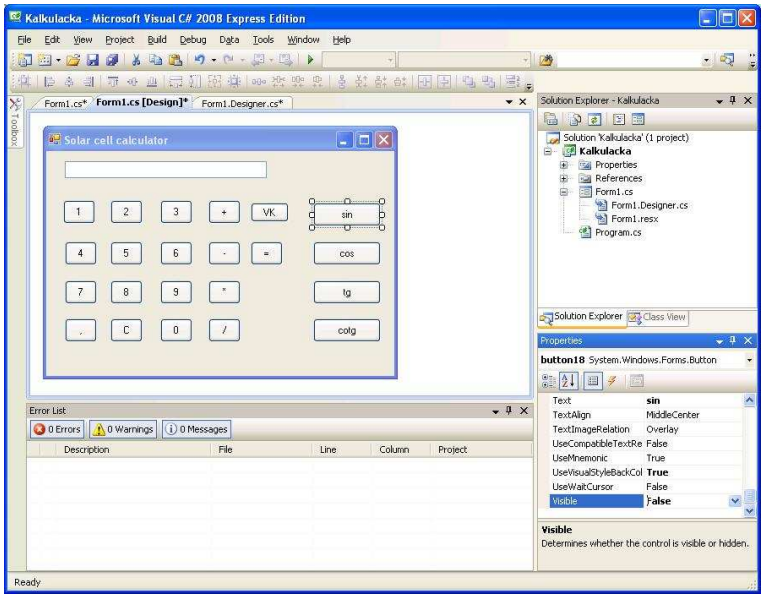
Nakonec jste si jistě povšimli přiřazení (inicializace nulou) na počátku funkce `clear`, kdy jsme přiřadili rovnou do několika proměnných najednou nulu. Mohlo by se stát, že toto přiřazení dopadne úplně jinak, než jsme si představovali, ovšem pouze za předpokladu, že by jednotlivé proměnné nebyly všechny stejného typu. V našem případě ale jde o samé integery, proto je vše v pořádku. Kdyby tomu tak nebylo, překladač by začal nulu přetypovávat a při dostatečně divokých konverzích by nula mohla přejít na jinou hodnotu, je proto potřeba dávat pozor!

1.2.2 Vědecká kalkulačka

Pod krycím označením *Vědecká kalkulačka* se skrývá aplikace, která pro začátek vypadá jako základní kalkulačka, je však osazena tajemným tlačítkem *VK*, po jehož stisknutí se vzhled okna zcela změní, veškerá tlačítka se přetvoří a místo osmnácti jich bude mnoho. Tlačítko *VK* se přemění na tlačítko *NK* (jako *Nevědecká kalkulačka*). Jde tedy zejména o to, jak nechávat prvky na formuláři objevovat se a mizet. Z technických důvodů se omezíme pouze na to, že po stisku tlačítka *VK* přibydou tlačítka *sin*, *cos*, *tg* a *cotg* počítající (v tomto pořadí) sinus, cosinus, tangens a cotangens. Tlačítko *VK* se přejmenuje na *NK* a po jeho (sudém) stisknutí tato nová tlačítka zmizí. Jakožto správní mastňáci implementujeme goniometrické funkce ve stupních, zatímco v C# jsou ve třídě `Math` implementovány v radiánech.

Formulář kalkulačky tedy upravíme jako na obrázku 1.5. Tedy zejména formulář poněkud natáhneme do šířky (buďto potahováním za okraje, nebo přenastavením atributu `Size`. Kromě popisky (atribut `Text`) nastavíme nová tlačítka jako neviditelná nastavením atributu `Visible` na `false` (viz 1.6). Až nečekaně jednoduché – že? A tlačítka se po startu neobjeví. Aby se objevila po stisku tlačítka *VK*, přidělíme mu takovýto handler události `click`:

```
private void button17_Click(object sender, EventArgs e)
{
    if (button17.Text == "VK")//Prehazujeme na vedeckou:
```



Obrázek 1.6: Formulář *vědecké kalkulačky*. Nastavíme-li *Buttonu* atribut `Visible` na `false`, *Button* se po spuštění programu na formuláři neobjeví.

```
    {
        button18.Visible = true;
        button19.Visible = true;
        button20.Visible = true;
        button21.Visible = true;
        button17.Text = "NK";//jako Nevedecka Kalk...
    }
    else
    { // Prehazujeme na "nevedeckou" kalkulacku...
        button18.Visible=button19.Visible=button20.Visible=
                                    button21.Visible = false;
        button17.Text = "VK";
    }

}
```

Cvičení

1. K dokonalosti programu nyní zbývá drobnost: Implementovat handler (opět společný) pro jednotlivé goniometrické funkce. Proveďte.

2. Hrajte si s kalkulačkou a zkoušejte přenastavovat další atributy. Pozorujte, co příslušné atributy mění, všimněte si, jaké všechny atributy kterých objektů máte k dispozici a dejte pozor na to, které atributy můžete beztrestně měnit a které jsou z nějakého důvodu pouze pro čtení.

1.2.3 Jednoduchá kalkulačka

Jako další příklad si uděláme *jednoduchou kalkulačku*, tedy na formulář rozmístíme dvě textová políčka, do kterých zapíšeme čísla, ovládací prvek, kterým vybereme, zda budeme sčítat, odčíat, násobit nebo dělit, tlačítko *Počítej!* a výstupní prvek, kam oznámíme výsledek. Tento příklad jsme nechali až na teď, jelikož na něm zkusíme demonstrovat co nejvíce ovládacích prvků. Místo jednoho prvku, kterým vybereme, totiž vyrobíme současně několik prvků a každý z nich bude mít vlastní tlačítko *Počítej!*, které provede dotyčný aritmetický úkon s ohledem na vybraný ovládací prvek. Ovládací prvky, které si ukážeme v tomto příkladu budou: *menu, listBox, comboBox* a *radioButton*. K tomu si ukážeme ještě prvek *Label*, který je ovšem prvkem výstupním (ostatní prvky, které také použijeme, již známe).

- *ListBox* je prvek sestávající z textového políčka a šipek nahoru a dolů. Šipky mění obsah políčka. Obsahem nemusí být jen čísla, v našem případě to budou aritmetické operace. Je poměrně podobný prvku následujícímu prvku:
- *ComboBox* se chová jako meníčko, ze kterého vybíráme.
- *RadioButton* je pak prvek ukazující všechny volby rozložené vedle sebe (nebo nad sebou), u každé položky je zaklikávací kotouček.

Vytvoříme formulář podobný tomu na obrázku 1.7. Popis zahájíme od shora. Nahoře vidíme *menu*. To nyní přeskočíme a vrátíme se k němu později. Pod ním vidíme dobře známé *textBoxy*. Jejich popisky jsou typu *Label*, na kterých není nic komplikovaného, chovají se jako *textBox*, do kterého ovšem uživatel nemůže zapisovat (a nejsou viditelně olemovány). Úplně vespod jsou dva další *Labely*, ze kterých je vidět ovšem jen jeden (*Výsledek je:*), ten hned vedle něj má prázdný `Text`, proto není vidět. Mezi *textBoxy* a dolními *Labely* jsou při pravé straně tři tlačítka (typu *Button*), která rovněž již velmi dobře známe a zbývá *ListBox* (i s popiskou typu *Label*), *ComboBox* (taktéž s popiskou) a *RadioButton* (pro prostorovou složitost bez popisky) v tomto pořadí shora dolů.

RadioButton se ovládá odlišně od ostatních dvou, proto jím začneme. *RadioButton* tvoříme jako čtyři různé prvky *radioButton1* až *radioButton4*.

Generickou popisku mu změníme na znaménka. *ListBox* a *ComboBox* se používají tak, že je naklikáme a v panelu *Properties* si u nich najdeme položku *Items*. Jedná se o pole hodnot, které zadáváme (po clicknutí na trojtečku za poznámkou *(Collection)*) každou hodnotu na jeden řádek. Nyní bez vysvětlení vyzvu k zadání hodnoty pro odečítání vždy jako "- ", tedy znak minus a mezeru. V tomto případě mezera za minusem roli nehraje, roli začne hrát v dalším prvku (tedy v *menu*), ale tato drobnost nám pak usnadní práci.

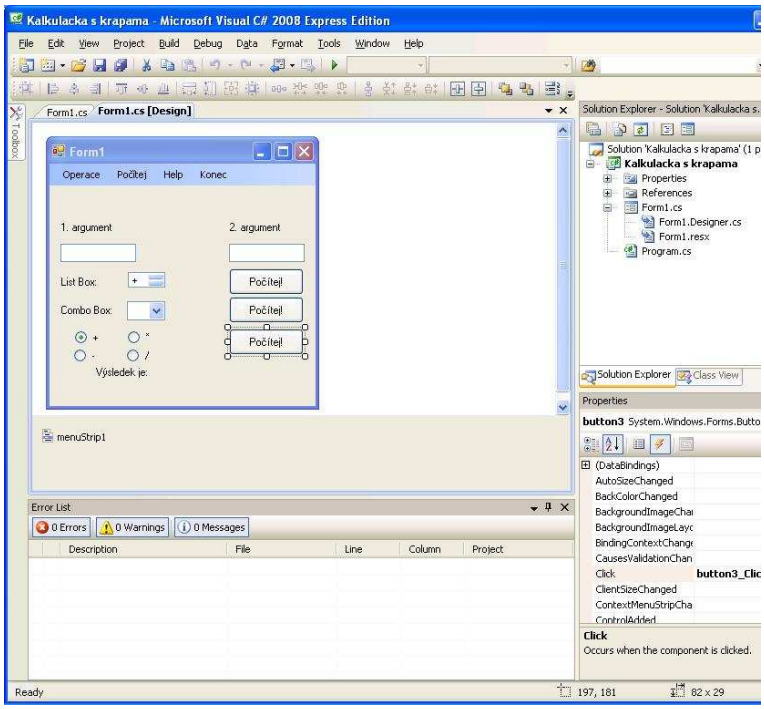
Menu naklikáme podobně jako všechny ostatní prvky, tedy vybereme si v *Toolboxu* v sekci *Menus & Toolbars* položku *MenuStrip*. Postupně vyplňujeme jednotlivá menu. První položku pojmenujeme *Operace* a vytvoříme jí čtyři podpoložky. Ostatním položkám podmenu dělat nebudeme, tedy prvek *Počítej* je ve své roletě sám, stejně tak *Help* a *Konec*. Prvkům, na kterých je zajímavá operace clicknutí myší, je třeba ponastavovat metodu *click*, toto se týká zejména menu (resp. jednotlivých jeho položek). Začneme tím nejjednodušším, a to handlerem *click* pro položku menu *Konec*. Tu definujeme například takto:

```
private void konecToolStripMenuItem_Click(object s,EventArgs e)
{
    Application.Exit();
}
```

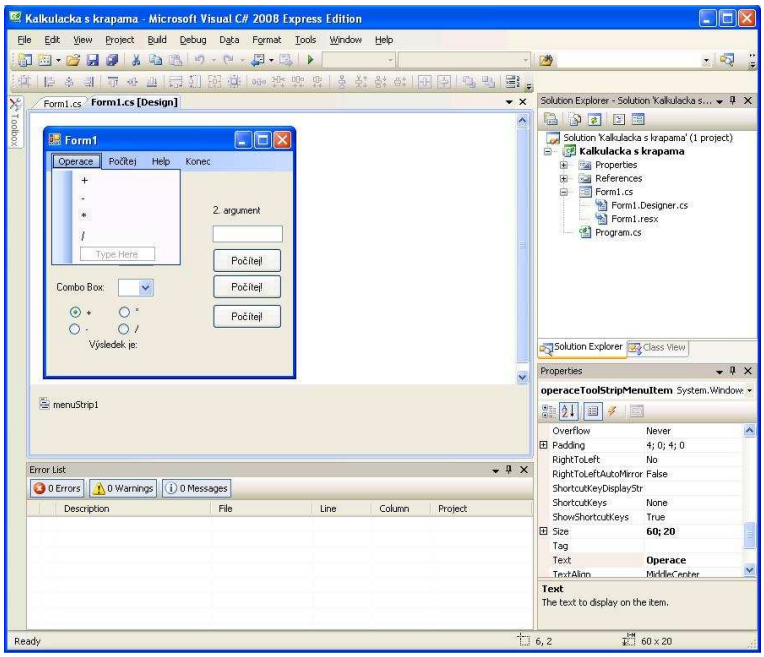
Název metody je automaticky vygenerovaný. Není možné položku v menu pojmenovat "-" (to vyústí ve vodorovnou čáru napříč přes menu), ovšem název "- " už být může). Jelikož nakonec ve všech případech využijeme funkci vyhodnocující výraz pomocí výběru v menu, nastavili jsme obsah *ListBoxu* a *ComboBoxu* pri odčítání taktéž na "- " (ačkoliv to zatím může vypadat jen jako vrtoch). Důvod se objeví až při psaní metod vyhodnocujících výraz.

Nyní si na handlerech události `click` u tlačítek *Počítej*, příslušných k jednotlivým ovládacím prvkům (resp. na položce *Počítej* v menu) ukážeme, jak se přistupuje k položkám, které si uživatel vybral v *ListBoxu, ComboBoxu, RadioButtonu* a *menu*. K *ListBoxu* a *ComboBoxu* přistupujeme velmi podobně. Oba mají atribut *SelectedItem*, ve kterém se najde ten z řetězců, který si uživatel vybral. Výběru lze i „napomoci" například inicializací proměnné *SelectedItem* v konstruktoru třídy formuláře (abychom nemuseli řešit případ, že uživatel nic nevybral). Modifikujeme tedy konstruktor třídy `Form1` takto (je uveden i s počátkem definice třídy):

```
public partial class Form1 : Form
    {
        String zn;
        public Form1()
```



Obrázek 1.7: Formulář *jednoduché kalkulačky*. Nahoře menu, pod ním *textBoxy, listBox, comboBox* a *radioButton*, ostatní jsou buďto *Buttony* nebo *Labely*.



Obrázek 1.8: Menu *Operace* a jeho submenu.

```
    {   // Pri inicializaci rovnou vybereme scitani
// aby nepadaly vyjimky z nevybrane operace.
        InitializeComponent();
        listBox1.SelectedIndex = 0;
        comboBox1.SelectedIndex = 0;
    }
```

Řetězec `zn` bude použit pro uložení výběru operace v menu, zatím jej ignorujme. Chceme-li se dozvědět, který prvek je vybraný, zeptáme se na hodnotu `listBox1.SelectedIndex`, resp. `comboBox1.SelectedIndex` a víme, která operace bude prováděna.

U *RadioButtonu* je potřeba zjistit, který z nich [`radioButton1.Checked` až `radioButton4.Checked`] je vybraný [ten bude `true` a ostatní `false`].

Nejvíce legrace si ovšem užijeme při ovládání skrze *menu*. Tam si je třeba volbu následujícího operatoru zapamatovat potenciálně libovolně dlouho před vyvoláním výpočtu. Opět pro všechny prvky v menu *Operace* vytvoříme jeden společný handler události `click` a pojmenujeme ho `znameni`:

```
private void znameni(object sender, EventArgs e)
{ //Vyber operace pres menu - operaci ulozime do promenne "zn":
    ToolStripMenuItem m = (ToolStripMenuItem)sender;
    zn = m.Text.ToString();
```

```
}
```

Tento handler pouze uloží do proměnné `zn` textový obsah vybrané položky.

Handler pro položku *Počítej!* v menu vytvoříme pak například takto:

```
private void poc_menu(object sender, EventArgs e)
{// Handler vypoctu pres menu.
    String vysledek;
    try
    {
        double a = Convert.ToDouble(textBox1.Text),
               b = Convert.ToDouble(textBox2.Text);
        switch (zn)
        {
            case "+":
                vysledek = Convert.ToString(a + b);
                break;
            case "- ": vysledek = Convert.ToString(a - b);
                break;
            case "*": vysledek = Convert.ToString(a * b);
                break;
            case "/": vysledek = Convert.ToString(a / b);
                break;
            default: //Neni vybrana operace:
                vysledek = "Chyba!!!";
                break;
        }
    }
    catch (Exception q)
    { vysledek = "Chyba!"; }
    label4.Text = vysledek;
}
```

Handler vypadá podivně, jelikož je v něm třeba ošetřit možnou výjimku (slovo `try` a blok `catch`). Tuto konstrukci bychom mohli vynechat, ale program by při nekorektním vstupu padal. Jinak děláme pouze to, že načteme obsahy *textBoxů*, zjistíme, jaký je obsah proměnné `zn` a podle toho provedeme operaci.

Tento handler použijeme i pro vybavení ostatních ovládacích prvků, tedy pro *listBox, comboBox* a *radioButton* (v tomto pořadí):

```
private void button1_Click(object sender, EventArgs e)
```

```
{ //Znamenko je vybrane v listBoxu:
    String pom = zn;//Ulozime jake znamenko je vybrano v menu
    zn = listBox1.SelectedItem.ToString();
    poc_menu(sender, e);//Zavolame vyhodnoceni pro "menu"
    zn = pom;//Obnovime znamenko z menu

}

private void button2_Click(object sender, EventArgs e)
{// Znamenko je vybrane v comboBoxu:
    String pom = zn;//Ulozime znamenko vybrane z menu
    zn = comboBox1.SelectedItem.ToString();
    poc_menu(sender, e);//Zavolame obsluhu pro variantu "menu"
    zn = pom;//Obnovime znamenko z menu
}

private void button3_Click(object sender, EventArgs e)
{
    String pom = zn;
    if(radioButton1.Checked)zn="+";
    if (radioButton2.Checked) zn = "- ";
    if (radioButton3.Checked) zn = "*";
    if (radioButton4.Checked) zn = "/";
    poc_menu(sender, e);
    zn = pom;
}
```

Když už jsme vyrobili v menu položku *Help*, uděláme handler i pro ni. V něm vytvoříme nové okno (pro jednoduchost prázdné, protože aplikace je snad natolik snadná, že ani nápovědu nepotřebuje. K tomu definujeme (na konci souboru `Form1.cs`) formulář `Form2` jako třídu, která je potomkem třídy `Form` a definujeme jí konstruktor:

```
public class Form2: Form // vyrobime nove okno, tedy tridu
                         // jsouci potomkem tridy Form
{
    public Form2() { Text = "Taky okno!";
    }
}
```

Handler `click` položce *Help* pak pojmenujeme třeba `help` a definujeme takto:

```
private void help(object sender, EventArgs e)
{   //Vyrobime nove okno (ktere se da zavrit) :-)
    Form2 formular = new Form2();
    formular.Show();
}
```

Cvičení

Nastudujte si ovládání prvku *checkBox*.

1.3 Dialogová okna

Velmi často se stává, že chceme uživatele na něco naléhavě upozornit, anebo se ujistit, že opravdu ví, co dělá. K tomu se hodí vytvořit nové okno. Pro jednodušší účely můžeme s výhodou využít třídy `MessageBox`, konkrétně její statické metody `Show`:
`MessageBox.Show("Pozor!");`
Toto volání vytvoří výstražné okénko s textem `,,Pozor!"`. Na toto okénko nečekáme žádnou odpověď. Pokud bychom chtěli od uživatele potvrdit, zda opravdu ví, co dělá, můžeme použít volání téže metody, resp. jinou její variantu (metoda `Show` je totiž velmi silně přetížena.[1]) Do argumentů této metody můžeme zapsat plno informací. Například můžeme popsat, jaká tlačítka se mají zobrazit. Dosáhneme toho tak, že mezi argumenty „zamícháme" konstantu kódující údaje o tlačítkách. Tyto konstanty lze získat jako hodnoty výčtového typu `MessageBoxButtons`. Pokud chceme dialog *OK/Cancel*, správná hodnota je:
`MessageBoxButtons.OKCancel`
a příslušné volání by vypadalo takto:

```
MessageBox.Show("Chcete zničit počítač?",MessageBoxButtons.OKCancel);
```

Nedosti na tom, že můžeme udat text dialogového okénka, my můžeme ještě specifikovat nadpis okénka a dokonce i ikonu, která se k dialogu má zobrazit:
`MessageBox.Show("Text","Nadpis",MessageBoxIcon.Hand);`
Jak vidíme v příkladu, pro zobrazení ikony lze využít jednu z předpřipravených ikon, které jsou k dispozici ve výčtovém typu `MessageBoxIcon`.

Jak jsme si již řekli, tímto způsobem můžeme zobrazovat i okna s více než jedním tlačítkem. V tom případě nás zřejmě zajímá výsledek uživatelova

[1] Připomeňme, že funkci nazveme přetíženou, pokud existuje více jejích definic lišících se parametry.

rozhodnutí (tedy které tlačítko stiskl). Abychom se dobrali tohoto výsledku, podotkněme, že výsledek volání `MessageBox.Show` je typu `DialogResult`. Etalony (tedy vzory) možných výsledků pak najdeme jako prvky tohoto výčtového typu. Takže pokud chceme znát výsledek rozhodování, můžeme vytvořit proměnnout typu `DialogResult` a porovnávat ji s vybranými hodnotami této třídy například takto:

```
DialogResult vys=MessageBox.Show("Opravdu ukončit program?",
                                        MessageBoxButtons.OKCancel);
if(vys==DialogResult.OK)
    Application.Exit();
```

Asi je to zřejmé, ale dávejte si pozor, že `DialogResult.OK` není totéž jako `DialogResult.Yes`.

Cvičení:

1. Probádejte hodnoty výčtových typů `DialogResults, MessageBoxButtons` a `MessageBoxIcon`.

2. Prozkoumejte všechny možné přetížené varianty metody `MessageBox.Show` a naučte se je používat.

1.4 Kreslení

Divácky velmi vděčnou částí jakéhokoliv studijního materiálu je kreslení. Jelikož jazyk C# umožňuje kreslit prakticky po všem, na co si ukážete, nemůžeme své čtenáře o tuto pasáž ochudit. Na druhé straně předpokládáme, že jednotlivé metody kreslící různé útvary si průměrný čtenář dovede nastudovat, proto se omezíme pouze na popis metodický, tedy jakým způsobem vůbec jazyk C# ke kreslení donutit.

Kreslení probíhá pomocí instance třídy `Graphics`, která implementuje jednotlivé grafické prvky (kreslení čáry, kreslení křivky...). Abychom mohli kreslit, musíme si vytvořit „pero", tedy instanci třídy `Pen`, která bude určovat, jak chceme kreslit (tedy barvu a tloušťku čáry):

```
Pen pero=new Pen(Color.Blue,5);
```

Jak na příkladu vidíme, barvu můžeme popsat pomocí statických atributů třídy `Color`, tloušťku čáry popíšeme číslem (typu `double`).

Instanci třídy `Graphics` nezískáme tak, jak bychom očekávali, ale necháme si ji vytvořit od konkrétního objektu, po kterém chceme malovat. Jak jsme uvedli hned zkraje, malovat lze v C# téměř po čemkoliv (s výjimkou formuláře, tedy kupř. `Form1`). Prvek po kterém chceme kreslit, musí být osazen metodou `CreateGraphics`. Tedy chceme-li kreslit například po tlačítku

s číslem 1, vytvoříme si příslušnou „grafiku" takto:
`Graphics grafika=button1.CreateGraphics();`

Nyní jsme se ovšem dostali do ne příliš záviděníhodné situace. To, po čem bychom nejspíše chtěli kreslit (tedy samotná plocha formuláře) nepřipadá v úvahu a naopak kreslení po tlačítkách asi málokoho nadchne. Aby nebylo problémů málo, občas bývá problém najít mezi ovládacími prvky v *Toolbox*u prvek *Canvas*. Pokud je to náš případ, nemusíme hledat prvek *Canvas* a pomůžeme si například prvkem *PictureBox* (teoreticky by mohl být vhodný i prvek `Label`, ale tomu není tak snadné nastavit velikost, jako *PictureBox*u.

Kreslení si demonstrujeme na tomto jednoduchém příkladu: Vytvoříme *PictureBox* a nastavíme mu obsluhu události *Click* tak, aby nakreslil modrou čáru z bodu $(0, 0)$ do bodu $(500, 500)$. Začneme naklikáním formuláře, které necháme jako samostatnou práci. Dvojclicknutím na *PictureBox* se před námi objeví předepsaný ovladač události. Modifikujeme jej takto:

```
private void pictureBox1_Click(object sender, EventArgs e)
{   Graphics x = pictureBox1.CreateGraphics();//Nova grafika
    Pen pero = new Pen(Color.Blue, 5);//Barva, sirka stetce
    x.DrawLine(pero, 0, 0, 500, 500);//cim, odkud a kam
}
```

1.4.1 Pozice myši

S kreslením nepřímo, leč velmi těsně souvisí určování pozice myši. Uveďme si toto na příkladu. Chceme udělat velmi snadný nástroj s plochou na kreslení, po níž budeme kreslit čáru clickáním myší. Prvním clicknutím zahájíme čáru řekněme z pozice (0,0). Dále budeme vždy kreslit na sebe navazující úseky čáry. K tomu potřebujeme zjistit polohu myši.

Polohu myši můžeme zjistit ve vztahu k určité události (např. `MouseDown`, `MouseUp...`), tedy handler se dozví, kde byla myš v okamžiku příslušné události. Polohu myši zjistíme velice jednoduše. Jako druhý argument handleru příslušné události (související s myší) najdeme `MouseEventArgs e` (povšimněte si, že u událostí s myší alespoň názvem nesouvisejících, kupř. *Click* je druhý argument obecnějšího typu, a to `EventArgs`). Chceme-li znát polohu myši při dotyčné události, prostě odkážeme objektu `e` k atributům `X` a `Y`.

Tedy chceme-li implementovat náš příklad (jednoduché malovátko, které bude spojovat lomenou čárou body, které označíme clicknutím myši), vyrobíme formulář, na který umístíme prvek *PictureBox*. Abychom věděli, kam jsme clickli minule, vyrobíme „něco jako globální proměnné", i když při skutečném programování bychom se pokusili situaci vyřešit systematičtěji.

Naší náhražkou globálních proměnných může být například statická třída se dvěma veřejnými integerovými statickými atributy:
`public static class minula_poloha{public static int x=0,y=0;}`

Nyní se vrátíme zpět k našemu *PictureBoxu* a definujeme mu ovladač události `MouseClick` takto:

```
private void pictureBox1_MouseClick(object sender, MouseEventArgs e)
{   Graphics x = pictureBox1.CreateGraphics();
    Pen p = new Pen(Color.Blue, 1);
    x.DrawLine(p,minula_poloha.x, minula_poloha.y, e.X, e.Y);
    minula_poloha.x = e.X;
    minula_poloha.y = e.Y;
}
```

1.4.2 Kreslení dalších objektů

Podobným způsobem jako čáru lze nakreslit i plno dalších obrázků, například kružnici. Je třeba nenechat se zmást tím, že neexistuje `g.DrawCircle`, jelikož existuje obecnější metoda `g.DrawEllipse`. Tato metoda bere pět argumentů. Prvním je opět pero (tedy barva a šířka čáry), další dva určují souřadnici středu elipsy a další dva určují délky os. Pokud nastavíme tyto délky shodné, vznikne kružnice.

Samozřejmě můžeme objekty též vyplňovat, nebo dokonce již kreslit vyplněné. V tom případě jsou nám k dispozici podobné metody, jako pro kreslení (kupříkladu `FillEllipse`) mající obdobnou strukturu argumentů. Jen je třeba dát pozor na to, že jako první argument tentokrát nezadáváme pero (`Pen`), ale štětec (`Brush`). Štětce můžeme získat jako statické atributy třídy `Brushes`. Tedy správné použití metody `Fill Ellipse` vypadá asi takto:
`x.FillEllipse(Brushes.Black,20,20,10,10);`

Kromě statických atributů třídy Brushes se ke štětcům můžeme dostat tak, že si je vytvoříme. A máme na výběr ze dvou možností, jaké štětce si vybrat. Buďto můžeme použít štětec dané barvy (`SolidBrush`) anebo štětec kreslící barevným vzorem (`TextureBrush`). V tomto případě se štětce vytvářejí tak, jak by člověk očekával, tedy například:
`Brush b=new SolidBrush(Color.Black);`
resp. `Brush c=new TextureBrush(bitmapa);`

Příklad s texturovou bitmapou by možná snesl delší povídání, jednalo by se však o poměrně technické povídání, tak jen odkažme, že proměnná `bitmapa` je objekt typu `Bitmap` a nastudování vlastní práce s bitmapou necháme individuálním zájemcům jako samostatnou práci.

Cvičení:

1. Prostudujte jednotlivé metody třídy `Graphics`.
2. Modifikujte program tak, aby kolem kreslicí plochy byla tlačítka nastavující barvu (červenou, zelenou a modrou) a umožňující kreslit lomenou čáru či Beziérovy křivky nebo kružnice. Vhodnou parametrizaci kružnic (pomocí myši) sami navrhněte.

Závěr

Zatím jsme si ukázali, jak obecně pracovat s formulářem při tvorbě *Windows Application*, předvedli jsme si obecnou tvorbu handlerů jednotlivých událostí a speciálně jsme se věnovali prvkům *Button, Combo Box, Label, List Box, Radio Button, Text Box* a *menu*. Ukázali jsme si, jak zjistit polohu myši při konkrétní události a jakým způsobem donutit C# kreslit. Tyto prvky tvoří již poměrně dobrý repertoir v mnoha případech postačující. Předpokládám, že v této chvíli jste schopni samostatně tvořit netriviální aplikace v prostředí Visual Studia v jazyku C#. Pokud je tomu tak, materiál splnil svůj cíl. Pokud máte další požadavky, či nápady, jak materiál vylepšit, napište autorovi na adresu `mper7437@artax.karlin.mff.cuni.cz`. Pokud vás napadají impertinence na adresu autora, ty můžete s úspěchem napsat do jakéhokoliv *textBoxu*, který teď už jistě umíte nejen vytvořit, ale i pomalovat. :-)

Rejstřík

Argument *sender*, 8
Atribut *Checked*, 18
Atribut *Items*, 16
Atribut *Maximum*, 5
Atribut *ReadOnly*, 2
Atribut *SelectedIndex*, 18
Atribut *SelectedItem*, 16
Atribut *Tag*, 8
Atribut *Text*, 2
Atribut *Visible*, 13

Dialogová okna, 21

Enum *DialogResult*, 22
Enum *MessageBoxButtons*, 21

Funkce *Application.Exit()*, 5, 16

Ikona *Events*, 4

Kompilace a spuštění, 5

Metoda *CreateGraphics*, 23
Metoda *DrawEllipse*, 24
Metoda *DrawLine*, 23
Metoda *FillEllipse*, 24
Metoda *MessageBox*, 21
Metoda `Show` (třídy `MessageBox`), 21

Prvek *button*, 1, 4, 7
Prvek *checkbox*, 1
Prvek *combo box*, 15–17, 19
Prvek *label*, 1, 15, 17
Prvek *list box*, 15–17, 19
Prvek *menu strip*, 16, 17
Prvek *PictureBox*, 23
Prvek *progress bar*, 1, 2, 4
Prvek *radio button*, 15, 17–19
Prvek *text box*, 1, 17

Třída *Brushes*, 24
Třída *Color*, 22
Třída *Graphics*, 22
Třída *Pen*, 22
Třída *SolidBrush*, 24
Třída *TextureBrush*, 24

Událost *click*, 2, 4, 8, 10
Událost *MouseClick*, 24

Vytvoření nového okna, 20

Zjištění pozice myši (při události), 23