

Cvičení č. 2 – Struktura aplikace

Cíl

Ukázat si, jak snadno lze ve Visual Studiu vytvořit WinForms aplikaci, která je ale špatně navržena a porušuje pravidla objektového přístupu a jak postupovat lépe.

Úvod

Visual Studio vysloveně svádí k tomu, že se WinForms aplikace dají vytvořit čistě pomocí definice kódu v metodách obsluhy událostí (event handlers) základního formuláře aplikace. Tento přístup, byť funkční (dokud neporušuje syntaktická pravidla a jde přeložit, je kompilátoru struktura kódu jedno), však není vhodný, protože porušuje prakticky všechna pravidla a doporučení, která se pro softwarový design běžně uvádí, mj.:

KISS – Keep It Simple Stupid – Kód by měl být co nejjednodušší. Čím jednodušší kód, tím lépe se v něm dá orientovat, porozumět mu a tím pádem je i snadněji udržitelný. Složitý problém by měl být ideálně rozdělen na co nejmenší a nejjednodušší kroky, které jsou implementovány co nejmenším množstvím zdrojového kódu.

DRY – Don't Repeat Yourself – Neopakovat se! Je velmi jednoduché nakopírovat kus funkčního kódu na jiné místo, kde je potřebná stejná funkcionality. Kód, který vzniká tímto způsobem je ale velmi těžko udržitelný a rozšiřovatelný.

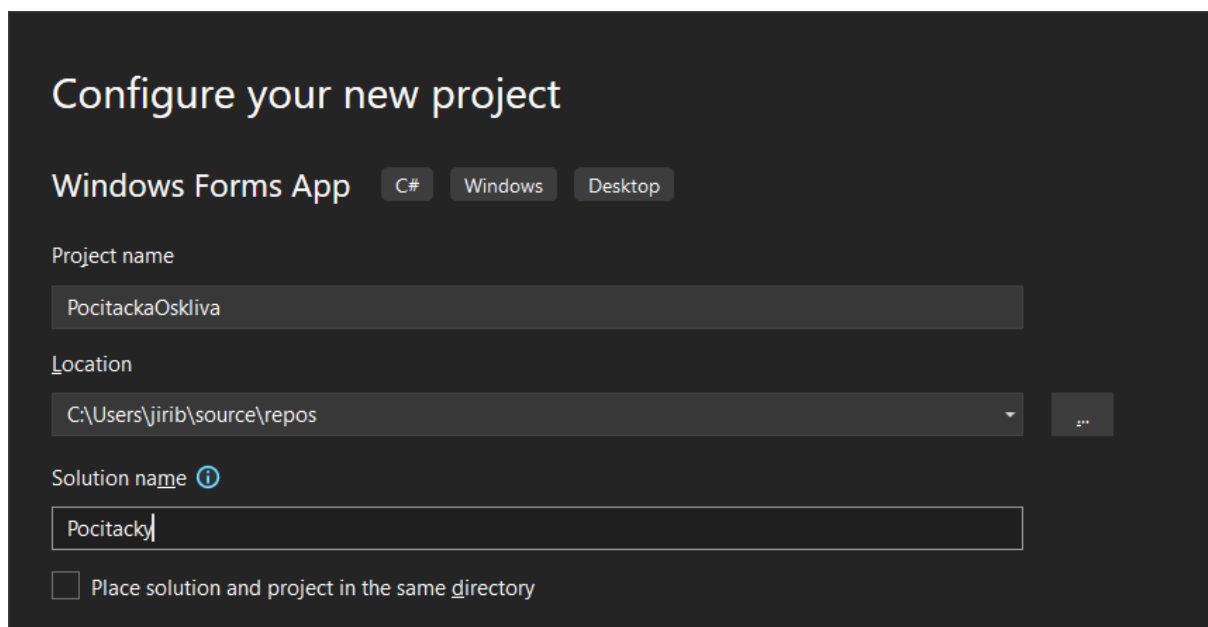
SRP – Single Responsibility Principle – Princip jediné odpovědnosti říká, že každá třída by měla mít na starosti jen určitou část funkcionality aplikace a tato funkcionality by v ní měla být zapouzdřena. Zkrátka a dobře, každá třída či metoda by měla mít jen jeden účel a ne víc.

Praktické řešení

V rámci tohoto cvičení vytvoříme tři podoby téže aplikace. První dvě budou klíčové. První verze bude vytvořena právě oním výše zmíněným způsobem, který porušuje pravidla. Na druhé verzi si pak ukážeme, jak zásadní nedostatky odstranit. Třetí verze pak bude vycházet z druhé a bude jí rozšiřovat o použití pokročilejších technik. Pro váš semestrální projekt ale bude stačit, když bude v duchu druhé verze.

Naše cvičná aplikace bude představovat jednoduchý výpočetní nástroj, který bude umět sčítat, odečítat, násobit a dělit čísla zadaná do dvou vstupních polí a vypisovat výsledek operace do třetího vstupního pole. Abychom se mohli přepínat mezi jednotlivými verzemi, umístíme všechny tři projekty do jednoho řešení (solution).

Spustíme Visual Studio a vytvoříme nový projekt „Aplikace Windows Forms“/“Windows Forms App“. Tento projekt nazveme *PocitackaOskliva* a umístíme ho do řešení *Pocitacky*.



Configure your new project

Windows Forms App C# Windows Desktop

Project name

PocitackaOskliva

Location

C:\Users\jirib\source\repos

Solution name ⓘ

Pocitacky

☐ Place solution and project in the same directory

Jako platformu můžete vybrat jak .NET 6.0, tak .NET 7.0, protože nebudeme používat žádné funkce, které by dostupné jen v C# verze 11.

Vytvoření GUI

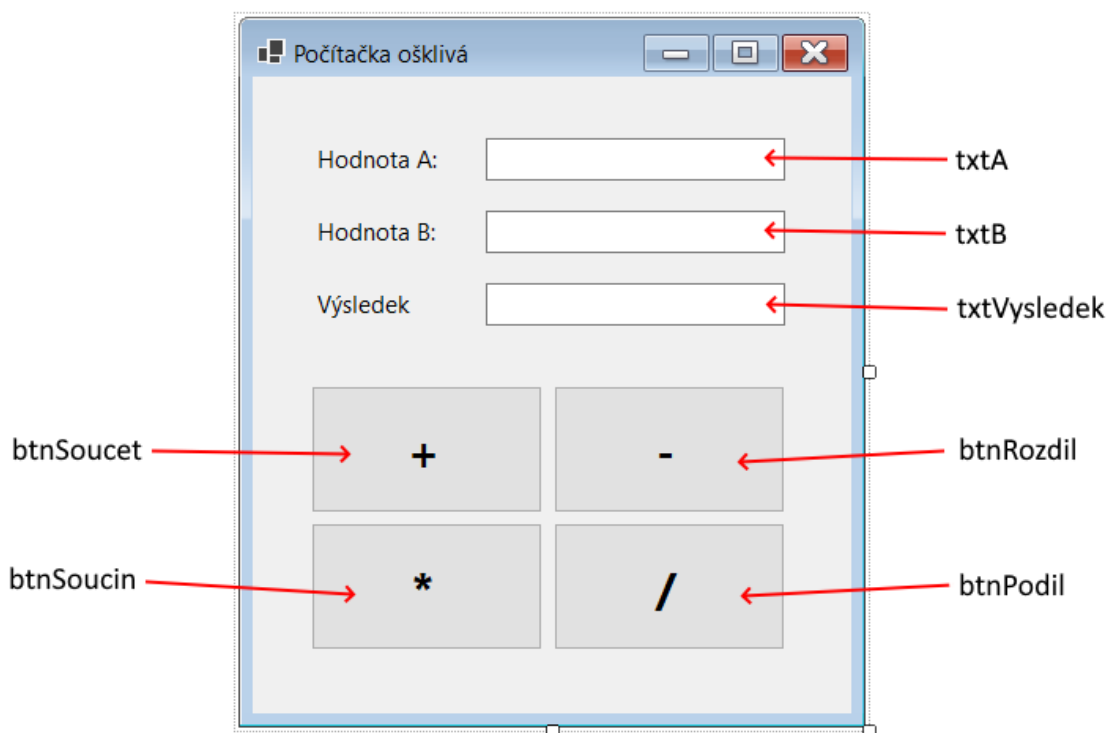
Všechny tři verze aplikace budou mít stejný vzhled. Na prázdný formulář postupně následující komponenty:

- 3x Label
- 3x TextBox
- 4x Button

Komponenty Label budou sloužit čistě jako popisky ke vstupním polím TextBox a tlačítka Button se budou spouštět jednotlivé početní operace.

Abychom si usnadnili práci, všechna vstupní pole a tlačítka si přejmenujeme z výchozích hodnot (textBox1, button1 atd.) na smysluplnější názvy změnou jejich vlastnosti (Name) a změníme i jejich vlastnost Text. U popisků Label není nutné měnit jméno objektu a změníme jen jeho Text. Proč? Protože popisky nebudeme vůbec v kódu volat, takže není nutné, aby měly srozumitelnější jména. Pokud bychom však chtěli obsah popisků dynamicky měnit za běhu aplikace, pak by samozřejmě jejich přejmenování bylo vhodné. Obrázek níže ukazuje výsledné GUI. Jednotlivé prvky už na obrázku mají nastavenou patřičnou hodnotu Text a u všech vstupních polí a tlačítek jsou připojeny i jejich nové názvy. U tlačítek je navíc zvětšena velikost písma pro snazší čtení. Jména objektů samozřejmě můžete zvolit i po svém, jen je pak nutné patřičně upravit zdrojové kódy, které budeme vytvářet dále.

U formuláře samotného změníme jeho vlastnost Text na „Počítačka ošklivá“, abychom později na první pohled poznali, kterou verzi aplikace jsme spustili.



Když je GUI hotové, můžeme začít do aplikace přidávat samotnou funkcionalitu. Všechny 4 výpočetní operace budou fungovat na stejném principu – vezmeme číslo zadané do pole **txtA** a číslo zadané do pole **txtB**, provedeme s nimi danou operaci a výsledek zapíšeme do pole **txtVysledek**. Pro potřeby tohoto příkladu nebudeme řešit ošetření chybných vstupů (zadání písmene místo čísla atp.) a budeme očekávat, že hodnoty A a B budou vždy celá čísla.

Nesmíme zapomenout, že vlastnost Text obsahuje hodnotu typu string. Pokud bychom tedy vzali čistě hodnoty vlastností Text u prvků **txtA** a **txtB**, operace součtu (operátor +) by sice fungovala, ale vracela by delší textový řetězec vzniklý sloučením obou dohromady. Operace rozdílu, součinu a podílu by ale nefungovaly vůbec, protože nejsou pro string definovány. Takový kód by nešel ani přeložit. Musíme tedy provést konverzi textové hodnoty na číselnou. K tomu můžeme použít buď metodu **int.Parse(string)** nebo třídu Convert a její metodu pro převod na Int32 – **Convert.ToInt32(string)**. Po výpočtu nesmíme ovšem zapomenout výsledek převést zpět na string, protože typ int/Int32 se nedokáže implicitně (ani explicitně) převést na textový řetězec. K tomu musíme využít buď opět třídu Convert a metodu **ToString(int)** nebo zavolat metodu **ToString()**, kterou každý typ dědí z výchozího typu Object. V našem příkladu použijeme právě tuto metodu.

Kód vykonávající jednotlivé operace zadáme do obsluh událostí Click jednotlivých tlačítek. Stačí tedy postupně vždy dvojkliknout na tlačítko na formuláři a počkat, až se vygeneruje patřičná metoda a přepne se do ní editor. Po dvojkliku na tlačítko btnSoucet se otevře editor s následující prázdnou metodou sloužící k obsluze události Click tohoto tlačítka:

```
private void btnSoucet_Click(object sender, EventArgs e)
{
}

```

Doplníme do ní následující řádek, který vykonává samotný součet.

```
txtVysledek.Text = (Convert.ToInt32(txtA.Text) + Convert.ToInt32(txtB.Text)).ToString();
```

Všimněte si, že aby mohl být výsledek převeden na string, musíme součet obalit závorkou. Závorka pak představuje výslednou hodnotu typu int, na kterou už můžeme aplikovat metodu **ToString()**. Obdobně bychom museli postupovat i se třídou Convert, kde by obsah závorky byl předáván jako parametr při volání metody **Convert.ToString(...)**.

Obdobně, jako je zapsán kód pro součet, bude vypadat i kód pro zbylé operace. Jediným rozdílem bude použití jiného operátoru při výpočtu – -, *, /. Obsluhy události Click pro všechna čtyři tlačítka tak budou vypadat takto:

```

private void btnSoucet_Click(object sender, EventArgs e)
{
    txtVysledek.Text = (Convert.ToInt32(txtA.Text) + Convert.ToInt32(txtB.Text)).ToString();
}

private void btnRozdil_Click(object sender, EventArgs e)
{
    txtVysledek.Text = (Convert.ToInt32(txtA.Text) - Convert.ToInt32(txtB.Text)).ToString();
}

private void btnSoucin_Click(object sender, EventArgs e)
{
    txtVysledek.Text = (Convert.ToInt32(txtA.Text) * Convert.ToInt32(txtB.Text)).ToString();
}

private void btnPodil_Click(object sender, EventArgs e)
{
    txtVysledek.Text = (Convert.ToInt32(txtA.Text) / Convert.ToInt32(txtB.Text)).ToString();
}

```

Aplikaci můžeme spustit a vyzkoušet. Pokud budeme zadávat čísla správně (tj. celá), bude aplikace ve většině případů fungovat bez problémů. U dělení si však můžeme všimnout dvou věcí – při dělení nulou aplikace očekávaně spadne s výjimkou `DivideByZeroException` a pokud by výsledek dělení měl být desetinné číslo, zobrazí se jen jeho celá část. První problém je způsoben tím, že pro celá čísla není operace dělení nulou definována. Druhý problém pak tím, že operátor `/` se při dělení dvou celých čísel chová jako operátor celočíselného dělení.

Dělení nulou můžeme ošetřit buď kontrolou dělitele před samotným dělením, tj. nedopustit, aby k dělení vůbec došlo, nebo naopak řešit jeho následek pomocí ošetření výjimky (k tomu jsme se na přednášce zatím nedostali).

Aby se správně zobrazil desetinný výsledek, je nutné, aby alespoň jedna z hodnot při dělení nebyla celé číslo, tj. musí být typu `float/Single`, `double/Double` nebo `decimal/Decimal`. V našem případě tedy bude stačit, aby se v metodě `btnPodil_Click` jedno z volání `Convert.ToInt32()` změnilo na `Convert.ToSingle()` nebo `Convert.ToDouble()`. Převod na jeden z těchto formátů nám pak vyřeší i problém s dělením nulou. Typy `Single` i `Double` provádí kódování hodnoty podle standardu IEEE 754, který jsme si zmiňovali v předmětu Úvod do principu počítačů, a pro který mimo jiné platí to, že má vyhrazené hodnoty pro hodnotu nekonečno a mínus nekonečno. Při dělení nulou tak v tomto případě dojde právě k vrácení jedné z těchto hodnot.

Zkuste provést tyto změny a zkontrolujte si, že vše pracuje tak, jak má.

Pokud ano, tak gratuluji, máte první aplikaci. Bohužel není ale strukturovaná tak, jak by správně měla být, a porušuje princip jediné odpovědnosti.

S tímto jednoduchým příkladem to není tak zjevné, ale představte si, že jednotlivé metody pro obsluhu události (event handlers) neobsahují jen jeden řádek kódu, ale nějaký algoritmus, který provádí složitější činnost. Veškerý kód je obsažen v metodách, které jsou součástí třídy `Form1`, tedy třídy představující samotné okno aplikace. Odpovědností takové třídy by tedy mělo být všechno související se zobrazením aplikace uživateli a vstupem a výstupem prostřednictvím uživatelského rozhraní. V našem případě ale v metodách provádíme výpočty, což je operace nesouvisející s účelem této třídy. Pokud bychom si místo výpočtů představili něco složitějšího, pak je tento problém ještě znatelnější.

Jak z toho ven? Musíme kód rozdělit do tříd, které budou odpovědné za určitou část funkcionality. V našem příkladu bude stačit třída jedna, ve složitějších aplikacích by jich samozřejmě bylo třeba více. Až úpravy dokončíme, bude se na první pohled zdát, že jsme si naopak přidali práci, protože kód třídy `Form1` se až tolik lišit nebude. To je ale dáno tím, že je náš příklad velmi jednoduchý.

Pojďme tedy vytvořit nový projekt, který bude splňovat naše požadavky. Nezavírejte stávající projekt a z menu `Visual Studio` vyberte `Soubor > Nový > Projekt...`, vyberte stejný typ projektu, pojmenujte ho třeba `PocitackaHezci` a co je nejdůležitější, u položky `Řešení (Solution)`, vyberte možnost `Přidat k řešení (Add to solution)`.

Configure your new project

Windows Forms App

C#

Windows

Desktop

Project name

PocitackaHezci

Location

C:\Users\jirib\source\repos\Pocitacky

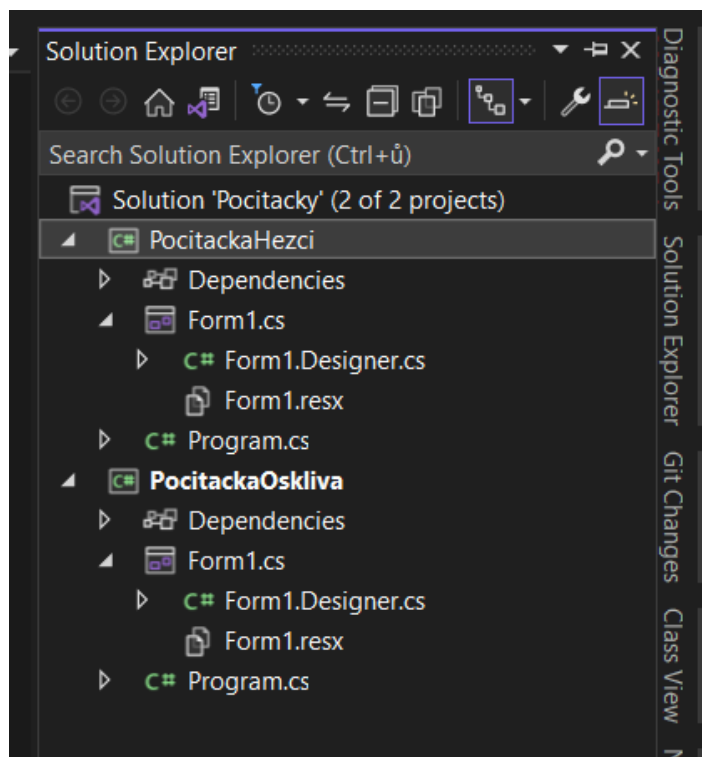
Solution

Add to solution

Solution name ⓘ

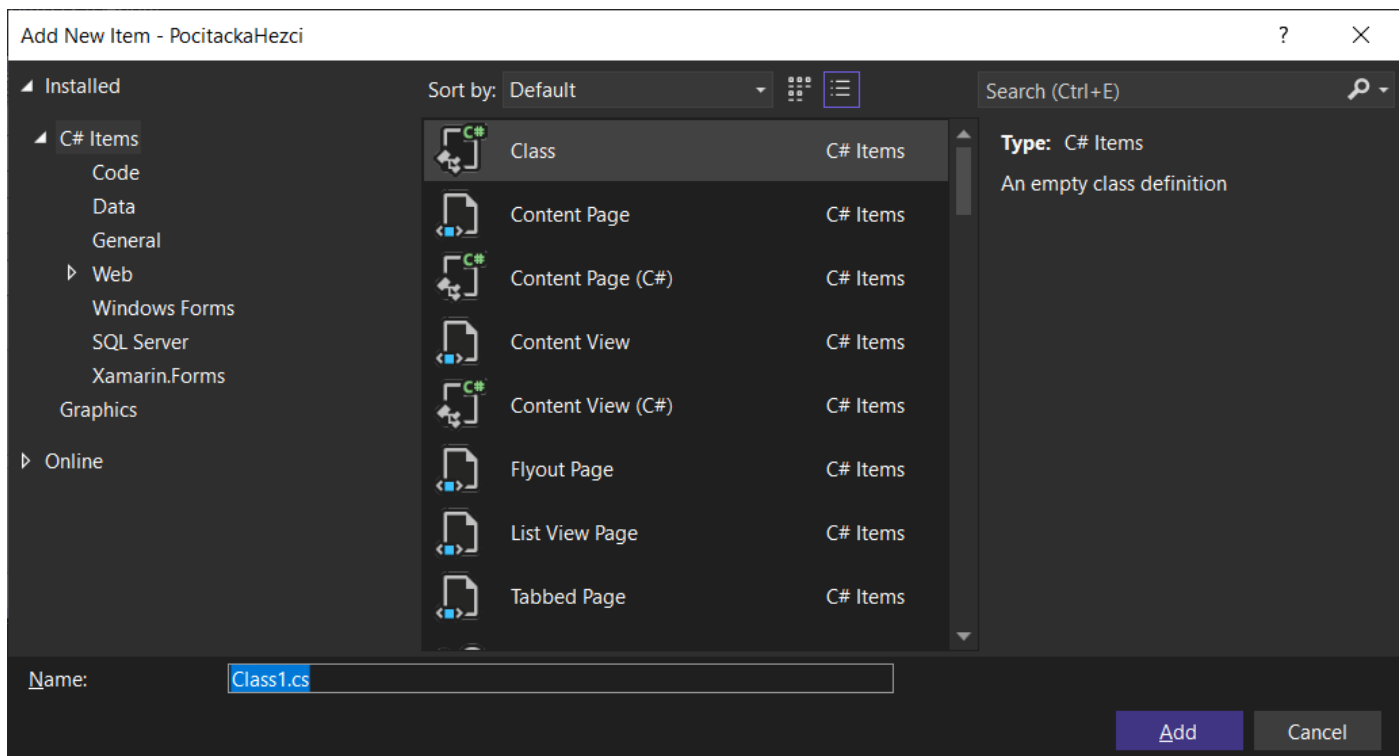
Pocitacky

Tím zajistíme, že tento projekt bude uložen ve stejném řešení jako projekt předchozí a budeme tak mít dostupné oba. Ušetříme si díky tomu práci, protože mimo jiné budeme moci velmi jednoduše zkopírovat všechny prvky z GUI původního projektu do nového. Takto by nějak měl poté vypadat Průzkumník řešení.



Nyní můžeme překopírovat všechny prvky z formuláře naší první aplikace do aplikace, kterou jsme právě vytvořili. Označte všechny prvky pomocí myši a klávesovou zkratkou Ctrl+C je zkopírujte do schránky. Poté se přepněte na formulář nové aplikace a zkratkou Ctrl+V je vložte. Výhodou je, že všechno nastavení, které jsme provedli v první aplikaci, se překopíruje a nemusíme nic měnit. Jen u samotného formuláře změníme výchozí hodnotu vlastnosti Text na *Počítačka hezčí*.

Nyní musíme do našeho projektu přidat novou třídu, do které bude vyčleněna odpovědnost za provádění početních operací. V menu proto vybereme položku Projekt > Přidat třídu... (Project > Add class...).



Zkontrolujeme si, že v dialogovém okně, které se otevře, je jako typ přidávané položky vybrána Třída (Class) a do spodního vstupního pole, kde je prozatím uveden název Class1.cs, vyplníme název třídy, kterou chceme vytvořit. V našem případě to bude **Vypocty**. Není nutné doplňovat příponu .cs, Visual Studio si jí doplní automaticky.

Visual Studio otevře nový soubor, který bude obsahovat výchozí definici naší nové třídy. Tu budeme muset pozměnit. Místo modifikátoru **internal** použijeme **public** a přidáme ještě klíčové slovo **static**, protože naše nová třída bude statická.

```
public static class Vypocty
{
}
```

Co to pro nás znamená? Statická třída je specifickou podobou třídy, která se používá jako „kontejner na funkcionalitu“. Protože výsledky metod, které budou ve třídě **Vypocty** definovány, nebudou závislé na vnitřním stavu objektu, ale pouze hodnotách parametrů, které budou metodám předány, není potřebné od takové třídy vytvářet instance a třída tak může být definována jako statická. V praxi se takto řeší omezení jazyka C#, který neumožňuje definovat funkce mimo třídy (na rozdíl např. od C++). Volání metod třídy pak probíhají přímo přes název třídy, stejně jako u tříd Console nebo Convert, které jsou obě taktéž statické. Nesmíme jen zapomenout na to, že vše, co je ve statické třídě definováno, musí být taktéž statické (statická třída nesmí obsahovat instanční prvky).

Nyní musíme tedy ve třídě definovat potřebné metody, které budou sloužit k provedení výpočtu. Ty budou fungovat podobně jako metody, které jsme definovali v první verzi aplikace.

```
public static class Vypocty
{
    public static int Soucet(int a, int b) { return a + b; }
    public static int Rozdil (int a, int b) { return a - b; }
    public static int Soucin (int a, int b) { return a * b; }
    public static double Podil (double a, double b) { return a / b; }
}
```

Protože je tělo metody jen jeden výraz, můžeme použít i definici metody přímo pomocí výrazu:

```
public static class Vypocty
{
    public static int Soucet(int a, int b) => a + b;
    public static int Rozdil(int a, int b) => a - b;
    public static int Soucin(int a, int b) => a * b;
    public static double Podil(double a, double b) => a / b;
}
```

Všimněte si, že u metody pro výpočet podílu je jako datový typ obou parametrů i výstupní datový typ použit **double**. Protože převod hodnoty typu **int** na hodnotu **double** probíhá implicitně, není nutné se o něj starat a metoda přijme jako parametry celočíselné hodnoty bez jakékoliv další práce. Zároveň tím ale zajistíme, že pokud bude třeba, výsledek bude zobrazen správně jako desetinné číslo.

Stejně jako jsme vytvářeli metody pro obsluhu událostí v prvním programu, vytvoříme je nyní i zde. Opět stačí dvojklikem vygenerovat event handler pro každé z tlačítek. Ve vygenerovaných metodách tentokrát ale budeme pro výpočet volat patřičnou metodu ze třídy **Vypocty**. Nesmíme samozřejmě zapomenout provést potřebné změny typu.

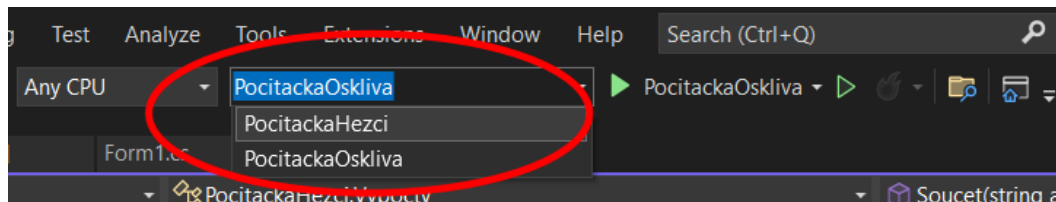
```
private void btnSoucet_Click(object sender, EventArgs e)
{
    txtVysledek.Text = Vypocty.Soucet(Convert.ToInt32(txtA.Text),
                                      Convert.ToInt32(txtB.Text)).ToString();
}

private void btnRozdil_Click(object sender, EventArgs e)
{
    txtVysledek.Text = Vypocty.Rozdil(Convert.ToInt32(txtA.Text),
                                      Convert.ToInt32(txtB.Text)).ToString();
}

private void btnSoucin_Click(object sender, EventArgs e)
{
    txtVysledek.Text = Vypocty.Soucin(Convert.ToInt32(txtA.Text),
                                      Convert.ToInt32(txtB.Text)).ToString();
}

private void btnPodil_Click(object sender, EventArgs e)
{
    txtVysledek.Text = Vypocty.Podil(Convert.ToInt32(txtA.Text),
                                      Convert.ToInt32(txtB.Text)).ToString();
}
```

Program můžeme spustit a vyzkoušet. Protože ale máme v rámci řešení už dva projekty, musíme před spuštěním určit, který z nich se má spustit/ladit. To nejjednodušeji uděláme přepnutím na nový projekt ve vyklepávacím seznamu v liště nástrojů.



Protože hodnoty přicházejí ze vstupních polí na formuláři v podobě textových řetězců, dala by se uvažovat i taková úprava třídy **Vypocty**, která by umožnila výpočetním metodám předat jako parametry hodnoty typu **string** a také by **string** vracela. Ke konverzi hodnot by tedy nemuselo docházet přímo v event handlerech, ale teprve v rámci výpočtu. Tato úprava je vcelku snadná, spočívá v přetížení každé z výpočetních metod. Tj. v definici stejných metod, ale s jinými parametry. Uvnitř jednotlivých metod by pak de facto probíhal podobný proces, jaký je v současné chvíli definovaný v event handlerech, tj. vstupní hodnoty se převedou na čísla, provede se výpočet a nakonec se výsledek převede na **string**, aby mohl být zobrazen.

Takto by mohla vypadat přetížení všech výpočetních metod ve třídě **Vypocty**:

```
public static string Soucet(string a, string b)
{
    return Soucet(Convert.ToInt32(a), Convert.ToInt32(b)).ToString();
}

public static string Rozdil(string a, string b)
{
    return Rozdil(Convert.ToInt32(a), Convert.ToInt32(b)).ToString();
}
```

```

public static string Soucin(string a, string b)
{
    return Soucin(Convert.ToInt32(a), Convert.ToInt32(b)).ToString();
}

public static string Podil(string a, string b)
{
    return Podil(Convert.ToInt32(a), Convert.ToInt32(b)).ToString();
}

```

Poté bychom mohli aktualizovat event handlers jednotlivých tlačítek a upravit volání výpočetních metod tak, aby se jako parametry předávaly přímo textové řetězce zadané do vstupních polí:

```

public static string Soucet(string a, string b)
{
    return Soucet(Convert.ToInt32(a), Convert.ToInt32(b)).ToString();
}

public static string Rozdil(string a, string b)
{
    return Rozdil(Convert.ToInt32(a), Convert.ToInt32(b)).ToString();
}

public static string Soucin(string a, string b)
{
    return Soucin(Convert.ToInt32(a), Convert.ToInt32(b)).ToString();
}

public static string Podil(string a, string b)
{
    return Podil(Convert.ToInt32(a), Convert.ToInt32(b)).ToString();
}

```

Jak bylo zmíněno na začátku, tato verze aplikace bude působit komplikovaněji než verze první, ale oproti ní odstraňuje problém s porušením principu jediné odpovědnosti. Což mimochodem přináší i výhodu oproti první verzi, kde veškerá logika byla natvrdo zakódována ve třídě formuláře a celá aplikace tak byla v podstatě anti-vzorem „božský objekt“. Třidu **Vypocty** totiž můžeme dle libosti přenést do jiného projektu a využít ji tam. To by s první verzí aplikace udělat nešlo.

Pokročilé úpravy

S rozdělením logiky a uživatelského rozhraní můžeme zajít i ještě o krok dále. Budeme k tomu ale potřebovat využít typ **delegate**, který je vlastně takový ukazatel na metodu, případně jeho generickou podobu **Func<>**. Ve třetí podobě aplikace si ukážeme využití typu **delegate**, úpravu na využití generického delegáta **Func<>** si ukážeme, až ho proberem na přednášce.

Založte si v našem řešení další nový projekt, pojmenujte si ho třeba *PocitackaDelegat* a vytvořte jeho GUI stejným způsobem (copy-paste) jako jsme vytvořili GUI pro druhý projekt. I v tomto projektu budeme mít statickou třídu **Vypocty**, ale ta se bude od podoby ve druhém projektu lišit, takže si jí postupně vytvoříme znovu.

Idea třetí podoby aplikace je taková, že při stisku tlačítka, které iniciuje početní operaci, se už nebude muset čekat na výstup výpočetní metody, ale spuštěním této metody činnost tlačítka skončí. Výpočetní metoda převezme vstupy, zpracuje je a vypočítá výsledek. Zobrazení výsledku pak spustí sama výpočetní metoda právě pomocí využití delegáta.

Aby program fungoval, budeme muset rozšířit třídu Form1 o metodu, která bude sloužit k zobrazení textu v textovém poli **txtVysledek**. Protože bude obsahovat jen jedno přiřazení, může mít třeba tuto podobu:

```
private void ShowResult(string s) => txtVysledek.Text = s;
```

Je definována jako privátní, protože náleží jen formuláři a ten jí nebude vystavovat okolnímu světu. Nicméně i svou privátní metodu může objekt svázat s delegátem, takže nám nebude nic bránit v tom, abychom jí využili.

Dalším krokem bude implementace třídy pro výpočty. Pokud jste ji ještě nezaložili, udělejte to teď. Postup je úplně stejný, jako v případě druhého projektu, včetně změny **internal** na **public** a upřesnění, že třída bude statická.

Abychom mohli provést propojení, musíme nadefinovat strukturu delegátského typu. Ve stejném souboru jako máme třídu **Vypocty**, ale mimo ni, tedy nadefinujeme delegátský typ tak, aby se jeho definice shodovala se signaturou metody, kterou chceme volat. Nazveme ho třeba **VystupVysledku**.

```
public delegate void VystupVysledku(string s);
```

Nyní musíme ve třídě **Vypocty** přidat vlastnost typu **VystupVysledku**, která nám umožní připojit výstupní metodu z formuláře:

```
public static VystupVysledku Vystup { get; set; } = null;
```

Touto definicí říkáme, že vlastnost je statická (musí být, protože celá třída je statická), má automaticky vytvořený getter a setter a její výchozí hodnota je **null**. Pomocí této vlastnosti pak budeme iniciovat výpis výsledku ve formuláři. Můžeme tak naimplementovat chybějící metody, které budou počítat, ale místo toho, aby přímo vracely výsledek, budou ho vypisovat právě pomocí tohoto delegátu.

Metoda tedy bude brát jako parametry dvě hodnoty typu **int** (respektive **double** pro dělení), provede patřičnou operaci a zavolá delegát **Vystup**, kterému předá výsledek převedený do textové podoby. Protože volání **Vystup** lze provést jen když je opravdu připojena alespoň jedna metoda, musí být zároveň zajištěno, že k volání dojde opravdu jen pokud hodnota **Vystup** nebude rovna **null**.

```
public static void Soucet(int a, int b)
{
    if (Vystup != null)
    {
        Vystup((a + b).ToString());
    }
}
```

```
public static void Rozdil(int a, int b)
{
    if (Vystup != null)
    {
        Vystup((a - b).ToString());
    }
}
```

```
public static void Soucin(int a, int b)
{
    if (Vystup != null)
    {
        Vystup((a * b).ToString());
    }
}
```

```
public static void Podil(double a, double b)
{
    if (Vystup != null)
    {
        Vystup((a / b).ToString());
    }
}
```

V posledních verzích C# máme možnost toto volání zjednodušit. Můžeme využít operátor **?.**, který nahrazuje klasický operátor **.** a sám provádí kontrolu, zda proměnná vlevo od něj je skutečně objekt a nejedná se o **null**. Pokud proměnná obsahuje **null**, volání se dále neprovádí, pokud obsahuje reálný objekt, pak se zavolá metoda, která následuje za operátorem. V našem případě není na první pohled vidět, jak by se tento operátor dal využít, nicméně stačí se podívat do dokumentace a zjistíme, že v případě delegátů je přímé použití názvu delegátské proměnné jako volání metody (tj. forma použitá výše) stejné, jako bychom připojené metody vyvolali pomocí metody **Invoke()** delegáta. Ta už použití **?.** umožňovat bude. Všechny metody bychom tak mohli zjednodušit takto:

```
public static void Soucet(int a, int b) => Vystup?.Invoke((a + b).ToString());
public static void Rozdil(int a, int b) => Vystup?.Invoke((a - b).ToString());
public static void Soucin(int a, int b) => Vystup?.Invoke((a * b).ToString());
public static void Podil(double a, double b) => Vystup?.Invoke((a / b).ToString());
```

Volání této metody pak můžeme provádět přímo v event handlers jednotlivých tlačítek na formuláři (vytvoříme je opět dvojklikem na každé tlačítko). Rozdíl je však v tom, že tentokrát už nebudeme očekávat výsledek metody a přiřazovat ho přímo do **txtVysledek**. V této podobě tedy jen vezmeme vstupní hodnoty, převedeme je z textového řetězce na čísla a zavoláme výpočetní metody ze třídy **Vypocet**.

```
private void btnSoucet_Click(object sender, EventArgs e)
{
    Vypocty.Soucet(Convert.ToInt32(txtA.Text), Convert.ToInt32(txtB.Text));
}

private void btnRozdil_Click(object sender, EventArgs e)
{
    Vypocty.Rozdil(Convert.ToInt32(txtA.Text), Convert.ToInt32(txtB.Text));
}

private void btnSoucin_Click(object sender, EventArgs e)
{
    Vypocty.Soucin(Convert.ToInt32(txtA.Text), Convert.ToInt32(txtB.Text));
}

private void btnPodil_Click(object sender, EventArgs e)
{
    Vypocty.Podil(Convert.ToInt32(txtA.Text), Convert.ToInt32(txtB.Text));
}
```

Pokud bychom teď přepnuli spouštění na tento projekt a aplikaci spustili, zjistili bychom, že se po stisknutí tlačítka nic neděje. To je dáno tím, že jsme sice nadefinovali všechny potřebné metody a jejich funkcionalitu, ale nepropojili jsme formulář s delegátem. V této chvíli se po stisknutí tlačítka sice vyvolá výpočetní metoda, ale protože k delegátu není připojena žádná metoda (jeho hodnota je **null**), tak se výpočet a zobrazení výsledku neprovede. Musíme tedy provést poslední úpravu třídy **Form1**. Na konec jejího konstruktoru je nutné přidat příkaz, který do delegátské vlastnosti **Vystup** uloží odkaz na naši metodu **ShowResult()**:

```
Vypocty.Vystup += ShowResult;
```

Konstruktor po úpravě tak bude vypadat takto:

```
public Form1()
{
    InitializeComponent();
    Vypocty.Vystup += ShowResult;
}
```

Všimněte si, že pro připojení stačí jako parametry za operátor **+=** napsat jméno metody, kterou chceme odkazovat. Delegát může odkazovat více metod najednou, takže pokud bychom přidali například další prvek, na který bychom chtěli výsledek zobrazovat, stačí vytvořit metodu, která se o to bude starat (podobně jako **ShowResult()**), připojit tuto metodu k delegátu výše uvedeným způsobem a po provedení výpočtu se výsledek zobrazí na obou místech.

Vlastní procvičování

1. Podobně jako u druhé podoby aplikace, doplňte třídu **Vypocty** u verze s delegáty tak, aby se z formuláře při volání výpočetní metody předávaly jen hodnoty **string**.
2. Zkuste rozšířit uživatelské rozhraní aplikace z třetího projektu tak, aby se výsledek mohl zobrazovat i na jiném místě, než jen v **txtVysledek** (např. v popisku), doplňte vše co je potřeba (náповědu najdete na konci předchozí sekce) a vyzkoušejte, že jedním voláním opravdu dokážete vypsát výsledek na víc míst najednou.