Učební text pro výuku odborného předmětu
Programování a vývoj aplikací

Ing. Jan Kopřiva

2015

Obsah

Úvod ... 4
Teoretický úvod do programování ... 4
1 Vstup a výstup v konzoli ... 7
1.1 Práce s proměnnými v konzoli ... 9
1.2 Přetypování ... 10
1.3 Komentáře ... 11
2 Hodnotové a referenční datové typy ... 13
2.1 Numerické datové typy ... 13
2.2 Textové datové typy ... 14
2.3 Logický datový typ ... 15
2.4 Výčtové datové typy ... 15
2.5 Struktury ... 16
2.6 Hodnotové a referenční datové typy ... 17
3 Aritmetické operátory a logické operátory ... 19
4 Konverze typů ... 21
4.1 Explicitní přetypování ... 22
5 Řízení toku programu ... 24
5.1 Přiřazení ... 24
5.2 Porovnání ... 24
5.3 Větvení ... 25
5.4 If-else ... 26
5.5 If-else if-else ... 26
5.6 Switch ... 28
5.7 While ... 29
5.8 Do-while ... 31
5.9 Cyklus for ... 33
5.10 Foreach ... 34
6 Pole ... 35
7 Seznámení s prostředím Windows Forms ... 38
8 Dialogy ... 43
8.1 MessageBox ... 43
8.2 OpenFileDialog a SaveFileDialog ... 46
8.3 FolderBrowserDialog ... 48
8.4 ColorDialog ... 48

9 Výjimky 49
10 Práce s textovými soubory 51
11 Kolekce 54
11.1 Negenerické kolekce 54
11.2 Generické kolekce 56
12 Vlastní definice objektů 58

Použité značky

Stěžejní látka – nutné věnovat zvýšenou pozornost

Doplňující informace, rady či poznámky

Krátká úloha k procvičení probrané látky

Shrnutí kapitoly

Závěrečná úloha k procvičení látky celé kapitoly

Úvod

C# je moderní objektově orientovaný programovací jazyk platformy .NET. Syntakticky vychází zejména z C/C++ a Javy a spolu s nimi patří mezi v současnosti nejpoužívanější programovací jazyky. V C# je možné vyvíjet mnoho typů aplikací pod různými operačními systémy. V předkládaném textu se pro jednoduchost omezíme na tvorbu aplikací využívajících pro interakci s uživatelem konzoli (tzv. Console Application) a formuláře systému Windows (Windows Forms Application). Pro vývoj budeme používat prostředí Microsoft Visual Studio, popsané koncepty je nicméně možné aplikovat i při tvorbě jiných typů aplikací v jiných vývojových prostředích.

Teoretický úvod do programování

Při zkoumání možností jazyku C# nás bude provázet množství odborných teoretických pojmů, jejich znalost není pro praktické programování ve většině případů stěžejní, umožňuje nám nicméně lépe se v problematice orientovat a stát se tak lepšími programátory. Většina pojmů bude vysvětlena v odpovídajících kapitolách dále v textu, již nyní budeme nicméně několik málo z nich potřebovat a s jejich definicemi se tedy seznámíme zde.

Algoritmus

Tento termín je z hlediska teorie programování relativně podstatný a bude nás provázet až do konce naší pomyslné společné cesty zkoumání jazyku C#. Existuje mnoho různých definic, které jej popisují, pro naší potřebu však bude zcela postačovat tato:

Algoritmus je návod či postup řešení určité úlohy nebo problému, skládající se z více kroků a splňující určité podmínky.

Uvedené podmínky se v různých definicích drobně liší, zpravidla jsou nicméně uváděny požadavky následující (některé definice uvádějí i další):

Konečnost – algoritmus musí skončit v konečném počtu kroků

Determinovanost / určitost – každý krok algoritmu musí být přesně definován a musí být vždy jasné, jaký krok bude následovat

Hromadnost / obecnost – algoritmus musí řešit obecný soubor problémových úloh (například sečtení dvou celých čísel), ne jen jeden konkrétní problém (sečtení čísel 1 a 2)

Resultativnost – kroky algoritmu vedou k určitému řešení, které je výsledkem (řešením) algoritmu

Překlad/kompilace

Převedení kódu programu do strojově čitelné (spustitelné) formy. V C# a některých dalších jazycích nedochází při překladu k převedení přímo do spustitelného kódu, ale do tzv. mezijazyku (CIL – Common Intermediate Language), který se teprve po spouštění aplikace překládá pomocí virtuálního stroje do formy spustitelné procesorem.

Proměnná

Termín proměnná je čtenáři zajisté znám z matematiky a stejně jako v ní, i v programování označuje reprezentaci určité hodnoty či objektu pomocí symbolického označení. Proměnné mají v jazyce C# tak zvané pevné typování, což znamená, že při jejich vytváření je explicitně třeba uvést, jaké typy hodnot či objektů budou reprezentovat a toto určení již není možné později změnit.

Proměnnou obecně vytvoříme voláním

```
typ jméno;
```

nebo

```
typ jméno = hodnota;
```

Typů proměnných existuje velké množství (je možné vytvářet i uživatelsky definované datové typy), z toho důvodu je jim věnována celá kapitola 2. Jména proměnných píšeme vždy s malým počátečním písmenem.

Konstanta

Další termín vycházející z matematiky a jeho význam v programování je opět stejný – konstantou je určitá hodnota, u níž nemůžeme měnit její výši. V C# lze konstantu vytvořit stejně jako proměnnou, pouze je před požadovaný datový typ třeba napsat klíčové slovo `const`. Definice konstanty by tedy vypadala následovně:

```
const typ jméno = hodnota;
```

Jména konstant píšeme, stejně jako v případě proměnných, vždy s malým počátečním písmenem.

Syntaxe a sémantika

Poslední dva pojmy, s nimiž se seznámíme předtím, než se pustíme do hlubšího zkoumání jazyku C#, jsou syntaxe a sémantika. Oba termíny představují vlastnosti obecných jazyků, tedy ne jenom jazyků programovacích, ale i „běžných“, užívaných v mezilidské komunikaci.

První uvedené slovo se vztahuje k formě či tvaru jazyka – syntaxe udává konstrukce, které jsou v daném jazyku přípustné, a to specifikací gramatických pravidel, povolených znaků, interpunkce a dalších faktorů. Syntaxe českého jazyku nám tedy mimo jiné udává, že každá věta má začínat velkým písmenem a končit některým z interpunkčních znamének. Syntaxe jazyku C# nám podobně například udává, že zápis:

```
proměnná1 = proměnná2 + 5;
```

je zcela korektní, zatímco zápis:

```
proměnná2 + 5 = proměnná1;
```

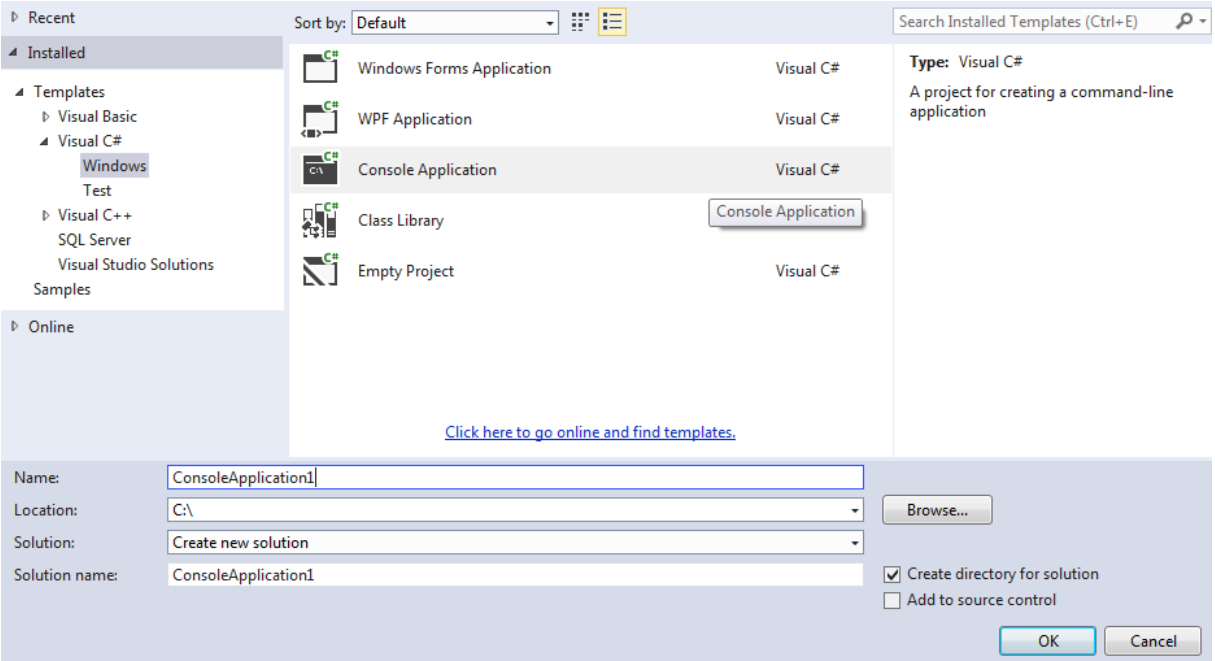
je nekorektní. Je tomu tak proto, že gramatika jazyku C# specifikuje, že proměnná, do níž se zapisuje, se vždy nachází na levé straně operátoru rovná se (=), zatímco zapisovaná hodnota se nachází na jeho pravé straně. Obrácený zápis není v C# přípustný (totéž platí pro většinu ostatních moderních programovacích jazyků).

Syntaxe tedy specifikuje, jaké konstrukce se mohou v jazyce objevovat. Sémantika je potom tím, co těmto konstrukcím dává význam – pří vytvoření proměnné (viz výše) například uvádíme její jméno, které tak získává sémantický smysl. Stejně tak sémantika určuje význam níže uvedených struktur, jako je `if`, nebo `while`.

1 Vstup a výstup v konzoli

Následující kapitola obsahuje popis prostředí Visual Studio a vysvětluje způsob, jakým lze v jazyce C# získávat/číst vstupy uživatele a vypisovat mu výstupy, jinými slovy, jak s ním komunikovat. Po přečtení této kapitoly budete schopni vytvářet v něm jednoduché textové programy.

Nejjednodušší formou aplikace, kterou lze v jazyce C# vytvářet, je aplikace konzolová – umožňuje interakci s uživatelem založenou na jednoduchém textovém rozhraní, jehož programové ovládání je relativně snadné. Konzolovou aplikaci lze vytvořit vybráním možnosti File->New Project z menu prostředí Visual Studio a zvolení varianty Console Application. V posledních 3 řádcích na zobrazeném formuláři je možno zvolit cestu, kde bude aplikace vytvořena, jméno řešení (Solution), které ji bude obsahovat (jedno řešení může obsahovat větší množství projektů/aplikací) a označení vlastní aplikace.



Po potvrzení formuláře se zobrazí editor kódu, obsahující automaticky vygenerovanou kostru programu. Prvních několik řádků kódu začíná klíčovým slovem `using`, které je následováno různými názvy tzv. jmenných prostorů. Jmenný prostor (`namespace`) je základní organizační jednotkou jazyku C#, v níž může být obsažen kód, to znamená, že v jednotlivých jmenných prostorech se mohou nacházet různé aplikace a jejich komponenty. Slovo `using` pak vyjadřuje připojení určitého jmenného prostoru k naší aplikaci, čímž nám dává možnost všechny v něm obsažené aplikace a programové komponenty využívat.

Jako příklad můžeme uvést řádek

```
using System;
```

připojující k naší aplikaci jmenný prostor `System`, což je základní jmenný prostor rozhraní .NET a obsahuje mnoho komponent, které budeme využívat. Mimo jiné se v něm nachází také třída (pro začátek můžeme třídy považovat za základní funkční bloky, více viz níže) `Console`, kterou budeme používat při interakci s uživatelem. Povšimněte si, že když do složených závorek pod řádkem

```
static void Main(string[] args)
```

napíšeme slovo `Console`, zbarví se světle modře. Je tomu tak proto, že v rámci naší aplikace, respektive připojeného jmenného prostoru `System`, existuje třída tohoto jména. Pokud smažeme řádek `using System`, slovo `Console` bude zbarvené černě, neboť žádná třída toho jména nebude v aplikaci existovat.

Pro použití tříd a komponent z nepřipojených jmenných prostorů je možné napsat název jmenného prostoru následovaný tečkou a jménem komponenty. Nemáme-li tedy připojený jmenný prostor `System`, můžeme psát `System.Console` a výsledek by byl stejný, jako při připojení uvedeného jmenného prostoru a napsání samotného slova `Console`. Z uvedeného je zřejmé, že připojování jmenných prostorů nám zjednodušuje práci, protože při použití komponent z nich nemusíme psát před jejich označení název prostorů, v nichž jsou obsaženy.

Podívejme se nyní na zbytek kódu aplikace. Pod řádky připojujícími jmenné prostory se nachází řádek začínající slovem `namespace`, následovaným jménem aplikace, které jsme zvolili při jejím vytvoření. Tento řádek vytváří vlastní, námi pojmenovaný jmenný prostor, který obsahuje vše, co se nachází ve složených závorkách pod ním (první a poslední složená závorka), což je v tuto chvíli pouze 6 vnitřních řádků kódu. První z nich obsahuje klíčové slovo `class`, následované slovem `Program` – říká, že v našem jmenném prostoru je vytvářena třída jménem Program, která obsahuje vše, co se nachází v páru složených závorek pod ní (detailní proniknutí do problematiky tříd není pro tuto chvíli stěžejní, podstatné je pouze uvědomit si, že všechen kód, který budeme v rámci konzolových aplikací vytvářet, budeme vždy psát do těla třídy Program, tedy do složených závorek, patřících k ní). Jediným obsahem třídy `Program` je po vytvoření aplikace metoda `Main`, uvozená klíčovými slovy `static` a `void` a následována závorkami s textem `string[] args` (problematika metod je detailněji popsána níže, prozatím můžeme metodu definovat jako pojmenovaný blok kódu, nebo podprogram). Metoda `Main` je část kódu, která se provede po spuštění konzolové aplikace a tedy kód, který budeme vytvářet, budeme psát do složených závorek pod ní.

První program, který společně vytvoříme, se často označuje jako aplikace „Hello world“, jediná její funkce je vypsání tohoto textu uživateli. Do metody `Main` stačí pro její správnou funkci doplnit následující dva řádky:

```
Console.WriteLine("Hello world");
Console.ReadLine();
```

Pomocí prvního řádku vypisujeme do konzole text Hello world, pomocí druhého řádku pak říkáme programu, aby nepokračoval, dokud uživatel nestiskne klávesu enter.

Metoda, kterou používáme v prvním řádku (`WriteLine`) slouží k provedení výstupu do konzole, tedy vypsání určitého textu do ní – dokáže, mimo jiné, vypsat téměř libovolný text zapsaný v uvozovkách do závorek za jejím jménem.

Metoda ve druhém řádku slouží k získání uživatelského vstupu z konzole – v tuto chvíli se vstupem nijak nepracujeme, v následujících programech se jej nicméně naučíme využít.

Úloha – vytvořte aplikaci, která vypisuje do dvou řádků vaše jméno.

1.1 Práce s proměnnými v konzoli

Stejným způsobem, jakým můžeme vypisovat pevně daný text, můžeme do konzole vypsat také určité proměnné. Prvním typem proměnných, který si představíme, je `string`. Jedná se o datový typ, schopný ukládat textové řetězce. Abychom odlišili příkaz jazyku od textu, uvádíme v kódu textové řetězce v uvozovkách, jak ukazuje následující program:

```
static void Main(string[] args)
{
    string s = "Ahoj";
    Console.WriteLine(s);
    Console.ReadLine();
}
```

V uvedeném kódu jsme vytvořili proměnnou typu `string`, do níž jsme uložili text „Ahoj“, který jsme následně vypsali do konzole. Textové řetězce pro vypsání nemusíme nutně zadávat již při psaní programů, pomocí metody `ReadLine` můžeme od uživatele získat jím zadaný text ve formě typu `string`. Můžeme tak napsat například následující program, který uživatele požádá o vstup a následně mu jej vypíše zpět:

```
static void Main(string[] args)
{
    Console.WriteLine("Zadej text");
    string s = Console.ReadLine();
    Console.WriteLine(s);
    Console.ReadLine();
}
```

Aby byla práce s textem v jazyce C# jednodušší, umožnili nám jeho tvůrci skládat textové řetězce (jako text, i jako proměnné) za sebe pomocí operátoru plus (+). Můžeme tedy napsat kód

```
string s = "jak se máš?";
string pozdrav = "Ahoj" + ", " + s;
```

který do proměnné `pozdrav` uloží text „Ahoj, jak se máš?“.

V tuto chvíli již umíme vypsat uživateli požadavek na vstup, ten načíst, přidat k němu jiný text a zpětně vypsat. Neměl by pro nás tedy být problém pochopit následující program, v němž požádáme uživatele o zadání jména a aktuálního dne v týdnu a následně mu vypíšeme pozdrav obsahující tyto informace.

```
static void Main(string[] args)
{
    Console.WriteLine("Zadej své jméno:");
    string jmeno = Console.ReadLine();
    Console.WriteLine("Zadej dnešní den:");
    string den = Console.ReadLine();
    Console.WriteLine("Ahoj " + jmeno + ", dnes je " + den + ". Přeji krásný den!");
    Console.ReadLine();
}
```

Úloha – vytvořte aplikaci, která požádá uživatele o zadání tří slov a vypíše je v obráceném pořadí.

1.2 Přetypování

Již jsme zmínili, že v jazyce C# existuje více datových typů – s většinou z nich se blíže seznámíme později, pro tuto chvíli si však představíme ještě jeden z nich – datový typ `int`. Na rozdíl typu `string` nám umožňuje ukládat ne text, ale celá čísla. Proměnnou typu `int` vytvoříme obdobně, jako proměnnou typu `string`, čísla, která do ní budeme vkládat, nicméně nebudou v uvozovkách (pamatujte, nejedná se o text), jak ukazuje následující příklad:

```
int i = 5;
```

V tuto chvíli je na místě otázka, proč je vhodné rozlišovat datový typ pro celá čísla. Celé číslo bychom do proměnné `string` mohli bez potíží uložit, přináší tedy tento postup nějaké výhody? Odpověď zní, samozřejmě, ano. S většinou z nich se seznámíme až v průběhu dalšího zkoumání jazyku C#, pro začátek nicméně stačí zmínit, že v programu se s číslem v proměnné typu `string` dá pracovat pouze jako s textem, ne jako s číslem – nemohli bychom jej například používat v rámci numerických operací, což ukazuje následující příklad:

```
string a = "5";
string b = "3";
string c = a + b;
```

V proměnné `c` je po proběhnutí výše uvedeného kódu text „53“, vzhledem k tomu, že operátor plus mezi dvěma proměnnými typu string způsobí jejich spojení.

```
int a = 5;
int b = 3;
int c = a + b;
```

Po proběhnutí tohoto kódu, je v proměnné `c` číslo 8, vidíme tedy jeden rozdílů v chování proměnných `string` a `int`.

Je vhodné zmínit, že metoda `WriteLine` může přijímat a následně zobrazovat jako text i proměnné typu `int`. Můžeme tedy psát:

```
int i = 9;
Console.WriteLine(i);
```

(což vypíše do konzole text „9“) nebo

```
Console.WriteLine(7);
```

(což vypíše do konzole text „7“).

Pro schopnost psát jednoduché aplikace pracující s datovým typem `int` zbývá již jen seznámit se se způsobem, kterým můžeme číslo získat od uživatele. Vzhledem k tomu, že metoda `ReadLine` vrací hodnotu datového typu `string`, nemůžeme pouze napsat řádek

```
int i = Console.ReadLine();
```

Pro změnu datového typu hodnoty je třeba provést tzv. konverzi neboli přetypování. Jeden ze způsobů, kterým můžeme konverze/přetypování provádět, je využít metod třídy `Convert`. Jednou z těchto metod je i `ToInt32`, která dokáže převádět různé datové typy na `int` (s dalšími metodami třídy `Convert` se seznámíme později). Funguje podobně, jako metoda `WriteLine` v tom ohledu, že proměnnou, s níž má pracovat, umístíme dovnitř jejích závorek. Předchozí kód tedy můžeme upravit na zápis

```
string s = Console.ReadLine();
int i = Convert.ToInt32(s);
```

případně jej můžeme zkrátit do podoby

```
int i = Convert.ToInt32(Console.ReadLine());
```

Výše uvedenou aplikaci komunikující s uživatelem můžeme tedy upravit tak, aby uživatele požádala o věk a sdělila mu, jak bude starý příští rok, jak ukazuje následující příklad.

```
static void Main(string[] args)
{
    Console.WriteLine("Zadej své jméno:");
    string jmeno = Console.ReadLine();
    Console.WriteLine("Kolik je ti let?");
    string textovyVek = Console.ReadLine();
    int ciselnyVek = Convert.ToInt32(textovyVek);
    int vekPristiRok = ciselnyVek + 1;
    Console.WriteLine("Ahoj " + jmeno + ", příští rok ti bude " + vekPristiRok);
    Console.ReadLine();
}
```

Úloha – vytvořte aplikaci, která požádá uživatele o dvě čísla a vypíše jejich součet.

1.3 Komentáře

Vzhledem k tomu, že v kódu aplikace je občas vhodné (například při provádění složitějších operací) si poznamenat např. co jsme chtěli daným kódem provést, umožňuje nám C# vkládat do kódu tzv. komentáře. Jde v podstatě o texty, které jsou při překladu ignorovány. Můžeme je vytvářet dvěma způsoby:

1. Jednořádkové komentáře se vkládají za dvě lomítka (`//`), jako v.:

   ```
   Console.ReadLine(); //Aplikace čeká na stisknutí tlačítka enter.
   ```

2. Víceřádkové komentáře se vkládají mezi kombinaci znaků lomítko-hvězdička (`/*`) a hvězdička-lomítko (`*/`), jak ukazuje následující příklad:

   ```
   static void Main(string[] args)
   {
       /*
        Toto je metoda Main
        uvnitř které
        je pouze
        víceřádkový
        komentář
        */
   }
   ```

Shrnutí kapitoly – Základní organizační jednotkou jazyku C# je `namespace`, jednotky tohoto typu můžeme připojovat do našeho programu pomocí klíčového slova `using`, čímž nám bude umožněno používat všechny prvky obsažené v daném `namespace`. Výstup do konzole je v C#

realizován nejjednodušeji pomocí metody `WriteLine` třídy `Console`, vstup, který vrací hodnotu datového typu `string`, pak pomocí její metody `ReadLine`. Jednotlivé hodnoty typu `string` je možné skládat za sebe pomocí operátoru plus (+). Hodnoty typu `string` je možné přetypovat na typ `int` pomocí metody `ToInt32` třídy `Convert`. Komentáře do kódu můžeme psát za dvě lomítka (`//`), nebo mezi znaky lomítko-hvězdička (`/*`) a hvězdička-lomítko (`*/`).

Závěrečná úloha – Vytvořte aplikaci, která od uživatele příjme jeho jméno, den v měsíci a měsíc a popřeje mu pomocí odpovídajících dat krásný dnešek, zítřek a pozítřek den. Na vstup Karel, 10, leden by měl výstup vypadat například následovně:

„Zdravím uživatele Karel, přeji ti krásný 10., 11. a 12. leden“.

2 Hodnotové a referenční datové typy

Již v úvodní části jsme se seznámili se skutečností, že v jazyce C# můžeme data ukládat do tzv. proměnných. Zjistili jsme též, že proměnné mají vždy určitý datový typ, který specifikuje, jaký druh dat je možné do které proměnné uložit. Prozatím jsme se seznámili se dvěma datovými typy – jedním pro uložení textových hodnot (`string`) a jedním pro uložení číselných hodnot (`int`). V této kapitole si představíme zbytek tzv. základních datových typů, zjistíme, jaký je mezi nimi rozdíl a také jakým způsobem pracuje C# s pamětí počítače. Datové typy se totiž mimo jiné liší tím, kolik místa v paměti pro data alokují, čímž do značné míry omezují hodnoty, které v nich mohou být uloženy. Zjistíme tedy, jak nejlépe využít paměť počítače pro data, která budeme v našich programech potřebovat.

2.1 Numerické datové typy

Prvními datovými typy, s nimiž se blíže seznámíme, jsou typy pro uložení a reprezentací čísel – tzv. numerické datové typy. Je možné je dělit do dvou základních typů – na celočíselné datové typy a datové typy pro ukládání čísel s plovoucí desetinnou čárkou. Již z jejich označení je zřejmý rozdíl mezi nimi – do druhých jmenovaných je možno uložit i čísla desetinná – taková, která mají část za desetinnou čárkou. Nejprve se podíváme na celočíselné datové typy.

Každý ze zde uvedených numerických datových typů má přesně stanovený rozsah paměti, který zabírá – tím je omezeno i množství hodnot, které může reprezentovat. Aby bylo uvedené množství o něco větší, umožňuje C# pro každý typ zvolit, zda má být se znaménkem (tzv. signed – rozsah je dělen na dvě části – polovina je záporná, polovina kladná), či bez něj (tzv. unsigned – číslo je vždy kladné, celý rozsah se počítá od nuly výše). Seznam celočíselných datových typů, na němž jsou rozdíly v rozsazích dobře vidět, uvádí následující tabulka.

Název	Popis	Rozsah
sbyte	8-bitové (1 byte) číslo se znaménkem	-128 až 127
byte	8-bitové (1 byte) číslo bez znaménka	0 až 255
short	16-bitové (2 byty) číslo se znaménkem	-32 768 až 32 767
ushort	16-bitové (2 byty) číslo bez znaménka	0 až 65 535
int	32-bitové (4 byty) číslo se znaménkem	-2 147 483 648 až 2 147 483 647
uint	32-bitové (4 byty) číslo bez znaménka	0 až 4 294 967 295
long	64-bitové (8 bytů) číslo se znaménkem	-9 223 372 036 854 775 808 až 9 223 372 036 854 775 807
ulong	64-bitové (8 bytů) číslo bez znaménka	0 až 18 446 744 073 709 551 615

Je dobré si povšimnout, že s výjimkou typu `byte` začíná označení neznaménkové varianty datového typu vždy písmenem „u“, značícím „unsigned“ (viz výše).

Proměnné všech datových typů se vytváří stejným způsobem, s jakým jsme se seznámili v minulé kapitole. Proměnné různých typů tedy můžeme vytvořit například takto:

```
byte b = 8; //vytvoření proměnné typu byte a přiřazení hodnoty 8 do ní
short s = -12345; //vytvoření proměnné typu short a přiřazení hodnoty do ní
uint u = 123456789; //vytvoření proměnné typu uint a přiřazení hodnoty do ní
long l = 789456123789456123; //vytvoření proměnné typu long a přiřazení hodnoty do ní
```

V zájmu minimalizace nároků aplikace na paměť je vždy vhodné volit pro proměnné nejmenší datový typ, který umožňuje uložení všech hodnot, s nimiž budeme v daném programu pracovat.

Úloha – vyzkoušejte si vytvoření všech výše uvedených typů proměnných a zjistěte, jak reagují na vložení hodnot, které jsou mimo jejich rozsahy.

O výše uvedenou minimalizaci nároků na paměť bychom se měli snažit také při využití numerických datových typů ukládajících čísla s plovoucí desetinnou čárkou. V jejich případě v C# neexistují znaménkové a neznaménkové varianty, jazyk nabízí pouze 3 typy, lišící se rozsahem. Jejich přehled ukazuje následující tabulka.

Název	Popis	Rozsah
float	32-bitové (4 byty) desetinné číslo	$\pm 1{,}5 * 10^{-45}$ až $\pm 3{,}4 * 10^{38}$
double	64-bitové (8 bytů) desetinné číslo	$\pm 5{,}0 * 10^{-324}$ až $\pm 1{,}7 * 10^{308}$
decimal	128-bitové (16 bytů) desetinné číslo	$(-7.9 * 10^{28}$ až $7{,}9 * 10^{28}) / (10^{0 \text{ až } 28})$

Přiřazení do proměnných s plovoucí desetinnou čárkou se provádí analogicky, jako v případě celočíselných proměnných. Je nicméně třeba mít na paměti, že základním z těchto tří typů je `double`, a pokud v kódu napíšeme desetinné číslo (jako oddělovač desetinných míst se v C# používá tečka), jazyk jej jako `double` automaticky rozpozná. Hodnotu typu `double` však nemůžeme vložit do proměnné typu `float` nebo `decimal`. Následující kód je tedy chybný:

```
float f = 2.71;
decimal m = 3.14;
```

Abychom specifikovali, že desetinné číslo je jiného typu, než `double`, musíme za něj doplnit písmeno, určující jeho typ – tím je v případě typu `float` písmeno `f` a v případě typu `decimal` písmeno `m`, jak ukazuje následující příklad.

```
float f = 1.123f; //vytvoření proměnné typu float a přiřazení hodnoty do ní
double d = 4.445656; //vytvoření proměnné typu double a přiřazení hodnoty do ní
decimal m = 456.123m; //vytvoření proměnné typu decimal a přiřazení hodnoty do ní
```

Úloha – vyzkoušejte si vytvoření všech výše uvedených typů proměnných a zjistěte, jak reagují na vložení hodnot, které jsou mimo jejich rozsahy. Vyzkoušejte, zda jdou proměnné různých typů přiřadit jedna do druhé.

2.2 Textové datové typy

Textové datové typy umožňují ukládat, jak již jejich jméno napovídá, textová data. Ta mohou mít podobu víceznakových řetězců či jednoznakových hodnot. S textovým datovým typem umožňujícím ukládání víceznakových hodnot jsme se seznámili již v minulé kapitole – je jím `string`. Základní práce s ním je, jak jsme zjistili, poměrně jednoduchá – do proměnné typu `string` můžeme pomocí operátoru rovná se uložit de-facto libovolný text (který však musí být uzavřen do uvozovek).

```
string s1 = "Toto je text, který se uloží do proměnné s"; // Korektní zápis
string s2 = Toto je text, který vyvolá chybové hlášení; // Nekorektní zápis
```

Jak jsme zjistili v minulé kapitole, proměnné typu `string` je možno převádět na proměnné typu `int` a obráceně. Tyto dva převody nejsou jediné, které C# nabízí, dalo by se dokonce říci, že většinu základních datových typů je možno poměrně jednoduchým způsobem převádět mezi sebou.

Vzhledem k tomu, že tato problematika si zaslouží bližší prozkoumání, věnujeme ji 4. kapitolu předloženého textu.

Podobně, jako řetězcové proměnné, fungují proměnné pro uložení jednotlivých znaků. V jazyce C# je jejich typ označován jako `char`. Abychom odlišili jednoznakový zápis typu `char` a `string` v kódu, jsou první jmenované místo uvozovkami ohraničeny apostrofami (`'`).

```
char c = 'a'; // Korektní zápis
char d = "d"; // Nekorektní zápis - vkládáme hodnotu typu string do proměnné typu char
```

Pro úplnost je vhodné uvést, že proměnné typu `char` jsou 16-bitové a znaky do nich ukládané jsou kódovány pomocí kódování Unicode, proměnné typu `string` jsou pak de-facto řetězce tvořené proměnnými `char`.

2.3 Logický datový typ

C#, stejně jako většina ostatních programovacích jazyků, umožňuje kromě textových a numerických hodnot ukládat také hodnoty logické, tedy hodnoty:

- Pravda (`true`)
- Nepravda (`false`)

Datovým typem, do nějž je možno logické hodnoty (a žádné jiné) ukládat, je `bool`. S využitím logických proměnných se blíže seznámíme v průběhu dalších kapitol, pro tuto chvíli bude postačující předvedení jejich korektního vytvoření a nastavení v kódu.

```
bool b = true; // Hodnota true značí pravdu
bool c = false; // Hodnota false značí "nepravdu"
```

Popisem logických typů končí naše exkurze mezi základní datové typy nabízené jazykem C#, vyjma nich je však možno používat i některé další.

2.4 Výčtové datové typy

Pro některé úlohy je vhodné přesně specifikovat hodnoty, jichž může určitá proměnná nabývat (podobně, jako v případě logických datových typů, které mohou nabývat pouze hodnot `true` a `false`). Například v případě, kdy bychom v programu chtěli reprezentovat jednotlivé dny v týdnu, by bylo vhodné, kdybychom mohli vytvořit datový typ, který by mohl ukládat pouze hodnoty `pondělí` až `neděle`. C# nám do jisté míry možnost tvorby vlastních datových typů dává a to pomocí tak zvaných výčtových datových typů. Pro jejich použití je třeba stanovit pouze název datového typu a výčet (odtud výčtové) všech hodnot, kterých může nabývat. Obecný zápis výčtového typu vypadá následovně.

```
enum Jméno { hodnota1, hodnota2, hodnota3,... };
```

Pro nás je podstatné, že definice výčtových typů se musí nacházet mimo metody, píšeme je tedy např. nad hlavičku metody `Main`. Vytvoření datového typu, který by umožnil ukládat výše uvedené dny v týdnu, a jeho použití by mohlo vypadat například následovně:

```
enum Dny { pondělí, úterý, středa, čtvrtek, pátek, sobota, neděle };

static void Main(string[] args)
{
```

```
    Dny d1 = Dny.pondělí;
    Dny d2 = Dny.pátek;
}
```

Vlastní výčtové typy v aplikacích používáme jen relativně zřídka, C# sám však výčtové datové typy obsahuje a znalost principu jejich funkce je tedy pro dobrou orientaci v jazyku velmi žádoucí.

2.5 Struktury

Posledními hodnotovými typy, které si představíme, jsou struktury. Struktury jsou v podstatě obalem, který pojmenujeme a do něhož můžeme vložit několik proměnných (i různých typů), které též pojmenujeme. Tyto proměnné se často označují jako „členy“ struktury. Uvedený přístup se může hodit zejména ve chvílích, kdy chceme ukládat více informací o jednom předmětu či situaci – u člověka bychom například mohli chtít ukládat jeho jméno, příjmení a věk. Obecný zápis struktury je následující.

```
struct JménoStruktury
{
    public datový_typ1 JménoVnitřníProměnné1;
    public datový_typ2 JménoVnitřníProměnné2;
}
```

Je vhodné povšimnout si, že před uvedení datového typu proměnných je třeba vložit modifikátor `public`. Ten specifikuje, že k dané proměnné budeme moci přistupovat mimo samotný zápis struktury (viz 12. kapitola). Vytvoření popsané struktury by mohlo vypadat takto:

```
JménoStruktury jménoStrukturníProměnné = new JménoStruktury();
```

nastavení vnitřních proměnných struktury pak probíhá následovně.

```
jménoStrukturníProměnné.JménoVnitřníProměnné1 = hodnota1;
jménoStrukturníProměnné.JménoVnitřníProměnné2 = hodnota2;
```

Stejně jako výčtové datové typy, i struktury je třeba vytvářet mimo metody, vhodným místem pro definici struktur je tak například prostor nad hlavičkou metody `Main`. Níže uvedená ukázka demonstruje definici výše popsané struktury reprezentující člověka a nastavení jejích vnitřních proměnných.

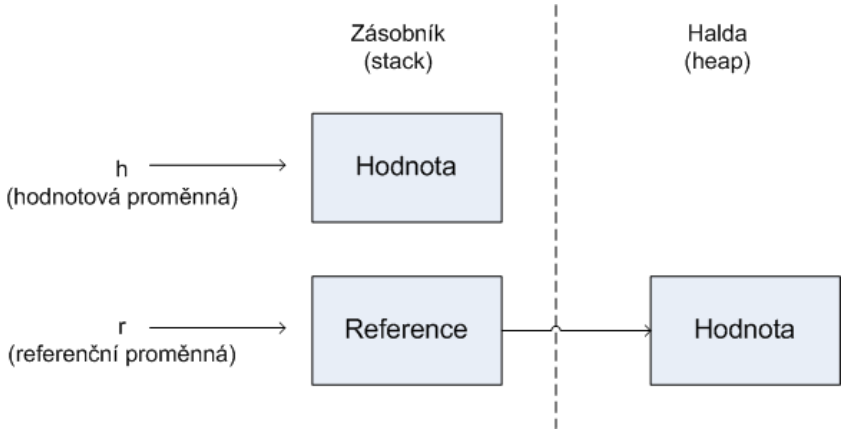
```
struct Clovek
{
    public string Jmeno;
    public string Prijmeni;
    public int Vek;
}

static void Main(string[] args)
{
    Clovek c = new Clovek();
    c.Jmeno = "Anna";
    c.Prijmeni = "Malá";
    c.Vek = 35;
}
```

Výhodou struktur je, že všechny členské proměnné jsou přístupné pomocí jména jedné společné „obalové“ proměnné a je tedy na první pohled jasné, že popisují stejný prvek. Podobně jako v případě výčtových typů, ani struktury nebudeme v našich programech zřejmě vytvářet příliš často,

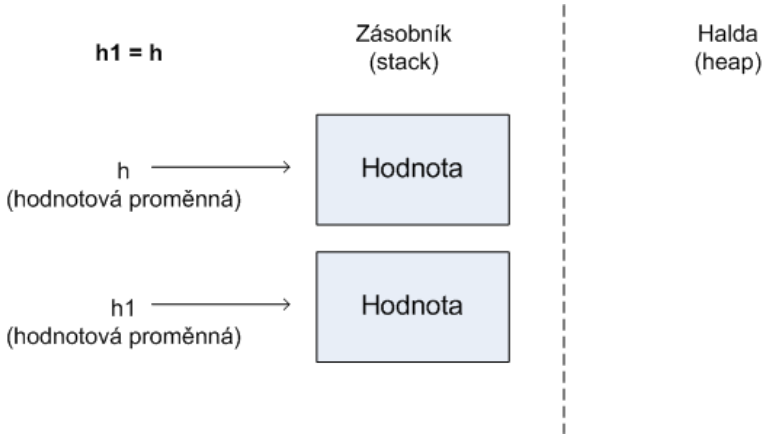
C# sám nicméně struktury obsahuje a znalost práce s nimi je pro nás tedy, jako v případě výčtových typů, velmi žádoucí.

2.6 Hodnotové a referenční datové typy

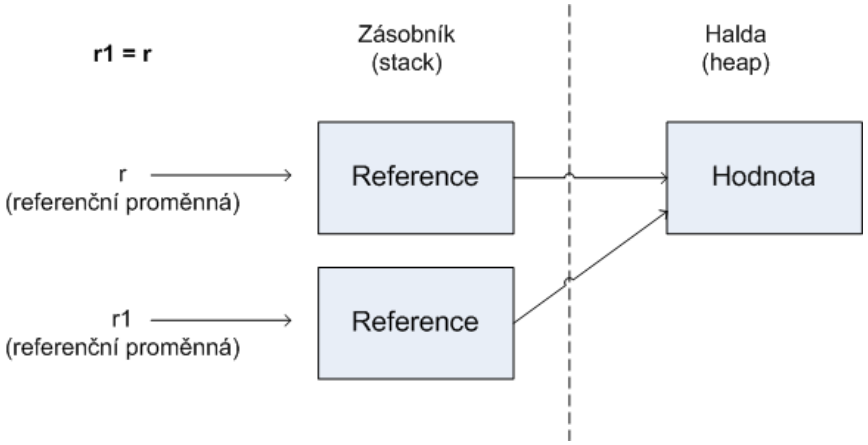
Všechny výše popsané datové typy patří mezi typy hodnotové. Krom nich se v jazyce C# můžeme setkat také s datovými typy referenčními. Rozdíl mezi oběma skupinami je ve způsobu, kterým pracují s pamětí. Hodnotové datové typy jsou ukládány v části paměti, která se označuje zásobník (stack). Při vytvoření nové proměnné se v této části paměti alokuje prosto o odpovídající velikosti a na něj následně odkazuje jméno proměnné. Princip referenčních datových typů je jiný – jejich obsah je ukládán v části paměti, která se označuje halda (heap), v zásobníkové části paměti je ale při jejich vytvoření alokován prostor, do něhož se uloží tak zvaná reference (odkaz), která ukazuje na část hady, kde je uložen jejich obsah. Jméno proměnné v popsaném případě odkazuje na referenci v zásobníku, ne na vlastní obsah.



Pro nás je uvedený rozdíl podstatný zejména ve chvíli, kdy chceme položit jednu proměnnou rovnu druhé. V případě hodnotových datových typů vše proběhne tak, jak bychom čekali, obsah proměnné se zkopíruje na místo v paměti, na které ukazuje nová proměnná.



U referenčních typů však dojde pouze ke zkopírování reference, na níž ukazuje nová proměnná, avšak obsah proměnné zůstane na původním místě. To znamená, že pokud provedeme v obsahu nově vytvořené proměnné změnu, ta se promítne i do staré proměnné, neboť reference obou odkazují na stejný obsah.



Uvedený rozdíl není v praxi nijak problematické v aplikacích zohlednit, je však třeba držet jej v patrnosti. My se s ukázkou práce s těmito typy seznámíme v poslední kapitole, které je věnována objektům, ty jsou totiž svou povahou typy referenčními.

Shrnutí kapitoly – V jazyce C# existuje několik základních datových typů, které se liší v hodnotách, které mohou ukládat, základní dělení je na datové typy

- numerické, umožňující ukládání celých a desetinných čísel,
- znakové, ukládající textové řetězce nebo samostatné znaky a
- logické, ukládající boolovské hodnoty `true` a `false`.

Numerické datové typy se liší rozsahem, který je do nich možno ukládat. Nejpodstatnější jsou `int` (pro celočíselné hodnoty) a `double` (pro hodnoty s plovoucí desetinnou čárkou). Vyjma základních datových typů se v C# můžeme setkat s typy komplexnějšími, např. výčtovými datovými typy a strukturami. Všechny uvedené datové typy jsou hodnotové, což znamená, že jejich data jsou uložena přímo v paměti, na níž odkazuje proměnná, v C# však existují ještě typy referenční, u nichž proměnná odkazuje na tzv. referenci, která teprve odkazuje na paměť s daty.

Závěrečná úloha – Vyzkoušejte, pro které datové typy platí, že jejich proměnné lze mezi sebou vzájemně ukládat.

3 Aritmetické operátory a logické operátory

V následujícím textu se naučíme provádět aritmetické a logické operace v C#, což nám umožní užívat v našich programech jednoduché i složitější výpočty a pracovat s logickými hodnotami.

Při práci s numerickými datovými typy umožňuje C# použití běžných aritmetických binárních (pracujících se dvěma vstupy – operandy) operátorů plus (+), mínus (-), krát (*) a děleno (/). Jejich použití je poměrně intuitivní a můžeme tedy například používat kód

```
int i = 2 + 5;
```

či

```
double d = 2.456 * 0.12;
```

Uvedené operátory lze použít de-facto se všemi základními numerickými datovými typy, je nicméně dobré vést v patrnosti fakt, že v případě některých (např. `byte`) dochází k automatickému přetypování na jiný datový typ (`int`).

Výsledné hodnoty jsou omezeny přesností a rozsahem vyplývajícím z typu výstupní hodnoty, viz minulá kapitola.

Vyjma uvedených nabízí C# ještě jeden, v programování často používaný, binární numerický operátor. Je jím modulo (%), jehož výstupem je zbytek po celočíselném dělení prvního operandu druhým operandem. Velmi často se v praxi používá například ke zjištění, zda je číslo sudé (`vstup%2` – pokud je výsledek roven jedné, je číslo liché, pokud je roven nule, je číslo sudé).

Kromě aritmetických operátorů obsahuje C# předdefinované také tři binární logické operátory AND (&), OR (|), XOR (^) a jeden unární – pracující s jedním operandem – logický operátor NOT (!), jejichž funkce odpovídá jmenovaným logickým operacím, jak ukazuje následující pravdivostní tabulka.

Operand 1	Operand 2	AND (&)	OR (|)	XOR (^)	NOT(!) – op. 1
true (1)	true (1)	true (1)	true (1)	false (0)	false (0)
true (1)	false (0)	false (0)	true (1)	true (1)	false (0)
false (0)	true (1)	false (0)	true (1)	true (1)	true (1)
false (0)	false (0)	false (0)	false (0)	false (0)	true (1)

Použití binárních logických operátorů není omezeno pouze na operandy datového typu `bool`, jak by se možná dalo předpokládat, ale lze je použít i s numerickými datovými typy. V takovém případě je nicméně třeba vzít v úvahu rozdílnou funkci výsledného kódu při použití jednotlivého (&, |, ^) a zdvojeného operátoru (&& a ||). Při použití zdvojených operátorů probíhá odpovídající operace s logickými hodnotami operandů jako vstupy (operandy tedy musí být typu `bool`),

```
bool a = true;
bool b = false;
bool c = a || b; // V proměnné c je hodnota true
```

zatímco při použití operátorů jednotlivě probíhá daná operace nad bitovým zápisem operandů – lze tedy pomocí nich provádět bitové operace jako logické sčítání, násobení a XOR s numerickými datovými typy.

```
int i = 31;
int j = 16;
```

```
int k = i ^ j; // V proměnné k je hodnota 15
```

Všechny popsané binární operátory lze užít spolu se znakem = ve zkráceném zápisu, kdy se výsledek operace ukládá do prvního z uvedených operandů. Zápis

```
i += j;
```

je tedy ekvivalentní zápisu

```
i = i + j;
```

a zápis

```
a &= b;
```

je ekvivalentní zápisu

```
a = a & b;
```

Shrnutí kapitoly – Jazyk C# nám umožňuje s některými numerickými typy používat klasické binární aritmetické operátory plus, mínus, krát a děleno. Vyjma nich je možné pracovat také s operátorem modulo, jehož výsledkem je zbytek po celočíselném dělení prvního operandu druhým. Vedle aritmetických operátorů C# nabízí operátory logické (binární a unární), které umožňují jednoduchým způsobem provádět s operandy operace AND, OR, XOR a NOT.

Závěrečná úloha – Vytvořte aplikaci, která od uživatele přijme dvě čísla a vrátí jejich součet, rozdíl, součin, podíl a modulo.

4 Konverze typů

Konverze datových typů jsou mechanismem, který umožňuje převádět obsah proměnné určitého typu do obsahu proměnné typu jiného. V této kapitole se naučíme převádět jednotlivé typy mezi sebou, což nám umožní mnohem jednodušším způsobem získávat hodnoty jiných datových typů než `string` od uživatele.

S jedním příkladem převodů mezi datovými typy jsme se setkali již v první kapitole, kde jsme pomocí metody `ToInt32` třídy `Convert` převedli obsah proměnné typu `string` na datový typ `int`.

```
string s = "534";
int i = Convert.ToInt32(s); // V proměnné i je nyní hodnota 534
```

Třída `Convert` nám nabízí i další možnosti převodů, pro nás zřejmě nejzajímavější z jejích zbývajících metod je `ToDouble`, která, jak již název napovídá, dokáže převádět proměnné jiných typů na datový typ `double` (je dobré povšimnout si, že metody třídy `Convert` umí převádět nejenom z proměnných typu `string`, ale také mnoha jiných). Metodu `ToDouble` můžeme použít obdobně, jako `ToInt32` ve výše uvedeném příkladu.

```
string s = "534,456";
double d = Convert.ToDouble(s); // V proměnné d je nyní hodnota 534.456
```

Obdobně, jako v uvedených příkladech můžeme metody třídy `Convert` použít například pro převod obsahu proměnné typu `int` na datový typ `char` (metoda `ToChar`), či převod proměnné typu `double` na typ `bool` (metoda `ToBoolean`).

Kromě třídy `Convert` nabízí C# ještě tři základní možnosti převodu hodnot různých typů mezi sebou. Dvě z nich (metody `Parse` a `TryParse`) umožňují provádět převody hodnot typu `string` na většinu jiných datových typů a třetí (metoda `ToString`) umožňuje převést hodnotu proměnné de-facto libovolného datového typu na typ `string`.

Metody `Parse` a `TryParse` jsou vlastní třídám základních datových typů – implementují je mimo jiné třídy `int`, `double`, `string` a všechny další základní typy, popsané ve druhé kapitole. Výsledkem převodu metodou `Parse` je vždy hodnota typu, odpovídajícího třídě, nad níž byla metoda volána (`int.Parse` tedy vrátí hodnotu typu `int`, `float.Parse` vrátí hodnotu typu `float`, apod.), vstupem však musí být vždy proměnná typu `string`.

```
string s1 = "123";
int i = int.Parse(s1); // V proměnné i je hodnota 123
string s2 = "456,123";
float f = float.Parse(s2); // V proměnné f je hodnota 456.123
string s3 = "true";
bool b = bool.Parse(s3); // V proměnné b je hodnota true
```

Metoda `TryParse` je o něco komplexnější, její návratovou hodnotou je vždy `bool`, určující, zda byl převod úspěšný (`true`) či neúspěšný (`false`). Tuto hodnotu nemusíme nutně využívat. `TryParse` tedy nevrací převedenou hodnotu jako návratovou. Obecný zápis této metody je poměrně netradiční:

```
datový_typ.TryParse(stringProPřevedení, out proměnnáDatovéhoTypuNaKterýPřevádíme);
```

Klíčové slovo `out` v tomto zápisu specifikuje, že proměnná, která je za ním uvedena, bude uvnitř metody změněna. Právě do ní se v tomto případě ukládá převedená hodnota. Abychom mohli metodu `TryParse` použít, je třeba nejprve vytvořit prázdnou proměnnou odpovídajícího typu, do níž budeme vkládat převedenou hodnotu.

```
string s = "123";
int i;
int.TryParse(s, out i); // V proměnné i je hodnota 123
```

Použití metody `TryParse` je vhodné, pokud se chceme pokusit převést vstup, u něhož si nejsme jisti, zda je možné jej na námi požadovaný datový typ převést. Pokud se totiž převod nepovede, `TryParse` pouze vrátí `false`, ale aplikace pokračuje v běhu. Kdyby obdobný případ, při němž se převod nepovede, nastal při použití metody `Parse`, nebo některé metody třídy `Convert`, dojde k výjimce a aplikace „spadne“. Vhodné je například použití metody `TryParse` je ve spojení s konstrukcí `if` (viz níže).

Poslední převodní metodou, kterou si představíme blíže, je již zmíněná `ToString`. Jde o metodu, jejíž použití je velmi jednoduché, vzhledem k tomu, že ji implementují v podstatě všechny datové typy. Pro převod de-facto libovolné proměnné na `string` tak stačí použít kód:

```
jménoPřeváděnéProměnné.ToString();
```

V praxi bychom mohli tuto metodu využít například následovně:

```
int i = 123;
string s1 = i.ToString(); // V proměnné s1 je hodnota "123"
bool b = false;
string s2 = b.ToString(); // V proměnné s2 je hodnota "false"
char c = 'x';
string s3 = c.ToString(); // V proměnné s3 je hodnota "x"
```

Úloha – Vytvořte aplikaci, která přijme od uživatele 3 desetinná čísla a vrátí jejich součet a součin.

4.1 Explicitní přetypování

Pro převody mezi některými typy je možné použít ještě tzv. explicitního přetypování. Pro naše programy je tento způsob převodu ne zcela vhodný, je dobré nicméně alespoň zmínit jeho základní princip. Pomocí explicitního přetypování v podstatě říkáme, že chceme na obsah určité proměnné pohlížet, jako by byl jiného datového typu (nejde tedy o převod v pravém slova smyslu). Obecný zápis explicitního přetypování zní:

```
(typNaKterýChcemePřetypovat)jménoProměnnéKterouChcemePřetypovat;
```

V praxi se tento způsob změny typů používá například v případech, kdy by klasické přetypování neuspělo, například z důvodu snahy o uložení hodnoty většího rozsahu než proměnná umožňuje, jako v případě uložení hodnoty 400 do proměnné typu `byte` (která, jak víme, má rozsah 0 až 255). Při pokusu o klasický převod by aplikace skončila výjimkou (došlo by k jejímu pádu).

```
int i = 400;
byte b = Convert.ToByte(i); // Zde dojde k pádu aplikace
```

Pokud bychom ve stejném případě použili explicitní přetypování, na paměť, v níž je uložena proměnná i se „podíváme“, jako by byla typu `byte` (`byte` má 8 bitů, takže budeme sledovat pouze posledních 8 bitů proměnné `i`) a takto získaný obsah uložíme do nové proměnné. V popisovaném případě bychom tak již nesledovali 9. bit a bity vyšší, ale jen posledních 8 bitů a jejich hodnota je 144.

Následující tabulka ukazuje postup explicitního přetypování proměnné `int i`, ukládající hodnotu 400.

Bit	9.	8.	7.	6.	5.	4.	3.	2.	1.
i	1	1	0	0	1	0	0	0	0
									= 400

(byte) i	1	1	0	0	1	0	0	0	0
									= 144

```
int i = 400;
byte b = (byte)i; // Do proměnné b se uloží hodnota 144
```

Explicitní přetypování se v praxi také užívá při pokročilé práci s objekty (viz kapitola 12) při převodu mezi jednotlivými třídami, pro naše potřeby je nicméně znalost výše uvedeného použití naprosto dostačující.

Shrnutí kapitoly – Obsah proměnných určitého datového typu můžeme v C# převádět na jiné datové typy. Převody z/na většinu datových typů je možno provést pomocí metod třídy `Convert`. Vedle nich můžeme využít metody datových typů `Parse` a `TryParse`, které umožňují převedení hodnoty typu `string` na de-facto libovolný jiný datový typ. Pro vybrané typy převodů je pak možno užít tzv. explicitního přetypování.

Závěrečná úloha – Vytvořte aplikaci, která požádá uživatele o, a následně přijme, dvě hodnoty typu `bool`, hodnotu typu `char`, hodnotu typu `sbyte` a hodnotu typu `decimal`.

5 Řízení toku programu

V páté kapitole se seznámíme se základními konstrukcemi, které můžeme v jazyce C# užívat k řízení toku programu. Po přečtení této kapitoly budeme schopni reagovat různým způsobem na různé vstupy uživatele nebo opakovat část kódu dokud nebude splněna určitá podmínka, což nám umožní psát plnohodnotné a reálně využitelné aplikace.

5.1 Přiřazení

Základní operací v jazyce C#, s níž jsme se již seznámili, je přiřazení, realizované pomocí operátoru rovná se (=). Pomocí něj můžeme proměnným nastavovat hodnoty a to tak, že napíšeme jméno proměnné, operátor rovná se a následně hodnotu, na niž chceme danou proměnnou nastavit. Tento postup je možné provádět pro libovolný datový typ (viz výše), jak ukazují následující dva řádky, v nichž nastavujeme hodnotu celočíselné proměnné jménem `i` na hodnotu 5 a textovou/řetězcovou proměnnou označenou `s` na hodnotu „abcdef“:

```
int i = 5;
string s = "abcdef";
```

5.2 Porovnání

Podobně, jako přiřadit hodnotu do proměnné, je možno proměnné a konstanty také porovnávat. Operátorem pro porovnání jsou dva znaky rovná se za sebou (==). Výsledkem porovnání je hodnota datového typu `bool`, která nabývá hodnoty `true` (pravda), pokud jsou obě strany porovnání shodné, a hodnoty `false` (nepravda), jsou-li rozdílné. V následujícím řádku by byla výsledkem porovnání hodnota `true`:

```
1 == 1
```

zatímco následující řádek by byl vyhodnocen jako nepravda, tedy `false`:

```
1 == -1
```

Stejným způsobem jako lze porovnávat obsahy proměnných a konstant na rovnost je možné zkoumat i zda je určitá proměnná či konstanta větší než jiná (operátor >) či menší než jiná (operátor <). Uvedená porovnání opět vrací hodnotu typu `bool`, tedy `true`, je-li porovnání pravdivé, a `false`, je-li nepravdivé. Následující řádek by byl vyhodnocen na hodnotu `true`:

```
1 < 2
```

a následující řádek na hodnotu `false`:

```
1 > 2
```

Operátory pro porovnání je možné kombinovat a ptát se, zda je jedna proměnná větší než nebo rovna jiné (operátor >=), menší než nebo rovna jiné (operátor <=), nebo se nerovná jiné (operátor !=). Návratovým typem je opět `bool`, následující řádky by tak byly vyhodnoceny jako hodnota `true`:

```
2 >= 2
2 >= 1
4 <= 4
3 <= 4
2 != 1
```

a níže uvedené řádky pak jako hodnota `false`:

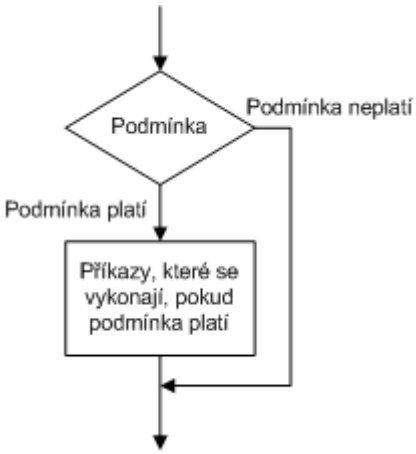
```
1 >= 2
3 >= 4
1 != 1
2 != 2
```

5.3 Větvení

První jazykovou strukturou, umožňující řídit tok programu, kterou si představíme, je tzv. jednoduché větvení, v jazyce C# označované `if` (což přeloženo znamená „pokud“, nebo „když“). Jeho syntaxe je poměrně jednoduchá:

```
if (podmínka)
{
        //co se stane, pokud podmínka platí
}
```

Podmínkou může být libovolná operace či porovnání, které vrací hodnotu typu `bool` (tedy lze jej vyhodnotit jako pravdivé či nepravdivé). Pokud je podmínka vyhodnocena jako pravdivá, provedou se příkazy zapsané ve složených závorkách, pokud ne, jsou tyto příkazy přeskočeny, jak ukazuje následující obrázek



a následující příklady.

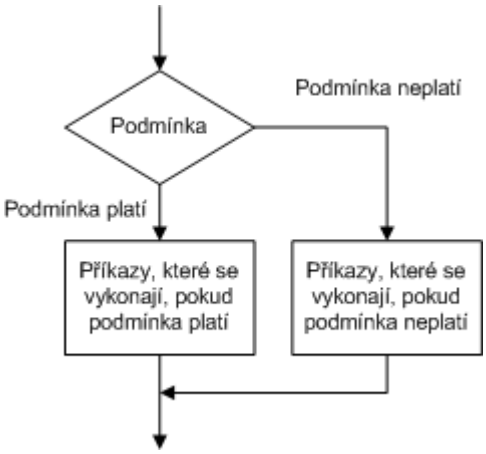
```
if (1 == 1)
{
    //tento kód se provede vždy - podmínka platí
    Console.WriteLine("Jedna se rovná jedné.");
}


if (1 < 0)
{
    //tento kód se neprovede nikdy - podmínka neplatí
    Console.WriteLine("Jedna je menší než nula.");
}


int i = Convert.ToInt32(Console.ReadLine());
if (i == 5)
{
    //tento kód se provede, pokud v proměnné i bude hodnota 5
```

```
    Console.WriteLine("Uživatel zadal hodnotu 5.");
}
```

Úloha – Vytvořte aplikaci, která požádá uživatele o zadání kladného čísla a pokud zadá číslo záporné, informuje jej o chybě.

5.4 If-else

Větvení pomocí konstrukce `if` může mít také o něco složitější podobu, v níž specifikujeme nejenom akce, které se mají vykonat, pokud podmínka platí, ale také akce, které se mají vykonat, je-li podmínka vyhodnocena jako nepravdivá. Konstrukci `if` lze za tímto účelem doplnit o větev `else`, jejíž obsah se vykoná při nesplnění podmínky.



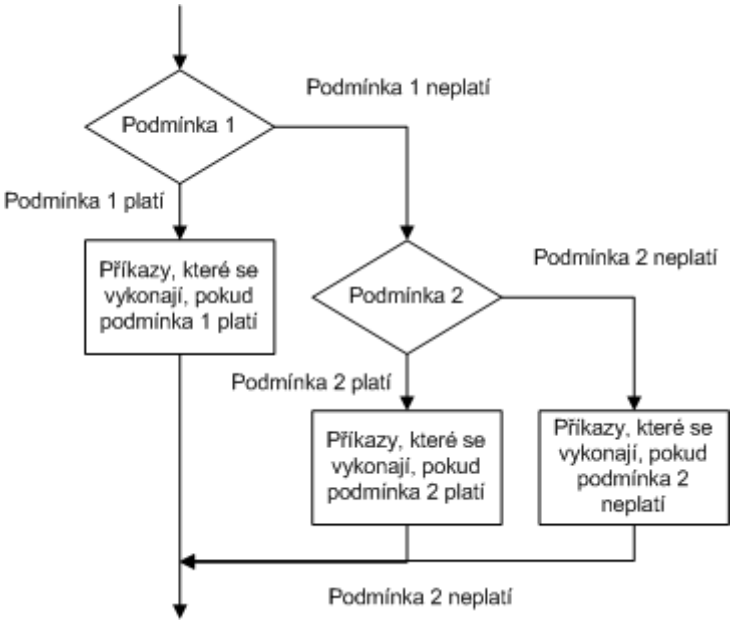
V níže uvedeném příkladu by se tak do konzole vypsal text „Hodnota je vyšší nebo rovna pěti“.

```
int i = 5;
if (i <= 4)
{
    //Nastane, pokud podmínka platí
    Console.WriteLine("Hodnota je nižší než 5");
}
else
{
    //Proběhne, je-li podmínka neplatná
    Console.WriteLine("Hodnota je vyšší nebo rovna pěti");
}
```

Úloha – Vytvořte aplikaci, která přijme od uživatele numericky vyjádřený den v týdnu, pokud uživatel zadá 1, popřeje mu „Krásné pondělí“, pokud zadá jiné číslo, popřeje mu „Krásný den“.

5.5 If-else if-else

Větvení pomocí konstrukce `if` však může mít i vícenásobnou podobu, v níž pokud neplatí původní podmínka, testuje se podmínka druhá, pokud neplatí ani ta, testuje se podmínka třetí, atd. Maximální počet podmínek v jedné `if` konstrukci není v jazyce C# omezen. V kódu toto větvení začíná jednoduchou konstukcí `if` a dále je realizováno pomocí konstrukce `else if`, za níž následuje další podmínka.



Tak jako v případě jednoduchého větvení, i v popsaném případě je blok `else` volitelný.
Ukázka bez bloku `else`:

```
string s = Console.ReadLine();
if (s == "a")
{
    Console.WriteLine("Zmáčknul/a jste písmeno A.");
}
else if (s == "b")
{
    Console.WriteLine("Zmáčknul/a jste písmeno B.");
}
Console.ReadLine();
```

Ukázka s blokem `else`:

```
string s = Console.ReadLine();
if (s == "a")
{
    Console.WriteLine("Vložil/a jste písmeno A.");
}
else if (s == "b")
{
    Console.WriteLine("Vložil/a jste písmeno B.");
}
else
{
    Console.WriteLine("Nevložil/a jste ani písmeno A ani písmeno B.");
}
Console.ReadLine();
```

Úloha – Vytvořte aplikaci, která požádá uživatele o zadání dvou čísel a vypíše mu, zda je první větší než druhé, druhé větší než první, či zda jsou si obě čísla rovna.

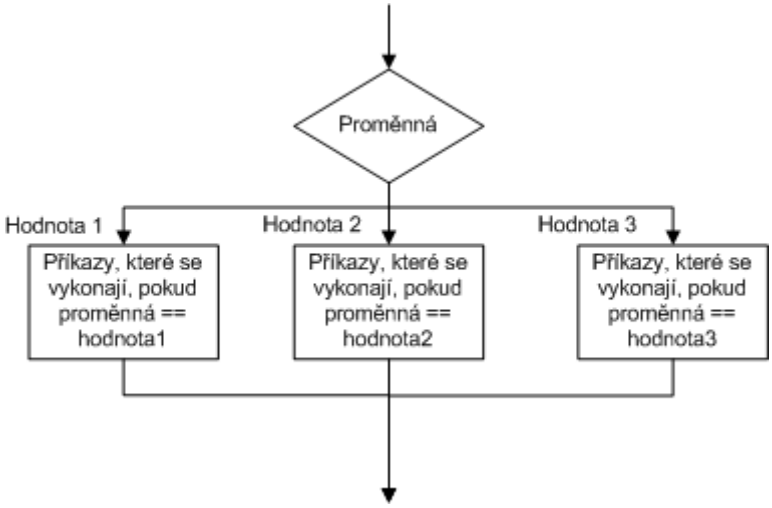
5.6 Switch

Další konstrukcí, pomocí níž je možné realizovat vícenásobné větvení, je switch. Funguje obdobně, jako konstrukce `if-else if` (případně konstrukce `if-else if-else`), rozdílem je pouze umožněný způsob zápisu podmínky, respektive porovnání. Zatímco v konstrukci `if` (a souvisejících blocích) jsme schopni použít libovolné podmínky, jejichž vyhodnocení vede na logický typ `bool` (`true`, nebo `false`), v případě konstrukce `switch` specifikujeme pouze jednu podmínku, v níž můžeme testovat pouze rovnost dvou hodnot. Na rozdíl od konstrukce `if` v ní tak nemůžeme provést například porovnání dvou proměnných na „je větší“ či „je menší“, apod.

Obecný zápis konstrukce switch vypadá následovně:

```
switch (proměnná) //za proměnnou můžeme dosadit hodnotu libovolného datového typu
{
    case hodnota1:
        //kód, který se vykoná, pokud platí, že proměnná je rovna hodnotě1
        break;
    case hodnota2:
        //kód, který se vykoná, pokud platí, že proměnná je rovna hodnotě2
        break;
    default: // nepovinný blok, funguje stejně, jako blok else u konstrukce if
        //kód, který se vykoná, pokud neplatí ani jedna z uvedených možností
        break;
}
```

Klíčové slovo `case` specifikuje konkrétní případ, který se má porovnávat a je následováno hodnotou. Klíčové slovo `break` specifikuje ukončení konkrétní větve (lze jej též použít pro ukončení běhu cyklů – viz níže). Poslední z klíčových slov – `default` – uvádí blok, který se vykoná, pokud neplatí ani jeden z uvedených případů/porovnání. Je tedy obdobou bloku `else`.



Praktický příklad a ekvivalentní konstrukci `if` můžete vidět níže.

Switch:

```
int i = 2;
switch (i)
{
    case 1:
        Console.WriteLine("V proměnné i je hodnota 1.");
        break;
```

```
    case 2:
        Console.WriteLine("V proměnné i je hodnota 2.");
        break;
    default:
        Console.WriteLine("V proměnné i je není ani hodnota 1, ani hodnota 2.")
        break;
}
```

If:

```
int i = 2;
if (i == 1)
{
    Console.WriteLine("V proměnné i je hodnota 1.");
}
else if (i == 2)
{
    Console.WriteLine("V proměnné i je hodnota 2.");
}
else
{
    Console.WriteLine("V proměnné i je není ani hodnota jedna, ani hodnota dvě.");
}
```

Chování obou výše uvedených programů by bylo zcela stejné, často je tedy pouze na nás, jako programátorech, který z popsaných postupů zvolíme pro řešení určitého problému. Pro doplnění uvedeme ještě použití konstrukce `switch` pro zjištění hodnoty proměnné typu `string` (postup je v podstatě stejný, jako pro datový typ `int`, pouze nesmíme zapomenout, že hodnoty typu `string` jsou vždy „obaleny“ uvozovkami).

```
Console.WriteLine("Jste muž či žena?");
string s = Console.ReadLine();
switch (s)
{
    case "muž":
        Console.WriteLine("Dobrý den, pane!");
        break;
    case "žena":
        Console.WriteLine("Dobrý den, paní!");
        break;
    default:
        Console.WriteLine("Nezvolil/a jste ani jednu přípustnou možnost");
        break;
}
```

Úloha – Vytvořte aplikaci, která při zadání čísla od 1 do 7 vypíše odpovídající den v týdnu.

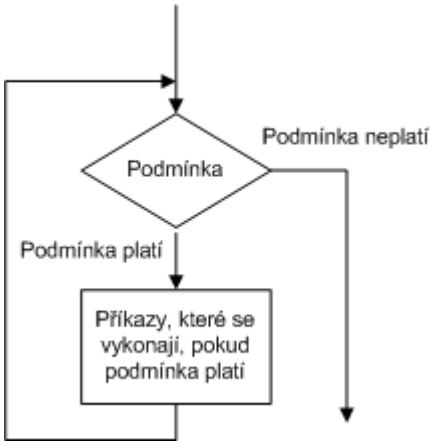
5.7 While

Struktura `while` je prvním z cyklů, který si představíme. Jak označení cykly napovídá, jedná se o struktury, umožňující (cyklické) opakování vybraných kroků, respektive částí kódu. Cyklus `while` má zápis podobný jazykové struktuře `if`:

```
while (podmínka)
{
    // kód, který se bude vykonávat, dokud podmínka platí
}
```

Na rozdíl od struktury `if` se však obsažený kód neprovede (pokud podmínka platí) pouze jednou, ale bude se provádět opakovaně, dokud podmínka platí. Z uvedeného je zřejmé, že pokud nechceme, aby cyklus probíhal nekonečně dlouho (existují aplikace, kdy je to žádoucí, ty jsou nicméně mimo náš aktuální zájem), musíme uvnitř cyklu podmínku, která jej řídí, určitým způsobem ovlivňovat.

Většinou je tak v podmínkové části cyklu použita proměnná, která je následně měněna uvnitř cyklu. Tuto proměnnou někdy nazýváme řídící a její změnu uvnitř cyklu (pokud se jedná o numerickou proměnnou) pak inkrement (pokud k řídící proměnné přičítáme), nebo dekrement (pokud od ní odečítáme).



Konečná podoba cyklu while tak může být například následující:

```
int i = 0;
while (i < 5)
{
    //Zde můžeme vložit kód, který se provede 5x
    i++;
}
```

Řídící proměnnou je v tomto případě `i` a v každém průběhu cyklu ji inkrementujeme o jedničku. Kód obsažený na místě komentáře by se v ukázce provedl 5x a to z následujícího důvodu:

- Na počátku je `i` rovno nule, nula je menší než pět (podmínka platí), tudíž vkročíme do cyklu a provedeme první průchod, při němž k `i` přičteme jedna.
- Na počátku druhého cyklu je tedy v proměnné `i` uložena hodnota jedna, ta je stále menší než pět (podmínka tedy stále platí), provedeme druhý průchod cyklem, v něm přičteme k `i` jedna.
- Na počátku třetího cyklu je v proměnné `i` hodnota 2, podmínka tedy stále platí (2 < 5), projdeme cyklem a přičteme k `i` číslo jedna.
- Na počátku čtvrtého cyklu je v proměnné `i` číslo 3, podmínka platí, přičteme k `i` jedničku.
- Na počátku pátého cyklu obsahuje `i` hodnotu 4, podmínka stále platí (4 < 5), přičteme tedy jedničku.
- Na počátku šestého opakování cyklu je v proměnné `i` hodnota 5, podmínka tedy již neplatí (není pravdou, že 5 < 5), z cyklu tedy vyskočíme a po šesté se neprovede.

V praxi tedy můžeme cyklus `while` použít k provedení určitých kroků přesně daný počet opakování, jak ukazují následující příklady, v nichž vypisujeme desetkrát pozdrav „Ahoj“,

```
int i = 0;
while (i < 10)
{
    Console.WriteLine("Ahoj!");
    i++;
}
```

čísla od jedné do sta a

```
int i = 1;
while (i <= 100)
{
    Console.WriteLine(i.ToString());
    i++;
}
```

čísla od dvaceti do nuly s odečítáním čísla 2

```
int i = 20;
while (i >= 0)
{
    Console.WriteLine(i.ToString());
    i = i - 2;
}
```

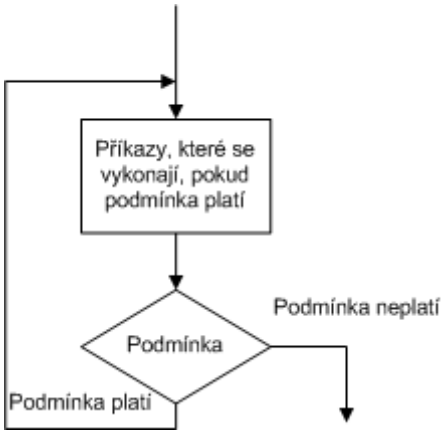
Cyklus `while` se však primárně hodí k jinému účelu – lze jej snadno použít pro předem neurčený počet opakování kódu (pak o `while` hovoříme jako o cyklu s dynamicky určeným počtem opakování), například v případech, kdy očekáváme od uživatele určitý vstup, jak ukazuje následující ukázka:

```
Console.WriteLine("Zadejte slovo 'pes'");
string s = Console.ReadLine();
while (s != "pes") //Cyklus se bude opakovat, dokud v proměnné s nebude text pes
{
    Console.WriteLine("Udělal/a jste chybu. Zadejte slovo 'pes'");
    s = Console.ReadLine();
}
Console.WriteLine("Zadal/a jste slovo správně.");
```

Úloha – Vytvořte aplikaci (sčítačku) která bude sčítat uživatelem zadávaná čísla, dokud nezadá nulu, a následně vypíše výsledek.

5.8 Do-while

Obdobně, jako výše popsaný cyklus `while`, funguje cyklická struktura `do-while`. Rozdíl mezi těmito cykly je v podstatě jen jeden – zatímco konstrukce while začíná podmínkou a ta se tak kontroluje na počátku (před prvním průchodem cyklu), konstrukce `do-while` obsahuje podmínku až na samém svém konci, ke kontrole splnění podmínky tedy dojde až po prvním průchodu cyklem.



To znamená, že obsah cyklu `do-while` se vykoná vždy alespoň jednou – i tehdy, pokud neplatí podmínka v něm uvedená. Zápis cyklu `do-while` je následující:

```
do
{
    // kód, který se bude vykonávat, dokud podmínka platí
} while (podmínka);
```

Jeho rozdílné chování od cyklu `while` ukazuje následující příklad.

While:

```
int i = 0;
while (i > 0)
{
    i--;
    Console.WriteLine("while"); // Tento text se nevypíše ani jednou
}
```

Do-while:

```
int i = 0;
do
{
    i--;
    Console.WriteLine("do-while"); // Tento text se vypíše jedenkrát
} while (i > 0);
```

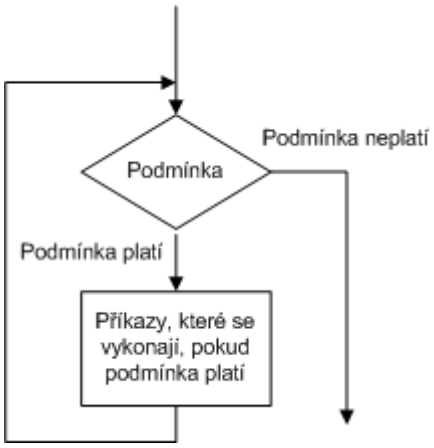
Použití konstrukce `do-while` je vhodné například tam, kde žádáme uživatele o vstup, s nímž následně pracujeme, jak ukazuje následující příklad:

```
string s;
do
{
    Console.WriteLine("Pro skončení aplikace napište písmeno 'k'.");
    s = Console.ReadLine();
} while (s != "k"); //Cyklus se bude opakovat, dokud v proměnné s nebude písmeno k
```

Úloha – Vytvořte aplikaci, která bude zobrazovat uživateli menu s možnostmi „1 – sčítání, 2 – odčítání, 3 – násobení, 4 – dělení a 5 – konec“, která skončí v případě, že uživatel zadá jako vstup číslo 5, jinak vždy znovu zobrazí menu.

5.9 Cyklus for

Dalším z cyklů, který si představíme, je `for`. Funguje podobně, jako cyklus `while` (lze prohlásit, že cokoli naprogramujeme s použitím jednoho, lze naprogramovat i s použitím druhého, totéž částečně platí i pro cykly `for` a `do-while`), jeho použití je však podstatně vhodnější pro algoritmy, u nichž předem známe přesný počet opakování, zatímco `while` se používá spíše v případech, kdy počet opakování předem neznáme. Není to však pravidlem. V hlavičce cyklu `for` se kromě podmínky (jde tedy o další cyklus s podmínkou na začátku) uvádí též definice řídící proměnné a inkrement.



Obecný zápis cyklu `for` vypadá následovně

```
for (definice řídící proměnné; podmínka; inkrement/dekrement)
{
    // kód, který se bude vykonávat, dokud podmínka platí
}
```

V praxi se jako řídící proměnná zpravidla užívá `int` pojmenovaný `i` a inkrementem je nejčastěji přičtení či odečtení o jedna:

```
for (int i = 0; i < 5; i++)
{
    //Zde uvedený kód se vykoná 5x
}
```

Pro demonstraci jeho funkce můžeme uvést následující příklady, v nichž jsou vypsána čísla od 50 do 100 s krokem +1,

```
for (int i = 50; i <= 100; i++)
{
    Console.WriteLine(i);
}
```

čísla od 10 do -20 s krokem -1 a

```
for (int i = 10; i > -21; i--)
{
    Console.WriteLine(i);
}
```

3x slovo C#

```
for (int i = 0; i < 3; i++)
{
```

```
    Console.WriteLine("C#");
}
```

Úloha – Vytvořte aplikaci, která přijme od uživatele tři čísla a následně vypíše sekvenci od prvního zadaného čísla do druhého se třetím číslem jako krokem.

5.10 Foreach

Poslední jazykovou konstrukcí, s níž se v této kapitole seznámíme, je `foreach` – cyklus, užívaný pro průchod víceprvkovými datovými typy (jedním z takových typů je pole, jehož popis najdete níže). Jeho zápis je následující:

```
foreach (datovýTyp proměnná in víceprvkovýDatovýTyp)
{
    //Zde napsaný kód se provede pro každý prvek datovéhoTypu
    //obsažený ve víceprvkovémDatovémTypu
}
```

Klíčové slovo `in` v uvedené definici dává v překladu naprosto jasný smysl – specifikuje, v jaké proměnné víceprvkového typu se mají hledat části odpovídající uvedenému datovému typu. Překlad celé konstrukce pak dává poměrně jasnou představu o jeho funkci – lze jej volně provést například jako: *„pro každou proměnnou daného datového typu, obsaženou ve víceprvkovém typu, proveď to, co je uvedeno v cyklu“*. Do uvedené proměnné se tedy v každém průběhu cyklu vloží jeden z prvků víceprvkového typu.

My v tuto chvíli známe jen jeden datový typ, který je tvořen menšími částmi jiného datového typu – je jím `string`, který je tvořen částmi typu `char`. Funkci cyklu `foreach` si tedy demonstrujeme na něm – následující kód vypíše řetězec „Hello world“ do sloupce (resp. jedno písmeno na jeden řádek):

```
string s = "Hello world";
foreach (char z in s) //Do proměnné z se bude po znacích přiřazovat obsah stringu s
{
    Console.WriteLine(z);
}
Console.ReadLine();
```

Použití této konstrukce je nejčastěji spojeno s již zmíněnými poli, vrátíme se k ní tedy v následující kapitole, až se naučíme s poli pracovat.

Shrnutí kapitoly – Tok programu můžeme v jazyce C# řídit pomocí několika typů konstrukcí, z nichž základními jsou jednoduché a vícenásobné větvení (konstrukce `if` , `if-else` a `switch`) a cykly (`while`, `do-while`, `for` a `foreach`). Větvení můžeme užít k provedení rozhodnutí na základě určité podmínky a cykly nám umožňují prováděni určitých kroků předem daný či dynamicky určený počet opakování.

Závěrečná úloha – Vytvořte aplikaci/kalkulačku, která umožní uživateli zadat operaci, která se má vykonat (+, -, *, /), následovanou dvěma čísly a vypíše výsledek dané operace. Aplikaci modifikujte tak, aby po výpisu výsledku vždy znovu požádala o vstupy a vypnula se pouze, pokud uživatel jako operaci vloží „X“.

6 Pole

Polem v programování označujeme souvislý paměťový blok, do něhož můžeme uložit předem určený počet proměnných stejného datového typu. Výhodou použití pole je, že všechny v něm uložené proměnné můžeme adresovat jménem pole a numerickým vyjádřením jejich pořadí v poli, tzv. indexem. V této kapitole se tedy naučíme efektivněji pracovat s větším počtem proměnných stejného typu.

Obecný zápis kódu pro vytvoření pole zní

```
datový_typ[] jménoPole = new typ[velikost_pole];
```

pokud bychom tedy například chtěli vytvořit pole schopné pojmout 5 čísel, můžeme použít následující zápis.

```
int[] cisla = new int[5];
```

Ten zajistí vytvoření pěti proměnných, k nimž se můžeme odkazovat pomocí jediného jména. K rozlišení mezi nimi slouží již zmíněný index. Ten nabývá hodnoty od 0 (nulovým indexem je označen první prvek) až do délky pole snížené o jedna (poslední prvek v poli o délce pět by měl index 4). Uvádíme jej za jménem pole v hranatých závorkách. Následující tabulka ukazuje možnosti adresace jednotlivých prvků ve výše vytvořeném poli „cisla“.

	První prvek	Druhý prvek	Třetí prvek	Čtvrtý prvek	Pátý prvek
Označení prvku	cisla[0]	cisla[1]	cisla[2]	cisla[3]	cisla[4]

Zápis i čtení jednotlivých prvků pole je možno provádět stejně, jako v případě klasické proměnné, jak ukazuje následující příklad.

```
int[] cisla = new int[3];
cisla[0] = 5;
cisla[1] = 7;
cisla[2] = cisla[0] + cisla[1];
```

Po proběhnutí uvedeného kódu by bylo v prvním prvku uloženo číslo 5, ve druhém číslo 7 a ve třetím číslo 12.

Jedna z výhod polí spočívá v možnosti adresace prvků de-facto pouze číselnou proměnnou. Tu totiž můžeme zvyšovat např. cyklem, pomocí čehož může být průchod celým polem o libovolné délce, resp. jeho naplnění či výpis, vcelku jednoduchou záležitostí. Nejčastěji se k popsanému účelu užívá cyklu `for`. Abychom si nemuseli neustále pamatovat délku všech námi vytvořených polí, je vhodné si navíc představit vlastnost `Length`, v níž je v podobě proměnné typu `int` uložena délka pole (počet jeho prvků).

```
int[] cisla = new int[3]; //cisla.Length bude rovno třem
string[] texty = new string[10]; //texty.Length bude rovno deseti
```

Použití cyklu pro průchod polem se současným využitím vlastnosti `Length` ukazuje následující aplikace.

```
Console.WriteLine("Zadejte 5 jmen");
string[] jmena = new string[5]; // Vytvoření pole o délce 5
for (int i = 0; i < jmena.Length; i++) // Cyklus v němž se i mění od 0 do délky pole-1
{
```

```
    Console.WriteLine("Zadej jméno"); // Výpis požadavku na i-té jméno
    jmena[i] = Console.ReadLine(); // Uložení vstupu do i-tého prvku pole
}
Console.WriteLine("Zadal/a jste následující jména:");
for (int i = 0; i < jmena.Length; i++) // Cyklus v němž se i mění od 0 do délky pole-1
{
    Console.WriteLine(jmena[i]); // Výpis i-tého jména
}
```

S poli chceme často provádět další operace, jednou z nejpodstatnějších je obracení pořadí prvků. Abychom ji nemuseli implementovat, učinili tak autoři jazyku C# za nás. Je realizována pomocí metody `Reverse` třídy `Array`, která nabízí i mnoho dalších možností pro práci s poli. Použití této metody může vypadat následovně.

```
int[] cisla = new int[3];
cisla[0] = 1;
cisla[1] = 50;
cisla[2] = 9;
Array.Reverse(cisla); // Obrácení pořadí prvků
for (int i = 0; i < cisla.Length; i++)
{
    Console.WriteLine(cisla[i]); // Výpis prvků by proběhl v pořadí: 9, 50, 1
}
```

Úloha – Vytvořte aplikaci, která požádá uživatele o zadání délky pole celých čísel, následně on něj uvedený počet čísel přijme, uloží do pole a vypíše v opačném pořadí.

Provedení výpisu celého obsahu pole je vyjma výše uvedené cesty možno realizovat jednoduchým způsobem také pomocí cyklu `foreach`. Víceprvkovým datovým typem, který cyklu předáváme, je v takovém případě pole samo. Níže uvedený příklad demonstruje použití cyklu `foreach`.

```
string[] texty = new string[3];
texty[0] = "Hello";
texty[1] = "world";
texty[2] = "!";
foreach (string s in texty)
{
    Console.WriteLine(s);
    //Cyklus by vypsal na jednotlivé řádky text "Hello", "world" a "!"
}
```

Kromě klasických, jednorozměrných polí, můžeme v C# vytvářet také pole vícerozměrná. Pro naše potřeby bude zcela postačující znalost polí dvojrozměrných, princip funkce je nicméně stejný i pro vyšší počet rozměrů. Dvojrozměrné pole je možné vytvořit obdobným způsobem, jako pole jednorozměrné, jak ukazuje následující obecný zápis.

```
datový_typ[,] jméno_pole = new typ[první_rozměr, druhý_rozměr];
```

Jediným rozdílem je tedy uvedení čárky v první hranaté závorce a dvou čísel udávajících rozměry pole místo jednoho v závorce druhé. Pole 2x3 proměnné typu `double` bychom tak mohli vytvořit následovně.

```
double[,] desetinnaCisla = new double[2, 3];
```

Označení prvků v tomto poli ukazuje následující tabulka.

desetinnaCisla[0,0]	desetinnaCisla[0,1]	desetinnaCisla[0,2]
desetinnaCisla[1,0]	desetinnaCisla[1,1]	desetinnaCisla[1,2]

Pro specifikaci konkrétního prvku v poli tedy v tomto případě potřebujeme dva indexy.

Podobně, jako jsme využili v cyklu for u jednorozměrného pole vlastnost `Length` můžeme u dvojrozměrného pole použít metodu `GetLength`, které předáme index rozměru, který nás zajímá (nulu pro první rozměr, jedničku pro druhý rozměr, atp.). V případě výše vytvořeného pole by byly výsledky dané metody následující.

```
desetinnaCisla.GetLength(0); // Velikost pole v prvním rozměru - výsledkem je číslo 2
desetinnaCisla.GetLength(1); // Velikost pole ve druhém rozměru - výsledkem je číslo 3
```

Zápis i čtení hodnoty proměnných ve dvojrozměrném poli je možné realizovat stejným způsobem, jako v poli jednorozměrném, jediný rozdíl nastává ve chvíli, kdy chceme zápis či čtení provést s použitím `for` cyklu. V takovém případě je totiž třeba vytvořit cykly dva – jeden uvnitř druhého – z nichž první udává řádek (hodnotu prvního rozměru) a druhý sloupec (hodnotu druhého rozměru) určující konkrétní buňku. Vytvoření i výpis dvojrozměrného pole ukazuje následující příklad.

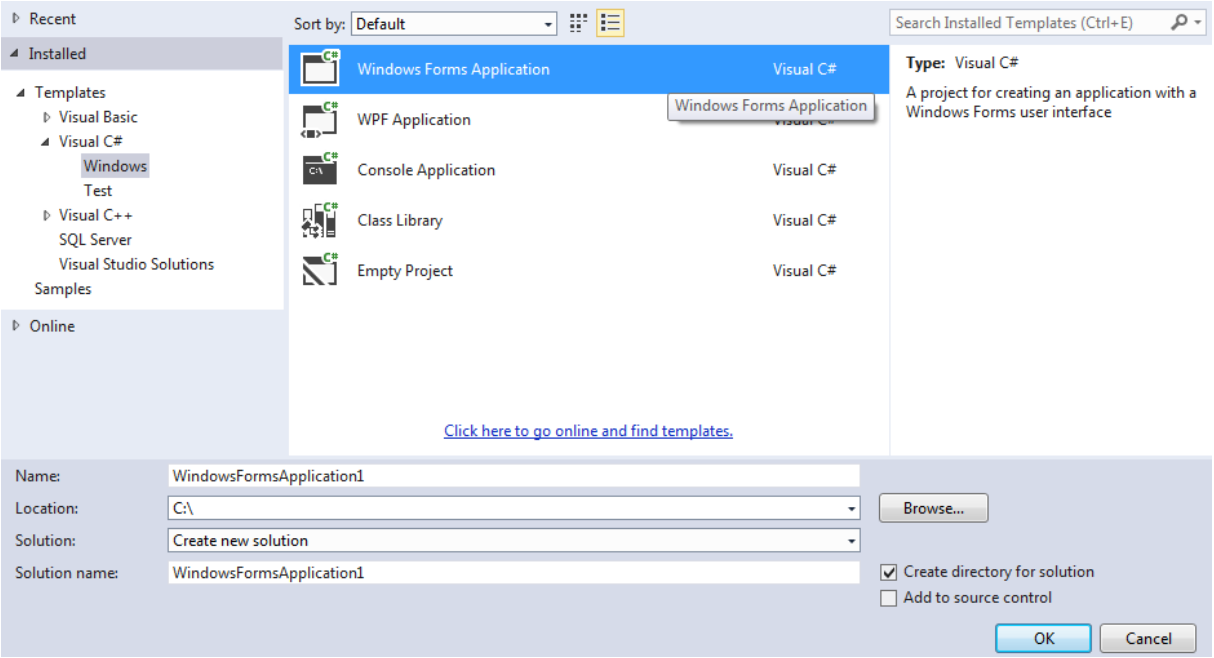
```
string[,] texty= new string[2,3];
texty[0, 0] = "První řádek, první sloupec";
texty[0, 1] = "První řádek, druhý sloupec";
texty[0, 2] = "První řádek, třetí sloupec";
texty[1, 0] = "Druhý řádek, první sloupec";
texty[1, 1] = "Druhý řádek, druhý sloupec";
texty[1, 2] = "Druhý řádek, třetí sloupec";

for (int i = 0; i < texty.GetLength(0); i++) // Cyklus procházející řádky
{
    for (int j = 0; j < texty.GetLength(1); j++) // Cyklus procházející sloupce
    {
        Console.WriteLine(texty[i, j]);
        // Výpisem by byl obsah prvního řádku následovaný obsahem druhého řádku
    }
}
```

Shrnutí kapitoly – Pro uložení většího počtu proměnných stejného typu můžeme použít tzv. pole, což nám umožní adresovat jednotlivé proměnné pouze pomocí identifikátoru pole a pořadového čísla dané proměnné v něm – tzv. indexu. Pole mohou být vícerozměrná, v takovém případě jednotlivé prvky v nich adresujeme pomocí většího počtu indexů (počet indexů je vždy stejný, jako počet rozměrů pole). K obsahu polí můžeme jednoduše přistupovat pomocí cyklů.

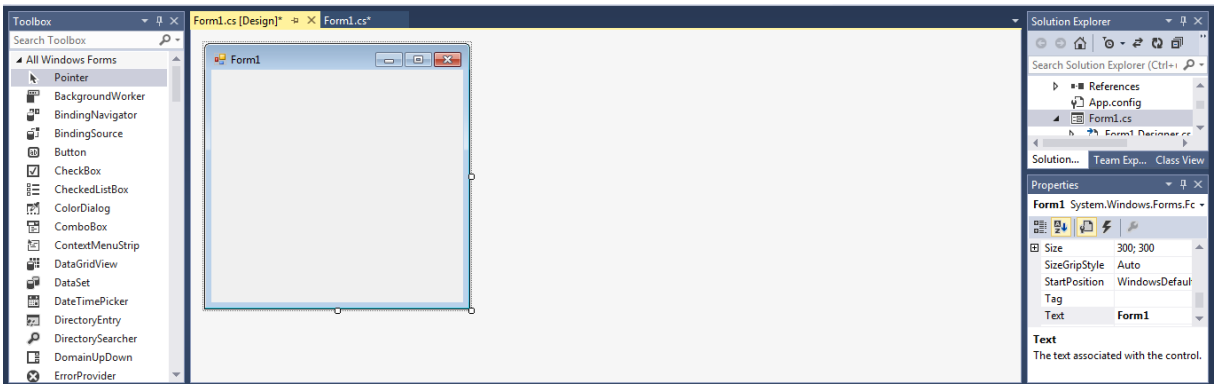
Závěrečná úloha – Vytvořte aplikaci, která pomocí dvourozměrného pole o velikosti 26x2, obsahujícího v jednom „sloupci“ klasickou abecedu a ve druhém Morseovu abecedu umožní převést uživatelem vložený text do Morseovy abecedy.

7 Seznámení s prostředím Windows Forms

Jak víme, konzolové aplikace nejsou jediným typem, který můžeme v jazyce C# ve Visual Studiu vytvářet. V této kapitole se naučíme vytvářet aplikace typu Windows Forms, tedy programy s pokročilými uživatelskými rozhraními založeným na formulářových oknech užívaných v operačních systémech Windows. Aplikaci tohoto typu vytvoříme obdobně, jako aplikaci konzolovou, při výběru nového projektu v prostředí Visual Studio pouze zvolíme možnost „Windows Forms Application“.



Po vytvoření nového projektu se nám otevře vizuální prostředí pro návrh formuláře, který bude sloužit jako základní prvek celé aplikace a bude pro nás v tomto ohledu tedy nahrazovat konzoli.



Z nabídky Toolbox po levé straně můžeme pomocí myši volit ovládací prvky (ovládací prvky jsou svou povahou objekty, jak bude zmíněno v poslední kapitole) a umísťovat je na formulář. Po kliknutí na formulář nebo na ovládací prvek se v pravém spodním rohu prostředí v nabídce Properties zobrazí jméno daného prvku a jeho vlastnosti, které zde můžeme jako proměnné nastavovat.

Nejprve se podívejme na nabídku Properties a nastavení některých vlastností pro zobrazený formulář. Po kliknutí na něj se zobrazí jeho označení (Form1) a soupis vlastností, které můžeme nastavit. První vlastností, kterou si představíme, je `Text`. Tato vlastnost ukládá v datovém typu `string` text zobrazený v nadpisu formuláře. Pokud tedy změníme obsaženou hodnotu, změna se

promítne také do nadpisu formuláře. Obdobným způsobem bychom mohli měnit jeho ostatní vlastnosti.

Nyní si představíme první ovládací prvek – `Button` – jde o klasické tlačítko, s jakým se setkáváme ve většině formulářových aplikací.

Po umístění na formulář můžeme měnit nastavení jeho vlastnosti stejně jako v případě samotného formuláře. Kliknutí na nově vytvořené tlačítko zobrazí v nabídce Properties související vlastnosti, nejprve je však vhodné povšimnout si jeho označení – button1. Visual Studio za nás na formulář přidané prvky automaticky pojmenovává a to vždy pomocí jejich typu a pořadového čísla. Vlastnost `Text` v případě tlačítka nastavuje popisek, který je na něm zobrazen. `Text` není jedinou zajímavou vlastností tlačítek, než se však podíváme na další, představíme si koncept událostí.

Událostí je v prostředí Windows Forms myšlena akce vykonaná uživatelem nebo programem, na níž je možno v kódu reagovat. Vzhledem k tomu, že na rozdíl od konzole není v prostředí formulářů aplikace nucena čekat na uživatelské vstupy, je celý její běh v podstatě událostmi řízen. Praktickým příkladem události může být ukončení načítání formuláře po startu aplikace, kliknutí myší na tlačítko, nebo změna obsahu textového pole (s nímž se seznámíme níže). Každá událost na sebe může mít navázaný kód, který se má vykonat v případě, že nastane. K seznamu událostí pro konkrétní ovládací prvek se dostaneme pomocí jeho označení a následného kliknutí na ikonu Events (ikona s bleskem) v nabídce Properties. Do editoru kódu je možné dostat se několika způsoby, nejjednodušším je zřejmě kliknout na formulář pravým tlačítkem myši a zvolit možnost „View Code“, pro nás je však v tuto chvíli zajímavější možnost jiná. Tou je otevření kódu pomocí automatického vygenerování metody, do níž můžeme doplnit kód pro určitou událost. To učiníme pomocí dvojitého kliknutí na událost, na níž chceme reagovat, v nabídce Properties.

V případě tlačítka je pro nás pravděpodobně nejzajímavější událost `Click`, která nastane po kliknutí myší na tlačítko. Po jejím zvolení a potvrzení můžeme do vygenerovaného kódu dopsat vlastní příkazy.

```
private void button1_Click(object sender, EventArgs e)
{
    // Do tohoto místa je možné doplnit vlastní kód
}
```

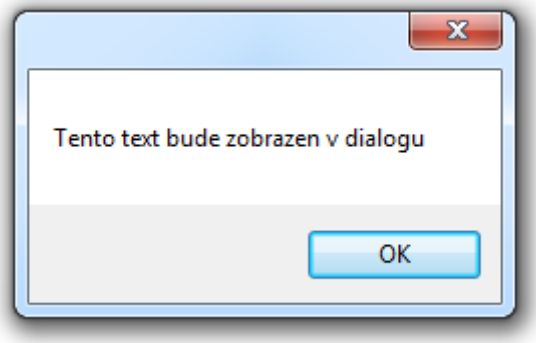
Pro nastavení vlastnosti ovládacího prvku v kódu je třeba zapsat jeho označení následované tečkou a jménem vlastnosti. Výjimkou z výše uvedeného je do určité míry formulář – pro nastavení jeho vlastností se nepoužívá označení Form1, ale klíčového slova `this` (funkce tohoto klíčového slova je popsána v kapitole 12). Popisek tlačítka bychom mohli změnit například pomocí kódu

```
button1.Text = "abc";
```

Pokud uvedený kód vložíme do vytvořené metody, která je navázaná na událost `Click`, bude tlačítko po stisknutí měnit svůj popisek na text „abc“.

Popisek tlačítka bezpochyby není ideálním prostředkem pro výpis informací uživateli, představíme si zde tedy třídu `MessageBox`, která nám umožní zobrazovat relativně elegantní formou jednoduché výstupy uživateli. Pomocí její metody `Show`, které předáme (podobně, jako jsme to dělali u metody `WriteLine` třídy `Console`) proměnnou typu `string`, můžeme zobrazit jednoduché dialogové okno, v němž bude předaný text zobrazen.

```
MessageBox.Show("Tento text bude zobrazen v dialogu");
```



Pokud bychom tedy chtěli v aplikaci užít tlačítko „O aplikaci“, po kliknutí na nějž by došlo k zobrazení informací o autorovi, mohli bychom použít například následující kód.

```
private void button1_Click(object sender, EventArgs e)
{
    MessageBox.Show("Ukázková aplikace ©2015 SSPŠ");
}
```

`MessageBox` není jedinou cestou, kterou jsme v prostředí formulářových aplikací schopni uživateli předávat textový výstup. Další možností je použití ovládacího prvku `Label`, což je uživatelem needitovatelné textové pole.

Jeho obsah můžeme nastavit – jak jinak – pomocí vlastnosti `Text`.

```
label1.Text = "Tento text bude zobrazen v labelu";
```

Provedení jednoduchého výstupu ve formulářové aplikaci tedy není nijak složité, ať použijeme metodu `Show` třídy `MessageBox`, nebo vlastnost `Text` prvku `Label`. Obdobně jednoduchou záležitostí je také získávání vstupů od uživatele. Prvků, které získání vstupu uživatele umožňují, je více, my se prozatím seznámíme pouze se zřejmě nejpodstatnějším z nich. Je jím `TextBox` – uživatelem editovatelné textové pole.

Obsah, který do něj uživatel zadá, můžeme získat z vlastnosti `Text`. Pokud bychom tedy chtěli například nastavit nadpis formuláře na text zadaný do prvku `TextBox`, mohli bychom použít následující kód.

```
this.Text = textBox1.Text;
```

Ovládací prvky typu `TextBox` je samozřejmě možné využívat jak pro vstup, tak pro výstup a to i víceřádkový. K tomu, abychom nastavili možnost zapisovat do pole `TextBox` více řádků, je třeba změnit nastavení jeho boolovské vlastnosti `Multiline` na `true`. Odřádkování při výstupu je možné provést vložením hodnoty `Environment.NewLine`, jak ukazuje následující příklad.

```
textBox1.Text = "První řádek" + Environment.NewLine + "Druhý řádek";
```

Úloha – Vytvořte aplikaci – kalkulačku – schopnou provádění všech základních numerických operací s desetinnými čísly. Použijte tři pole `TextBox` (2 pro čísla, jeden pro znaménko), jedno tlačítko `Button` (pro spuštění výpočtu) a jedno pole `Label` (pro výpis výsledku).

Podobně, jako jsme reagovali na událost stisknutí tlačítka, je možné u prvků `TextBox` reagovat na změnu obsahu. Událost, která při změně textového pole nastává, se označuje `TextChanged`. Pokud necháme do kódu vygenerovat s ní spojenou metodu, můžeme například zajistit, aby se obsah prvku `Label` umístěného na formuláři dynamicky měnil a uchovával vždy stejný obsah jako prvek `TextBox`.

```
private void textBox1_TextChanged(object sender, EventArgs e)
{
    label1.Text = textBox1.Text;
}
```

Nyní již dokážeme provádět v prostředí formulářů v podstatě stejné vstupně-výstupní operace jako v konzoli. Prostředí formulářů nám však nabízí také další možnosti, zejména ve vizuální oblasti. Uvedeme zde alespoň dvě z nich.

První možností, kterou nám formuláře oproti konzoli nabízí, je možnost snadné práce s barvami různých prvků. Barvu pozadí většiny vizuálních ovládacích prvků (uvedené platí pro všechny prvky zde zmíněné a také pro formulář) je možné nastavit pomocí vlastnosti `BackColor`. Tato vlastnost ukládá hodnotu strukturního typu `Color`. Strukturu `Color` můžeme pro nejjednodušší nastavení barvy pozadí prvků užít jako výčtový datový typ (hodnotu tedy vybereme z nabízených možností po napsání tečky za `Color`), jak demonstruje následující kód.

```
private void button1_Click(object sender, EventArgs e)
{
    this.BackColor = Color.Blue; // Nastavení barvy formuláře na modrou
}
```

Podobně jako barvu ovládacích prvků můžeme nastavovat také jejich pozici na formuláři. Ta je určována vlastností `Location` vizuálních ovládacích prvků. `Location` ukládá hodnotu datového typu `Point`, což je struktura s dvěma členy typu `int` pojmenovanými `X` a `Y` (pojmenování členů odpovídá osám, v nichž nastavují pozici prvku). `Point` je možné vytvořit mimo jiné následujícím způsobem:

```
Point p = new Point(Xová_souřadnice, Yová_souřadnice);
```

Nastavení pozice tlačítka `button1` na bod (0;0) by tedy mohlo být provedeno pomocí

```
button1.Location = new Point(0, 0);
```

a nastavení jeho pozice na bod (100;100) pomocí kódu

```
button1.Location = new Point(100, 100);
```

Nastavení pozice je možné využít mimo jiné k „posouvání“ ovládacích prvků po formuláři, jak ukazuje následující kód, pomocí něhož se tlačítko po kliknutí posune o 5 pixelů doprava.

```
private void button1_Click(object sender, EventArgs e)
{
    button1.Location = new Point(button1.Location.X + 5, button1.Location.Y);
}
```

Úloha – Vytvořte aplikaci, v níž bude pomocí obsahu dvou textových polí možno nastavit pozici tlačítka v osách X a Y.

Shrnutí kapitoly – Vedle konzolových aplikací můžeme v C# vytvářet také aplikace s uživatelským rozhraním založeným na formulářích užívaných nejčastěji v operačním systému Windows. Na rozdíl od konzolových aplikací je běh programu řízen tzv. událostmi, tedy akcemi uživatele či programu, na něž můžeme v kódu reagovat. Pro získání uživatelského vstupu můžeme použít textové pole typu `TextBox`, případně tlačítko typu `Button`. Pro výstup pak můžeme kromě uvedeného textového pole použít prvky typu `MessageBox` a `Label`. Ovládací prvky mají tzv. vlastnosti – de-facto proměnné, které ovlivňují jejich vzhled a chování, pro nás jsou podstatné vlastnosti `Text`, `Color` a `Location`, nastavující popisek prvku, jeho barvu a pozici.

Závěrečná úloha – Vytvořte jednoduchou hru piškvorky, s devíti tlačítky užitými jako pole, která v případě úspěšného dohrání vypíše, který z hráčů zvítězil.

8 Dialogy

Získávání vstupů od uživatele je, jak víme, pro funkci většiny aplikací zcela stěžejní. Probrané ovládací prvky nám umožňují přijímat většinu uživatelských vstupů, jako text, či například čísla poměrně elegantním způsobem, pro získání některých dalších (například barva či cesta k souboru) však jednoduché ovládací prvky umístitelné na formulář neexistují.

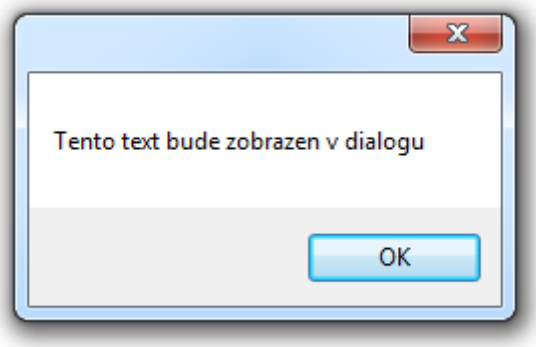
Místo nich vývojáři platformy .NET vytvořily specializované dialogy, tedy samostatná okna/formuláře, které lze pro získání některých složitějších vstupů nebo pro určitou interakci s uživatelem využít.
V této kapitole se naučíme pracovat s nimi.

8.1 MessageBox

Prvním z dialogových oken, kterým se budeme zabývat, je nám již známý `MessageBox`. Prozatím jsme jej v úlohách používali pouze pro zobrazení textu uživateli a ten mohl zobrazené okno vypnout pomocí tlačítka OK nebo stisknutím křížku. `MessageBox` ale můžeme modifikovat tak, aby kromě textu v okně zobrazoval také text v nadpisové oblasti, ikonu odpovídající povaze textu (zda jde o informaci, varování, otázku, apod.) a několik kombinací tlačítek, namísto původního OK.

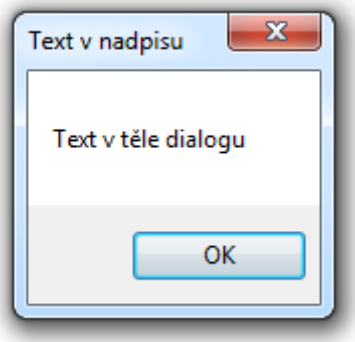
Možná jste si při používání dialogu `MessageBox` již všimnuli, že existuje celkem 11 přetížení (možností, jak ji volat) jeho metody `Show`. To, které jsme používali až doposud, přijímá jako parametr pouze jeden textový řetězec, který byl následně zobrazován v těle formuláře dialogu.

```
MessageBox.Show("Tento text bude zobrazen v dialogu");
```



Další přetížení, s nímž se seznámíme, přijímá dva textové řetězce – první z nich opět zobrazí v těle dialogu, druhý potom v jeho nadpisové oblasti.

```
MessageBox.Show("Text v těle dialogu", "Text v nadpisu");
```

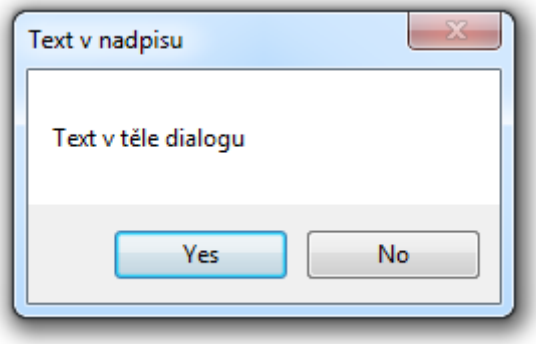


Třetí přetížení nám umožňuje zobrazit další/jiná tlačítka v těle dialogu. Volá se uvedením dvou textových řetězců, následovaných prvkem výčtového typu `MessageBoxButtons`. Kombinace tlačítek, jichž je možno dosáhnout, jsou:

- Abort, Retry a Ignore – `MessageBoxButtons.AbortRetryIgnore`
- OK – `MessageBoxButtons.OK`
- OK a Cancel – `MessageBoxButtons.OKCancel`
- Retry, Cancel – `MessageBoxButtons.RetryCancel`
- Yes a No – `MessageBoxButtons.YesNo`
- Yes, No a Cancel – `MessageBoxButtons.YesNoCancel`

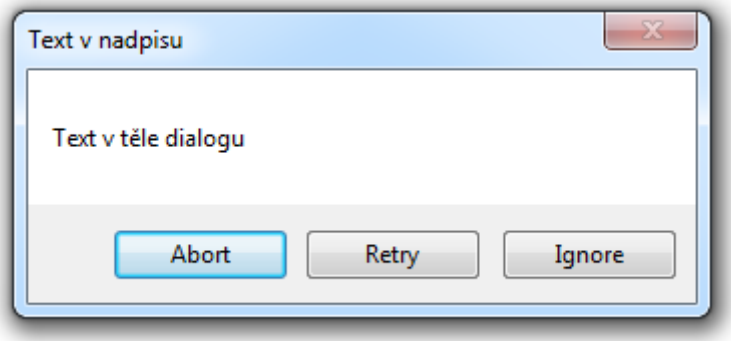
Text tlačítek se automaticky překládá – v české verzi systému Windows budou například místo tlačítek Yes a No zobrazena tlačítka s popisky Ano a Ne. Volání metody Show pomocí tohoto přetížení může vypadat například následovně

```
MessageBox.Show("Text v těle dialogu", "Text v nadpisu ", MessageBoxButtons.YesNo);
```



nebo

```
MessageBox.Show("Text v těle dialogu", "Text v nadpisu ",
MessageBoxButtons.AbortRetryIgnore);
```



Je vhodné poznamenat, že metoda `Show` nemá přetížení, které by umožňovalo nastavení zobrazených tlačítek, aniž by byl nastaven jak text těla dialogu, tak text jeho nadpisu.

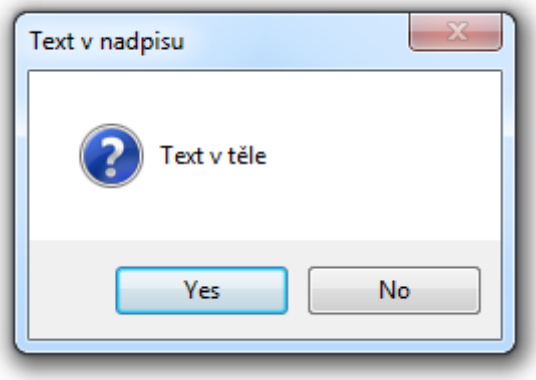
Poslední přetížení metody Show, které si představíme, je rozšířením předchozího – přijímá dva textové řetězce, prvek výčtového typu `MessageBoxButtons` a prvek výčtového typu `MessageBoxIcon`. Ten může nabývat hodnot:

- Asterisk (hvězdička) – `MessageBoxIcon.Asterisk`
- Error (chyba) – `MessageBoxIcon.Error`
- Exclamation (výkřik) – `MessageBoxIcon.Exclamation`
- Hand (kritické zastavení) – `MessageBoxIcon.Hand`
- Information (informace) – `MessageBoxIcon.Information`
- None (žádná) – `MessageBoxIcon.None`
- Question (otázka) – `MessageBoxIcon.Question`
- Stop (zastavení) – `MessageBoxIcon.Stop`
- Warning (varování) – `MessageBoxIcon.Warning`

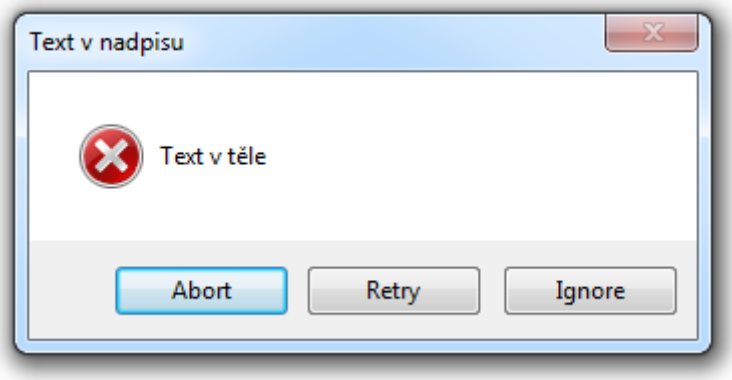
Datový typ `MessageBoxIcon` nám umožňuje nastavit ikonu, která se má v těle dialogu zobrazit, současně nastavuje také zvuk, který bude přehrán při jeho zobrazení.

Metodu `Show` můžeme pomocí uvedeného přetížení volat například tak, jak ukazují následující dva příklady:

```
MessageBox.Show("Text v těle", "Text v nadpisu", MessageBoxButtons.YesNo,
MessageBoxIcon.Question);
```




```
MessageBox.Show("Text v těle", "Text v nadpisu", MessageBoxButtons.AbortRetryIgnore,
MessageBoxIcon.Error);
```



Stejně jako všechny ostatní předdefinované dialogy vrací `MessageBox` hodnotu typu `DialogResult`, což je výčtový typ, který může nabývat hodnot Abort, Cancel, Ignore, No, None, OK, Retry a Yes. Jak

lze z uvedeného výčtu možných hodnot usuzovat, dialogy vrací pomocí hodnoty typu `DialogResult` informace o tlačítku, pomocí nějž byly ukončeny.

`DialogResult` je, stejně jako ostatní datové typy, možno porovnávat s jinými hodnotami téhož typu a tím pádem i přímo s metodami, které dialogy zobrazují a mají `DialogResult` jako návratovou hodnotu.

V případě prvku `MessageBox` s tlačítky Ano a Ne (`MessageBoxButtons.YesNo`) tedy můžeme pomocí jednoduchého větvení za použití struktury `if` reagovat odpovídajícím způsobem na rozhodnutí uživatele, jak ukazuje následující kód:

```
DialogResult dr = MessageBox.Show("Text v těle", "Text v nadpisu",
MessageBoxButtons.YesNo);
if (dr == DialogResult.Yes)
{
    // Operace, které se mají provést, pokud uživatel stiskne tlačítko Ano
}
else
{
    // Operace, které se mají provést, pokud uživatel stiskne tlačítko Ne
}
```

Následující kód pak ukazuje složitější větvení při zobrazení tlačítek Zrušit, Opakovat a Ignorovat (`MessageBoxButtons.AbortRetryIgnore`):

```
DialogResult dr = MessageBox.Show("Text v těle", "Text v nadpisu",
MessageBoxButtons.AbortRetryIgnore);
if (dr == DialogResult.Abort)
{
    // Operace, které se mají provést, pokud uživatel stiskne tlačítko Zrušit
}
else if (dr == DialogResult.Retry)
{
    // Operace, které se mají provést, pokud uživatel stiskne tlačítko Opakovat
}
else
{
    // Operace, které se mají provést, pokud uživatel stiskne tlačítko Ignorovat
}
```

Úloha – Vytvořte aplikaci, která postupně zobrazí uživateli 8 dialogů `MessageBox` s tlačítky Ano a Ne, pomocí nichž uživatel nastaví jednotlivé bity 8bitového čísla na 1 či 0. Výsledné číslo uživateli vypište v desítkové soustavě.

8.2 OpenFileDialog a SaveFileDialog

Jak vyplývá z uvedených jmen, dva dialogy, které si představíme nyní, slouží k výběru souboru pro uložení, nebo otevření. Oba dialogy (a všechny další, které budeme dále probírat) je pro použití napřed nutné přidat mezi ovládací prvky formuláře – je možné je najít v paletě Toolbox a „umístit“ je na formulář jako kterýkoli jiný ovládací prvek. Dialogy se nicméně po přidání neobjeví na formuláři, ale v liště pod ním (podobně se chovají také některé jiné ovládací prvky).

Jejich použití je obdobné jako v případě prvku `MessageBox`, rozdíl je pouze ve jménu metody, kterým způsobíme otevření dialogu (ta se u všech dialogů s výjimkou dialogu `MessageBox` nazývá `ShowDialog`) a skutečnosti, že této metodě nebudeme předávat žádný parametr. Pro zobrazení dialogu typu `OpenFileDialog` pojmenovaného openFileDialog1 tedy můžeme volat:

```
openFileDialog1.ShowDialog();
```

Zobrazení dialogu typu `SaveFileDialog` pak probíhá obdobně.

Stejně jako v případě dialogu `MessageBox`, i `SaveFileDialog` a `OpenFileDialog` (a také všechny ostatní, které budeme probírat) vrací, jak již bylo zmíněno, jako návratovou hodnotu datový typ `DialogResult`, konkrétně jde o hodnotu `DialogResult.OK`, pokud uživatel dialog uzavře kliknutím na tlačítko "Otevřít", resp. "Uložit", nebo `DialogResult.Cancel`, pokud jej ukončí tlačítkem "Storno".

Nyní je na místě otázka, jakým způsobem zjistíme, který soubor uživatel v dialogu zvolil. Odpověď zní, že cesta k souboru se po potvrzení dialogů pomocí tlačítka "Otevřít", případně "Uložit", uloží do vlastnosti `FileName` daného dialogu (pozor – při stornování dialogu nikoli) jako datový typ `string`.

V praxi má tedy oba dialogy smysl užívat v následující konstrukci:

```
DialogResult dr = openFileDialog1.ShowDialog();
if (dr == DialogResult.OK)
{
    // Zde můžeme pokračovat v práci se zvoleným souborem
}
```

Možnosti dalšího nastavení obou diskutovaných dialogů jsou z našeho pohledu o něco omezenější, než v případě dialogu `MessageBox`.

My si tedy představíme již jen dvě jejich další vlastnosti. První z nich nese označení `Title` a jedna se o vlastnost ukládající datový typ `string`, jehož obsah se objeví v nadpisu dialogu. Použití může vypadat například následovně:

```
openFileDialog1.Title = "Zvolte soubor pro otevření";
```

Poslední vlastnost dialogů `OpenFileDialog` a `SaveFileDialog`, kterou si představíme, je také typu `string` – jedná se o `Filter`. Pomocí této vlastnosti můžeme uživateli umožnit vybírat pro otevření či uložení jen soubory určitého typu (resp. s určitou příponou). To provedeme tak, že do uvedené vlastnosti vložíme text ve formátu

```
popisek prvního typu souborů|maska prvního typu souborů|popisek druhého typu
souborů|maska druhého typu souborů|...
```

Nastavení umožňující načtení souboru s příponou txt by tedy vypadalo následovně:

```
openFileDialog1.Filter = "Textové soubory|*.txt";
```

a nastavení, které uživateli umožňuje ukládat soubory typu csv, nebo libovolného jiného typu bychom mohli realizovat takto:

```
openFileDialog1.Filter = "CSV soubory|*.csv|Všechny soubory|*.*";
```

8.3 FolderBrowserDialog

Dalším dialogem, s nímž se seznámíme, je `FolderBrowserDialog`, který, jak již název napovídá, slouží k procházení a volbě složek. Jeho použití je podobné, jako v případě dialogů `OpenFileDialog` a `SaveFileDialog` – o zobrazení se opět stará metoda `ShowDialog` a návratovou hodnotou je `DialogResult`. Cestu ke zvolené složce tentokrát najdeme ve vlastnosti `SelectedPath`. Použití dialogu `FolderBrowserDialog` může vypadat například tak, jak ukazuje následující příklad.

```
DialogResult dr = folderBrowserDialog1.ShowDialog();
if (dr == DialogResult.OK)
{
    //Zde můžeme vložit kód pro práci s vybranou složkou
}
```

8.4 ColorDialog

Dialogy můžeme pochopitelně užívat i pro jiné účely, než je volba složek či souborů. Další z dialogů, který nám C# nabízí, například umožňuje zvolit hodnotu datového typu `Color` a tedy získat od uživatele jednoduchým způsobem jím vybranou. Jde o dialog typu `ColorDialog`. Pomocí metody `ShowDialog` tentokrát zobrazíme dialogové pole pro výběr barvy, které, v případě potvrzení výběru tlačítkem OK, nastavuje odpovídajícím způsobem vlastnost dialogu jménem `Color`. Jeho použití je poměrně přímočaré, jak demonstruje následující ukázka, v níž nastavujeme barvu pozadí formuláře.

```
colorDialog1.ShowDialog();
this.BackColor = colorDialog1.Color;
```

Je vhodné podotknout, že v jazyce C# existují ještě další předdefinované dialogy, pro naše potřeby bude však zcela postačující schopnost použití dialogů výše uvedených.

Úloha – Vytvořte aplikaci se třemi tlačítky, z nichž každé po stisknutí způsobí zobrazení dialogu `ColorDialog`, pomocí nějž mu uživatel bude moci nastavit barvu.

Shrnutí kapitoly – Pro získání určitých specifických vstupů od uživatele můžeme použít speciální formuláře označované dialogy. Nejčastěji užívaným z nich je `MessageBox`, pomocí nějž můžeme uživateli zobrazovat jednoduché zprávy a následně zjistit, kterým ze zobrazených tlačítek na ni reagoval. Uvedený výsledek bude dialogem vrácen jako hodnota typu `DialogResult`. Z dalších dialogů jsou nejpodstatnější `OpenFileDialog` a `SaveFileDialog`, které nám umožňují získat cestu k souboru pro otevření či uložení, `FolderBrowserDialog`, sloužící k získání cesty k adresáři a `ColorDialog`, umožňující uživateli nastavovat barvy.

Závěrečná úloha – Vytvořte aplikaci se třemi tlačítky, která po stisknutí zobrazí `MessageBox` s dotazem, zda si uživatel přeje nastavit barvu pozadí všech prvků. Pokud uživatel zvolí „Ano“, zobrazte postupně 4 dialogy `ColorDialog` pro nastavení barev formuláře a tlačítek.

9 Výjimky

Při tvorbě programů v jazyce C# se můžeme velmi často setkat se situací, při níž aplikace přestane pracovat a „spadne“. Vzhledem k tomu, že i v dokončených a odladěných aplikacích může k neočekávaným chybovým stavům dojít, je na místě snažit ošetřit program takovým způsobem, aby nezpůsobovaly pád aplikace, pokud nastanou. V této kapitole se naučíme jak je možné ošetření kódu provést.

Hlavním nástrojem, který nám k tak zvanému ošetřování výjimek slouží, je `try-catch` blok. Jedná se o část kódu s následující strukturou:

```
try
{
    // Zde uvedeme kód, který chceme ošetřit
}
catch
{
    // Zde uvedeme kód, který má nastat v případě, že dojde k výjimce
}
```

Při vzniku výjimky (chybového stavu) v kódu obsazeném v `try` bloku je proveden okamžitý skok do bloku `catch`, jehož obsah tudíž můžeme využít ke vhodnému zpracování chyby (například vypsání informace uživateli – viz příklad) či zotavení z ní.

```
try
{
    int i = Convert.ToInt32(textBox1.Text);
    textBox1.Text = (i + 1).ToString();
}
catch
{
    MessageBox.Show("V aplikaci došlo k chybě.");
}
```

Úloha – Vytvořte aplikaci, která přijme do textových polí 3 desetinná čísla a vrátí jejich součet. Aplikaci ošetřete pomocí bloku `try-catch` a v případě výjimky vypište chybové hlášení.

Při programování složitějších algoritmů často dochází k situaci, kdy musíme zajistit, aby se určité kroky vykonaly v případě, kdy nedojde k výjimce v předchozím kódu, ale potřebujeme zajistit jejich proběhnutí i v případech, kdy k nějaké výjimce dojde. Uvedený problém řeší blok `finally`, kterým můžeme doplnit bloky `try` a `catch` (zapisuje se za ně), případně nahradit blok `catch` a získat tak konstrukci `try-finally`.

```
try
{ ... }
catch
{ ... }
finally
{
    // Zde uvedeme kód, který má proběhnout vždy
}
```

Příkazy, které zapíšeme do bloku `finally` se vykonají bez ohledu na to, zda v předchozím bloku `try` došlo k výjimce. Uvedeným způsobem se velmi často ošetřuje například práce se

souborem (viz níže), u níž potřebujeme zajistit, aby i v případě, kdy dojde k chybě při přístupu k souboru, byl uzavřen odpovídající `StreamReader` či `StreamWriter`.

Ve chvíli, kdy v aplikaci dojde k výjimce, je automaticky vytvořen objekt (viz níže) třídy `Exception`, který v sobě nese informace o chybě, k níž došlo. Tuto informaci jsme schopni v bloku `catch` zpracovávat, stačí jej doplnit do následujícího tvaru.

```
catch (Exception jménoVýjimky)
{

}
```

Uvnitř bloku catch je v takovém případě možné pomocí jména výjimky zjistit informace o chybě, respektive zprávu o ní. Ta je uložena ve vlastnosti `Message` (která je typu `string`) naší pojmenované výjimky a můžeme ji tedy například zobrazit uživateli. V následujícím příkladu by tak byl pomocí prvku `MessageBox` zobrazen text informující uživatele, že vstupní řetězec nebyl ve správném formátu (vzhledem k tomu, že text „x“ nedokáže C# pomocí metody `ToInt32` třídy `Convert` převést na číslo).

```
try
{
    int i = Convert.ToInt32("x");
}
catch (Exception vyjimka)
{
    MessageBox.Show(vyjimka.Message);
}
```

Úloha – Vytvořte aplikaci, která do dvou textových polí přijme 2 celá čísla a vrátí jejich podíl. Aplikaci ošetřete pomocí `try-catch` bloku a v případě výjimky vypište chybové hlášení, které se nachází ve vlastnosti `Message` dané výjimky.

Shrnutí kapitoly – Ve vytvářených programech může často docházet k nepředvídanému chování, jehož důsledkem může být i pád celé aplikace. Abychom těmto pádům předešli, je možné ošetřit kód pomocí bloku `try-catch`, případně bloku `finally`. V takovém případě výjimka vyvolaná v bloku `try` způsobí pouze skok do kódu uvedeného v bloku `catch`, němž můžeme provést zotavení z chyby, nebo vypsat uživateli informace o ní. V bloku `catch` můžeme z výjimky, což je prvek typu `Exception`, získat zprávu vysvětlující důvod, proč k výjimce došlo. Text s odůvodněním je uložen ve vlastnosti `Message`.

Závěrečná úloha – Vytvořte aplikaci/kalkulačku, v níž ošetříte výjimky.

10 Práce s textovými soubory

Jazyk C# nám poskytuje několik tříd, které umožňují poměrně elegantním a jednoduchým způsobem pracovat s obsahem textových souborů a tedy ukládat do nich data a zpětně je načítat. V této kapitole si představíme způsob, jakým je v C# možné vstupně výstupní operace se soubory provádět. V podstatě všechny třídy k tomuto účelu využitelné se nacházejí ve jmenném prostoru `System.IO`, chceme-li tedy číst z textového souboru, nebo do něj zapisovat, nesmíme jej zapomenout připojit. Možností, které nám třídy z prostoru `System.IO` nabízejí je poměrně velké množství, pro naše potřeby je však zcela postačující seznámení se třemi třídami, umožňujícími provádění základních vstupně výstupních operací.

První z nich je třída `FileStream`, která poskytuje obecný přístup k souborům, budeme ji tedy využívat jak při čtení, tak při zápisu. Její konstruktor nabízí několik přetížení, my si představíme a budeme používat pouze jedno z nich, přijímající jako parametr `string`, obsahující cestu k otevíranému souboru (pozor – lomítka v cestě k souboru musí být dvojitá, nebo musí být `string` s cestou uvozen zavináčem), a instanci výčtového typu `FileMode`, která udává způsob, jímž má být daný soubor otevřen. Je vhodné zmínit, že zřejmě nejjednodušší cestou pro získání cesty k souboru od uživatele je užití dialogů `OpenFileDialog` nebo `SaveFileDialog`. Typ `FileMode` může nabývat celkem 5 hodnot:

- `Append`
- `Create`
- `CreateNew`
- `Open`
- `OpenOrCreate`
- `Truncate`

z nichž pro nás jsou podstatné pouze hodnota `Open`, kterou budeme používat pro otevření souboru pro čtení, a hodnota `Create`, jíž budeme užívat k vytvoření nového souboru a jeho otevření pro zápis.

Vytvoření instancí třídy `FileStream` pro čtení a zápis může vypadat například takto:

```
FileStream cteciStream = new FileStream(cesta_k_souboru, FileMode.Open);
FileStream zapisovaciStream = new FileStream(cesta_k_souboru, FileMode.Create);
```

Samotná třída `FileStream` umožňuje se soubory pracovat a provádí jejich otevření, pro čtení či zápis je nad ní ještě třeba vytvořit další specializovanou třídu. Tou je `StreamReader` (pro čtení) či `StreamWriter` (pro zápis). Jejich vytvoření i použití je velmi podobné

- obě třídy vytvoříme s použitím klíčového slova `new` a jako parametr jim předáme v předchozím kroku vytvořenou instanci třídy `FileStream`, tedy například
  ```
  FileStream cteciStream = new FileStream(cesta_k_souboru, FileMode.Open);
  StreamReader ctecka = new StreamReader(cteciStream);
  ```
- třída `StreamReader` poskytuje metodu `ReadLine`, která funguje obdobně, jako stejnojmenná metoda třídy `Console`, tedy načte (ze souboru) řádek a vrátí jej jako `string`
- třída `StreamWriter` poskytuje metodu `WriteLine`, která opět funguje velmi podobně, jako její protějšek že třídy `Console`, tedy zapíše (do souboru) textový řetězec typu `string`, který jí předáme, zakončený odřádkováním
- instance obou tříd je třeba po ukončení čtení/zapisování třeba uzavřít pomocí metody `Close`

Zápis textu do souboru bychom s použitím výše uvedených tříd mohli provést následujícím způsobem:

```
FileStream fs = new FileStream(@"C:\a.txt", FileMode.Create);
StreamWriter sw = new StreamWriter(fs);
sw.WriteLine("Toto je první řádek");
sw.WriteLine("Toto je druhý řádek");
sw.WriteLine("Toto je poslední řádek");
sw.Close();
```

Vygenerovaný soubor má pak následující obsah:

Toto je první řádek

Toto je druhý řádek

Toto je poslední řádek

Úloha – Vytvořte aplikaci, která umožní uživateli ukládat do pevně nastaveného souboru obsah víceřádkového textového pole.

Načtení řádku z obsahu souboru bychom pak učinili následovně:

```
FileStream fs = new FileStream(@"C:\a.txt", FileMode.Open);
StreamReader sr = new StreamReader(fs);
string radek = sr.ReadLine(); // Uložení řádku do proměnné radek
sr.Close();
```

Zápis řádku do souboru je poměrně jednoduchou operací, kterou, jak jsme viděli, stačí pouze opakovat v případě, že chceme zapsat řádků více.

Čtení jednoho řádku je obdobné triviální, složitější situace však nastává ve chvíli, kdy chceme přečíst celý obsah souboru – metodu `ReadLine` sice můžeme volat opakovaně, zpravidla však nevíme, kolik řádků vstupní soubor obsahuje, a tedy nevíme ani kolikrát máme uvedenou metodu volat. Tento problém řeší ve třídě `StreamReader` boolovská vlastnost `EndOfStream`, jejíž hodnota je nastavena na `false`, dokud nebyl přečten poslední znak aktuálně otevřeného souboru. S vhodným využitím hodnoty `EndOfStream` v rámci podmínky cyklu jsme tak schopni zajistit přečtení celého souboru, jak demonstruje následující ukázka, v níž jej aplikace vypisuje po řádcích do pole `TextBox`.

```
FileStream fs = new FileStream(@"C:\a.txt", FileMode.Open);
StreamReader sr = new StreamReader(fs);
while (!sr.EndOfStream)
{
    textBox1.Text += sr.ReadLine() + Environment.NewLine;
}
sr.Close();
```

Úloha – Vytvořte aplikaci, která do textového pole přijme cestu k souboru a následně obsah daného souboru vypíše do víceřádkového textového pole.

Provádění vstupně výstupních operací se soubory může být relativně nebezpečné z hlediska možného vzniku výjimek, například v důsledku nedostatečných oprávnění pro přístup k souboru, z tohoto důvodu je vhodné v blocích kódu pracujících se soubory výjimky ošetřovat. Jeden z možných přístupu je ukázán zde.

```
FileStream fs = new FileStream(@"C:\a.txt", FileMode.Create);
StreamWriter sw = new StreamWriter(fs);
try
{
    sw.WriteLine("První řádek");
    sw.WriteLine("Poslední řádek");
}
catch (Exception vyjimka)
{
    MessageBox.Show(vyjimka.Message);
}
finally
{
    sw.Close();
}
```

Shrnutí kapitoly – Podobným způsobem, jakým je možné provádět výstup do konzole a získávat z ní uživatelské vstupy, můžeme v C# provádět také zápis do souborů a čtení z nich. Pro práci se soubory je třeba připojit jmenný prostor `System.IO`. Z něj využijeme třídu `FileStream`, která nám, dle nastavení, umožní přistupovat k souborům jako prvkům, do nichž lze zapisovat, nebo z nichž lze číst. Pro zápis do souboru potřebujeme vytvořit instanci třídy `StreamWriter` a užít její metodu `WriteLine`, pro čtení pak můžeme vytvořit instanci třídy `StreamReader` a užít její metodu `ReadLine` a vlastnost `EndOfStream`. Vytvořený `StreamWriter` a `StreamReader` nesmíme po skončení přístupu zapomenout uzavřít pomocí metody `Close`.

Závěrečná úloha – Vytvořte aplikaci, která pomocí dialogů `OpenFileDialog` a `SaveFileDialog` získá od uživatele cestu ke vstupnímu a výstupnímu souboru a následně obsah vstupního souboru zkopíruje do souboru výstupního.

11 Kolekce

Pole jsou v programování často užívanými konstrukcemi – není divu, umožňují nám adresovat pomocí jediného jména množství proměnných. Pole však mají určité vlastnosti, které mohou být v některých případech značně omezující. Zřejmě největší omezení je dáno nemožností změnit rozměry již existujícího pole a zvětšit (což je často žádoucí) či zmenšit jej (ke zmenšování polí v jednoduchých aplikacích zpravidla přistupovat nepotřebujeme, v případě složitějších programů však může být uvedená možnost užitečná). Možnost ukládání většího počtu prvků do jediného „kontejneru“ a jeho dynamické zmenšení či zvětšení nabízí právě kolekce, s nimiž se naučíme pracovat v této kapitole. Jejich nevýhodou však je, mimo jiné, větší paměťová náročnost oproti polím při stejném počtu prvků, je tedy vždy otázkou konkrétního algoritmu, zda je vhodnější použít kolekci či pole.

Kolekce jsou v C# rozděleny do několika skupin, my si představíme dvě nejpodstatnější z nich, jsou jimi kolekce negenerické a kolekce generické.

11.1 Negenerické kolekce

Tento typ kolekcí je specifický tím, že nemusíme specifikovat datový typ, který do něj chceme ukládat, a umožňuje nám tak vložit do něj najednou proměnné více datových typů – vedle sebe se v jedné negenerické kolekci může vyskytovat například proměnná typu `string`, proměnná typu `double` a proměnná typu `int`. Je to dáno způsobem, jímž tyto kolekce ukládají svůj obsah – před vložením jej totiž vždy přetypují na datový typ `Object`, což je základní datový typ, na nějž lze převést téměř libovolný datový typ. Tento postup má však několik nevýhod, z nichž pro nás zřejmě nejpodstatnější je nutnost zpětně přetypovat proměnné uložené v kolekci před použitím jejich hodnoty.

Cest k přetypování nabízí jazyk C# několik, odlišnosti mezi nimi jsou však z našeho pohledu poměrně drobné, bude nám tedy postačovat seznámit se s jednou z nich. Je jí explicitní přetypování, při němž, jak jsme již zmínili, aplikaci de-facto říkáme „dívej se na proměnnou `proměnná`, jako by šlo o typ `typ`“. Připomeňme, že zápis by pak byl následující:

```
(typ)proměnná;
```

V tuto chvíli nám zbývá jen poslední krok, abychom byli schopni používat negenerické kolekce, pro správné přetypování je totiž bezpodmínečně nutné znát původní datový typ, k němuž daný prvek v kolekci náležel před přidáním do kolekce. Abychom si nemuseli tuto informací ukládat „bokem“ a udržovat ji v paměti spolu s kolekcí, nabízí nám C# možnost jednoduchého dynamického stanovení datového typu určitého prvku. To se provádí pomocí operátoru `is` – jedná se de-facto o porovnání, jehož zápis je následující:

```
proměnná is datový_typ
```

Výsledkem uvedeného porovnání je boolovská hodnota (`true` nebo `false`), jejíž výsledek závisí na tom, zda vlevo uvedená proměnná je datového typu zapsaného vpravo. Následující kód demonstruje použití operátoru `is`.

```
int i = 456;
object o = (object)i;

if (o is int)
{
    // Kroky, které se vykonají, pokud je v objektu o uložena hodnota typu int
```

```
}
```

Použití negenerických kolekcí v nových projektech se nedoporučuje, pro některé aplikace jsou nicméně jejich vlastnosti přínosné. Je tedy minimálně vhodné vědět, jak fungují, a znát alespoň jednoho z jejich představitelů – my si zde představíme negenerickou kolekci `ArrayList`. K jejímu použití je však nejprve třeba připojit jmenný prostor `System.Collections`.

Kolekci je před použitím nutné vytvořit podobně jako je tomu v případě pole. Kolekci `ArrayList` je možno vytvořit (již tradičně) pomocí klíčového slova `new`, jak ukazuje následující příklad:

```
ArrayList kolekce1 = new ArrayList();
```

Jak vidíme, pro vytvoření kolekce není třeba stanovit její velikost – ta je na počátku určena automaticky a následně je dynamicky zvětšována/zmenšována.

Do kolekce můžeme po jejím vytvoření vkládat prvky. Nejjednodušším způsobem je využití její metody `Add`, která přijímá jako argument nově přidávaný prvek. V případě výše vytvořené kolekce `ArrayList` bychom tak mohli učinit například následujícím způsobem.

```
int i = 50;
kolekce1.Add(i);
```

Pomocí obdobné metody – `Remove` – můžeme prvky z kolekce vyjímat.

```
int i = 50;
kolekce1.Remove(i);
```

V tuto chvíli již zbývá jen představit možnost přístupu k jednotlivým prvkům v kolekci. Pro různé kolekce jsou tyto možnosti rozdílné, v případě kolekce typu `ArrayList` je pak nejjednodušší variantou přístupu k prvkům indexace (přístup tedy probíhá stejným způsobem, jako v případě polí).

```
ArrayList al = new ArrayList();
int i = 50;
int j = 60;
string s = "abc";
al.Add(i);
al.Add(j);
al.Add(s);
textBox1.Text = (string)al[2]; // Do textboxu bude vypsán text "abc"
```

Vzhledem ke skutečnosti, že prvky kolekce lze získávat pomocí indexů, není nijak překvapující, že pro výpis obsahu kolekce můžeme použít cykly. V případě cyklu foreach je situace relativně přímočará – pokud v něm použijeme jako datový typ prvků `var`, přetypování na původní hodnotu se navíc provede automaticky.

```
ArrayList al = new ArrayList();
al.Add(5);
al.Add(78.5);
al.Add("STR");
foreach (var item in al)
{
    textBox1.Text += item.ToString() + Environment.NewLine;
}
```

Pro použití cyklu for však potřebujeme znát počet prvků, který se v kolekci nachází. Ten lze získat pomocí vlastnosti `Count`. Následující příklad ukazuje výpis kolekce obsahující prvky typu `int` a `string`.

```
ArrayList al = new ArrayList();
al.Add(55);
al.Add("Abc");
```

```
al.Add(2);
al.Add(812);
al.Add("def");
al.Add(3);
al.Add("z");
for (int i = 0; i < al.Count; i++)
{
    if (al[i] is int)
    {
        textBox1.Text += (int)al[i] + Environment.NewLine;
    }
    else
    {
        textBox1.Text += (string)al[i] + Environment.NewLine;
    }
}
```

Úloha – Vytvořte aplikaci, která umožňuje pomocí tří textových polí a tří tlačítek přidávat do kolekce `ArrayList` proměnné typu `string`, `int` a `double`. Po stisknutí čtvrtého tlačítka vypište do víceřádkového textového pole postupně datové typy všech v kolekci uložených proměnných.

11.2 Generické kolekce

Generické kolekce jsou na rozdíl od svých negenerických protějšků pevně typované, což znamená, že při jejich vytváření musíme specifikovat datový typ, který do nich bude možno ukládat. V tomto ohledu jsou tedy o něco bližší polím. Výhodou je mimo jiné skutečnost, že zpětně načítané proměnné není pro použití třeba přetypovat. Pro práci s nimi je vhodné připojit jmenný prostor `System.Collections.Generic`.

Použití generických kolekcí je doporučováno ve většině případů, kdy není bezpodmínečně nutné využívat výhod kolekcí negenerických. Pro naše potřeby bude postačující seznámení s jednou z nich – kolekcí `List<T>`.

Jméno uvedené kolekce působí v rámci nám známé syntaxe jazyku C# poměrně netradičně. Zápis je nicméně o něco pochopitelnější když zjistíme, že výše uvedené písmeno `T` slouží pouze pro určení prostoru, do něhož je třeba vložit označení datového typu, který chceme do kolekce ukládat. Vytvoření kolekce `List<T>`, schopné ukládat prvky typu `string` bychom tak mohli provést pomocí kódu:

```
List<string> seznam = new List<string>();
```

Použití kolekce `List<T>` je analogické kolekci `ArrayList`, jednotlivé prvky tudíž můžeme přidávat pomocí metody `Add`,

```
seznam.Add("abc");
```

vyjímat je pomocí metody `Remove`

```
seznam.Add("abc");
```

a přistupovat k ním využitím jejich indexu

```
string s = seznam[0];
```

Pro výpis obsahu kolekce `List<T>` můžeme opět využít cyklů for a foreach.

Výpis pomocí cyklu for:

```
List<string> seznam = new List<string>();
seznam.Add("abc");
seznam.Add("DEF");
seznam.Add("xyz");
seznam.Add("fsa456");
for (int i = 0; i < seznam.Count; i++)
{
    textBox1.Text += seznam[i] + Environment.NewLine;
}
```

Výpis pomocí cyklu foreach:

```
List<string> seznam = new List<string>();
seznam.Add("abc");
seznam.Add("DEF");
seznam.Add("xyz");
seznam.Add("fsa456");
foreach (string s in seznam)
{
    textBox1.Text += s + Environment.NewLine;
}
```

Shrnutí kapitoly – Vedle polí nabízí C# ještě další konstrukce, které umožňují ukládání většího počtu prvků, jsou jimi kolekce. Od polí se liší zejména tím, že nemají staticky stanovenou délku a tedy mohou se dynamicky zvětšovat a zmenšovat. Dva důležité typy kolekcí jsou kolekce negenerické, umožňující ukládání proměnných libovolných datových typů, a kolekce generické, u nichž je potřeba při vytvoření zvolit datový typ, který budou ukládat. Z negenerických kolekcí je klasickým polím nejblíže `ArrayList`, z kolekcí generických pak `List<T>`.

Závěrečná úloha – Vytvořte aplikaci, která umožní uživateli pomocí textového pole a tlačítka přidávat do kolekce typu `List<T>` textové řetězce a pomocí dalšího textového pole a tlačítka odebírat zvolené textové řetězce z něj. Obsah kolekce po každé změně vypisujte do víceřádkového textového pole.

12 Vlastní definice objektů

Objekty jsou datovými typy velmi podobnými výše popsaným strukturám, jedním z rozdílů mezi nimi však je, že objekty jsou referenčními (viz kapitola 2) datovými typy, na což je třeba brát při práci s nimi ohled. Objekty jsou základem většiny moderních programů, v rámci této kapitoly se tedy naučíme tvořit profesionálněji působící kód.

Stejně jako u struktur je v případě objektů třeba je před použitím definovat. Definice, resp. popis toho, jak mají objekty vypadat a jak se mají chovat, se nazývají třídy. Obecná definice (prázdné) třídy je následující.

```
class JménoTřídy
{

}
```

Definice tříd zapisujeme mimo metody, případně i mimo jiné třídy – např. mimo třídu `Form1`.

To, jak má výsledný objekt „vypadat“, specifikujeme ve třídě pomocí tak zvaných atributů. Atributy neboli vlastnosti jsou proměnné, které se vytvářejí pomocí speciálního zápisu:

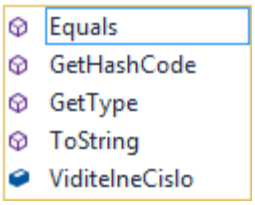
```
modifikátor_přístupu datový_typ jménoProměnné { get; set;}
```

Modifikátory přístupu slouží obecně ke specifikaci viditelnosti určitého prvku. Dvěma nejdůležitějšími jsou `public` (s nímž jsme se již setkali u struktur) a `private`. Pomocí prvního uvedeného nastavujeme určitému prvku viditelnost de-facto odkudkoli z našeho kódu, pomocí modifikátoru `private` pak říkáme, že daný prvek nebude viditelný mimo složené závorky, v nichž je definován. Rozdíl demonstrujeme na již zmíněných strukturách. Vytvoříme-li následující definici struktury

```
struct Struktura
{
    public int ViditelneCislo;
    private int NeviditelneCislo;
}
```

pak po vytvoření instance – proměnné typu Struktura – můžeme nastavovat pouze její člen ViditelneCislo, člen NeviditelneCislo nebude totiž mimo závorky definice struktury přístupný.

```
Struktura s = new Struktura();
s.
```



Pro naše další seznamování s objekty bude zcela postačující používání modifikátoru `public`.

Klíčová slova `get` a `set` uvedená ve složených závorkách za jménem atributu specifikují, že je možné jej nastavovat i číst. Čtení i zápis je pomocí určitých kroků možno dále omezovat, pro naše potřeby však stačí uvést, že pro umožnění běžné práce s vlastnostmi je nezbytné uvést obě slova.

Definice třídy Člověk s vlastnostmi věk a výška by na základě výše uvedeného mohla být následující:

```
class Clovek
{
```

```
    public int Vek { get; set; }
    public int Vyska { get; set; }
}
```

Objekty se vytvářejí (a jejich vlastnosti nastavují) stejným způsobem, jako v případě struktur. Pro objekt vytvořený na základě třídy `Clovek` a nastavení jeho vlastností bychom tedy mohli použít takovýto kód:

```
Clovek c = new Clovek();
c.Vek = 21;
c.Vyska = 175;
```

Úloha – Vytvořte aplikaci, v níž bude definována třída Automobil s vlastnostmi Model (`string`) a Výrobce (`string`). Pomocí tlačítka a dvou textových polí umožněte uživateli vytvářet a přidávat objekty typu Automobil do vhodné kolekce. Při změně kolekce vypisujte informace o všech v ní uložených automobilech do víceřádkového textového pole.

Chování objektů specifikujeme pomocí metod – pojmenovaných částí kódu – které v nich vytvoříme. Metody musí být vytvářeny mimo těla jiných metod. Zápis metody je obecně následující:

```
modifikátor_přístupu datový_typ JménoMetody(datový_typ jménoParametru)
{

}
```

Je vhodné zmínit, že metoda nemusí přijímat žádné parametry (pak jsou závorky v její definici prázdné), nebo jich může přijímat více (pak jsou odděleny čárkou). Datový typ uvedený před jménem metody specifikuje typ její návratové hodnoty. Metoda nemusí vracet žádnou hodnotu (v takovém případě jí v definici přiřazujeme datový typ `void`), nebo může vracet hodnotu v podstatě libovolného datového typu. Pokud metoda určitou hodnotu vrací, musí být v jejím těle (na konci) uvedeno klíčové slovo `return` následované hodnotou, kterou má metoda vrátit. Jak již víme, metodu lze v kódu zavolat pomocí jejího jména.

Níže je vidět příklad metody, která nepřijímá žádné parametry a nevrací žádnou hodnotu,

```
public void VypisPozdrav()
{
    MessageBox.Show("Hello world!");
}
```

metody, která přijímá parametr typu `int` a nevrací žádnou hodnotu

```
public void VypisCislo(int i)
{
    if (i == 0)
    {
        MessageBox.Show("nula");
    }
    else if (i == 1)
    {
        MessageBox.Show("jedna");
    }
    else
    {
        MessageBox.Show("číslo je větší než jedna nebo menší než nula")
```

```
    }
}
```

a metody, která přijímá dva parametry typu `int` a vrací hodnotu typu `int`.

```
public int SectiCisla(int a, int b)
{
    int c = a + b;
    return c;
}
```

Naši třídu `Clovek` můžeme upravit tak, aby námi vytvořený objekt člověk uměl stárnout – přidáme mu metodu Zestárni, která přijímá jako parametr počet let, o které má člověk zestárnout a vrací jeho nový věk. Pokud chceme v metodě uvnitř objektu upravit jeho vlastnost, používáme před jejím označením klíčové slovo `this`. Celá třída by pak vypadala následovně:

```
class Clovek
{
    public int Vek { get; set; }
    public int Vyska { get; set; }

    public int Zestarni(int pocetLet)
    {
        this.Vek += pocetLet;
        return this.Vek;
    }
}
```

Je vhodné zmínit, že všechny ovládací prvky, s nimiž jsme v prostředí Windows Forms pracovali, stejně jako samotné formuláře, jsou objekty. Při používání ovládacích prvků jsme tak nastavovali jejich vlastnosti a užívali jejich metody, další práce s objekty by pro nás tedy neměla být nikterak strastiplná.

Shrnutí kapitoly – Objekty jsou referenčním typem podobným strukturám, jejich definice se nazývají třídy a jsou uvozeny klíčovým slovem `class`. Třídy mohou obsahovat definici vlastností (proměnných se zvláštním druhem zápisu obsahujícím klíčová slova `get` a `set`) a metody (pojmenované části kódu, které mohou přijímat hodnoty a/nebo je vracet pomocí klíčového slova `return`. Pro vlastnosti i metody bývá třeba uvést odpovídající modifikátor přístupu, dva nejpodstatnější jsou `public` a `private`. Objekty vytváříme pomocí klíčového slova `new` a jména třídy.

Závěrečná úloha – Vytvořte aplikaci obsahující definici třídy Nábytek s vlastnostmi Typ (`sting`), Barva (`string`), Rozměry (`string`) a Cena (`int`) a metodou Sleva (s návratovou hodnotou `void` a nepřijímající žádný parametr), která sníží hodnotu uloženou ve vlastnosti Cena o 10%. Pomocí čtyř textových polí a jednoho tlačítka umožněte uživateli vytvářet a přidávat objekty typu Nábytek do vhodné kolekce. Detailní informace (vlastnosti objektů) o jejím obsahu při každé změně vypisujte do víceřádkového textového pole. Pomocí druhého tlačítka umožněte uživateli volat metodu Sleva u objektu, jehož index v kolekci uživatel zadá do šestého textového pole.