C# pro 1. ročník

Skripta PVA pro první ročník. Cílem tohoto materiálu je seznámit studenty se základy programování.

Algoritmizace

Úvodní sekce

V této kapitole se budeme věnovat principům logického převodu problému na kód, jeho zápisem a následným vyřešením. Tím se zabývá algoritmizace.

Co je to algoritmus?

Algoritmem rozumíme posloupnost konečného počtu přesně definovaných kroků potřebných k dosažení určitého výsledku či vyřešení určité úlohy. Příklad algoritmu pro opravu žárovky:

```
OpravaZarovky :
{
    Pokud: Svítí žárovka
    {
        Skoč na: Konec
    }

    Pokud: Není žárovka připojena na zdroj energie
    {
        Připojit žárovku na zdroj energie

        Pokud: Svítí žárovka
        {
            Skoč na: Konec
        }
    }

    Vyměnit žárovku

    Konec
}
```

Důležité vlastnosti algoritmu

Konečnost

Každý algoritmus musí vrátit výsledek v konečném počtu kroků.

Obecnost

Každý algoritmus by měl být univerzální. Například by cílem navržení algoritmu nemělo být vyřešit úlohu kolik je 3x4, ale implementace operace násobení pro jakýkoli vstup. Tedy pokud správně navrhneme algoritmus pro operaci násobení, pak by měl fungovat jak v případě, kdy jako vstupní data zadáme 3 a 4, ale také v případě, kdy zadáme čísla 23567 a 856.

Determinovanost (jednoznačnost výsledku)

Každý algoritmus musí vždy pro stejná vstupní data, vrátit stejná data výstupní.

Jednoznačnost (determinismus)

Každý algoritmus by měl být jednoznačný. Tedy každý jeho krok by měl být přesně a jednoznačně určen a mělo by být zcela zřejmé, co a jak který krok algoritmu provádí.

Resultativnost

Každý algoritmus by měl mít nějaký výstup, který je řešením zadaného problému pro vstupní data.

Elementárnost

Každý algoritmus by se měl skládat z konečného počtu jednoduchých kroků. Například by jeden krok neměl být “uvař vodu na kávu”, ale měl by obsahovat kroky jako vezmi konvici do ruky, otevři víko konvice atd.

> Ačkoli se všechny tyto vlastnosti mohou zdát jednoznačné a přirozeně jasné, je opravdu důležité na všechny tyto body při tvorbě programu myslet. Zejména elementárnost je velice důležitá pro další údržbu a změny v kódu. Na to však náš čtenář přijde postupem času sám.

Zápis algoritmu

V předchozí sekci jsme si představili dvě možnosti zápisu algoritmu. Prvnímu, spíše textovému způsobu se říká pseudokód. Druhý grafický způsob zápisu nazýváme vývojový diagram. Pseudokód je čistě slovní popis kroků, které je potřeba udělat k dosažení výsledku. Bývá členěn podobným způsobem, jako reálný kód zapsaný v programovacím jazyce. Jeho výhodou je, že po jeho napsání je poměrně snadné přejít k reálnému kódu. Z popsaných kroků nám vzniknou názvy funkcí a z dalších postupů nám vzniknou komentáře. Je tedy také vhodný pro další převod do různých jazyků. Vývojový diagram se zase hodí k vizualizaci problému a kroků algoritmu a dosažení lepšího

přehledu o algoritmu. Základními a nejčastěji používanými prvky vývojového diagramu jsou:

Obdélník se zakulacenými rohy

Používá se pro znázornění začátku a konce programu.

Obdélník s ostrými rohy

Používá se pro znázornění nějaké akce, funkce programu, či běhu nějakého podprogramu.

Kosočtverec posazený na hranu

Používá se pro znázornění nějaké podmínky nebo-li větvení běhu programu. Většinou z něj vedou dvě výstupní šipky. Jedna pro splněnou a jedna pro nesplněnou podmínku.

Obdélník se zkosenými stranami

Znázorňuje místo algoritmu, kde se načítá nějaký vstup, či vypisuje výstup.

> V praxi se pro menší programy používá buďto pouze jeden způsob zápisu algoritmu před převedením do reálného kódu nebo žádný. Pro větší aplikace se však většinou používají oba způsoby. Další možností zápisu algoritmu je samozřejmě zapsání ve námi vybraném programovacím jazyce. Této variantě se však budeme věnovat později.

https://skripta.ssps.cz/uploaded/IMG_5e396e467ede5.png

Shrnutí

Algoritmus je sada kroků užitých pro dosažení určitého výsledku. Základními vlastnostmi jsou konečnost, obecnost, determinovanost, jednoznačnost, resultativnost a elementárnost. K jeho zápisu se využívá pseudokód a vývojový diagram. Pseudokód v odborné programátorské literatuře naprosto převažuje a s vývojovým diagramem se setkáváme hlavně ve starší dokumentaci a v dokumentaci, která popisuje firemní procesy na obecnější úrovni. Nicméně je důležité umět číst obě notace - pseudokód i vývojové diagramy. Proto se je učíme. :)

Programovací jazyky

Úvod

`Programovací jazyk` je prostředek pro zápis `příkazů`, jež mohou být poté provedeny na počítači. Souhrn takovýchto příkazů pak nazýváme `programem`. Programovací jazyky můžeme dělit do rodin podobně, jako to děláme například s jazyky lidskými. Můžeme je dělit podle jejich účelu, generace nebo na jazyky s vysokou či nízkou úrovní abstrakce. Cílem této kapitoly bude seznámit se základními vlastnostmi programovacích jazyků, vývojovým prostředím a poté již napíšeme i náš první kód.

Účel jazyka

K dnešnímu dni existuje opravdu velké množství programovacích jazyků. Kromě jazyka `C#`, kterému se budeme věnovat my můžeme vyjmenovat například JavaScript, Kotlin, Lisp nebo Swift. Nyní se možná sami sebe ptáte: „Na co jich potřebujeme takové množství?“ Na počátku vzniku nového jazyka nejčastěji stojí potřeba řešit nějaký problém. V případě jazyka JavaScript to byla například nutnost přidání logiky do jinak statických (= ne příliš interaktivních) webových stránek a jazyk Kotlin vznikl jako moderní alternativa k Javě. Ne vždy ale tato potřeba byla technického rázu. Například C# byl odpovědí Microsoftu na neustálé licenční spory s firmou Sun (později odkoupena firmou Oracle), která Microsoftu nedovolovala editovat zdrojové kódy Javy. Z tohoto důvodu bychom dnes nalezli velké podobnosti mezi oběma jazyky.

Úroveň jazyka

Jazyky můžeme dělit na `nízkoúrovňové` a `vysokoúrovňové`. Nízkoúrovňové jazyky obvykle operují na nejnižší softwarové úrovni a disponují velkou rychlostí. Daní za tuto rychlost je pak člověkem obtížná čitelnost, která prakticky znemožňuje psaní složitějšího softwaru, a tudíž klade i velké nároky na programátora. Vysokoúrovňové jazyky (někdy se též můžete setkat s pojmem jazyky s vysokou mírou `abstrakce`) jsou pak pravým opakem předešlé skupiny. Takovéto jazyky odstiňují programátora od rutinních úkolů jako je například alokace paměti nebo její další správa, což z nich dělá skvělé kandidáty na psaní rozsáhlejších aplikací, kde není na první místě výkon. Tyto pokročilé nástroje se totiž vždy negativně podepisují na celkovém výkonu jazyka.

> Nic nebývá jen černé nebo bílé a našli bychom jazyky, které se pohybují na pomezí obou výše zmíněných skupin.

>

> Pojem abstrakce v informatice označuje způsob práce s daty nebo technickými prostředky počítače, jakým by s nimi pracoval člověk. Příkladem budiž například proměnná, která je pro člověka abstraktním pojmem a každý si pod ní představíme například libovolnou hodnotu. Počítač však pojem proměnná nezná a pracuje pouze s jedničkami a nulami. Míru abstrakce tedy vždy zajišťuje daný programovací jazyk.

Interpretace a kompilace

Jazyk jako takový je pouze sbírka pravidel (specifikace), které musíme dodržovat. Pokud tak neučiníme, náš zápis není v daném jazyce validní (správný). O samotný překlad našeho kódu (například do spustitelné aplikace) se stará nejčastěji `interpreter` nebo `kompilátor`.

Kompilátor

Začneme tou jednodušší variantou. Kompilátor je program, který analyzuje náš kód, odhalí chyby a v případě, že žádné nenajde zkompiluje naší aplikaci. V případě jazyka C# výsledkem kompilace nejčastěji bývá spustitelný soubor (například `.exe`). V tento moment už kompilátoru není zapotřebí a jeho úloha zde končí. Výsledný soubor .exe je v tuto chvíli plně přenositelný například mezi zařízeními operujícími pod Windows a to i mezi těmi, které kompilátor nainstalovaný nemají.

> V případě jazyka C# je náš zdrojový kód přeložen do CIL. Jedná se o nejnižší člověkem čitelnou implementaci našeho programu. CIL se postará o správný běh našeho programu na specifickém hardwaru počítače, který se pochopitelně může lišit stroj od stroje.

Interpreter

Interpreter je na rozdíl od kompilátoru přítomen po celou dobu běhu programu. Bez interpreteru daného jazyka si v něm napsaný program nespustíme. Překlad takového programu nejčastěji probíhá příkaz po příkazu, kdy interpreter obdrží příkaz daného jazyka a poté jej rovnou přeloží na spustitelnou instrukci.

> Ve skutečnosti je vše ještě trochu složitější a našli bychom případy, kdy je využívána například i kombinace obojího. Za účelem stručnosti článku se tímto tématem nebudeme hlouběji zabývat. Kdo by měl však zájem si může jistě spoustu informací dohledat sám, neboť o této problematice byl napsán nespočet odborných publikací a článků.

Syntaxe, sémantika a konvence

Pojmem `syntaxe` rozumíme souhrn pravidel pro psaní výrazů v programovacím jazyce. Syntaxe jazyka C# nám například přikazuje, že veškeré příkazy ukončujeme znakem středníku (`;`) nebo že

názvy proměnných nesmí začínat číslovkou. `Sémantika` je pak slovník daného jazyka. Jako příklad si ukážeme funkci `Console.Write()`, jež slouží pro výpis textového řetězce (tzn. nějaké věty nebo slova) do konzole. V jazyce JavaScript bychom stejný úkol provedli příkazem `console.log()` a v Ruby za pomocí `puts`. `Konvence` je souhrn pravidel, jež bychom měli dodržovat, ale jejich porušení neohrozí běh a funkčnost programu. Příkladem budiž třeba doporučení, že náš zdrojový kód by neměl obsahovat speciální znaky jako je třeba `á`, `ü` nebo `□`.

Jazyk C# a platforma .NET

Když už známe některé druhy jazyků, jejich účel a vlastnosti, pojďme se nyní i blíže podívat na samotný C#.

Jazyk C# (vysluvujeme [sí šarp], někdy též psáno jako C Sharp) je kompilovaný, staticky typovaný vysokoúrovňový jazyk. Spadá do rozsáhlé platformy jménem `.NET` (vyslovujeme [dot net]), kterou si lze představit jako sbírku softwarových řešení, která zastřešuje například mobilní telefony, web od serverové části po klientskou, desktopové aplikace a kromě C Sharpu bychom zde našli i několik dalších programovacích jazyků.

Dříve platilo, že byl tento jazyk (včetně platformy .NET) úzce spjat s operačním systémem Windows. Dnešní .NET framework umožňuje psaní mobilních aplikací například pro operační systémy iOS, WatchOS, iPadOS, Android nebo Android Wear (Xamarin framework), psaní serverových aplikací (ASP.NET), vývoj webového frontendu (Blazor), vývoj aplikací na platformě Linux (MonoDevelop) a našli bychom toho ještě určitě víc.

K samotnému vývoji v jazyce C# nám postačí vždy nějaká konkrétní implementace frameworku .NET. Například pokud chceme vyvíjet pouze na platformě Windows, postačí nám původní verze jménem `.NET Framework`, která je vyvíjena a průběžně aktualizována již od roku 2002. Pokud nás zajímá multiplatformní vývoj nebo vývoj na webu, sáhneme po `.NET Core`. Pokud jsou našim cílem mobilní zařízení nebo vývoj her v herním enginu Unity, použijeme `MonoDevelop`. V případě, že nás zajímají univerzální aplikace pro Windows 10, framework `UWP` (Universal Windows Platform) nám poskytne potřebné nástroje.

Každá tato odnož frameworku (.NET Framework, Core, Mono a UWP) navíc vždy obsahuje kompilátor, dodatečné pomocné frameworky (např. Windows Forms nebo Xamarin) a všechny další nástroje potřebné k vývoji.

> Pokud to pro vás bylo příliš mnoho informací najednou, nemusíte si dělat starosti. Microsoft nabízí `vývojové prostředí`, které udělá všechnu náročnou práci za vás a je tedy velice snadné hned začít s psaním aplikací.

Vývojové prostředí

`Vývojové prostředí` též někdy označované jako `IDE` (integrated development environment – integrované vývojové prostředí) je chytrý editor kódu, který nám pomůže při psaní našich programů a pokud ho budeme používat efektivně, výrazně urychlí i čas vývoje. Microsoft pro vývojáře dodává hned několik variant svého vlastního řešení. Pro Windows je to `Visual Studio` (v době psaní článku Visual Studio 2019), pro Mac OS `Visual Studio for Mac` a na Linuxu můžeme využít například `MonoDevelop`. Pro všechny tyto 3 platformy je dostupné také vývojové prostředí `Visual Studio Code`, které se nejlépe hodí pro webový vývoj.

Každé toto vývojové prostředí na dané platformě nabízí trochu jiné možnosti, neboť operuje pod jinou odnoží frameworku .NET. V tomto materiálu se budeme primárně zabývat pouze konzolovými aplikacemi (tzn. aplikacemi jež ovládáme přes příkazovou řádku) a pro tyto účely je vyhovující jakékoliv výše zmíněné řešení.

> Jak již bylo avizováno výše, prostředí Visual Studio Code je vhodné spíše pro webový vývoj a jeho konfigurace pro kompilaci konzolových aplikací by mohla být pro začátečníka náročná. Doporučujeme tedy používat nejlépe standardní Visual Studio pro Windows nebo Mac.

Instalace vývojového prostředí

Nyní si ukážeme, jak probíhá instalace vývojového prostředí `Visual Studio Community 2019`. Odkaz na stažení tohoto prostředí můžete velmi snadno nalézt na oficiálních stránkách microsoftu. Po spuštění instalačního souboru se vám otevře průvodce instalací:IMG_5e39724798307.png*Úvodní obrazovka instalátoru.*

IMG_5e397251b09cf.png*Instalátor nabízí velké množství sad, které můžeme využít. Nás bude zajímat pouze "Vývoj desktopových aplikací pomocí .NET".*

IMG_5e3972608d86b.png*Tuto volbu zaškrtněte.*

U programování platí stejně jako ve válce - těžko na cvičišti, lehko na bojišti. Je moc pěkné, že si Microsoft dal tu práci a přeložil Visual Studio do češtiny, ale programátorů tím prokázal medvědí službu - když budou chtít najít pomoc na StackOverflow nebo jinde, dostanou instrukce v angličtině, které budou potom v českém prostředí horko-těžko aplikovat.

Je lepší si hned od začátku zvykat na tvrdší dosed a instalovat anglickou verzi namísto české. Hodně dobře je to vidět u pokročilejších funkcí VS, jako je práce s gitem - tam jsou překlady do češtiny naprosto kontraproduktivní.

Pro potřeby těchto skript si bohatě vystačíme pouze s balíčkem Vývoj desktopových aplikací pomocí .NET. Pokud jste spokojeni s výběrem balíčků, dokončete instalaci.

Založení nového projektu

Po instalaci a spuštění vývojového prostředí nyní zkusíme založit i náš první projekt. Z menu vybíráme možnost `Vytvořit nový projekt` a poté `Konzolová aplikace (.NET framework)`. Po potvrzení by se nám měl otevřít textový editor s kostrou naší aplikace.

> Instalace vývojového prostředí a následné založení nového projektu na systému MacOS probíhá velmi podobně. Na macu ovšem zakládáme konzolový projekt na platformě `.NET Core` (jak je vidět na posledním obrázku níže), neboť odnož platformy .NET jménem `.NET Framework` zaštiťuje pouze systém Windows. Tato problematika byla podrobněji rozepsána v předchozí sekci.

IMG_5e397327dbec3.png

IMG_5e39733d251dc.png*Poté vybíráme "Konzolová aplikace (.NET Framework)"*

IMG_5e39732e48280.png*Zadáme podrobnosti a klikáme na tlačítko "Vytvořit".*

IMG_5e39733570196.png*Takto by měl vypadat prázdný projekt.*

IMG_5e397344863c1.png*Uživatelé systému MacOS vybírají konzolovou aplikaci pod záložkou ".NET Core".*

Struktura našeho programu by nyní měla vypadat nějak takto:

```
using System;

namespace MojeAplikace
{
    class Program
    {
        public static void Main(string[] args)
        {

        }
    }
}
```

Prozatím si ukážeme pouze nutné základy a vše ostatní probereme v příslušných kapitolách tohoto materiálu. V jazyce C# má každý znak svůj smysl a význam. Jak si můžeme povšimnout, ačkoliv jsme založili úplně nový projekt, vývojové prostředí nám již předgenerovalo určitou aplikační strukturu. Složené závorky tvoří tzv. blok kódu. Tento blok začíná otevřenou složenou závorkou (`{`) a končí uzavřenou složenou závorkou (`}`) na stejné úrovni:

```
using System;

namespace MojeAplikace
{ // Začátek bloku "namespace"
    class Program
    { // Začátek bloku "class"
        public static void Main(string[] args)
        { // Začátek bloku Main

        } // Konec bloku Main
    } // Konec bloku "class"
} // Konec bloku "namespace"
```

Na ukázce výše jsme použili tzv. komentář. Cokoliv za dvěma lomítky (//) na stejném řádku bude kompilátorem ignorováno. Setkat se můžeme i s víceřádkovým komentářem:

```
using System;

namespace MojeAplikace
{
    class Program
    {
        public static void Main(string[] args)
        {
            /*
                Komentář přes více
                řádků!
            */

            // Komentář na jeden řádek
        }
    }
}
```

Prozatím si zvykneme na to, že veškerý kód budeme vždy psát do nejhlouběji zanořených složených závorek, než si postupem času odtajníme význam všech nám neznámých jazykových konstruktů. Pro kontrolu, že jsme vše nainstalovali správně si nyní zkusíme spustit tento fragment kódu poklepáním na ikonku zelené šipky v horním toolbaru. Pokud vše proběhlo bez chyb, měli bychom nyní vidět konzoli s nápisem `Ahoj světe!`. Po poklepání na libovolnou klávesu by se konzole měla zavřít.

```
using System;

namespace MojeAplikace
{
    class Program
    {
        public static void Main(string[] args)
        {
            /*
                Všechny budoucí ukázky budou počítat s tím, že kód bude vkládán
                do nejhlouběji zanořených složených závorek takto:
            */

            Console.WriteLine("Ahoj světě!");
            Console.ReadKey();
        }
    }
}
```

Shrnutí

V tomto článku jsme si představili některé základní vlastnosti programovacích jazyků, nainstalovali jsme si vývojové prostředí a napsali náš první program. V dalších sekcích se podíváme na základní jazykové kontrukce a objasníme si pojem proměnná.

Proměnné a konstanty

Úvod

Pojem `proměnná` (`variable`) už známe z matematiky (především z nauky o funkcích). V programování se budeme s proměnnými setkávat neustále. Co to tedy proměnná vlastně je

Vymezení pojmu

Proměnné nám umožňují uchovávat libovolnou hodnotu pod námi zvoleným názvem. Zavedení proměnné a následné přiřazení její hodnoty pak matematicky vyjádříme například takto: `x = 1`.

> V jazyce C# budeme proměnné vytvářet obdobným způsobem. Než si ale ukážeme konkrétní postup, budeme muset pochopit, co jsou to datové typy a typový systém. To je předmětem další kapitoly.

Kdybychom chtěli osvětlit pojem z hlediska hardwaru, mohli bychom říci, že proměnná je pro nás označením nebo pojmenováním určité části paměti počítače. Z hlediska našeho oboru (informatiky a programování) je proměnná jazykový konstrukt, který nám umožní ukládat data všeho druhu.

Pravidla pro práci s proměnnými

Program nemůže obsahovat více proměnných stejného názvu. Daná problematika je ve skutečnosti ještě o něco složitější a existují případy, kdy je i toto možné. Pro zachování jednoduchosti článku ale zamlčím určitá fakta a zatím si postačíme s tímto neúplným pravidlem, než ho v budoucích kapitolách uvedeme na pravou míru.

Pro volbu názvů proměnných existuje několik jasných pravidel a několik poněkud méně jasných, ale stále důležitých doporučení. Název proměnné může obsahovat písmena, číslice a některé speciální znaky. Začínat ale musí vždy písmenem nebo povoleným speciálním znakem – například podtržítko (`_`) nebo v některých případech i zavináč (`@`). Jazyk C# je velmi benevolentní a dovoluje například i použití písmen z ruské abecedy nebo jiné cizojazyčné znaky. **Že to dovoluje ale neznamená, že byste měli znaky z jiných abeced používat.** Pro naše účely bude nejlepší si pamatovat, že háčky, čárky ani jiné exotické znaky do zdrojového souboru nepatří. Jedinou výjimkou z tohoto pravidla jsou obsahy proměnných typu string, ve kterých jsou české znaky přípustné (a žádoucí). Tedy string `p = "Zadejte desetinné číslo:"` je v pořádku, ale deklarace `string příjmení = "Novák"` v pořádku není. Přestože kompilátor C# nebude protestovat proti názvu proměnné "příjmení" (s háčkem a čárkami), považuje se mezi programátory použití takového názvu za nepřijatelné a

trestuhodné.

Pokud z nějakého důvodu i tak chceme použít klíčové slovo jako název proměnné, prefixujeme daný název znakem zavináče (`@`). Například tedy `@int` nebo `@out`. Pokud to není nezbytně nutné, nedělejte to a najděte lepší název!

Konvence pro práci s proměnnými

Pro jistotu znovu uvedu, že konvence jsou doporučení, která dodržujeme (nebo se o to aspoň snažíme), ale při jejich porušení neohrozíme funkčnost programu. K proměnným se váže velké množství konvencí a my si nyní představíme některé z nich.

Mezi první doporučení patří, že názvy proměnných píšeme s malým počátečním písmenem a vždy bez diakritiky nebo jiných speciálních znaků. Například tedy `jmeno`, `vyska` nebo `prumer`. Pokud se název proměnné skládá z více slov, používáme tzv. `camel case` (česky velbloudí notace). To znamená, že jednotlivá slova neoddělujeme mezerou, ale pouze velkým počátečním písmenem nového slova. Příkladem budiž `pocetStudentu`, `prumernaMzda` nebo `vazenyAritmetickyPrumer`.

Dále se snažíme, aby názvy proměnných byly „sebepopisné“. Tedy například budeme-li popisovat nějakou osobu, její jméno budeme mít v proměnné `jmeno`, věk v proměnné `vek` a datum narození v proměnné `datumNarozeni`. Výjimku z tohoto pravidla tvoří především celočíselné indexační proměnné, pro které se typicky volí jednopísmenné názvy jako `i`, `j`, `k` apod. O takovýchto proměnných se více dozvíme v budoucích kapitolách. Nikdy však nezacházejme v sebepopisnosti do extrémů. Délka jména proměnné by neměla být neúnosná.

> Ačkoliv to nemusí být na první pohled zřejmé, dodržování konvencí má svůj smysl a je naprosto klíčové pro psaní přehledného a do budoucna rozšiřitelného kódu.

Konstanty

Pro `konstanty` platí stejná pravidla a doporučení jako pro proměnné. Jak ale jejich název napovídá, narozdíl od proměnných jsou neměnné.

Příklad

```
const int minDelkaHesla = 8;
```

Hodnota konstanty musí být známá v okamžiku překladu (kompilace) programu. Například

Chybná ukázka použití konstanty

```
int i = 5;
const int minDelkaHesla = 4 * i;
```

vyhodí chybu, protože hodnota na pravé straně přiřazení není v okamžiku kompilace známá.

> Přesný význam tohoto kódu si osvětlíme v příští kapitole.

Kompilátor kontroluje, zda se kdekoliv v programu nevyskytuje `minDelkaHesla` na levé straně od operátoru přiřazení (`=`). Na rozdíl od jiných jazyků se konstanty obvykle nepíší velkými písmeny (v tomto případě `MINDELKAHESLA`), ale dodržuje se obvyklá "velbloudí" konvence.

Literály

Hodnoty, které definujeme ve zdrojovém kódu se nazývají literály. Zde je několik příkladů:

Ukázka literálů

```
bool jeDoma = false;  // typ bool může mít hodnoty true nebo false
char velkeA = 'A';  // jednoduché uvozovky, není totéž, co string velkeA = "A"!!
char velkeA = '\u0065';  // totéž co předchozí deklarace, tentokrát pomocí kódu znaku v
Unicode
char tabulator = '\t';
char novyRadek = '\n';
byte b = 64;
int i = 16;
int k = 0x10;  // celočíselný literál v šestnáctkovém(hexadecimálním) tvaru
long velkeCislo = 68000L;
int velkeCislo = 68000L;  // vyhodí chybu, přestože hodnota je v rozsahu povoleném pro int,
znak L vynutí větší alokovanou paměť
double polomer = 2.5;  // když je to bez písmene, považuje se hodnota 2.5 za double
decimal polomer = 2.5m;  // když chceme zvýšenou přesnost, používáme typ decimal. Deklarace
decimal polomer = 2.5 vyhodí chybu - viz implicitní konverze v další kapitole
float polomer = 2.5f;  // i v tomto případě float polomer = 2.5 vyhodí chybu - viz implicitní
konverze v další kapitole
double vzdalenostMesice = 384.4e6; // == 384.4 * 106
double deformace = 1.3e-3; // == 1.3 * 10^-3
string cesta = "C:\\Windows\\Notepad.exe";
string cesta = @"C:\Windows\Notepad.exe";
string nazor = "Nejlepší skladbou Jimmiho Hedrixe je \"Hey Joe\"";
```

Složené závorky

Složené závorky definují blok kódu a zároveň obor platnosti proměnných:

Blok kódu (chyba)

```
{
  int i = 1;
  int i = 2;
}
```

V tomto případě kompilátor vyhodí chybu, protože v jednom bloku nemohou být deklarované dvě proměnné stejného jména.

Blok kódu (v pořádku)

```
{ // blok A
  int i = 1;
}
{ // blok B
  int i = 2;
}
```

Tento kód je v pořádku. První proměnná i má rozsah platnosti v bloku A a když program opustí blok A, proměnná i, která je v něm definovaná, zanikne. Další proměnná i má rozsah platnosti v bloku B a není tedy v konfliktu s první proměnnou i z bloku A.

Závorky nejsou jen oddělovače a matoucí smetí!

Shrnutí

Proměnné jsou cesta, jak dočasně uchovávat data všeho druhu. V další kapitole si ukážeme, jak proměnnou založit a napíšeme tedy i náš první kód.

Datové typy a konverze

Úvod

Svět kolem nás je plný informací. Informace může být tak prostá, jako počet žáků ve třídě vyjádřený číslem nebo naopak velmi komplexní – například školní databáze studentů. Pro zpracování těchto informací v jazyce C# nestačí znát jen konkrétní data (jména a počty studentů), ale zároveň i vědět, jak s těmito daty pracovat. To je hlavním důvodem existence `datových typů` v jazyce C#.

Datové typy

Vzpomínáte si na proměnné z minulé kapitoly? Ke každé proměnné se v jazyce C# váže i tzv. `datový typ` (`data type`). Ten nám říká, jaké povahy proměnná je a zároveň tím i určujeme, jakým způsobem budeme s danými daty uvnitř proměnné pracovat. Datových typů je velké množství a my si pro začátek uvedeme jenom jeden základní - `int`. Datový typ `int` (z anglického integer) slouží k ukládání celých čísel. Nejlepší bude si vše ukázat na jednoduchém příkladu:

Deklarace proměnné

```
int x;
```

Deklarace konstanty

```
const int y = 5;
```

Tímto zápisem jsme založili proměnnou se jménem `x`. Druhá ukázka pak demonstruje založení konstanty s hodnotou `5`. Povšimněte si, že samotnému názvu proměnné `x` ještě předchází i onen datový typ `int` a v případě konstanty i klíčové slovo `const`. Naše první proměnná zatím nedrží žádnou konkrétní hodnotu. Pouze jsme vyjádřili, že tuto proměnnou chceme založit (`inicializovat`) a prozatím ji ponechat „prázdnou“. Tomuto procesu říkáme odborně `deklarace proměnné`. Každý příkaz v jazyce C# je pak vždy ukončen středníkem (`;`). Nyní už pojďme zkusit naší proměnné přiřadit i nějakou hodnotu:

Proměnná s přiřazením

```
int x = 1;
```

Touto změnou jsme kromě deklarace proměnné provedli i přiřazení hodnoty `1`. Je důležité si uvědomit, že když přiřazujeme hodnotu k proměnné, jsme vždy limitováni jejím datovým typem. V našem případě jsme omezeni pouze na celá čísla. Pokoušet se tedy například uložit do proměnné `x` hodnotu `3.1415` povede k chybovému hlášení z důvodu typové chyby. Omezeni jsme rovněž i pamětí počítače. Rozsah datového typu `int` je přibližně od mínus dvou miliard do plus dvou miliard. Pro ukládání větších čísel nebo čísel s desetinným rozvojem nám poslouží jiné datové typy.

> Zatímco v matematice znak „`=`“ vyjadřuje rovnost dvou výrazů (levé a pravé strany), v jazyce C# (a řadě dalších programovacích jazyků) tento znak znamená pokyn k „přiřazení pravé strany do levé“. Po (úspěšném) provedení takovéhoto řádku bude sice skutečně platit ona matematická rovnost mezi levou a pravou stranou, ale to je až přirozený důsledek přiřazení.

Kdybychom nyní chtěli, můžeme hodnotu proměnné opět změnit:

Změna hodnoty proměnné

```
// Deklarace a přiřazení k proměnné
int x = 1;

// Změna hodnoty proměnné
x = 32;
```

Pokud byla proměnná již deklarována, není nutné při práci s ní opětovně zmiňovat její datový typ. Pokud bychom tak učinili ve stejném kontextu (o tom více v dalších kapitolách), kompilátor by náš zápis vnímal jako snahu založit novou stejnojmennou proměnnou, což by opět vedlo k pádu. Po našich posledních úpravách bude tedy konečná hodnota proměnné `32`.

Nyní si představíme ještě jeden důležitý datový typ. Je jím `string`. Ten nám dobře poslouží při ukládání textových řetězců. Opět si vše ukážeme na příkladu:

Datový typ string (1)

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

Datový typ string (2)

```
// Deklarace
string pozdrav;

// Přiřazení
pozdrav = "Ahoj, jak se máš?";
```

Způsob zápisu je pořád stejný. Datový typ předchází názvu proměnné a poté je provedeno samotné přiřazení. Tento příkaz samozřejmě může být také opět rozdělen do dvou samostatných řádků (ukázka 2). Nová je pro nás pouze pravá část přiřazení. V jazyce C# vždy obalujeme textové řetězce do uvozovek (`"`). Dáváme tím najevo, že se nejedná o klasický přeložitelný kód, ale náš psaný text. Do datového typu `string` můžeme rovněž ukládat i čísla, protože ta jsou sama o sobě ostatně také jenom textovým řetězcem. Musíme je ale opět obalit do uvozovek tak, jak je to vidět v následující ukázce:

Číslo jako řetězec

```
// Celé číslo uložené jako textový řetězec
string y = "10";
```

Na ukázce číslo 2 je pak porovnání obou dvou přístupů:

Porovnání

```
int x = 10;
string y = "10";
```

> V budoucích kapitolách si ukážeme, jaký je mezi oběma přístupy rozdíl a jak přesně datové typy ovlivňují práci se samotnými daty uvnitř proměnných.

> V některých jazycích je rovněž povoleno psát textové řetězce mezi apostrofy ('). Jazyk C# toto neumožňuje, protože apostrof je vyhrazen pro práci s datovým typem char, který slouží k uložení pouze jednoho znaku:

Datový typ char

```
char znak = 'A';
```

Použití datových typů

Vhodná volba datového typu je zásadní pro funkci i čitelnost programu. Datový typ vyplývá z charakteru ukládaných informací a musí je co nejlépe vystihovat. Například:

Proměnná `pocatecniPismeno` musí mít typ `char` a nemůže mít typ `string` (ten je vyhrazen pro text a počáteční písmeno je JEDNO písmeno, ne text.

Proměnná `polomer` definuje vzdálenost bodů kružnice od středu, což je fyzikální veličina, která může nabývat desetinných hodnot, například `2.35` centimetru. Musí tedy mít desetinný typ (double, float nebo decimal - podle požadavků na rychlost nebo přesnost) a nemůže být `byte`, `int`

nebo `long`. Totéž se týká času, rychlosti, zrychlení, teploty, tepelné kapacity, proudu, napětí, atd.

Proměnná nadpis musí být typu `string` a nemůže být typu `char` - jedná se o více písmen za sebou.

Proměnná `pocetClenuDomacnosti` nemůže být jiný než byte nebo `int`, nikdy desetinný typ. Lidé nejsou ve zlomcích, stejně tak počet čísel, které jsou větší než `2.5`. Ani ta nemůžeme vyjádřit jinak, než celočíselným typem (`byte`, `int`, `long`, `uint`...)

Další datové typy

V předchozí sekci jsme si představili datové typy `int` a `string`. Nyní si naše portfolio rozšíříme ještě o několik dalších. Prvním takovým bude datový typ `double`. Ten se nám bude hodit při ukládání čísel s desetinným rozvojem. V jazyce C# oddělujeme desetinnou část čísla vždy tečkou (`.`):

Datový typ double

```
double d = 3.14;
```

Je nutné zmínit, že tento datový typ není úplně přesný a při dlouhém desetinném rozvoji nad 15 až 16 míst má tendence zaokrouhlovat. Nehodí se tedy například pro práci s penězi nebo kdekoliv jinde, kde je vyžadována vysoká přesnost. Právě pro tyto případy tu máme datový typ `decimal`:

Datový typ decimal

```
decimal x = 34.58m;
```

Za samotné desetinné číslo ještě v případě decimalu píšeme i suffix (příponu) `m` nebo `M`. Díky této příponě můžeme odlišit desetinné číslo typu `double` od desetinného čísla typu `decimal`. Kdybychom na konec čísla příponu nedoplnili, jazyk by naše počínání vnímal jako snahu přiřadit hodnotu typu `double` do proměnné typu `decimal`, což by vyústilo v chybové hlášení.

> Jakékoliv desetinné číslo bez přípony je díky `implicitní konverzi` bráno jako `double`. Kdybychom ale chtěli, můžeme i přesto příponu `d` nebo `D` doplnit. O implicitních konverzích a jejich dopadu na náš kód se dozvíme více později.

Double - přípona

```
double y = 16.73d;
```

Mezi datové typy s plovoucí desetinnou čárkou ještě řadíme typ jménem `float`. Proces založení je stejný jako u obou předchozích, mění se opět pouze přípona:

Datový typ float

```
float z = 8.763f;
```

Nyní by někoho mohlo zajímat, proč pro práci s desetinnými čísli máme 3 odlišné datové typy. Důvodem jsou různé přesnosti a rychlosti těchto typů. Jako nejpřesnější se osvědčil datový typ `decimal` s 28 až 29 desetinnými řády přesnosti. Daní za tuto přesnost je ale podle některých měření až 20x nižší rychlost oproti zbylým dvěma datovým typům. Zlatou střední cestou je typ `double`, který zachovává přesnost na 15 až 16 míst s přiměřenou rychlostí. Nejrychlejší je pak datový typ `float` s garantovanou přesností pouze na 7 řádů. Posledním naším datovým typem je `char`. Ten slouží k uložení pouze jednoho znaku. Ukládaný znak pak obalujeme vždy mezi dva apostrofy, jak bylo avizováno výše:

Datový typ char

```
char c = 'a';
```

> Tímto pochopitelně náš výčet datových typů nekončí. Další si ukážeme v budoucích kapitolách.

Typový systém

Jazyk C# používá `statický typový systém`, který byl popsán v předchozích sekcích. V praxi to znamená, že proměnné vždy vyžadují definovat datový typ (například tedy `int` nebo `string`), který je dále neměnný. Jakmile jednou proměnnou deklarujeme, není možné její datový typ změnit. Když bychom například chtěli do proměnné typu `int` uložit desetinné číslo `10.67`, dostaneme chybové hlášení.

Opakem je `dynamický typový systém`, který nás plně odstiňuje od toho, že proměnná nějaký datový typ vůbec má. Dynamické typování jde mnohdy tak daleko, že proměnné nemusíme ani deklarovat. Jakmile do nějaké proměnné uložíme hodnotu a jazyk zjistí, že nebyla nikdy deklarována, sám ji založí. Do jedné proměnné můžeme ukládat text a poté ji třeba přepsat celým číslem. Interpreter nebo kompilátor daného jazyka se s tím sám popere a vnitřně automaticky mění datový typ. Zástupci dynamicky typovaných jazyků jsou například `JavaScript` nebo `Ruby`.

Dynamické typování sice nyní vypadá jako velmi výhodné, ale zdrojový kód pak není možné automaticky kontrolovat proti typovým chybám. Pokud někde očekáváme textový řetězec a přijde nám tam místo toho číslo, odhalí se chyba až za běhu a v lepším případě program spadne a v tom horším nebude pracovat správně (mnohdy bez našeho vědomí). Ačkoliv tedy statické typování klade větší nároky na programátora, ochrání nás před nepříjemnými chybami, jejichž hledání a následné řešení často může zabrat i hodiny. Dynamické typování rovněž ubírá na celkové výkonnosti jazyka, protože automatická úprava typů za běhu je výpočetně náročná.

Na závěr je potřeba říct, že každý typový systém má své pro a proti. Jazyk C# je staticky typovaný, a proto pro nás bude klíčové, abychom problematiku datových typů dokonale chápali.

Konverze

Při programování velmi často vzniká potřeba převést proměnnou na jiný datový typ než ten stávající. Příkladem může být situace, kdy uživatelem zadaný řetězec (`string`) chceme převést na desetinné číslo (například `float`). Jak již bylo napsáno výše, po deklaraci proměnné je v jazyce C# nemožné její datový typ měnit. Pro účely převodu proměnné na jiný datový typ tedy musíme založit úplně novou pomocnou proměnnou a samotné `přetypování` (převod datového typu) poté provést za pomocí `implicitní konverze`, `explicitní konverze` nebo za pomocí `podpůrných tříd`. Vše si nyní pokusíme co nejjednodušeji vysvětlit.

 # Implicitní konverze

Tento druh konverzí se děje automaticky podle předdefinovaných pravidel. Pro demonstraci této látky trošku předběhneme tematický plán a předvedeme si tuto problematiku v kombinaci s operátorem součtu, kde se tento typ převodu děje nejčastěji:

Implicitní konverze

```
double x = 1 + 2.5; // x => 3.5
```

Na této ukázce sčítáme číslo 1 (datový typ `int`) s číslem 2.5 (typ `double`) a výsledek ukládáme do proměnné typu `double`. Za normálních okolností by nebylo možné provádět operaci součtu na dvou odlišných datových typech (`int` a `double`), ale právě díky `implicitní konverzi` se o převod proměnných na společný datový typ nemusíme starat a vše je vyřízeno za nás. Implicitní konverze dokonce probíhá i na samotném číslu 2.5, neboť není doplněno o žádnou z možných přípon (d - `double`, f - `float` nebo m - `decimal`). Výsledkem takovéto implictní konverze pak bude `double`, protože jakékoliv desetinné číslo bez přípony je převedeno právě na tento datový typ. Implicitní konverze jsou předem dány a tabulky těchto konverzí lze dohledat v dokumentaci jazyka C#.

V jazyce C# je dokonce možné psát i vlastní implicitní konverze. Touto problematikou se ale nebudeme v tomto materiále zabývat.

 # Explicitní konverze

`Explicitní konverze` se na rozdíl od té implicitní děje z naší vůle za použití kulatých závorek a námi požadavaného datového typu tak, jako je to vidno na ukázce níže:

Explicitní konverze

```
// Proměnná typu int
int i = 1;


// Explicitní konverze z intu do doublu
double d = (double)i; // d => 1.0d
```

> Explicitní konverze stejně jako ta implicitní musí být podporována daným datovým typem! Pokud se nám povede napsat konverzi, která není podporovaná, budeme na tuto skutečnost upozorněni chybovým hlášením.

 # Konverze za pomocí podpůrných tříd

Ke konverzi datových typů můžeme využít i tzv. `podpůrných tříd` a jejich metod. Třída a metoda jsou pro nás novým pojmem a budeme se jim podrobně věnovat v budoucích kapitolách. Prozatím si ukážeme, jak takováto konverze vypadá a přesné vysvětlení jednotlivých znaků ponecháme na později:

Podpůrné třídy

```
// Podpůrná třída Convert:
int i = Convert.ToInt32("45");
double d = Convert.ToDouble(11);

// Podpůrná třída Int32:
int i = Int32.Parse("14");
```

Tímto pochopitelně náš list podpůrných tříd nekončí. Další si ukážeme v budoucích kapitolách až se podrobněji seznámíme s metodami a třídami.

Shrnutí

Při založení (deklaraci) proměnné musíme vždy definovat i její datový typ, který je poté už neměnný. Datový typ určuje, jaký druh hodnoty můžeme do proměnné uložit. Zatím jsme si představili datové typy `int`, `string`, `double`, `float`, `decimal` a `char`. V další kapitole si představíme operátory a ukážeme si základní operace s proměnnými.

Operátory a výrazy

Úvod

S jedním ze základních operátorů už jsme se nevědomě setkali v předchozí kapitole. Byl to operátor přiřazení (`=`). V této kapitole se podíváme na další z nich a ukážeme si, jak s našimi proměnnými provádět některé operace.

Vymezení pojmu

Operátory jsou symboly, které specifikují, jakou z předdefinovaných operací provést na daném výrazu. Některé operátory jsou snadno čitelné, protože odpovídají operacím dobře známých z matematiky. Na začátku si představíme tzv. `binární operátory`.

Binární operátory

Mezi binární operátory řadíme `+` (sčítání), `-` (odčítání), `*` (násobení), `/` (dělení) a `%` (vyslovujeme „modulo“ - zbytek po celočíselném dělení). Opět si vše ukážeme na příkladu:

Binární operátory

```
// Pomocné proměnné
int a = 5;
int b = 2;

// Binární operace
int c = a + b; // => 7  (vrátí součet hodnot)
int d = a - b; // => 3  (vrátí rozdíl hodnot)
int e = a * b; // => 10 (vrátí součin hodnot)
int f = a / b; // => 2  (vrátí podíl hodnot, zbytek zahodí)
int g = a % b; // => 1  (vrátí zbytek po dělení)
```

Všechny tyto operace můžeme samozřejmě provádět i na jiných datových typech (pokud to umožňují, ale o tom více později) a dokonce i na různých datových typech mezi sebou. Musíme ale vždy dát pozor na to, aby výsledný datový typ výrazu korespondoval s datovým typem, do kterého se výsledek chystáme uložit. Příkladem budiž následující situace:

Podíl různých datových typů

```
// Pomocné proměnné
double a = 4.2;
int b = 2;

// Binární operace
double c = a / b; // => 2.1;
```

Z ukázky je patrné, že mezi sebou dělíme desetinné číslo typu `double` a celé číslo typu `int.` Protože je zde šance, že konečná hodnota toho výrazu bude opět desetinné číslo (a protože jeden z operandů je typu `double`), volíme pro proměnnou `c` opět datový typ `double`. Pokud bychom se i tak rozhodli výraz uložit do „intové“ proměnné, díky typové kontrole jazyka C# budeme upozorněni na naši chybu.

Rovněž se nyní nabízí otázka, co se stane, když mezi sebou budeme dělit například čísla typu `float` a `double`. Bude výsledkem `float` nebo `double`? Tato problematika je předmětem tzv. implicitních konverzí, které jsme probírali v předchozí kapitole.

Je také důležité si uvědomit, že ne všechny operace mohou být provedeny na všech datových typech. Dobrým příkladem je třeba datový typ `string`. Jednotlivé řetězce můžeme mezi sebou skládat pomocí operátoru `+`:

Podíl různých datových typů

```
// Pomocné proměnné
string jmeno = "Petr";
string prijmeni = "Novák";

// Spojení řetezců
string celeJmeno = jmeno + " " + prijmeni; // => Petr Novák
```

Použití ostatních binárních operátorů (`-`, `*`, `/` a `%`) by v tomto případě skončilo chybou. Pokud se nám podaří zapsat jakoukoliv neplatnou operaci, budeme vždy varování chybovým hlášením. Při zapsání více operací na ráz se postupuje stejným způsobem, který známe z matematiky. Násobení a dělení (popřípadě modulo) má přednost před sčítáním a odčítáním, neurčí-li kulaté závorky jinak. Pokud chceme použít více vnořených závorek v sobě, používáme pouze kulaté závorky. Hranaté, špičaté a složené závorky využijeme k jiným účelům. Pokud použijeme více operátorů stejné úrovně za sebou, postupuje se zleva doprava:

Kombinace operátorů

```
// Násobení, dělení a modulo má
// přednost před sčítáním a odčítáním
int a = 2 * 2 + 5; // => 4 + 5 = 9
```

```
// Kulaté závorky mohou toto
// pravidlo změnit
int b = 2 * (2 + 5); // => 2 * 7 = 14

// Více operátorů stejné úrovně –
// postupujeme zleva doprava
int c = 2 * 8 / 4; // => 16 / 4 = 4
```

Unární operátory

Unární operátory (česky jednočlenné) vyžadují pouze jeden operand (proměnnou) pro své fungování. Jedním takovým operátorem je i negace (obecný zápis `-x`, kde `x` je název proměnné):

Podíl různých datových typů

```
// Proměnná věk
int vek = 5;

// Negace proměnné věk
int negaceVek = -vek; // => -5
```

Mezi unární operátory patří celá řada dalších zástupců. My si pro jednoduchost a srozumitelnost zatím vystačíme pouze s tímto.

Operátory přiřazení

Jak bylo řečeno v úvodu, operátor přiřazení (`=`) je nám již známý. Pro jistotu jen znovu zopakuji, že tento operátor přiřadí pravou část výrazu do části levé. Musí však být vždy zachováno pravidlo, že výsledné datové typy obou částí jsou shodné. Zkušenější programátoři mi jistě odpustí, protože toto není díky dědičnosti a implicitním konverzím úplná pravda. Pro větší jednoduchost ale opět zamlčíme některá fakta.

Do této skupiny mimo jiné spadají i operátory složeného přiřazení, mezi které řadíme `+=`, `-=`, `*=`, `/=` a `%=`. Tyto operátory zjednodušují přiřazování hodnot do proměnných. Jejich užití si osvětlíme na následující ukázce:

Operátory složeného přiřazení (1)

```
// Pomocná proměnná
int a = 2;
```

```
// Operátory složeného přiřazení
a += 1; // => 3 (přičteme jedničku)
a -= 1; // => 2 (odečteme jedničku a vrátíme se na původní hodnotu)
a *= 4; // => 8
a /= 2; // => 4
a %= 3; // => 1
```

Složené operátory jsou kratší variantou pro klasický operátor přiřazení spojený s binární operací:

Operátory přiřazení (2)

```
// Pomocná proměnná
int a = 2;

// Operátory složeného přiřazení přepsané na
// své ekvivalenty pomocí operátoru přiřazení
// a binárního operátoru
a = a + 1; // => 3
a = a - 1; // => 2
a = a * 4; // => 8
a = a / 2; // => 4
a = a % 3; // => 1
```

Důležitost typového systému

V minulé kapitole jsme si ukázali, že číslici `10` lze jednou uložit jako textový řetězec a poté i jako číslo. Pro připomenutí znovu uvedu příklad:

Způsoby uložení

```
// Číslice 10 jako číslo
int a = 10;

// Číslice 10 jako text
string b = "10";
```

S naší novou znalostí binárního operátoru `+` si nyní můžeme ukázat, jak datový typ kompletně změní výsledek operace součtu:

Podíl různých datových typů

```
// Číslice 10 jako číslo
int a = 10;

// Číslice 10 jako text
string b = "10";

// Součet "číslic"
int x = a + a; // => 20
string y = b + b; // => "1010"
```

Toto je pouze jeden příklad z mnoha. V budoucnosti zjistíme, že datové typy budou ovlivňovat téměř každou část našeho kódu.

Shrnutí

V této kapitole jsme se seznámili s operátory. Pro naše další účely nebude důležité znát jejich přesné rozdělení, ale pouze znát jejich význam a reálné využití.

Metody

Úvod

V minulých kapitolách jsme se seznámili s proměnnými a se základními operacemi, které s nimi můžeme provádět. V této kapitole si představíme pojmy `třída` a `metoda`.

Metody

Metoda je úsek kódu, který můžeme opakovaně spouštět. Jazyk C# (přesněji framework .NET) v základu disponuje velkým množstvím metod, které Microsoft předpřipravil pro usnadnění naší práce. Jako první z nich si ukážeme metodu `Console.Beep()`, na které si budeme demonstrovat základy nové látky:

Metoda

```
Console.Beep();
```

Po spuštění (odborně `zavolání`) této metody se ozve krátký zvuk. Často se budeme setkávat s tím, že volaná metoda bude přijímat jeden nebo více tzv. `parametrů`. To je i případem naší metody:

Metoda s parametry (1)

```
// Pípnutí o frekvenci 450Hz a
// délce 100 milisekund
Console.Beep(450, 100);
```

Metoda s parametry (2)

```
// Proměnné
int frekvence = 450;
int delka = 100;

// Pípnutí o frekvenci 450Hz a
// délce 100 milisekund
Console.Beep(frekvence, delka);

// Argumenty ve spojení s operátory
Console.Beep(100 * 2, (100 % 60) / 2);
```

Metoda umožňuje zadat celkem dva parametry typu `int` – frekvenci a délku pípnutí. Konkrétní předaná čísla `450` a `100` jsou v tomto případě tzv. `argumenty` naší metody, které píšeme vždy do kulatých závorek. Pojmy parametr a argument jsou velmi často nesprávně zaměňovány. Pokud metoda přejímá více parametrů, jednotlivé argumenty pak vždy oddělujeme čárkou (`,`). Argumenty můžeme do metod pochopitelně předávat i formou proměnných (ukázka 2). Musíme pouze dohlédnout na to, aby nám korespondovali datové typy mezi předávanou proměnnou a mezi odpovídajícím parametrem metody.

> Při volání metod záleží na pořadí argumentů! – První argument odpovídá prvnímu parametru, druhý argument odpovídá druhému parametru a tak dále.

Některé parametry mohou být `nepovinné`. Nepovinné parametry nalezneme vždy až na samotném konci všech parametrů. Nestane se tedy, že by za nepovinným parametrem následoval parametr povinný.

> Pro úplnost ještě uvedu, že výchozí hodnoty parametrů metody `Console.Beep()` jsou 800 Hz a 200 ms.

Častěji se ale budeme setkávat s metodami, jejichž parametry budou povinné a jejich vynechání bude znamenat chybové hlášení. Narazit můžeme dokonce i na kombinaci obou výše zmíněných případů. Celá situace ohledně parametrů se ještě navíc komplikuje tím, že jedna metoda může mít různé kombinace parametrů, které je schopna pojmout. O takovýchto metodách říkáme, že jsou tzv. `přetížené` (anglicky overloaded methods). Dobrým příkladem může být naše výše zmíněná metoda `Console.Beep()`:

Přetížená metoda

```
// První overload
Console.Beep();


// Druhý overload
Console.Beep(450, 100);
```

Ještě doplním, že není možné zavolat metodu pouze s jedním argumentem nesoucí frekvenci nebo délků trvání. Bylo by to možné pouze za podmínky, že by existoval třetí overload metody, který by vyžadoval pouze jeden parametr. Při volání metod musíme tedy vždy respektovat jednotlivé variace metod (overloady) a pořadí jejich parametrů.

Setkat se můžeme i s `pojmenovanými argumenty`. Ty nám výrazně usnadní práci v případech, kdy si nepamatujeme přesné pořadí argumentů nebo chceme zvýšit přehlednost našeho kódu:

Pojmenované argumenty (1)

```
// Pojmenované argumenty
Console.Beep(frequency: 450, duration: 100);
```

Další výhodou tohoto zápisu je, že pořadí argumentů pak můžeme i prohazovat:

Pojmenované argumenty (2)

```
// V případě pojmenovaných argumentů nemusíme
// dodržovat přesné pořadí parametrů. Argumenty
// můžeme libovolně prohazovat.
Console.Beep(duration: 100, frequency: 450);
```

Pojmenované argumenty jsou validní pouze pokud nejsou následovány argumenty klasickými (`pozičními`) a od C# 7.2 pokud jsou poziční argumenty použity ve správném pořadí:

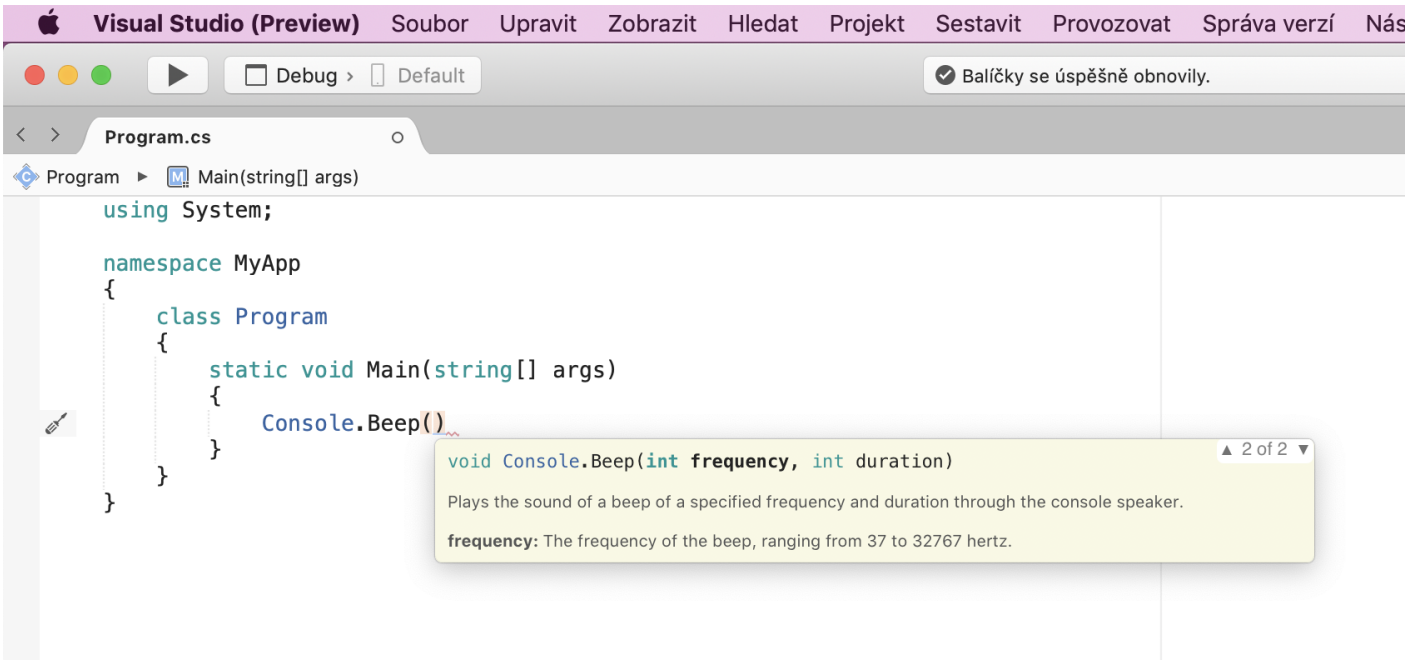
Pojmenované argumenty (2)

```
// Validní kód. Pojmenovaný argument není
// následován argumentem pozičním.
Console.Beep(450, duration: 100);

// Validní kód pro C# verze 7.2 a vyšší.
// Pojmenovaný argument je sice následován
// argumentem pozičním, ale byla zachována
// správná pozice vzhledem k pořadí
// parametrů v metodě.
Console.Beep(frequency: 450, 100);

// Chybný kód. Za pojmenovaným argumentem
// se nachází argument poziční. Nebyla
// dodržena ani správná pozice druhého
// argumentu - frekvenci píšeme na první
// pozici.
Console.Beep(duration: 100, 450);
```

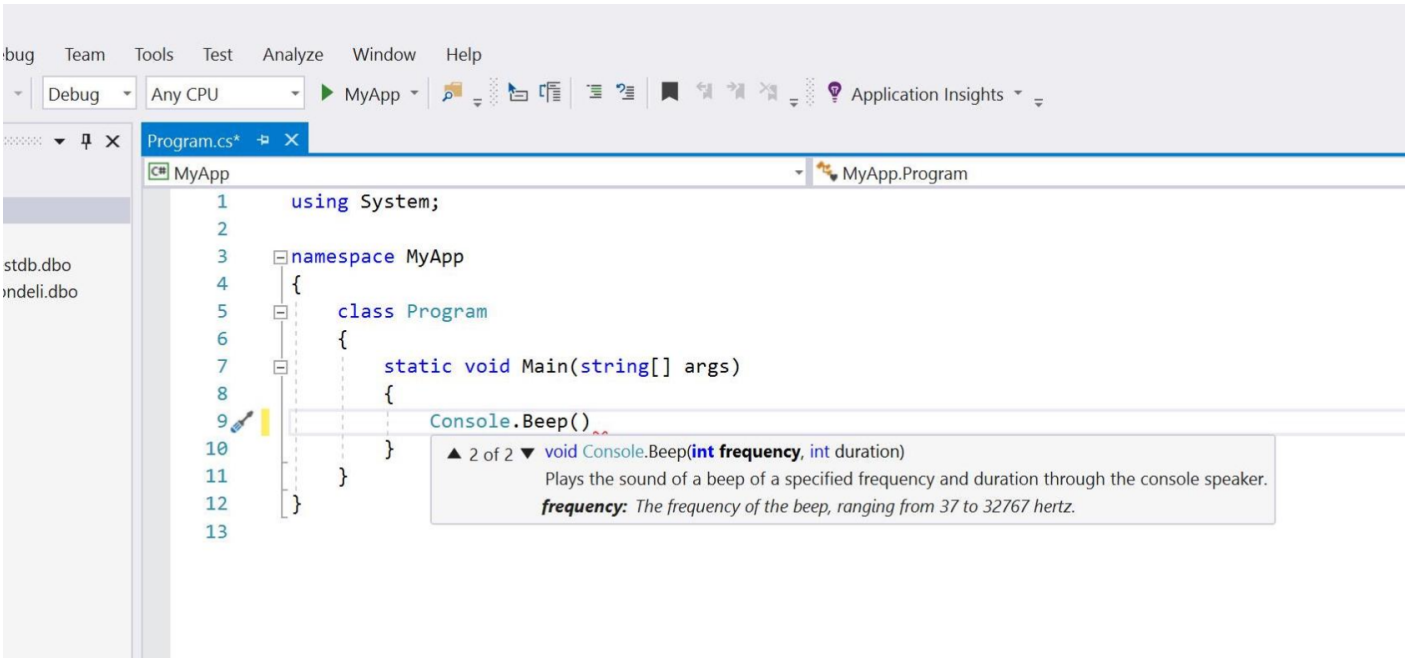
> Pojmenované argumenty jsou spíše rozšiřující látkou a běžně se s nimi setkávat nebudeme.

Jak jsme se nyní sami přesvědčili, situace kolem metod a jejich parametrů může být občas velmi komplikovaná. Úkolem programátora pochopitelně není znát zpaměti přesné pořadí a možné kombinace všech parametrů. Naši práci většinou výrazně zjednoduší vývojové prostředí, které nám všemožně napovídá a urychluje tedy proces celého vývoje. Souhrnu takovýchto chytrých funkcí říkáme obecně `intelligent code completion` (inteligentní doplňování kódu). Vývojové prostředí

Visual Studio má vlastní implementaci tohoto systému nazvanou `IntelliSense`, na kterou se nyní blíže podíváme. IntelliSense za nás chytře dokončuje slova nebo zobrazuje pomocné nabídky. Kdykoliv například dopíšeme kulaté závorky za konec metody, zobrazí se nám list se všemi možnými overloady dané metody a výčet jejich parametrů:



Ukázka IntelliSense nabídky ve Visual Studio for Mac.



Ukázka IntelliSense nabídky ve standardním Visual Studiu pro Windows.

Kromě stručného popisu metody nám vyskakovací list ještě napovídá jednotlivé parametry a jejich význam. Šipkami se pak můžeme pohybovat mezi jednotlivými overloady metody. Pokud bychom narazili na nepovinný parametr, bude za jeho datovým typem a názvem ještě operátor přiřazení (`=`) a jeho výchozí hodnota. Jedinou neznámou je pro nás slovo `void` na samém začátku listu. Jedná se o tzv. `návratový typ` metody. Tento konkrétní návratový typ značí, že po zavolání metody

nečekáme nazpět žádný výsledek. Metoda se prostě provede a pokračuje se na další řádek. Některé metody ale po svém zavolání mohou vrátit hodnotu, kterou můžeme poté odchytit a uložit do proměnné. Vše si ukážeme na příkladu:

Návratová hodnota metody

```
int x = Math.Abs(-27); // x = 27
```

Jak název napovídá, metoda `Math.Abs()` vrátí absolutní hodnotu předaného čísla a uloží jej do proměnné `x`. Návratová hodnota je pak v tomto případě typu `int`. Pro upřesnění ještě dodám, že metoda může vracet jakýkoliv datový typ. Záleží pouze na konkrétní metodě a její implementaci. Pokud metoda žádnou hodnotu nevrací, návratovým typem je pak vždy `void`.

> Později si ukážeme, jak napsat naší vlastní metodu.

Jako další si nyní ukážeme metodu `Console.WriteLine()`. Tato metoda přijímá parametr typu `string`, jehož obsah pak vypíše na nový řádek konzole. Pokud bychom zavolali bezparametrický overload této metody, vypíšeme jednoduše prázdný řádek a další budoucí výpis se bude provádět až po mezeře (prázdném řádku):

Nový řádek

```
// Bezparametrická verze metody - vypíše
// prázdný řádek
Console.WriteLine();

// Parametrická verze metody. Vypíše předaný
// řetězec
Console.WriteLine("Ahoj světe!");
```

Pokud bychom se nyní pokusili náš kód otestovat, okno konzole se zavře téměř okamžitě po spuštění. Důvodem je, že po vypsání našeho řetězce program končí a okno konzole se tedy zavře. Dodatečným voláním metody `Console.ReadKey()` zajistíme, že po vypsání řetězce bude program ještě čekat na stisknutí libovolné klávesy. Tím zabráníme samovolnému zavření okna. Náš program tedy nyní vypadá nějak takto:

Program

```
using System;
□□□□□
public class Program
{
□public static void Main()
□{
```

```
        // Vypíše řetězec do konzole
□□Console.WriteLine("Ahoj světe!");

□□// Počká na stisk libovolné klávesy
□□Console.ReadKey();
□}
}
```

Rovněž se někdy můžete setkat i s voláním metody `Console.ReadLine()`. Ta uživateli umožní napsat celý řetězec (na místo jednoho znaku) a samotné potvrzení je provedeno klávesou enter. Uživatelem zadaný řetězec je pak vrácen jako návratová hodnota typu `string` a může být poté uložena do proměnné. Pokud vrácenou hodnotu neuložíme, nic závažného se nestane a uživatelem zadaný řetězec je pak zahozen. Pro úplnost ještě zmíním existenci metody `Console.Write()`, která předaný řetězec píše na stejný řádek, jako předchozí výpis. Na malé ukázce si nyní budeme demonstrovat použití metody `Console.ReadLine()`:

Vstup od uživatele

```
// Získá textový řetězec od uživatele
string vstup = Console.ReadLine();
```

Takovýto kus kódu počká, až uživatel zadá textový řetězec a po stisknutí klávesy enter se uživatelem zadaný řetězec uloží do proměnné vstup a program poté dále pokračuje.

> Metoda `Console.ReadLine()` **VŽDY** vrací datový typ string a to i v případech, kdy uživatelem zadaný řetězec je pouze číslo (např. `82`). Jakákoliv úprava vstupu, validace (ověření správnosti) nebo přetypování je pak prací programátora.

Vlastní metody

V jazyce C# můžeme psát i naše vlastní metody. To se velmi hodí v situacích kdy víme, že daný úsek kódu budeme používat na více místech. Metody pro nás budou nástrojem, jak minimalizovat duplicity v kódu a dekomponovat (rozložit) logiku naší aplikace do více snadněji spravovatelných celků. Při zakládání nové metody kromě názvu vždy určujeme i její parametry a návratovou hodnotu. Vše si ukážeme na následujícím příkladu:

Deklarace proměnné

```
using System;


namespace MojeAplikace
{
```

```
    class Program
    {
        // Hlavní metoda Main. Je automaticky spuštěna
        // po startu programu.
        public static void Main(string[] args)
        {
            // Volání naší nové metody
            Pozdrav();
        }

        // Naše nová metoda
        public static void Pozdrav()
        {
            Console.WriteLine("Ahoj!");
        }
    }
}
```

Metody vždy píšeme na úroveň třídy. V našem případě budou metody vždy definovány ve třídě `Program`, neboť jsme si ještě neukázali, jak zakládat třídy jiné. Při definici metody začínáme tzv. modifikátorem přístupu `public`, který označí metodu jako veřejnou. Taková metoda je pak viditelná mimo její rodičovskou třídu (v našem případě třídu `Program`). Opačným modifikátorem je pak `private`, který naopak zařídí, že naše metoda mimo třídu nebude viditelná a půjde zavolat jen v té dané třídě, ve které se nachází. Modifikátory přístupu se nyní nebudeme blíže zabývat a naše metody vždy budeme označovat jako `public`, než si v pozdějších kapitolách tuto problematiku rozebreme více do podrobna. Obdobně budeme postupovat i u modifikátoru `static`, jenž bude předmětem poslední kapitoly. Oba modifikátory tedy budeme brát jako nutné „zlo“, než si v pravý čas osvětlíme jejich význam. Dále pak následuje návratová hodnota metody. Ta může být buď typu `void` (metoda nevrací žádný výsledek viz příklad výše) nebo může vracet libovolný datový typ (například `int` viz ukázka níže). Dále pak uvádíme název metody a kulaté závorky, jejichž význam si za chvíli osvětlíme. Metoda pak vždy končí složenými závorkami, jež definují její tělo a do kterých píšeme samotnou funkčnost naší metody. Nyní si ještě ukážeme metodu s parametry a návratovou hodnotou:

Deklarace proměnné

```
using System;

namespace MojeAplikace
{
    class Program
    {
        // Hlavní metoda Main. Je automaticky spuštěna
```

```
        // po startu programu.
        public static void Main(string[] args)
        {
            // Volání naší nové metody
            Pozdrav();
        }

        // Naše nová metoda
        public static void Pozdrav()
        {
            Console.WriteLine("Ahoj!");
        }

        // Metoda s parametry a návratovou hodnotou
        public static int Secti(int a, int b)
        {
            return a + b;
        }
    }
}
```

Na této ukázce jsme naše řešení ještě doplnili o metodu se jménem `Secti`, která jako dva své parametry očekává čísla typu `int` a za pomocí klíčového slova `return` vrací jejich součet. Klíčové slovo return ukončí činnost metody a vrátí hodnotu výrazu, který jsme za toto slovo dopsali (v našem případě součet proměnných `a` + `b`). V praxi to znamená, že jakýkoliv další kód po tomto klíčovém slově již nebude vykonán. Pokud jsme metodě definovali návratovou hodnotu jinou než `void` (tzn. očekáváme od metody nazpět nějaký výsledek), musíme v těle metody vždy klíčové slovo `return` použít. Samotné parametry metody pak vždy oddělujeme čárkou a nezapomínáme určit jejich datový typ. Parametry pak v těle metody vystupují jako obyčejné proměnné, s kterými můžeme libovolně manipulovat. Na závěr už jen doplním, že nezáleží na pořadí metod uvnitř třídy (`class`). Metody mohou být volány v jiných metodách nezávisle na svém pořadí.

> V případě všech datových typů, které jsme se zatím naučili (tzv. primitivních datových typů) platí pravidlo, že veškeré změny provedené na parametrech metody se neprojeví mimo metodu samotnou. Vše lépe osvětlí následující příklad:

Deklarace proměnné

```
using System;

namespace MojeAplikace
{
```

```
    class Program
    {
        public static void Main(string[] args)
        {
            // Naše pomocná proměnná
            int x = 0;

            // Volání metody, jež upravuje obsah proměnné
            Metoda(x);

            // Vypíšeme obsah proměnné
            Console.WriteLine(x); // => 0
        }

        public static void Metoda(int x)
        {
            // Změníme hodnotu předané proměnné
            x = 1;

            // Vypíšeme hodnotu proměnné
            Console.WriteLine(x) // => 1
        }
    }
}
```

Tento kód by po spuštění nejdříve vypsal `1` a pak `0`. Nejdříve by se tedy provedl výpis v těle metody `Metoda`, kde hodnota `x` je po změně z nuly rovna jedné a poté výpis v těle metody `Main`, kde hodnota proměnné `x` je stále `0`, neboť do těla metody je předána pouze samotná hodnota proměnné a ne odkaz (reference) na proměnnou samotnou. Tato problematika je předmětem primitivních a referenčních datových typů a budeme se ji zabývat podrobněji v poslední kapitole.

Setkat se můžeme i se situací, kdy metoda volá sama sebe. Jedná se o tzv. rekurzi a existují případy, kdy je toto chování velmi užitečné. Rekurze ale musí být vždy nějak ukončena, neboť by v opačném případě vznikl nekonečný cyklus, jež by vedl k přetečení zásobníku a pádu programu. V programování tuto chybu nejčastěji označujeme jako stack overflow a byla po ní pojmenována i velmi oblíbená a známá webová stránka pro vývojáře.

Třídy

Než prozatím uzavřeme kapitolu metod, uvedeme si ještě pár teoretických faktů a rozšíříme si naše dosavadní portfolio operátorů.

Pozornější čtenáři si mohli povšimnout, že samotný název metody (např. `Beep` nebo `WriteLine`) je kromě kulatých závorek doplněn i o tečku (`.`), které předchází vždy ještě nějaký název (v našem případě `Console` nebo třeba `Math`). Odborně hovoříme o tzv. `třídách` a naše tečka je pro nás novým operátorem sloužícím pro přístup ke členu. V třídách sdružujeme metody, které spolu logicky souvisí. Třída `Math` například slouží jako kontejner pro všechny metody souvysejícími s matematickými procedurami. Třída `Console` pak zaštiťuje všechny důležité nástroje pro manipulaci s konzolí. Například tedy výstup (výpis na konzoli) nebo získávání vstupu od uživatele. Operátor členu (`.`) pak slouží jako vstupní brána, která nám po svém napsání odhalí obsah dané třídy. Třída kromě metod může obsahovat i další členy, jako třeba nám dobře známé proměnné. Dále pak i vlastnosti nebo události, o kterých se budeme bavit v budoucích kapitolách. Setkat se můžeme dokonce i s třídou, která uvnitř sebe definuje další třídu (poté hovoříme o tzv. nested class, kterými se nebudeme v tomto materiále zabývat). Na těchto členech můžeme poté opět volat operátor členu (`.`) a prohlížet si prostřednictvím IntelliSense nabídek obsah daného členu ještě více do hloubky. Prozatím si ale vystačíme pouze se znalostí zavolat z dané třídy námi požadovanou metodu, než se blíže seznámíme z dalšími členy tříd.

Na závěr už jen zmíním, že v jazyce C# mají třídy a metody vždy velké počáteční písmeno a stále je dodržována camel case syntaxe – jednotlivá slova tedy oddělujeme velkým počátečním písmenem na místo mezery. Díky počátečnímu velkému písmenu pak můžeme snadno odlišit název třídy od názvu proměnné. Metodu pak vždy bezpečně poznáme podle kulatých závorek.

Shrnutí

Metody jsou jedním ze základních stavebních kamenů naší aplikace. Při práci s metodami musíme dát pozor na důsledné dodržování pořadí jednotlivých parametrů. Konkrétní hodnoty dosazované za tyto parametry pak nazýváme argumenty. Nezapomínejme na to, že metody rovněž mohou po svém zavolání vracet návratovou hodnotu, která může (ale nemusí) být odchycena a uložena do proměnné.

Pole a listy

Úvod

V této kapitole se budeme věnovat složitějším datovým strukturám `pole` (array) a `List`, které slouží pro vytvoření seznamu hodnot, přístupných v jedné proměnné pod určitými indexy.

Co je to pole a k čemu se používá?

`Pole` (array) je datová struktura umožňující udržování hodnot stejného datového typu pohromadě a ulehčující práci s daty. Pro ukázku praktičnosti si představme úlohu, kdy máme skupinu lidí, která má předem známý počet osob. Chtěli bychom si u každé osoby zaznamenat, jaký má měsíční příjem a poté zjistit průměrný příjem v této skupině. K této úloze se perfektně hodí právě pole. Pro jednoduchost předpokládejme, že všechny příjmy jsou celočíselné a naše skupina lidí obsahuje 5 osob.

Příklad

```
// Založení nového pole o délce 5
int[] prijmy = new int[5];

// Naplnění pole hodnotami
prijmy[0] = 20000;
prijmy[1] = 15000;
prijmy[2] = 25000;
prijmy[3] = 30000;
prijmy[4] = 12000;

// Výpočet průměru
double prumer = (prijmy[0] + prijmy[1] + prijmy[2] + prijmy[3] + prijmy[4]) / prijmy.Length;

// Výpis průměru
Console.WriteLine(prumer);
```

Výstupem tohoto kódu bude `20 400`, což je náš hledaný průměrný příjem. Pojďme si nyní projít náš kód. Pole deklarujeme pomocí hranatých závorek za datovým typem (v našem případě `int[]`). Dále následuje název samotného pole (`prijmy`) a poté operátor přiřazení `=`. Nakonec uvádíme klíčové slovo `new` následované znovu zvoleným datovým typem s hranatými závorkami, ve kterých

je maximální počet prvků, které chceme do pole uložit.

K jednotlivým prvkům v poli pak přistupujeme pomocí tzv. `indexů`. Pro pole o `n` prvcích jsou to celá čísla, počínající `0` a končící hodnotou `n-1`. Pro `5` prvků jsou to tedy čísla `0`, `1`, `2`, `3` a `4`. Pro výběr prvku na určité pozici slouží hranaté závorky, uvnitř kterých uvádíme samotný index. Pro přiřazení na zvolený index slouží operátor přiřazení (`=`). Pro přiřazení hodnoty `20 000` na první pozici (index `0`) tedy použijeme příkaz `prijmy[0] = 20000`. Pro vypsání této hodnoty použijeme `Console.WriteLine(prijmy[0])`.

Poslední část kódu by měla být poměrně jasná, ale pro jistotu si projdeme ještě tu. Ukládáme zde do proměnné prumer typu `double` (jelikož průměrný příjem by mohl vyjít jako desetinné číslo) součet všech prvků z pole vydělený počtem prvků. To je obecně známý vzorec pro výpočet průměru. Zajímavá část tohoto úseku je `prijmy.Length`. `Length` je vlastnost, kterou obsahuje každé pole a její návratovou hodnotou je počet prvků konkrétního pole.

> V jazyce C# mají objekty a třídy své členy přístupné pod tečkou. Člen `Length` je tzv. vlastností pole. Vypisovat všechny tyto vlastnosti by však bylo na velmi dlouho a není to nutné, jelikož ve vývojovém prostředí je k dispozici intellisense, která vám vždy nabídne všechny tyto členy k výběru. Co je to objekt, třída nebo vlastnost nyní ponecháme stranou, protože se jimi budeme podrobněji zabývat v dalších kapitolách.

> V tomto příkladu bychom mohli použít místo vlastnosti `Length` rovnou číslo `5`. V rozsáhlejších aplikacích je však vhodné použít zmíněnou vlastnost, protože pokud bychom chtěli později spočítat průměr například pro 6 osob, ubývá nám místo, kde musíme modifikovat kód a zlepšuje se nám tedy přehlednost a udržitelnost našeho řešení.

Inicializace pole

Zapsáním `int[] array = new int[5]`; se vytvoří pole se jménem `array` o pěti prvcích, jak jsme již zmiňovali výše. Co jsme však nezmínili je, že všechny tyto prvky jsou implicitně nastaveny na určitou defaultní hodnotu. Například pro `int` je to 0, pro `string` je to prázdný řetězec (`""`). Pole lze však také inicializovat přímo jeho hodnotami. Pojďme se podívat, jak by to vypadalo pro náš předchozí příklad:

Příklad

```
// Inicializace pole s přednastavenými hodnotami
int[] prijmy = new int[] { 20000, 15000, 25000, 30000, 12000 };


// Výpočet průměru podle vzorce
double prumer = (prijmy[0] + prijmy[1] + prijmy[2] + prijmy[3] + prijmy[4]) / prijmy.Length;
```

```
// Výpis průměru
Console.WriteLine(prumer);
```

Tento zápis je o poznání kratší, avšak někomu by mohl přijít méně přehledný. Výsledek je však stejný, jako v první příkladu a je tedy na čtenáři, který způsob zvolí.

> Povšimněte si, že v druhém příkladu již ze závorek zmizel počet prvků. Kompilátor si již sám zjistí, pro kolik prvků má alokovat místo podle počtu čísel ve složených závorkách.

List

V této kapitole bychom se však měli také dozvědět, co je to List. Představte si jej, jako takové chytřejší pole. Hlavním rozdílem mezi těmito strukturami je to, že pro pole je nutné znát počet prvků již při inicializaci. Pro list ovšem nikoli. Další zásadní rozdíl je, že list je narozdíl od pole `generická datová struktura`. Co to přesně znamená teď není důležité. Nyní je pro nás důležité si pamatovat, že u generických datových struktur píšeme vždy za název struktury špičaté závorky (`<>`) a do nich datový typ prvků uložených v takové struktuře. Dalšími rozdíly jsou odlišné přidávání a mazání prvků. Do pole není možné přidávat další prvky, ani žádné mazat. Jediný způsob, jak nějaký prvek smazat je vynulovat jej. To však nemusí být vždy ideální. Pojďme se teď tedy ještě jednou vrátit k našemu příkladu a přepsat jej s použitím datové struktury `List<>`:

Příklad

```
// Založení nového listu. Důležitý je datový typ int ve
// špičatých závorkách kterým říkáme, že náš list bude
// obsahovat pouze číselné hodnoty. Kombinace více
// datových typů není v jazyce C# možná.
List<int> prijmy = new List<int>();

// Přidávání a odebírání hodnot listu. V případě,
// že list obsahuje více shodných hodnot, které se snažíme
// odebrat, je vždy odstraněn prvek s větším indexem (naposledy
// přidaný shodný prvek)
prijmy.Add(20000);
prijmy.Add(15000);
prijmy.Add(25000);
prijmy.Add(30000);
prijmy.Add(12000);
prijmy.Add(40000);
prijmy.Remove(40000);
prijmy.Add(40000);
```

```
prijmy.RemoveAt(5);

// Výpočet průměru. K hodnotám v listu
// můžeme přistupovat jako k prvkům v poli
// za pomocí hranatých závorek
double prumer = (prijmy[0] + prijmy[1] + prijmy[2] + prijmy[3] + prijmy[4])/prijmy.Count;

// Výpis výsledku
Console.WriteLine(prumer);
```

> Pro práci s listy je nutné nainportovat si knihovnu `System.Collections.Generic`! Knihovna může být naimportována ručně vložením using-u nad namespace nebo automaticky za použití vývojového prostředí (např. Visual Studio), které by vám mělo samo automaticky using nabídnout.

Import knihovny

```
using System;
using System.Collections.Generic; // Důležité!
```

Tento kód nám vrátí opět stejný výsledek `20 400`. Pro demonstraci mazání jsme zde dvakrát přidali další hodnotu, kterou jsme následně smazali. Jak si můžete povšimnout, pro přidávání slouží metoda `Add()`. Pro mazání `Remove()`, nebo `RemoveAt()`. Metoda `Remove()` se chová jinak, než při přístupu k prvku v poli přes index. Zde místo indexu používáme přímo hodnotu, kterou chceme odstranit. `RemoveAt()` se na druhé straně chová stejně, jako bychom k prvku přistupovali v poli přes index. Prvky v Listu jsou opět indexovány od `0` do `n-1`, jako tomu bylo v poli. Pokud však odstraníme prvek například na indexu 2, všechny další prvky se přesunou. Pokud tedy máme prvky s indexy 0, 1, 2, 3, 4 a odstraníme prvek na indexu 2, budeme mít indexy 0, 1, 2, 3. Přístup k datům a jejich změna je stejná jako u pole. Tedy za použití hranatých závorek (`[]`) a indexu prvku.

Na závěr si povšimněte, že při inicializaci Listu zapisujeme `new List<int>();`! Kulaté závorky na konci jsou nezbytné. Narozdíl od pole se zde totiž volá konstruktor třídy `List`. Co je to konstruktor si vysvětlíme až v druhém ročníku. Nyní nám stačí vědět, že při inicializaci proměnných některých datových typů je potřeba napsat nakonec kulaté závorky (`()`), a že `List` je jedním z nich.

> Všimněte si také, že zde není nutné znát předem počet osob v naší skupině. Pokud bychom tento počet tedy neznali, použili bychom právě list.

> Všimněte si také, že v této ukázce se nám metoda `prijmy.Length` změnila na `prijmy.Count`. Je to proto, že List nemá vlastnost `Length`, ale má vlastnost `Count`, která vrací stejnou hodnotu jako `Length` pro pole - celkový počet prvků v listu.

Referenční a primitivní datové typy

V předchozích podkapitolách jsme si představili pole a list a ukázali jsme si, jak s nimi můžeme pracovat. V této kapitole bych rád poukázal na častý problém začátečníků, jež plyne z neznalosti problematiky referenčních a primitivních datových typů. Veškeré datové typy, které jsme si do teď ukazovali (např. `int` nebo `char`) byli tzv. primitivní. Oba naše nové datové typy (list a pole) jsou zástupci tzv. referenčních datových typů. Rozdíl mezi těmito dvěma skupinami si budeme demonstrovat na malé ukázce:

Primitivní a referenční typy

```
// Pomocné pole
int[] suda = {2, 4, 6};
int[] licha = {1, 3, 7};

// Přemažeme lichá čísla těmi sudými (nelogičnost tohoto kroku nyní ponechme stranou)
licha = sude;

// Smažeme všechna sudá čísla
Array.Clear(sude, 0, suda.Length);

// Vypíšeme první prvek lichých čísel
Console.WriteLine(licha[0]);
```

Co by bylo výstupem takovéhoto programu? Na posledním řádku vypisujeme první prvek pole `licha`, jež jsme na řádku 6 přepsali polem sudých čísel. Nabízí se tedy odpověď, že výstupem tohoto programu bude číslo `2`, jelikož dvojka v době vykonávání řádku 6 byla prvním prvkem pole `suda`, než později došlo k jeho úplnému promazání. Kdybychom se však podívali na obsah pole licha, zjistíme, že je úplně prázdné. Ačkoliv jsme tedy na řádku 9 pročistili pouze pole `suda`, nějakým způsobem došlo i k promazání pole `licha`. Proč?

Odpověď na tuto otázku bychom nalezli na řádku 6, kde ve skutečnosti nepřepisujeme pole licha (1, 3, 7) polem sude (2, 4, 6) nýbrž tímto zápisem říkáme, že proměnná `licha` bude nyní odkazovat na proměnnou `suda`. Došlo tedy k předání tzv. `reference` (odkazu) na místo `hodnoty`, jak jsme zvyklí u primitivních datových typů. To je také důvod, proč někdy těmto typům říkáme odkazové a hodnotové.

> Problematika primitivních a referenčních datových typů může být pro začátečníka velmi zavádějící a budeme se jí více věnovat v poslední kapitole.

>

> Pokud bychom chtěli doopravdy nahradit obsah pole licha obsahem pole sude (na místo pouhého předání odkazu / reference), můžeme k tomuto účelu využít metodu `CopyTo` - `sude.CopyTo(licha , 0)`;

Shrnutí

V této kapitole jsme si ukázali list a pole. Tyto konstrukty jsou nejužitečnější v kombinaci s tzv. cykly, které si ukážeme v příští kapitole. Níže jsou pak pro zopakování užitečné příkazy pro práci s poli a listy:

Užitečné ukázky

```
// Základní inicializace pole o 10 prvcích:
int[] pole = new int[9];

// Inicializace pole i s jeho hodnotami:
int[] pole = { 3, 7, 69 };

// Získání hodnoty z pole:
int prvek = pole[0];

// Nastavení hodnoty na libovolném indexu:
pole[0] = 32;

// Inicializace listu:
List<int> list = new List<int>();

// Přístup k prvkům listu probíhá stejně jako u pole:
int prvek = list[0];

// Přidání hodnoty do listu:
list.Add(32)

// Odebrání hodnoty z listu:
list.Remove(11);
```

Řízení toku programu

Úvod

V této kapitole si představíme nástroje pro řízení toku programu. K tomu slouží v programovacích jazycích `podmínky` a `cykly`.

O čem je vlastně řeč?

Pojmem řízení toku programu se rozumí větvení programu. O tuto funkci se starají konstrukty `if`, else, `switch-case` a cykly `for`, `foreach`, `while` a `do-while`. Až do této chvíle jsme mohli vytvářet jen velice jednoduché a přímočaré programy. Neměli jsme totiž k dispozici nástroje pro provedení určitého fragmentu kódu jen za určité podmínky a když jsme chtěli provést nějaký kus kódu vícekrát, museli jsme jej reálně napsat několikrát pod sebou. To přirozeně snižuje přehlednost kódu a prakticky znemožňuje jeho udržitelnost. To by se však po této kapitole mělo změnit. Jistě si vzpomenete na náš příklad z kapitoly o polích. Byl to velmi jednoduchý příklad, avšak i ten se dal zpracovat lépe. Náš program nebyl příliš univerzální a nepracoval se vstupy zadanými uživatelem. I takové programy je možné vytvářet, avšak jejich použití je velmi omezené a to alespoň do doby, než se všichni naučí programovat.

Pojďme si nyní představit naše podpůrné funkce, jejich používání, a nakonec si náš předchozí kód rozšířit za použití těchto funkcí. Než s tím vším ale začneme, představíme si ještě jeden důležitý datový typ.

Datový typ bool

Datový typ `bool` slouží k uchovávání informace o pravdě či nepravdě. Může nabývat hodnot `true` (pravda) nebo `false` (lež / nepravda). Vše si ukážeme na jednoduchém příkladě:

Deklarace boolovské proměnné

```
// Deklarace boolovské proměnné
bool x; // => x = false

// Přiřazení hodnoty (lež)
x = false;

// Přiřazení hodnoty (pravda)
```

```
x = true;
```

Hodnoty `true` a `false` jsou zde tzv. klíčovými slovy podobně jako třeba `int` nebo `class`. Klíčová slova jsou termíny rezervované jazykem a nelze je tedy například použít pro název proměnné. Mezi hodnotami typu `bool` můžeme provádět i operace. Ukážeme si tedy hned několik nových operátorů:

Nové operátory

```
// Pomocné proměnné
bool x = true;
bool y = false;

// Operátor porovnání
x == y // => false

// Operátor nerovnosti
x != y // => true
```

> Operátor porovnání (`==`) se začátečníkům často plete s operátorem přiřazení (`=`). Jedním rovnítkem tedy přiřazujeme a dvěma porovnáváme!

Podmínky

Pro vytváření podmínek nám slouží dvě struktury. První z nich je `if` (případně `else if`) a druhá je `else`. Vše si opět ukážeme na příkladu:

Konstrukt if

```
// Pomocné proměnné
bool y = true;

// Je proměnná y pravdivá ?
if (y == true)
{
    // Ano je - vykoná se tělo podmínky
    Console.WriteLine("Podmínka byla naplněna. ");
}

// Program pokračuje dále
Console.WriteLine('Jedeme dál.')
```

Konstrukt if (krátká verze)

```
// V případě, že je výsledkem logického výrazu bool můžeme vynechat operátory porovnání. Je totiž zbytečné se ptát, zda je pravda opravdu pravda nebo zda nepravda je pravda :-)
if (y)
{
    Console.WriteLine("Proměnná y drží hodnotu true.");
}
```

V kulatých závorkách je očekáván logický výraz, který je možné vyhodnotit jako `true` (pravda, podmínka je splněna) nebo `false` (lež, podmínka není splněna). Příklady takovýchto výrazů mohou být: `a == b`, `a < 10` nebo například `a == "muž"`. Tyto výrazy lze také skládat za použití logických spojek `&&` (AND / a) nebo `||` (OR / nebo). Například jako `((a == b && c > 5) || a > 20)`.

> Logické spojky se vyhodnocují v pořadí `&&` a poté `||`. Pro přehlednost a jistotu správné funkčnosti je však doporučeno používat kulaté závorky, které zaručí, že budou tyto spojky vyhodnoceny přesně tak, jak chceme.

Blok `else if` se bude vyhodnocovat a případně vykonávat pouze v případě, že první podmínka byla vyhodnocena jako `false`. Poslední blok else se provede tehdy, pokud žádná předchozí podmínka nebyla vyhodnocena jako `true`. Opět si to ukážeme na příkladu:

Konstrukt if-else

```
// Je proměnná y pravdivá?
if (y)
{
    // Ano je!
    Console.Write("Pravda!");
}
else
{
    // Ne není!
    Console.Write("Lež!");
}
```

Konstrukt if-else else

```
// Je proměnná y pravdivá?
if (y)
{
    // Ano je!
}
```

```
// Proměnná y je lež. Je tedy alespoň proměnná n pravdivá?
else if (n)
{
    // Proměnná n je pravdivá.
}
else
{
    // Proměnné n ani proměnná y nejsou pravdivé!
}
```

> Podmínka `if` vždy začíná klíčovým slovem `if`, poté může být použit libovolný počet fragmentů `else if` a nakonci může nebo nemusí být fragment `else`.

Jako další si nyní ukážeme konstrukt `switch`:

Konstrukt switch

```
// Pomocná proměnná
int x = 10;

// Jazykový konstrukt switch
switch (x)
{
    // Je proměnná x rovna jedné?
    case 1:
        Console.WriteLine("Hodnota proměnné x je 1.");
        break; // Klíčové slovo break ukončí činnost switche
    case 5:
        Console.WriteLine("Hodnota proměnné x je 5.");
        break;
    default: // Pokud nevyhověl ani jeden case, provede se blok default
        Console.WriteLine("Neznámá hodnota.");
        break;
}
```

Při používání switche začínáme klíčovým slovem `switch` následovaným kulatými závorkami, ve kterých je tzv. řídící proměnná. To je proměnná, jejíž hodnota se bude v těle vyhodnocovat a na základě jejíž hodnoty se pak provede příslušný kus kódu. Dále jsou v těle switche použita klíčová slova `case` následovaná předpokládanou hodnotou řídící proměnné, dvojtečkou (`:`) a poté tělem příslušného kódu. Takovýchto fragmentů může být použit libovolný počet. Na závěr se používá ještě klíčové slovo `default`, které označuje kus kódu, který se má provést, pokud proměnná

nenabyla žádné z předchozích hodnot. Každý case fragment pak musí být vždy ukončen klíčovým slovem `break`.

 # Konstrukce goto:

Na závěr bychom ještě mohli zmínit konstrukci zvanou `goto`. Skládá se z tzv. `návěstí` umístěných v kódu a příkazu `goto: návěstí`, který přeskočí na dané návěstí. Tato konstrukce je však zastaralá a v jazyku C# je pouze z historických důvodů. Dodnes se používá v tzv. nižších programovacích jazycích jako například Assembleru, kde neexistují podmínky, a tudíž zde není jiná možnost. Tato konstrukce je však velice nepřehledná a důrazně doporučujeme tuto konstrukci nepoužívat! Proto se jí zde příliš nevěnujeme.

Cykly

 # Cyklus for

Nyní si pojďme představit `cykly`, které slouží pro opakování určité části kódu po určitý počet kroků, nebo do té doby, dokud není splněna určitá podmínka. Jako první si ukážeme cyklus `for`. Ten může vypadat třeba nějak takto:

Cyklus for

```
for (int i = 0; i < 5; i++)
{


}
```

Takovýto zápis způsobí, že kód v těle `for cyklu` se provede pětkrát. For cyklus přebírá `iniciátor`, `podmínku` a `iterátor`. Tyto tři části jsou odděleny středníky (`;`). Iniciátor je část, která inicializuje `řídící proměnnou`, podmínka určuje do kdy se má cyklus provádět a iterátor určuje, co se má stát na konci každého průchodu cyklem. Velmi důležitá pro nás bude již zmíněná řídící proměnná `i`, která nám říká, po kolikáté již `iterujeme` (tzn. po kolikáté cyklus již prochází). Ta se nám bude velmi hodit při práci s poli:

Cyklus for

```
// Pomocná proměnná
string[] pole = {"Jan", "Petr", "Vašek"};


// Cyklus for
for (int i = 0; i < pole.Length; i++)
{
    Console.WriteLine(pole[i]);
}
```

```
// Výstup:
// => Jan
// => Petr
// => Vašek
```

Za pomocí zápisu `pole[i]` dosadíme v každém průběhu za index pole čísla od `0` do `pole.Length`, která je v tomto případě rovna číslu `3`. Pokud bychom chtěli zajít ještě do většího detailu, v prvním průběhu cyklu dostaneme `pole[0]`, v druhém `pole[1]` a v posledním cyklu `pole[2]`. Poté již cyklus končí, protože přestane platit námi daná podmínka `i < pole.Length (tzn. i < 3)`. Pokud bychom chtěli dostat předchozí nebo následující prvek relativně k současnému průchodu, mohli bychom napsat `pole[i+1]` (následující prvek) nebo `pole[i-1]` (předchozí prvek). Musíme ale dát pozor na krajní situace, kdy iterujeme přes první nebo poslední prvek. V takovém případě bychom skončili s chybou, protože se nemůžeme odkazovat na prvek s negativním indexem (`pole[0 - 1]` - první průchod) nebo na prvek, jež přesahuje celkový počet prvků v poli (`pole[2 + 1]` - poslední průchod). V těle for cyklu můžeme rovněž manipulovat i se samotnou hodnotou řídící proměnné:

Cyklus for

```
for (int i = 0; i < pole.Length; i++)
{
    // Pokud narazíme na sudé číslo,
    // přeskočíme v iteraci číslo následující
    if (pole[i] % 2 == 0)
    {
        // Navýšení iterační proměnné způsobí
        // přeskočení následujícího čísla
        i++;
    }
}
```

Při manipulaci s řídící proměnnou si musíme dávat pozor na to, abychom nechtěně nevytvořili nekonečný cyklus! To je situace, kdy nedáme našemu cyklu šanci dojít ke svému konci. Takový program většinou zamrzne a poté končí pádem z důvodu nedostatku paměti. Na závěr for cyklu si ještě ukážeme několik užitečných ukázek kódu:

For pozpátku

```
// Pomocná proměnná
int[] pole = {1, 2, 3};

// Průchod polem pozpátku
for (int i = pole.Length - 1; i >= 0; i--)
```

```
{
    Console.WriteLine(pole[i]);
}

// Výstup
// => 3
// => 2
// => 1
```

For sudý

```
// Iterace pouze přes sudé prvky
for (int i = 0; i < pole.Length; i++)
{
    if (i % 2 == 0)
    {

    }
}
```

For krátký

```
// Pokud for cyklus obsahuje pouze jeden příkaz,
// mohou být složené závorky vynechány
for (int i = 0; i < pole.Length; i++)
    Console.Write(pole[i]);
```

For krátký

```
// Pokud for cyklus obsahuje pouze jeden příkaz,
// mohou být složené závorky vynechány
for (int i = 0; i < pole.Length; i++)
    Console.Write(pole[i]);
```

Při práci s cyklem for v kombinaci s poli musíme dát pozor na podmínku, neboť pole začínáme indexovat od nuly a jeho vlastnost `Length` nám vrací celkový počet prvků. Velmi častou chybou při práci s poli je tzv. `IndexOutOfRange` exception. Tato chyba nám sděluje, že se snažíme přistoupit k prvku, který v poli neexistuje tzn. náš index je mimo hranice pole (např. záporný nebo větší než počet prvků v poli `n-1`).

 # Cyklus foreach

Dalším naším cyklem je foreach:

Cyklus foreach

```
// Pomocný list
List<int> numbers = {8, 32, 11, 9};

// Cyklus foreach
foreach (int number in numbers)
{
    Console.Write(number + ",");
}

// Výstup: 8, 32, 11, 9,
```

Tento cyklus se používá nejčasteji pro práci s listy a prochází každý prvek z této kolekce prvků. V kulatých závorkách se předává typ (v našem případě `int`) a název proměnné, jež bude reprezentovat v každé iteraci jeden z prvků kolekce (v našem případě proměnná `number`). Jako poslední uvádíme kolekci nebo pole, kterou chceme procházet. Hlavním rozdílem oproti for cyklu je ten, že u cyklu `foreach` nemáme iterační proměnnou. Víme tedy, jakým prvekem momentálně procházíme (proměnná `number`), ale nevíme, kolikátý v pořadí prvek je (u for cyklu proměnná `i`). Cyklem `foreach` lze samozřejmě procházet i pole:

Cyklus foreach

```
// Pomocná proměnná
int[] numbers = {1, 2, 3};

// Foreach cyklus
foreach (int number in numbers)
{

}
```

 # Cyklus while

Posledním cyklem a jeho dvěma variacemi, které si představíme je cyklus `while`. První variací kterou si ukážeme je samostatný `while`:

Cyklus while

```
// Pomocná proměnná
int x = 0;

// Cyklus while
while(x < 5)
{
```

```
    // Vypíšeme řetězec
    Console.WriteLine("Hurá!");

    // Navýšíme iterační proměnnou
    x++;
}

// Výstup programu
// => Hurá!
// => Hurá!
// => Hurá!
// => Hurá!
// => Hurá!
```

Tento cyklus přebírá pouze podmínku, která se vždy vyhodnotí před vstupem do těla cyklu. Pokud se podmínka vyhodnotí jako `true`, pokračuje se provedením těla cyklu. Pokud se vyhodnotí jako `false`, cyklus skončí. Cyklus se tedy provádí do té doby, dokud je podmínka platná. Musíme si dávat pozor na to, abychom cyklu dali opravdu možnost někdy skončit. V našem případě jsme to zajistili navyšováním iterační proměnné `x` v těle cyklu. Druhou variací je do-while:

Cyklus do-while

```
// Pomocná proměnná
int x = 8;

// Cyklus do-while
do
{
    Console.WriteLine("Hurá!");
}
while (x < 5);

// Výstup:
// => Hurá
```

Rozdíl mezi touto a předchozí variací je ten, že zatímco u `while` cyklu se podmínka vyhodnocovala vždy na začátku cyklu (tedy cyklus nemusel proběhnout ani jednou), u `do-while` cyklu se vyhodnocuje podmínka až na konci cyklu. To mimo jiné znamená, že tělo tohoto cyklu se vždy provede alespoň jednou.

> Ač se to může zdát zvláštní, je možné každý cyklus zapsat jako jakýkoli jiný. Každý cyklusmá však své nejvhodnější použití, a proto si ukazujeme všechny.

 # Klíčová slova break a continue

V cyklech se ještě často používají dvě klíčova slova. Prvním je `break`, které provede to, že se konkrétní cyklus, ve kterém se nacházíme ukončí. Druhé slovo je `continue`. Toto slovo se používá, pokud za určité podmínky nechceme vykonat celé tělo cyklu, ale chceme přeskočit rovnou od tohoto slova na konec cyklu.

Praktická ukázka

Pojďme si nyní náš kód z polí z edukativních důvodů 2x přepsat za pomoci podmínek a cyklů a použít všechny výše zmíněné funkce. Naše zadání si nyní upravme tak, že v době psaní kódu nám nebude znám počet osob ve skupině, a program bude načítat vstup až do té doby, dokud uživatel nezadá slovo "END".

 # Varianta 1

Ukázka

```
// Náš list připravený na ukládání příjmů
List<int> prijmy = new List<int>();

// Proměnná připravená na ukládání vstupu od uživatele;
string input;

// Proměnná na uchování konečného výsledku
double prumer = 0;

// Nekonečný cyklus!
while(true)
{
    // Vyzve uživatele k zadání příjmu
    Console.WriteLine("Zadejte příjem (Pro skončení napište END):");

    // Uloží vstup od uživatele
    input = Console.ReadLine();

    // Zkontrolujeme, zda uživatel nechce skončit
    if (input == "END")
    {
```

```
    break;
}
else
{
    // Uživatel nechce skončit - přetypujeme příjem
    // z řetězce na číslo a přidáme ho do listu
    prijmy.Add(Convert.ToInt32(input));
}
}

// Projdeme všechny příjmy a dočasně uložíme
// jejich součet do proměnné prumer
for (int i = 0; i < prijmy.Count; i++)
{
    prumer += prijmy[i];
}

// Vydělíme součet příjmů jejich počtem a
// dostáváme průměr
prumer /= prijmy.Count;

// Vypíše výsledek
Console.WriteLine("Průmerný příjem ve skupině je: {0}", prumer);
```

 # Varianta 2

Ukázka

```
// List příjmů
List<int> prijmy = new List<int>();

// Vstup od uživatele
string input;

// Průměr
double prumer = 0;

// Cyklus do-while
do
{
    // Vyzve uživatele k zadání příjmu
    Console.WriteLine("Zadejte příjem (Pro skončení naište END):");
```

```
    // Získá příjem od uživatele
    input = Console.ReadLine();

    // Konstrukce switch
    switch(input)
    {
        case "END":
            // Ukončí switch (přeskočí i fragment default)
            continue;
        default:
            // Přidá příjem do listu
            prijmy.Add(Convert.ToInt32(input));

            // Ukončí switch
            break;
    }
}
while (input != "END");

// Projdeme všechny příjmy a dočasně uložíme jejich součet do
// proměnné prumer
foreach(int prvek in prijmy)
{
    prumer += prvek;
}

// Vypočítá průměr a uloží ho do proměnné prumer
prumer /= prijmy.Count;

// Vypíše výsledek
Console.WriteLine("Průmerný příjem ve skupině je: {0}", prumer);
```

> Všimněte si, že v první ukázce jsme do podmínky while cyklu vložili přímo hodnotu `true`. Vytvořili jsme tak tzv. nekonečný cyklus. V některých případech může být tento způsob užitečný, ale je nutné si pohlídat, aby byl v těle řádně ukončen. Pro začátek doporučujeme se takovýmto konstrukcím vyhnout, alespoň do té doby, než si budete stoprocentně jisti, co děláte.

Shrnutí

V této kapitole jsme se seznámili s používáním podmínek a cyklů. Funkcemi pro konstrukci podmínek jsou `if`, `else if`, `else` a `switch`. Mezi cykly řadíme `for`, `foreach`, `while` a `do-while`. For cyklus se používá v případě, kdy předem známe počet opakování, foreach se používá nejčastěji pro práci s listy a while pokud neznáme počet opakování, ale chceme něco provádět do té doby, dokud není splněna nějaká podmínka.

> Konstrukce `goto` je zastaralá a nepřehledná a ještě jednou bychom rádi zdůraznili, že je silně doporučeno ji nepoužívat.