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.

Revision #1
Created 2025-05-21 08:29:26 UTC by Magdalena Dobešová
Updated 2025-05-21 08:51:56 UTC by Magdalena Dobešová