PROGRAMOVÁNÍ

OBSAH:

SEZNAM OBRÁZKŮ

Řád učebny, pravidla školní sítě, příprava na výuku

Řád učebny

- viz vyvěšený řád v učebně

Pravidla školní sítě

- seznam dostupných disků
- vlastnosti dostupných disků

Příprava na výuku

- na disku S vytvořit složku pro předmět **PRO1**, složku pro programy **PRO1 programy**
- na ploše vytvořit zástupce těchto složek
- jednotlivé programy – projekty – se budou pojmenovávat takto:
  konzolové aplikace 001C, 002C, 003C atd.
  windowsovské aplikace 001W, 002W, 003W atd.

Učivo předmětu, pomůcky

Učivo

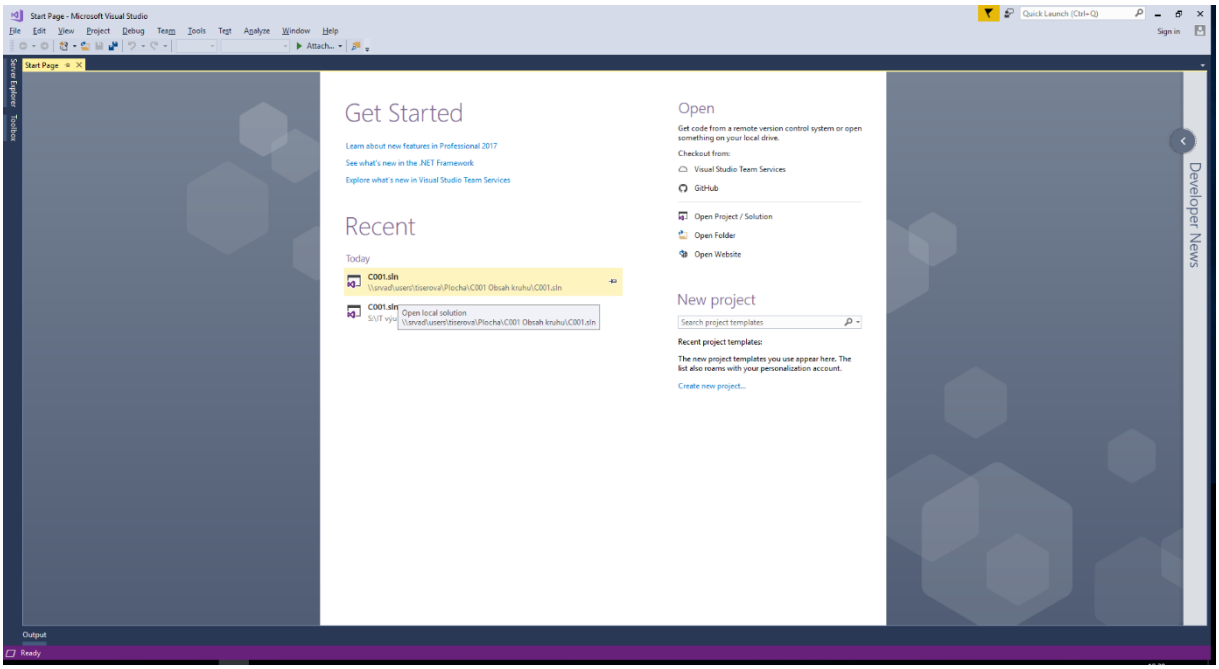
- seznámení s učivem podle ŠVP a časově tematického plánu.

Pomůcky

- kniha;
- soubory pro výuku;
- složka na volné listy formátu A4;
- volné nelinkované listy formátu A4;
- psací potřeby: propiska, tužka, guma.

Nastavení vzhledu Visual Studia 2017

Po spuštění Visual Studio se nám objeví okno:

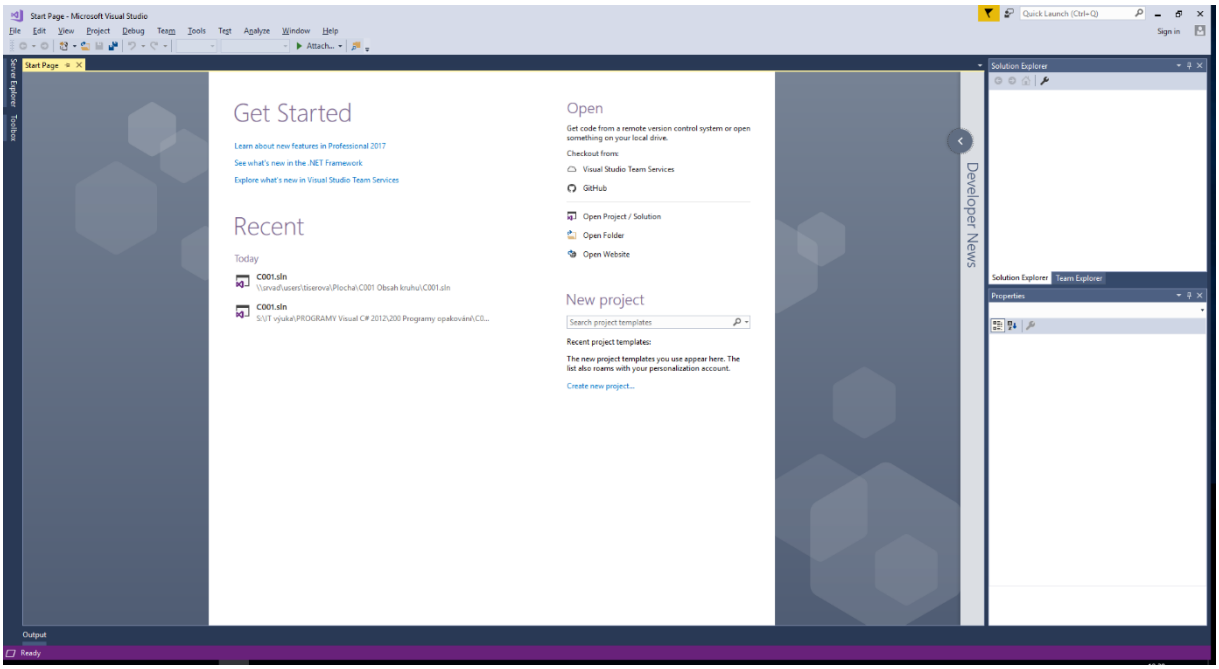


Obrázek 1: Okno spuštěné aplikace

Z menu si nastavíme vzhled:

View → Solution Explorer
View → Error List
View → Properties Windows

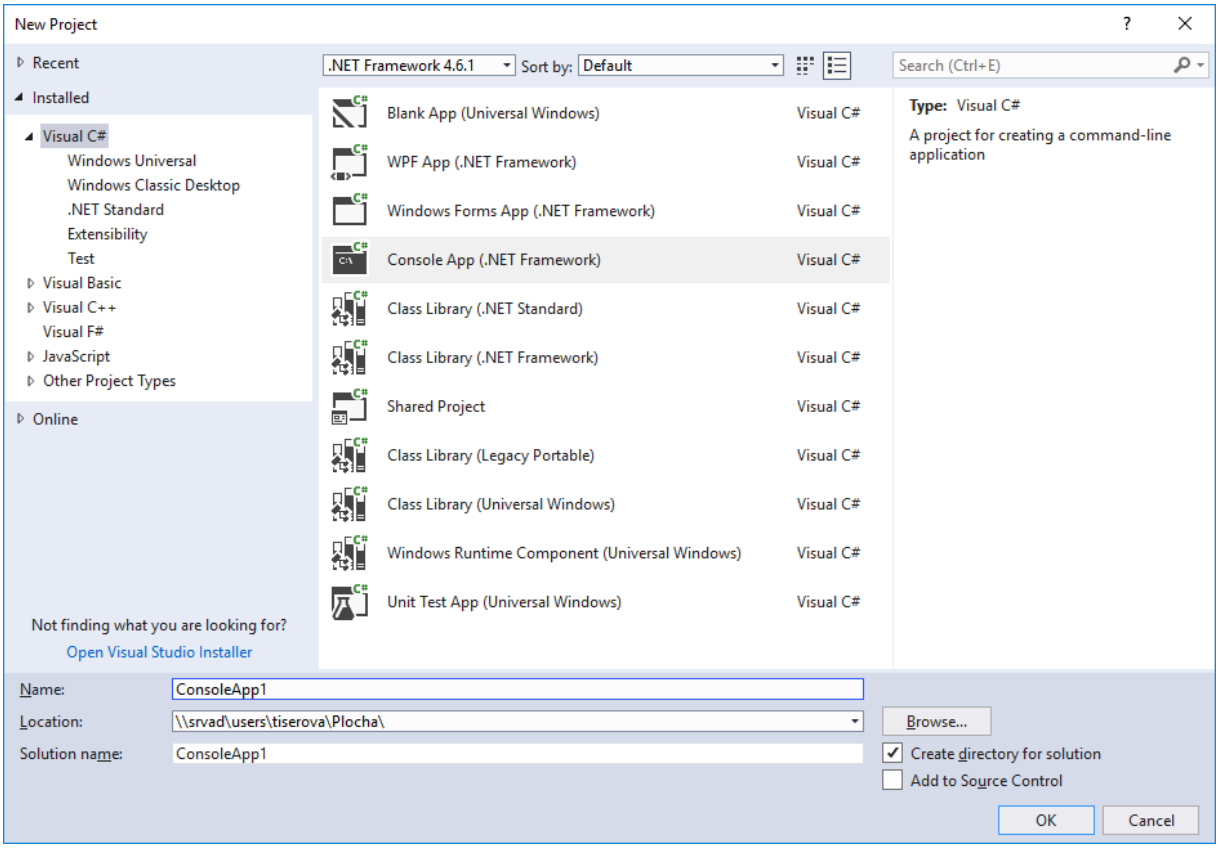
Upravené okno bude mít následující vzhled:



Obrázek 2: Nastavení vzhledu okna

Vytvoření prvního projektu – konzolové aplikace

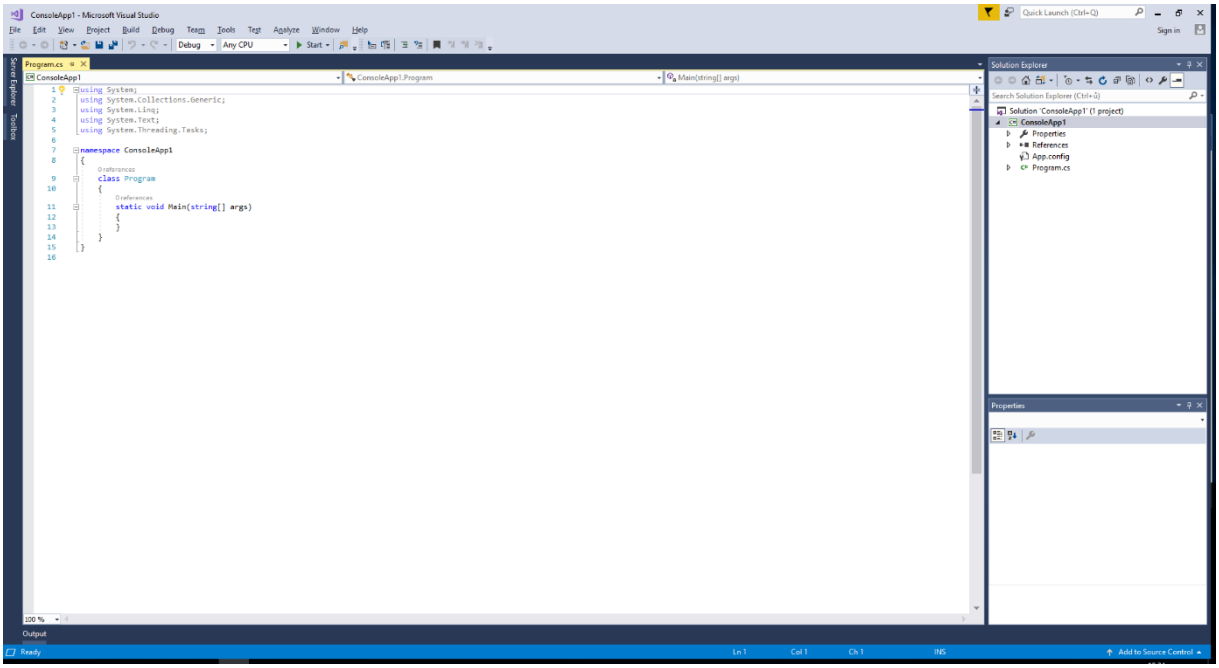
V zobrazeném okně zadáme
File → New Project nebo Create new project a zobrazí se okno:



Obrázek 3: Vytvoření projektu konzolové aplikace

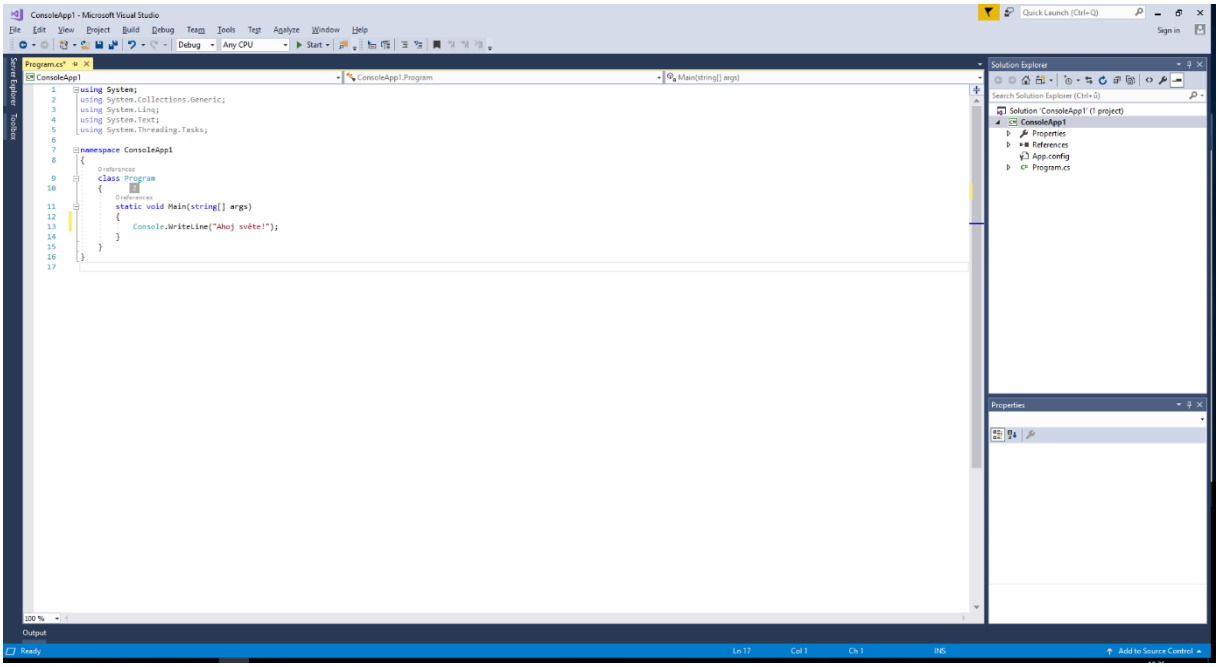
vybereme druh aplikace Console App
napíšeme název projektu do Name
vybereme umístění pro projekt Location
potvrdíme OK

V Visual Studiu se nám vytvořila záložka Program.cs (přípona označuje jazyk C#), ve které jsou vypsány důležité části programu, do které se zapisuje zdrojový kód programu.



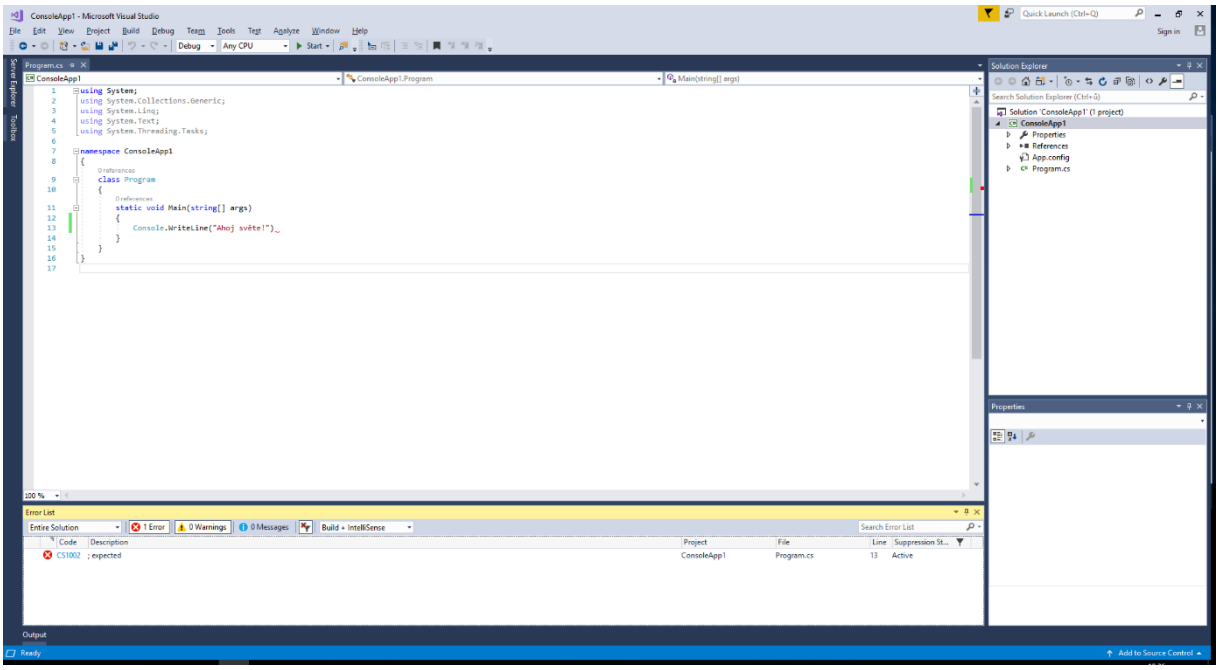
Obrázek 4: Záložka Program.cs

My budeme zapisovat do metody Main – do jejích složených závorek.
Napíšeme text:
Console.WriteLine(“Ahoj světe!”);



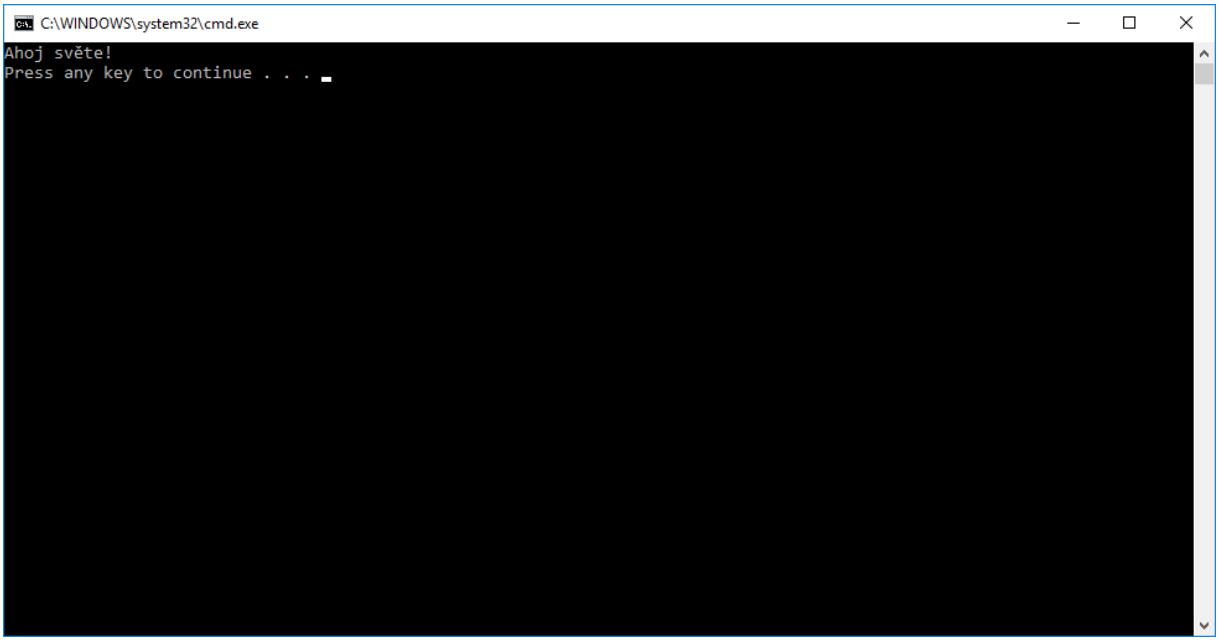
Obrázek 5: Místo pro zápis kódu

Program je napsán, další postup je:
Tlačítko Start nebo Build → Build Solution
v okně Error List vypíše zprávy, pokud program obsahuje chyby



Obrázek 6: Výpis chyb v programu

Tlačítko Start nebo Debug → Start Without Debugging
zobrazí program v okně konzole:

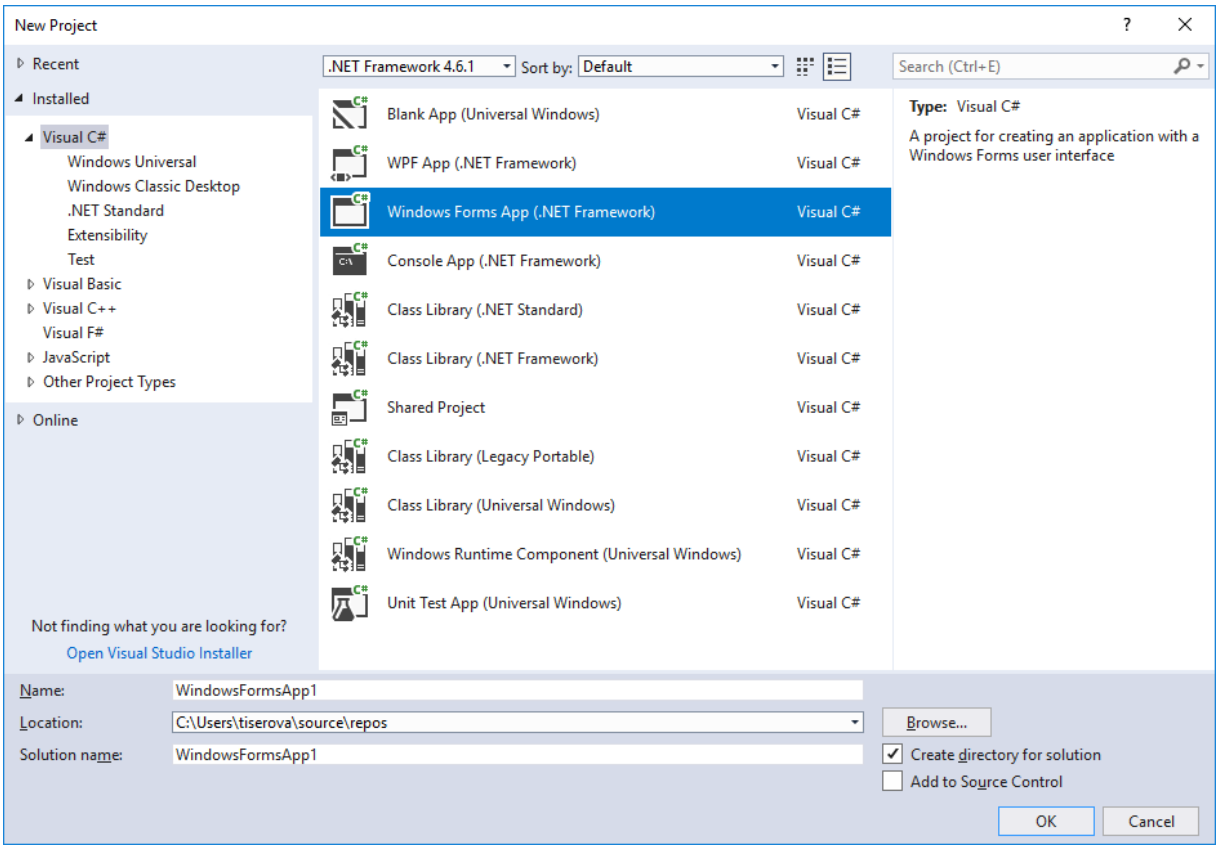


Obrázek 7: Spuštění konzolové aplikace

Kliknutím na Enter na klávesnici okno konzoly zavřete.
Práci můžeme ukončit:
File → Save All
Vše se uloží podle nastavení.

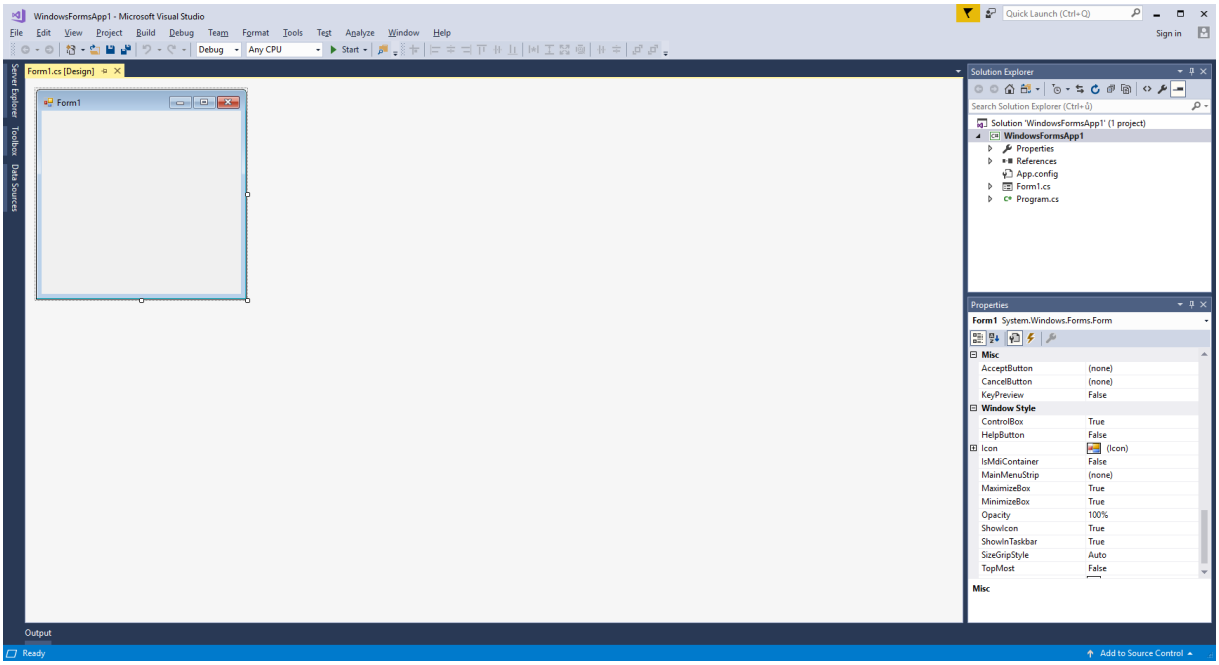
Vytvoření projektu s grafickým uživatelským prostředím

Otevřeme nový projekt a zvolíme typ Windows Forms Application. Napíšeme název a potvrdíme.



Obrázek 8: Vytvoření projektu windowsovské aplikace

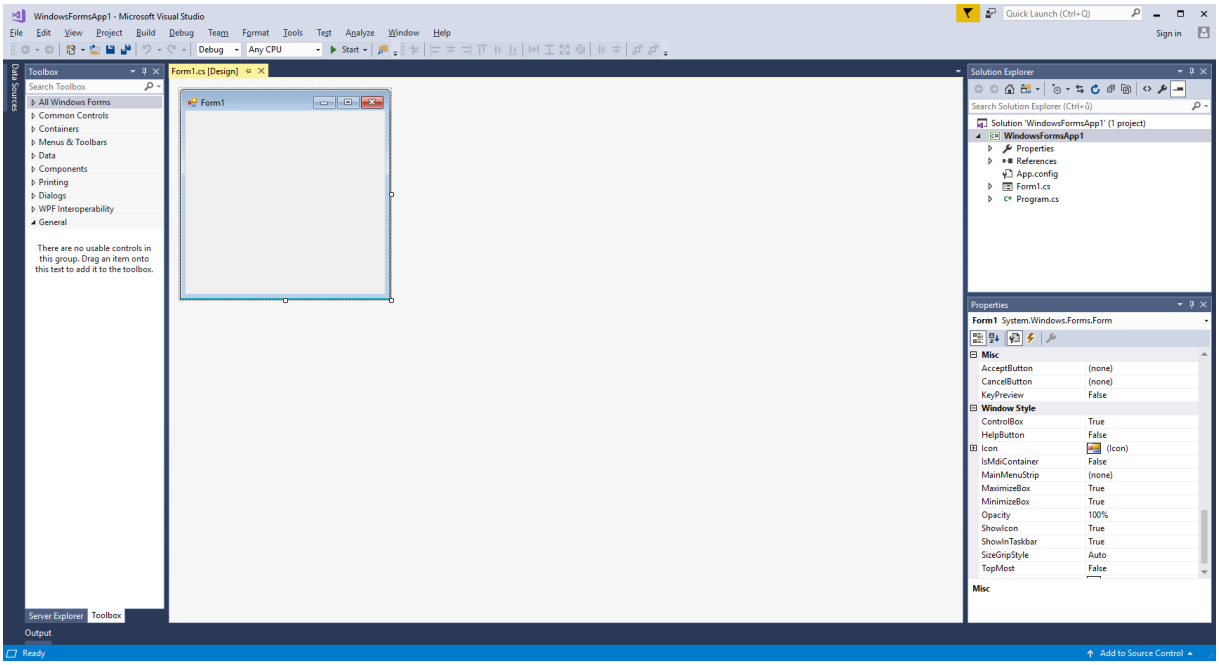
Zobrazí se okno:



Obrázek 9: Formulář pro vytváření windowsovské aplikace

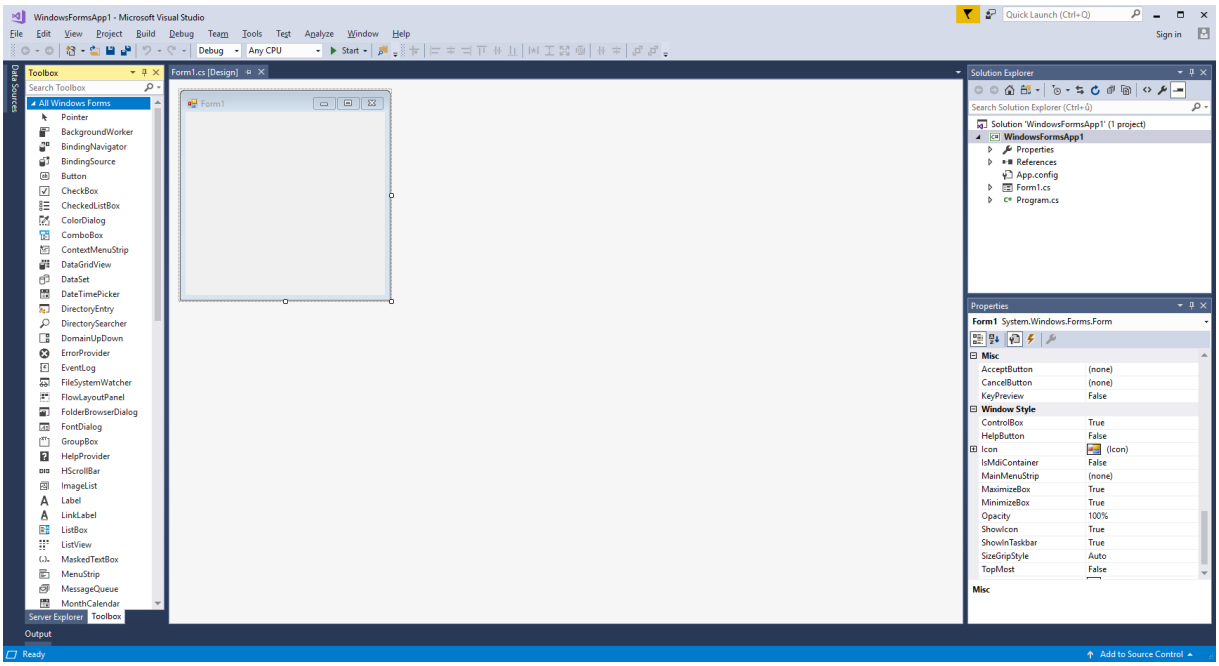
Jak se vytvoří uživatelské prostředí

Klepněte na záložku Toolbox vlevo nebo ji zobraze z menu View → Toolbox.



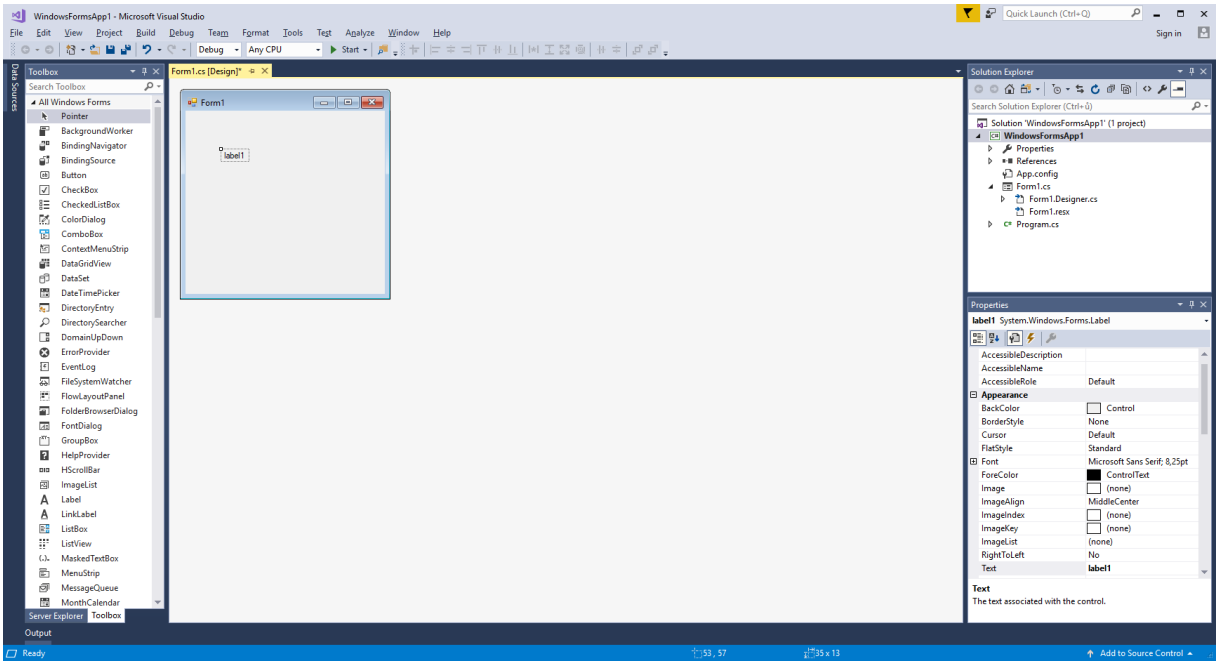
Obrázek 10: Nastavení pro vytváření windowsovské aplikace

Rozviňte v panelu nástrojů Toolbox sadu All Windows Forms pomocí znaménka +. Zobrazí se seznam ovládacích prvků, nejčastěji používaných v aplikacích Windows.



Obrázek 11: Rozbalení nabídky objektů ToolBox

Umístěte prvek Label klepnutím (nebo jej přetáhněte myší) do formuláře Form1.

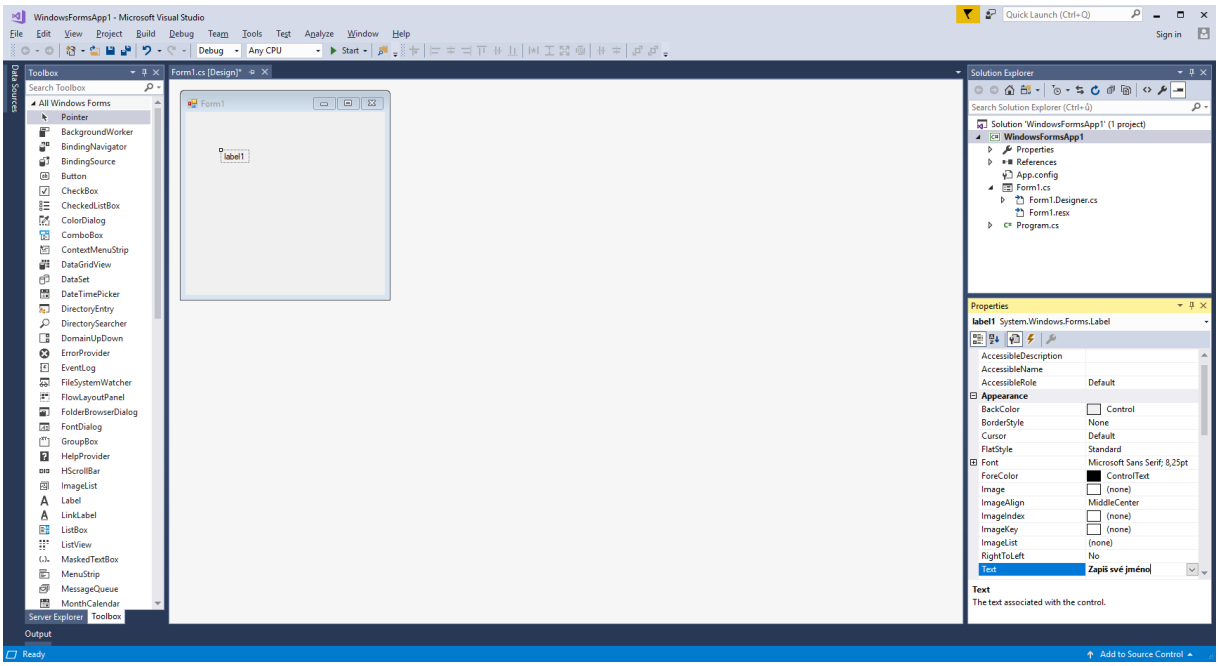


Obrázek 12: Vložení objektu do formuláře

Jeho pozici můžete měnit pomocí myši.

V dialogu vlastností Properties se pak nastavují hodnoty vlastností jednotlivých prvků. Označte prvek label1 ve formuláři Form1.

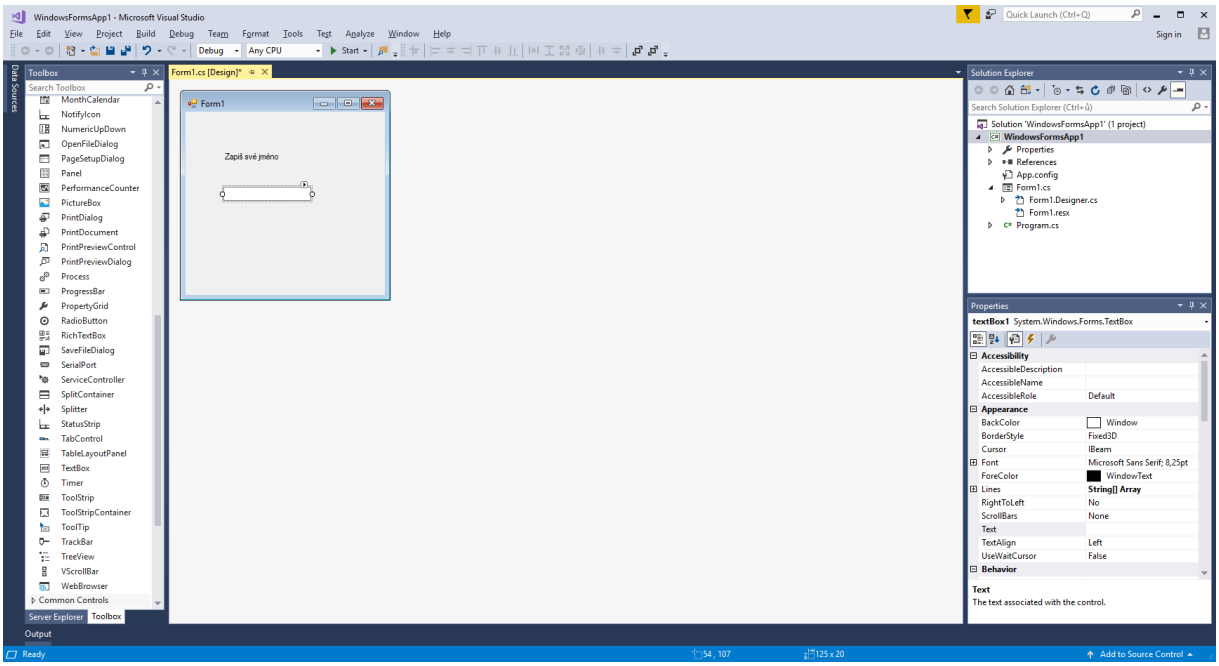
Najděte vlastnost Text a změňte její hodnotu na: Zapiš své jméno.



Obrázek 13: Nastavení hodnoty objektu label1

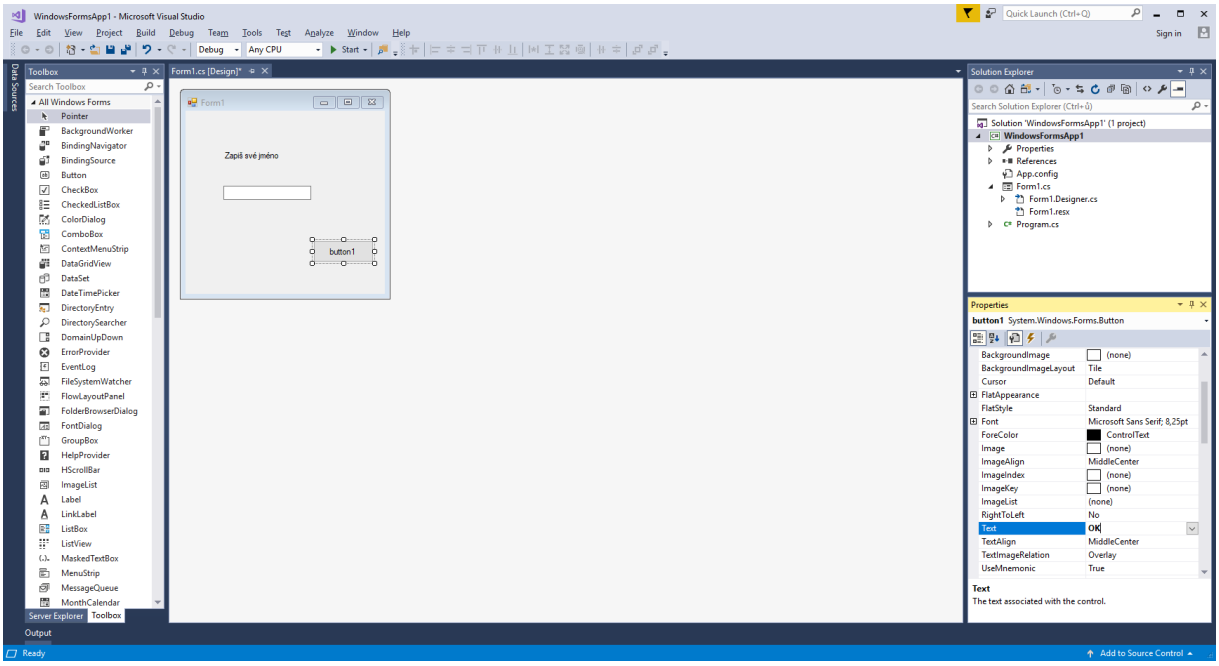
Ze sady nástrojů vyberte prvek TextBox a umístěte jej pod prvek label.

Najděte vlastnost Text tohoto prvku a změňte její hodnotu na: zapiš jej sem.



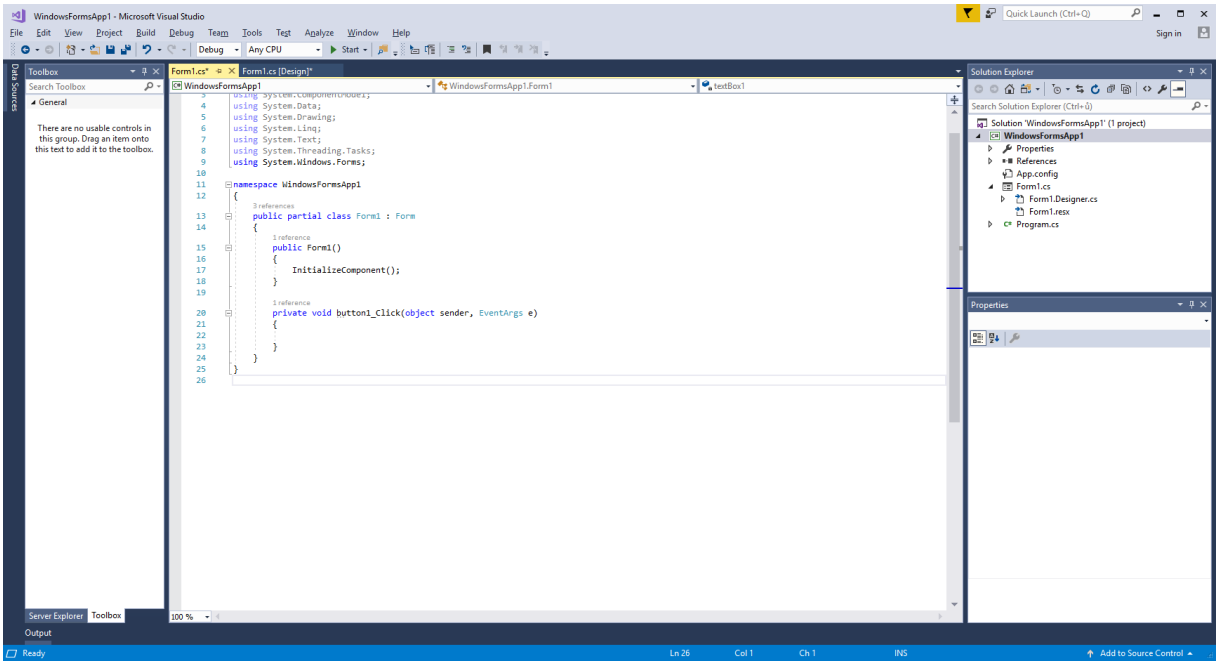
Obrázek 14: Nastavení hodnoty objektu textBox1

Ze sady nástrojů vyberte prvek button a umístěte jej do formuláře.
Změňte jeho vlastnost Text na: OK.



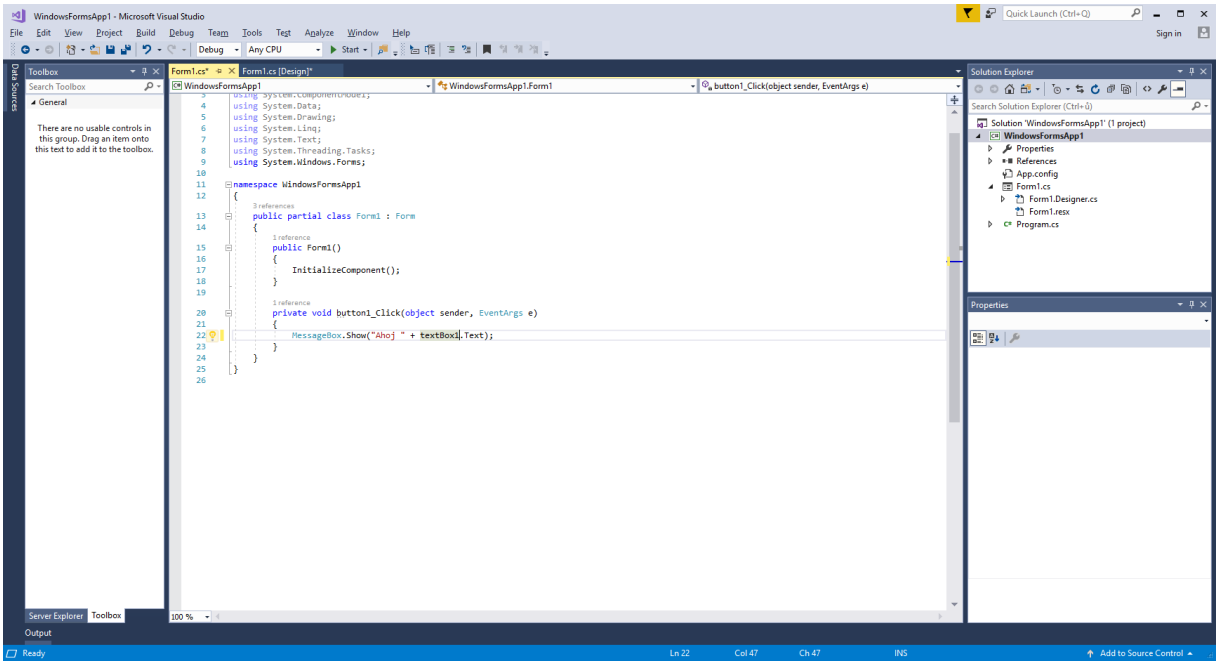
Obrázek 15: Nastavení hodnoty objektu button1

V dialogu Solution Explorer klepněte myší na soubor Form1.cs a pak na ikonku View Code. Objeví se obsah zdrojového kódu Form1.cs.



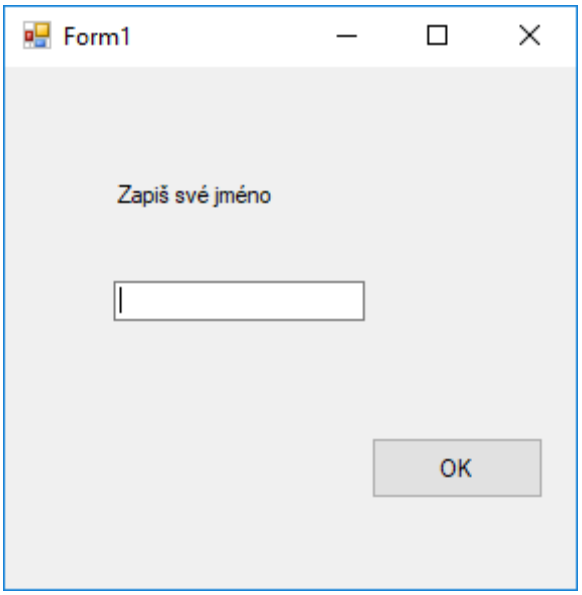
Obrázek 16: Zobrazení kódu programu

Přibyla nová záložka Form1.cs.
Je zapotřebí zapsat kód pro tlačítko OK.
Klikněte dvojitě na tlačítko OK ve formuláři – otevře se zdrojový kód Form1.cs.
Do metody ok_Click zapište:
MessageBox.Show("Ahoj " + jmenoUzivatele.Text);



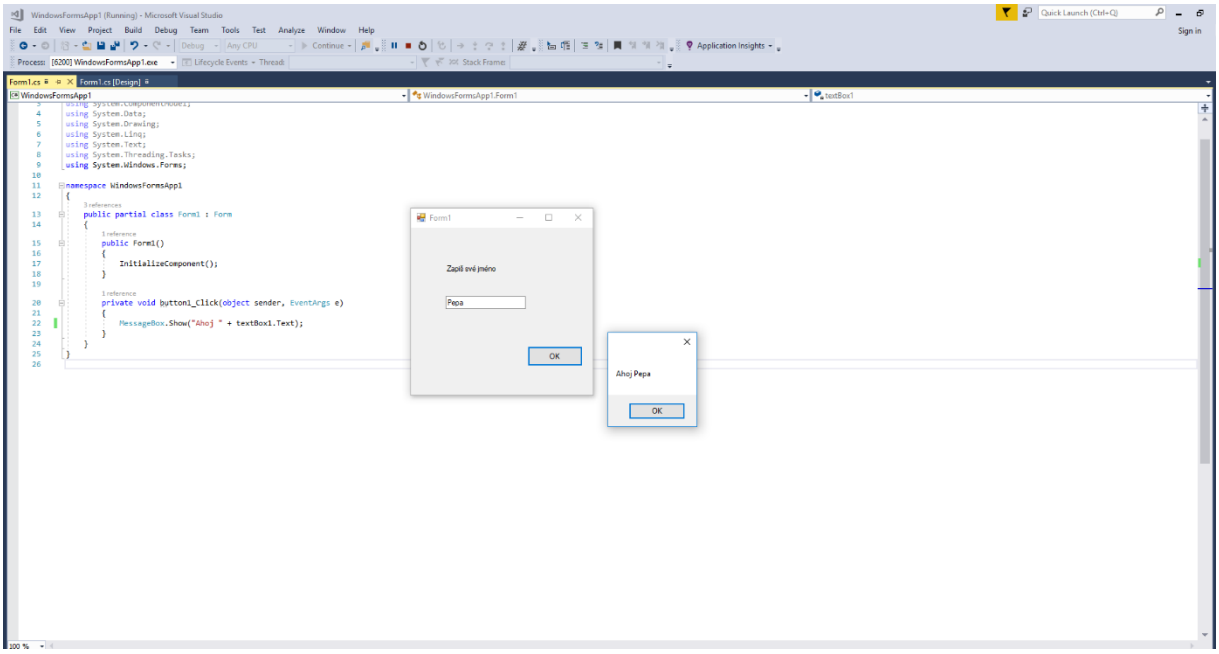
Obrázek 17: Zápis kódu do metody

Nyní lze soubor spustit tlačítkem Start nebo Debug → Start Without Debugging.



Obrázek 18: Spuštění windowsovské aplikace I

Do spuštěné aplikace napište jméno a potvrďte kliknutím na tlačítko OK.
Zobrazí se okno, tzv. MessageBox se zapsaným jménem.



Obrázek 19: Spuštění windowsovské aplikace II

Celou aplikaci pak zavřete křížky.

Ladění programu

Ladicí prostředky se dodávají s prostředím Visual C#.
Používá se pro hledání chyb, porozumění chodu programu nebo jeho částí, pro inspekci paměti, kdy jsou vypisovány hodnoty jednotlivých proměnných i celých objektů.

Zarážka
umisťuje se na kterýkoli příkaz kliknutím na pravé tlačítko myši volbou
Breakpoint→Insert Breakpoint
ukazuje příkaz, který se má v následujícím kroku vykonat

Krokování
spouští se klávesou F11 nebo F10

Inspekce paměti
provádí se v okně Debug → Windows → Locals

Psaní vybraných znaků

Složené závorky

{ pravý ALT + B

} Pravý ALT + N

Hranaté závorky

[pravý ALT + F

] Pravý ALT + G

Znak pro logický součin

& pravý ALT + C

Znak pro logický součet

| pravý ALT + W

Zpětné lomítko

\ pravý ALT + Q

Apostrof

' Shift + ň (Shift + klávesa s písmenem ň – pod Backspace)

Zaokrouhlování čísel

Zaokrouhlení na 1 desetinné místo	`o = Math.Round(o * 10) / 10;`
Zaokrouhlení na 2 desetinná místa	`o = Math.Round(o * 100) / 100;`
Zaokrouhlení na 3 desetinná místa	`o = Math.Round(o * 1000) / 1000;`
Zaokrouhlení na celá čísla	`o = Math.Round(o);`
Zaokrouhlení na desítky	`o = Math.Round(o * 0.1) / 0.1;`
Zaokrouhlení na sta	`o = Math.Round(o * 0.01) / 0.01;`
Zaokrouhlení na tisíce	`o = Math.Round(o * 0.001) / 0.001;`

Algoritmus a vývojový diagram

Algoritmus

Algoritmus je přesný předpis definující výpočtový proces vedoucí od měnitelných výchozích údajů až k žádaným výsledkům", posloupnost konečného počtu elementárních kroků vedoucí k vyřešení úlohy.

Má dvě základní varianty - verbální a písemnou.

Každý algoritmus musí splňovat základní požadavky, musí mít vlastnosti:

1. **determinovanost**: algoritmus musí být přesný a srozumitelný. V žádné etapě řešení nesmí připouštět pochyby o tom, co je třeba v dané etapě udělat a jaká bude jednoznačně navazující etapa, tj. jak postupovat dále. Musí být jednoznačně určeno, jak za sebou následují jednotlivé kroky řešení.
2. **masovost**: algoritmus musí být popisem řešení nikoliv jediné úlohy, ale celé skupiny příbuzných úloh lišících se od sebe jen výchozími údaji.
3. **resultativnost**: musí vždy vést k jednoznačnému výsledku (a nemusí to být výsledek přijatelný, i výsledek špatný, ne chybný, je výsledkem) a ukončení řešení.

Varianty zápisu algoritmu

1. **Ústní -** slovní popis návodu řešení daného problému.
2. **Písemná - grafické zobrazení algoritmu pomocí vývojového diagramu nebo strukturogramu.**

Vývojový diagram

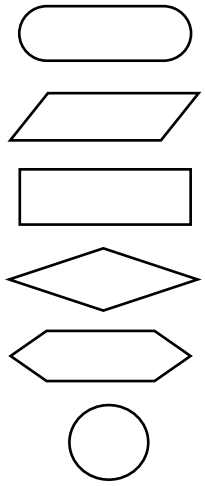
Vývojové diagramy představují normou (dnes již nadnárodní) definované symbolické značky a pravidla pro jejich používání, sloužící k jednoznačnému grafickému vyjadřování výpočetních operací a postupů. Vývojový diagram tak slouží jednak k popisu výpočetního algoritmu a zároveň jako podklad pro sestavení programu pro počítač.

Vývojový diagram se používá pro názorné zobrazení algoritmu zpracování informací a případnou stručnou publikaci programů. Tento jazyk je tvořen značkami s jejich jednoznačným významem a pravidly, jak tyto značky ve vzájemné souvislosti používat.

Shrnutí

- je to graficky zobrazený algoritmus;
- slouží pro zápis postupu řešení problému;
- popisuje výpočetní algoritmus;
- slouží jako podklad pro sestavení programu pro počítač;
- používá se pro případnou stručnou publikaci programů;
- je tvořen přesně definovanými symbolickými značkami s jejich jednoznačným významem (sémantika – slovník) a pravidly pro jejich používání ve vzájemné souvislosti (syntaxe – gramatika);

Používané (normované) grafické symboly:



start, stop (začátek, konec)

instrukce pro vstup a výstup

příkazy, úkony

podmínka, větvení

přípravná činnost

mezní značka

Strukturogramy

používají obdobné symboly ale přesnější - tento systém přesně splňuje podmínky důležité pro strukturované programování

Postup algoritmizace při řešení složitějších úloh

1. zadání úlohy, formulace problému;
2. analýza problému a nástin řešení;
3. analýza vstupních a výstupních dat - návrh použitých datových struktur v programu (pole apod.);
4. návrh algoritmu;
5. zápis v programovacím jazyce a jeho následné ladění;
6. zkušební provoz programu + tvorba dokumentace;
7. zhodnocení řešení a jeho následné updatování.

Další pojmy

Program
je algoritmus přepsaný do programovacího jazyka.

Programovací jazyk
je prostředek pro vytvoření programu (např. z algoritmu, vývojového diagramu).
Typy programovacích jazyků – obecné (Pascal, C, Visual Basic), databázové (MS Access), orientované (Java).

Strojový kód
je program vyjádřený ve dvojkové soustavě, je srozumitelný počítači a je zpracovatelný počítačem.

Zdrojový kód
je text programu.

Syntaxe, syntax
je skladba, nauka o vzájemných vztazích skladebných prvků jazyka, pravidla pro vytváření přípustné kombinace symbolů, přesné určení, jak se mají jednotlivé příkazy a části programu zapisovat.

Sémantika
je nauka o významu jazykových jednotek.

Základní prvky jazyka C#

Bílé znaky

- mezera, nový řádek, tabulátor, …
- nejsou vidět na obrazovce.

ASCII tabulka

- popisuje kódy, které jsou přiděleny jednotlivým znakům;
- má rozsah 0–255 znaků, ale běžně se používá 0–127.

Identifikátory

- pojmenovávají klíčová slova, proměnné, metody, třídy, …
- pro jejich vytvoření platí pravidla;
- identifikátor musí začínat písmenem, nikdy číslicí: ~~1student~~ student1 (podtržítko je považováno za znak).

Pojmenování proměnných

- název začíná malým písmenem;
- rozlišují se malá a velká písmena: objem ≠ Objem ≠ oBjem, ale nevytváříme identifikátory, které se liší jen velikostí písma;
- identifikátor začíná písmenem, nikdy číslicí: student1 ~~1student~~;
- podtržítka se nepoužívají;
- při vytváření delších identifikátorů používáme velbloudí zápis – každé druhé a další slovo začíná velkým písmenem: vzdalenostBodu, obsahCtverce;
- identifikátor nesmí obsahovat mezeru: ~~objem valce~~.

Klíčová slova

- mají předdefinovaný význam, píší se malými písmeny, označují např. příkazy: namespace, class, using, if, for, while;
- klíčová slova se nepoužívají pro pojmenování částí programu, proměnných atd.

Komentáře

- zpřehledňují zdrojový kód programu pro programátora;
- jsou ignorovány kompilátorem;
- komentář do konce řádky píšeme za dvě lomítka; `//text`
- komentář na více řádků píšeme do závorek;

```
/*text*/
nebo
/*
text na více řádků
až po další
*/
```

- komentář, který s pomocí nástroje a předepsaných XML značek slouží k vygenerování vývojové dokumentace. `///text`

Základní struktura programu

Základní programovou jednotkou v jazyce C# je metoda. Program má vždy alespoň jednu metodu, metodu Main. Metoda je volaná kompilátorem a lze z ní volat další metody, se kterými si může předávat hodnoty nazývané vstupní parametry a návratovou hodnotu.
Použití metod umožňuje rozčlenit program na menší části.

Základní kód konzolové aplikace

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApplication1
{
   class Program
   {
      static void Main(string[] args)
      {
      }
   }
}
```

Části programu

using

- je příkaz, který zařazuje jmenný prostor do oboru platnosti. Ve jmenných prostorech jsou umístěny třídy, které se používají tak často, že Visual Studio je do projektu zařazuje automaticky, pokud potřebujeme metody, které jsou v jiných třídách, musíme je vložit.

namespace

- je jmenný prostor, v němž se nacházejí třídy nebo jiné identifikátory.

class

- je třída.

static void Main

- Main je metoda, která se automaticky vyvolává při startu programu, za název metody se zapisují jednoduché závorky a do nich případně parametry předávané metodě;
- identifikátor (název) metody Main musí být napsán velkým počátečním písmenem.

Další pokyny

- každý příkaz je ukončen středníkem;
- do složených závorek {} se zapisuje blok příkazů, je to skupina příkazů, které se vykonávají vždy společně, jako celek, bývají to např. příkazy jedné metody nebo příkazy které se mají provést při splnění určité podmínky.

Příklad bloku příkazů

```
{                       //začátek bloku
   příkaz_1;
   příkaz_2;
   ...
   příkaz_n;
}                       //konec bloku
```

Proměnné, deklarace proměnných

Proměnné

označují místo v paměti pojmenované identifikátorem. Jsou v nich uloženy hodnoty dat (číslo, text). Hodnotou proměnné je tedy hodnota uložená na tomto místě.

Proměnná je datový objekt s pevně stanoveným jménem (identifikátorem), jehož hodnota se může v průběhu výpočtu měnit.

Deklarace proměnných

Proměnná se musí před použitím deklarovat, tj. musí se sdělit překladači název proměnné a typ hodnoty, která je v ní uložená.

Deklarace proměnné je příkaz, který přidělí proměnné určitého typu jméno a paměť.

Příklad:

```
int i;              //deklarace proměnné s názvem i typu  int
int vyska;          //deklarace proměnné s názvem vyska typu  int
int i, j, k;        //deklarace proměnných s názvy i, j, k typu int
double delka;       //deklarace proměnné s názvem delka typu  double
```

Inicializace proměnné

Abychom mohli s proměnnou - její hodnotou - pracovat, musí obsahovat nějakou hodnotu.

Hodnotu proměnné můžeme přiřadit při její deklaraci (při deklaraci můžeme proměnnou ihned inicializovat, tj. přiřadit jí počáteční hodnotu) nebo ji můžeme načíst z klávesnice.

Příklad:

```
double vzdalenost = 0;      //deklarace proměnné s názvem vzdalenost, typu double,
                            iniciované hodnotou 0
```

Deklarace proměnné tedy obsahuje:

- datový typ proměnné;
- jméno proměnné, jíž je datový typ přiřazen, je-li definováno více proměnných, oddělují se čárkami;
- inicializace proměnné – určení počáteční hodnoty (je-li třeba).

Primitivní datové typy C#

[primitivní = vestavěné v jazyce]

Přehled nejčastěji používaných typů

Datový typ	Název typu v .Net Framework	Popis	Velikost (bity)	Rozsah hodnot
int	System.Int32	Celé číslo	32	-2147483648 až 2147483647
uint	System.UInt32	Celé číslo	int bez znaménka	0 až 4294967295
byte	System.Byte	Celé číslo	8	0 až 255
sbyte	System.SByte	Celé číslo	byte se znaménkem	-128 až 127
short	System.Int16	Celé číslo	16	-32768 až 32767
ushort	System.Uint16	Celé číslo	short bez znaménka	0 až 65535
long	System.Int64	Celé číslo	64	-9223372036854775808 až 9223372036854775807
ulong	System.Uint64	Celé číslo	long bez znaménka	0 až 18446744073709551615
float	System.Single	Desetinné číslo, přesnost 7 desetinných míst	32	$\pm 1.5 \times 10^{-45}$ do $\pm 3.4 \times 10^{38}$ se 7 významnými číslicemi
double	System.Double	Desetinné číslo, přesnost na 15 nebo 16 desetinných míst	64	$\pm 5.0 \times 10^{-324}$ do $\pm 1.7 \times 10^{308}$ s 15 nebo 16 významnými číslicemi
decimal	System.Decimal	Desetinné číslo, peněžní hodnoty	128	$\pm 1.0 \times 10^{-28}$ až $\pm 7.9 \times 10^{28}$ 28 významných číslic
bool	System.Boolean	Pravdivostní hodnota		true nebo false
char	System.Char	Znak	16	znak
string	Systém.String	Textový řetězec znaků	16 bitů na každý znak	nelze vyjádřit

Datové typy proměnných

Datové typy jsou typy proměnných, které je možno vytvořit. Proměnná může představovat paměťovou buňku, ale většinou se jedná o blok paměti, kde je uložena informace jistého typu. Právě typ proměnné je nejdůležitější atribut proměnné. Jedna proměnná může uchovat záporné číslo a druhá zase jen kladné. Musíte předem vědět, k čemu proměnnou

chcete využít, aby program fungoval správně. Některé chyby odhalí kompilátor při překladu, ale pozor si musíte dát především na ty skryté, které se projeví nesprávnou funkcí programu. V průběhu programu se hodnota proměnné mění, podle druhu operace. Můžeme například sečíst dvě proměnné stejného typu atd.

Datový typ určuje, jakým způsobem bude v paměti uchována hodnota proměnné. Dále říká, jaké operace mohou být s proměnnou prováděny. Určuje také rozsah proměnné.

Některé typy mají dvě varianty a to sice znaménkové (signed) a neznaménkové (unsigned). Implicitně jsou tyto typy znaménkové. Pokud chcete vytvořit neznaménkový typ, stačí před klíčové slovo typu vložit další klíčové slovo unsigned. Co ale přesně znamená znaménkový a neznaménkový typ? Je to snadné.

Znaménkové typy mohou nabývat i záporných hodnot, ale mají oproti neznaménkovým typům poloviční maximální hodnotu, takže například platí:

Typ	Rozsah hodnot
unsigned char	0 až 255
char	-128 až +127

Maximální hodnotu proměnné zjistíme takto: 2n-1, kde n je počet bitů, které proměnná zabírá v paměti.

Vliv znaménkového bitu na hodnoty, které lze v bitu uchovat

|_|
1 bit → 2^1 = 2 hodnoty
→ {0,1}

|_||_||_||_||_||_||_||_|
1 byte bez znaménkového bitu → 2^8 = 256 hodnot
→ <0; 255>

|_||_||_||_||_||_||_||_|
1 byte s 1 znaménkovým bitem → 2^7 = 128 hodnot,
ale 128 kladných a 128 záporných
→ <-128; -1> U <0; 127>

Metody pro vstup a výstup na konzoli

Výstup

Console.WriteLine(výraz);
provede vypsání výrazu na konzoli a umístí kurzor na další řádek.

Console.Write(výraz);
provede vypsání výrazu na konzoli, kurzor zůstane umístěn na stávajícím řádku.

Výrazem může být např. proměnná, text – což je řetězec znaků.

Výpis hodnoty proměnné:

```
Console.WriteLine(nazevpromenne);
```

Výpis textu:

```
Console.WriteLine("Text určený k zobrazení!!!");
```

text musí být uzavřen v uvozovkách.

Výpis kombinace textu a hodnoty proměnné:

- Textový výstup:

```
Console.WriteLine("Hodnota proměnné a = " + a);
Console.WriteLine("Hodnota proměnné a = " + a + ", hodnota proměnné b = " + b);
```

- Složené formátování:

```
Console.WriteLine("Hodnota proměnné a = {0}", a);
Console.WriteLine("Hodnota proměnné a = {0}, hodnota proměnné b = {1}", a, b);
```

Vstup

Console.ReadLine();
po načtení hodnoty z klávesnice do proměnné umístí kurzor na další řádek.

Console.Read();
po načtení hodnoty z klávesnice do proměnné kurzor zůstane umístěn na stávajícím řádku.

Parsování

Pro načtení hodnoty do proměnné, je zapotřebí provést její převod na potřebný datový typ. Převodu z textové podoby do specifické podoby se říká parsování.

Při načtení hodnoty z klávesnice je tato datového typu string a musí se převést do datového typu proměnné, do které se hodnota načítá.

Výjimkou je načítání do proměnné typu string, kde se parsování neprovádí.

Příklady pro různé datové typy proměnných:

```
int a = int.Parse (Console.ReadLine());
double c = double.Parse (Console.ReadLine());
bool d = bool.Parse (Console.ReadLine());
string e = Console.ReadLine();
```

Příkazy pro vstup a výstup ve windowsovské aplikaci

Příkaz pro výstup do textového pole - obecně

textBox1.Text = (výraz);

v objektu textBox1 se zobrazí výraz.

Zobrazení textu v textovém poli

```
textBox1.Text = "AHOJ SVĚTE!";
```

v textovém poli se zobrazí text v uvozovkách

Zobrazení proměnné x v textovém poli

```
textBox1.Text = Convert.ToString(x);
```

v textovém poli se zobrazí hodnota proměnné x

Zobrazení textu a proměnné x v textovém poli

```
textBox1.Text = "Hodnota proměnné x = " + Convert.ToString(x);
```

Příkaz pro vstup

```
x = int.Parse (textBox1.Text);
```

do proměnné x se uloží hodnota typu int z objektu textBox1, konkrétně jeho vlastnosti Text

Zobrazení výstupu v messageboxu

```
MessageBox.Show("Dnes je " + textBox1.Text);
```

Vymazání textového pole

```
textBox1.Clear();
```

Zavření aplikace

```
Application.Exit();
```

Operátory

Aritmetické operátory

Operátor	Význam
+	Sčítání
-	Odčítání
*	Násobení
/	Dělení
%	Dělení modulo (zbytek po celočíselném dělení)

Tyto aritmetické operátory operují s hodnotami a vytvářejí z nich hodnoty nové.
Všechny operátory nelze aplikovat na všechny datové typy, např. na int, string a bool.

Další operátory

Operátor	Význam
<	Menší než
<=	Menší nebo rovno
>	Větší než
>=	Větší nebo rovno
==	Test na rovnost
!=	Test na nerovnost
!	Logická negace
&&	Logický součin (and)
||	Logický součet (or)
++	Inkrementace (přičtení 1)
--	Dekrementace (odečtení 1)

Priorita operátorů

V jazyce C# mají přednost operátory násobení (*), dělení (/) a modulo (%) před operátory sčítání (+) a odčítání (-) .
Pořadí vyhodnocování výrazu lze změnit použitím závorek.

Asociativita operátorů

Pokud se ve výrazu vyskytují operátory se stejnou prioritou, pak začíná hrát roli asociativita, tj. směr vyhodnocování operátorů buď doleva nebo doprava.

Operátory * a / jsou asociativní zleva.

Příklad na asociativitu operátorů
6 / 3 * 4

při asociativitě zleva (což je u těchto operátorů správné) je výsledek získán řešením:
6 / 3 = 2
2 * 4 = 8

při asociativitě zprava by řešení vypadalo:
3 * 4 = 12
6 / 12 = 0,5

Příklady na dělení modulo
Celočíselné dělení

3 / 2 = 1 zb. 1
5 / 2 = 2 zb. 1
6 / 3 = 2 zb. 0

Dělení modulo

3 % 2 = 1
5 % 2 = 1
8 % 3 = 2
9 % 5 = 4

Sudá čísla

8 % 2 = 0
10 % 2 = 0
12 % 2 = 0

Lichá čísla

9 % 2 = 1
11 % 2 = 1
13 % 2 = 1

Dělitelnost čísel např. 3

9 % 3 = 0 je děl. 3
10 % 3 = 1 není děl. 3
11 % 3 = 2 není děl. 3
12 % 3 = 0 je děl. 3

Příkaz přiřazení

V jazyce C se přiřazení provádí pomocí operátoru = (rovná se).
Znamená, že výrazu na levé straně rovnice se přiřadí hodnota výrazu na straně pravé.

Příklady přiřazení

```
a = 4;                  // a = 4
b = ( a + 7 ) * 5;      // b = 55
c = d = a + b ++;       // d = 59, c = 59,  b = 56
d + 6 = b – 5;          // takto nelze zapsat příkaz přiřazení
```

Běžné operátory mají stejný smysl jako v jiných jazycích, běžným způsobem se používají i závorky.

Pozor však na to, že = je přiřazovací operátor a jako test na rovnost je nutno zapsat dvě rovnítka ==.

V řadě případů je však používání výrazů v C unikátní. Některé příkazy mají vedlejší efekt, tedy jednak provedou nějakou akci, která mění okolí, a současně vracejí hodnotu, díky čemuž mohou být součástí složitějšího příkazu.

Například následující řádek

```
i = j = k = 30;
```

je korektní příkaz. Protože operátory = se vyhodnocují (poněkud neobvykle) zprava doleva, chápe se tento příkaz jako

```
i = (j = (k = 30));
```

a všem třem proměnným se přiřadí hodnota 30.

Podobně

```
i = i + 1;
```

je tzv. přiřazovací příkaz, který zvýší hodnotu i o jedničku. Zároveň však tento příkaz má hodnotu, která je rovna nové hodnotě i.

Inkrementace a dekrementace

Zvýšení hodnoty celočíselné proměnné o jedničku je možno provést i jinými příkazy:

```
i+=1;
i++;
++i;
```

Jak i++ tak ++i způsobují, že hodnota proměnné se zvýší o jedničku. Rozdíl je v tom, že ++i znamená napřed zvětšit i o jedničku a použít tuto novou hodnotu, zatímco i++ sice rovněž i zvětší, ale do dalšího vyhodnocení obklopujícího příkazu použije ještě starou hodnotu.

Tedy po provedení

```
i = 5;
j = ++i;
```

bude mít i a j hodnotu 6, ale po

```
m = 5;
n = m++;
```

bude mít m hodnotu 6, ale n hodnotu 5.

Operátor -- znamená samozřejmě analogicky odečtení jedničky.

Výrazy versus příkazy

Zapíšeme-li za přiřazovací výraz středník, stane se z něj přiřazovací příkaz, např. i = i + j;

Podobně lze psát např.

```
i++;
```

Středník opět udělal z výrazu příkaz.

Poznamenejme, že středník má v jazyce C jiný význam než v Pascalu a řadě dalších jazyků. Neslouží k oddělení dvou příkazů od sebe, ale je integrální částí příkazu. Proto musí být zapsán i za posledním příkazem. Nepíše se však za příkazy řídicích struktur, leda jako součást vloženého příkazu.

Řídicí struktury

Program provádí příkazy v pořadí, v jakém jsou zapsány v těle funkce Main. Pro změnu posloupnosti prováděných příkazů se používají tzv. řídicí struktury.

Tyto řídicí struktury obsahují řídicí výrazy (booleovské), na kterých závisí, jaká část programu se bude právě provádět. Tyto výrazy nabývají hodnoty pravda = true nebo nepravda = false. Tyto hodnoty jsou v jazyce C reprezentovány hodnotami nenulovou a nulovou.

Pozn. pozor na rozdílnost v zápisech:

i = 3	přiřazení hodnoty proměnné
i == 3	porovnání pravé a levé strany

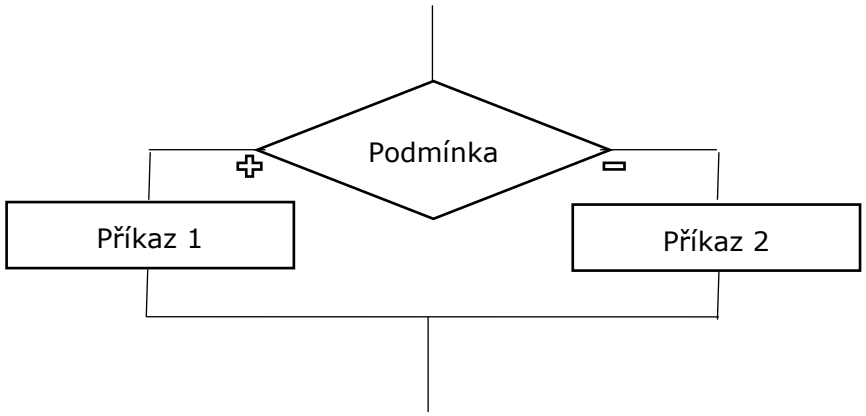
Rozhodovací příkaz if

Příkaz if se používá v případě, že je třeba provést jeden ze dvou bloků příkazů vybraných pomocí logické podmínky.

Tento příkaz je vyhodnocen podle toho, jaké je řešení logického výrazu (booleovského), na kterém závisí, jaká část programu se bude právě provádět. Tento výraz nabývá hodnoty pravda = true nebo nepravda = false.

Příkaz if

Příkaz používáme v případě, chceme-li, aby se jistý příkaz1 provedl v případě, že platí určitá podmínka, pokud ale neplatí, provede se příkaz2.

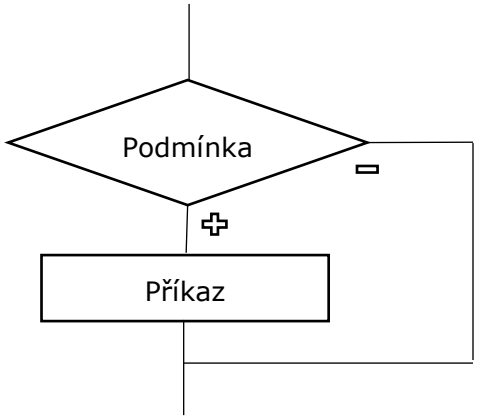


Syntaxe příkazu

```
if (výraz)
{
   příkaz1;
}
else
{
   příkaz2;
}
```

Příkaz může existovat i ve zkrácené podobě, pokud to zadání umožňuje. Pak neobsahuje klíčové slovo else a za ním uvedený příkaz2.

Příkaz používáme v případě, chceme-li, aby se jistý příkaz provedl v případě, že platí určitá podmínka, pokud neplatí, neprovede se nic.



Syntaxe příkazu

```
if (výraz)
{
   příkaz;
}
```

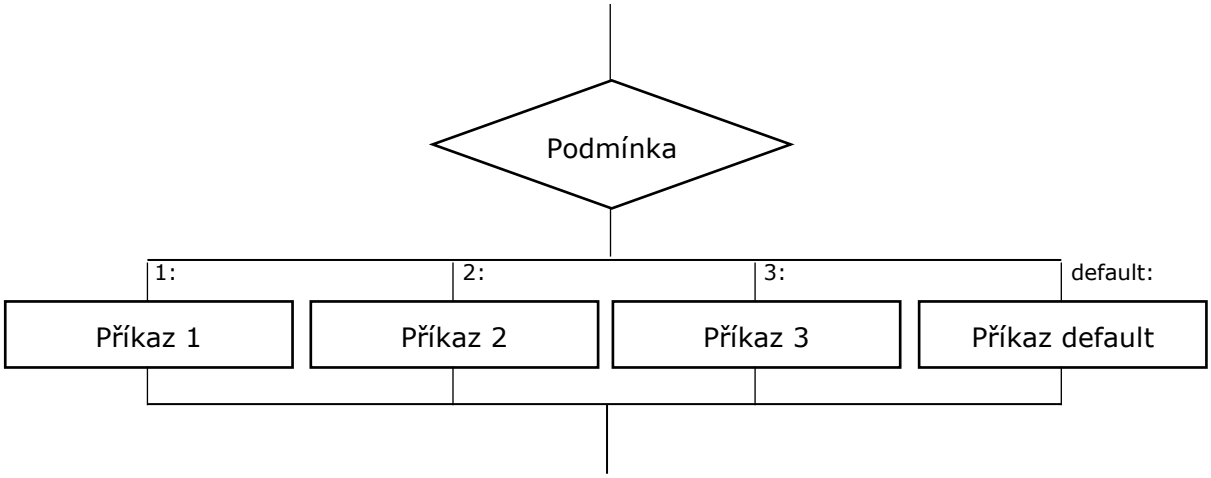
Příkaz switch

C obsahuje jakýsi přepínač neboli **switch**, který umožňuje mnohonásobné větvení programu.

Přepínač slouží k rozdělení posloupnosti příkazů na části, následné vybrání a provedení některé, či některých z nich.

Přepínač slouží k větvení výpočtu podle hodnoty **celočíselného výrazu**.

V jednom přepínači se nesmí používat dvě návěští se stejnou hodnotou.



Syntaxe příkazu

```
switch (vyraz)
{
   case 1:                 // Toto je jedna větev
      prikaz_1;            // ...
      break;               // ...
   case 2:                 // a zde začíná další větev
      prikaz_2;
      break;
   case 3:
      prikaz_3;
      break;
   default:
      prikaz_default;
      break;
 }
```

Po klíčovém slově switch následuje celočíselný výraz uzavřený v závorkách. Za ním je příkaz, zpravidla tvořený blokem.

Program porovnává vyraz s jednotlivými hodnotami každé větve a provede tu větev, kde se vyraz rovná hodnotě. Tzn. Nastane-li shoda hodnoty case návěští s hodnotou switch výrazu, je přeneseno řízení programu na toto návěští. Jinak je řízení přeneseno na návěští default v rámci příslušného přepínače. Pro návěští default platí stejné zásady jako pro jiná návěští.

Není-li větev ukončena příkazem break, program začne zpracovávat další větve v pořadí dokud nenarazí na break. Proto je třeba na konci každé větve psát break. Z tohoto ale

vyplývá, že pokud chceme pro více hodnot zpracovat pouze jednu větev, stačí vynechat příkaz break v těchto větvích.

Pokud nenapíšeme default návěští a hodnota výrazu switch se nebude shodovat s žádným z návěští, bude řízení přeneseno na příkaz následující za přepínačem.

Pokud program při provádění přepínače vykoná příkaz break, bude řízení přeneseno na příkaz následující za přepínačem.

Příklad:

```
switch (vyraz)
{
   case hodnota_1:
      prikaz_1;
      break;
   case hodnota_2:                  //Zde není break
   case hodnota_3:
      prikaz_23;
      break;
}
```

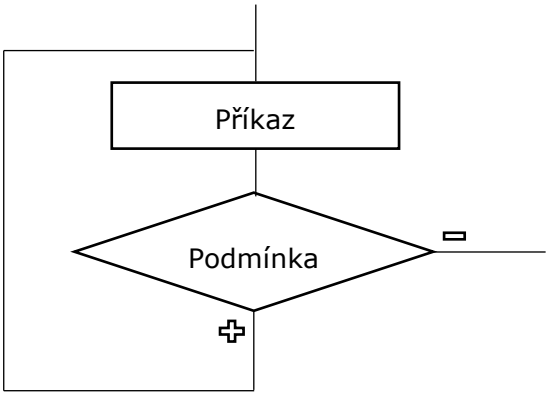
Prikaz_23 se provede, když je vyraz roven buď hodnote_2 nebo hodnote_3.

Cykly

umožňují opakování příkazů za předpokladu splnění zadaných podmínek.

do while

- příkaz s podmínkou na konci;
- zpracování cyklu je takové, že se provedou příkazy cyklu, pak se testuje podmínka cyklu, jejíž pravdivost znamená, že se cyklus opakuje znovu, tzn., že se opakuje provedení příkazů cyklu, nepravdivost znamená, že se cyklus ukončí, tzn.:
  provedou se příkazy cyklu, dále má-li podmínka - výraz hodnotu True = pravda, provedou se příkazy cyklu opakovaně, má-li podmínka - výraz hodnotu False = nepravda, cyklus je ukončen;
- příkaz se provede vždy nejméně jednou.

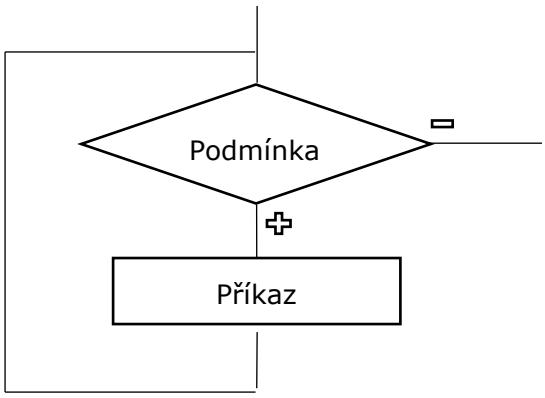


Syntaxe příkazu

```
do
{
prikaz;
}
while (podminka);
```

while

- příkaz s podmínkou na začátku;
- řešení cyklu spočívá v tom, že se testuje podmínka na začátku cyklu a podle toho, je-li pravdivá nebo nepravdivá se vykonají příkazy cyklu nebo je cyklus ukončen (bez provedení příkazů cyklu)
  tzn.:
  má-li podmínka - výraz hodnotu True = pravda, provedou se příkazy cyklu, má-li výraz hodnotu False = nepravda, vnořený příkaz se neprovede, cyklus je ukončen;
- příkaz se tedy nemusí provést ani jednou.

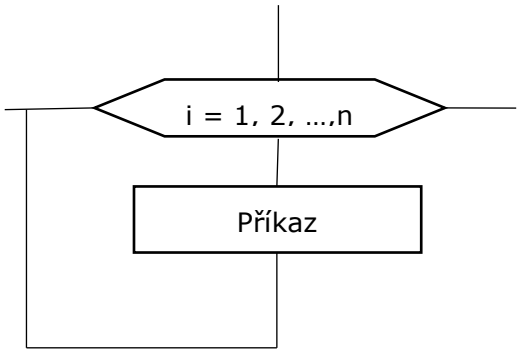


Syntaxe příkazu

```
while (podmínka)
{
   prikaz;
}
```

for

- příkaz for zajistí předem určený počet opakování příkazu;
- počet opakování musí být znám před spuštěním příkazu;
- používá se nejčastěji, pokud známe předem počet průchodů cyklem.



Syntaxe příkazu

```
for (vyraz_start; vyraz_stop; vyraz_iter)
{
   prikaz;
}
```

kde:

vyraz_start přiřadí řídicí proměnné cyklu počáteční hodnotu;

vyraz_stop je při každém průchodu cyklu vyhodnocen a je-li pravdivý, pak příkaz cyklu je proveden, jinak je cyklus ukončen;

vyraz_iter slouží ke změně hodnoty řídicí proměnné.

Příklad:

```
for (i = 1; i <= 10; i++)
{
   Console.WriteLine(i);
}
```

Tento příklad cyklu for zajistí vypsání hodnot:

..1..2..3..4..5..6..7..8..9..10

Pracuje tak, že na počátku se přiřadí proměnné i hodnota 1, příkaz cyklu je proveden, je-li hodnota proměnné i menší nebo rovna 10. Po každém průchodu cyklem je hodnota proměnné i zvětšena o 1. Cyklus je ukončen pro i = 11.

Ekvivalentní cyklus while:

```
i = 1;
while (i <= 10)
{
    Console.WriteLine(i);
    i++;
}
```

Příkaz skoku goto

- pokud kompilátor při překladu narazí na příkaz goto, pak následuje skok na uvedené návěští;
- pro provedení příkazu goto je zapotřebí zapsat příkaz samotný a návěští;
- v jazyce C je použití příkazu málo časté, používá se pouze výjimečně.

Syntaxe příkazu

```
.
x:                      //návěští
.
goto x;                 //příkaz skoku
.
.
```

nebo

```
.
goto y:                 //příkaz skoku
.
y:                      //návěští
.
```

Metody

V jazyce C# patří metody mezi základní výkonné mechanismy. Metodám v jiných programovacích jazycích odpovídají procedury nebo funkce.

Program v C# musí mít alespoň jednu metodu - metodu Main. Tato metoda je volaná kompilátorem a z ní lze volat další metody, se kterými si může předávat hodnoty nazývané vstupní parametry a návratovou hodnotu.

Deklarace metody

Každá metoda – deklarace metody - se skládá **z hlavičky a těla**.

Hlavička funkce obsahuje

- návratovýTyp;
- názevMetody;
- seznamParametrů.

Tělo funkce

- zapisuje se do složených závorek { };
- tvoří blok příkazů, na jehož začátku mohou být deklarace lokálních proměnných, dále obsahuje příkazy metody a podle návratového typu metody příkaz return.

Příklad

```
 návratovýTyp názevMetody (seznamParametrů)      } hlavička metody
{
  deklarace proměnných;
  prikazy;                                       } tělo metody
  return 0;
}
```

konkrétně

```
int soucet(int x1, int x2)                       } hlavička metody soucet
{
    int x;
    x=x1+x2;                                     } tělo metody
    return x;
}
```

Návratové typy metod

int, double, string, . . . – je typ metody s návratovou hodnotou, obsahuje příkaz return;
void - metoda bez návratové hodnoty, nevrací žádnou hodnotu, neobsahuje příkaz return.

Parametry

jsou typy a názvy informací, které si metody mohou předávat.

Volání metody

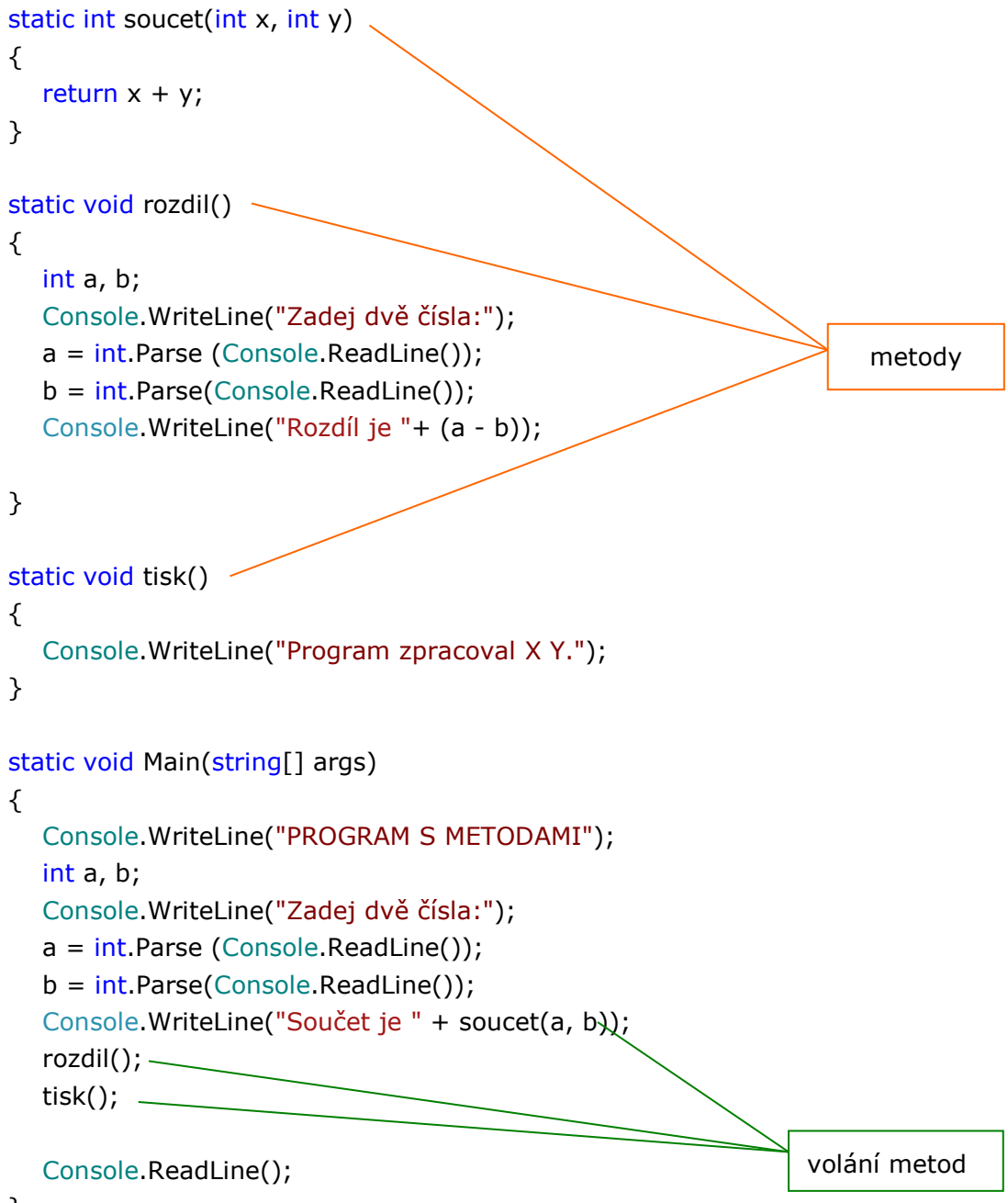
je zapotřebí použít, když chceme provést metodu.

Syntaxe volání metody
NazevMetody(seznamArgumentů)

Příklad volání metody
soucet(a,b);

Program s metodami

```
static int soucet(int x, int y)
{
   return x + y;
}

static void rozdil()
{
   int a, b;
   Console.WriteLine("Zadej dvě čísla:");
   a = int.Parse (Console.ReadLine());
   b = int.Parse(Console.ReadLine());
   Console.WriteLine("Rozdíl je "+ (a - b));

}

static void tisk()
{
   Console.WriteLine("Program zpracoval X Y.");
}

static void Main(string[] args)
{
   Console.WriteLine("PROGRAM S METODAMI");
   int a, b;
   Console.WriteLine("Zadej dvě čísla:");
   a = int.Parse (Console.ReadLine());
   b = int.Parse(Console.ReadLine());
   Console.WriteLine("Součet je " + soucet(a, b));
   rozdil();
   tisk();

   Console.ReadLine();
}
```



Správa chyb a výjimek

Pomocí správy chyb a výjimek se zajistí funkčnost programu v případě, že jsou na vstupu programu nebo v průběhu jeho zpracování zjištěny chyby.

Bloky try a catch

Pomocí výjimek a obslužných rutin se oddělí kód programu, který řeší zadanou úlohu, od kódu pro ošetření chyb.

Celý kód se uzavře do bloku **try**, za kterým následují bloky **catch**.

Dojde-li při zpracování kódu v bloku try k chybě, vyvolá se pomocí bloku catch výjimka, která zachytí a ošetří chybové stavy a vykonají se příkazy zapsané v tomto bloku.

Příklad

```
try
{
   blok s kódem, který chceme ošetřit
}
catch (FormatException X)          //rutina ošetřuje chyby datového typu na vstupu
{
   Console.WriteLine(X.Message);
)
catch (OverflowException X)        //rutina ošetřuje chyby přetečení
{
   Console.WriteLine(X.Message);
)
catch (Exception X)                //rutina ošetřuje všechny systémové chyby
{
   Console.WriteLine(X.Message);
)
```

Checked – unchecked

Tyto bloky se používají při sledování přetečení.

Blok checked kontroluje, zda nedošlo u zpracovávaného kódu k přetečení, pokud k němu dojde, vyvolá se výjimka a provede se zpracování kódu v příslušném bloku catch.

Pokud je sledování nastaveno na unchecked, pak dojde k přetečení a vznikne výsledek, který není správný.

Třídy výjimek

Object

Exception

SystemException

1. IndexOutOfRangeException
2. ArithmeticException
    a. DivideByZeroException – dělení nulou
    b. OverflowException - přetečení
3. InvalidCasrException

Pole

Pole je proměnná, která obsahuje prvky stejného datového typu. Prvky jsou uloženy v souvislém bloku paměti a k přístupu k nim se používá celočíselný index (nejčastěji i).

Deklarace pole

Deklarace pole obsahuje datový typ položek pole (int) a název pole (pole):

```
int[] pole;
```

Pro práci s polem je třeba vytvořit **instanci třídy** (pomocí klíčového slova new a určit velikost pole (počet položek pole) n:

```
pole = new int[n];
```

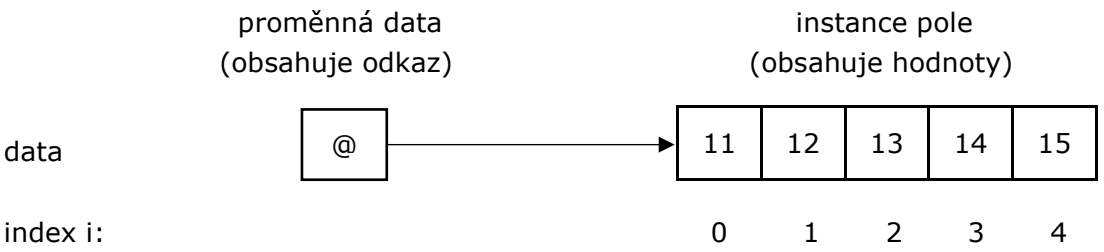
Oba příkazy lze zapsat v jednom:

```
int[] pole = new int[n];
```

Velikost pole nemusí být určena jen konstantou, ale může být vypočítána za běhu programu.

Příklad pole

Pole o 5 položkách:



Typy polí

jednorozměrné - položka pole je pole[i], kde i = 0, 1, . . , n – 1 (n je počet položek),

vícerozměrné - dvourozměrné - položka pole je pole[i, j],
kde i = 0, 1, . . , r – 1 (r je počet řádků)
kde j = 0, 1, . . , s – 1 (s je počet sloupců);
- trojrozměrné - položka je data[i, j, k].

Práce s jednorozměrným polem

Načtení hodnot do položek pole, velikost pole je určena hodnotou 10:

```
for (int i = 0; i < 10; i++)
{
   pole[i] = int.Parse(Console.ReadLine());
}
```

Velikost pole je určena hodnotou proměnné n načtenou z klávesnice:

```
int n = int.Parse(Console.ReadLine());
for (int i = 0; i < n; i++)
{
   pole[i] = int.Parse (Console.ReadLine());
}
```

Výpis položek pole

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

Výpis položek v opačném pořadí

```
for (int i = n - 1; i >= 0; i--)
{
   Console.WriteLine(pole[i]);
}
```

Inicializace pole

Každé položce určíme hodnotu přiřazením

```
pole[0] = 2;
pole[1] = -5;
pole[2] = 41;
pole[3] = 12;
pole[4] = 9;
```

Hodnotu určíme zápisem do závorky

```
pole = new int[5] { 22, 14, 3, 4, 5 };
```

Hodnotu určíme výpočtem

```
pole = new double[5] { a + 1, a / 2, b - 3, a / 4, b + 5 };
```

Dvourozměrné pole

Pole je proměnná, která obsahuje prvky stejného datového typu. Prvky jsou uloženy v souvislém bloku paměti a k přístupu k nim se používá celočíselný index (nejčastěji i). Jeli pole dvourozměrné, indexy jsou dva (nejčastěji i, j), u třírozměrného pole tři atd.

Deklarace pole

Deklarace pole obsahuje datový typ položek pole (int) a název pole (pole):

```
int[,] pole;
```

Pro práci s polem je třeba vytvořit **instanci pole** (pomocí klíčového slova new a určit velikost pole (počet řádků pole – r, počet sloupců pole s):

```
pole = new int[r, s];
```

Oba příkazy lze zapsat v jednom:

```
int[,] pole = new int[r, s];
```

Příklad dvourozměrného pole

je deklarováno pole

```
int[,] pole = new int[3, 4];
```

počet řádků: r = 3
počet sloupců: s = 4

pak každá položka má název pole[i, j]
kde index řádku: i = 0, 1, . . , r – 1 (r je počet řádků)
kde index sloupce: j = 0, 1, . . , s – 1 (s je počet sloupců)

Uspořádání položek v poli si představujeme takto:

j / i	0	1	2	3
0	-5	6	4	2
1	1	2	5	6
2	9	5	-12	6

Práce s dvourozměrným polem

Načtení hodnot do položek pole

```
Console.WriteLine("Zadej položky: " + (r * s));
for (i = 0; i < r; i++)
{
   for (j = 0; j < s; j++)
   {
      pole[i, j] = int.Parse(Console.ReadLine());
   }
}
```

Výpis hodnot

```
Console.WriteLine("Byla zadána čísla:");
```

```
for (i = 0; i < r; i++)
{
    for (j = 0; j < s; j++)
    {
        Console.Write(pole[i, j] + " ");
    }
    Console.WriteLine();
}
```

Příkaz foreach

foreach

- používá se pro práci s jednorozměrným polem
- prochází vždy celé pole (pokud je třeba pracovat jen s vybranými položkami, pak je lepší použít příkaz for)
- příkaz iteruje od indexu 0 do počet položek – 1, pro opačný směr musíme použít for
- pokud potřebujeme získat nějaký index položky, musíme použít for
- chceme-li měnit hodnoty položek prvků, musíme použít for.

Syntaxe příkazu

```
foreach (int pin in Number)          //(datový_typ název_proměnné in název_pole
{
   Console.WriteLine(pin);
}
```

Třídy a metody pro práci s polem

Setřídění hodnot pole Number vzestupně
Array.Sort(Number);

Setřídění hodnot pole Number sestupně
Array.Reverse(number);

Vyhledání maxima v poli
Number.Max();

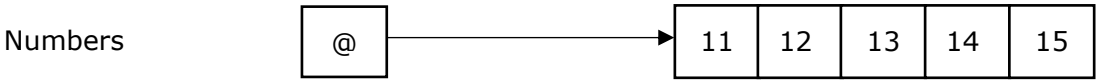
Vyhledání maxima v poli
Number.Min();

Kopírování polí

Kopírování polí lze provést několika způsoby.

Deklarujeme pole Numbers, vytvoříme instanci pole, zadáme hodnoty položek:

```
int pocet = 5;
int[] Numbers = new int[pocet];
```



Kopírování 1

Vytvoříme kopii Cisla pole Numbers:

```
int[] Cisla = Numbers;
```

Tímto příkazem zkopírujeme pouze proměnnou pole Numbers s jejím odkazem, tzn. že hodnoty položek pole Cisla budou vždy stejné jako hodnoty, protože odkazy v proměnných ukazují na stejnou instanci.

Kopírování 2

Vytvoříme nové pole Data o stejném počtu položek:

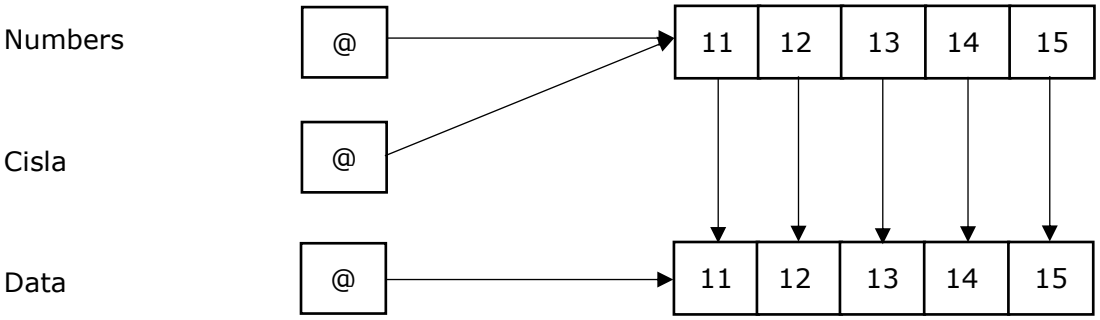
```
int[] Data = new int[pocet];
```

Každé položce pole Data přiřadíme položku se stejným indexem z pole Cisla:

```
for (int i = 0; i < pocet; i++)
{
   Data[i] = Cisla[i];
}
```

Vznikne pole Data, které má svoji instanci a je na poli Cisla nezávislé.

Výsledek kopírování 1 a 2:



Kopírování 3

Hodnoty z pole Numbers nakopírujeme do pole Data:

```
Numbers.CopyTo(Data, 0);
```

Kopírování 4

Vytvoříme kopii NumbersKopie pole Numbers:

```
int[] NumbersKopie = new int[Numbers.Length];
Array.Copy(Numbers, NumbersKopie, NumbersKopie.Length);
```

Kopírování 5

Vytvoříme klon (kopii) CislaKopie pole Cisla:

```
int[] CislaKopie = (int[])Cisla.Clone();
```

Výčty

Deklarace výčtu

Deklarace výčtu obsahuje klíčové slovo enum, za ním je uveden název výčtu a ve složených závorkách jsou, odděleny čárkou, uvedeny hodnoty výčtu:

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

Pokud chceme mít proměnnou den, která může nabývat hodnot z výčtu, nadeklarujeme ji:

```
tyden den = tyden.pondělí;
```

Třídy a metody

https://cs.wikipedia.org/wiki/Objektov%C4%9B_orientovan%C3%A9_programov%C3%A1n%C3%AD

https://www.itnetwork.cz/csharp/oop

Pojem třída patří mezi základní pojmy z teorie objektů. Třída je vzor, podle kterého se vytvářejí objekty. Říká, jaké atributy a metody objekt má.

Objekt, instance třídy (konkrétní příklad realizace třídy), má své atributy (vlastnosti, data, stavy) a metody (operace, které objekt provádí, schopnosti).

Při používání objektu nás zajímá, jaké operace (služby) poskytuje a ne, jakým způsobem to provádí.

Každá třída obsahuje alespoň jednu metodu. Např. třída Program obsahuje metodu Main.

Příklad

Třída

- Metoda_1
- Metoda_2
- Metoda_3

Program

- Main

Console

- Write
- WriteLine
- Readline

Math

- Pow
- Sin
- Cos
- Max
- Min

ArrayList

- Add
- Insert
- Remove
- RemiveAt

Řízení přístupnosti pro třídy a metody

private

datové složky a metody privat jsou přístupné jen metodám uvnitř této třídy, označení private není nutné psát, je nastaveno implicitně, pokud jej ale chceme zapsat, pak klíčové slovo private píšeme hned na začátek deklarace

public

datové složky a metody public jsou přístupné uvnitř i vně této třídy, v deklaraci píšeme klíčové slovo public hned na začátek deklarace

datové složky

jsou to vlastně deklarace proměnných, které jsou zapsány ve třídě, ne v metodě

Konstruktory

- je to speciální metoda, která se spustí automaticky při vytvoření instance nějaké třídy
- každá třída musí mít konstruktor - pokud nenapíšete vlastní, vygeneruje kompilátor automaticky implicitní konstruktor (který však nedělá vůbec nic)
- vlastní implicitní konstruktor napíšete velmi jednoduše: stačí přidat do třídy veřejnou (public) metodu, která nevrací žádnou hodnotu, a jejíž název je stejný jako název dané třídy
- konstruktor má stejný název jako třída a může přijímat parametry, nemůže však vracet žádnou hodnotu (ani void)
- provádí inicializaci vytvořeného objektu

Statické metody a data

umožňují přistupovat (volat) k metodě nebo datové složce přímo, bez vytváření instancí.

Třída Math obsahuje např. metody Sqrt, Sin, Cos, …, datovou složku PI.

Metoda deklarovaná jako statická může pracovat pouze se statickými datovými složkami.

Hodnotové a referenční typy

Hodnotové typy

Jsou typy int, float, double, char.

Při deklaraci se vytvoří kompilátorem kód, který vyčlení v paměti blok pro hodnotu proměnné podle jejího typu, např. int → 32 bitů.

Příklad:

```
int prom1 = 42;          // proměnné prom1 je přiřazena hodnota 42
```

Příklad vytvoření kopie proměnné hodnotového typu:

```
int prom1 = 42;          // proměnná prom1 obsahuje hodnotu 42
int prom2 = prom1;       // prom2 obsahuje také hodnotu 42
prom1 = 20;              // prom1 obsahuje hodnotu 20, prom2 42
```

Obě proměnné jsou na sobě nezávislé.

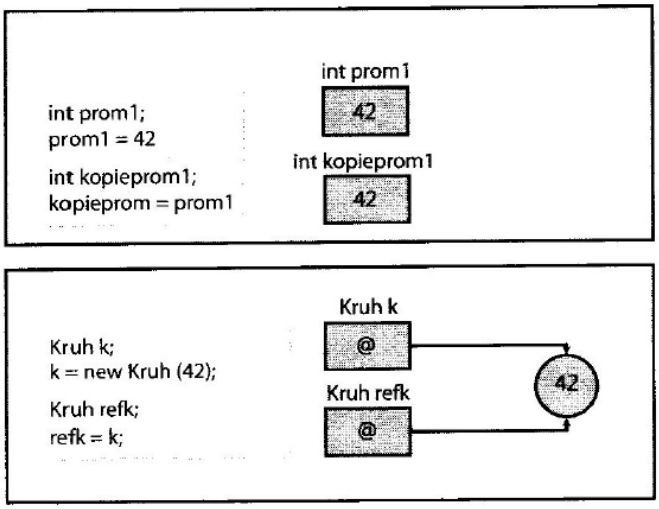
Referenční typy

Jsou typy string, třídy.

Při deklaraci třídy se nevytvoří blok, aby se do něj vešla celá třída, ale jen kousek paměti, do něhož je možné vložit adresu jiného bloku paměti, ve kterém bude instance třídy uložena.

U referenčního typu proměnné odkazují na instanci třídy.

Při vytváření kopie se kopíruje odkaz na blok v paměti, proto mají proměnné referenčního typu vždy stejnou hodnotu.



Obrázek 20: Vytvořrní kopie proměnné hodnotového a referenčního typu

Uspořádání paměti v počítači

Operační systémy a běhová prostředí rozdělují paměť pro ukládání dat na dva samostatné celky, které jsou řešeny odlišným způsobem: zásobník (stack) a halda (heap).

Při volání metody je paměť zapotřebí pro parametry a lokální proměnné, ta je brána ze zásobníku. Po ukončení metody (vrátí hodnotu nebo vyvolá výjimku) je přidělená paměť uvolněna pro další metody.

Při vytvoření objektu (instance třídy) pomocí klíčového slova new je paměť potřebná pro vytvoření objektu brána z haldy.

Poznámka: Všechny hodnotové typy se vytvářejí na zásobníku. Naproti tomu všechny referenční typy (objekty) jsou vytvářeny na haldě (ačkoli samotný odkaz je také v zásobníku). Nulovatelné typy jsou ve skutečnosti referenčními typy, takže se vytvářejí na haldě.

Hodnotové typy – zásobník

Referenční typy- halda

Názvy zásobník a halda byly zvoleny na základě způsobu, jakým běhové prostředí spravuje danou paměť:

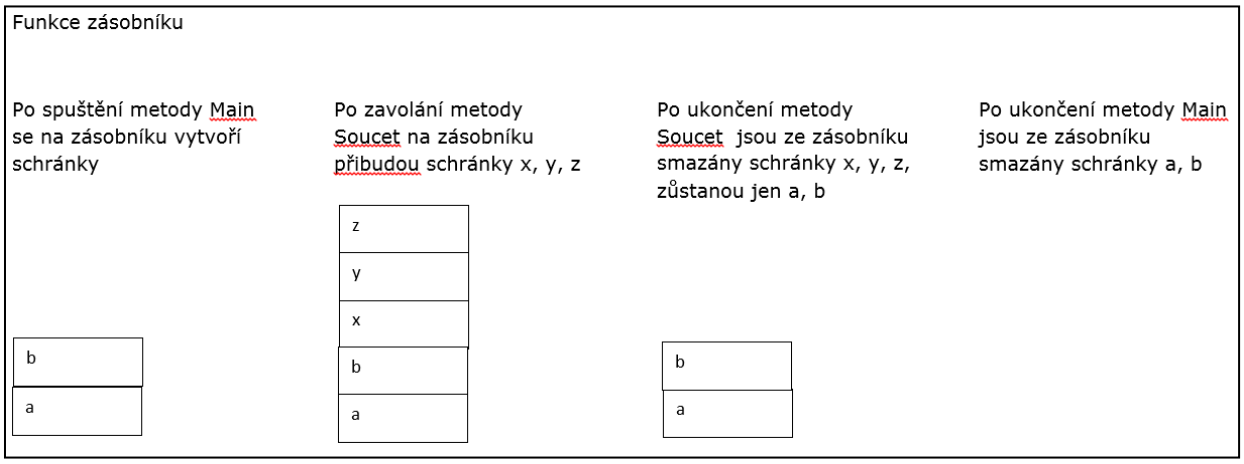
- Paměť zásobníku je uspořádána ve formě na sobě položených schránek. Při volání metody je každý parametr umístěn do " schránky", která je umístěna na vrchol zásobníku. Obdobně je i každá lokální proměnná vložena do své schránky a ta je také umístěna na vrchol zásobníku, nad všechny ostatní schránky, které tam již jsou. Když volání metody skončí, jsou všechny schránky ze zásobníku odebrány.

- Paměť na haldě připomíná velkou haldu (hromadu) tvořenou volně poházenými schránkami. Každá schránka má jmenovku, na které je napsáno, zda je používána nebo ne. Když je vytvářen nový objekt, běhové prostředí vyhledá prázdnou schránku a přidělí ji objektu. Odkaz na objekt je uložen do lokální proměnné v zásobníku. Běhové prostředí sleduje počet odkazů na každou schránku (vzpomeňte si, že na jeden objekt může ukazovat více proměnných). Když zanikne poslední odkaz, označí běhové prostředí schránku jako nepoužitou a někdy poté ji vyprázdní a uvolní pro další použití.

Příklad funkce zásobníku

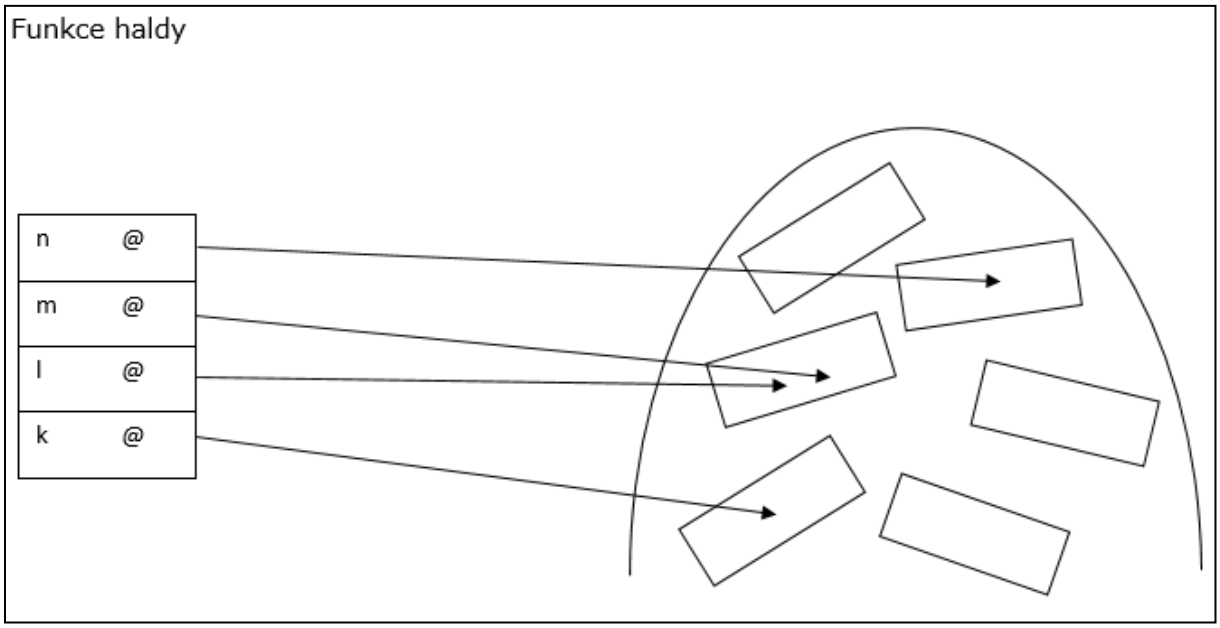
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApplication1
{
   class Program
   {
      static void Soucet(int x, int y)
      {
         int z = x + y;
         Console.WriteLine("Součet je: " + z);
      }

      static void Main(string[] args)
      {
         Console.WriteLine("PROGRAM S METODAMI");
         int a, b;
         Console.WriteLine("\nSOUČET ČÍSEL:");
         Console.WriteLine("Zadej dvě čísla:");
         a = int.Parse(Console.ReadLine());
         b = int.Parse(Console.ReadLine());
           Soucet(a, b);

         Console.ReadLine();
      }
   }
}
```

Příklad zásobníku



Obrázek 21: Ukládání proměnných na zásobníku

Příklad haldy



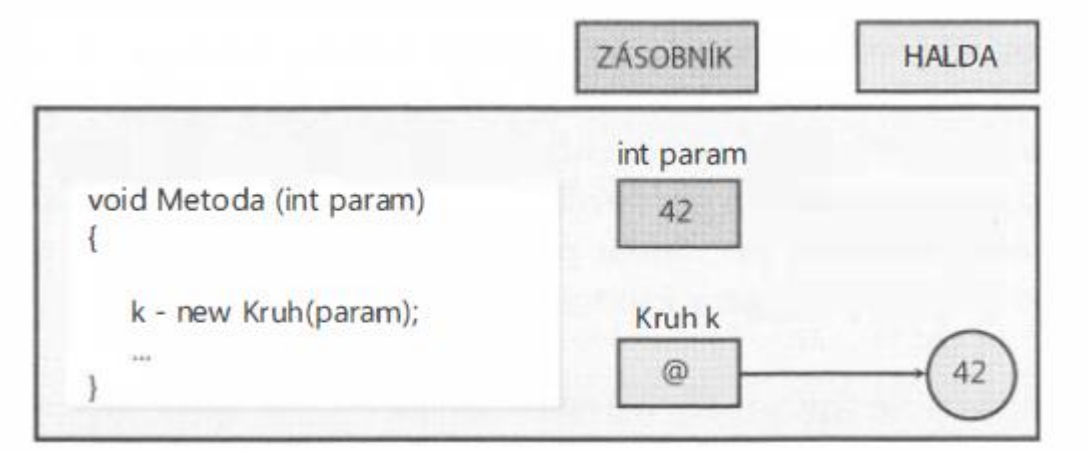
Obrázek 22: Ukládání proměnných na haldě

Jak používat zásobník a haldu

Podívejme se nyní, co se stane při volání následující metody:

```
void Metoda (int param)
{
   Kruh k;
   k = new Kruh(param);
   . . .
}
```

Dejme tomu, že parametru param je předána hodnota 42. Když dojde k zavolání metody, přidělí se ze zásobníku blok paměti (dostatečně velký pro typ int) a je inicializován číslem 42. Jak pokračuje provádění uvnitř metody, je ze zásobníku opět přidělena část paměti, do kterého se vejde odkaz (adresa v paměti), ale není prozatím inicializován (to je pro proměnnou k). Klíčové slovo new přidělí další kousek paměti, tentokrát z haldy, a to tak velký, aby se do něj vešel celý objekt typu Kruh. Konstruktor třídy Kruh převede tuto surovou paměť z haldy na objekt typu Kruh. Odkaz na objekt typu Kruh se uloží do proměnné k. Celou situaci ilustruje následující schéma:



Obrázek 23: Vztah mezi zásobníkem a haldou

Na tomto obrázku jsou zajímavé dvě věci:

1. Přestože objekt je uložen na haldě, odkaz na tento objekt (proměnná k) je uložen v zásobníku.
2. Paměť na haldě není nekonečná. Pokud dojde, vyvolá operátor new výjimku typu OutOfMemoryException a objekt vytvořen nebude.

Poznámka: Konstruktor třídy Kruh může také vyvolat výjimku. Pokud k tomu dojde, bude paměť přidělená objektu Kruh uvolněna a konstruktor vrátí prázdný odkaz (null).

Když metoda skončí, přestanou parametry a lokální proměnné platit. Paměť přidělená proměnné k a parametru param bude automaticky vrácena zpět do zásobníku. Běhové prostředí zaznamená, že na daný objekt typu Kruh již neexistují žádné odkazy, a v blízké budoucnosti objekt zruší a paměť vrátí zpět do haldy.

Předávání parametru hodnotou a referencí

Předávání parametru hodnotou

```
void Zamen(int a, int b) //zameni hodnoty v promennych
{
   int pom;
   pom = a;
   a = b;
   b = pom;
}
```

Tuto metodu budeme volat v programu následně:

```
static void Main(string[] args)
{
   int promennaA = 10, promennaB = 20;
   Zamen(promennaA, promennaB);
   return 0;
}
```

Předávání parametrů metodě hodnotou má výsledek, že **promennaA** a **promennaB** budou po zavolání metody beze změny.

Předávání odkazem

```
void Zamen(ref int a, ref int b) //zameni hodnoty v promennych
{
   int pom = a;
   a = b;
   b = pom;
}
```

Tuto metodu budeme volat v programu následně:

```
static void Main(string[] args)
{
   int promennaA = 10, promennaB = 20;
   Zamen(ref promennaA, ref promennaB);
   return 0;
}
```

Po předání parametrů metodě tímto způsobem dosahujeme výsledku, že po zavolání metody budeme mít v našich proměnných **promennaA** a **promennaB** hodnotu 20 resp. 10.

Jazyk C# umí další možnosti jak předávat parametry metodě. Je zde možnost proměnného počtu parametrů funkce a také možnost použití modifikátoru out, který je blízký modifikátoru ref.

Unikání paměti

Únik paměti

(anglicky memory leak) označuje v informatice situaci, kdy počítačový program neúmyslně alokuje operační paměť a není ji schopen uvolnit poté, co ji již dále ani nepotřebuje ani nevyužívá. Únik paměti nastává vinou chyby v programu, která uvolnění již nevyužívané paměti zabraňuje. Negativní vliv se zvyšuje, pokud dochází k opakovanému úniku paměti, což může způsobit zřetelné zpomalení počítače nebo dokonce až vyčerpání veškeré dostupné paměti (OOM), které může nakonec způsobit násilné ukončení programů nebo i fatální selhání počítače. V reálném světě by se únik paměti dal přirovnat k člověku, který použité věci neustále uschovává, i když je už nikdy nebude potřebovat.

Za únik paměti jsou někdy označovány i jiné situace (viz níže), avšak takové označení není přesné (například když program jen vyžaduje pro svůj běh neadekvátní množství paměti). Oprava úniku paměti může být většinou diagnostikována a provedena pouze programátorem s přístupem ke zdrojovému kódu chybného programu.

Příčiny

Při programování jsou úniky paměti poměrně běžnou chybou, obzvlášť v případě programovacích jazyků, neobsahujících zabudovanou automatickou správu paměti, jako jsou například C a C++. Programátor musí v těchto jazycích zajistit uvolňování veškeré alokované paměti vlastním kódem, a proto k chybám a opomenutím dochází. Chybějící automatickou správu paměti je však možné i do těchto programovacích jazyků doplnit použitím vhodných knihoven.

Automatická správa paměti

Proti paměťovým únikům nejsou imunní ani jazyky vybavené automatickou správou paměti (garbage collector), jako jsou Java, C#, VB.NET nebo LISP. Jednoduchým příkladem budiž seznam, do něhož program vkládá data, ale po použití je již nesmaže, i když je už dále nebude potřebovat. Garbage collector sám nepozná, jestli program ještě bude v budoucnu uložená data znovu používat. Proto musí sám program indikovat další nepotřebnost uložených dat. Obvykle se to dělá tak, že program řekne "dealokuj!" nebo když je líny, tak spoléhá na to, že jsou odstraněny všechny odkazy (reference) na daná data (resp. na použité části paměti).

Automatická správa paměti uvolňuje pro další použití již nedosažitelnou a tedy zbytečně alokovanou paměť. Nemůže však uvolnit paměť, která obsahuje data stále dosažitelná a potenciálně využitelná. Moderní správa paměti proto poskytuje programátorům možnost sémanticky označit paměť s různou úrovní užitečnosti, odpovídající její dosažitelnosti. Správce tak neuvolní silně dosažitelný objekt, což znamená objekt dosažitelný přímo silnou referencí, nebo nepřímo několika silnými referencemi (silná reference na rozdíl od slabé brání garbage collectoru odstranit odkazovaný objekt). Takovémuto úniku paměti musí zabránit programátor – obvykle nastavením dané reference na null v okamžiku, kdy již není dále potřeba, a případně odregistrováním všech případných event listenerů, udržujících na objekt silnou referenci.

Automatická správa paměti je pro vývojáře obecně robustnější a spolehlivější, neboť není třeba implementovat paměť uvolňující rutiny (jako kdyby nebyly v garbage collektoru) či starat se o případné zbylé reference na čištěné objekty. Pro programátora je jednodušší hlídat, kdy není reference již víc potřebná, než zda už objekt není referencován. Na druhou stranu automatická správa může zvyšovat náročnost na výkon systému a neumí odstranit všechny programové chyby, způsobující únik paměti.

Bibliografie

SHARP, John. 2008. *Microsoft Visual C# 2008: krok za krokem.* [překl.] Jaroslav Černý Lukáš Krejčí. Brno : Computer Press, a. s., 2008. str. 592. ISBN 978-80-251-2027-9.