Cvičenia z programovania

Autor: **Jaroslav Synek**

20. septembra 2024

Discord Server
https://discord.gg/vntVVG26wd

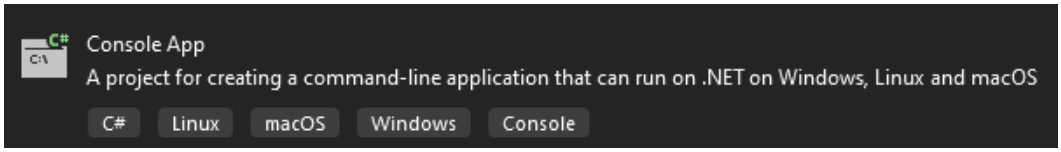
Obsah

1 Zoznámenie sa s IDE, premenné, dátové typy, pretypovanie, operátory

1.1 IDE (Integrované vývojárske rozhranie)

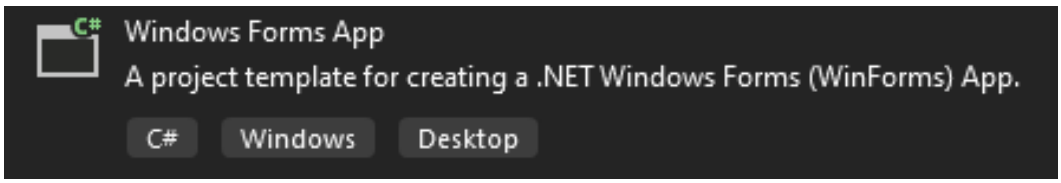
Tento školský rok sa budeme v prvom polroku zaoberať **objektovo orientovaném programovaním** (OOP) a v druhom polroku prejdeme na prácu s **grafickým rozhraním** (GUI). Využívať na to budeme jazyk **C#**. V tomto dokumente je využívané **Visual Studio 2022**, ale existujú aj iné možnosti (napríklad **JetBrains Rider**).

Ako bolo spomínané, na začiatok budeme tvoriť konzolové aplikácie - bez akéhokoľvek grafického rozhrania, všetok vstup/výstup sa bude riešiť cez konzolu. Pre vytvorenie konzolovej aplikácie vo Visual Studio 2022 vyberieme pri tvorbe projektu šablónu **Console App** (POZOR! chceme C#, nie C++) ako je vidno na obrázku.



Obr. 1: Tvorba konzolovej aplikácie

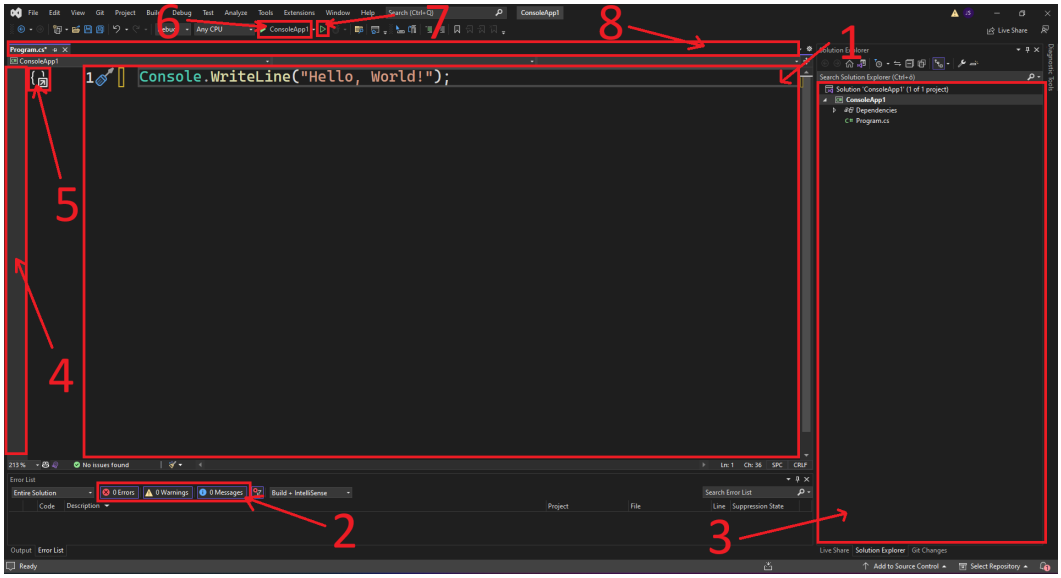
Keď prejdeme do druhej časti, budeme využívať šablónu **Windows Forms App**. Ako je už z názvu zrejmé, budeme potrebovať využívať **Windows** (na macOS alebo Linuxe to nanešťastie nerozbehneme).



Obr. 2: Tvorba WinForms aplikácie

Vo Visual Studiu sa nachádza množstvo rôznych panelov, z ktorých sa oplatí rozumieť aspoň niektorým.

1. **Textový editor** - Tu píšeme všetok kód.
2. **Errors/Warnings/Messages**
   - Error je chyba, pre ktorú nie je možné kód ani len preložiť.
   - Warning je chyba, ktorá možno ani nie je chybou, ale prekladač nie je sto-percentne spokojný, preklad sa však aj napriek tomu podarí.
   - Message je len správa, môže sa jednať o nejakú dôležitú informáciu.
3. **Solution Explorer** - Tu môžeme vidieť všetky (viditeľné) súbory a adresáre projektu.
4. **Miesto na breakpointy** - Breakpointy su dôležité pri debuggovaní. V skratke to je miesto, kde sa program zastaví pri spustení s debuggerom a z ktorého následne môže pokračovať.
5. **Na pozadí importované knižnice**
6. **Spustiť s debuggerom**
7. **Spustiť (bez debuggera)**
8. **Otvorené súbory**

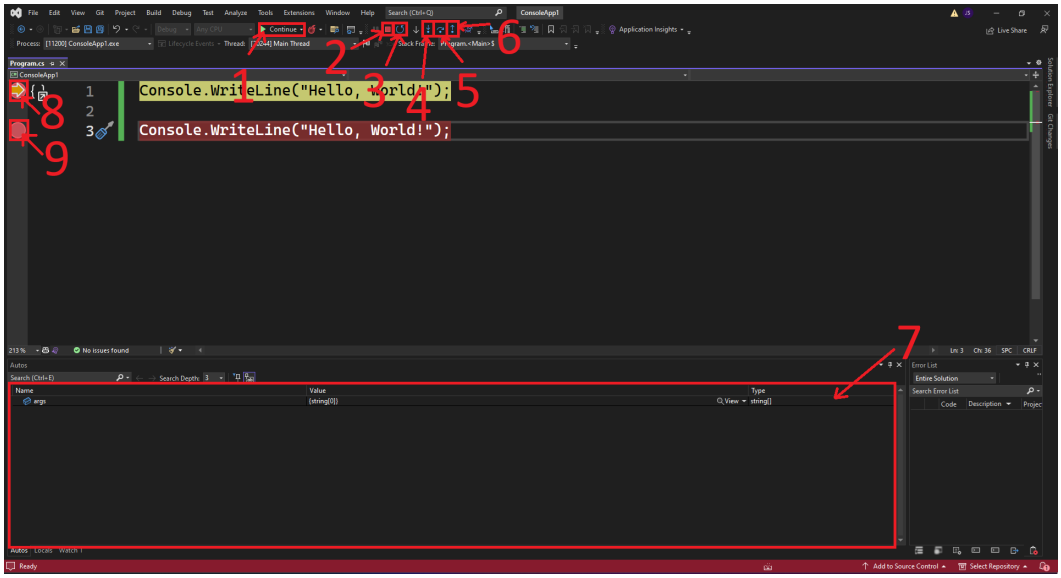


Obr. 3: IDE

1.1.1 Debugger

Debugger je nástroj, ktorý nám uľahčuje proces hľadania logických chýb (takých, ktoré prekladač nevie odchytiť, pretože nevie, že to sú chyby). Dôležitou súčasťou sú napríklad krokovanie, breakpointy, monitor premenných...

1. **Pokračovať** - Po kliknutí pustíme program ďalej a opäť sa zastaví až na najbližšom breakpointe.
2. **Ukončiť** - Ukončíme debugger.
3. **Znovu spustiť** - Ukončíme a znovu spustíme debugger, môže sa hodiť keď skúšame niečo opraviť a chceme vidieť, že aký vplyv mala zmena na premenné, prípadne nejaké výpisy.
4. **Vojsť dnu** - Aj volanie funkcie je len príkaz, pokiaľ chceme vidieť, čo sa deje vo funkcií, môžeme do nej vojsť a krokovať si samotnú funkciu, v opačnom prípade by sa celá funkcia vykonala v jednom kroku.
5. **Ďalší krok** - Vykoná ďalší príkaz.
6. **Výjsť von** - Pokiaľ sme vošli do funkcie a nechceme už ju krokovať až dokonca, tak týmto z nej môžeme rovno výjsť von.
7. **Monitor premenných** - Môžeme vidieť všetky aktuálne dostupné premenné a čo sa v nich nachádza.
8. **Šípka** - Ukazuje, ktorý príkaz sa vykoná v ďalšom kroku. V tomto prípade je kombinovaná s breakpointom.
9. **Breakpoint** - Miesto, kde sa program zastaví, ak máme spustený debugger.



Obr. 4: Debugger

1.2 Premenné

Premenná je niečo, čo nám umožňuje ukladať **dáta** za behu programu. Na vytvorenie premennej ju potrebujeme **deklarovať** a následne **definovať**.

Deklarácia premennej v podstate znamená, že počítaču povieme, nech si v pamäti zaberie určité množstvo bajtov (závisí od veľkosti **dátového typu**) a že od tohto bodu budeme toto miesto v pamäti referovať názvom tejto premennej. Premenné by mali mať výstižné (ale nie až príliš dlhé) názvy. Názov premennej môže obsahovať iba **číslice** (0-9), **malé a veľké písmená** (A-Z,a-z) a podčiarkovník (_). Pritom názov premennej nemôže začínať číslicou.

Definícia premennej je samotné priradenie nejakej hodnoty do premennej, teda premenná pred jej definíciu už musí byť deklarovaná.

Deklaráciu vykonávame **iba raz**, definíciu môžeme opakovať **neobmedzene** - len sa prepíše hodnota v tejto premennej.

```
1 int moje_cislo;     // deklaracia -> <datovy_typ> <nazov_premennej>;
2 moje_cislo = 7;     // definicia  -> <nazov_premennej> = <hodnota>;
```

Keďže často už pri vytvorení premennej chceme do nej priradiť nejakú hodnotu, môžeme deklaráciu a definíciu spojiť do jednoho.

```
1 int moje_cislo = 7;     // vysledok bude rovnaky ako v predchadzajucom priklade
```

Pokiaľ vieme, že premenná bude mať počas celej svojej existencie **rovnakú** hodnotu a nechceme riskovať, že ju niekto omylom zmení, vieme túto premennú označiť ako **konštantu**. Pokiaľ je premenná označená ako konštanta, hocijaký pokus o prepísanie jej hodnoty bude braný ako **chyba**.

```
1 const int moja_konstanta = 10;
2 moja_konstanta = 20;     // ERROR - nemozno menit hodnotu konstanty
```

1.3 Dátové typy

Názov	Kľúčové slovo	Rozsah (Presnosť pri desatinných číslach)	Veľkosť v bitoch
Boolean	bool	true | false	8
Byte	byte	0 — 255	8
Signed Byte	sbyte	-128 — 127	8
Char	char		16
Decimal	decimal	28-29 číslic	128
Double	double	15-17 číslic	64
Float	float	6-9 číslic	32
Integer	int	-2 147 483 648 — 2 147 483 647	32
Unsigned Integer	uint	0 — 4 294 967 295	32
Native Integer	nint		závisí na platforme
Native Unsigned Integer	nuint		závisí na platforme
Long Integer	long	-9 223 372 036 854 775 808 — 9 223 372 036 854 775 807	64
Unsigned Long Integer	ulong	0 — 18 446 744 073 709 551 615	64
Short Integer	short	-32 768 — 32 767	16
Unsigned Short Integer	ushort	0 — 65 535	16
Object	object		
String	string		
Dynamic datatype	dynamic		
Void	void	—	0

Typy, ktoré je potrebné poznať:

```
1 bool   b = false;   // hodnota: bud true alebo false
2 char   c = 'A';     // hodnota: znak
3 float  f = 0.1f;    // hodnota: desatinne cislo (moze byt zaporne)
4 int    i = 64;      // hodnota: cele cislo (moze byt zaporne)
5 string s = "text";  // hodnota: text
```

1.4 Pretypovanie

Pretypovanie je zmena dátového typu nejakého výrazu na iný (POZOR! Výraz bude takého dátoveho typu, akým je samotná **premenná**).

Pokiaľ prevádzame z menšieho dátového typu na väčší, môžeme proste hodnotu menšej premennej **implicitne** priradiť do väčšej.

```
1 float cislo1 = 0.5f;           // vytvorime si premennu a priradime do nej hodnotu 0.5
2 double cislo2 = cislo1;        // prevadzame z floatu na double
3 Console.WriteLine(cislo2);     // >>> 0.5
```

Problém by nastal, kebyže sa o to pokúsime opačne.

```
1 double cislo1 = 0.5f;          // vytvorime si premennu a priradime do nej hodnotu 0.5
2 float cislo2 = cislo1;         // ERROR - nemozno implicitne pretypovat
3 Console.WriteLine(cislo2);
```

Tento problém však vieme vyriešiť **explicitným** pretypovaním.

```
1 double cislo1 = 0.5f;          // vytvorime si premennu a priradime do nej hodnotu 0.5
2 float cislo2 = (float)cislo1;  // explicitne pretypujeme na float
3 Console.WriteLine(cislo2);     // >>> 0.5
```

POZOR! Pri explicitnom pretypovaní sa môže stať, že hodnota v premennej bude iná, ako by sme očakávali. Tento problém nastáva, pokiaľ sa táto hodnota **nezmestí** do menšej premennej.

```
1 int cislo1 = 50000;            // vytvorime si premennu a priradime do nej hodnotu 50 000
2 short cislo2 = (short)cislo1;  // explicitne pretypujeme na short (rozsah -32 768 az 32 767)
3 Console.WriteLine(cislo2);     // >>> -15536
```

Ďalší problém nastáva, pokiaľ chceme pretypovať z **nekompatibilného** dátového typu.

```
string cislo1 = "73";          // vytvorime premennu a priradime do nej 73 (ale ako retazec)
int cislo2 = (int)cislo1;      // ERROR - prekladac nevie konvertovat string na int
Console.WriteLine(cislo2);
```

Tento problém však môžeme vyriešiť viacerými spôsobmi. Prvý spôsob je použitie **Convert.ToInt32()** funkcie.

```
string cislo1 = "73";                          // vytvorime premennu a priradime do nej 73
int cislo2 = Convert.ToInt32(cislo1);          // pretypujeme na int
Console.WriteLine(cislo2);                     // >>> 73
```

Alternatívne môžeme použiť **int.Parse()** funkciu. Rozdielom oproti Convert.ToInt32() funkcii je ten, že int.Parse() v prípade, že ako argument dostane null vyhodí výnimku, zatiaľ čo Convert.ToInt32() vráti 0.

```
string cislo1 = "73";                          // vytvorime premennu a priradime do nej 73
int cislo2 = int.Parse(cislo1);                // pretypujeme na int
Console.WriteLine(cislo2);                     // >>> 73
```

Tieto možnosti fungujú fajn, kým reťazec obsahuje iba číslice a nie iné znaky. Preto by sme potrebovali, aby ak užívateľ má zadať číslo, tak aby to naozaj bolo číslo a nie **náhodné znaky**. Ošetriť sa to dá funkciou **int.TryParse()**, ktorá funguje ako int.Parse(), ale okrem hodnoty vracia aj **status**, či sa pretypovanie podarilo.

```
string cislo1 = "73";                          // vytvorime premennu a priradime do nej 73
int cislo2;                                    // vytvorime premennu na vysledok
bool status = int.TryParse(cislo1, out cislo2);  // vysledok sa ulozi do premennej cislo2
                                               // status sa ulozi do premennej status
Console.WriteLine(cislo2);                     // >>> 73
Console.WriteLine(status);                     // >>> True
```

1.5 Operátory

Operátor	Operácia	Príklad použitia	Výsledok príkladu
`++`	inkrementácia premennej	`x++`	najskôr použije x, potom ho inkrementuje
		`++x`	najskôr inkrementuje x, potom ho použije
			ekvivalentné `x = x + 1`
`--`	dekrementácia premennej	`x--`	najskôr použije x, potom ho dekrementuje
		`--x`	najskôr dekrementuje x, potom ho použije
			ekvivalentné `x = x - 1`
`-`	airtmetická negácia	`-x`	zmení znamienko čísla x ($+ \mapsto -$; $- \mapsto +$)
`~`	bitová negácia	`~x`	zneguje všetky bity v x ($0 \mapsto 1$; $1 \mapsto 0$)
`!`	logická negácia	`!x`	prevráti pravdivostnú hodnotu x (true $\mapsto$ false ; false $\mapsto$ true)
`+`	sčítanie / konkatenácia	`x + y`	sčíta x a y / spojí dva reťazce
`-`	odčítane	`x - y`	odčíta y od x
`*`	násobenie	`x * y`	vynásobí x a y
`/`	delenie	`x / y`	vydelí $x \div y$
`%`	modulo	`x % y`	zvyšok po vydelení $x \div y$
`==`	rovnosť	`x == y`	pravdivostná hodnota, či sa x a y rovnajú
`!=`	nerovnosť	`x != y`	pravdivostná hodnota, či sa x a y lýšia
`<`	menší ako	`x < y`	pravdivostná hodnota, či x je menšie ako y
`>`	väčší ako	`x > y`	pravdivostná hodnota, či x je väčšie ako y
`<=`	menší alebo rovný ako	`x <= y`	pravdivostná hodnota, či x je menšie ako y alebo sa mu rovná
`>=`	väčší alebo rovný ako	`x >= y`	pravdivostná hodnota, či x je väčšie ako y alebo sa mu rovná
`&&`	logický AND	`x && y`	pravdivostná hodnota, či sú aj x, aj y pravdivé
`||`	logický OR	`x || y`	pravdivostná hodnota, či je aspoň jedno z x a y pravdivé
`&`	bitový AND	`x & y`	porovnáva x a y bitovo, v oboch 1 $\implies$ bit výsledku = 1, inak 0
`|`	bitový OR	`x | y`	porovnáva x a y bitovo, v oboch 0 $\implies$ bit výsledku = 0, inak 1
`^`	bitový XOR	`x ^ y`	porovnáva x a y bitovo, odlišné $\implies$ bit výsledku = 1, inak 0
`<<`	bitový posun vľavo	`x << y`	posúva bity vľavo v x o y miest
`>>`	bitový posun vpravo	`x >> y`	posúva bity vpravo v x o y miest
`=`	priradenie hodnoty	`x = y`	nastavenie x na y
`+=`	zväčšenie premennej o	`x += y`	ekvivalentné `x = x + y`
`-=`	zmenšenie premennej o	`x -= y`	ekvivalentné `x = x - y`
`*=`	násobenie premennej	`x *= y`	ekvivalentné `x = x * y`
`/=`	delenie premennej	`x /= y`	ekvivalentné `x = x / y`
`%=`	ponechanie zvyšku delenia	`x %= y`	ekvivalentné `x = x % y`
`?`	ternárny operátor	`x = a ? y : z`	ekvivalentné kódu: if (a) x = y; else x = z;

Kompletný zoznam operátorov je možné nájsť v dokumentácii jazyka C# od Microsoftu.

Poznámka: Priradenie sa skladá z 3 častí: **L_HODNOTA = R_HODNOTA**. R_HODNOTA môže byť výraz (teda môže byť premenná, číslo a ich kombinácie pomocou operátorov). L_HODNOTA s pravidla nemôže byť výraz (teda na mieste L_HODNOTY musí byť jedine premenná). Z toho dôvodu dekrementácia a inkrementácia taktiež nemôže byť L_HODNOTOU, pretože premenná pri dekrementácii a inkrementácii je zároveň použitá ako L_HODNOTA aj R_HODNOTA.

2 Vetvenie, cykly, podmienky

2.1 Vetvenie

V programoch sa nie vždy vykonávajú všetky časti kódu v závislosti od niečoho. Na to sa používajú **podmienky**.

2.1.1 If

Základnú podmienku **if** si môžeme ukázať na príklade. Uvažujme program, ktorý vypíše absolútnu hodnotu čísla zadaného užívateľom.

```
string vstup = Console.ReadLine();                    // nacitame vstupne data z konzole
int cislo = Convert.ToInt32(vstup);                   // konvertujeme vstup na integer
if (cislo < 0)                                        // pokial je cislo mensie ako nula
{
    cislo = -cislo; // iba ak je podmienka splnena    // otocime mu znamienko
}
Console.Write("absolutna hodnota cisla je " + cislo); // vypis
```

Poznámka: pokiaľ sa v blokových (množinových) zátvorkách nachádza iba jeden príkaz, je možné ich **vynechať**.

Podmienok môžeme v kóde mať aj viacero. Uvažujme program, ktorý na vstupe opäť načíta nejaké číslo a vypíše, či je párne alebo nepárne.

```
int cislo = Convert.ToInt32(Console.ReadLine());     // nacitanie a konverzia na integer
if (cislo % 2 == 0)                                  // ak zvysok po deleni dvomi je nula
    Console.Write("cislo je parne");
if (cislo % 2 != 0)                                  // ak zvysok po deleni dvomi nie je nula
    Console.Write("cislo je neparne");
```

Ikeď kód ako taký funguje, má v sebe ukryté dva problémy. Oba súvisia s tým, že jedna podmienka je len **negáciou** tej druhej. Prvým problémom je to, že aj pokiaľ je prvá podmienka splnená, stále sa testuje aj druhá podmienka, ktorá logicky **nikdy nemôže byť splnená**, pokiaľ bola splnená prvá z nich. Druhým problémom je to, že **negácia výrazu v podmienke** nemusí byť taká jednoduchá, ako tomu bolo v tomto prípade, čo vedie k horšie čitateľnému kódu a väčšej náchylnosti na vznik chýb.

Riešením je pridanie **else** za **if**. Tým pádom, pokiaľ je podmienka splnená, vykoná sa blok za if, pokiaľ nie je, vykoná sa blok za else a nič sa nemusí opäť porovnávať.

Rovnaký program v lepšom prevedení by vyzeral takto.

```
int cislo = Convert.ToInt32(Console.ReadLine());     // nacitanie a konverzia na integer
if (cislo % 2 == 0)                                  // ak zvysok po deleni dvomi je nula
    Console.Write("cislo je parne");
else                                                 // ak zvysok po deleni dvomi nie je nula
    Console.Write("cislo je neparne");
```

Podmienky možno aj reťaziť (dať podmienku do podmienky). Uvažujme porgram, ktorý vypíše, či je číslo na vstupe kladné, záporné alebo nula.

```
int cislo = Convert.ToInt32(Console.ReadLine());
if (cislo > 0)
    Console.Write("cislo je kladne");
else
{
    if (cislo < 0)
        Console.Write("cislo je zaporne");
    else
        Console.Write("cislo je nula");
}
```

Keďže else nemôže existovať bez if, považuje sa **if...else** ako 1 príkaz a tým pádom ani tu nepotrebujeme zátvorky, čo nás posúva k prehľadnejšiemu kódu.

```
int cislo = Convert.ToInt32(Console.ReadLine());
if (cislo > 0)
    Console.Write("cislo je kladne");
else if (cislo < 0)
    Console.Write("cislo je zaporne");
else
    Console.Write("cislo je nula");
```

Tým pádom môžeme **else if** kľudne mať medzi **if** a **else** koľkokrát chceme a stále to bude považované za 1 príkaz.

2.1.2 Switch

Okrem if...else poznáme ešte jeden typ vetvenia - **switch**. Switch možno použiť pokiaľ dookola porovávame **jednu (rovnakú) premennú** s viacerými číslami. Tieto čísla usporiadame do **case**. Pokiaľ sa žiaden case nezhoduje s tým, čo hľadáme, spadne to do **default**. Ten je niečo ako else v if...else a rovnako tak jeho prítomnosť **nie je nutná**. Každý case by mal byť ukončený príkazom **break;**.

Uvažujme program, ktorý bude mať za úlohu zo vstupu načítať číslo a prekonvertovať ho na deň v týždni (1-pondelok, 2-utorok, ..., 7-nedeľa).

```
int den = Convert.ToInt32(Console.ReadLine());
switch (den)
{
    case 1:
        Console.WriteLine("PONDELOK");
        break;
    case 2:
        Console.WriteLine("UTOROK");
        break;
    case 3:
        Console.WriteLine("STREDA");
        break;
    case 4:
        Console.WriteLine("STVRTOK");
        break;
    case 5:
        Console.WriteLine("PIATOK");
        break;
    case 6:
        Console.WriteLine("SOBOTA");
        break;
    case 7:
        Console.WriteLine("NEDELA");
        break;
    default:
        Console.WriteLine("neznamy den");
        break;
}
```

V niektorých prípadoch je možné použiť aj **ternárny operátor**.

2.2 Cykly

Ďalším bežným konceptom v programovaní je **cyklenie**. Pokiaľ chceme, aby sa určitý úsek kódu opakoval, je vhodné použiť cyklus. Nechceme predsa mať zbytočne **duplicitný kód**, pretože by to výrazne skomplikovalo úpravu tohoto kódu. Niekedy by to ani jednoducho bez cyklu nešlo spraviť, pretože nemusíme presne poznať presný **počet opakovaní** už pri písaní kódu (spravidla pokiaľ počet opakovaní závisí na nejakej premennej).

2.2.1 While

Najjednoduchším typom cyklu je cyklus **while**. Funguje tak, že sa opakuje kým podmienka je splnená.

```
int i = 0;                      // deklaracia a inicializacia i
while(i < 5)                    // podmienka - cyklus sa bude opakovat kym i < 5
{
    Console.Write(i);           // >>> 01234
                                // 5 sa uz nevypise, lebo ked i = 5 neplati podmienka
                                // a tym padom sa cyklus ukonci
    i++;                        // inkrementacia i
}
```

2.2.2 For

Treba si všimnúť, že cyklus prebehol práve päťkrát - podmienka bola kým i je menšie ako 5. Toto je bežný spôsob ako si vynútiť určitý počet opakovaní. Preto existuje ďalší typ cyklu, ktorý sprehľadňuje tento zápis. Tento cyklus sa nazýva **for**. Skladá sa z **3 výrazov**. Prvý výraz sa vykoná **pred cyklom** (spravidla deklarácia a inicializácia nejakej premennej), druhý výraz je **podmienkou** a tretí výraz sa vykoná na konci každej iterácie cyklu.

Teda predchádzajúci kód by šlo prepísať takto.

```
for(int i = 0; i < 5; i++)
    Console.Write(i);           // >>> 01234
```

2.2.3 Do...while

V niektorých prípadoch zas vieme, že určite chceme, aby sa cyklus vykonal **aspoň raz** a podľa potreby možno aj **viackrát**. V takom prípade sa hodí využiť cyklus **do...while**, ktorý je presne na toto stavaný. Kľúčovým slovom **do** is označíme miesto, odkiaľ sa cyklus opakuje v prípade **splnenia podmienky** a na koniec bloku dáme **while**, čo označuje **podmienku opakovania**.

Uvažujme program, ktorý má za úlohu od užívateľa načítať číslo ako znak, v takom prípade ho pretypuje na integer a vypíše, v opačnom prípade vyzve užívateľa zadať ďalší znak až do bodu, kedy je zadané číslo.

```
char vstup;                             // premenna existuje iba v bloku kde bola deklarovana
do                                      // zaciatok cyklu
{
vstup = Console.ReadKey().KeyChar;      // citanie znaku z konzole
} while (vstup < '0' || vstup > '9');   // podmienka - na zaklade ASCII tabulky
Console.WriteLine();                    // ukoncenie riadku aby vystup bol oddeleny od vstupu
Console.WriteLine(vstup);               // vypis
```

2.2.4 Foreach

Celkom bežné je tiež cyklenie **cez každý prvok** nejakého iterovateľného objektu. Aj v tomto prípade by sme sa vedeli zaobísť aj s inými cyklami, ale môžeme sa vyhnúť indexovaniu a zároveň aj tvorbe podmienky. Tento cyklus sa nazýva **foreach**.

Uvažujme program, ktorý na vstupe dostane text zložený iba s malých písmen (teda žiadne medzery, žiadne číslice, žiadne veľké písmená a ani iné znaky). Čo tento program spraví, je že prevedie tento text na veľké písmená. Nechceme pri tom používať funkciu ToUpper().

```
string vstup = Console.ReadLine();      // nacitanie vstupu
string vystup = "";                     // vytvorenie premennej vystup
foreach (char znak in vstup )           // cyklime cez kazdy znak v retazci vstup
{
    vystup += (char)(znak + 'A' - 'a'); // konkatenujeme, vyuzivame rozdiel
                                        // velkych a malych pismen v ASCII tabulke
}
Console.WriteLine(vystup);              // vypis
```

2.2.5 Kľúčové slová skoku

Úzko s cyklami súvisia aj príkazy **break** a **continue**. Príkaz **break** ihneď **ukončí cyklus**. Príkaz **continue** ihneď **ukončí konkrétnu iteráciu cyklu** a pokiaľ je podmienka splnená pokračuje ďalšou.

```
for (int i = 0; i < 10; i++) // cykli cez cisla (0,1,...,8,9)
{
    if (i == 3) continue;    // pokial je i == 3 preskoci iteraciu
    if (i == 7) break;       // pokial je i == 7 ukonci cyklus
    Console.Write(i);        // >>> 012456
}
```

3 Polia, kolekcie

3.1 Polia

Pole je premenná, ktorá obsahuje **viacero hodnôt rovnakého typu**. Tento typ môže byť **ľubovoľný**. Tieto hodnoty budú v pamäti uložené hneď za sebou, preto je potrebné už pri **inicializácii** poznať, že koľko prvkov sa do tohto poľa vojde.

```
int[] pole  = new int[5];                       // vytvorenie pola obsahujuceho 5 prvkov
int[] pole2 = new int[5] { 1, 2, 3, 4, 5 };     // vytvorenie pola obsahujuceho 5 prvkov a
                                                // priradenie hodnot kazdeho z nich
// int[] pole2 = new int[] { 1, 2, 3, 4, 5 };   // pokial rovno priradzujeme hodnoty
                                                // nie je nutne specifikovat velkost pola
```

Pre prístup k jednotlivých prvkom využívame **indexy**.

```
int[] pole = new int[5] { 1, 2, 3, 4, 5 };      // vytvorenie pola
Console.WriteLine(pole[0]);                     // index [0]          // >>> 1
Console.WriteLine(pole[1]);                     // index [1]          // >>> 2
Console.WriteLine(pole[2]);                     // index [2]          // >>> 3
Console.WriteLine(pole[3]);                     // index [3]          // >>> 4
Console.WriteLine(pole[4]);                     // index [4]          // >>> 5
```

Rovnakým spôsobom aj **zapisujeme do poľa**.

```
int[] pole = new int[5];     // vytvorenie pola
pole[0] = 6;                 // pole = {6, 0, 0, 0, 0}
pole[1] = 7;                 // pole = {6, 7, 0, 0, 0}
pole[2] = 8;                 // pole = {6, 7, 8, 0, 0}
pole[3] = 9;                 // pole = {6, 7, 8, 9, 0}
pole[4] = 10;                // pole = {6, 7, 8, 9, 10}
```

3.1.1 Cyklenie cez pole

Na **precyklenie** celého poľa (napríklad na výpis) môžeme použiť cyklus **for**. V takom prípade však potrebujeme poznať dĺžku poľa. Tú však môžeme zistiť vďaka vlastnosti **Length**.

```
int[] pole = new int[5] { 1, 2, 3, 4, 5 };      // vytvorenie pola
for (int i = 0; i < pole.Length; i++)           // cyklus na prechod polom
    Console.Write(pole[i]);                     // >>> 12345
```

Alternatívne môžeme použiť aj cyklus **foreach**.

```
int[] pole = new int[5] { 1, 2, 3, 4, 5 };      // vytvorenie pola
foreach (int i in pole)                         // cyklus na prechod polom
    Console.Write(i);                           // >>> 12345
```

3.1.2 Viacrozmerné pole a pole polí

Viacrozmerné pole spravidla obsahuje polia, ktoré majú všetky **rovnakú dĺžku**. V pamäti sa tieto polia nachádzajú **za sebou**. Indexuje sa 2 číslami v 1 indexe **oddelenými čiarkou**. Prvé číslo špecifikuje **pole**, druhé **konkrétny prvok**.

```
int[,] pole = new int[3, 3] { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } }; // viarozmerne pole
for (int i = 0; i < 3; i++)
{
    for (int j = 0; j < 3; j++)
        Console.Write(pole[i, j]);
    Console.WriteLine();
}
// >>> 123
//     456
//     789
```

Pole polí môže obsahovať polia **rôznej dĺžky**. V pamäti sa polia nachádzajú **nezávisle na sebe**, v hlavnom poli sú len **odkazy na adresy jednotlivých polí**. Indexuje sa 2 indexami. Prvý index špecifikuje **pole**, druhý **konkrétny prvok**.

```
int[][] pole = new int[3][];                // pole poli
pole[0] = new int[3] { 1, 2, 3 };           // kazde pole moze mat inu dlzku
pole[1] = new int[2] { 4, 5 };
pole[2] = new int[4] { 6, 7, 8, 9 };
for (int i = 0; i < 3; i++)
{
    for (int j = 0; j < pole[i].Length; j++)
        Console.Write(pole[i][j]);
    Console.WriteLine();
}
// >>> 123
//     45
//     6789
```

3.2 List

List je veľmi podobný poľu. Hlavným rozdielom je, že listu **nešpecifikujeme dĺžku** - tá sa **môže** meniť. Zároveň máme k dispozícii množstvo **metód**. Stále je ho možné **indexovať**. Dĺžku listu zistíme cez **list.Count** (POZOR! Nie list.Length).

```
List<int> list = new List<int>(); // inicializacia listu
list.Add(10);                     // pridanie prvku
list.Add(30);                     // pridanie prvku
list.Add(20);                     // pridanie prvku
list.Add(30);                     // pridanie prvku
list.Remove(30);                  // odstrani prvy prvok, ktory sa zhoduje (index 1)
list.RemoveAt(0);                 // odstrani prvok na indexe 0
foreach (int i in list)           // cyklus cez list
    Console.WriteLine(i);         // vypis
// >>> 20
//     30
```

3.3 Slovník

Slovník je podobný listu. Jeho výhodou je, že sa neindexuje číslami, ale **kľúčmi**. Kľúč je prvá hodnota, môže byť **ľubovoľného dátového typu**. Druhá hodnota je samotná **hodnota** pre daný kľúč.

```
Dictionary<string, string> slovnik = new Dictionary<string, string>();
slovnik.Add("1", "raz");
slovnik.Add("2", "dva");
slovnik.Add("3", "tri");
slovnik.Add("4", "styri");
slovnik.Remove("1");
Console.WriteLine(slovnik["2"]); // mozeme kluc vyuzivat ako index // >>> dva
for (int i = 0; i < slovnik.Count; i++)
    Console.WriteLine(slovnik.ElementAt(i).Key + " " + slovnik.ElementAt(i).Value);
// >>> 2 dva
//     3 tri
//     4 styri
```

4 Funkcie

Funkcia je blok kódu, ktorý má nejaké meno a vykoná sa iba ak je **zavolaný**. Tým pádom ten istý kód môže prebehnúť **viacero ráz**. Funkciu zavoláme tým, že napíšeme jej **meno** a za to **okrúhle zátvorky**, do nich prípadne môžu ísť **parametre**. Funkciu **vytvoríme** tak, že napíšeme **návratový typ**, **názov** funkcie a do **okrúhlych zátvoriek** deklarujeme premenné, ktoré predstavujú **parametre** tejto funkcie. **Telo funkcie** (to, čo funkcia robí) píšeme do **množinových zátvoriek** (POZOR! V tomto prípade ich nemožno vynechať). Hodnotu z funkcie vraciame pomocoum kľúčového slova return.

Uvažujme program obsahujúci funkciu na súčet dvoch čísiel. Táto funkcia je zavolaná na súčet čísel 2 a 3 a ich výsledok je vypísaný do konzole.

```
// navratovy typ = int
// nazov funkcie = suma
// parametre = cislo1 a cislo2, oba typu int
int suma(int cislo1, int cislo2)          // deklaracia funkcie
{
    return cislo1 + cislo2;               // definicia funkcie
}
int vysledok = suma(2, 3);                // volanie funkcie
Console.WriteLine(vysledok);              // >>> 5
```

Pri volaní funkcie sa niekedy môže stať, že nie je prehľadné, ktorý argument je ktorý, pretože ich je **veľa** a **rovnakého dátového typu**. V takom prípade pre vyššiu prehľadnosť môžeme pri volaní funkcie priamo napísať, **o ktorý argument sa jedná**. Pokiaľ to robíme takto, na poradí argumentov **nezáleží**.

```
int vysledok = suma(cislo2: 3, cislo1: 2) // ekvivalentne volanie funkcie
```

Pokiaľ návratový typ nechceme, môžeme použiť špeciálny dátový typ **void**, ktorý znamená, že funkcia nič nevracia. Uvažujme program obsahujúci funkciu, ktorá vypíše "Hello World!".

```
void funkcia()
{
    Console.WriteLine("Hello World!");
}
funkcia();
```

5 Úvod do OOP – tvorba triedy, objektu, metód

Doteraz sme sa zaoberali takzvaným **procedurálnym programovaním**. Ďalšou možnosťou je **objektovo orientované programovanie** (OOP), ktoré je založené na triedach a objektoch. **C#** je objektovo orientovaný jazyk, čo znamená, že v podstate všetko je stavané na objektoch a triedach. Každá trieda by po správnosti mala byť v **samostatnom súbore**, ktorý má **zhodný názov** s touto triedou. Pre vytvorenie novej triedy môžeme vo Visual Studiu využiť klávesovú skratku **Ctrl + Shift + A**.

5.1 Trieda a objekt

Triedu si môžeme predstaviť ako nejaký blueprint/šablónu, na základe ktorej vieme vytvárať **objekty**. Dá sa povedať, že objekt je **premenná**, ktorej dátový typ je nejaká konkrétna **trieda**. Trieda obsahuje **atribúty** (premenné), **metódy** (funkcie), **konštruktor**, **vlastnosti** a mnohé ďalšie. Pristupujeme k nim cez **bodku**.

```
public class Clovek
{
    // atributy = premenne v triede
    public string meno = "";
    public int vek = -1;

    // metody = funkcie v triede
    public void Vypis()
    {
        Console.WriteLine("\"" +  meno + "\" ma " + vek + " rokov");
    }

}
```

```
Clovek clovek = new Clovek();     // vytvorenie objektu clovek z triedy Clovek

clovek.Vypis();                   // volanie metody vypis z triedy Clovek
                                  // >>> "" ma -1 rokov
```

Vypísalo sa to, čo sme do atribútov natvrdo nastavili, môžeme to však zmeniť tak, aby to ovplyvnilo **iba objekt, v ktorom to zmeníme**. Tým pádom reálne vidíme, že objekty sú v podstate iba **premenné dátového typu triedy**, v tomto prípade triedy Clovek.

```
Clovek clovek1 = new Clovek();
Clovek clovek2 = new Clovek();

clovek1.meno = "Fero";
clovek1.vek = 15;
clovek2.meno = "Jozo";
clovek2.vek = 16;

clovek1.Vypis();                  // >>> "Fero" ma 15 rokov
clovek2.Vypis();                  // >>> "Jozo" ma 16 rokov
```

5.2 Konštruktor

Konštruktor je špeciálna metóda, ktorá sa zavolá **pri vytvorení** objektu z danej triedy. Tým pádom môžeme rovnaký výsledok ako v predchadzajúcom príklade dosiahnuť týmto spôsobom.

```
public class Clovek
{
    // atributy
    public string meno = "";
    public int vek = -1;

    // konstruktor
    public Clovek(string meno, int vek)
    {
        this.meno = meno;
        this.vek = vek;
    }

    // metody
    public void Vypis()
    {
        Console.WriteLine("\"" +  meno + "\" ma " + vek + " rokov");
    }

}
```

```
Clovek clovek1 = new Clovek("Fero", 15);     // konstruktoru predavame parametre
Clovek clovek2 = new Clovek("Jozo", 16);     // konstruktoru predavame parametre


clovek1.Vypis();                             // >>> "Fero" ma 15 rokov
clovek2.Vypis();                             // >>> "Jozo" ma 16 rokov
```

V konštruktore môžeme vidieť kľúčové slovo **this**. Jedná sa o **ukazateľ na triedu, v ktorej sa konštuktor nachádza**. Tým pádom vieme vyriešiť ten problém, že máme rovnako nazvaný **argument konštruktora** a **atribút**.

POZOR! Tým, že sme pridali **konštruktor s parametrami** nám **zaniká existencia predvoleného prázdneho konštruktora**! Pokiaľ by sme chceli, konštruktorov môžeme mať v triede aj **viac**.

V tomto prípade by niečo takéto nefungovalo - prekladač by nám to nepreložil.

```
Clovek clovek = new Clovek();     // ERROR! Clovek neobsahuje bezparametricky konstruktor
```

6 Modifikátory prístupu, modifikátor static

6.1 Modifikátory prístupu

Modifikátory prístupu sa používajú na **povolenie/zakázanie** prístupu k určitým prvkom v triede z rôznych miest.

modifikátor	možný prístup z
public	všade
protected	iba v tejto triede a v triedach, ktoré túto triedu dedia
private	iba v tejto triede

Kompletný zoznam modifikátorov prístupu je možné nájsť v dokumentácii jazyka C# od Microsoftu.

Modifikátory prístupu aplikujeme tým, že ho vložíme **pred deklaráciu** nejakého prvku. Tým prvkom môže byť čokoľvek - **metóda**, **atribút**, **vlastnosť**, **konštruktor**...

```
public class Trieda
{
    private int a;
    protected int b;
    public int c;

    public void metoda()
    {
        a = 0; // OK
        b = 0; // OK
        c = 0; // OK
    }
}
```

```
Trieda trieda = new Trieda();
trieda.a = 0; // ERROR
trieda.b = 0; // ERROR
trieda.c = 0; // OK
trieda.metoda();
```

Poznámka: Modifikátor protected bude lepšie vysvetlení pri **dedičnosti**.

6.2 Zapúzrenie

Nechceme, aby nám každý pristupoval k **atribútom triedy**. Môže tomu byť tak napríklad preto, že dáta musíme mať nejakú formu a nechceme a chceme mať teda ich zápis nejako ošetrený. V c# na to máme **Vlastnosti**. **Vlastnosť** je v podstate spojenie **atribútu** s **metódami**, konkrétne s metódami zvanými **getter** a **setter**. Vlastnosti sa zvyknú nazývať **rovnako ako atribút, s ktorým pracujú, ale s veľkým prvým písmenom**, ale nie je to **pravidlo**.

```
public class Trieda
{
    private int cislo;          // atribut

    public int Cislo
    {
        get { return cislo; }   // metoda int getCislo()
        set { cislo = value; }  // metoda void setCislo(int value)
    }
}
```

```
Trieda trieda = new Trieda();
trieda.Cislo = 7;
Console.Write(trieda.Cislo);    // >>> 7
```

Pokiaľ chceme, môžeme zápis vlastnosti skrátiť a to tak, že **atribút** ani **nedeklarujeme** a tým pádom berieme predvolené metódy **get** a **set**. Ekvivalentná trieda by vyzerala takto.

```
public class Trieda
{
    // atribut nie je

    public int Cislo
    {
        get; // funguje rovnako
        set; // funguje rovnako
    }
}
```

Samozrejme, kebyže chceme tieto metódy upraviť, stačí zameniť **bodkočiarku** za **telo metódy**.

6.3 Modifikátor static

V niektorých prípadoch nedáva zmysel vytvárať **objekt** len pre to, aby sme mohli pristupovať k prvkom z týchto tried. Príkladom statickej metódy môže byť **Console.WriteLine()**, kebyže nie je statická, museli by sme si najskôr vytvoriť objekt z triedy **Console** a potom na tento **objekt** volať metódu **WriteLine()**. Aby sme spravili prvok v triede **statický**, stačí pred jeho deklaráciu (spravidla po **modifikátoroch prístupu**) napísať **static**.

```
public class Trieda
{
    public static void StatickaMetoda()
    {
        Console.WriteLine("test");
    }
}
```

```
Trieda.StatickaMetoda();      // >> test
```

7 Dedičnosť

Dedičnosť umožňuje jednej triede zdediť a tým pádom využívať **atribúty**, **metódy**, **vlasnosti**. Triedy môžu dediť priamo iba z **jednej** triedy (ale trieda môže dediť z triedy, ktorá dedí z ďalšej triedy, atď.). Nemôže však nastať cyklické dedenie. Pre príklad, pokiaľ trieda A dedí z triedy B, trieda B **nemôže dediť** z **triedy A** a ani zo žiadnej inej triedy, ktorá **triedu A dedí**.

```
class Clovek // rodicovska trieda
{
    public string Meno { get; set; }
    public string Priezvisko { get; set; }
    public int Vek { get; set; }
    public Clovek(string meno, string priezvisko, int vek)
    {
        Meno = meno;
        Priezvisko = priezvisko;
        Vek = vek;
    }
    public void vypis()
    {
        Console.WriteLine($"{Meno}, {Priezvisko}, {Vek}");
    }
}
```

```
class Student : Clovek                // dcerska trieda, dedi z rodicovskej triedy
{
    public string Skola { get; set; }
    public int Rocnik { get; set; }
    public Student(string meno, string priezvisko, int vek, string skola, int rocnik)
        : base(meno, priezvisko, vek) // volanie konstruktora rodicovskej triedy
    {
        Skola = skola;
        Rocnik = rocnik;
    }
    public void vypis2()
    {
        Console.WriteLine($"{Meno}, {Priezvisko}, {Vek}, {Skola}, {Rocnik}");
    }
}
```

```
Clovek c = new Clovek("Ludovit", "Stur", 25);
c.vypis();      // >>> Ludovit, Stur, 25
Student s = new Student("Jozko", "Ferko", 17, "SPSIT", 3);
s.vypis();      // >>> Jozko, Ferko, 17
s.vypis2();     // >>> Jozko, Ferko, 17, SPSIT, 3
```

Pri dedení sú prvky s modifikátorom **protected** dostupné v **zdedenej triede**, ale nie inde. Dá sa teda povedať, že modifikátor **protected** sa správa ako public pre **zdedené triedy**, ale ako private pre všetko ostatné.

```
class A
{
    private int a;
    protected int b;
    public int c;
}
```

```
class B : A
{
    void metoda ()
    {
        a = 0;  // ERROR
        b = 0;  // OK
        c = 0;  // OK
    }
}
```

```
B b = new B();
b.a = 0;  // ERROR
b.b = 0;  // ERROR
b.c = 0;  // OK
```

8 Rozhranie

Rozhranie (interface) funguje ako šablóna na **triedy**. Do **rozhrania** píšeme iba deklarácie. Všetky definície doplní **trieda**, ktorá bude tento interface používať. Nie je teda možné definovať iba **niektoré**. Pokiaľ chceme rozhranie používať, musíme ho **zdediť**. Na rozdiel od triedy je možné dediť **viacero rozhraní**, prípadne zároveň triedu a zároveň rozhrania. Rovnako ako pri triede, je vhodné umiestniť toto rozhranie do **samostatného súboru**. Meno rozhrania väčšinou začína písmenom I (Interface).

```
interface ITvar
{   // iba deklaracie
    public string Nazov { get; set; }
    public int Obvod();
    public int Obsah();
}
```

```
class Obdlznik : ITvar   // pouzivame interface
{   // doplnime definicie
    public string Nazov { get; set; }
    public int A { get; set; }
    public int B { get; set; }
    public Obdlznik(int a, int b)
    {
        Nazov = "Obdlznik";
        A = a;
        B = b;
    }
    public int Obsah() { return A * B; }
    public int Obvod() { return 2 * (A + B); }
}
```

```
ITvar obdlznik = new Obdlznik(5, 4);  // ako datovy typ dame konkretny interface,
                                      // ale samozrejme by fungoval aj Obdlznik
Console.WriteLine(obdlznik.Obvod());  // >>> 18
Console.WriteLine(obdlznik.Obsah());  // >>> 20
```

Prečo využívať rozhrania? Jedným z dôvodov je podpoda **polymorfizmu**, o ktorom sa viac dozvieme v ďalšej kapitole.

9 Polymorfizmus

Polymorfizmus vychádza z gréckeho **poly morph**, čo znamená **mnoho foriem**. Jedná sa o jeden zo základných konceptov **objektovo orientovaného programovania**. Polymorfizmus nám umožňuje vykonať **rovnakú akciu inak**. Inak povedané chceme byť schopní rôzne **podtriedy** považovať za objekty z **nadradenej triedy**, aby sme nemuseli pre každú podtriedu definovať všetko zvlášť. Poznáme 2 druhy polymorfizmu: **run-time polymorfizmus** a **compile-time polymorfizmus**.

9.1 Run-time polymorfizmus

To, ktorá metóda sa zavolá sa rozhodne **počas behu programu**. Využívajú sa **virtuálne** metódy. To sú také metódy, ktoré je možné **preťažovať**. Jedná sa o niečo podobné ako metódy v **rozhraní**, ale s tým rozdielom, že virtuálne metódy **nie je nutné preťažovať**. Teda majú nejakú **predvolenú definíciu**. Virtuálne metódy preťažujeme kľúčovým slovom **override**.

```
class Zviera
{
    public virtual void Zvuk()          // virtualna metodu
    {
        Console.WriteLine("zvuk");      // predvolena definicia metodu
    }
}
```

```
class Pes : Zviera
{
    public override void Zvuk()        // pretazujeme metodu
    {
        Console.WriteLine("haf haf");  // pretazena definicia metody
    }
}
```

```
class Macka : Zviera
{
    public override void Zvuk()        // pretazujeme metodu
    {
        Console.WriteLine("mnau mnau");// pretazena definicia metody
    }
}
```

```
static void Main()
{
    Zviera zviera = new Zviera();
    zviera.Zvuk();                      // >>> zvuk

    zviera = new Pes();
    zviera.Zvuk();                      // >>> haf haf

    zviera = new Macka();
    zviera.Zvuk();                      // >>> mnau mnau
}
```

9.2 Compile-time polimorfizmus

To, ktorá metóda sa zavolá sa rozhodne už **pri preklade**. Dosiahneme toho tak, že budeme mať **rôzne metódy** s **rovnakým názvom**. Je dôležité, aby tieto metódy mali buď **rôzny počet parametrov**, alebo **iné dátové typy parametrov**.

```
class Trieda
{
    public static void Metoda()
    {
        System.Console.WriteLine("Metoda triedy");
    }
    public static void Metoda(int a)
    {
        System.Console.WriteLine("Metoda triedy s parametrom int");
    }
    public static void Metoda(string a)
    {
        System.Console.WriteLine("Metoda triedy s parametrom string");
    }
    public static void Metoda(int a, string b)
    {
        System.Console.WriteLine("Metoda triedy s parametrami int a string");
    }
    public static void Metoda(string a, int b)
    {
        System.Console.WriteLine("Metoda triedy s parametrami string a int");
    }
}
```

```
Trieda.Metoda();          // >>> Metoda triedy
Trieda.Metoda(1);         // >>> Metoda triedy s parametrom int
Trieda.Metoda("a");       // >>> Metoda triedy s parametrom string
Trieda.Metoda(1, "a");    // >>> Metoda triedy s parametrami int a string
Trieda.Metoda("a", 1);    // >>> Metoda triedy s parametrami string a int
```

10 Generika

10.1 Generické triedy

Generika nám umožňuje písať triedy nezávisle na **dátových typoch**. Teda môžeme mať triedu, ktorá bude fungovať pre **všetky dátové typy** a nemusíme pre **každý dátový typ** definovať vlastnú triedu.

```
class Trieda<T>                                 // T = lubovonly datovy typ, mozeme premenovat
{
    public T Premenna { get; set; }             // vlastnost majuca datovy typ T
}
```

```
Trieda<int> trieda = new Trieda<int>();         // nastavime T na int pre tento objekt
trieda.Premenna = 5;
Console.WriteLine(trieda.Premenna);             // >>> 5

Trieda<string> trieda2 = new Trieda<string>();// nastavime T na string pre tento objekt
trieda2.Premenna = "text";
Console.WriteLine(trieda2.Premenna);            // >>> text
```

10.2 Generické funkcie

Okrem generický tried môžeme mať aj **generické funkcie**. Ako príklad môžeme uvažovať funkciu, ktorá zameňuje hodnoty dvoch premenných.

```
namespace VymenaHodnot
{
    public class Program
    {
        public static void Main()
        {
            int a = 1, b = 0;
            Console.WriteLine($"{a}, {b}");             // >>> 1, 0
            Vymen<int>(ref a, ref b);
            Console.WriteLine($"{a}, {b}");             // >>> 0, 1

            string x = "a", y = "b";
            Console.WriteLine($"{x}, {y}");             // >>> a, b
            Vymen<string>(ref x, ref y);
            Console.WriteLine($"{x}, {y}");             // >>> b, a
        }

        // Pozn. ref - referencia, nie je sucastou kazdej generickej triedy, jedna
        // sa iba o to, aby program vedel, ze chceme parametre ako referencie a nie hodnoty
        public static void Vymen<T>(ref T a, ref T b)  // genericka trieda
        {
            T temp = a;                                 // pomocna premenna typu T
            a = b;
            b = temp;
        }
    }
}
```

11 Výnimky

Výnimka je **error za behu programu**. Tento error by za normálnych okolností spôsobil **pád programu**, ale mi ho vieme **odchytiť** a **riešiť**.

Uvažujme jednoduchý program, kde na vstupe načítame číslo a vypíšeme ho.

```
int cislo = Convert.ToInt32(Console.ReadLine());
Console.WriteLine(cislo);
```

Program funguje, ale nie úplne. Čo sa stane ak na vstup zadáme niečo, čo nie je integer? **System.FormatException: 'Input string was not in a correct format.'** a program padá.

11.1 Odchytávanie výnimiek

Výnimku môžeme odchytiť. Robí sa to tak, že kód, ktorý môže **vyhodiť výnimku** vnoríme do **try** bloku a čo sa má stať ak **výnimka nastane** hodíme do bloku **catch**.

```
try                                 // ak v tomto bloku vyhodi vynimku, ideme do catch bloku
{
    int cislo = Convert.ToInt32(Console.ReadLine());
    Console.WriteLine(cislo);
}
catch (Exception e)                 // Exception je datovy typ vynimky a e je teda premenna
{
    Console.WriteLine(e.Message);   // vypiseme text vynimky do konzole
}
```

11.1.1 Blok finally

V niektorých prípadoch chceme vykonať nejaký kód po try...catch bloku **nezávisle na tom, či výnimka bola vyhodená**. Nemá zmysel písať tento kód viacero ráz. Stačí pridať ďalší blok - **finally**.

```
try
{
    int cislo = Convert.ToInt32(Console.ReadLine());
    Console.WriteLine(cislo);
}
catch (Exception e)
{
    Console.WriteLine(e.Message);
}
finally
{
    Console.WriteLine("koniec");    // vykona sa nezavisle po try/catch
}
```

Poznámka: asi sa ponúka otázka, že na čo je **finally** blok vôbec dobrý, keďže aj ak je výnimka vyhodená, tak aj tak **pokračujeme vo vykonávaní kódu**. Rozdiel je taký, že **finally blok** sa vykoná **VŽDY**. Teda sa vykoná aj keby v **catch bloku** bolo **return**. Do **finally** bloku chceme dávať kód na **upratanie**.

11.2 Vlastné výnimky

Výnimky sú užitočné, pretože vďaka ním vieme, že niečo nefunguje tak, ako by sme čakali. Niekedy je teda vhodné **vyhadzovať výnimky**. Môžeme tak urobiť cez kľúčové slovo **throw**.

Napríklad takto by sme vyhodili výnimku, pokiaľ by sa niekto snažil zavolať ešte neimplementovanú funkciu.

```
void NeimplementovanaFunkcia()
{
    throw new NotImplementedException("Tuto funkciu je potreba implementovat");
    // vynimiek je viacero druhov, zakladnu vyhodime ‘throw new Exception();‘
}

NeimplementovanaFunkcia();
// Program spadne: System.NotImplementedException: ’Tuto metodu je potreba implementovat’
```

12 Práca s textovými, binary a csv súbormi

Dôležitým občas môže byť **čítanie** a **zápis** do súborov rôznych typov. Môže sa to hodiť napríklad na lokálne **lokálne úložisko** nejakých dát, prípadne pri spracovaní dát, ktoré už v **súboroch sú**. Aby sme so súboru mohli čítať, musíme poznať jeho **umiestnenie** a aj čítanie, aj zápis nám zjednoduší trieda **FileStream**.

12.1 Práca s binárnymi súbormi

```
string cestaKSuboru = Path.Combine(Environment.CurrentDirectory, "subor.bin");

// zapis

using (var fs = new FileStream(cestaKSuboru, FileMode.Create))
{
    using (var bw = new BinaryWriter(fs))
    {
        bw.Write(1);
        bw.Write(0.7);
        bw.Write('c');
    }
}

// citanie

using (var fs = new FileStream(cestaKSuboru, FileMode.Open))
{
    using (var br = new BinaryReader(fs))
    {
        Console.WriteLine(br.ReadInt32());  // >>> 1
        Console.WriteLine(br.ReadDouble()); // >>> 0,7
        Console.WriteLine(br.ReadChar());   // >>> c
    }
}
```

12.2 Práca s textovými súbormi

```
string cestaKSuboru = Path.Combine(Environment.CurrentDirectory, "subor.txt");

// zapis

using (StreamWriter sw = new StreamWriter(cestaKSuboru))
{
    sw.WriteLine("prvy riadok");
    sw.WriteLine("druhy riadok");
    sw.WriteLine("treti riadok");
}

// citanie
// cele naraz

using (StreamReader sr = new StreamReader(cestaKSuboru))
{
    Console.WriteLine(sr.ReadToEnd());   // >>> prvy riadok
                                         //     druhy riadok
                                         //     treti riadok
}

// po riadkoch

using (StreamReader sr = new StreamReader(cestaKSuboru))
{
    string riadok;
    while ((riadok = sr.ReadLine()) != null)
    {
        Console.WriteLine(riadok);       // >>> prvy riadok
                                         // >>> druhy riadok
                                         // >>> treti riadok
    }
}
```

12.3 Práca s csv súbormi

CSV znamená **Comma Separated Values**, teda **Hodnoty Oddelené Čiarkou**. Pre prácu s CSV súbormi máme knižnicu, ktorú môžeme použiť. Volá sa **CsvHelper** a aby sme ju mohli využívať ju musíme najskôr nainštalovať. To môžeme dosiahnuť tak, že vo Visual Studiu klikneme pravým tlačidlom na **Solution** a vyberieme **Manage NuGet packages for Solution** a pridáme ju.

```
using CsvHelper;
using System.Globalization;

string cestaKSuboru = Path.Combine(Environment.CurrentDirectory, "subor.csv");

// zapis

using (var writer = new StreamWriter(cestaKSuboru))
using (var csv = new CsvWriter(writer, CultureInfo.InvariantCulture))
{
    csv.WriteRecords(new List<dynamic>
    {
        new { Id = 1, Prezyvka = "Jozef", Xp = 100, Level = 1 },
        new { Id = 2, Prezyvka = "Martin", Xp = 200, Level = 2 },
        new { Id = 3, Prezyvka = "Peter", Xp = 300, Level = 3 },
    });
}

// citanie

using (var reader = new StreamReader(cestaKSuboru))
using (var csv = new CsvReader(reader, CultureInfo.InvariantCulture))
{
    var records = csv.GetRecords<dynamic>().ToList();
    foreach (var record in records)
    {
        Console.WriteLine($"{record.Id}, {record.Prezyvka}, {record.Xp}, {record.Level}");
        // >>> 1, Jozef, 100, 1
        // >>> 2, Martin, 200, 2
        // >>> 3, Peter, 300, 3
    }
}
```

13 Serializácia a deserializácia XML a JSON súborov

Serializácia je proces, kedy sa prevádza z **objektu** do (väčšinou) **textového formátu**, ktorý je jednoduchší na **uloženie** a **prenesenie**. **Deserializácia** je opačný proces, čiže prevod späť na **objekt**. Používajú sa dva typy tohoto textového formátu a to **JSON** a **XML**.**JSON** (JavaScript Object Notation) sa dnes používa častejšie ako **XML** (eXtensible Markup Language). Pre prácu s nimi v jazyku **C#** môžeme využívať **knižnice**.

13.1 XML

Ako vyzerá XML?

```
<?xml version="1.0" encoding="utf-8"?>
<Person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Name>John</Name>
  <Age>30</Age>
</Person>
```

```
using System.Xml.Serialization;  // potrebna kniznica pre pracu s XML
namespace XML
{
    class Program
    {
        static void Main(string[] args)
        {
            var person = new Person { Name = "John", Age = 30 };

            // serializacia
            var xml = new XmlSerializer(typeof(Person));
            using (var writer = new StreamWriter("person.xml"))
            {
                xml.Serialize(writer, person);
            }

            // deserializacia
            using (var reader = new StreamReader("person.xml"))
            {
                var person2 = xml.Deserialize(reader) as Person;
                Console.WriteLine(person2.Name);
                Console.WriteLine(person2.Age);
            }
        }
    }
    public class Person
    {
        public string Name { get; set; }
        public int Age { get; set; }
    }
}
```

13.2 JSON

Ako vyzerá JSON?

```
{"Name":"John","Age":30}
```

```
using System.Text.Json;  // potrebna kniznica pre pracu s JSON
namespace JSON
{
    class Program
    {
        static void Main(string[] args)
        {
            var person = new Person { Name = "John", Age = 30 };

            // serializacia
            var json = JsonSerializer.Serialize(person);

            // deserializacia
            var person2 = JsonSerializer.Deserialize<Person>(json);
            Console.WriteLine(person2.Name);
            Console.WriteLine(person2.Age);
        }
    }
    public class Person
    {
        public string Name { get; set; }
        public int Age { get; set; }
    }
}
```

14 GUI – Úvod do Windows Forms

Po vytvorení projektu nám Visual Studio vygeneruje nejaké súbory. Každý z týchto súborov je svojim spôsobom dôležitý.

Pri otvorení **Form1.cs** sa nám zobrazí **prázdne okno**. Do tohto okna môžeme z **Toolboxu** pridať rôzne prvky. Pokiaľ Toolbox **nevidíme**, je potrebné ho zobraziť. Môžeme tak urobiť cez **View** > **Toolbox**. Z tadiaľ môžeme cez **Drag&Drop** pridávať do okna jednotlivé prvky. Každému z týchto prvkov môžeme meniť **vlastnosti** a **udalosti** v okne s názvom **Properties**.

Pokiaľ na toto okno klikneme **pravým tlačidlom myši**, máme možnosť zobraziť kód - **View Code**. Do tohto kódu budeme neskôr pridávať rôzne funkcie.

```
namespace WinFormsApp1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
    }
}
```

Môžeme si všimnúť, že trieda je označená ako **parciálna**. To znamená, že všetko, čo do tejto triedy patri, nie je **len v tomto súbore**. Druhú časť tejto triedy vieme nájsť v súbore **Form1.Designer.cs**. Jedná sa o automaticky generovaný kód, ktorý zobrazuje okno so všetkými prvkami, preto ho priamo editovať väčšinou **nechceme**.

V neposlednom rade máme **Program.cs**, ktorý sa stará o samotné **spustenie aplikácie**.

```
namespace WinFormsApp1
{
    internal static class Program
    {
        /// <summary>
        ///  The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            // To customize application configuration such as set high DPI settings or
                default font,
            // see https://aka.ms/applicationconfiguration.
            ApplicationConfiguration.Initialize();
            Application.Run(new Form1());
        }
    }
}
```

Okrem vymenovaných súborov máme ešte **Dependencies** a **Form1.resx**, ale tie sú pre nás irelevantné - nepotrebujeme im rozumieť.

15 GUI - Komponenty

Aby sme samotné **grafické uživateľské rozhranie** (GUI) vedeli vytvoriť, potrebujeme poznať rôzne **komponenty**. Podrobnejšie si rozoberieme tie najbežnejšie. **Button** (tlačidlo) využívame na to, že užívateľ ho môže zmáčknuť a na základe **zmáčknutia** sa niečo stane. **Checkbox** (zaškrtávacie políčko) môžeme kliknutím buď **označiť**, alebo **odznačiť**. **ComboBox** je textové pole, pri ktorom môžeme **vyberať zo zoznamu**. **Label** je proste text, ktorý **nemôže** upraviť užívateľ. **PictureBox** využívame ak chceme zobraziť obrázok. Ak chceme zobraziť stav, že koľko percent už je hotových, môžeme použiť **ProgressBar**. **RadioButton** je veľmi podobný **CheckBoxu**, ale rozdiel je ten, že RadioButton môžeme zo skupiny mať vybraný len jeden. Ak chceme, aby užívateľ mohol písať, môžeme mu vytvoriť políčko na text - **TextBox**.

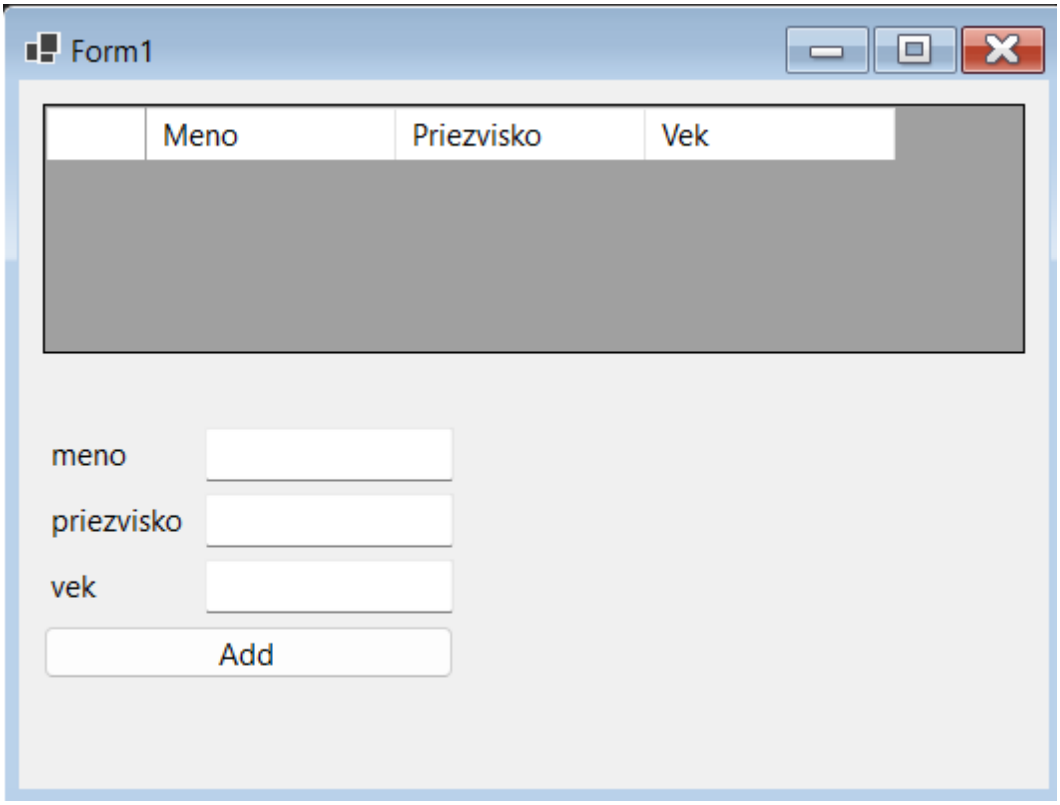
16 GUI – Práca s dátami (DataGridView, BindingList)

Na zobrazovanie dát v tabuľkovom formáte využívame **DataGridView**. Riadky reprezentujú jednotlivé **záznamy** a stĺpce reprezentujú **vlastnosti** týchto záznamov.

Kebyže ako zdroj dát chceme používať **List**, tak sa objaví problém, že dáta v tabuľke sa **neaktualizujú**. Tento problém však ide vyriešiť tým, že namiesto **List** budeme používať **BindingList**. Ten má špeciálnu vlastnosť, že pokiaľ sa zmení, tak informuje všetky ovládacie prvky naň viazané.

16.1 Ukážka

Uvažujme program, kde chceme pridávať rôzne osoby do tabuľky a chceme si uchovať ich meno, priezvisko a vek.



Obr. 5: GUI ukážky

Form1.cs môžeme upraviť takto.

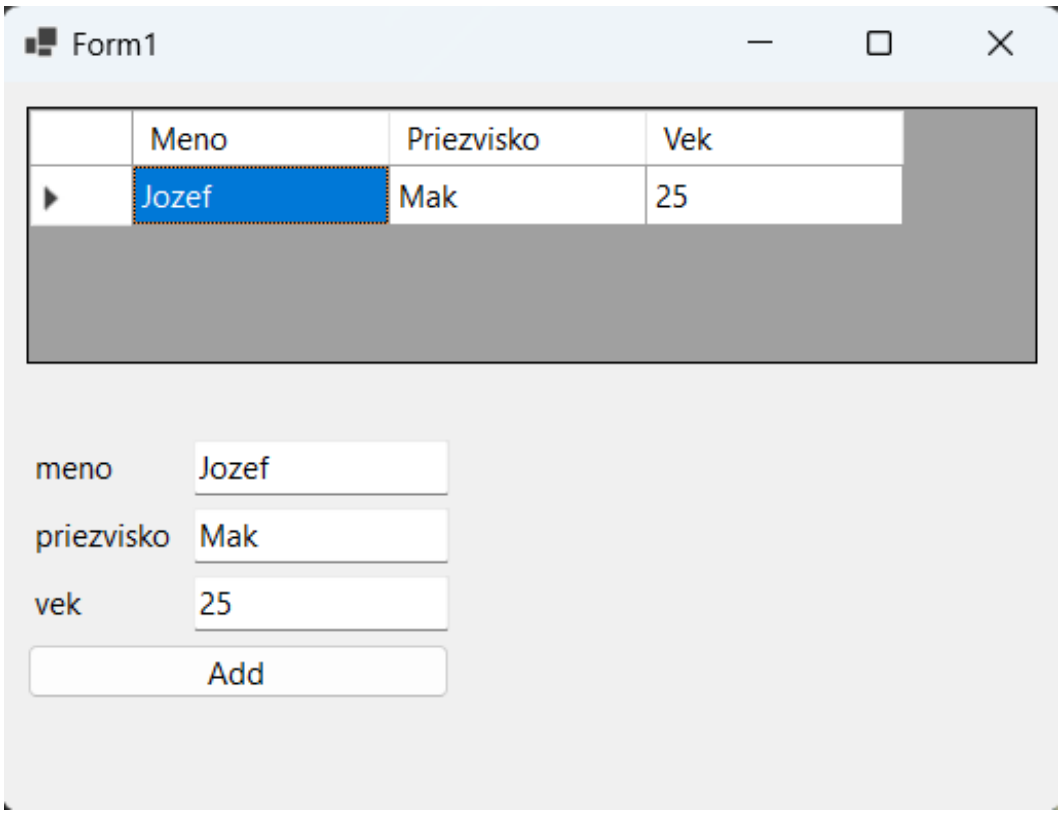
```
1  using System.ComponentModel;
2
3  namespace Ukazka
4  {
5      public partial class Form1 : Form
6      {
7          public Form1()
8          {
9              InitializeComponent();
10
11             // nastavenie DataGridView
12             dataGridView1.DataSource = zoznamOsob;
13         }
14
15         BindingList<Osoba> zoznamOsob = new();
16
17         private void AddButton_Click(object sender, EventArgs e)
18         {
19             Osoba osoba = new Osoba(MenoTextBox.Text, PriezviskoTextBox.Text,
                   int.Parse(VekTextBox.Text));
20             zoznamOsob.Add(osoba);
21         }
22     }
23 }
```

Potrebujeme preto vytvoriť triedu **Osoba**.

```
public class Osoba
{
    public string Meno { get; set; }
    public string Priezvisko { get; set; }
    public int Vek { get; set; }

    public Osoba(string meno, string priezvisko, int vek)
    {
        Meno = meno;
        Priezvisko = priezvisko;
        Vek = vek;
    }
}
```

Teraz keď chceme pridať záznam, tak vyplníme údaje osoby a klikneme na tlačidlo.



Obr. 6: Pridanie záznamu

Poznámka: pre správne fungovanie je nutné si premenovať jednotlivé komponenty.

17 GUI – Udalosti (Event Handler)

Event Handler je špeciálna metóda, ktorá je spojená s nejakou **udalosťou**. Táto metóda sa zavolá vždy, keď daná udalosť nastane. Udalosťou môže byť napríklad **kliknutie na tlačidlo**, **zmena textu v TextBoxe**... V predchádzajúcej kapitole sme sa už s jedným Event Handlerom stretli - **AddButton_Click**. Môžeme si všimnúť, že táto metóda mala dva parametre - **object sender** a **EventArgs e**, kde **sender** je referencia na komponent, ktorá túto udalosť vyvolala a **e** je objekt obsahujúci nejaké doplňujúce informácie.

Komponenty vo Winfows Forms majú týchto udalostí mnoho, ich zoznam si môžeme pozrieť v okne s názvom **Properties**, ale musíme sa prekliknuť na udalosti (**Events**), ktoré majú ikonku blesku. Tu aj k jednotlivým udalostiam vieme pridať **Event Handlery**.

18 GUI – Layouty

Layout je spôsob, ktorým sú jednotlivé komponenty rozložené.

Základnou možnosťou je **ručné rozloženie**. V jednoduchosti to znamená to, že každý prvok my určíme, že kde sa má **nachádzať**, aký má byť **veľký** a že či je **ukotvený** k nejakému okraju. Na to všetko využívame vlastnosti **Location**, **Size** a **Anchor**.

Pri komplexnejších formulároch toto nemusí byť úplne najlepšie riešenie a preto môžeme radšej použiť rôzne **kontajnery**.

FlowLayourPanel je kontajner, ktorý umiestňuje prvky jeden za druhým a to buď do riadku, alebo do stĺpca v závislosti od vlastnosti **FlowDirection**. Pokiaľ sa všetky prvky nevojdu na jeden riadok (do jedného stĺpca), tak vlastnosť **WrapContents** určuje, či sa má pokračovať na novom riadku (v novom stĺpci).

TableLayoutPanel je kontajner, ktorý jednotlivé prvky umiestňuje do mriežky v závislosti na vlastnostiach **RowCount** a **ColumnCount**.

Pri layoutoch je dôležitá aj vlastnosť **Dock**, ktorá ukotvuje k okrajom **kontajneru** alebo **formulára**.

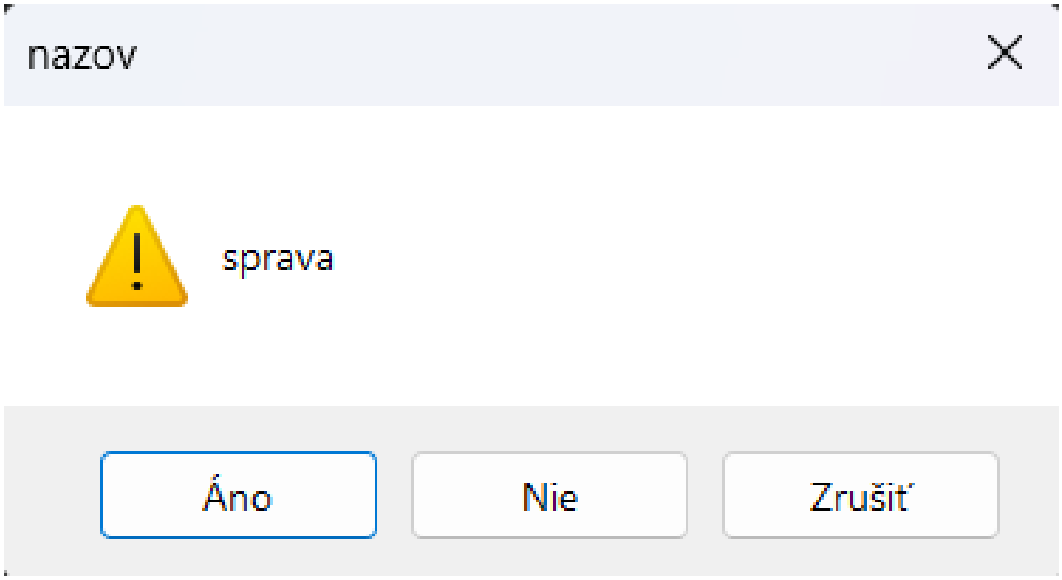
AutoSize je vlastnosť, ktorá umožňuje komponentu mať veľkosť takú, aby sa do nej vošiel všetok obsah.

19 GUI – Diaogové okná

Dialógové okná sú okná slúžiace na **interakciu s používateľom**. Poskytujú možnosti ako zobrazenie správ, výzvu na zadanie údajov alebo výber možností.

Najjednoduchším dialógovým oknom je **MessageBox**. Ten sa využíva hlavne na zobrazenie **správ a upozornení**.

```
1 MessageBox.Show("sprava", "nazov", MessageBoxButtons.YesNoCancel, MessageBoxIcon.Warning);
```



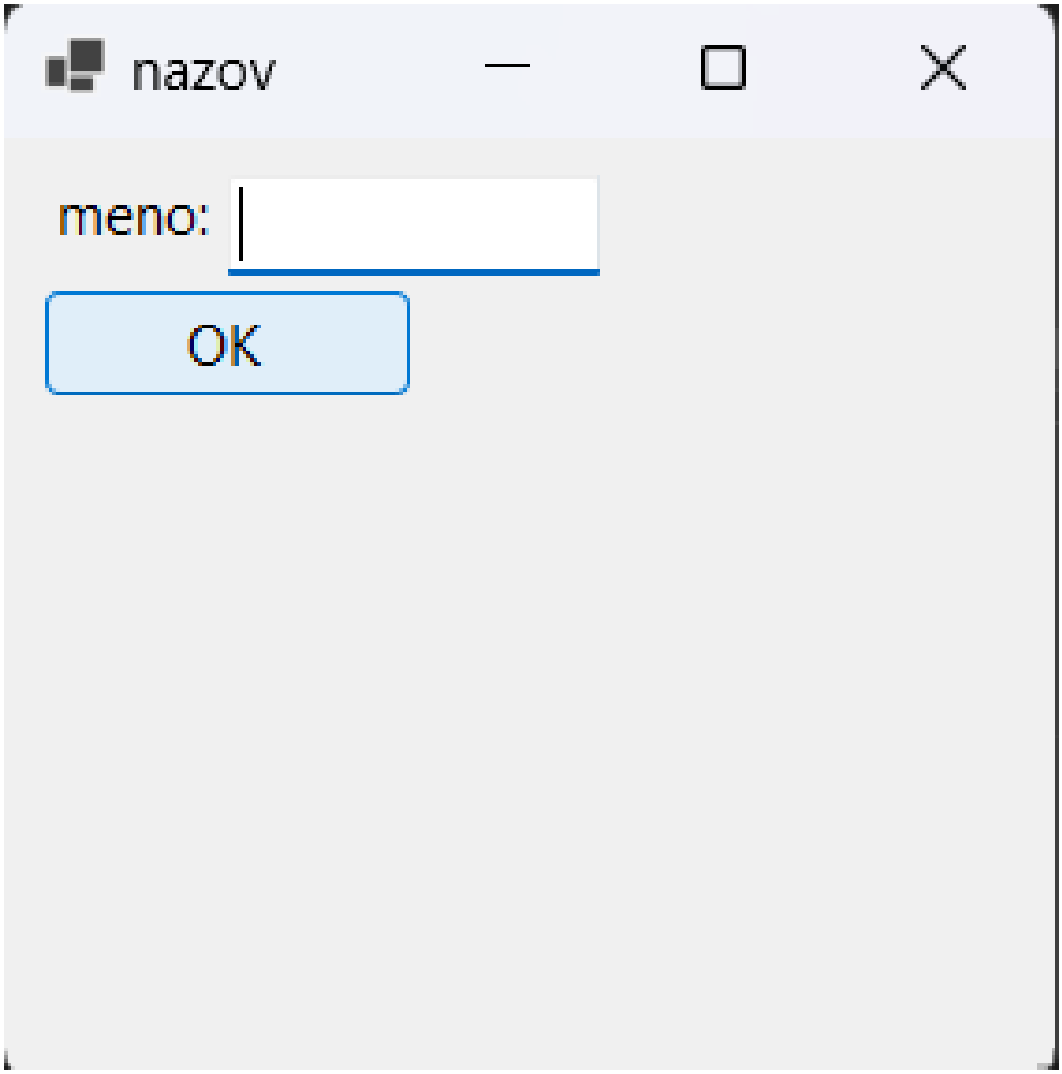
Obr. 7: MessageBox

Ďalším typom dialógového okna je **otvorenie a uloženie súboru**.

```
1 // nacitanie suboru
2 OpenFileDialog openFileDialog = new OpenFileDialog();
3 if (openFileDialog.ShowDialog() != DialogResult.OK) return;
4
5 // praca so suborom
6
7 // ulozenie suboru
8 SaveFileDialog saveFileDialog = new SaveFileDialog();
9 if (saveFileDialog.ShowDialog() != DialogResult.OK) return;
```

Pokiaľ nám žiadna z možností nevyhovuje, môžeme si vytvoriť **vlastné dialógové okno**. Jednalo by sa o **Form**.

```
Form customDialog = new Form();
customDialog.Text = "nazov";
Label label = new Label();
label.Text = "meno:";
label.Location = new Point(10, 10);
label.Size = new Size(50, 20);
customDialog.Controls.Add(label);
TextBox textBox = new TextBox();
textBox.Location = new Point(60, 10);
customDialog.Controls.Add(textBox);
Button button = new Button();
button.Text = "OK";
button.Location = new Point(10, 40);
button.Size = new Size(100, 30);
button.Click += (sender, e) => { customDialog.Close(); };
customDialog.Controls.Add(button);
customDialog.ShowDialog();
```



Obr. 8: Vlastné dialógové okno

Poznámka: Pri vlastnom dialógovom okne máme možnosť umožniť užívateľovi **interagovať s inými oknami** našej aplikácie, inal to nejde. Docielime toho tak, že **customDialog.ShowDialog();** zmeníme na **customDialog.Show();**.