C# (sí šarp)

Tomucha

Vopsach

Typy, operátory a proměnné

Typ v programování určuje, jaký druh hodnoty proměnná může obsahovat. Jinými slovy: typ definuje, jaká data jsou uložena v proměnné a jaké operace s nimi lze provádět.

Například typ může určit, že proměnná bude obsahovat celé číslo (např. 42), textový řetězec (např. "Ahoj světe"), nebo třeba desetinné číslo (např. 3.14).

Deklarace proměnné

Deklarace proměnné znamená, že vytvoříme proměnnou a přiřadíme jí nějaký typ. V C# se to dělá tak, že nejdříve napíšeme typ a potom název proměnné. Často se také rovnou uvede nějaká první hodnota.

```
int pocetDraku = 3;
string jmenoHrdiny = "Aragorn";
double silaMagie = 7.5;
```

- **`int`** znamená, že **`pocetDraku`** bude obsahovat celé číslo.
- **`string`** znamená, že **`jmenoHrdiny`** bude obsahovat text.
- **`double`** znamená, že **`silaMagie`** bude obsahovat desetinné číslo.

Jakmile proměnnou deklarujeme, můžeme ji dále používat v kódu a manipulovat s její hodnotou podle jejího typu. Pro přiřazení hodnoty do proměnné se používá operátor **`=`**.

```
int pocetDraku = 3;
pocetDraku = 5;
pocetDraku = 2;
pocetDraku = ( 1 + 7 ) / 2; // tj. 4
```

Primitivní typy

Primitivní typy jsou ty nejjednodušší a nejzákladnější typy, které jazyk C# nabízí. Jsou to typy, které nejsou složené z jiných typů a představují základní stavební kameny pro další typy a operace.

Zde jsou některé z nejčastěji používaných primitivních typů v C#:

Typ	Popis
`int`	Celé číslo. Např. 1, -42, 1000.
`double`	Desetinné číslo s dvojitou přesností. Např. 3.14, -0.001, 2.71828.
`char`	Jednotlivý znak. Např. 'A', 'z', '7'.
`bool`	Logická hodnota, která může být `true` (pravda) nebo `false` (nepravda).

NOTE `string`, nebo-li textový řetězec, není tak docela primitivní typ, protože to je ve skutečnosti zamaskovaný char[] (pole znaků).

Operátory

Operátory jsou speciální symboly, které provádějí operace na jedné nebo více hodnotách. Pomocí operátorů můžeme provádět různé druhy výpočtů nebo logických operací.

Například:

- Sčítání (`+`)
- Odčítání (`-`)
- Porovnání hodnot (`==`, `>`, `<`, `>=`, `<=`)
- Logické operace (`&&`, `||`)

Výraz

Výraz je kombinace hodnot (konstant nebo proměnných) a operátorů, která se vyhodnotí na nějakou hodnotu. V praxi to znamená, že výraz provede nějakou operaci (např. sčítání) a vrátí výsledek.

Například: `2 + 3` je výraz, který používá operátor `+` k sečtení čísel `2` a `3`, a výsledkem je `5`.

Příklady různých výrazů

1. Výraz: `5 + 3` výsledek: `8`
2. Výraz: `10 - 2` výsledek: `8`
3. Výraz: `4 * 7` výsledek: `28`
4. Výraz: `20 / 5` výsledek: `4`
5. Výraz: `7 % 3` (operátor modulo, tj. zbytek po dělení celých čísel) výsledek: `1`
6. Výraz: `2 + 3 * 4` výsledek: `14` (násobení má vyšší prioritu než sčítání, takže se nejprve násobí 3 * 4 a potom přičte 2)
7. Výraz: `10 > 3` výsledek: `true` (10 je větší než 3)
8. Výraz: `5 == 5` výsledek: `true` (5 se rovná 5)
9. Výraz: `!(2 < 1)` (operátor `!` znamená negaci - převrácení hodnoty) výsledek: `true` (2 není menší než 1, takže výraz uvnitř závorek je `false`, negací se změní na `true`)
10. Výraz: `(3 + 2) * 4` výsledek: `20` (závorky mají vyšší prioritu, takže se nejprve sečte 3 + 2 a poté vynásobí 4)

Bool výrazy

Bool výrazy (logické výrazy) jsou výrazy, které se vyhodnocují na logickou hodnotu `true` (pravda) nebo `false` (nepravda). V C# používáme pro tyto výrazy typ `bool`.

Tyto výrazy často používají logické operátory a porovnávací operátory. Bool

výrazy jsou klíčové v podmínkách, jako jsou if-else bloky, cykly a další logické rozhodování v kódu.

Operátory používané v bool výrazech

1. **Porovnávací operátory**:
    - `==` : Rovná se (např. `5 == 5` je `true`)
    - `!=` : Nerovná se (např. `5 != 3` je `true`)
    - `>` : Větší než (např. `10 > 2` je `true`)
    - `<` : Menší než (např. `2 < 10` je `true`)
    - `>=` : Větší než nebo rovno (např. `5 >= 5` je `true`)
    - `<=` : Menší než nebo rovno (např. `3 <= 4` je `true`)
2. **Logické operátory**:
    - `&&` : Logické AND - vrátí `true`, pokud jsou oba výrazy pravdivé (např. `true && true` je `true`)
    - `||` : Logické OR - vrátí `true`, pokud je alespoň jeden výraz pravdivý (např. `true || false` je `true`)
    - `!` : Logické NOT - neguje hodnotu výrazu (např. `!true` je `false`)

Příklady bool výrazů

1. Výraz: `5 > 3` výsledek: `true` (5 je větší než 3)
2. Výraz: `4 == 4` výsledek: `true` (4 se rovná 4)
3. Výraz: `6 != 7` výsledek: `true` (6 není rovno 7)
4. Výraz: `!(2 > 3)` výsledek: `true` (2 není větší než 3, takže uvnitř závorky je `false`, a negace `!` změní na `true`)
5. Výraz: `true && false` výsledek: `false` (oba výrazy musí být `true`, aby byl výsledek `true`)
6. Výraz: `true || false` výsledek: `true` (stačí, že jeden z výrazů je `true`)

Bool výrazy jsou důležité pro rozhodování v programu, protože nám umožňují

kontrolovat, zda určité podmínky platí, a podle toho řídit tok programu.

Ve výrazech (nejen bool, ale i ve výpočtech apod.) můžeme samozřejmě pracovat i s proměnnými, nejen s pevnými hodnotami.

```
int vekHrdiny = 25;
bool jeKouzloAktivni = false;
string jmenoHrdiny = "Aragorn";
```

Bool výrazy s proměnnými:

```
bool jeDospely = vekHrdiny >= 18;
```

- Výraz: **`vekHrdiny >= 18`** výsledek: **`true`** (25 je větší nebo rovno 18, takže **`jeDospely`** bude **`true`**)

```
bool muzePouzitKouzlo = jeKouzloAktivni == true;
```

- Výraz: **`jeKouzloAktivni == true`** výsledek: **`false`** (hodnota **`jeKouzloAktivni`** je **`false`**, takže **`muzePouzitKouzlo`** bude **`false`**)

Je to ovšem totéž, jako kdybychom napsali:

```
bool muzePouzitKouzlo = jeKouzloAktivni;
```

Můžeme pracovat se string proměnnou:

```
bool jeSpravneJmeno = jmenoHrdiny == "Aragorn";
```

- Výraz: **`jmenoHrdiny == "Aragorn"`** výsledek: **`true`** (hodnota proměnné **`jmenoHrdiny`** je **`"Aragorn"`**, takže **`jeSpravneJmeno`** bude **`true`**)

```
bool jeDospelyAKouzloAktivni = jeDospely &&
jeKouzloAktivni;
```

- Výraz: **`jeDospely && jeKouzloAktivni`** výsledek: **`false`** (výraz **`jeDospely`** je **`true`**, ale **`jeKouzloAktivni`** je **`false`**, takže celý výraz bude **`false`**)

```
bool maSpravneJmenoNeboJeDospely = jeSpravneJmeno ||
jeDospely;
```

- Výraz: **`jeSpravneJmeno || jeDospely`** výsledek: **`true`** (alespoň jeden z výrazů **`jeSpravneJmeno`** nebo **`jeDospely`** je **`true`**, takže celý výraz bude **`true`**)

Když pracujeme s proměnnými v bool výrazech, hodnota výrazu závisí na hodnotách těchto proměnných. Kombinací proměnných a operátorů můžeme vytvářet složitější logické podmínky a rozhodovat se podle nich, co v programu dělat dál.

Někdy může být bool výraz velmi složitý. Vyplatí se ho rozebrat na menší kousky, ty si uložit do proměnných a z nich pak složit výsledek.

Funkce

Funkce (někdy se jim říká metody) jsou základní stavební kameny programování. Umožňují nám zabalit určitý kus kódu, který plní konkrétní úkol, do jednoho bloku. Tento blok pak můžeme opakovaně používat v programu, což zjednodušuje a zpřehledňuje náš kód.

Definice funkce

Definice funkce je místo v kódu, kde specifikujeme, co funkce dělá (resp. co BUDE dělat, až ji někdo zavolá). Obsahuje:

1. Název funkce (pomocí kterého ji budeme volat).
2. Parametry (volitelně) - hodnoty, které můžeme funkci předat
3. Návratový typ (jaký druh hodnoty funkce vrací).
4. Tělo funkce - blok kódu, který se provede, když funkci zavoláme.

Příklad jednoduché funkce:

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

- `int` před názvem funkce `SectiCisla` je návratový typ, což znamená, že funkce vrací celé číslo (`int`).
- `int a, int b` jsou parametry funkce, což jsou hodnoty, které funkci předáme.
- `return vysledek;` je příkaz, který ukončí funkci a vrátí hodnotu `vysledek` zpět tam, odkud byla funkce zavolána.

Rozdíl mezi definicí a zavoláním funkce

- **Definice funkce** je to, co jsme právě napsali výše - specifikujeme, co funkce udělá až ji zavoláme, jaké má parametry a co vrací.
- **Zavolání funkce** je to, když někde v programu tuto funkci použijeme, abychom provedli její kód a získali její výsledek.

Příklad zavolání funkce:

```
int soucet = SectiCisla(5, 3); // Zavolání funkce s
parametry 5 a 3
```

V tomto případě se funkce `SectiCisla` provede s hodnotami `5` a `3`, a výsledkem

bude číslo **8**, které se uloží do proměnné **soucet**.

Co znamená **return**?

Příkaz **return** v těle funkce:

1. Ukončí provádění funkce.
2. Vrátí hodnotu uvedenou za **return** zpět do místa, odkud byla funkce zavolána.

Příklad:

```
int Nasob(int a, int b)
{
    return a * b;
}
```

Když zavoláme funkci **Nasob(4, 5)**, funkce okamžitě vrátí hodnotu **20** (4 * 5) a ukončí se.

return na více místech ve funkci

Pokud je ve funkci více příkazů **return**, funkce se ukončí při prvním z nich, který se vykoná. To znamená, že se zbytek funkce už neprovede. To platí i v případě, kdy je return uveden uvnitř cyklu (for, while, …).

Příklad:

```
int ZkontrolujCislo(int cislo)
{
    if (cislo < 0)
    {
        // Pokud je číslo záporné, vrátí -1 a funkce
skončí
        return -1;
    }

    if (cislo == 0)
    {
        // Pokud je číslo 0, vrátí 0 a funkce skončí
        return 0;
    }
    // Pokud je číslo kladné, vrátí 1 a funkce skončí
    return 1;
}
```

V tomto příkladu se funkce ukončí po prvním **`return`**, který se provede. Například, když zavoláme **`ZkontrolujCislo(-5)`**, funkce vrátí **`-1`** a zbytek kódu se už neprovede. Neexistuje způsob, jak by se mohl return zavolat vícekrát.

Užití funkce ve výrazu

Funkci můžeme využít jako součást výrazu stejně jako běžné operátory nebo proměnné. Když funkci zavoláme ve výrazu, její výsledek se stane hodnotou, kterou můžeme dále použít.

Představme si funkci **`Nasob`**, kterou jsme definovali dříve:

```
int Nasob(int a, int b)
{
    return a * b;
}
```

Tuto funkci můžeme použít ve výrazech různými způsoby:

Použití funkce přímo při přiřazení hodnoty do proměnné

```
int vysledek = Nasob(4, 5);
```

- Výraz: **`Nasob(4, 5)`** výsledek: **`20`**
- Hodnota **`20`** se přiřadí do proměnné **`vysledek`**.

Použití funkce jako součást složitějšího výrazu

```
int soucet = Nasob(2, 3) + Nasob(4, 5);
```

- Výraz: **`Nasob(2, 3) + Nasob(4, 5)`** výsledek: **`6 + 20 = 26`**
- Hodnota **`26`** se přiřadí do proměnné **`soucet`**.

Použití funkce v podmínce

```
if (Nasob(3, 3) > 8)
{
    Console.WriteLine("Výsledek je větší než 8.");
}
```

- Výraz: **`Nasob(3, 3) > 8`** výsledek: **`9 > 8`**, což je **`true`**, takže se vykoná **`Console.WriteLine`**.

Použití funkce jako argument jiné funkce

Funkci **`Nasob`** můžeme použít jako argument pro další volání funkce (třeba i úplně jiné).

```
int vysledek = Nasob(Nasob(2, 3), 4);
```

- První volání: **`Nasob(2, 3)`** výsledek: **`6`** (2 * 3)
- Druhé volání: **`Nasob(6, 4)`** výsledek: **`24`** (6 * 4)
- Výsledná hodnota proměnné **`vysledek`** je **`24`**.

Funkce jsou tedy nejen způsobem, jak zabalit a znovu použít kód, ale také nástrojem pro stavbu složitějších výpočtů a logických kontrol v programu.

Pole

Pole (v angličtině "array") je datová struktura, která umožňuje uložit více hodnot stejného typu pod jedním názvem. Každá z těchto hodnot (tzv. prvek pole) je dostupná prostřednictvím indexu (čísla), který určuje pozici prvku v poli.

Pole je užitečné, když potřebujeme pracovat s více hodnotami stejného typu a chceme je mít pohromadě v jednom kontejneru. Například můžeme mít pole čísel, textů nebo objektů.

Deklarace pole

Pro deklaraci pole musíme specifikovat typ prvků, které pole bude obsahovat, a počet prvků.

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

- **`int[]`** znamená, že pole **`cisla`** bude obsahovat prvky typu **`int`** (celá čísla).
- **`new int[5]`** vytvoří pole s 5 prvky, které jsou inicializovány na výchozí hodnoty (pro **`int`** je to **`0`**, ve všech prvcích bude **`0`**).

Inicializace pole

Pole můžeme rovnou naplnit hodnotami při jeho deklaraci:

```
int[] cisla = { 1, 2, 3, 4, 5 };
```

- Pole **`cisla`** obsahuje pět čísel: 1, 2, 3, 4, 5.

Přístup k prvkům pole

K jednotlivým prvkům pole přistupujeme pomocí jejich indexu. Indexy začínají od nuly, což znamená, že první prvek má index **`0`**, druhý prvek index **`1`**, a tak dále.

Příklad:

```
int prvniCislo = cisla[0]; // Získá první prvek pole,
což je 1
cisla[2] = 10;             // Změní třetí prvek pole na
hodnotu 10
```

- **`cisla[0]`** vrací hodnotu prvního prvku (v tomto případě **`1`**).
- **`cisla[2] = 10`** změní třetí prvek (který měl hodnotu **`3`**) na **`10`**.

Délka pole

Délku pole (tedy počet prvků, které pole obsahuje) zjistíme pomocí vlastnosti **`Length`**.

Příklad:

```
int delkaPole = cisla.Length; // Vrátí hodnotu 5,
protože pole má pět prvků
```

Jako všechno - i pole můžeme použít při stavbě výrazu:

```
bool faktDlouhyPole = cisla.Length > 100; //
faktDlouhyPole bude `false`,
                                          // protože
prvků je v `cisla` jen 5.
```

Podmínky: `if` a `else`

Podmínky **`if`** a **`else`** jsou základní stavební kameny rozhodovací logiky v programování. Umožňují nám provádět různé části kódu na základě toho, zda určitá podmínka platí (**`true`**), nebo neplatí (**`false`**).

`if` podmínka

Příkaz **`if`** slouží k tomu, abychom vykonali určitý blok kódu pouze tehdy, pokud je daná podmínka pravdivá (**`true`**).

Příklad:

```
int pocetZivotu = 3;

if (pocetZivotu > 0)
{
    Console.WriteLine("Hrdina je stále naživu.");
}
```

- Výraz **`pocetZivotu > 0`** je podmínka. Pokud je **`pocetZivotu`** větší než 0, podmínka je **`true`**.
- Pokud je podmínka pravdivá, vykoná se kód uvnitř bloku **`{ }`**, který vypíše zprávu "Hrdina je stále naživu."

`else` větev

Příkaz **`else`** se používá k vykonání alternativního bloku kódu, pokud podmínka v **`if`** není pravdivá (**`false`**).

Příklad:

```
int pocetZivotu = 0;

if (pocetZivotu > 0)
{
    Console.WriteLine("Hrdina je stále naživu.");
}
else
{
    Console.WriteLine("Hrdina je mrtvý.");
}
```

- Pokud je **`pocetZivotu > 0`** **`true`**, vykoná se první blok kódu a vypíše se "Hrdina je stále naživu."
- Pokud je **`pocetZivotu > 0`** **`false`** (což je v tomto případě pravda, protože **`pocetZivotu`** je 0), vykoná se blok kódu uvnitř **`else`** a vypíše se "Hrdina je mrtvý."

`else if` větev

Někdy potřebujeme otestovat více podmínek. K tomu slouží příkaz **`else if`**, který nám umožní přidat další podmínky mezi **`if`** a **`else`**.

Příklad:

```
int pocetZivotu = 1;

if (pocetZivotu > 1)
{
    Console.WriteLine("Hrdina je v dobrém stavu.");
}
else if (pocetZivotu == 1)
{
    Console.WriteLine("Hrdina je zraněn.");
}
else
{
    Console.WriteLine("Hrdina je mrtvý.");
}
```

- Pokud je **`pocetZivotu > 1`**, vypíše se "Hrdina je v dobrém stavu."
- Pokud první podmínka není pravdivá, ale **`pocetZivotu == 1`** je pravdivé, vykoná se kód v **`else if`** a vypíše se "Hrdina je zraněn."
- Pokud ani jedna z předchozích podmínek není pravdivá, vykoná se blok v **`else`** a vypíše se "Hrdina je mrtvý."

Vnořené podmínky

Podmínky můžeme také vnořovat, což znamená, že do jednoho bloku **`if`** můžeme vložit další podmínku **`if`**.

Příklad:

```
int pocetZivotu = 2;
bool maStit = true;

if (pocetZivotu > 0)
{
    if (maStit)
    {
        Console.WriteLine("Je naživu a má štít.");
    }
    else
    {
        Console.WriteLine("Je naživu, ale nemá štít.");
    }
}
else
{
    Console.WriteLine("Hrdina je mrtvý.");
}
```

- Nejprve se zkontroluje, zda je hrdina naživu (**`pocetZivotu > 0`**).
- Pokud ano, zkontroluje se, zda má hrdina štít (**`maStit`**).
- Na základě toho se vypíše příslušná zpráva.

Cyklus `while`

Cyklus **`while`** opakuje blok kódu, dokud je daná podmínka pravdivá (**`true`**). Tento typ cyklu používáme, když nevíme předem, kolikrát bude potřeba cyklus vykonat, ale chceme, aby běžel, dokud platí určitá podmínka.

Příklad:

```
int pocetKroku = 0;

while (pocetKroku < 5)
{
    Console.WriteLine("Hrdina udělal krok.");
    pocetKroku++;
}
```

- **`while (pocetKroku < 5)`** znamená, že cyklus se bude opakovat, dokud hodnota **`pocetKroku`** bude menší než 5.
- Uvnitř cyklu se pokaždé vypíše "Hrdina udělal krok." a hodnota **`pocetKroku`** se zvýší o 1.
- Cyklus skončí, jakmile **`pocetKroku`** dosáhne hodnoty 5.

Nekonečný cyklus

Pokud podmínka v cyklu **`while`** nikdy nebude nepravdivá, cyklus se nikdy nezastaví. Tomu se říká nekonečný cyklus.

```
while (1==1)
{
    Console.WriteLine("Tento cyklus běží navždy.");
}
```

- Tento cyklus se nikdy nezastaví, protože podmínka **`1==1`** je vždy splněna.

Cyklus `for`

Cyklus **`for`** je užitečný, když předem víme, kolikrát chceme cyklus vykonat. Typicky se používá pro iteraci přes pole nebo provádění nějaké akce přesně stanovený počtkrát.

Struktura cyklu **`for`** vypadá takto:

```
for (int i = 0; i < 5; i++)
{
    Console.WriteLine("Toto je iterace číslo " + i);
}
```

- **`int i = 0`** je inicializace, kde **`i`** je počítadlo cyklu (tzv. řídící proměnná) a zde začíná na hodnotě **`0`** (může začít kdekoliv)
- **`i < 5`** je podmínka, která určuje, dokud se cyklus bude opakovat. V tomto případě, dokud je **`i`** menší než 5 (opět může jít o libovolný bool výraz, obvykle v něm ale figuruje řídící proměnná)
- **`i++`** znamená, že se hodnota **`i`** po každé iteraci zvýší o 1 (opět ze může být cokoliv, třeba zvýšení o dvě: **`i+=2`**)

Každý průchod cyklem (iterace) se vypíše zpráva a **`i`** se zvýší o 1, dokud nedosáhne hodnoty 5, kdy se cyklus ukončí.

Použití cyklu `for` k iteraci přes pole

Cyklus **`for`** je často používán k procházení (iteraci) polem, kde potřebujeme přistupovat ke každému prvku pomocí jeho indexu.

Příklad:

```
int[] cisla = { 1, 2, 3, 4, 5 };

for (int i = 0; i < cisla.Length; i++)
{
    Console.WriteLine("Prvek na indexu " + i + " je " +
cisla[i]);
}
```

- **`cisla.Length`** vrací délku pole, takže cyklus běží tak dlouho, dokud je **`i`** menší než počet prvků v poli.
- **`cisla[i]`** vrací prvek pole na pozici **`i`**.

Divnofory

For může v praxi vypadat různě. To že se často používá pro počítání od 0 do délky pole neznamená, že je to jedinné použití.

```
for (;true;) {  /* Nekonečná smyčka */  }

for (int a = 10; a>0; a--) {  /* 10, 9, 8, 7 ... */  }

for (int a = 0; a < 10; a+=2)  {  /* 0, 2, 4 ... */  }
```

Třídy a objekty

TODO

Kolekce

TODO

Práce se soubory

TODO

Příloha 1: Užitečné funkce

V C# existuje řada vestavěných funkcí, které nám usnadňují práci s textem, vstupem a výstupem. V této kapitole si představíme některé z nejčastěji používaných funkcí, jako jsou `WriteLine`, `ReadLine`, `String.Contains`, a další.

`Console.WriteLine`

Funkce `Console.WriteLine` se používá k výpisu textu (nebo jiných hodnot) do konzole. Je to základní způsob, jak zobrazit výstup programu uživateli.

Příklad:

```
Console.WriteLine("Vítej, hrdino!");
Console.WriteLine(42);
Console.WriteLine("Tvoje skóre je: " + 100);
```

- První volání vypíše text "Vítej, hrdino!".
- Druhé volání vypíše číslo `42`.
- Třetí volání kombinuje text a číslo, takže výsledkem bude "Tvoje skóre je: 100".

`Console.WriteLine` automaticky přidává nový řádek za každý výstup, takže každý výpis bude na novém řádku.

String Interpolation

Při výpisu stringu na uživatelské rozhraní se často využívá String Interpolation, což je způsob, jak jednoduše a přehledně vkládat hodnoty proměnných nebo výrazy přímo do řetězců. Místo toho, abychom používali spojování řetězců pomocí operátoru `+`, můžeme pomocí znaků `$` a složených závorek `{}` vložit hodnoty proměnných přímo do textu. Tento přístup zlepšuje čitelnost kódu a snižuje pravděpodobnost chyb při sestavování řetězců. Příklad:

```
string jmeno = "Aragorn";
int vek = 87;
string zprava = $"Hrdina {jmeno} je {vek} let starý.";
Console.WriteLine(zprava);
```

Výstupem bude: "Hrdina Aragorn je 87 let starý."

Console.ReadLine

Funkce **Console.ReadLine** se používá k načtení vstupu od uživatele. Tato funkce čeká, až uživatel zadá nějaký text a stiskne klávesu Enter, a poté vrátí tento text jako řetězec (**string**).

Příklad:

```
Console.WriteLine("Zadej své jméno:");
string jmeno = Console.ReadLine();
Console.WriteLine("Vítej, " + jmeno + "!");
```

- Program nejprve vypíše "Zadej své jméno:".
- Poté čeká na vstup uživatele, který se uloží do proměnné **jmeno**.
- Nakonec vypíše přivítání s tímto jménem, například "Vítej, Aragorn!".

String.Contains

Funkce **String.Contains** se používá k ověření, zda určitý podřetězec (kus textu) je obsažen v jiném řetězci. Vrací hodnotu **true**, pokud podřetězec existuje, a **false**, pokud ne.

Příklad:

```
string hrdina = "Aragorn";
```

```
bool obsahujeA = hrdina.Contains("A");
bool obsahujeZ = hrdina.Contains("Z");

Console.WriteLine("Obsahuje 'A': " + obsahujeA); //
Výstup: true
Console.WriteLine("Obsahuje 'Z': " + obsahujeZ); //
Výstup: false
```

- **`hrdina.Contains("A")`** vrátí **`true`**, protože text "Aragorn" obsahuje znak "A".
- **`hrdina.Contains("Z")`** vrátí **`false`**, protože "Aragorn" neobsahuje znak "Z".

`Console.Write`

Funkce **`Console.Write`** je podobná **`Console.WriteLine`**, ale nevkládá na konec výstupu nový řádek. Další výstup tedy bude pokračovat na stejném řádku.

Příklad:

```
Console.Write("Hrdina ");
Console.Write("Aragorn ");
Console.Write("bojuje.");
```

- Výstup bude: "Hrdina Aragorn bojuje." vše na jednom řádku. Někdy se to hodí.

`String.ToUpper` a `String.ToLower`

Tyto funkce slouží k převodu celého řetězce na velká nebo malá písmena.

Příklad:

```
string hrdina = "Aragorn";
string velkaPismena = hrdina.ToUpper();
string malaPismena = hrdina.ToLower();

Console.WriteLine(velkaPismena); // Výstup: ARAGORN
Console.WriteLine(malaPismena);  // Výstup: aragorn
```

- **`ToUpper()`** převede všechny znaky na velká písmena.
- **`ToLower()`** převede všechny znaky na malá písmena.

String.Length

Tato vlastnost vrací délku řetězce, tedy počet znaků, které obsahuje.

Příklad:

```
string hrdina = "Aragorn";
int delka = hrdina.Length;

Console.WriteLine("Délka jména: " + delka); // Výstup:
Délka jména: 7
```

- **`hrdina.Length`** vrací **`7`**, protože jméno "Aragorn" má 7 znaků.

String.Substring

Tato funkce slouží k získání podřetězce z řetězce, začínající na určité pozici a případně o určité délce.

Příklad:

```
string hrdina = "Aragorn";
```

```
string castJmena = hrdina.Substring(0, 4);

Console.WriteLine(castJmena); // Výstup: Arag
```

- **`hrdina.Substring(0, 4)`** vrátí první čtyři znaky "Aragorn", což je "Arag".

String.Split

Metoda **`Split`** slouží k rozdělení textového řetězce (string) na pole menších řetězců, podle zadaného oddělovače (např. mezera, čárka apod.). Používá se například, když potřebujeme rozdělit větu na jednotlivá slova.

```
string kouzelneZaklinadlo = "Abra Kadabra Alakazam";
string[] slova = kouzelneZaklinadlo.Split(' ');

foreach (string slovo in slova)
{
    Console.WriteLine(slovo);
}
```

String.Trim

Metoda **`Trim`** odstraňuje bílé znaky (mezery, tabulátory, nové řádky) z začátku a konce řetězce. Je užitečná, když potřebujeme vyčistit vstup uživatele od náhodně přidaných mezer.

```
string zaklinadlo = "  Abrakadabra!  ";
string cisteZaklinadlo = zaklinadlo.Trim();

Console.WriteLine(cisteZaklinadlo); // Vypíše
"Abrakadabra!"
```

Random

Třída **Random** slouží k generování náhodných čísel. Je často využívána, například když chceme náhodně rozhodnout výsledek nějaké akce (např. hod kostkou).

```
Random nahodnyGenerator = new Random();

// Generuje číslo mezi 1 a 6(! ne 7!)
int nahodneCislo = nahodnyGenerator.Next(1, 7);

Console.WriteLine($"Hodil jsi kostkou a padlo číslo:
{nahodneCislo}");
```

Int.Parse

Metoda **Int.Parse** slouží k převodu textového řetězce, který obsahuje číslo, na celočíselnou hodnotu (typ **int**). Je užitečná, když máme například číslo jako text a potřebujeme s ním provést nějakou matematickou operaci.

```
string textoveCislo = "42";
int cislo = Int.Parse(textoveCislo);

Console.WriteLine($"Text '{textoveCislo}' bylo převedeno
na číslo {cislo}");
```

Thread.Sleep

Metoda **Thread.Sleep** zastaví provádění programu na zadaný počet milisekund. To může být užitečné například při simulaci čekání nebo při zpomalování opakujících se akcí.

```
Console.WriteLine("Příprava kouzla...");
```

```
Thread.Sleep(2000); // Zastaví program na 2 sekundy

Console.WriteLine("Kouzlo je připraveno!");
```

Příloha 2: Slovníček programátorských pojmů

Deklarace

Deklarace je proces, kdy programu říkáme, že budeme používat nějakou proměnnou nebo funkci, a specifikujeme její jméno a typ (druh hodnoty, kterou může obsahovat).

Inicializace

Inicializace je proces, kdy proměnné přiřadíme počáteční hodnotu. Bez inicializace může být proměnná neplatná nebo obsahovat nečekané hodnoty.

Typ

Typ určuje, jaké druhy hodnot může proměnná obsahovat a jak s nimi může být manipulováno (např. čísla, texty, pravdivostní hodnoty).

Primitivní typy

Primitivní typy jsou základní datové typy, které jsou přímo podporovány jazykem, jako jsou celé číslo (int), desetinné číslo (float), znak (char) nebo logická hodnota (bool).

Operátory

Operátory jsou speciální symboly nebo klíčová slova, které umožňují provádět operace s hodnotami, jako je sčítání, porovnávání nebo logické operace.

Výraz

Výraz je kombinace hodnot, proměnných, operátorů a funkcí, která se vyhodnotí do jedné hodnoty. Výrazy mohou být aritmetické, logické nebo kombinované.

Bool výraz

Bool výraz je výraz, který se vyhodnotí na jednu ze dvou logických hodnot: pravda (true) nebo nepravda (false). Bool výrazu lze použít kdekoliv, kde se bool očekává - while, for, if, …

Funkce

Funkce je blok kódu, který provádí konkrétní úkol. Může přijímat vstupní hodnoty (parametry) a vracet výstupní hodnotu.

Návratový typ

Návratový typ určuje, jaký typ hodnoty funkce vrátí, když je dokončena. Může to být například číslo, text nebo logická hodnota.

Tělo funkce

Tělo funkce je blok kódu uvnitř funkce, kde jsou definovány instrukce, které se vykonají, když je funkce zavolána.

Pole

Pole je datová struktura, která umožňuje uložit více hodnot stejného typu do jedné proměnné. Hodnoty v poli jsou uspořádány a přístupné pomocí indexů.

Index

Index je číslo, které označuje konkrétní pozici v poli. Většinou začíná od nuly, což znamená, že první prvek má index 0.

Nekonečný cyklus

Nekonečný cyklus je cyklus, který se nikdy nezastaví, protože podmínka pro jeho ukončení není nikdy splněna.

String Interpolation

String Interpolation je způsob, jakým můžeme vložit hodnoty proměnných přímo do textového řetězce, aniž bychom museli řetězce a proměnné spojovat.

Substring

Substring je část textového řetězce. Můžeme získat podřetězec z většího řetězce určením počáteční a koncové pozice.

Cyklus

Cyklus je konstrukce, která umožňuje opakovaně vykonávat blok kódu, dokud je splněna určitá podmínka. Existují různé typy cyklů, jako je `for`,

`while` nebo `do-while`.

Definice

Definice je proces, kdy nejen deklarujeme proměnnou nebo funkci, ale také jí dáváme konkrétní obsah nebo implementaci. Definice proměnné zahrnuje její deklaraci a inicializaci, zatímco definice funkce zahrnuje její jméno, parametry, návratový typ a tělo.

Proměnná

Proměnná je pojmenovaná oblast paměti, kde se ukládají hodnoty, které mohou být během běhu programu měněny.

Konstantní hodnota

Konstantní hodnota je hodnota, která se po své inicializaci nemůže změnit. V kódu se používá pro hodnoty, které zůstávají po celý běh programu stejné.

Podmínka

Podmínka je výraz, který se vyhodnocuje na pravdivostní hodnotu (true/false) a rozhoduje o tom, zda se určitý blok kódu vykoná, nebo ne.

Parametr

Parametr je hodnota, kterou funkce přijímá jako vstup. Parametry jsou specifikovány při deklaraci funkce a používají se při jejím volání.

Blok kódu

Blok kódu je skupina instrukcí, která je uzavřena mezi složené závorky `{}` a vykonává se jako celek. Typicky se používá v podmínkách, cyklech nebo funkcích.

Komentář

Komentář je text v kódu, který slouží pro vysvětlení nebo poznámky. Komentáře se nevyhodnocují a nemají vliv na běh programu.

Příkaz

Příkaz je základní jednotka kódu, která provádí konkrétní akci. Většinou končí středníkem a říká programu, co má udělat.

Debugging

Debugging je proces hledání a odstraňování chyb v programu. Tento proces může zahrnovat krokování kódu, kontrolu hodnot proměnných a analyzování chybových hlášek.

Kompilace

Kompilace je proces překladu zdrojového kódu do spustitelného formátu, kterému rozumí počítač. Tento proces provádí kompilátor.

Výjimka

Výjimka je událost, která se stane během běhu programu a signalizuje, že došlo k chybě. Výjimky lze zachytit a zpracovat, aby se předešlo zhroucení programu.