Základy programování 3: C#

Martin Kauer

Palacký University
Olomouc
Czech Republic

3. října 2018

Klíčové slovo var

- C# je staticky typovaný jazyk.
- Kompilátor ale zvládne odvodit *většinu* typů z kontextu = tzv. implicitní typování.
- Pozor, není to dynamické typování!
- Můžeme použít klíčové slovo `var` v definici proměnné.

```
var num = 3; // kompilator spravne urci typ promenne num jako int
var num = 15U; // zde odvodi uint
var str = "Hello"; // zde odvodi typ string
var nums = {1, 2, 3}; // CHYBA, u pole s inicializacni casti var
    nelze pouzit
```

- Je doporučené používat implicitní typování kde je to možné.

Jednorozměrná pole

- Zápis rozdílný od C: `typ[]`, např `int[]`.
- Při vytvoření musíme zadat velikost = `int`.

```
int[] nums = new int[3];
```

- Velikost lze zadat jako proměnnou.
- Lze při definici přímo inicializovat pomocí tzv. incializační částí, ta se píše do složených závorek.

```
int[] nums = {1, 2, 3}; // kompilator odvodi velikost pole
```

- Pole je vlastní typ, tzn. má vlastní členy a metody.
- Např. vlastnost `Length` vrací velikost pole.

```
Console.WriteLine(nums.Length); // vypise velikost pole
```

- Podobně jako v C pole nelze kopírovat přiřazením.
- Pro práci s jednorozměrnými existuje třída `System.Array`.

Výčtové typy

- Výčtové typy zastupují související množinu pojmenovaných celočíselných hodnot.
- Značí se klíčovým slovem `enum`.

```
enum NazevVyctovehoTypu
{
    JmenoPrvniHodnoty = celeCislo1,
    JmenoDruheHodnoty = celeCislo2
}

enum Gender
{
    Male = 1,
    Female = 2
}
```

Větvení

- Obdobně jako v C máme podmínkový operátor `?:`, `if`, `switch`.
- Oproti C několik rozdílů.
- Podmínková část musí vracet pravdivostní hodnotu tj. typ `bool`.
- Switch je mnohem flexibilnější, v aktuální verzi C# se může rozhodovat dle libovolného výrazu, nejen celočíselného typu jako v C.

```
if(podminka)
{
    prikazySpleno;
}
else if(podminka)
{
    prikazySplneno;
}
else
{
    prikazyJinak;
}
```

Větvení pomocí `switch`

- Konstrukt `switch` umožňuje rozhodování na základě jednoho výrazu (nesmí být `null`).
- Větve musejí být ukončeny návratovým příkazem `break` popř. `return`.
- Větve nemusí být výlučné, ale provede se pouze první odpovídající.
- Výchozí větev se značí `default`.
- V aktuální verzi C# můžeme používat rozšířený *pattern matching*.

```
switch(expr)
{
    case kostanta1:
        ... break;
    case typ identifikator:
        ... break; // lze pouzit identifikator
    case typ identifikator when podminka: // v podmince lze pouzit
       identifikator
        ... break; // lze pouzit identifikator
    case null:
        ... break;
    default:
        ... break;
}
```

Větvení pomocí `switch`

```
object obj = new int[] { 2, 4, 6 };
switch(obj)
{
    case 1:
        Console.WriteLine("obj is a number 1");
        break;
    case int[] vals when vals.Length == 3:
        Console.WriteLine("P1 obj is array of ints with length == 3");
        break;
    case int[] vals:
        Console.WriteLine($"P2 obj is array of ints with length == {
            vals.Length}");
        break;
    default:
        Console.WriteLine("Default case");
        break;
}
```

Cykly

- `for`, `while`, `do-while` jako v C.
- Deklarace řídících proměnných často v inicializační části cyklu s použitím implicitního typování.

```
int[] nums = { 2, 4, 6 };
for (var i = 0; i < nums.Length; i++)
    Console.WriteLine($"Prvek na indexu {i} je {nums[i]}");

int[] nums = { 2, 4, 6 };
var i = 0;
while (i < nums.Length)
{
    Console.WriteLine($"Prvek na indexu {i} je {nums[i]}");
    i++;
}
```

Cykly

- Navíc máme cyklus `foreach`.
- Nelze použít vždy, ale jen na výčtové kolekce.
- Lze použít na téměř všechny standardní kolekce včetně polí, listů, slovníku, množin, apod.
- Zápis obecně:

```
foreach (typ identifikator in kolekce)
{
    prikazy;
}

int[] nums = { 2, 4, 6 };
foreach (var item in nums)
    Console.WriteLine($"Prvek pole {item}");
```

- Nepoužívá se explicitní indexace.
- Průchod výčtovou kolekcí je realizován samotnou kolekcí.

OOP v C#

- Třídu můžeme naivně chápat jako popis objektu.
- Třída popisuje všechny členy tj. nejen sloty a vlastnosti objektu, ale také metody, apod.
- Přístup ke členům pomocí operátoru tečky.
- Každá metoda, slot, vlastnost, apod. v C# patří k nějaké třídě.
- V C# se základní třída objektového systému jmenuje `System.Object`.
- Primitivní typ `object`, za běhu programu je vše konvertibilní na `object`.
- Máme pouze jednoduchou dědičnost tříd, interfaců můžeme implementovat libovolný počet.
- Pokud není uvedený předek, je automaticky doplněn `System.Object`.
- Každá třída je většinou definována v samostatném souboru.
- Objekt, který je popsán třídou se nazýva její instancí.
- Vytvoření nové instance pomocí klíčového slova `new`.
- Přístup k aktuální instanci pomocí `this`.

```
var my = new MyClass(arg1, arg2, ...);
```

Základní definice třídy

```
pristupnost modifikatory class MyClass : Predek, Interface1,
    Interface2, ...
{
    pristupnost modifikatory typ nazevSlotu = hodnota;
    pristupnost modifikatory typ NazevVlastnosti { get; set; }

    pristupnost MyClass() // konstruktor
    {
        // prikazy konstruktoru
    }

    pristupnost ~MyClass() // destruktor, pozor pri jeho pouzivani!
    {
        // prikazy destruktoru
    }

    pristupnost modifikatory navratovyTyp MyMethod()
    {
        // prikazy metody
    }
}
```

Přístupnosti a modifikátory

- Základní přístupnosti:
  - `private` - přístupné pouze zevnitř třídy;
  - `protected` - jako `private` + přístupné i zevnitř potomků;
  - `internal` - přístupné ze stejného sestavení;
  - `public` - přístupné odkudkoli.
- Modifikátory:
  - `sealed` - označuje třídu, která nemůže mít potomky;
  - `abstract` - u tříd značí neinstancovatelnost, u metod chybějící tělo, které potomci musí implementovat;
  - `static` - označuje člen, který nepatří instancím, ale celému typu, u tříd značí neinstancovatelnost, lze nahlížet jako na `sealed` + `abstract`
  - `virtual` - u metod, vlastností, indexerů nebo událostí značí možnost přepsat je v potomcích;
  - `override` - v potomcích označuje přepsání virtuálních členů;
  - `new` - u členů značí jejich překrytí v potomcích.

Sloty

- Ve slotech ukládáme infomace vztahující se ke konkrétní instanci.
- Existují *statické* sloty, které se vztahují ke konkrétní třídě (typu).
- Sloty mohou být konstantní (`const`) a ty jsou vždy implicitně statické. Takovým slotům lze přiřadit hodnotu pouze v inicializační části.
- Sloty mohou být označeny jako pouze ke čtení pomocí `readonly`. Takovým slotům lze přiřadit hodnotu pouze v inicializační částí nebo v konstruktoru dané třídy.
- **Jinak sloty budeme vždy dělat privátní!**

```
pristupnost modifikatory typ nazevSlotu = hodnota;
```

```
class MyClass
{
    private int myIntSlot = 2;
    private string myStringSlot = "Hello";
}
```

Sloty

```
class MyClass
{
    public const int MyConstSlot = 1;
    private readonly int myReadonlySlot = 2;

    public MyClass()
    {
        this.myReadonlySlot = 3; // V poradku, jsme v konstruktoru
        this.MyConstSlot = 2; // CHYBA! konstanta nelze zmenit
    }
}
```

Vlastnosti (Properties)

- Vlastnosti definuje třída a mají většinou dvě částí: `get` a `set`.
- Obě části mají svá těla.
- Tělo `get` se vykoná, když požadujeme hodnotu vlastnosti.
- Tělo `set` se vykoná, když vlastnosti přiřazujeme hodnotu a lze k ní přistupovat klíčovým slovem `value`.
- Oproti slotům nemusí být vlastnost spojená s žádným slotem.
- Naopak hodnota vlastnosti může záviset na více hodnot slotů.
- Většinou jsou vlastnosti veřejné.
- Přes vlastnosti se často kontrolovaně zveřejňují sloty.
- Pro jednopříkazové těla můžeme použít zkrácený zápis: `get => navrat;`

```
pristupnost typ NazevVlastnosti
{
    pristupnost get { teloGetu; }
    pristupnost set { teloSetu; }
}
```

Vlastnosti (Properties)

```
class MyClass()
{
    private int mySlot = 2;
    public int MyProperty
    {
        get { return mySlot; }
        set
        {
            // zde muzeme udelat kontroly na validitu vstupu
            mySlot = value;
            if(value % 2 == 0)
                Console.WriteLine("MyProperty is set to be even");
            else
                Console.WriteLine("MyProperty is set to be odd");
        }
    }
}
```

Vlastnosti (Properties)

- Existují tzv. *automatické* vlastnosti.
- Pro ně kompilátor automaticky vytvoří slot daného typu.
- Přístup k tomuto slotu je možný jen přes vlastnost.
- Těla `get` a `set` jsou generovaná kompilátorem.

```
pristupnost typ NazevVlastnosti
{
    pristupnost get;
    pristupnost set;
}
```

Indexery

- Podobný zápis jako vlastnosti.
- Umožňuje vlastní implementaci operátoru [] na dané třídě.
- Nemusíme indexovat pouze pomocí čísel, ale můžeme použít i jiné typy.

```
pristupnost typNavratu this[typIndexu identifikator]
{
    pristupnost get { teloGetu; } // lze pouzit identifikator
    pristupnost set { teloSetu; } // lze pouzit identifikator
}

class MyIntArray
{
    public int this[int index]
    {
        get { vratPrvekNaIndexuIndex; }
        set { nastavPrvekNaIndexuIndex; }
    }
}
```

Metody

- V C# nejsou funkce, protože každá metoda, slot, vlastnost apod. patří nějaké třídě.
- Ekvivalentem funkcí jsou *statické* metody.
- Statické metody lze volat bez vytvoření instance dané třídy.
- Pro jednopříkazové funkce lze použít zkrácenou syntax: `predpis => navratovyVyraz;`

```
pristupnost modifikatory navratovyTyp NazevFunkce(typ1 arg1, typ2
    arg2, ...)
{
    telo;
}

static void Main()
{
    Console.WriteLine("Hello world!");
}
int Calculate(int a, int b)
{
    return a + b;
}
int Calculate2(int a, int b) => a + b;
```

Metody

- Pokud chceme aby metoda vracela více výsledků, můžeme použít např. tzv. `out` parametry.

```
typNavratu Metoda(typ1 arg1, ... , out typOutu identifikator)
```

- Obvykle se používá pro metody které mohou selhat, ale je nežádoucí aby vyhazovaly výjimky nebo vraceli nesmyslné hodnoty.
- Metoda musí `out` parametru vždy přiřadit hodnotu.
- `out` parametr nemusí být inicializovaný.
- Pokud potřebuje metoda manipulovat s proměnnou, která není v jejím rozsahu, musí být předána jako reference pomocí `ref`.

```
typNavratu Metoda(ref typ argRef, ...)
{
    argRef = ...;
}
```

Parametry metod

- Parametrům můžeme přiřadit výchozí hodnotu pomocí =
- Parametry s výchozí hodnotou se musí vyskytovat až za standardními parametry a uživatel je může vynechat.

```
typNavratu Metoda(typ1 arg1, ... , typN argN = vychoziHodnota)
```

- Můžeme psát metody přijmající libovolný počet parametrů pomocí `params` a polí.

```
typNavratu Metoda(typ1 arg1, ... , params typN[] identifikator)
```

- Při volání metod můžeme změnit pořádí parametrů pomocí jejich identifikátorů.
- Nelze libovolně kombinovat se standardním zadáváním parametrů.

```
Metoda(argN: hodnotaN, arg3: hodnota3, ...);
```

Přetěžování metod

- Můžeme mít více stejnojmených metod v jedné třídě, lišící se v seznamu parametrů / návratovém typu = přetěžování metod.

```
public static class Console {
    public void WriteLine();
    public void WriteLine(string value);
    public void WriteLine(bool value);
    ...
}
```

- Je potřebovat dodržovat nějaké zásady, jako pořadí parametrů, jejich pojmenování apod.
- Existuje několik omezení na které časem narazíme.
- Která verze přetížitelné metody se bude volat určuje již kompilátor!

Interface

- `Interface` definuje **veřejné** rozhraní, které třída implementující toto rozhraní musí podporovat.
- Rozhraní definuje pouze předpisy veřejných členů a nemůže obsahovat jejich definice ani sloty.

```
pristupnost modifikatory interface IMyInterface : Interface1,
    Interface2, ...
{
    // zde nesmi byt zadna implementace, pouze predpisy
    modifikatory typ NazevVlastnosti { get; set; }
    modifikatory navratovyTyp MyMethod();
}
```

- Často používané interfacy:
  - `IEnumerable` - pro výčtové kolekce, díky němu můžeme používat `foreach`;
  - `ICollection` - pro obecné kolekce;
  - `IDisposable` - rozhraní pro typy, které pracují s (vnějšími) zdroji, které se musejí explicitně uvolňovat, umožňuje používat konstrukt `using`;

Cvičení 1

Vytvořte třídu `Person`, která bude reprezentovat člověka.

- Třída bude mít veřejné vlastnosti jako jméno, datum narození (typu DateTime), věk, pohlaví apod.
- Napište metodu `bool IsOlderThan(Person)`, která zjistí zda je daná osoba starší než jiná.
- Napište metodu / vlastnost `string InfoString()`, která vrátí přehledně formátovaný záznam o dané osobě.

Cvičení 2

Vytvořte třídu `ListInt`, která bude realizovat flexibilnější pole čísel typu int.

- Třída bude mít veřejnou vlastnost `Count` určující počet prvků v kolekci.
- Implementujte metody na přidání prvku na konec a na daný index: `Add`, `Insert`.
- Obdobně pro odebírání: `Remove`, `RemoveAt`.
- Implementujte číselný indexer: `[int index]`.
- Implementujte metody pro vyhledávání prvku zleva a zprava: `IndexOf`, `LastIndexOf`.
- Implementujte metodu pro setřízení kolekce: `Sort`.
- Zajistěte, aby na bylo možné použít `foreach` na tuto kolekci.