Základy programování 3: C#

Martin Kauer

Palacký University
Olomouc
Czech Republic

31. října 2018

Výjimky

- V C# existuje vestavěná podpora výjimek.
- Máme k dispozici klasickou konstrukci:

```
try
{
    // prikazy tela
}
catch(typVyjimky1 identifikator) when (podminka1)
{
    // kod osetrujici vyjimku typu typVyjimky1
    // typ vyjimky musi dedit po System.Exception
    // samotna vyjimka je ulozena v promenne identifikator
}
catch(typVyjimky2 identifikator) when (podminka2)
{ }
finally
{
    // kod zde se provede vzdy jako posledni vec pri vyhodnocovani
       tento konstrukce
}
```

Konstrukce `try-catch-finally`

- Povinná je část `try` a poté musí následovat alespoň jedna část `catch` nebo `finally`.
- Částí `catch` může být libovolný počet, `finally` maximálně jedna.
- `catch` lze použít i bez parametrů, ale není to doporučené. Proč?
- Pokud se při vykonávání příkazů v těle `try` vyvolá výjimka, prohledají se postupně shora větve `catch` a vyhodnotí se ta, kde aktuální výjimka je instance daného typu a popř. je splněna podmínka za `when`. Pokud taková větev neexistuje, výjimka se nezpracuje a program zastaví s touto výjimkou.
- Nakonec se vždy vyhodnotí nepovinný blok `finally`.

Konstrukce try-catch-finally

```
object o = "hello";
try
{
    int a = (int)o;
}
catch(InvalidCastException ex)
{
    // osetreni vyjimky
}

object o = null;
try
{
    Console.WriteLine(o.ToString());
}
catch(NullReferenceException ex)
{
    // osetreni vyjimky
}
```

Vyvolání výjimky

- Máme k dispozici klíčové slovo `throw`, vyvolávající výjimku.

```
throw vyjimka;
```

- Výjimka musí být instancí `System.Exception`.
- Můžeme si vytvářet vlastní typy výjimek.
- Základní typy výjimek jsou v .NETu k dispozici: `NullReferenceException`, `ArithmeticException`, `InvalidCastException`, `ArgumentNullException`, ...

Konstrukce try-catch-finally

```
static void BeCareful(object arg1, object arg2)
{
    if (arg1 == null)
        throw new ArgumentNullException(nameof(arg1), $"{nameof(arg1)}
            can not be null.");
    if (arg2 == null)
        throw new ArgumentNullException(nameof(arg2), $"{nameof(arg2)}
            can't be a null reference.");
}
static void Main(string[] args)
{
    try
    {
        BeCareful(new object(), null);
    }
    catch(ArgumentNullException ex) when (ex.ParamName == "arg1")
    { }
    catch (ArgumentNullException ex) when (ex.ParamName == "arg2")
    { }
}
```

First chance výjimky

- Když kód vyvolá výjimku, prvně je označená jako “first chance” - dává debuggeru možnost ji prozkoumat jako první.
- Debugger se může rozhodnout co dělat dále - např. pozastavit program.
- Aplikace samotná se může přihlásit k odběru first chance výjimek.
- First chance výjimka nemusí znamenat chybu v kódu, pokud se kód správně stará o zpracování této výjimky.
- Výhodné pro logování a odlaďování - můžeme si nechat program pozastavit vždy když dojde k výjimce a to ať je ošetřená nebo ne.

Výchozí hodnoty typů

- Máme k dispozici klíčové slovo `default(T)`, kde `T` je nějaký typ ať už hodnotový nebo referenční.
- `default(T)` produkuje výchozí hodnotu typu `T`.
  - Pokud je `T` referenční typ, vrací `null`.
  - Pokud je `T` číselný typ, vrací `0`.
  - Pokud je `T` `bool`, vrací `false`.
  - Pokud je `T` `char`, vrací `'\0'`.
  - Pokud je `T` struktura, vrací její instanci, kde jsou všechny sloty inicializované na své výchozí hodnoty.
  - ...
- Kde bude `default(T)` zejména užitečný?

Struktury

- Často se vidí "Struktury jsou odlehčené objekty" a to proto, že často mohou přinést nárust výkonu pro jednoduché typy.
- Jedná se o hodnotové typy.
- Definují se obdobně jako třídy, ale použije se klíčové slovo `struct` namísto `class`.
- Oproti třídám máme několik rozdílů/omezení.

Rozdíly u struktur oproti třídám

- Nelze definovat vlastní bezparametrový konstruktor. Pokud vytvoříme strukturu pomocí implicitního bezparametrového konstruktoru, všechny sloty se nastaví na výchozí hodnotu (určené jako `default`).
- Pro struktury nelze specifikovat dědičnost - struktura nemůže dědit po jiné struktuře ani třídě.
- Všechny struktury ale implicitně dědí ze `System.ValueType`, což je abstraktní třída dědící po `System.Object`.
- Ve vlástním kódu nelze vytvořit třídu, která by dědila po `System.ValueType`, ale tento typ je pro nás stále užitečný. Proč?
- Metoda `Equals` pro struktury implicitně realizuje hodnotové porovnání, tzn. dvě struktury se rovnají, pokud se rovnají na všech slotech a vlastnostech.
- Struktury (a obecně hodnotové typy) **mohou** být uloženy na zásobníku. Toto je ale téma na samostatnou prezentaci.

Struktury

```
struct Point
{
    public int X { get; set; }
    public int Y { get; set; }

    public Point(int x, int y)
    {
        X = x;
        Y = y;
    }

    public Point Move(Point p) =>
        new Point(X + p.X, Y + p.Y);
}
```

Struktury

```
var p1 = new Point(1, 2);
var p2 = p1;
p2.X = 4;
p1 = p1.Move(p2);
// jake hodnoty budou mit souradnice p1 a p2?
// jak by se vysledek zmenil, kdyby Point byla trida
```

Přetížení operátorů

- C# dovoluje přetížit většinu operátorů.
- Je doporučeno přetěžovat operátory jen u typů, které se blíží primitivním vestavěným typům.
- Je vhodné přetížit operátory u číselných typů. Např. kdybychom implementovali komplexní čísla apod.
- "DO NOT be cute when defining operator overloads." © Microsoft Guidelines 2018.

```
public static navratovyTyp operator oznaceniOperatoru(typ1 arg1, typ2
    arg2)
{
    // implementace operatoru
}
```

Přetížení operátorů

```
public static Point operator +(Point p1, Point p2) =>
    new Point(p1.X + p2.X, p1.Y + p2.Y);

public static Point operator -(Point p1, Point p2) =>
    new Point(p1.X - p2.X, p1.Y - p2.Y);

var p1 = new Point(1, 2);
var p2 = new Point(4, 2);
var p3 = p1 + p2;
var p4 = p1 - p2;
```

Rozšiřující metody

- Dovolují "přidat" metody k již existujícímu typu bez nutnosti vytváření odvozeného typu nebo změny originálního typu.
- Jedná se o zvláštní typ statických metod, které ale voláme jako klasické metody.
- Nelze pomocí nich přepsat metody daného typu. Tzn. rozšiřující metoda se stejným názvem a předpisem jako klasická metoda třídy (popř. rozhraní) nebude nikdy zavolána.
- Lze je definovat pouze v ne-generické statické třídě.
- Použité hojně v `LINQ`, např. rozšíření pro `IEnumerable<T>`: `Average`, `Distinct`, `Where`, `Intersect`, ...

```
public static navratovyTyp Nazev(this rozsirovanyTyp instance)
{
    // implementace rozsirujici metody
    // aktualni instanci mame ulozenou v promenne instance
}
```

Rozšiřující metody

```
static class IEnumerableExtensions
{
    public static IEnumerable<int> OnlyEvenNumbers(this IEnumerable<
        int> nums)
    {
        foreach (var num in nums)
            if (num % 2 == 0)
                yield return num;
    }
}

List<int> nums = new List<int>() { 0, 1, 2, 3, 4, 5 };
foreach (var num in nums.OnlyEvenNumbers())
    Console.WriteLine(num);
```

Cvičení 1

- Minule jsme mluvili o lambda výrazech a anonymních metodách. Řekli jsme si, že při vytvoření anonymní metody dojde k vytvoření uzávěru, tzn. zachycení prostředí vzniku metody. Rozmyslete si co se vypíší následující kódy a poté kód vyzkoušejte. Souhlasí vaše představa s realitou?

```
// code 1
List<Action> actions = new List<Action>();
for (int i = 0; i < 5 ; i++)
    actions.Add(() => { Console.WriteLine(i); });
foreach (var action in actions)
    action();

// code 2
List<Action> actions = new List<Action>();
int[] nums = new[] {0, 1, 2, 3, 4};
foreach (var num in nums)
    actions.Add(() => { Console.WriteLine(num); });
foreach (var action in actions)
    action();
```

Cvičení 2

- Implementujte strukturu `Point3D` a `Vector3D`, dodělejte přetížení vhodných operátorů a vše náležitě otestujte, popř. ošetřete výjimky apod.
- Implementujte typ pro komplexní čísla, implementujte vhodná přetížení operátorů.
- Projděte si třídu `System.Exception` a její veřejné členy.
- V naší generické třídě `MyList<T>` dodělejte výjimky, tzn. zachycení výjimek a hlavně vyvolávání vlastních výjimek.