

6 - Pokročilé konstrukce C#

IW5 - Programování v .NET a C#

Obsah přednášky

- Výjimky
- Delegáti
- Události
- Lambda výrazy

Výjimky a jejich zpracování

- Slouží k ošetření výjimečných (chybových) stavů programu kontrolovaných blokem `try` { }
- Eskalují se, dokud nejsou zpracovány `catch` { } blokem s odpovídajícím typovým parametrem
- Kontrolovaný blok může následovat blok `finally`, který se provede vždy

```
try
{
    // Controlled block
}
catch (ExceptionA ex)
{ /* Exception handling of type ExceptionA */ }
catch (ExceptionB ex)
{ /* Exception handling of type ExceptionB */ }
finally
{
    // Cleanup block
}
```

Výjimky a jejich zpracování – catch

- Reaguje na daný typ výjimky.
- Typový parametr - pouze třída `Exception` nebo potomci
- Instanční proměnná - informace o chybě (`Message`, `InnerException`, `StackTrace`)
- 0-N catch bloků - nutné specifikovat od nejvíce specifických typů výjimek
- Jméno proměnné nebo celý typových paremetr je možné vynechat
- Zpracovávané výjimky je možné explicitně eskalovat pomocí `throw`;

```
try
{ ... }
catch (DivideByZeroException)
{
    Console.WriteLine("Divison by zero is not allowed!");
}
catch (Exception excpInstance)
{
    Console.WriteLine(„Error occured" + excpInstance);
    throw; //Escalates the exception. Upper callstack levels can catch the exception
}
```

Výjimky a jejich zpracování – catch when

- Filtrování výjimek
 - Novinka v C# 6
 - `when`
 - výraz může mít i side-effect (logování)

```
try
{ ... }
catch (WebException ex) when (ex.Status == WebExceptionStatus.Timeout)
{
    ...
}
catch (Exception ex) when (DateTime.Now.DayOfWeek == DayOfWeek.Saturday
    || DateTime.Now.DayOfWeek == DayOfWeek.Sunday )
{
    ...
}
```

Výjimky a jejich zpracování – finally


- Slouží k ošetření otevřených zdrojů (soubory, databáze, síťová spojení, ...)
- Prove se (téměř) vždy ...
 - Po dokončení `catch` bloku
 - Ve chvíli opuštění `try` - bloku (konec bloku, return, break, continue, goto)
- Je možné ho uvést i bez bloku catch (try-finally)

```
StreamReader file = null;
try
{
    file = new StreamReader(File.OpenRead(fileName));
    int number = Int32.Parse(file.ReadLine());
    Console.WriteLine("Number is: " + number);
}
finally
{
    if (file != null)
    {
        file.Dispose();
    }
}
```

Výjimky a jejich zpracování – throw

- `throw`;
Možné pouze z `catch` bloku, eskaluje zachycenou/zpracovávanou výjimku.
- `throw new Exception("Message ...", innerException);`
- `ArgumentException`, `NullPointerException`,
`InvalidOperationException`, `FormatException`,
`FileNotFoundException`, `KeyNotFoundException`, ...
- `StackOverflowException`, `OutOfMemoryException`
Většinou je již není možné jednoduše zpracovat

Ošetření chyb – tips&tricks

- Nechytejte pokemony  - ošetřete specifické typy `Exception`.
- Generické zachytávání:
 - Tam, kde je možné zotavení i z neznámé chyby
 - Pro logování
- Ošetření vstupů veřejných metod
`ArgumentException`, `ArgumentNullException`, `ArgumentOutOfRangeException`
- Výjimky neslouží k běžnému řízení běhu programu!
- Chybové stavy by měly generovat specifickou `Exception`
 (vlastní zpráva, popř. Vlastní typ `Exception`)
- Nezpracované výjimky je možné odchytit centrálně
`AppDomain.CurrentDomain.UnhandledException +=`
`CurrentDomainUnhandledException;`

Delegáti

- Delegát = objekt spravující odkaz na metodu
- Přiřazení metody do proměnné typu delegát vytvoří instanci delegáta
- Tu je možné zavolat stejně, jako metodu

- Definice proměnné delegáta:

```
delegate int Transformer(int x);
```

- Kompatibilní metoda:

```
private static int Square(int x) {return x * x; }
```

- Vytvoření instance delegáta:

```
Transformer t = new Transformer(Square);  
Transformer t = Square; // equivalent
```

- Zavolání metody pomocí delegáta

```
int result = t(3);
```

Delegáti

- Mohou ukazovat na více metod:

```
HelloWorld helloWorld = Hello; helloWorld += World;
```

- Lze je otestovat, zda na někoho ukazují:

```
if (delegateVar == null) { ... }
```

- Mohou obsahovat typové parametry:

```
delegate T Transformer<T>(T arg);
```

- Generické typy delegátů (0 až 16 parametrů)

```
void Action<in T1,...in T16>(T1 arg1,...T16 arg16);  
TResult Func<in T1,...in T15,out TResult>(T1 arg1,...T16 arg16);
```

- Jsou typově kontravariantní – akceptují i metody, které mají argumenty se specifickějšími typy, než definuje delegát.

Delegáti vs. Rozhraní

- Kdy je vhodnější použít delegáty
 - Návrhový vzor využívající události
 - Zapouzdření (jedné) statické metody
 - Volající nemá mít přístup k jiným vlastnostem, metodám nebo rozhraním objektu implementujícím metodu
 - Třída potřebuje více než jednu implementaci metody
- Kdy je vhodnější použít rozhraní
 - Potřeba volat více souvisejících metod
 - Třída potřebuje jedinou implementaci metody
 - Potřeba přetypování rozhraní na jiné rozhraní / třídu
 - Metoda je svázána s typem nebo instancí třídy

Události – implementace

- Implementuje broadcast/subscribe model
- Vnitřně využívá delegáty, ale nepublikuje všechny funkce.

```
delegate void PriceChangedHandler( decimal oldPrice,  
                                   decimal newPrice);
```
- Definice eventu

```
public event PriceChangedHandler PriceChanged;
```
- Uvnitř třídy je možné pracovat s delegátem PriceChanged
- Vně třídy, je možné pouze provést přihlášení/odhlášení

```
wallStreet.PriceChanged += WallStreetPriceChanged
```

Události – vzor

- Předdefinovaný delegát `EventHandler` – vhodný pro bezparametrické událost

```
public delegate void EventHandler(object sender, EventArgs e);
```

- Existuje i generická verze, která usnadňuje implementaci událostí s vlastním typem argumentu

```
public delegate void EventHandler<TEventArgs>(object sender, TEventArgs e);
```

- Pro vyvolání události se implementuje metoda

```
protected virtual void OnPropertyChanged(PriceChangedEventArgs args)
{
    var handler = PriceChanged;
    if (handler != null)
    {
        handler(this, args);
    }
}
```

Události – INotifyPropertyChanged

- Generická notifikace změn na datovém modelu
- Podmínka pro použití DataBinding

```
public event PropertyChangedEventHandler PropertyChanged;

protected virtual void OnPropertyChanged([CallerMemberName]
                                         string propertyName = null)
{ ... }

public string FirstName
{
    get { return _firstName; }
    set
    {
        _firstName = value;
        OnPropertyChanged("FirstName"); //Property name can be omitted
    }
}
```

Lambda výrazy

- Lambda výraz – nepojmenovaná metoda

- Zápis:

(**<parametry>**) => **<výraz>** nebo { **<blok příkazů>** }

- Příklad:

```
Func<int, int> squareRoot = x => x * x;
```

```
Func<int, int> squareRoot = x => { return x * x; };
```

- Typy parametrů jsou většinou odvozené. Pokud to v dané situaci není možné je možné je specifikovat

```
Func<int, int, int> sum = (int a, int b) => a + b;
```

Lambda výrazy – mapované proměnné

- Lambda výrazy mohou mapovat vnější proměnné (lokální proměnné i atributy třídy):

```
int factor = 2;  
Func<int, int> multiplier = n => n * factor;
```

- Jejich hodnota je evaluována až při exekuci výrazu – pozor na smyčky!
- Použití lokální proměnné ji udrží „naživu“ po dobu platnosti lambda výrazu
Pozor na memory-leak v případě reference jiných objektů.

Reference

- <http://www.amazon.com/6-0-Nutshell-The-Definitive-Reference/dp/1491927062>