

(!) Warning

Informace níže jsou pouze orientační!

- ▶ Webové aplikace
- ▶ Mobilní aplikace
- ▶ Počítačové aplikace
- ▶ Databázový vývoj
- ▶ Herní vývoj
- ▶ CI & CD
- ▶ Testování

Vývojové metodiky

Techniky vývoje softwaru jsou postupy, které určují, jakým způsobem se vyvíjí software.

- ▶ [Agilní metodika \(Scrum\)](#)
- ▶ [Vodopádová metodika](#)
- ▶ [Kanban](#)

Rychlé prototypování

Proces pro vytvoření funkčního modelu projektu co nejrychleji, aby bylo možné testovat a iterovat nápady.

Note

V kontextu Unity to znamená vytvoření základní verze hry nebo aplikace, která zahrnuje pouze klíčové mechaniky a funkce.

Rychlá iterace

- Rychle testovat a provádět nápady.
- Pokud vytvoříte prototyp, zkuste ho co nejdříve otestovat a získat zpětnou vazbu.
- Poté můžete na základě této zpětné vazby upravit a vylepšit svůj prototyp.

Important

Cílem je vytvořit funkční model vašeho projektu, ne dokonalý produkt. Nebojte se udělat kompromisy v kvalitě, pokud to znamená, že můžete rychleji testovat a iterovat své nápady.

Postup

1. Definice konceptu

Než se začne s prototypováním, měli byste mít jasnou představu o tom, co chcete vytvořit.

Tip

To může zahrnovat definování klíčových mechanik, funkcí a cílů vašeho projektu.

2. Vytvoření základní scény v Unity

(i) Tip

Tato scéna bude sloužit jako základ pro váš prototyp.

3. Přidání základních objektů

Přidejte do scény základní objekty, jako jsou krychle, koule a válce, které můžete použít k reprezentaci různých prvků ve vaší hře.

4. Přidání mechanik a funkcí

Použijte skriptování a vestavěné nástroje Unity k přidání mechanik a funkcí do vašeho prototypu.

5. Testování a iterace

Jakmile máte základní prototyp, začněte ho testovat.

(i) Note

Získejte zpětnou vazbu od ostatních a na základě této zpětné vazby upravte a vylepšujte svůj prototyp.

6. Opakování procesu

Po provedení změn na svém prototypu ho znova otestujte a pokračujte v tomto cyklu, dokud nejste spokojeni s výsledkem.

(i) Tip

Rychlé prototypování je iterativní proces.

Pojmenování BEM

BEM = "Block Element Modifier"

Metodika pro pojmenování tříd v [HTML](#) a [CSS](#).

Note

Pomáhá udržet váš kód organizovaný a snadno pochopitelný, a to i pro ostatní vývojáře, kteří se na váš kód podívají.

Příklad:

```
<div class="block"> <!-- Block -->
  <div class="block__element"> <!-- Element -->
    </div>
  <div class="block__element--modifier"> <!-- Element with modifier -->
    </div>
</div>
```

```
.block { ... }
.block__element { ... }
.block__element--modifier { ... }
```

- ▶ Block
- ▶ Element
- ▶ Modifier
- ▶ Syntax BEM
- ▶ Použití v kódu

Nuget Packages

- ▶ Správa balíčků
- ▶ Globální složka balíčků

.NET CLI (Command Line Interface)

Important

Je zapotřebí mít nainstalovaný [.NET SDK \(Software Development Kit\)](#) a [.NET Runtime \(Framework\)](#)

- ▶ Umístění balíčků
- ▶ Seznam nainstalovaných balíčků
- ▶ Záloha globálních nástrojů
- ▶ Obnova globálních nástrojů

Příkazy

- ▶ Instalace
- ▶ Aktualizace
- ▶ Odinstalace

Modifikátory přístupu

Určuje přístup k danému prvku.

Caller's location	public	protected internal	protected	internal	private protected	private	file
Within the file	✓	✓	✓	✓	✓	✓	✓
Within the class	✓	✓	✓	✓	✓	✓	✗
Derived class (same assembly)	✓	✓	✓	✓	✓	✗	✗
Non-derived class (same assembly)	✓	✓	✗	✓	✗	✗	✗
Derived class (different assembly)	✓	✓	✓	✗	✗	✗	✗
Non-derived class (different assembly)	✓	✗	✗	✗	✗	✗	✗

Více podrobností [zde](#).

Složka 'runtimes' a multiplatformní nasazení

Slouží k ukládání **platformově specifických knihoven a binárních souborů**, které jsou nezbytné pro správné spuštění aplikace na různých operačních systémech a architekturách.

Používá se v aplikacích:

- WPF aplikace
- Konzolové aplikace
- ASP.NET Core aplikace
- WinForms aplikace
- .NET knihovny a služby, které se nasazují na různé platformy (Windows, Linux, macOS, atd.)

(i) Tip

V unity se používá složka '**Plugins**' k umístění knihoven (DLL), které jsou platformově specifické

Tato složka může obsahovat nativní kód pro různé platformy (Windows, Android, iOS, macOS, atd.)

Tato složka '**runtime**' se automaticky generuje při publikaci aplikace a hraje zásadní roli zejména při použití dvou typů nasazení.

Každý z těchto typů nasazení určuje, jakým způsobem aplikace zajišťuje dostupnost .NET runtime a dalších závislostí:

self-contained deployment

Aplikace je distribuována společně s **kompletním .NET runtime**

To znamená, že aplikace si nese vlastní runtime a může běžet nezávisle na tom, zda má uživatel na svém systému nainstalovaný správný .NET runtime.

(i) Tip

Toto nasazení je vhodné, pokud chcete zajistit, že aplikace poběží na jakémkoliv počítači, bez ohledu na její aktuální stav.

Výsledkem je větší velikost aplikace, protože obsahuje kompletní runtime pro cílové platformy, které jsou zahrnuty ve složce '**runtimes**'.

V praxi složka '**runtimes**' obsahuje potřebné knihovny a binární soubory pro různé platformy, jako jsou Windows, Linux, macOS, nebo různé architektury (x64, x86, ARM).

Díky tomu může aplikace běžet **out-of-the-box** bez nutnosti další instalace runtime na cílovém systému.

Použití

Nastavit v souboru projektu (.csproj):

```
<PropertyGroup>
    <SelfContained>true</SelfContained>
    <RuntimeIdentifier>win-x64</RuntimeIdentifier> <!-- Nebo jiný RID podle
platformy -->
</PropertyGroup>
```

nebo s více Runtime Identifiers (RID)

```
<PropertyGroup>
  <SelfContained>true</SelfContained>
  <RuntimeIdentifiers>win-x64;linux-x64;osx-x64</RuntimeIdentifiers>
</PropertyGroup>
```

Note

Runtime Identifiers (RID) Runtime Identifiers (RID) jsou klíčovou součástí procesu nasazení, protože určují, pro jaké platformy a architektury bude aplikace komplikována.

Můžete specifikovat různé RID podle cílové platformy:

- win-x64 (Windows 64-bit)
- linux-x64 (Linux 64-bit)
- osx-x64 (macOS 64-bit)
- win-arm (Windows na ARM procesorech)

framework-dependent deployment (runtime-specific deployment)

Aplikace **závisí na přítomnosti .NET runtime** na cílovém systému.

Aplikace neobsahuje runtime v sobě, což zmenšuje její velikost, ale předpokládá, že uživatel má již správnou verzi .NET runtime nainstalovanou.

Important

Pokud runtime není přítomen, aplikace nefunguje, dokud uživatel runtime nedoinstaluje.

V tomto případě složka "**runtimes**" může obsahovat pouze platformově specifické knihovny a závislosti, které nejsou součástí základního .NET runtime, a zajistit kompatibilitu aplikace s různými platformami.

Použití

i Note

Nastavte `SelfContained` na `false`, nebo tuto vlastnost úplně vynechte (výchozí nastavení je totiž `framework-dependent`).

Nastavit v souboru projektu (.csproj):

```
<PropertyGroup>
    <SelfContained>false</SelfContained>
</PropertyGroup>
```

Uvolnění zdrojů

- **Řízené zdroje**

= Objekty, které jsou spravovány garbage collectorem.

Zahrnuje všechny objekty vytvořené pomocí klíčového slova `new`.

Garbage collector automaticky sleduje tyto objekty a uvolňuje jejich paměť, když již nejsou potřebné.

i Tip

Programátoři nemusí explicitně uvolňovat paměť pro tyto objekty.

- **Neřízené zdroje**

= Objekty, které nejsou spravovány garbage collectorem.

Zahrnuje soubory, databázové připojení, síťové zdroje, atd...

i Note

Programátoři musí explicitně uvolnit tyto zdroje, aby zabránili úniku paměti.

Uvolnění zdrojů je důležité pro správné fungování aplikace.

Destruktor

- Automaticky volán, když je objekt zničen garbage collectorem.

- Definuje se pomocí syntaxe `~ClassName()`.
- Uvolňuje neřízené zdroje, které třída drží.
- Destruktory nejsou deterministické.

(i) Tip

Znamená, že nevíme přesně, kdy budou volány.

- **Jsou volány, když garbage collector rozhodne**, že je objekt vhodný ke zničení.

(i) Note

Může to být kdykoliv po tom, co objekt přestane být používán.

Dispose

- Součástí rozhraní `IDisposable` a je určen pro explicitní uvolnění zdrojů.
- Metoda `Dispose` je volána programátorem, když je známo, že objekt již nebude potřebný.

(i) Note

To umožňuje okamžité uvolnění zdrojů a zajišťuje, že nebudou drženy déle, než je nutné.

- Metoda `Dispose` je určena pro uvolnění jak řízených, tak neřízených zdrojů.

Volání funkcí z externích DLL

(i) Note

Platform Invocation Services (PInvoke) se používá pro volání knihoven z nativního kódu.

- Nativní kód = Kód kompilován do strojového kódu pro konkrétní platformu.
- Strojový kód = Kód přímo spustitelný na hardware dané platformy.

Postup:

1. Importování funkce z DLL

Atribut **DllImport** z **System.Runtime.InteropServices** k importování funkce z DLL.

Například:

```
using System.Runtime.InteropServices;

public class MyProgram
{
    [DllImport("User32.dll")]
    public static extern int MessageBox(IntPtr h, string m, string c, int type);
}
```

Note

MessageBox je funkce definovaná v knihovně: **User32.dll**.

Tato funkce je nyní dostupná v rámci tohoto .NET kódu.

Příklad knihovny v c++

Knihovna v c++ by mohla vypadat takto:

```
#include <windows.h>

extern "C" __declspec(dllexport) int MessageBox(HWND h, LPCSTR m, LPCSTR c,
int type)
{
    return MessageBoxA(h, m, c, type);
}
```

Metody a argumenty v c++

- **extern "C"**

Warning

Zajistí, že funkce jsou kompatibilní s C jazykem. (To je důležité pro interoperabilitu mezi C++ a jinými jazyky, jako je C#.)

(i) Tip

Když kompilátor narazí na funkci, změní její název na něco, co jednoznačně identifikuje nejen název funkce, ale také typy jejích parametrů.

To znamená, že název funkce, jak je viděn v DLL, nebude stejný jako název funkce v původním kódu.

Když použijete `extern "C"`, říkáte kompilátoru, aby tuto funkci nezměnil a zachoval její název tak, jak je. To umožňuje jiným jazykům, jako je C#, najít a volat tuto funkci správným názvem.

- **`_declspec(dllexport)`**

Říká kompilátoru C++, že tato funkce nebo proměnná bude exportována z DLL, takže ji může volat jiný kód, který tuto DLL používá.

(!) Warning

Důležité k viditelnosti a dostupnosti pro `PInvoke`

- **`HWND`**

"handle to a window" (rukojeť okna)

- **`LPCTSTR`**

"Long Pointer Constant String"

Item	8-bitů (Ansi)	16-bitů (Wide)	Různé
character	CHAR	WCHAR	TCHAR
string	LPSTR	LPWSTR	LPTSTR
string (const)	LPCSTR	LPCWSTR	LPCTSTR

Odkaz [zde](#)

2. Volání importované funkce

```
public class MyProgram
{
```

```
public static void Main()
{
    MessageBox(IntPtr.Zero, "Hello, World!", "Test MessageBox", 0);
}
```

IntPtr.Zero

Konstanta, která reprezentuje nulový ukazatel.

Je to ekvivalent NULL v C++.

Internal

Pokud se používá Internal jako název DLL v atributu **DllImport**, znamená to, že funkce se hledá přímo v hlavním spustitelném souboru.

(i) Tip

Hledá tedy v samotné aplikaci, pokud je to nativní kód, nebo v jedné z knihoven, na které aplikace odkazuje.

- Příklad použití v C#:

```
public class MyProgram
{
    [DllImport("__Internal")]
    public static extern int MyFunction();

    public static void Main()
    {
        MyFunction();
    }
}
```

- Příklad definice v C++

```
extern "C" __declspec(dllexport) int MyFunction()
{
    // Implementace vaší funkce
    return 0;
}
```

V tomto příkladu `MyFunction` je funkce definovaná v nativním kódu.

Je tedy součástí aplikace nebo jedné z jejích závislostí.

Tipy

- `PInvoke`

= **Platform Invocation Services**, což je technika v `.NET`, která umožňuje volání funkcí, které jsou implementovány v neřízeném kódu.

(i) Tip

To je obvykle používáno pro volání C API funkcí, které jsou definovány v DLL.

Příklad:

```
private static class PInvoke
{
    #if UNITY_IOS || UNITY_TVOS
        private const string DllName = "__Internal";
    #elif UNITY_STANDALONE_OSX
        private const string DllName = "MacOSAppleAuthManager";
    #endif

        public delegate void NativeMessageHandlerCallbackDelegate(uint
requestId, string payload);

[AOT.MonoPInvokeCallback(typeof(NativeMessageHandlerCallbackDelegate))]
        public static void NativeMessageHandlerCallback(uint requestId,
string payload)
        {
            try
            {
                CallbackHandler.ScheduleCallback(requestId, payload);
            }
            catch (Exception exception)
            {
                Console.WriteLine("Received exception while scheduling a
callback for request ID " + requestId);
                Console.WriteLine("Detailed payload:\n" + payload);
                Console.WriteLine("Exception: " + exception);
            }
        }

[System.Runtime.InteropServices.DllImport(DllName)]
        public static extern bool AppleAuth_IsCurrentPlatformSupported();

[System.Runtime.InteropServices.DllImport(DllName)]
        public static extern void
AppleAuth_SetupNativeMessageHandlerCallback(NativeMessageHandlerCallbackDelegate
callback);

[System.Runtime.InteropServices.DllImport(DllName)]
```

```
    public static extern void AppleAuth_GetCredentialState(uint requestId,
string userId);

    [System.Runtime.InteropServices.DllImport(DllName)]
    public static extern void AppleAuth_LoginWithAppleID(uint requestId,
int loginOptions, string nonceCStr, string stateCStr);

    [System.Runtime.InteropServices.DllImport(DllName)]
    public static extern void AppleAuth_QuickLogin(uint requestId, string
nonceCStr, string stateCStr);

    [System.Runtime.InteropServices.DllImport(DllName)]
    public static extern void
AppleAuth_RegisterCredentialsRevokedCallbackID(uint callbackId);

    [System.Runtime.InteropServices.DllImport(DllName)]
    public static extern void AppleAuth_LogMessage(string messageCStr);
}
```

Více info [zde](#).

Interface = rozhraní

- [ICloneable](#)

Kolekce FIFO/LIFO

Určují pořadí, ve kterém jsou prvky přidávány a odebírány.

- ▶ [Queue](#)
- ▶ [PriorityQueue](#)
- ▶ [Stack](#)

Seznamy

Seznamy jsou kolekce prvků, které lze indexovat a efektivně upravovat.

Umožňují přidávání, odstraňování a přístup k prvkům na základě jejich indexu.

- ▶ [List](#)
- ▶ [LinkedList](#)

Slovníky

Slovníky jsou kolekce klíč-hodnota, které umožňují efektivní vyhledávání, přidávání a odstraňování prvků na základě klíče.

Každý klíč v slovníku je jedinečný a je spojen s jednou hodnotou.

- ▶ [Dictionary](#)
- ▶ [SortedDictionary](#)

Kolekce bez duplicit

Neumožňují ukládání duplicitních prvků

- ▶ [HashSet](#)
- ▶ [Hashtable](#)

Kolekce Tuple

Umožňuje ukládání prvků různých typů v jedné kolekci.

Každý prvek v Tuple je přístupný pomocí pevně daného pořadí.

- ▶ [Tuple](#)
- ▶ [ValueTuple](#)

Pozorovatelné kolekce

Upozorňují na změny prvků, což je užitečné pro sledování změn v reálném čase.

- [ObservableCollection](#)

Kolekce pouze pro čtení

Kolekce, které nelze měnit po jejich vytvoření, což zajišťuje jejich neměnnost a bezpečnost

- [ReadOnlyCollection](#)
- [ReadOnlyDictionary](#)

Neměnné kolekce

Nelze měnit po jejich vytvoření, což zajišťuje jejich neměnnost a bezpečnost

- [ImmutableArray](#)
- [ImmutableList](#)
- [ImmutableDictionary](#)
- [Immutable HashSet](#)
- [Immutable SortedSet](#)
- [Immutable Queue](#)
- [ImmutableStack](#)

Paměťové kolekce

Umožňují bezpečný přístup k paměti a manipulaci s ní

- [Memory](#)
- [Span](#)

Slabé reference

Umožňují udržovat odkazy na objekty bez zabránění jejich uvolnění garbage collectorem

- [WeakReference](#)

Kolekce pro více vláken

Jsou bezpečné pro použití ve více vláknech, což zajišťuje synchronizaci a bezpečnost dat

- ▶ [ConcurrentQueue](#)
- ▶ [ConcurrentStack](#)
- ▶ [ConcurrentDictionary](#)
- ▶ [ConcurrentBag](#)
- ▶ [BlockingCollection](#)

Atributy obsahujá **Metadata**

Datové anotace

= System.ComponentModel.Annotations (namespace)

- Nejvíce používané anotace:

- ▶ [Required]
 - ▶ [Range]
 - ▶ [MaxLength]
 - ▶ [MinLength]
 - ▶ [StringLength]
 - ▶ [RegularExpression]
 - ▶ [DataType]
 - ▶ [Display]
- ▶ [Příklad](#)
- ▶ [Vlastní datová anotace](#)

FileHelpers

Important

Nepodporuje:

Záznamy s proměnnou délkou (každý záznam musí mít stejný počet polí)

Změnu formátu za běhu (každý záznam musí mít stejný formát po celou dobu běhu programu)

- Nejvíce používané atributy:

Třída

- ▶ [DelimitedRecord]
- ▶ [FixedLengthRecord]

Pole

- ▶ [FieldTrim]

- ▶ [FieldOptional]
- ▶ [FieldIgnore]
- ▶ [FieldConverter]
- ▶ [FieldOrder]
- ▶ [FieldQuoted]

- ▶ Příklad

Vlastní konvertor

1. Vytvořit třídu a rozšířit ji o třídu `ConverterBase`.
2. Přepsat metody `StringToField` a `FieldToString`.

- Příklad:

- ▶ Definice
- ▶ Použití

- ▶ Základní pojmy
- ▶ Druhy metod
- ▶ Ukazatel na metody
- ▶ Asynchronní a Paralelní metody
- ▶ Task Parallel Library (TPL)
- ▶ Tipy

NUnit

= Testovací framework

- ▶ [Multiple Asserts](#)

WPF (Windows Presentation Foundation)

- Tvorba desktopových aplikací na platformě Windows
- Odděluje logiku aplikace (C#) od vzhledu (XAML)
- Umožňuje datové vazby a stylování
- Podpora vektorové grafiky, animací a multimédií

Prvky

- ▶ [Button](#)
- ▶ [TextBox](#)
- ▶ [CheckBox](#)
- ▶ [ComboBox](#)
- ▶ [RadioButton](#)
- ▶ [Slider](#)
- ▶ [Vlastní ovládací prvek](#)

Styly

Styl se používá k definování vzhledu a chování více prvků najednou.

Definuje se pomocí **XAML**.

- Definování stylu

Styl se definuje v sekci **Resources**:

```
<Window.Resources>
    <Style x:Key="MyButtonStyle" TargetType="Button">
        <Setter Property="Background" Value="Blue"/>
        <Setter Property="Foreground" Value="White"/>
        <Setter Property="FontSize" Value="14"/>
        <Setter Property="Padding" Value="10"/>
    </Style>
</Window.Resources>
```

- **x:** Unikátní název stylu, který použijete později.
 - **TargetType:** Typ prvku, na který se styl vztahuje.
- Použití stylu

Chcete-li styl použít na prvek, využijete atribut **Style**:

```
<Button Style="{StaticResource MyButtonStyle}" Content="Klikni na mě"/>
```

Prefixy

x:

- Vyhrazen pro XAML standardní funkce a typy.
- Používá se pro přístup k základním vlastnostem, jako jsou `x:Class`, `x:Name`, `x:Key`, atd.
- Příklad

```
<Window x:Class="MyNamespace.MainWindow"  
        x:Name="mainWindow"  
        x:Key="myWindowKey">
```

xmlns:

- Používá se k deklaraci namespace.
- Obvykle se používá v kořenovém prvku XAML souboru.
- Příklad deklarace namespace:

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
        xmlns:local="clr-namespace:MyNamespace">
```

local:

- Používán k odkazování na vlastní namespace aplikace.
- Můžete ho použít k přístupu k vlastním ovládacím prvkům, datovým modelům a dalším třídám definovaným ve vaší aplikaci.
- Příklad:

```
<local:MyCustomButton Content="Moje vlastní tlačítko"/>
```

xmlns:sys:

- Pro přístup k základním typům .NET, jako jsou `System.String`, `System.Int32`, atd.
- Příklad:

```
xmlns:sys="clr-namespace:System;assembly=mscorlib"
```

xmlns:controls:

- Pro přístup k ovládacím prvkům z externích knihoven, jako je například **Windows Community Toolkit**.
- Příklad:

```
<controls:MyCustomControl/>
```

xmlns:mc:

- Používá se pro **Markup Compatibility**.
- Umožňuje použití starších XAML formátů a zajišťuje zpětnou kompatibilitu.
- Příklad:

```
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
```

xmlns:d:

- Používá se pro návrhové časové funkce a umožňuje definovat prvky, které se zobrazují pouze během návrhu.
- Příklad:

```
xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
```

Použití prefixů v XAML

```
<Window x:Class="MyNamespace.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:MyNamespace"
        xmlns:controls="clr-
        namespace:MyCustomControls;assembly=MyCustomControlsAssembly"
        xmlns:sys="clr-namespace:System;assembly=mscorlib"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        mc:Ignorable="d"
        Title="Hlavní okno" Height="350" Width="525">

    <Grid>
        <local:MyCustomButton Content="Moje vlastní tlačítko"
```

```
Width="200" Height="50"/>
    <Button Content="Tlačítko" Width="100" Height="30"/>
</Grid>
</Window>
```

Šablony (ControlTemplates)

Šablony umožňují plně přizpůsobit vzhled ovládacího prvku.

Šablona definuje strukturu a vzhled prvku.

- Vytvoření šablony pro tlačítko

```
<Window.Resources>
    <ControlTemplate x:Key="MyButtonTemplate" TargetType="Button">
        <Border Background="{TemplateBinding Background}"
            BorderBrush="Black"
            BorderThickness="2"
            CornerRadius="5">
            <ContentPresenter HorizontalAlignment="Center"
                VerticalAlignment="Center"/>
        </Border>
    </ControlTemplate>
</Window.Resources>
```

- **ControlTemplate**: Určuje, jak bude tlačítko vypadat.
- **TemplateBinding**: Slouží k vázání vlastností stylu na vlastnosti šablony.
- Použití šablony

```
<Button Template="{StaticResource MyButtonTemplate}"
    Background="LightBlue"
    Content="Stylizované tlačítko"/>
```

Responzivní design prvků

Responzivní design znamená, že se aplikace přizpůsobí různým velikostem a rozlišením obrazovky.

Layout Panely

WPF nabízí různé layout panely, které vám pomohou uspořádat ovládací prvky, jako jsou tlačítka nebo textová pole.

Zde jsou nejčastěji používané panely:

- **Grid:**

Rozděluje okno na řádky a sloupce.

Umožňuje flexibilní uspořádání.

Příklad:

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="*" />
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>

    <Button Grid.Row="0" Grid.Column="0" Content="Tlačítko 1" />
    <Button Grid.Row="0" Grid.Column="1" Content="Tlačítko 2" />
    <Button Grid.Row="1" Grid.Column="0" Content="Tlačítko 3" />
    <Button Grid.Row="1" Grid.Column="1" Content="Tlačítko 4" />
</Grid>
```

V tomto příkladu je **Grid** rozdělen na 2 řádky a 2 sloupce, každá buňka obsahuje jedno tlačítko.

- **StackPanel:**

Ukládá ovládací prvky buď vertikálně (jeden pod druhým), nebo horizontálně (vedle sebe).

Příklad:

```
<StackPanel Orientation="Vertical">
    <Button Content="Tlačítko 1" />
    <Button Content="Tlačítko 2" />
    <Button Content="Tlačítko 3" />
</StackPanel>
```

Zde jsou tlačítka naskládána vertikálně (jedno pod druhým).

Pokud chceme prvky uspořádat horizontálně:

```
<StackPanel Orientation="Horizontal">
    <Button Content="Tlačítko 1" />
    <Button Content="Tlačítko 2" />
    <Button Content="Tlačítko 3" />
</StackPanel>
```

- **WrapPanel:**

Pokud je na obrazovce málo místa, ovládací prvky se "obalí" na další řádek/sloupec.

Příklad:

```
<WrapPanel>
    <Button Content="Tlačítko 1" Width="100" Height="50" />
    <Button Content="Tlačítko 2" Width="100" Height="50" />
    <Button Content="Tlačítko 3" Width="100" Height="50" />
    <Button Content="Tlačítko 4" Width="100" Height="50" />
    <Button Content="Tlačítko 5" Width="100" Height="50" />
</WrapPanel>
```

- **DockPanel:**

Umožňuje umístit ovládací prvky na okraje okna (vlevo, vpravo, nahoře, dole) a zbývající prostor zaplní jeden prvek.

Příklad:

```
<DockPanel>
    <Button DockPanel.Dock="Top" Content="Horní tlačítko" />
    <Button DockPanel.Dock="Bottom" Content="Spodní tlačítko" />
    <Button DockPanel.Dock="Left" Content="Levé tlačítko" />
    <Button DockPanel.Dock="Right" Content="Pravé tlačítko" />
    <Button Content="Centrální tlačítko" />
</DockPanel>
```

Dynamické Velikosti

Dynamické velikosti znamenají, že se ovládací prvky automaticky přizpůsobují velikosti okna.

Místo pevně definovaných hodnot můžete použít:

- **Procentuální hodnoty**

Umožňují ovládacím prvkům zabírat procento rodičovského prvku.

Příklad:

```
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="50%"/> <!-- 50% šířky rodičovského prvku -->
        <ColumnDefinition Width="50%"/> <!-- 50% šířky rodičovského prvku -->
    </Grid.ColumnDefinitions>

    <TextBlock Text="Toto zabírá 50% šířky okna" Background="LightCoral"
Grid.Column="0" />
    <TextBlock Text="Toto také zabírá 50% šířky okna" Background="LightBlue"
Grid.Column="1" />
</Grid>
```

- **Hodnoty s ***

V Gridu můžete použít * k rozdělení zbývajícího místa.

Například 2* a 1* znamená, že první sloupec zabere dvakrát více místa než druhý.

Příklad:

```
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="2*"/> <!-- Dva díly prostoru -->
        <ColumnDefinition Width="1*"/> <!-- Jeden díl prostoru -->
    </Grid.ColumnDefinitions>

    <TextBlock Text="Toto zabírá 2/3 šířky okna" Background="LightCoral"
Grid.Column="0"/>
    <TextBlock Text="Toto zabírá 1/3 šířky okna" Background="LightBlue"
Grid.Column="1"/>
</Grid>
```

- **Auto**

Automatická velikost na základě obsahu.

Příklad:

```

<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>    <!-- Automatická výška pro hlavičku --
->
        <RowDefinition Height="*"/>      <!-- Zbytek prostoru pro další prvky --
->
    </Grid.RowDefinitions>

    <TextBlock Text="Hlavička" Grid.Row="0" FontSize="24"
Background="LightGray" />
    <Button Content="Toto tlačítko zabírá zbytek prostoru" Grid.Row="1" />
</Grid>

```

Sledování Změny Velikosti

Můžete sledovat změny velikosti okna a upravit rozložení ovládacích prvků.

Pomocí události `SizeChanged` můžete reagovat na změnu velikosti okna a provést potřebné úpravy.

Příklad:

```

private void Window_SizeChanged(object sender, SizeChangedEventArgs e)
{
    // Získání nové šířky a výšky okna
    double newWidth = e.NewSize.Width;
    double newHeight = e.NewSize.Height;

    // Například upravit velikost tlačítka na základě velikosti okna
    if (newWidth < 600)
    {
        myButton.Width = 100; // Menší šířka tlačítka
    }
    else
    {
        myButton.Width = 200; // Větší šířka tlačítka
    }

    // Můžete také změnit další vlastnosti na základě velikosti okna
}

```

Important

Nezapomeňte přidat událost do XAML:

```
<Window x:Class="ResponzivniDesign.MainWindow"
        ...
        SizeChanged="Window_SizeChanged">
```

ViewBox

Vše, co je uvnitř ViewBoxu, se přizpůsobí velikosti okna. (Automatické škálování obsahu.)

Příklad:

```
<ViewBox>
    <Grid>
        <TextBlock Text="Toto je responzivní text!" FontSize="20"/>
    </Grid>
</ViewBox>
```

Triggery

Triggery umožňují dynamicky měnit vzhled prvku na základě určitých událostí nebo podmínek.

- Použití triggeru

Zde je příklad stylu tlačítka, který mění barvu pozadí, když je kurzor myši nad tlačítkem:

```
<Style TargetType="Button">
    <Setter Property="Background" Value="Gray"/>
    <Setter Property="Foreground" Value="White"/>
    <Style.Triggers>
        <Trigger Property="IsMouseOver" Value="True">
            <Setter Property="Background" Value="Green"/>
        </Trigger>
    </Style.Triggers>
</Style>
```

Data Binding (Vazba Modelu na View)

1. Vytvoření ViewModel

Nejprve vytvoříme ViewModel, který bude obsahovat vlastnost, kterou chceme vázat.

Important

Použijeme `INotifyPropertyChanged`, aby WPF věděl, kdy se vlastnost změnila.

```
using System.ComponentModel;

public class MyViewModel : INotifyPropertyChanged
{
    private string _name;

    public string Name
    {
        get { return _name; }
        set
        {
            _name = value;
            OnPropertyChanged(nameof(Name));
        }
    }

    public event PropertyChangedEventHandler PropertyChanged;

    protected void OnPropertyChanged(string propertyName)
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    }
}
```

Note

`MyViewModel` má vlastnost `Name`, která implementuje `INotifyPropertyChanged`.

To zajišťuje, že pokud se `Name` změní, UI se automaticky aktualizuje.

2. Vytvoření XAML pro UI

Vytvoříme jednoduché uživatelské rozhraní, které umožní uživateli zadat jméno a zobrazit ho.

Použijeme **TextBox** pro zadání a **TextBlock** pro zobrazení.

```
<Window x:Class="WpfApp.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Binding Example" Height="200" Width="300">
    <Grid>
        <TextBox Text="{Binding Name, UpdateSourceTrigger=PropertyChanged}"
Width="200" Margin="10"/>
        <TextBlock Text="{Binding Name}" Margin="10,50,10,10"/>
    </Grid>
</Window>
```

Note

V XAML používáme **{Binding Name}** pro vázání **TextBox** a **TextBlock** na vlastnost **Name** ve ViewModelu.

UpdateSourceTrigger=PropertyChanged znamená, že binding se aktualizuje při každé změně textu v **TextBox**.

(Vlastnost ViewModelu se aktualizuje okamžitě při každé změně textu.)

Příklad:

```
<TextBox Text="{Binding Name, UpdateSourceTrigger=PropertyChanged}"
Width="200" Margin="10"/>
```

Bez explicitního nastavení **UpdateSourceTrigger** se binding aktualizuje při ztrátě fokusu (**LostFocus**).

(Vlastnost ViewModelu se aktualizuje pouze po ztrátě fokusu či při změně výběru v ComboBox.)

Příklad:

```
<TextBox Text="{Binding Name}" Width="200" Margin="10"/>
```

3. Nastavení DataContext

V `MainWindow.xaml.cs` nastavíme `DataContext` na instanci našeho `ViewModelu`.

```
using System.Windows;

namespace WpfApp
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
            DataContext = new MyViewModel(); // Nastavujeme DataContext
        }
    }
}
```

Note

V konstruktoru `MainWindow` nastavujeme `DataContext` na instanci `MyViewModel`.

To umožňuje XAML binding k vlastnostem ViewModelu.

Validace

Implementace

- `INotifyPropertyChanged + IDataErrorInfo`

Obvykle se používá pro vracení celkové chyby objektu.

```
public class MyViewModel : INotifyPropertyChanged, IDataErrorInfo
{
    private string _name;

    // Property s validací
    [Required(ErrorMessage = "Pole je povinné.")]
    public string Name
    {
        get => _name;
```

```

        set
    {
        _name = value;
        OnPropertyChanged(nameof(Name)); // Oznámení o změně
    }
}

// Notifikace o změně
public event PropertyChangedEventHandler PropertyChanged;

protected void OnPropertyChanged(string propertyName)
{
    PropertyChanged?.Invoke(this, new
PropertyChangedEventArgs(propertyName));
}

// Implementace IDataErrorInfo
public string this[string columnName]
{
    get
    {
        var validationResults = new List<ValidationResult>();
        var context = new ValidationContext(this) { MemberName =
columnName };
        Validator.TryValidateProperty(
            this.GetType().GetProperty(columnName).GetValue(this),
            context,
            validationResults
        );

        // Vrátí první chybu, pokud existuje, jinak vrátí null
        return validationResults.FirstOrDefault()?.ErrorMessage;
    }
}

public string Error
{
    get
    {
        return null; // Gets a message that describes any validation errors
for the object.
    }
}
}

```

Note

Použití `this[string columnName]`:

Tato metoda slouží k validaci konkrétní vlastnosti.

Pokud se objeví chyba, vrací odpovídající chybovou zprávu.

XAML pro validaci pomocí `IDataErrorInfo`

```
<TextBox Text="{Binding Name,
    UpdateSourceTrigger=PropertyChanged,
    ValidatesOnDataErrors=True}" />

<TextBox.Style>
    <Style TargetType="TextBox">
        <Style.Triggers>
            <Trigger Property="Validation.HasError" Value="True">
                <Setter Property="BorderBrush" Value="Red" />
                <Setter Property="BorderThickness" Value="2" />
            </Trigger>
        </Style.Triggers>
    </Style>
</TextBox.Style>
```

- `INotifyPropertyChanged + INotifyDataErrorInfo`

Umožňuje validovat více chyb na jedné vlastnosti a spravovat chybové zprávy pro každou vlastnost asynchronně.

```
public class MyViewModel : INotifyPropertyChanged, INotifyDataErrorInfo
{
    private string _name;
    private readonly Dictionary<string, List<string>> _errors = new
Dictionary<string, List<string>>();

    // Property s validací
    [Required(ErrorMessage = "Pole je povinné.")]
    [StringLength(10, ErrorMessage = "Maximálně 10 znaků.")]
    public string Name
    {
        get => _name;
```

```

    set
    {
        _name = value;
        OnPropertyChanged(nameof(Name)); // Oznámení o změně
        ValidateProperty(nameof(Name), value); // Spuštění validace
    }
}

// Notifikace o změně
public event PropertyChangedEventHandler PropertyChanged;

protected void OnPropertyChanged(string propertyName)
{
    PropertyChanged?.Invoke(this, new
PropertyChangedEventArgs(propertyName));
}

// Validace vlastnosti
private void ValidateProperty(string propertyName, object value)
{
    var context = new ValidationContext(this) { MemberName = propertyName };
    var validationResults = new List<ValidationResult>();

    bool isValid = Validator.TryValidateProperty(value,
context, validationResults);

    if (!isValid)
    {
        _errors[propertyName] = validationResults.Select(vr
=> vr.ErrorMessage).ToList();
    }
    else
    {
        _errors.Remove(propertyName);
    }

    OnErrorsChanged(propertyName);
}

// Implementace INotifyDataErrorInfo
public event EventHandler<DataErrorsChangedEventArgs> ErrorsChanged;

// Vrací, zda má objekt nějaké chyby
public bool HasErrors => _errors.Any();

// Získá seznam chyb pro konkrétní vlastnost

```

```

public IEnumerable GetErrors(string propertyName)
{
    if (_errors.ContainsKey(propertyName))
        return _errors[propertyName];
    return null;
}

protected void OnErrorsChanged(string propertyName)
{
    ErrorsChanged?.Invoke(this, new
DataErrorsChangedEventArgs(propertyName));
}

```

XAML pro validaci pomocí **INotifyDataErrorInfo**

```

<TextBox Text="{Binding Name,
UpdateSourceTrigger=PropertyChanged,
ValidatesOnNotifyDataErrors=True}" />

<TextBox.Style>
<Style TargetType="TextBox">
<Style.Triggers>
<Trigger Property="Validation.HasError" Value="True">
<Setter Property="BorderBrush" Value="Red" />
<Setter Property="BorderThickness" Value="2" />
</Trigger>
</Style.Triggers>
</Style>
</TextBox.Style>

```

Animace

WPF podporuje animace, které umožňují měnit vlastnosti prvků v čase.

Zde je příklad, jak animovat změnu barvy pozadí tlačítka, když na něj najedete:

```

<Button Content="Klikni na mě">
<Button.Triggers>
<EventTrigger RoutedEvent="Button.MouseEnter">
<BeginStoryboard>
<Storyboard>
<ColorAnimation Storyboard.TargetProperty=""

```

```
(Button.Background).(SolidColorBrush.Color)"  
        To="Red" Duration="0:0:1"/>  
    </Storyboard>  
  </BeginStoryboard>  
</EventTrigger>  
</Button.Triggers>  
</Button>
```

Vyhledat oddělovač

```
public static StreamReader FindDelimiter(StreamReader reader, out char delimiter,
int? linesToRead = null)
{
    const char semicolon = ';';
    const char comma = ',';

    Dictionary<char, int> delimiters = new Dictionary<char, int>
    {
        { semicolon, default(int) },
        { comma, default(int) },
    };

    string line;
    int linesRead = 0;
    while ((line = reader.ReadLine()) != null && (!linesToRead.HasValue || linesRead
< linesToRead.Value))
    {
        foreach (char c in line)
        {
            switch (c)
            {
                case semicolon:
                    delimiters[semicolon]++;
                    break;

                case comma:
                    delimiters[comma]++;
                    break;
            }
        }

        linesRead++;
    }

    delimiters = delimiters.Where(i => i.Value != 0).ToDictionary(i => i.Key, i =>
i.Value);

    if (delimiters.Count == 0)
        throw new Exception("Nepodařilo se dohledat jakýkoli oddělovač.");

    var highest = delimiters.Aggregate((item1, item2) => item1.Value > item2.Value ?
item1 : item2);
}
```

```

// Kontrola, zda existuje jiný oddělovač, který se vyskytuje alespoň v 70%
případů jako nejčastější oddělovač
const int failPercentage = 70;
if ((from val in delimiters
     where val.Key != highest.Key
     select new decimal(val.Value) / new decimal(highest.Value) * 100).Any(diff
=> diff >= failPercentage))
    throw new Exception("Typ oddělovače se nepodařilo jednoznačně
identifikovat.");

delimiter = highest.Key;

reader.DiscardBufferData();
reader.BaseStream.Seek(0, System.IO.SeekOrigin.Begin);
return reader;
}

```

Escape sekvence

- \uFFEF

= **Byte Order Mark (BOM)**.

BOM nám říká, jak číst data v souboru - od začátku nebo od konce.

! **Warning**

Může způsobit problémy s některými nástroji a knihovnami, které jej neočekávají.

- \u0000

= **null znak**.

Null znak je speciální znak, který se často používá k označení konce řetězce v některých programovacích jazycích a systémech.

! **Warning**

Je obvykle nepotřebný a může způsobit problémy při parsování nebo zpracování dat.

Náhrada znaků

Základní znaky:

- < (levá ostrá závorka)
 <
- > (pravá ostrá závorka)
 >
- & (ampersand)
 &
- ' (apostrof)
 '
- " (uvozovky)
 "

Znaky s diakritikou:

- Á
 Á
- á
 á
- Č
 Č
- č
 č

Note

Stejným způsobem lze následně nahradit i další znaky s diakritikou.

s prefixem `caron` (uvozovky) nebo `acute` (čárka nad písmenem).

Serializace a Deserializace objektu

Serializace

Proces, kdy se objekt převeďe na XML soubor.

```
public string SerializeObject(MyObject myObject)
{
    var serializer = new XmlSerializer(typeof(MyObject));
    using (var stringWriter = new StringWriter())
    {
        serializer.Serialize(stringWriter, myObject);
        return stringWriter.ToString();
    }
}
```

Deserializace

Proces, kdy se XML soubor převeďe zpět na objekt.

```
public MyObject DeserializeObject(string xml)
{
    var serializer = new XmlSerializer(typeof(MyObject));
    using (var stringReader = new StringReader(xml))
    {
        return (MyObject)serializer.Deserialize(stringReader);
    }
}
```

Namespace

Způsob jak zabránit konfliktu jmen v XML souborech.

i Note

Poznáme podle klíčového slova **xmlns** u atributu.

! Warning

Pokud je namespace použit, musí se uvést i při zpracování souboru.

i Note

URL namespace v XML dokumentu nemusí být skutečná existující URL adresa.

Je to pouze jedinečný identifikátor, který se používá k rozlišení názvů elementů a atributů v XML dokumentu.

Tento identifikátor je často ve formě URL pro snadnou identifikaci a jedinečnost, ale nemusí to být nutně platná URL adresa, kterou byste mohli otevřít v prohlížeči.

Například:

```
<?xml version="1.0" encoding="utf-16"?>
<p:Person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:p="test">
    <p:FirstName>John</p:FirstName>
    <p:LastName>Doe</p:LastName>
    <p:Age>30</p:Age>
</p:Person>
```

```
XmlSerializer s = new XmlSerializer(person.GetType());
StringBuilder sb = new StringBuilder();
using (StringWriter writer = new StringWriter(sb))
{
    XmlSerializerNamespaces ns = new XmlSerializerNamespaces();
    ns.Add("p", "test");
    s.Serialize(writer, person, ns);
}
```

XML soubor bez namespace

```
<person>
    <name>John Doe</name>
    <age>30</age>
</person>
```

⚠ Warning

Vlastnosti na objektu **nesmí mít atribut `XmlElement` s namespace**.

```
XmlSerializer s = new XmlSerializer(person.GetType());
XmlSerializerNamespaces ns = new XmlSerializerNamespaces();
ns.Add("", "");
StringBuilder sb = new StringBuilder();
using (StringWriter writer = new StringWriter(sb))
{
    s.Serialize(writer, person, ns);
}
```

ℹ Note

Řešením jak odebrat namespace úplně všude a nebere v potaz atributy `XmlElement` s namespace.

```
XDocument document = XDocument.Parse(dataOutput);
if (document.Root == null) return;
foreach (var element in document.Root.DescendantsAndSelf())
{
    element.Name = element.Name.LocalName;
    element.ReplaceAttributes(element.Attributes()
        .Where(x => !x.IsNamespaceDeclaration)
        .Select(x => new XAttribute(x.Name.LocalName, x.Value)));
}
dataOutput = document.ToString();
```

Pro více info [zde](#).

XML soubor s namespace

Příklad 1:

```
<?xml version="1.0" encoding="utf-16"?>
<Person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <FirstName>John</FirstName>
  <LastName>Doe</LastName>
  <Age>30</Age>
</Person>
```

```
XmlSerializer s = new XmlSerializer(person.GetType());
StringBuilder sb = new StringBuilder();
using (StringWriter writer = new StringWriter(sb))
{
    s.Serialize(writer, person);
}
```

Příklad 2:

```
<?xml version="1.0" encoding="utf-16"?>
<p:Person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:p="http://www.example.com">
  <p:FirstName>John</p:FirstName>
  <p:LastName>Doe</p:LastName>
  <p:Age>30</p:Age>
</p:Person>
```

```
XmlSerializer s = new XmlSerializer(person.GetType());
StringBuilder sb = new StringBuilder();
using (StringWriter writer = new StringWriter(sb))
{
    XmlSerializerNamespaces ns = new XmlSerializerNamespaces();
    ns.Add("p", "http://www.example.com");
    s.Serialize(writer, person, ns);
}
```

Note

V tomto kódu "p" je prefix pro namespace a "http://www.example.com" je URL namespace.

Konvence serializace XML

Umožňuje přizpůsobit jak se třídy a vlastnosti serializují do XML.

Kořenový element

Označí třídu, která se má serializovat jako kořenový element

Pojmenuje kořenový element.

```
[XmlRoot("MyClass")]
public class MyClass
{
    // ...
}
```

Ignorovat vlastnosti

Označuje vlastnosti, která se nemají serializovat.

```
[XmlAttribute]
public string MyProperty { get; set; }
```

Povolit jmenné prostory

Označuje, že se mají jmenné prostory serializovat jako atributy.

```
[XmlNamespaceDeclarations]
public class MyClass
{
    // ...
}
```

Povolit/Zakázat přes metodu

`ShouldSerialize{PropertyName}` je metoda, která se volá při serializaci objektu.

Tato metoda se používá k rozhodnutí, zda se daná vlastnost má serializovat.

- Pokud metoda vrátí `true`
vlastnost `{PropertyName}` se serializuje
- Pokud metoda vrátí `false`
vlastnost `{PropertyName}` se neserializuje

```
public bool ShouldSerializeMyProperty()
{
    // logika rozhodnutí zda serializovat MyProperty
}
```

Povolit/Zakázat přes vlastnost bool

`{PropertyName}Specified` je vlastnost typu `bool`, která se používá k rozhodnutí, zda se má vlastnost serializovat.

Toto je alternativa k `ShouldSerialize{PropertyName}`.

- `{PropertyName}Specified` pokud je `true`
vlastnost `{PropertyName}` se serializuje
- `{PropertyName}Specified` pokud je `false`
vlastnost `{PropertyName}` se neserializuje

```
public bool MyPropertySpecified { get; set; }
```

Pojmenování kolekce a položek v kolekci

Používá se k pojmenování kolekce a položek v kolekci.

```
[XmlArray("MyCollection"), XmlArrayItem("Item")]
public List<string> MyProperty { get; set; }
```

Xml element → xml atribut

Označuje vlastnost, která se serializují jako "XML atribut" místo jako "XML element".

```
[XmlAttribute]  
public string MyProperty { get; set; }
```

Xml element → textový obsah

Označuje vlastnosti, která má serializovat jako obsah XML elementu.

To znamená, že vlastnost se serializuje jako textový obsah XML elementu, nikoli jako samostatný element.

```
[XmlText]  
public string MyProperty { get; set; }
```

Enum → xml element

Označuje výčtový typ, který se má serializovat jako XML element.

```
public enum MyEnum  
{  
    [XmlAttribute("Value1")]  
    Value1,  
    [XmlAttribute("Value2")]  
    Value2  
}
```

Třída → xml element

Označí třídu, která se má serializovat jako XML element.

```
[XmlType("MyClass")]  
public class MyClass  
{  
    // ...  
}
```

Třída ze které se dědí ➔ xml element

Označuje třídy, které se mají serializovat jako potomky rodičovské třídy.

To znamená, že pokud máte třídu `MyBaseClass` a od ní odvozenou třídu `MyDerivedClass`, musíte označit `MyBaseClass` pomocí `XmlAttribute`.

```
[XmlAttribute(typeof(MyDerivedClass))]  
public class MyBaseClass  
{  
    // ...  
}  
  
public class MyDerivedClass : MyBaseClass  
{  
    // ...  
}
```

Libovolný xml element

Označuje vlastnost, která může obsahovat libovolný XML element.

⚠ Warning

Musí být typu `XmlElement[]`, nebo `XmlElement`

```
[XmlElement]  
public XmlElement[] MyProperty { get; set; }
```

Libovolný xml atribut

Označuje vlastnost, která může obsahovat libovolný XML atribut.

⚠ Warning

Musí být typu `XmlAttribute[]`, nebo `XmlAttribute`

```
[XmlAttribute]  
public XmlAttribute[] MyProperty { get; set; }
```

Balíčky

- ▶ [Globální balíčky](#)
- ▶ [Záloha](#)
- ▶ [Obnova](#)

Python

Balíčky

- ▶ Záloha balíčků
- ▶ Instalace balíčků ze zálohy