Object-oriented programming - lab in .NET environment

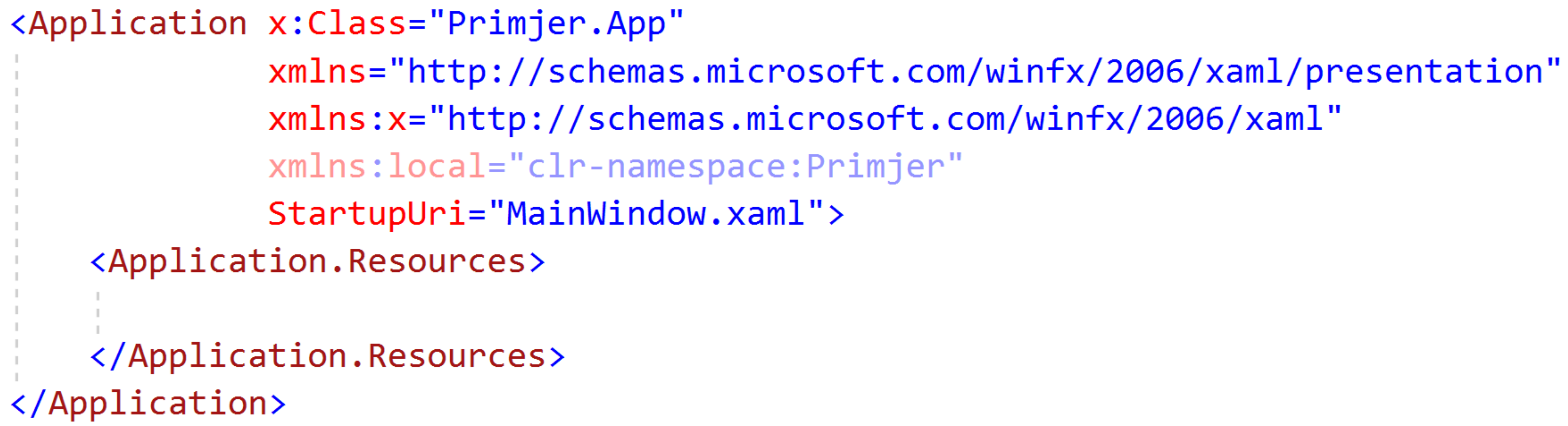
Lecture 05

Windows Presentation Foundation

- **WPF** is Microsoft's primary technology for creating graphical user interface (GUI)
- Main goal:
  - Separate the user interface from the program logic
- Basic features of WPF:
  - Emphasis on the visual component of the application
  - Declarative programming (**XAML** - Extensible Application Markup Language)
    - It is used to describe the user interface in a declarative way
    - The main goal is to facilitate the cooperation of developers with experts from other fields (e.g., UI designers)
  - Resolution independence
  - Hardware acceleration (uses DirectX for plotting)
  - Adaptability

The structure of the initial WPF project

- Dependencies
- AssemblyInfo.cs
- **App.xaml** - ***App.xaml.cs*** – declaratively describes what starts Main + events on the app level



```
<Application x:Class="Primjer.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:local="clr-namespace:Primjer"
             StartupUri="MainWindow.xaml">
    <Application.Resources>

    </Application.Resources>
</Application>
```

- **MainWindow.xaml** - ***MainWindow.xaml.cs*** – user interface and events on the window level

Declarative and procedural

- *Almost* anything that can be done with XAML can be done with the preferred .NET procedural language
- As it was done with the Main configuration, the paradigm continues to build the GUI
- XAML (*object element*):

```
<Button
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    Click="Button_Click"
    Content="Button" />
```

- C#:

```
Button b = new Button();
b.Content = "Button";
b.Click += Button_Click;
```

- Defining attributes (*property attributes* or *event attributes*) is identical to defining a property or event on an object

Namespaces

- The name of the element (eg Button) is the name of the class - but from which namespace?
- The mapping of XAML namespaces to .NET namespaces is built into the WPF assembly (*assembly*)
- The root element of the XAML file must define at least one (default) namespace to define itself and other child elements
- It includes a series of .NET namespaces that contain all the core WPF classes

```
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
```

Namespaces

- XAML files use the x-prefixed namespace

```
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

  - It includes a set of .NET namespaces that contain all the core XAML classes
  - *clr-namespace*:declared within assembly `xmlns:local="clr-namespace:Primjer"`

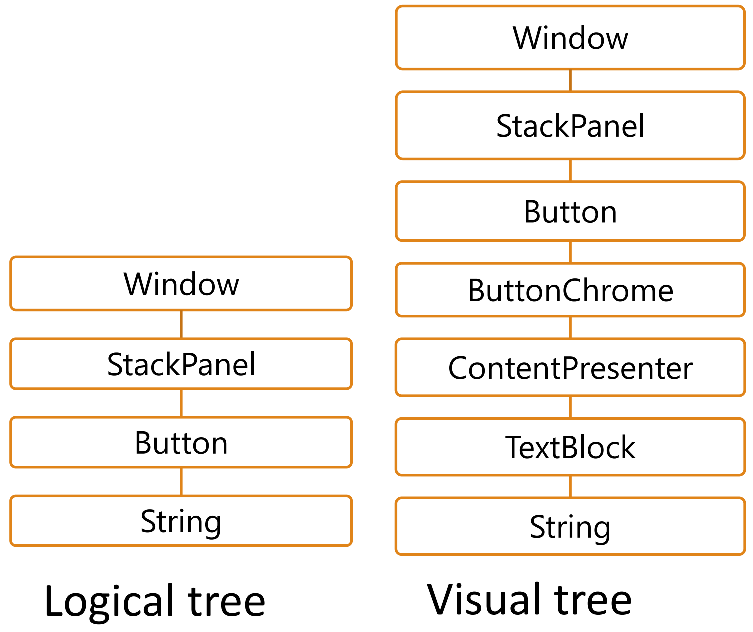
- **Relationship of XAML and *code-behind***
  - XAML document segment: `<Window x:Class="WpfApplication1.MainWindow"> </Window>`
    - We said we wanted an instance of the class `MainWindow` which inherits `Window`
  - Segment of *code-behind* document:

```
namespace WpfApplication1
{
    public partial class MainWindow : Window
```

Logical and visual trees

- **Logical tree**: a set of elements defined in XAML
- **Visual tree**: an expanded version of the logical tree in which each element is expanded into its constituent parts
- For example, if we have XAML:

```
<StackPanel>
    <Button Padding="3"
            Margin="3"
            Content="Gumb"/>
</StackPanel>
```



Elements as properties (*property elements*)

- One of the basic features of WPF is **composition of control**, eg the content of the button does not have to be just text:

```
Rectangle r = new Rectangle();
r.Width = 50;
r.Height = 50;
r.Fill = Brushes.Black;

Button b = new Button();
b.Content = r;
```

- The same can be done in XAML using elements as properties (*property elements*):

```
<Button x:Name="btn">
    <Button.Content>
        <Rectangle Fill="Black" Width="50" Height="50" />
    </Button.Content>
</Button>
```

Elements as properties (*property elements*)

- Property `Content` is set using a XAML element instead of an attribute
- Within `Button.Content` the dot is what makes the difference between an element as an object and an element as a property
- They are always in format `ClassName.PropertyName`

```
<Button x:Name="btn">
    <Button.Content>
        <Rectangle Fill="Black" Width="50" Height="50" />
    </Button.Content>
</Button>
```

Elements as properties (*property elements*)

- They can also be used when defining simple content:

```
<Button Background="Aqua" Content="Klikni me"/>

<!-- ili -->

<Button>
    <Button.Background>
        Aqua
    </Button.Background>
    <Button.Content>
        Klikni me
    </Button.Content>
</Button>
```

Type converters

- From the previous example, it can be concluded that properties whose values are not `string` or `object` are set by using `string` values
- This is possible due to implicit conversion to the appropriate type using *type converter*
- WPF provides converters for most common types (`Brush`, `Color`, `font,` …)
    - These are classes that inherit `TypeConverter` (`BrushConverter`, `ColorConverter`, `FontConverter,` …)
- Without *type converter* we would have to use elements as properties:

```
<Button>
    <Button.Background>
        <SolidColorBrush Color="Aqua" />
    </Button.Background>
</Button>
```

Type converters

- In the previous example we used `Color` *type convetrer*
- If it didn't exist, we would have to define the property as follows:

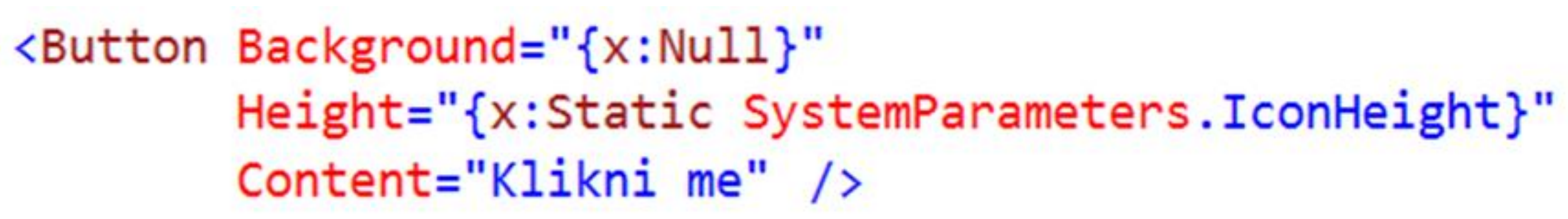
```
<Button.Background>
    <SolidColorBrush>
        <SolidColorBrush.Color>
            <Color A="255" R="255" G="0" B="0" />
        </SolidColorBrush.Color>
    </SolidColorBrush>
</Button.Background>
```

- This method can be used because there is *type converter* which can convert type `string` in `bytes` which is expected at `A`, `R`, `G` and `B` values

Markup extensions

- They represent a XAML technique for obtaining values that are not of a primitive type or of a specific XAML type
  - eg we want to change the background of the control to a gradient color using string values
- When an attribute value is enclosed within curly braces, XAML parser treats that value as a tag extension (*markup extension*)
- Within `System.Windows.Markup` namespace (that's why the prefix **x**) there are several built-in *markup extension* classes (according to convention suffix *extension* can be removed from the name)

Markup extensions

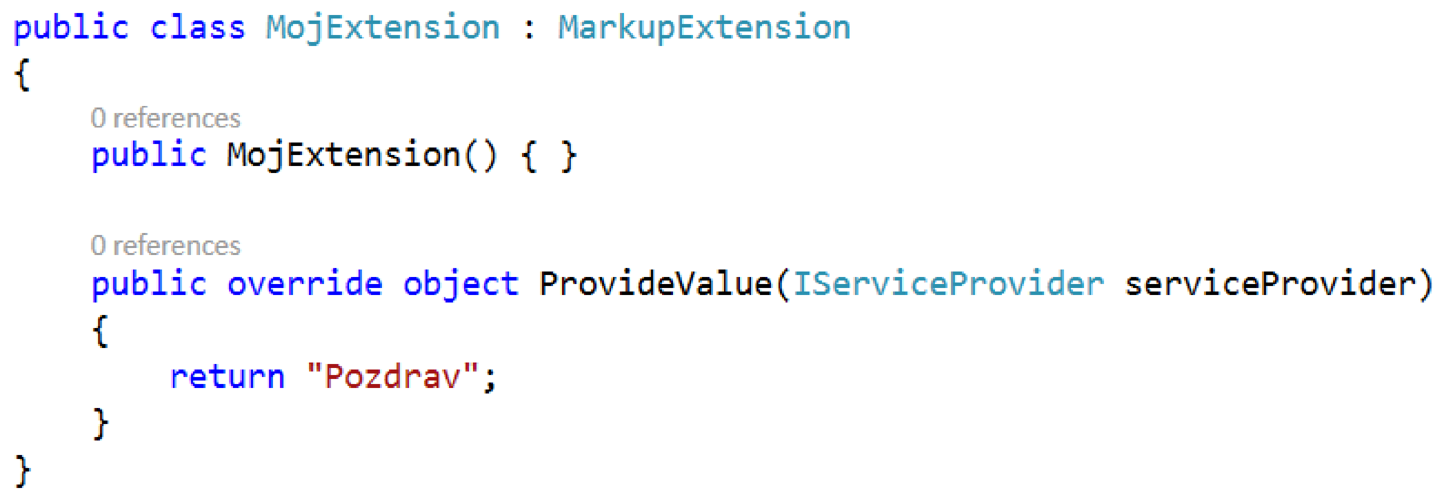


```
<Button Background="{x:Null}"
        Height="{x:Static SystemParameters.IconHeight}"
        Content="Klikni me"  />
```

- `NullExtension` allows `Background` property to has a value `null` which is otherwise not supported by `BrushConverter` class
- `StaticExtension` class allows the use of static property values of objects
- In the example is the height of the `Button` control set to the height value of the system icons, which is obtained from the static value of the property `IconHeight` in class `SystemParameters`

Creating your own tag extension

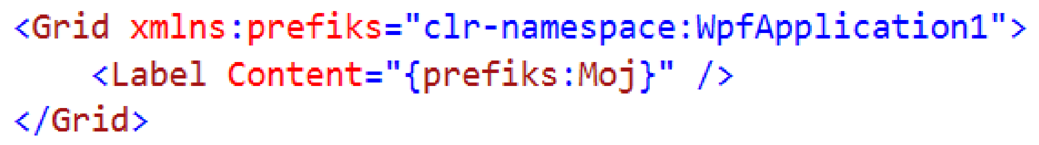
- A class must inherit `MarkupExtension`



```
public class MojExtension : MarkupExtension
{
    0 references
    public MojExtension() { }

    0 references
    public override object ProvideValue(IServiceProvider serviceProvider)
    {
        return "Pozdrav";
    }
}
```

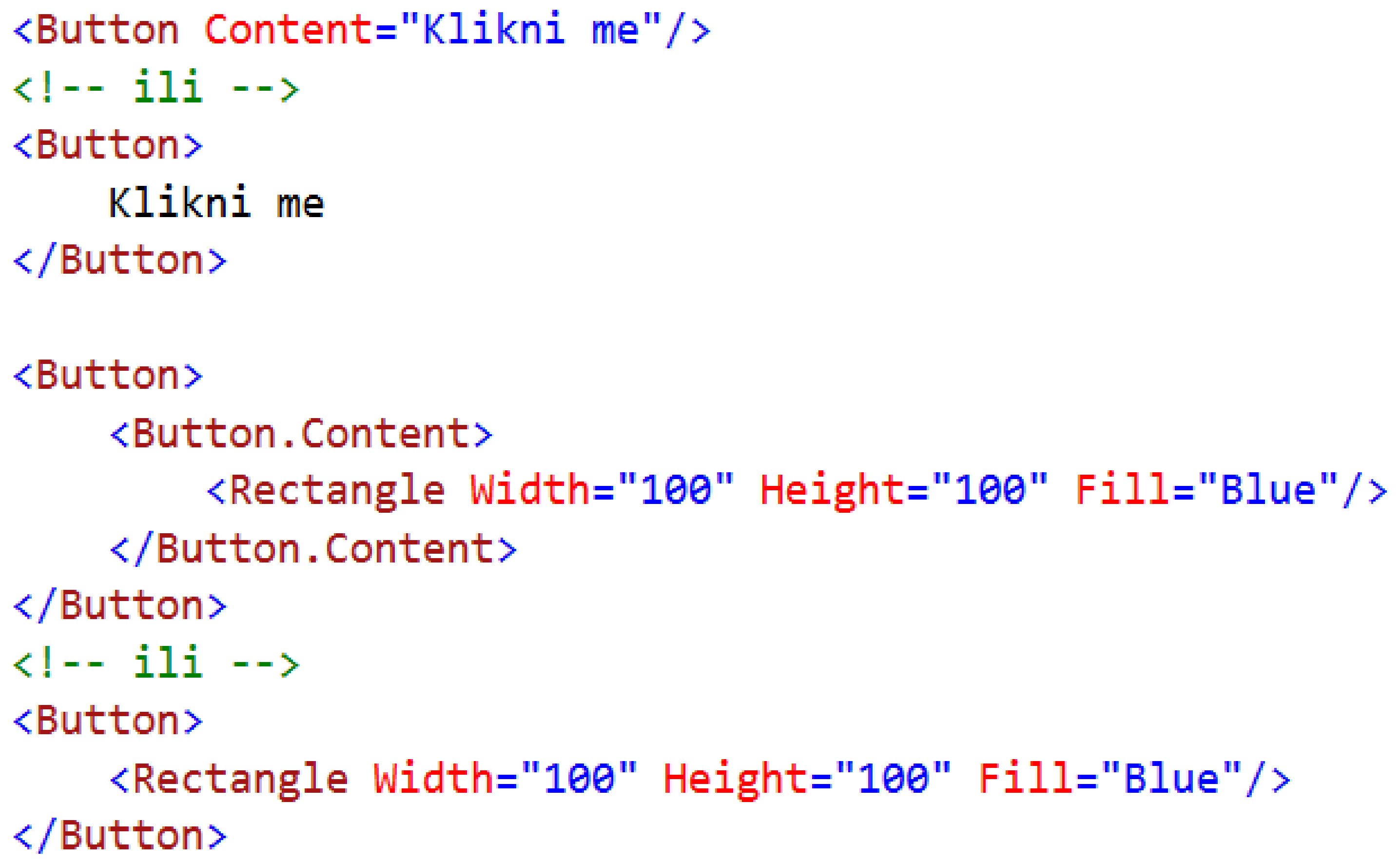
- When using a custom tag extension, the namespace must be specified



```
<Grid xmlns:prefiks="clr-namespace:WpfApplication1">
    <Label Content="{prefiks:Moj}" />
</Grid>
```

Controls with one child

- Individual WPF controls can be assigned a single object as their content (*content controls*)
- Typically, content can be assigned through a property `Content` or as a child, for example:

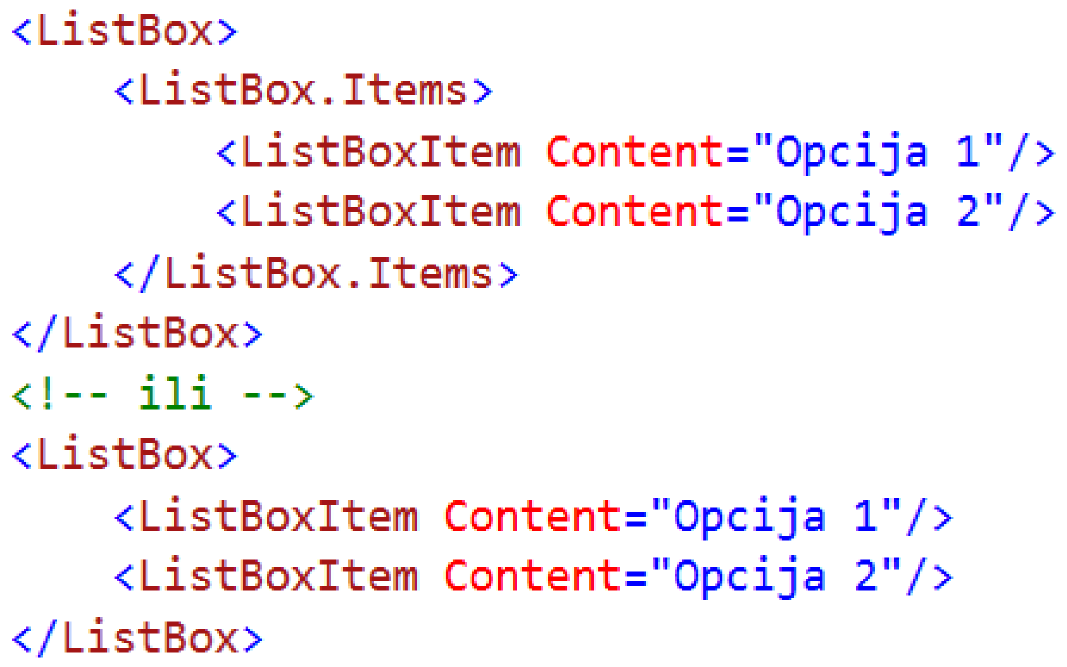


```
<Button Content="Klikni me"/>
<!-- ili -->
<Button>
    Klikni me
</Button>

<Button>
    <Button.Content>
        <Rectangle Width="100" Height="100" Fill="Blue"/>
    </Button.Content>
</Button>
<!-- ili -->
<Button>
    <Rectangle Width="100" Height="100" Fill="Blue"/>
</Button>
```

Controls with multiple children

- Individual WPF controls can have multiple objects as content
    - For example `ComboBox`, `ListBox`, `TabControl`, ...
    - Each object can be a control or some other object
- Typically, content can be assigned through a property `Items` or as multiple children (`Items` is *content property* for e.g. `ListBox`), for example:



```
<ListBox>
    <ListBox.Items>
        <ListBoxItem Content="Opcija 1"/>
        <ListBoxItem Content="Opcija 2"/>
    </ListBox.Items>
</ListBox>
<!-- ili -->
<ListBox>
    <ListBoxItem Content="Opcija 1"/>
    <ListBoxItem Content="Opcija 2"/>
</ListBox>
```

Attached properties

- An attached property is a dependent property that can be assigned values on classes other than the one where it is defined
- eg we want to define the font type and size to `StackPanel` class that does not have these properties
  - The desired properties are defined at `TextElement` class and can be assigned via attached properties

```xml
<StackPanel
    TextElement.FontFamily="Arial"
    TextElement.FontSize="30">
    <Label Content="Pozdrav"/>
</StackPanel>

<Canvas>
    <Button Canvas.Top="20"
            Canvas.Left="20"
            Content="Klikni me"/>
</Canvas>
```