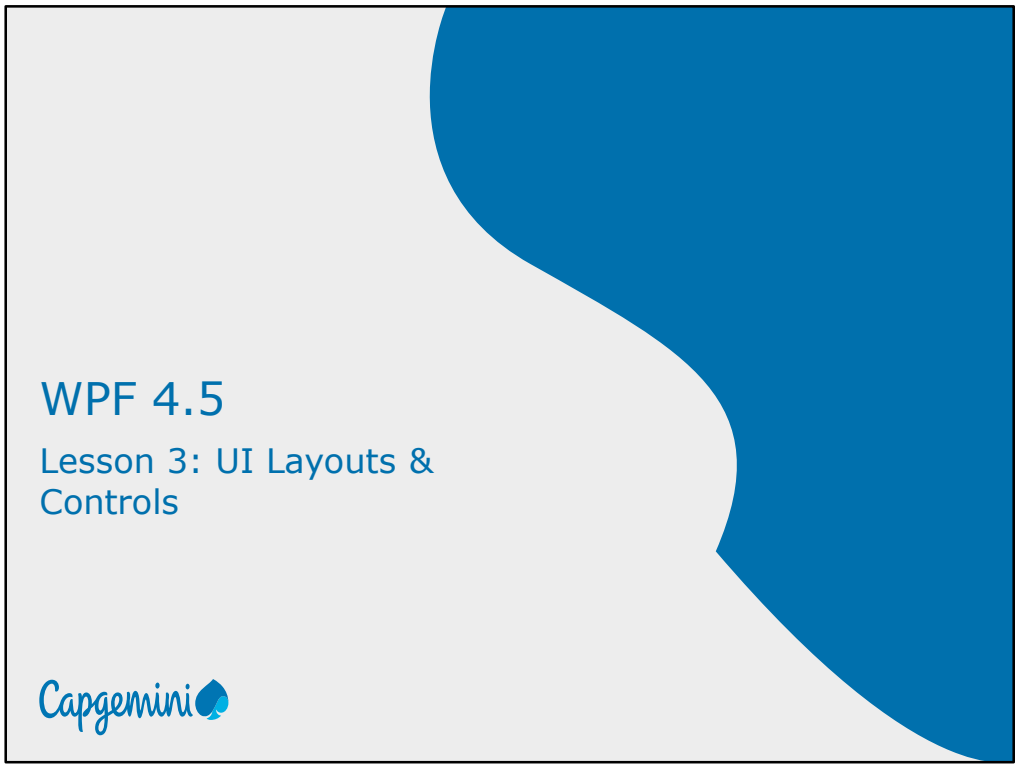
WPF 4.5
Lesson 3: UI Layouts &
Controls
Capgemini

Lesson Objectives

- In this lesson, you will learn:
  - UI Layouts
  - WPF Controls
  - Dependency Properties

3.1: UI Layouts

Concept

- While building a UI, one of the first issues you will deal with is how to arrange all the UI pieces on screen
- In previous MS technologies, we have had limited support for layout
- .NET 2.0 offers WinForms developers some long awaited options in this area
- WPF, however, has made layout a first class citizen from the beginning. There is quite a variety of layout options to choose from

WPF Design Principles: Shapes, Controls, and Layouts:
UI Layouts:
All applications need to present information to users. For this information to be conveyed effectively, it should be arranged onscreen in a clear and logical way. WPF provides a powerful and flexible array of tools for controlling the layout of the user interface.
WPF provides a set of panels – elements that handle layout. Each individual panel type offers a straightforward and easily understood layout mechanism. As with all WPF elements, layout objects can be composed in any number of different ways, so while each individual element type is fairly simple, the flexible way in which they can be combined makes for a very powerful layout system.

3.1: UI Layouts

Stack Panel

- StackPanel is one of the simplest layout options available
  - It does exactly what its name implies; stack elements, either vertically or horizontally
  - By default it is "vertical". However, you can specify "horizontal" by setting the **Orientation property**

WPF Design Principles: Shapes, Controls, and Layouts:
StackPanel:
StackPanel is a very simple panel. It arranges its child elements in a row or a column.
You will rarely use StackPanel to lay out your whole user interface. It is at its most useful for small-scale layout.
You use DockPanel or Grid to define the overall structure of your user interface. Subsequently, you use StackPanel to manage the details.

3.1: UI Layouts

Stack Panel (Contd…)

- Let us see an example of Stack Panel:

```
<StackPanel>
  <TextBlock>My UI</TextBlock>
  <ListBox>
    <ListBoxItem>Item 1</ListBoxItem>
    <ListBoxItem>Item 2</ListBoxItem>
  </ListBox>
  <RichTextBox/>
</StackPanel>
```

3.1: UI Layouts

Wrap Panel

- Wrap Panel positions the children from left to right, one after the other as long as they fit into the line Subsequently, it continues with the next line

3.1: UI Layouts

DockPanel

- DockPanel allows you to dock elements to the top, bottom, left, or right of the container
- The last element will, by default, fill the remaining space

3.1: UI Layouts

DockPanel (Contd…)

- Let us see an example of DockPanel:

```
<DockPanel>
  <TextBlock DockPanel.Dock="Top">My UI</TextBlock>
  <ListBox DockPanel.Dock="Right">
    <ListBoxItem>Item 1</ListBoxItem>
    <ListBoxItem>Item 2</ListBoxItem>
  </ListBox>
  <RichTextBox/>
</DockPanel>
```

3.1: UI Layouts

Grid Layout

- Using the Grid you can arrange your controls with rows and columns
- For every column you can specify a ColumnDefinition, and for every row a RowDefinition

WPF Design Principles: Shapes, Controls, and Layouts:
Grid Layout:
The Grid needs to know how many columns and rows we require, and we indicate this by specifying a series of ColumnDefinition and RowDefinition elements at the start. This may seem rather verbose.
A simple pair of properties on the Grid itself might seem like a simpler solution. However, you will typically want to control the characteristics of each column and row independently. So in practice, it makes sense to have elements representing them.

3.1: UI Layouts

Canvas

- Canvas is a panel that allows explicit positioning of controls
- Canvas defines the attached properties Left, Right, Top, and Bottom that can be used by the children for positioning within the panel

WPF Design Principles: Shapes, Controls, and Layouts:
Canvas:
Occasionally, the automatic layout offered by DockPanel, StackPanel, or Grid will not enable the look you require. Thus it will be necessary to take complete control of the precise positioning of every element.
For example: When you want to build an image out of graphical elements, the positioning of the elements is dictated by the picture you are creating, not by any set of automated layout rules. For these scenarios, you will want to use the Canvas.
The Canvas is the simplest of the panels. It allows the location of child elements to be specified precisely relative to the edges of the canvas. The Canvas does not really do any layout at all. It simply puts things where you tell it to.
While using a Canvas, you must specify the location of each child element. If you do not do so, all your elements will end up at the top left-hand corner. Canvas defines four attached properties for setting the position of child elements. Vertical position is set with either the Top or Bottom property, and horizontal position is determined by either the Left or Right property.

3.1: UI Layouts

Demo

- Demo of various Layouts

3.2: WPF Controls

Classification

- Controls can be classified as follows:
  - Simple controls
  - Content controls
  - Headered content controls
  - Items controls
  - Headered items controls

WPF Design Principles: Shapes, Controls, and Layouts:
Controls:
Controls are the building blocks of an application's user interface. They are interactive features such as text boxes, buttons, or listboxes. You may be familiar with similar constructs from other user-interface technologies most UI frameworks offer an abstraction similar to a control.
However, WPF is somewhat unusual, in that controls are typically not directly responsible for their own appearance. Many GUI frameworks require you to write a custom control when customizing a control's appearance. In WPF, this is not necessary. Nested content, and templates offer powerful yet simpler solutions. You only need to write a custom control if you need behavior that is different from any of the built-in controls.

3.2: WPF Controls

Simple Controls

- Simple controls are controls that do not have a Content property
  - Example: Slider, PasswordBox, ScrollBar, ProgressBar, TextBox, RichTextBox

WPF Design Principles: Shapes, Controls, and Layouts:
Controls – Simple Controls:
Slider and Scroll Controls:
WPF provides controls that allow a value to be selected from a range. They all offer a similar appearance and usage. They show a track, indicating the range, and a “thumb” with which the value can be adjusted.
There are two slider controls, HorizontalSlider and VerticalSlider. There are also two scrollbar controls, HorizontalScrollBar and VerticalScrollBar. The main difference is one of convention rather than functionality. The scrollbar controls are commonly used in conjunction with some scrolling viewable area, while the sliders are used to adjust values.

3.2: WPF Controls

Content Controls

- A ContentControl has a Content property
- With the Content property, you can add any content to the control.
- The Button class derives from the base class ContentControl, so you can add any content to this control
  - Button ,RepeatButton ,ToggleButton ,CheckBox ,RadioButton
  - Label, Frame, ListBoxItem, ToolTip, Window

WPF Design Principles: Shapes, Controls, and Layouts:
Controls – Content Control:
Buttons are controls that a user can click.
The result of the click is up to the application developer. However, there are common expectations, depending on the type of button.
For example: Clicking on a CheckBox or RadioButton is used to express a choice and does not normally have any immediate effect beyond visually reflecting that choice. By contrast, clicking on a normal Button usually has some immediate effect.

3.2: WPF Controls

Headered Content Controls

- Content controls with a header are derived from the base class HeaderedContentControl
  - HeaderedContentControl itself is derived from the base class ContentControl
  - The class HeaderedContentControl has a property Header to define the content of the header and HeaderTemplate for complete customization of the header
  - Example: Expander, GroupBox, TabItem

3.2: WPF Controls

Items Controls

- The class ItemsControl contains a list of items that can be accessed with the Items property
  - Example: Menu, ContextMenu, ListBox, ComboBox, TabControl, StatusBar
- HeaderedItemsControl is the base class of controls that include items but also has a header
  - Example: MenuItem, ToolBar, TreeViewItem

WPF Design Principles: Shapes, Controls, and Layouts:
Controls – Items and Headered Items Controls:
Many Windows applications provide access to their functionality through a hierarchy of menus. These are typically presented as either a main menu at the top of the window or as a pop-up “context” menu. WPF provides two menu controls. Menu is for permanently visible menus (such as a main menu), and ContextMenu is for context menus.
Most Windows applications offer toolbars as well as menus. Toolbars provide faster access for frequently used operations. This is because the user does not need to navigate through the menu system; the toolbar is always visible onscreen.

3.2: WPF Controls

Demo

- Using Controls

3.3: Dependency Properties

Classification

- WPF introduces a new type of property called a dependency property. It is used throughout the platform to enable styling, automatic data binding, animation, and more
- In practice, dependency properties are just normal .NET properties hooked into some extra WPF infrastructure
- This is all accomplished via WPF APIs. None of the .NET languages (other than XAML) have an intrinsic understanding of a dependency property.

WPF Design Principles:
Dependency Properties:
A dependency property depends on multiple providers for determining its value at any point in time.
These providers can be an animation continuously changing its value, a parent element whose property value trickles down to its children, and so on.
Arguably the biggest feature of a dependency property is its built-in ability to provide change notification.

3.3: Dependency Properties

Introduction

- The biggest feature of a dependency property is its built-in ability to provide change notification
- The motivation for adding such intelligence to properties is to enable rich functionality directly from declarative markup
- In practice, dependency properties are just normal .NET properties hooked into some extra WPF infrastructure

WPF Design Principles:
Dependency Properties (contd.):
Properties can be easily set in XAML (directly or by a design tool) without any procedural code.
However, without the extra plumbing in dependency properties, it will be hard for the simple action of setting properties to get the desired results without writing additional code.
This is all accomplished via WPF APIs. None of the .NET languages (other than XAML) have an intrinsic understanding of a dependency property.

3.3: Dependency Properties

Implementation

- Here is a standard dependency property implementation:

```
public class Button : ButtonBase
{
  // The dependency property
  public static readonly DependencyProperty IsDefaultProperty;

  static Button()
  {
    // Register the property
    Button.IsDefaultProperty = DependencyProperty.Register("IsDefault",
      typeof(bool), typeof(Button),
      new FrameworkPropertyMetadata(false,
      new PropertyChangedCallback(OnIsDefaultChanged)));
    ...
  }

  // A .NET property wrapper (optional)
  public bool IsDefault
  {
    get { return (bool)GetValue(Button.IsDefaultProperty); }
    set { SetValue(Button.IsDefaultProperty, value); }
```

WPF Design Principles:
Dependency Property Implementation:
The static IsDefaultProperty field is the actual dependency property, represented by the System.Windows.DependencyProperty class. By convention all DependencyProperty fields are public, static, and have a Property suffix. Dependency properties are usually created by calling the static DependencyProperty.Register method, which requires a name (IsDefault), a property type (bool), and the type of the class claiming to own the property (Button). Optionally (via different overloads of Register), you can pass metadata that customizes how the property is treated by WPF, as well as callbacks for handling property value changes, coercing values, and validating values. Button calls an overload of Register in its static constructor to give the dependency property a default value of false and to attach a delegate for change notifications. Finally, the traditional .NET property called IsDefault implements its accessors by calling GetValue and SetValue methods inherited from System.Windows.DependencyObject, a low-level base class from which all classes with dependency properties must derive.

GetValue returns the last value passed to SetValue, or if SetValue has never been called, the default value registered with the property. The IsDefault .NET property (sometimes called a property wrapper in this context) is not strictly necessary. Consumers of Button can always directly call the GetValue/SetValue methods because they are exposed publicly. However, the .NET property makes programmatic reading and writing of the property much more natural for consumers, and it enables the property to be set via XAML.

3.3: Dependency Properties

Implementation (Contd…)

- The static IsDefaultProperty field is the actual dependency property, represented by the System.Windows.DependencyProperty class
- By convention all DependencyProperty fields are public, static, and have a Property suffix
- Dependency properties are usually created by calling the static DependencyProperty.Register method, which requires a name (IsDefault), a property type (bool), and the type of the class claiming to own the property (Button)

3.3: Dependency Properties

Implementation (Contd…)

- Finally, the traditional .NET property called IsDefault implements its accessors by calling GetValue and SetValue methods inherited from System.Windows.DependencyObject, a low-level base class from which all classes with dependency properties must derive

3.3: Dependency Properties

Change Notification

- Whenever the value of a dependency property changes, WPF can automatically trigger a number of actions depending on the property's metadata
- These actions can be:
  - re-rendering the appropriate elements
  - updating the current layout
  - refreshing data bindings, and much more

WPF Design Principles:
Change Notification:
One of the most interesting features enabled by this built-in change notification is property triggers, which enable you to perform your own custom actions when a property value changes without writing any procedural code.
For example: You want the text in each Button on the Window to turn blue when the mouse pointer hovers over it.

3.3: Dependency Properties

Change Notification (Contd…)

- One of the most interesting features enabled by this built-in change notification is property triggers, which enable you to perform your own custom actions when a property value changes without writing any procedural code

- For example, if you want the text in each Button on the Window to turn blue when the mouse pointer hovers over it

3.3: Dependency Properties

Change Notification – With Property Triggers

```
<Button MinWidth="75" Margin="10">
 Button.Style>
<Style TargetType="{x:Type Button}">
<Style.Triggers>
<Trigger Property="IsMouseOver" Value="True">
<Setter Property="Foreground" Value="Blue"/>
</Trigger>
</Style.Triggers>
</Style>
</Button.Style>
OK
</Button>
```

WPF Design Principles:
Change Notification – With Property Triggers:
Without property triggers, you can attach two event handlers to each Button, one for its MouseEnter event and one for its MouseLeave event.
These two handlers can be implemented in a C# code-behind file with a property trigger. However, you can accomplish this same behavior purely in XAML.
This trigger can act upon Button's IsMouseOver property, which becomes true at the same time the MouseEnter event is raised and false at the same time the MouseLeave event is raised.
You do not have to worry about reverting Foreground to black when IsMouseOver changes to false. This is automatically done by WPF!

3.3: Dependency Properties

Change Notification (Contd…)

- This trigger can act upon Button's IsMouseOver property, which becomes true at the same time the MouseEnter event is raised and false at the same time the MouseLeave event is raised
- You don't have to worry about reverting Foreground to black when IsMouseOver changes to false
- This is automatically done by WPF!

3.3: Dependency Properties

Demo

- Implementing Dependency Properties
- Use of property Triggers

Summary

- WPF has made layout a first class citizen from the beginning. There is quite a variety of layout options to choose from
- The various Layouts like StackPanel, Wrap Panel, Grid etc.
- The various Controls -
  - Simple controls,Content controls, Headered content controls, Items controls, Headered items controls
- Working with Dependency Properties