WPF201-B

Module 6

Additional WPF Features & Controls

1 Other Resources & Controls

1.1 Value Converters

You can use custom value converters to convert data to other types. For example we may want certain elements to be painted using a different brush based AccountType. We can implement a value converter for this. A converter is any class implementing **IValueConverter** interface. Add **Converters** folder into your project containing the following class.

Implementing a value converter: Converters\AccountTypeToBrushConverter.cs

```
using System;
using System.Windows.Data;

namespace SymBank.Converters {
    public class AccountTypeToBrushConverter : IValueConverter {
                :
    }
}
```

A value converter can implement **Convert** and **ConvertBack** methods. For displaying purposes you only need to implement **Convert**. ConvertBack will only be required for input purposes. Even though we are only going to obtain a brush for display and not to obtain a brush for input we will still implement both methods. First declare an array to hold on to a set of brushes; one for each account type.

Brushes to be used in converter

```
private static Brush[] _brushes = {
    Brushes.Azure, Brushes.Beige, Brushes.Aquamarine
};
```

Check the **value** parameter for the original value and return the converted value. In our converter, Convert method accepts AccountType and returns Brush. ConvertBack accepts a Brush and returns the corresponding AccountType.

Convert AccountType to Brush

```
public object Convert(object value, Type targetType, object parameter,
    System.Globalization.CultureInfo culture) {
    if (!(value is AccountType)) return Brushes.White;
    var accountType = (AccountType)value;
    return _brushes[(int)accountType];
}
```

To use converters anywhere in your application, you can create an instance and place it into your application resource dictionary. Register prefix for **SymBank.Converters** namespace and add one instance of the converter.

Instantiate converter as an application resource: Themes\Default.xaml

```
<ResourceDictionary
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:vc="clr-namespace:SymBank.Converters">
    <vc:AccountTypeToBrushConverter
        x:Key="accountTypeToBrushConverter" />
                :
</ResourceDictionary>
```

Use the **Converter** binding expression property to assign a converter to use in any bindings where the binding value is not the same as the type of property it binds to. Following shows an example where the background of a Border is affected by the value from the binding source. There is no need to create Styles and DataTriggers to implement the same effect.

Example of using converter: AddAccountView.xaml

```
<Grid Background={Binding Account.Type,
    Converter={StaticResource accountTypeToBrushConverter}}"
                :
```

You do not actually need a converter if you have a viewmodel or an entity-viewmodel as you can add additional properties into them to support the UI. For example we can expose account type as a brush by adding a new property that returns a brush. We can make sure that the brush will be refreshed when the account type is changed. Thus there is no need to implement or assign a converter for bindings.

Brushes for each account type: ViewModels/AddAccountViewModel.cs

```
public static readonly Brush[] TypeBrushes = {
    Brushes.Azure, Brushes.Beige, Brushes.Aquamarine };
```

Property returns account type as brush

```
public Brush AccountTypeBrush {
    get { return TypeBrushes[(int)_m.Type]; }
}
```

Ensure brush is refreshed when account type changes

```
public AccountType AccountType {
    get { return _account.Type; }
    set { _account.Type = value;
        NotifyPropertyChanged("AccountType");
        NotifyPropertyChanged("AccountTypeBrush");
    }
}
```

Binding to brush property instead

```
<Grid
    Background={Binding AccountTypeBrush}"
```

Bind to AccountType instead that will also refresh AccountTypeBrush

```
<ComboBox x:Name="cbxType"
    SelectedItem="{Binding AccountType}" ... />
```

1.2 ContextMenu

ContextMenu is a great way to provide additional interaction with individual elements in the UI without cluttering the UI with too much UI functionality. The context menu will be opened automatically when the user right-click on the UI element where the menu is attached. You can also manually open or close the context menu by setting **IsOpen** property. Go back to the application and we will add an extra command to the **AddAccountViewModel** to clear the picture.

Adding clear picture feature: AddAccountViewModel.cs

```
public RelayCommand ClearPictureCommand { get; private set; }

public AddAccountViewModel() {
    ClearPictureCommand = new RelayCommand(
        p => Account.Picture = null);
                        :
```

You can attach ContextMenu to any UI element using the **ContextMenu** property. A ContextMenu will also generate **Opened** and **Closed** events if you wanted to detect and perform additional operations when menu is opened or closed. A ContextMenu just like Menu can contain **MenuItem** elements. You can use commands or attach event handler to the **Click** event. In the example below, we provide a menu option to clear the game picture.

Attaching a ContextMenu to an element: AddAccountView.xaml

```
<Border.ContextMenu>
    <ContextMenu x:Name="mnuPicture"
        Opened="mnuPicture_Opened"
        Closed="mnuPicture_Closed">
        <MenuItem Header="_Clear"
            Command="{Binding ClearPictureCommand}">
            <MenuItem.Icon>
                <Image Stretch="None"
                    Source="{StaticResource CancelImage}" />
            </MenuItem.Icon>
        </MenuItem>
    </ContextMenu>
</Border.ContextMenu>
```

1.3 File Drop

Rather than having the user to select a button and use the open file dialog to select a file to import into an application, you can also add support file drag and drop on UI elements. The onlu problem is commands are not supported so you still need to have an event handler to process the drop. Set **AllowDrop** property on the element and assign a handlde to the **Drop** event. Use **GetData** method on the **Data** property from event arguments to detect and retrieve what was dropped on the element. For **FileDrop**, you receive an array containing one or more filenames.

Allowing drop on game picture border: AddAccountView.xaml

```
<Border x:Name="bdrPicture"
        Background="Gray" BorderBrush="DarkGray"
        BorderThickness="2" Padding="8" Margin="4"
        AllowDrop="True" Drop="bdrPicture_Drop">
                :
```

If you are using a viewmodel, you can then retrieve the viewmodel and pass in one or more filenames so that the viewmodel can load the picture.

Loading and assigning game picture: AddAccountView.xaml.cs

```
private void bdrPicture_Drop(object sender, DragEventArgs e) {
    var files = (string[])e.Data.GetData(DataFormats.FileDrop, true);
    var vm = (AddAccountViewModel)DataContext;
    try { vm.LoadPicture(files[0]); }
    catch (Exception) { }
}
```

1.4 Animation

It is possible to create and perform dynamic animations using code without having to use Storyboards. For example let us add some animation to the ListBox to highlight the display of new results. Attach an event handler to the **PropertyChanged** event of the viewmodel to detect property changed notifications.

Attaching property changed event handler

```
public SearchAccountsView() {
    InitializeComponent();
    var vm = (SearchAccountsViewModel)DataContext;
    vm.PropertyChanged += ViewModel_PropertyChanged;
}
```

In the event handler, check if the correct property has been changed and then create the animation. To start the animation, call a **BeginAnimation** method on the element with the dependency property that you wish to animate. This is why you need to use Storyboards in XAML in order to help you call this method.

Creating and starting an animation

```
public void ViewModel_PropertyChanged(
    object sender, PropertyChangedEventArgs e) {
    if (!e.PropertyName.Equals("Results")) return;
    var d = new Duration(TimeSpan.FromSeconds(0.1));
    var a = new DoubleAnimation(1.0,0.5,d,FillBehavior.Stop);
    a.RepeatBehavior = new RepeatBehavior(3); a.AutoReverse = true;
    lsbResults.BeginAnimation(ListBox.OpacityProperty,a);
}
```

2 Printing

2.1 Report Design

A **UserControl** can be used to generate the visual content for a report. Data can be inserted into the control before it is printed. Always use a **Viewbox** to scale the visual content to match the printer resolution.

Using UserControl for page content: Reports/AccountListReport.xaml

```
<Viewbox Stretch="Uniform" VerticalAlignment="Top">
    <StackPanel Margin="16">
        <TextBlock Text="List of Accounts" FontFamily="Georgia" FontSize="32"
            Margin="16" HorizontalAlignment="Center"/>
        <Grid><Border BorderBrush="Black" BorderThickness="2" />
            <StackPanel Margin="8" Orientation="Horizontal"
                    HorizontalAlignment="Stretch">
                <Label FontSize="14" Width="100" Content="Code" />
                <Label FontSize="14" Width="400" Content="Name" />
                <Label FontSize="14" Width="200" Content="Balance"
                    HorizontalContentAlignment="Right" /></StackPanel>
        </Grid><ListBox x:Name="lsbAccounts">
            <ListBox.ItemTemplate>
                <DataTemplate>
                    <Grid>
                        <StackPanel Orientation="Horizontal"
                            HorizontalAlignment="Stretch">
                            <Label FontSize="14" Width="100"
                                Content="{Binding Code}" />
                            <Label FontSize="14" Width="400"
                                Content="{Binding Name}" />
                            <Label FontSize="14" Width="200"
                                Content="{Binding Balance}"
                                HorizontalContentAlignment="Right" />
                            </StackPanel>
                        </Grid>
                </DataTemplate>
            </ListBox.ItemTemplate></ListBox>
        <TextBlock Margin="16" HorizontalAlignment="Center"
            FontFamily="Georgia" FontSize="12">
            <Run x:Name="txtCurPage">0</Run> /
            <Run x:Name="txtMaxPage">0</Run></TextBlock>
    </StackPanel>
</Viewbox>
```

You can now add properties to the control to allow external data to be assigned to the report before priniting. The following properties allow a list of accounts to be assigned to a ListBox and page numbers to be assigned to Run elements.

Properties to assign data for printing

```
public AccountList Accounts {set { lsbAccounts.ItemsSource = value; }}
public int CurPage {set { txtCurPage.Text = value.ToString();    }}
public int MaxPage {set { txtMaxPage.Text = value.ToString();    }}
```

You can now add proint support to **SearchAccountsView**. Add a **Print** method to the view model and create a command to execute the method. In the view, add a Button to bind to the command.

Add printing support to view model: SearchAccountsViewModel.cs

```
public RelayCommand PrintCommand { get; private set; }

public void Print() { }

public SearchAccountsViewModel() {
    PrintCommand = new RelayCommand(p => Print());
            :
}
```

Add button to print list in view: SearchAccountsView.xaml

```
<Button Tooltip="print results" Command="{Binding PrintCommand}">
    <Image Stretch="None" Source="{StaticResource PrintImage}" />
</Button>
```

2.2 PrintVisual

Now you can program **Print** method to print the results. First thing that you will need to do is to calculate the number of pages that is required to print. This can be done by taking total number of items and dividing by number of items to fit into a single page.

Calculating number of pages

```
if (_results == null || _results.Count == 0) return; // nothing to print
var itemsPerPage = 10;
var totalItems = (uint)_results.Count;
var totalPages = totalItems / itemsPerPage;
if (totalItems % itemsPerPage > 0) ++totalPages;
```

You can then call up the print dialog and let the user confirm which pages that they wish to print. Then you can create your report template and then set it up for printing one or more pages.

Confirmation and setup for printing

```
var pd = new PrintDialog();
pd.MinPage = 1;
pd.MaxPage = (uint)totalPages;
if (pd.ShowDialog() != true) return;
var minPage = (int)pd.MinPage;
var maxPage = (int)pd.MaxPage;
```

Prepare template for printing

```
var template = new AccountListReport();
template.Width = pd.PrintableAreaWidth;
template.Height = pd.PrintableAreaHeight;
template.MaxPage = maxPage;
```

You can now enter your printing loop to print the pages selected. For each page you can easily retrieve the list of accounts for a particular page using **Skip** and **Take** LINQ operators. We have to subtract one from the page number as we only skip for the second page onwards. Multiply page number by number of items per page to find out how many items to skip. Then take only number of items that will fit into one page. Once the information has been assigned to the control, you can print out the control by using **PrintVisual**. This method can be used to print any UI element.

The page printing loop

```
try {
    for (var page = minPage; page <= maxPage; page++) {
        var items = _results.Skip((page - 1) * itemsPerPage).
            Take(itemsPerPage).ToList();
        template.Accounts = items;
        template.CurPage = page;
        pd.PrintVisual(template, "");
    }
}
    catch (Exception ex) {
        Shell.Failure("Error printing list.\n" + ex.Message);
}
```

2.3 PrintDocument

Currently we are using a single user control to print all the pages. You can also print different pages using different controls by checking the page number and then choose a different page template to print. One problem though is **PrintVisual** creates one job for each page. Thus if you're exporting to XPS, each page is saved as a separate document. To solve this we can embed the report within a **FlowDocument**. We can then print the document as a single file. Instead of creating a UserControl before the print loop, create and setup the flow document instead. In the loop, pages are added into paragraphs where each one is printed in a separate page. At the end of the print loop, print the entire **FlowDocument** instead.

Preparing the flow document

```
var doc = new FlowDocument();
doc.PageHeight = pd.PrintableAreaHeight;
doc.PageWidth = pd.PrintableAreaWidth;
doc.ColumnWidth = pd.PrintableAreaWidth;
doc.PagePadding = new Thickness(0);
doc.ColumnGap = 0;
```

Adding pages in paragraphs of the document

```
for (var page = minPage; page <= maxPage; page++) {
    var items = _results.Skip((page - 1) * itemsPerPage).
        Take(itemsPerPage).ToList();
    var p = new Paragraph(); p.BreakPageBefore = true;
    var template = new AccountListReport();
    template.Height = pd.PrintableAreaHeight;
    template.Width = pd.PrintableAreaWidth;
    template.Accounts = items;
    template.CurPage = page;
    template.MaxPage = maxPage;
    p.Inlines.Add(template);
    doc.Blocks.Add(p);
}

IDocumentPaginatorSource dps = doc;
pd.PrintDocument(dps.DocumentPaginator, "Account List");
```

3 Asynchronous Operations

3.1 Asynchronous Methods

In WPF/.NET 4.5 you can also make use of **async** / **await** together with TPL library to perform asynchronous operations from WPF UI. However this is more suitable for MVC than MVVM applications. First you need to implement an asynchronous method where operations are performed using a separate task. The asynchronous method creates a task, starts the task and returns it.

Asynchronous controller operation: AccountController.cs

```
public Task<AccountList> GetAccountsForNameAsync(string name) {
    var task = Task<AccountList>(() => GetAccountsForName(name));
    task.Start(); return task;
}
```

Calling the method does not block the caller thread since the task will be executed on a separate thread. However sometimes you still need to wait for the task to complete in order to obtain the results and update the UI. Waiting for results would also block the caller thread unless the caller method runs asynchronously as well. By applying **async** / **await** on a method ensures that a caller thread is not blocked when waiting for the results but would still process the results once the task is completed.

Using .NET 4.5 asynchronous call pattern: SearchAccountsView.xaml.cs

```
private async void btnSearch_Click(object sender, RoutedEventArgs e) {
    IsEnabled = false;
    try {
        var c = new AccountController();
        var results = await c.GetAccountsForNameAsync(txtName.Text);
        lsbResults.ItemsSource = results;
    }
    catch (Exception ex) {
        Shell.Failure(
            "Error searching for accounts.\n"
                + ex.Message);
    }
    finally {
        IsEnabled = true;
    }
}
```

3.2 Asynchronous Commands

Can commands be executed asynchronously from UI? In short terms yes, but you will need to mark **Execute** as **async** and implement a **Task** to execute command code in a separate thread that Execute can **await** on. The **CanExecute** must also be coded to return **false** when the task is running and only return back to original state when it is completed. If you're using delegates in your command, you should add one more to update the UI when the task is completed. Since the task may fail, the delegate must be able to accept an exception.

Asynchronous RelayCommand: AsyncRelayCommand.cs

```
public class AsyncRelayCommand : ICommand {
    private Action<object> _execute;
    private Predicate<object> _canExecute;
    private Action<Exception> _completed;
    private bool _running;

    public AsyncRelayCommand(Action<object> execute = null,
        Predicate<object> canExecute = null,
        Action<Exception> completed = null) {
        _execute = execute; _canExecute = canExecute;
        _completed = completed;
    }
    public bool Running { get { return _running; } }
    public event EventHandler CanExecuteChanged;
    public void NotifyCanExecuteChanged() {
        if (CanExecuteChanged != null)
            CanExecuteChanged(this, EventArgs.Empty);
    }
    public bool CanExecute(object parameter) {
        if (_running) return false;
        if (_canExecute == null) return true;
        return _canExecute(parameter);
    }
    private Task ExecuteTask(object parameter) {
        var task = new Task(() => _execute(parameter));
        task.Start(); return task;
    }
    public async void Execute(object parameter) {
        if (_running) return; _running = true; NotifyCanExecuteChanged();
        try {
            if (_execute != null) await ExecuteTask(parameter);
            if (_completed != null) _completed(null);
        }   catch (Exception ex) {
            if (_completed != null) _completed(ex);
        }
        _running = false; NotifyCanExecuteChanged();
    }
}
```

Methods to search and process the results

```
private AccountList _tempResults;

private void SearchExecute(object p) {
    _tempResults = _accountController.GetAccountsForName(_searchText);
}

private void SearchCompleted(Exception ex) {
    if (ex != null) {
        Shell.Failure(
            "Error searching for accounts.\n"
            + ex.Message);
        return;
    }
    Results = _tempResults;
}
```

Creating an asynchronous command instead

```
public AsyncRelayCommand SearchCommand { get; private set; }

public SearchAccountsViewModel() {
    SearchCommand = new AsyncRelayCommand(SearchExecute,null,SearchCompleted);
            :
}
```