WPF201-B

Module 1

Implementing a Basic WPF Application

1 Application

Windows Presentation Foundation (WPF) is a UI Framework for the Windows operating system to build visually impressive and highly interactive desktop applications. A WPF application can be implemented using a combination of code and a markup language called Extensible Application Markup Language (XAML). XAML is not compulsory but a complex user-interface will be much harder to implement only by code. It also allows designers without programming knowledge to implement the user-interface part of an application using design tools instead like Microsoft Expression Blend while developers complete the coding part using programming tools like Microsoft Visual Studio. XAML is based on Extensible Markup Language (XML) so the format and structure should be familiar to those that already have experience working with XML. The following shows what can be accomplished using XAML without coding.

Basic operations in XAML

1. Create objects
2. Set object properties
3. Add items to collection properties
4. Attach object event handlers

1.1 Application Project

You can start developing a WPF application by creating a WPF application project in Visual Studio using the information below. There are two XAML documents created; **App.xaml** and **MainWindow.xaml**. Once thing that XAML should not contain is .NET code. If you need to write program code or event handlers, you can do so in an extended class that is assigned to the base class in XAML using the **x:Class** extension property. This is commonly called as the code-behind class. Expand the XAML node in Solution Explorer and you can see the code files for these classes.

Application project Information

```
Project Name     : SymBank
Project Template: Visual C# | WPF Application
Project Location: C:\WPFDEV\SRC
Project Solution: Module1
```

As a programmer you are free to choose whether to use XAML or write program code. We suggest that if a certain operation can be done in XAML then do it in XAML. Only write code for operations that are too complex or impossible to be done with XAML. The basic operations that you can do in XAML is to create objects, set the properties of the objects and attach event handlers. Extended operations can be implemented through markup extensions. Note that we commonly use the term element to refer to an object in XAML.

You only need two XML namespaces to reference all the basic elements and markup extensions in WPF. The default namespace is used for all WPF elements and standard markup extensions. The namespace associated with the **x** prefix is used for extended markup extensions specially to integrate to Visual Studio and to support .NET code and features.

Default namespace for standard WPF features

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

Extension namespace for Visual Studio and .NET integration

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

Code-behind class for App.xaml (App.xaml.cs)

```
using System;
using System.Windows;

namespace SymBank {
    public partial class App : Application {
    }
}
```

Code-behind class for MainWindow.xaml (MainWindow.xaml.cs)

```
using System;
using System.Windows;
using System.Windows.Controls;

namespace SymBank {
    public partial class MainWindow : Window {
        public MainWindow() {
            InitializeComponent();
        }
    }
}
```

The class name does not have to be the same as the XAML document name as they are linked based on the **x:Class** property. Rename XAML documents to **MyApp.xaml** and **Shell.xaml**. Notice the class names remain the same. However if you do change the file name for the main window, make sure you update the **StartupUri** property in application XAML document since it refers to filename and not class name. The project should still build and run correctly.

If you do decide to change the class name then make sure **x:Class** property in XAML is also updated to reference the new name. You can actually do both at the same time using refactor to rename the classes instead. You can now try to rename the classes to **MyApp** and **Shell** respectively. Check **x:Class** property in XAML to make sure they refer to the new names. The project should still build and run correctly.

1.2 Application Properties & Events

An application by default will exit when all windows opened have been closed. You can change to **ShutdownMode** property to **OnMainWindowClose** to exit the application if only the main window is closed even though other windows may still be open. For **OnExplicitShutdown**, application exits only by calling the **Shutdown** method.

ShutdownMode values

```
OnLastWindowClose    exit application when all windows are closed
OnMainWindowClose    exit application only when main window is closed
OnExplicitShutdown   must call Shutdown method to exit application
```

Properties can be set in XAML or by code. Following shows setting of **ShutdownMode** and **StartupUri** properties in XAML. You can also do the same by writing code in the object constructor. It is up to you to decide whether to use XAML or write code but to do both is pointless. Any constructor property setting is overridden by same property setting in XAML as those settings are done after the constructor. Advantage of using code is that the settings can be dynamic rather than static.

Simple property assignment syntax in XAML: MyApp.xaml

```
<Application x:Class="SymBank.MyApp"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    ShutdownMode="OnMainWindowClose"
    StartupUri="Shell.xaml">
</Application>
```

Setting properties by code: MyApp.xaml.cs

```
public partial class MyApp : Application {
    public MyApp() {
        ShutdownMode = ShutdownMode.OnMainWindowClose;
        StartupUri = new Uri("Shell.xaml", UriKind.Relative);
    }
}
```

You can see that setting properties in XAML is much simpler than writing code. This is because standard value converters are automatically used in XAML to decipher and convert the strings you assign to the actual value or object. However there would not be converters available for every type of property so some property assignments may also be complicated.

Attaching event handlers is also simple like setting properties. The handlers can be automatically created in designer by using the events page of the *Properties* window or the *XAML Editor*. If you enter the name of the handler manually, you must navigate to the handler for it to be generated. You can right-click over the handler and select the *Navigate to Event Handler* option from the popup menu to navigate to the source code for that handler in the *Code Editor*. You can also assign handlers by code.

Assigning handlers to application events

```
<Application x:Class="SymBank.MyApp"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Startup="Application_Startup"
    Activated="Application_Activated"
    Deactivated="Application_Deactivated"
    DispatcherUnhandledException="Application_DispatcherUnhandledException"
    SessionEnding="Application_SessionEnding"
    Exit="Application_Exit"
            :
```

Attaching event handlers in code

```
public MyApp() {
    Startup += MyApp_Startup;
    Activated += MyApp_Activated; Deactivated += MyApp_Deactivated;
    DispatcherUnhandledException += MyApp_DispatcherUnhandledException;
    SessionEnding += MyApp_SessionEnding; Exit += MyApp_Exit;
}
```

A **Startup** event occurs when the application is created and initialized but before the main window is instantiated. Even though the constructor is normally used to initialize an object, it should not execute statements that generate runtime errors. Operations that may fail should be performed in the **Startup** event handler. The **Exit** event occurs just before application exits and the domain terminates. You can access or change the application exit code passed by the **ExitEventArgs** parameter but you cannot stop the application from exiting. These event handlers can become useful for loading and saving application settings. The **Activated** event occurs when the focus is switched over to windows in the current application domain from another application and **Deactivated** when focus is switched away from the current application.

Application events

```
private void Application_Startup(object sender, StartupEventArgs e) {
    Debug.WriteLine("Application started!");
}

private void Application_Exit(object sender, ExitEventArgs e) {
     int exitCode = e.ApplicationExitCode;
     Debug.WriteLine($"Application exiting with code {exitCode}!");
}

private void Application_Activated(object sender, EventArgs e) {
    Debug.WriteLine("Application activated!");
}

private void Application_Deactivated(object sender, EventArgs e) {
    Debug.WriteLine("Application deactivated!");
}
```

A WPF application uses **Dispatcher** object to manage and execute operations on the thread that runs the application. If an exception occurred in any of the operations was not handled by the application, the **DispatcherUnhandledException** event will then be raised. Once this handler has completed the application will exit unless you set the **Handled** property in **DispatcherUnhandledExceptionEventArgs** parameter is set. The original **Exception** that triggered this event is available through this parameter. Note the event only occurs for operations performed by the dispatcher thread and not any worker threads created by application code.

Detecting unhandled exceptions in the Dispatcher thread

```
private void Application_DispatcherUnhandledException(
    object sender, DispatcherUnhandledExceptionEventArgs e) {
    Debug.WriteLine($"Exception '{e.Exception.Message}' not handled!");
}
```

An application may terminate if the current user logs out or if the system is shutting down. This can be detected using the **SessionEnding** event. The exact reason can be checked using **ReasonSessionEnding** property in **SessionEndingCancelEventArgs** parameter and you will be able to stop this operation by setting its **Cancel** property. However the user can still enforce the shutdown.

Detecting Windows session terminating

```
private void Application_SessionEnding(
    object sender, SessionEndingCancelEventArgs e) {
    switch (e.ReasonSessionEnding) {
        case ReasonSessionEnding.Logoff:
            Debug.WriteLine("User is logging off!"); break;
        case ReasonSessionEnding.Shutdown:
            Debug.WriteLine("System is shutting down"); break;
    }
    e.Cancel = true;
}
```

1.3 Application Services

When you are building a real-life application, there are common designs, patterns and architectures that you can follow to make it easier for you to build applications of any size and complexity. Since the application object is accessible from anywhere in the application using the static **Application.Current** property it can be utilized to provide non-UI related services and resources to the rest of the application.

Accessing MyApp instance from code

```
var app = (MyApp)Application.Current;
```

However it will be simpler to get access to the application instance by implementing a singleton pattern. Declare a static field to hold the instance which can then be set by the constructor then provide your own static property to return the singleton.

Implementing a MyApp singleton

```
private static MyApp _instance;

public static MyApp Instance { get { return _instance; }}

public MyApp() {
    _instance = this;
}
```

Simpler access to MyApp instance

```
var app = MyApp.Instance
```

Security features like identity and authorization can be provided by the application as shown below. Other features such as logging that does not depend on the application object can still be exposed through the class as static members. You can use the .NET principal system to get the identity of the current user and for checking the roles that the user belongs to.

Exposing identity and authorization features

```
private static IPrincipal _principal;

private void Application_Startup(object sender, StartupEventArgs e) {
    AppDomain.CurrentDomain.SetPrincipalPolicy(
        PrincipalPolicy.WindowsPrincipal);
    _principal = Thread.CurrentPrincipal;
}

public static string UserName {
    get { return _principal.Identity.Name; }
}

public static bool IsInRole(string roleName) {
    return _principal.IsInRole(roleName);
}

public static bool IsInAnyRoles(params string[] roleNames) {
    foreach (var roleName in roleNames)
        if (_principal.IsInRole(roleName)) return true;
    return false;
}

public static bool IsInAllRoles(params string[] roleNames) {
    foreach (var roleName in roleNames)
        if (!_principal.IsInRole(roleName)) return false;
    return true;
}
```

Authorize only users in certain roles to use the application

```
private void Application_Startup(object sender, StartupEventArgs e) {
    AppDomain.CurrentDomain.SetPrincipalPolicy(
        PrincipalPolicy.WindowsPrincipal);
    _principal = Thread.CurrentPrincipal;
    if (!IsInAnyRoles("Administrators", "Banking"))
        throw new Exception("Access denied.");
}
```

Following is a list of static methods that can be used to log information, warnings and errors to a file. A good place to log is when the application encounters an unhandled exception that triggers a **DispatcherUnhandledException** event. The event handler can then log and display the error to the user before the application shuts down. If the current application is being debugged then there is no need for this operation as the developer can check the exception directly from the event arguments.

Exposing logging features with static methods

```
public static void Log(string type, string message) {
    File.AppendAllText("SymBank.log",
        $"[{DateTime.Now:yy-MM-dd hh:mm:ss}] {type}(\r\n{message}\r\n)\r\n"));
}
public static void LogMessage(string message) { Log("MESSAGE", message); }
public static void LogWarning(string message) { Log("WARNING", message); }
public static void LogFailure(string message) { Log("FAILURE", message); }
```

Logging unhandled exceptions when not debugging

```
private void Application_DispatcherUnhandledException(object sender,
    System.Windows.Threading.DispatcherUnhandledExceptionEventArgs e) {
    var ex = e.Exception;
    if (Debugger.IsAttached) Debugger.Break(); else {
        LogFailure(ex.ToString());
        MessageBox.Show(string.Format("A serious error has occurred.\n" +
            "Please contact the administrator.\n" +
            "The error is: {0}\n", ex.Message),
            "Error", MessageBoxButton.OK, MessageBoxImage.Error);
    }
}
```

1.4 Application Resources

Application can also be used to share resources that can be accessed from anywhere in the entire application. One of the common resources are **BitmapImage** objects. In the following example we will embed an icon file to be used for both the application as well as the main window. To embed files, add them to the project. Folders and sub-folders can be used to organize the files. As an example add a folder to the **SymBank** application project named **Themes** containing a sub-folder named **Images**.

Add **SymBank.ico** and other image icons provided by the instructor into the folder. WPF supports many standard image formats including ICO, BMP, PNG, GIF and JPG. To make sure the images are embedded and not distributed as a separate file, check to make sure that the file type is set as **Resource**. An icon is embedded if assigned as the application icon using *Project Properties* | *Application*. The application icon is also used as the default icon for windows.

In WPF all UI element can store resources by adding them to a **Resources** dictionary property for that element. To share a resource in a window use **Window.Resources**. However if you have multiple windows, to share resources across all the windows you should add it to **Application.Resources** instead. All objects except UI elements can be placed in a resource dictionary. A **x:Key** extension is required to assign the unique key to each item that will be used to fetch the resource.

Adding an application resource: MyApp.xaml

```
<Application.Resources>
    <BitmapImage x:Key="SymBankIcon" UriSource="Themes/Images/SymBank.ico" />
</Application.Resources>
```

Additional bitmap images resources to add

```
<BitmapImage x:Key="NotepadImage" UriSource="Themes/Images/notepad.png" />
<BitmapImage x:Key="UserAddImage" UriSource="Themes/Images/user_add.png" />
<BitmapImage x:Key="SearchImage" UriSource="Themes/Images/search.png" />
<BitmapImage x:Key="MoneyImage" UriSource="Themes/Images/money.png" />
<BitmapImage x:Key="TableImage" UriSource="Themes/Images/table.png" />
<BitmapImage x:Key="BankGoImage" UriSource="Themes/Images/bank_go.png" />
<BitmapImage x:Key="WorldGoImage" UriSource="Themes/Images/world_go.png" />
<BitmapImage x:Key="OpenImage" UriSource="Themes/Images/open.png" />
<BitmapImage x:Key="SaveImage" UriSource="Themes/Images/save.png" />
<BitmapImage x:Key="CancelImage" UriSource="Themes/Images/cancel.png" />
<BitmapImage x:Key="RefreshImage" UriSource="Themes/Images/refresh.png" />
<BitmapImage x:Key="PreviousImage" UriSource="Themes/Images/previous.png" />
<BitmapImage x:Key="NextImage" UriSource="Themes/Images/next.png" />
<BitmapImage x:Key="CutImage" UriSource="Themes/Images/cut.png" />
<BitmapImage x:Key="PasteImage" UriSource="Themes/Images/paste.png" />
<BitmapImage x:Key="TextBoldImage" UriSource="Themes/Images/text_bold.png" />
<BitmapImage x:Key="TextItalicImage" UriSource="Themes/Images/text_italic.png" />
<BitmapImage x:Key="TextUnderlineImage" UriSource="Themes/Images/text_underline.png" />
<BitmapImage x:Key="AlignLeftImage" UriSource="Themes/Images/align_left.png" />
<BitmapImage x:Key="AlignRightImage" UriSource="Themes/Images/align_right.png" />
<BitmapImage x:Key="AlignCenterImage" UriSource="Themes/Images/align_center.png" />
<BitmapImage x:Key="AlignJustifyImage" UriSource="Themes/Images/align_justify.png" />
<BitmapImage x:Key="ErrorImage" UriSource="Themes/Images/error.png" />
<BitmapImage x:Key="PrintImage" UriSource="Themes/Images/print.png" />
```

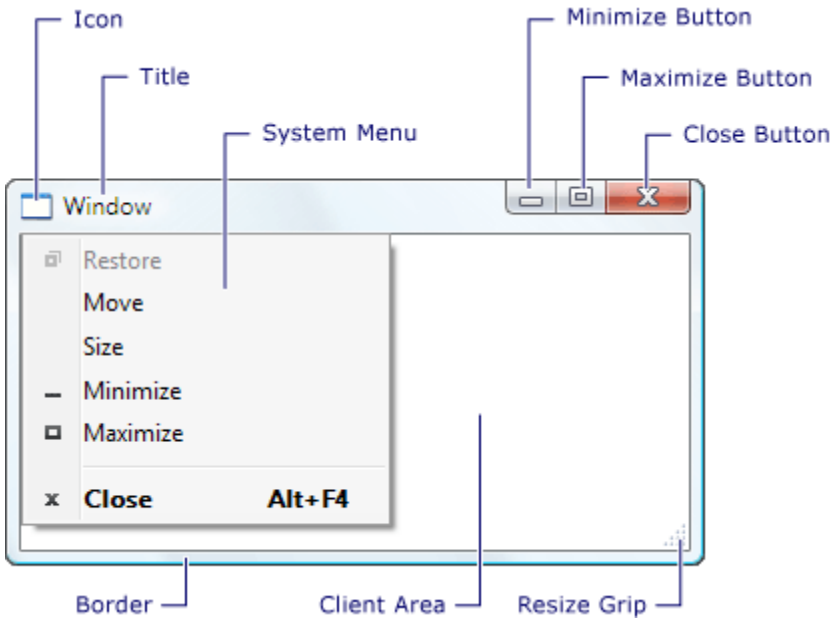
Use a **StaticResource** or a **DynamicResource** markup extension to bind a resource to properties of UI elements. A static resource binding will only lookup the resource one time while dynamic resources are reassigned when they are changed. However WPF would need to setup to listen to resource change events which is not required for resources that are never changed.

2 Window

2.1 Window Properties & Events

A **Window** is a **ContentControl** that displays content in a window frame. The frame is used to resize and move the window. Use the **ResizeMode** property to determine whether a window can be resized by the user. Actual size of the window is determined by **Width** and **Height** and use **MinWidth**, **MinHeight**, **MaxWidth** and **MaxHeight** to constraint the size. ResizeMode also determines if *Minimize* and *Maximize* buttons will appear. To make it easier to resize set it to **CanResizeWithGrip** to add a resize grip at bottom right corner of the window. A **SizeToContent** property can be used to automatically size the window based on the content.

Anatomy of a window



The **WindowStyle** property changes the border style. **Icon** and **Title** will not appear if the **WindowStyle** is set to **None**. Start position of the window is determined by the **WindowStartupLocation** property. If property is set to **Manual**, the actual position of the window will be determined by the **Left** and **Top** properties. You can also make a window to always appear in front of other normal windows with **Topmost** property. **WindowState** determines if the window is **Normal**, **Minimized** or **Maximized**. The above shows the standards part of a window that are commonly available as long as WindowStyle is not altered. If **Icon** is not set the application icon is used instead.

Setting Window properties in XAML: Shell.xaml

```
<Window x:Class="SymBank.Shell"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="SymBank Application"
    Icon="{StaticResource SymBankIcon}"
    WindowStartupLocation="CenterScreen"
    ResizeMode="CanResizeWithGrip"
    MinWidth="400" MinHeight="320"
    Width="600" Height="400">
```

There are a set of specific events for a window. The **Loaded** event occurs when the window is ready to be presented to the user for interaction. You can use this instead of the constructor to perform operations that may potentially fail. The **Closing** event occurs before the window is closed. This allows you to cancel the operation by setting **Cancel** property in **CancelEventArgs** parameter to true. **Closed** event occurs when the window has been closed. When user switches between windows, current window becomes deactivated and new window becomes activated. This causes **Activated** and **Deactivated** events to occur. You can use **IsActive** property to check if a window is the active window. If window location, size or state changes the **LocationChanged**, **SizeChanged** or **StateChanged** events occur. Following shows the handlers for all of these events and the properties to access the current location, size and state of the window.

Assigning handlers to Window events

```
<Window x:Class="SymBank.Shell"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Loaded="Window_Loaded"
    Activated="Window_Activated"
    Deactivated="Window_Deactivated"
    Closing="Window_Closing"
    Closed="Window_Closed"
    LocationChanged="Window_LocationChanged"
    SizeChanged="Window_SizeChanged"
    StateChanged="Window_StateChanged"
```

Windows load event & close events

```
private void Window_Loaded(object sender, RoutedEventArgs e) {
     Debug.WriteLine("Window loaded and ready!");
}

private void Window_Closing(object sender, CancelEventArgs e) {
    Debug.WriteLine("Window is closing!"); // e.Cancel = true;
}

private void Window_Closed(object sender, EventArgs e) {
    Debug.WriteLine("Window closed!");
}
```

Detecting switching between windows

```
private void Window_Activated(object sender, EventArgs e) {
     Debug.WriteLine("Window activated! IsActive is " + IsActive);
}

private void Window_Deactivated(object sender, EventArgs e) {
    Debug.WriteLine("Window deactivated! IsActive is " + IsActive);
}
```

Detecting changes to a window

```
private void Window_LocationChanged(object sender, EventArgs e) {
    Debug.WriteLine(string.Format("Location changed to {0},{1}!", Left, Top));
}

private void Window_SizeChanged(object sender, SizeChangedEventArgs e) {
    Debug.WriteLine(string.Format(
        "Window size changed to {0},{1}!", Width, Height));
}

private void Window_StateChanged(object sender, EventArgs e) {
    Debug.WriteLine("Window state changed to " + WindowState + "!");
}
```

2.2 User-Interface Services

The main window not only functions to display a main GUI but is useful to provide UI-related services to the rest of the application. Let us add some basic UI services to the application. For example it is common for applications to show messages to the user to inform them when operations succeed or failed. We also may need a user to confirm before performing critical operations. Following is methods for providing these features from the main window. Use **MessageBox** class for displaying message or to prompt the user for confirmation. Sometimes we also need to allow the user to select a file to open or to save to. We can simplify this by providing general methods to use both the **OpenFileDialog** and **SaveFileDialog** to obtain the file location.

Implementing a main window singleton

```
private static Shell _instance;

publis static Shell Instance {
    get {
        return _instance;
    }
}

public Shell() {
    _instance = this;
    InitializeComponent();
}
```

Methods for simple interaction with the user

```
public static void Success(string message) {
    MessageBox.Show(_instance, message, "Information",
        MessageBoxButton.OK, MessageBoxImage.Information);
}
public static void Failure(string message) {
    MessageBox.Show(_instance, message, "Error",
        MessageBoxButton.OK, MessageBoxImage.Error);
}
public static bool Confirm(string message) {
    return MessageBox.Show(_instance, message, "Confirmation",
        MessageBoxButton.OKCancel, MessageBoxImage.Warning)
        == MessageBoxResult.OK;
}
```

Methods to obtain filename to open or save

```
public static string GetOpenFileName(
    string title, string filter, string filename) {
    var dialog = new OpenFileDialog(); dialog.Title = title;
    dialog.Filter = filter; dialog.FileName = filename;
    return dialog.ShowDialog() != true? null : dialog.FileName;
}

public static string GetSaveFileName(
    string title, string filter, string filename) {
    var dialog = new SaveFileDialog(); dialog.Title = title;
    dialog.Filter = filter; dialog.FileName = filename;
    return dialog.ShowDialog() != true? null : dialog.FileName;
}
```

2.3 Window Contents

A window is a **ContentControl** which has a **Content** property. It can be assigned any object but only one object. To place multiple UI elements in the window you can use a **Panel** or **ItemsControl**. A **Grid** is the default panel used in Visual Studio. We will use a **DockPanel** containing UI elements commonly used in a real-life application such as menu bar, toolbar and status bar.

The window layout: Shell.xaml

```
<DockPanel x:Name="shellLayout">
    <Menu x:Name="mnuMain" DockPanel.Dock="Top"></Menu>
    <ToolBar x:Name="tbrMain" DockPanel.Dock="Top"></ToolBar>
    <StatusBar x:Name="sbrMain" DockPanel.Dock="Bottom"></StatusBar>
    <Grid x:Name="shellWorkspace"></Grid>
</DockPanel>
```

There can be more than one menu bar, toolbar or status bar. If you do have multiple toolbars, a **ToolBarTray** can then be used as the parent so that they can be resized and moved around. This is not required for our application as we only need one.

Example using ToolBarTray for multiple toolbars

```
<ToolBarTray DockPanel.Dock="Top">
    <ToolBar x:Name="tbrMain"></ToolBar>
    <ToolBar x:Name="tbrEdit"></ToolBar>
</ToolBarTray>
```

While we are not ready to implement the menu bar and toolbar, we can add elements into the status bar to display messages and basic information to the user. **StatusBar** is an **ItemsControl** which has an **Items** collection to add one or more objects. You can use a separator to separate items.

Adding labels and separators in the status bar

```
<StatusBar x:Name="sbrMain" DockPanel.Dock="Bottom">
    <Label x:Name="lblUserName" />
    <Separator Margin="4,8,4,8" />
    <Label x:Name="lblStatus" Content="Ready." />
</StatusBar>
```

The user name is assigned from our application class to the label during the window loaded event. You can also add a property for the status label to allow the status to be changed at any time. Note that **Label** is a **ContentControl** so it can be assigned any object and not just a string.

Displaying the user name from application: Shell.xaml.cs

```
private void Window_Loaded(object sender, RoutedEventArgs e) {
    lblUserName.Content = MyApp.UserName;
}
```

Property to display status in the shell window

```
public static object Status {
    get { return lblStatus.Content; }
    set { lblStatus.Content = value ?? "Ready."; }
}
```

All **ItemsControl** has an **Items** property for you to assign one more objects. While it is possible to add any object there are also special item classes implemented like for example **MenuItem** for **Menu** and **TabItem** for **TabControl**. Menu represents only a menu bar and each menu, sub-menu and items can be created using the **MenuItem** class. You can use **Header** to assign a header and **Icon** property to optionally assign any an icon. Both of these properties can be assigned any object not only text. If you assign a header use underscore to create an accelerator key to select the item. Use a **Separator** to separate menu items.

Creating menus on the menu bar: SymBank\Shell.xaml

```
<Menu x:Name="mnuMain" DockPanel.Dock="Top">
    <MenuItem Header="_Accounts"></MenuItem>
    <MenuItem Header="_Transactions"></MenuItem>
    <MenuItem Header="Too_ls"></MenuItem>
</Menu>
```

Items in the Accounts menu

```
<MenuItem x:Name="mnuAddNewAccount"
    Click="mnuAddNewAccount_Click"
    Header="_Add New Account">
    <MenuItem.Icon>
        <Image Stretch="None" Source="{StaticResource UserAddImage}" />
    </MenuItem.Icon>
</MenuItem>
<MenuItem x:Name="mnuSearchForAccounts"
    Click="mnuSearchForAccounts_Click"
    Header="_Search for Accounts">
    <MenuItem.Icon>
        <Image Stretch="None" Source="{StaticResource SearchImage}" />
    </MenuItem.Icon>
</MenuItem><Separator />
<MenuItem x:Name="mnuExit"
    Click="mnuExit_Click"
        Header="E_xit">
    <MenuItem.Icon>
        <Image Stretch="None" Source="{StaticResource CancelImage}" />
    </MenuItem.Icon>
</MenuItem>
```

Items in the Transactions menu

```
<MenuItem x:Name="mnuAddTransaction"
    Click="mnuAddTransaction_Click"
    Header="_Add Transaction">
    <MenuItem.Icon>
        <Image Stretch="None" Source="{StaticResource MoneyImage}" />
    </MenuItem.Icon>
</MenuItem>
<MenuItem x:Name="mnuViewAccountTransactions"
    Click="mnuViewAccountTransactions_Click"
    Header="_View Account Transactions">
    <MenuItem.Icon>
        <Image Stretch="None" Source="{StaticResource TableImage}" />
    </MenuItem.Icon>
</MenuItem>
```

Items in the Tools menu

```
<MenuItem x:Name="mnuTextEditor"
    Click="mnuTextEditor_Click"
    Header="_Text Editor">
    <MenuItem.Icon>
        <Image Stretch="None"
            Source="{StaticResource NotepadImage}" />
    </MenuItem.Icon>
</MenuItem>
```

The only event handler we can implement now is for the **Exit** menu item. You can call the **Shutdown** method in the application object and pass in an optional exit code but since the shutdown mode is **OnMainWindowClose** you can also call **Close** method to do the same operation but not able to customize the exit code.

Event handler for the Exit menu item

```
private void mnuExit_Click(object sender, RoutedEventArgs e) {
    Close();    // MyApp.Instance.Shutdown(0);
}
```

Unlike menus you can basically put any UI element you wish on a **ToolBar** where the most common is **Button**. Since there is not much space on this control images should be used for the content of the buttons rather than text. Since buttons are just a quick way to access options from a menu the buttons should share the same images as well as event handlers. Use **ToolTip** to assign a description to each button. Note that the tooltip is a **ContentControl** so can be assigned any object not just text. Button **Click** event can use back the same event handlers for the menu items.

Buttons in the toolbar

```
<Button ToolTip="Add a new account"
    Click="mnuAddNewAccount_Click">
    <Image Stretch="None" Source="{StaticResource UserAddImage}" />
</Button>
<Button ToolTip="Search for accounts"
    Click="mnuSearchForAccounts_Click">
    <Image Stretch="None" Source="{StaticResource SearchImage}" />
</Button><Separator />
<Button ToolTip="Add transaction"
    Click="mnuAddTransaction_Click">
    <Image Stretch="None" Source="{StaticResource MoneyImage}" />
</Button>
<Button ToolTip="View account transactions"
    Click="mnuViewAccountTransactions_Click">
    <Image Stretch="None" Source="{StaticResource TableImage}" />
</Button><Separator />
<Button ToolTip="Open text editor"
    Click="mnuTextEditor_Click">
    <Image Stretch="None" Source="{StaticResource NotepadImage}" />
</Button>
```

3 Regions & Views

3.1 Regions

Since **ItemsControl** can present multiple visual items and allow item selection they are good candidates to be used as region controls in a MVC/MVVM application. Space in an UI can be separated into regions and sub-regions where each region can display one or more views. In our example application we will break the remaining space into two separate regions; side and main. **ListBox** will be used for the side region and **TabControl** for the main region. The **ListBox** would allow you to see all the views at the same time and a user can scroll through them. The **TabControl** would only allow you to see one view at one time and you can switch between the views. To allow the user to control the size of each region, a **GridSplitter** can be added between regions. Following shows elements to construct the regions in the main window.

Dividing window space for regions using Grid: Shell.xaml.cs

```
<Grid x:Name="shellWorkspace">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <ListBox x:Name="sideRegion"
        Grid.Column="0" Visibility="Collapsed" />
    <GridSplitter Grid.Column="1"
        ResizeBehavior="PreviousAndNext"
        ResizeDirection="Columns"
        Width="3" />
    <TabControl x:Name="mainRegion"
        Grid.Column="2" Visibility="Collapsed" />
</Grid>
```

The next thing is to identify each region using a string or enumeration value. Create a **Views** folder in the project and add the following type. If you add more regions in the future, just add additional items to the enumeration.

An enumeration to identity regions: SymBank\Views\Region.cs

```
public enum Region {
    SideRegion,
    MainRegion
}
```

3.2 Views

A view is basically a UI component that user can interact with to complete a single-task. If what it takes for a user to do a task is to click on a button then a view can be just a button. However we will need to a flexible UI component that can be used to implement any UI regardless of how simple or complex they can be. The best element to use would be a **UserControl** especially as Visual Studio provides visual designer support for this type of element. However, you cannot just use a basic UserControl directly as it does not have all the features that you may need for your views. Extend a view class from the UserControl and then add in additional properties and methods that you require as shown below.

Creating a base view class: Views\BaseView.cs

```
using System;
using System.Windows;
using System.Windows.Controls;
namespace SymBank.Views {
    public class BaseView : UserControl {
    }
}
```

Some regions like **TabControl** can show headers for each item you add to it. A basic UserControl does not have a **Header** property but we can add it into our view class. To support full features of WPF you can implement dependency properties for header and the region. Only dependency properties can support automatic UI refresh when the property is changed and can be animated.

Declaring & registering properties with WPF

```
public static readonly DependencyProperty HeaderProperty;
public static readonly DependencyProperty RegionProperty;

static BaseView() {
    HeaderProperty = DependencyProperty.Register("Header",
        typeof(object), typeof(BaseView),new PropertyMetadata(null));
    RegionProperty = DependencyProperty.Register("Region",
        typeof(Region), typeof(BaseView),
        new PropertyMetadata(Region.MainRegion));
}
```

Exposing through standard CLR properties

```
public object Header {
    get { return GetValue(HeaderProperty); }
    set { SetValue(HeaderProperty, value); }
}
public Region Region {
    get { return (Region)GetValue(RegionProperty); }
    set { SetValue(RegionProperty, value); }
}
```

Even though a feature is standard does not mean implementation is also standard. It is possible to define virtual or abstract methods where called but the actual code is only implemented by extended classes. Following we attach a **Loaded** event handler to automatically focus whenever the view is loaded. However since every view have different display and input controls, focus has to be customized in each view.

Providing custom implemented features on a view

```
public BaseView() {
    Loaded += BaseView_Loaded;
}

private void BaseView_Loaded(object sender, RoutedEventArgs e) {
    ResetFocus();
}

public virtual void ResetFocus() {
//  to be implemented by each view that requires focus
}
```

3.3 View Hosts

Some **ItemsControl** require each item to be a specific type to fully utilize all features provided by the control. For example **MenuItem** is the standard item type for **Menu** and **TabItem** for **TabControl**. Since a **UserControl** is neither of this, it is unsuitable to directly add views into such item controls. Instead we place a view into a view host and then add the host to the region. The view host can be specific type like TabItem if the region is a TabControl. **ListBox** does not require any special item type so you choose whichever type you want. We will use **Expander** since it can display headers and the content can be expanded or collapsed. Since view hosts are created in code, we also need to style them in code. Add the following styles as resources. We need to fetch the styles at runtime. We also need one dictionary per region to associate views with view hosts so we know which host is for which view.

Fields to maintain information for view hosts: Shell.xaml.cs

```
private Dictionary<BaseView, Expander> _sideRegionViews;
private Dictionary<BaseView, TabItem > _mainRegionViews;

public Shell() {
    _instance = this;
    InitializeComponent();
    _sideRegionViews = new Dictionary<BaseView, Expander>();
    _mainRegionViews = new Dictionary<BaseView, TabItem >();
}
```

You can now implement methods to add and remove views. These methods can use the **Region** property to determine which region and what type of host will be needed for that region and then perform the necessary operation to get the view to appear in the region.

Methods to add views to regions

```
public void Attach(BaseView view) {
    switch (view.Region) {
        case Region.SideRegion:
            var host1 = new Expander();
            _sideRegionViews.Add(view, host1);
            host1.Header = view.Header;
            host1.Content = view;
            sideRegion.Items.Add(host1);
            sideRegion.SelectedItem = host1;
            if (!sideRegion.IsVisible)
                sideRegion.Visibility = Visibility.Visible;
            break;
        case Region.MainRegion:
            var host2 = new TabItem();
            _mainRegionViews.Add(view, host2);
            host2.Header = view.Header;
            host2.Content = view;
            mainRegion.Items.Add(host2);
            mainRegion.SelectedItem = host2;
            if (!mainRegion.IsVisible)
                mainRegion.Visibility = Visibility.Visible;
            break;
    }
}
```

Methods to remove views from regions

```
public void Detach(BaseView view) {
    switch (view.Region) {
        case Region.SideRegion:
            var host1 = _sideRegionViews[view];
            sideRegion.Items.Remove(host1);
            _sideRegionViews.Remove(view);
            if (_sideRegionViews.Count == 0)
                sideRegion.Visibility = Visibility.Collapsed;
            break;
        case Region.MainRegion:
            var host2 = _mainRegionViews[view];
            mainRegion.Items.Remove(host2);
            _mainRegionViews.Remove(view);
            if (_mainRegionViews.Count == 0)
                mainRegion.Visibility = Visibility.Collapsed;
            break;
    }
}
```

Since all views can easily access the shell, we can simplify adding and removing views by adding methods to the view to add or remove themselves from the shell as shown below.

Using the shell from the view: Views\BaseView.cs

```
public void Show() { Shell.Instance.Attach(this); }
public void Close() { Shell.Instance.Detach(this); }
```

Let us implement a sample view. To create a new view, first add a **UserControl** into the **Views** folder. Register a **xmlns** prefix for **SymBank.Views** namespace. You can now replace the base **UserControl** class with **BaseView** and you should now be able to set **Header** and **Region** properties. Ensure the base class of the code-behind class is also changed to **BaseView**.

Replacing UserControl with BaseView: Views\CalendarView.xaml

```
<v:BaseView x:Class="SymBank.Views.CalendarView"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:v="clr-namespace:SymBank.Views"
    mc:Ignorable="d" d:DesignHeight="300" d:DesignWidth="300"
    Header="Calendar" Region="SideRegion">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto" />
        </Grid.ColumnDefinitions>
        <Calendar x:Name="calendar" Margin="16" />
    </Grid>
</v:BaseView>
```

Extend class from BaseView instead: Views\CalendarView.xaml.cs

```
namespace SymBank.Views {
    public partial class CalendarView : BaseView {
        public CalendarView() {
            InitializeComponent();
        }
    }
}
```

For fixed views you can instantiate them directly when the main window is loaded. For optional views you can add menu items for user to activate them manually. Following shows the code to instantiate and show the above view.

Creating fixed views when the shell is loaded

```
private void Window_Loaded(object sender, RoutedEventArgs e) {
    lblUserName.Content = MyApp.UserName;
    new CalendarView().Show();
}
```

3.4 RoutedCommands

Commands are objects that contains code that can be executed automatically through menu items and buttons. RouteCommands are commands that are to be processed by UI elements. There is already a built-in set of commands that you bind to menu items and buttons. In the following example we will add in a view named **TextEditorView** with the following content and code. There is already a menu item to open this view, you just need to add the code to instantiate and show the view.

View to edit a flow document: Views\TextEditorView.xaml

```
<v:BaseView x:Class="SymBank.Views.TextEditorView"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:v="clr-namespace:SymBank.Views"
    mc:Ignorable="d" d:DesignHeight="300" d:DesignWidth="400"
    Header="Untitled" Region="MainRegion">
    <DockPanel>
        <ToolBar DockPanel.Dock="Top">
            <Button x:Name="btnClose" Click="btnClose_Click">
                <Image Stretch="None" Source="{StaticResource CancelImage}" />
            </Button>
        </ToolBar>
        <RichTextBox x:Name="txtDocument" />
    </DockPanel>
</v:BaseView>
```

Implementation in code-behind class: Views\TextEditorView.xaml.cs

```
using System.Windows;

namespace SymBank.Views {
    public partial class TextEditorView : BaseView {
        public TextEditorView() {
            InitializeComponent();
        }
        public override void ResetFocus() { txtDocument.Focus(); }
        private void btnClose_Click(object sender, RoutedEventArgs e) {
            Close();
        }
    }
}
```

Event handler to open the view: Shell.xaml.cs

```
private void mnuTextEditor_Click(object sender, RoutedEventArgs e) {
        new TextEditorView().Show();
}
```

We will now add buttons to allow the user to perform disk and edit operations on the document in **RichTextBox** control. Rather than using event handlers we will assign to standard commands in **ApplicationCommands** and **EditingCommands** classes.

Assigning standard commands: Views\TextEditorView.xaml

```
<Button Command="ApplicationCommands.Open">
    <Image Stretch="None" Source="{StaticResource OpenImage}" />
</Button>
<Button Command="ApplicationCommands.Save">
    <Image Stretch="None" Source="{StaticResource SaveImage}" />
</Button>
<Separator />
<Button Command="EditingCommands.AlignLeft">
    <Image Stretch="None" Source="{StaticResource AlignLeftImage}" />
</Button>
<Button Command="EditingCommands.AlignCenter">
    <Image Stretch="None" Source="{StaticResource AlignCenterImage}" />
</Button>
<Button Command="EditingCommands.AlignRight">
    <Image Stretch="None" Source="{StaticResource AlignRightImage}" />
</Button>
<Separator />
<Button Command="EditingCommands.ToggleBold">
    <Image Stretch="None" Source="{StaticResource TextBoldImage}" />
</Button>
<Button Command="EditingCommands.ToggleItalic">
    <Image Stretch="None" Source="{StaticResource TextItalicImage}" />
</Button>
<Button Command="EditingCommands.ToggleUnderline">
    <Image Stretch="None" Source="{StaticResource TextUnderlineImage}" />
</Button>
<Separator />
<Button Command="ApplicationCommands.Cut">
    <Image Stretch="None" Source="{StaticResource CutImage}" />
</Button>
<Button Command="ApplicationCommands.Paste">
    <Image Stretch="None" Source="{StaticResource PasteImage}" />
</Button>
<Separator />
```

Commands are by default passed to the control in focus unless you explicitly set the **CommandTarget** property. Since **RichTextBox** is in focus, it becomes the default command target. Even though the control receives a command does not mean that it will process it. The control developer can decide which commands that they support by adding bindings to **CommandBindings** property. Run application and test which commands that the above control supports and which ones it does not.

3.5 CommandBindings

So you may have discovered by now RichTextBox does not support **Open** and **Save** commands so your view will have to support them. Since they are RoutedCommands, the commands will be routed upwards and you can choose to process them at any level until the **Window**. For our example, we will accept them at the **BaseView** level and add CommandBindings to process both of the commands. The view should then be completed.

Adding CommandBindings to process RoutedCommands

```
<v:BaseView.CommandBindings>
    <CommandBinding x:Name="cmdOpen"
        Command="ApplicationCommands.Open"
        CanExecute="cmdOpen_CanExecute"
        Executed="cmdOpen_Executed" />
    <CommandBinding x:Name="cmdSave"
        Command="ApplicationCommands.Save"
        CanExecute="cmdSave_CanExecute"
        Executed="cmdSave_Executed" />
</v:BaseView.CommandBindings>
```

Both commands should always be executable: TextEditorView.xaml.cs

```
private void cmdOpen_CanExecute(object sender, CanExecuteRoutedEventArgs e) {
    e.CanExecute = true;
}

private void cmdSave_CanExecute(object sender, CanExecuteRoutedEventArgs e) {
    e.CanExecute = true;
}
```

Code to execute for Save command

```
private void cmdSave_Executed(object sender, ExecutedRoutedEventArgs e) {
    var filename = Shell.GetSaveFileName(
        "Save Document", "Documents|*.xaml", (string)Header);
    if (filename == null) return;
    try {
        var stream = File.Create(filename);
        XamlWriter.Save(txtDocument.Document, stream);
        stream.Close();
        Header = System.IO.Path.GetFileName(filename);
        Shell.Status = "Document saved.";
    }
    catch (Exception ex) {
        Shell.Status = "Error saving document.";
        Shell.Failure("Cannot save document.\n" + ex.Message);
    }
}
```

Code to execute for Open command

```
private void cmdOpen_CanExecute(object sender,
    System.Windows.Input.CanExecuteRoutedEventArgs e) {
    var filename = Shell.GetOpenFileName(
        "Open Document", "Documents|*.xaml", (string)Header);
    if (filename == null) return;
    try {
        var stream = File.OpenRead(filename);
        txtDocument.Document = (FlowDocument)XamlReader.Load(stream);
        stream.Close();
        Header = System.IO.Path.GetFileName(filename);
        Shell.Status = "Document opened.";
    }
    catch (Exception ex) {
        Shell.Status = "Error opening document.";
        Shell.Failure("Cannot open document.\n" + ex.Message);
    }
}
```

3.6 Data Bindings

You may have noticed in the *Open* and *Save* command event handlers that we change the **Header** of the view to display the latest filename. However you will notice that the **Expander** and **TabItem** header does not change. This is since we only copy the **Header** of the view when hosts were created and there are no physical link between the properties of the view host and the view. The following shows how to create and set bindings in code.

Bind instead of copying the view Header: Shell.xaml.cs

```
//  host1.Header = view.Header;
host1.SetBinding(Expander.HeaderProperty,
    new Binding("Header") { Source = view });
                :
// host2.Header = view.Header;
host2.SetBinding(Expander.HeaderProperty,
    new Binding("Header") { Source = view });
```

In this module we have implemented an application where some parts are static and other parts are dynamic. The static parts can be hardcoded into XAML documents and dynamic parts have to be implemented using code.