Microsoft

Programming Windows®

Sixth Edition

Writing Windows 8 Apps with C# and XAML

Professional

Charles Petzold

Microsoft

Programming Windows®, Sixth Edition

Charles Petzold

```
    // Why aren't these set in the generated C# files?
    editBox = splitContainer.Child1 as TabbableTextBox;
    resultContainer = splitContainer.Child2 as RulerContainer;

    // Set a fixed-pitch font for the TextBox
    Language language =
        new Language(Windows.Globalization.Language.CurrentInputLanguageTag);
    LanguageFontGroup languageFontGroup = new LanguageFontGroup(language.LanguageTag);
    LanguageFont languageFont = languageFontGroup.FixedWidthTextFont;
    editBox.FontFamily = new FontFamily(languageFont.FontFamily);

    Loaded += OnLoaded;
    Application.Current.Suspending += OnApplicationSuspending;
}

async void OnLoaded(object sender, RoutedEventArgs args)
{
    // Load AppSettings and set to DataContext
    appSettings = new AppSettings();
    this.DataContext = appSettings;

    // Load any file that may have been saved
    StorageFolder localFolder = ApplicationData.Current.LocalFolder;
    StorageFile storageFile = await localFolder.CreateFileAsync("XamlCruncher.xaml",
                                            CreationCollisionOption.OpenIfExists);
    editBox.Text = await FileIO.ReadTextAsync(storageFile);

    if (editBox.Text.Length == 0)
        await SetDefaultXamlFile();

    // Other initialization
    ParseText();
    editBox.Focus(FocusState.Programmatic);
    DisplayLineAndColumn();
    ...
}

async void OnApplicationSuspending(object sender, SuspendingEventArgs args)
{
    // Save application settings
    appSettings.Save();

    // Save text content
    SuspendingDeferral deferral = args.SuspendingOperation.GetDeferral();
    await PathIO.WriteTextAsync("ms-appdata:///local/XamlCruncher.xaml", editBox.Text);
    deferral.Complete();
}

async Task SetDefaultXamlFile()
{
    editBox.Text =
        "<Page xmlns=\"http://schemas.microsoft.com/winfx/2006/xaml/presentation\"\r\n" +
        "      xmlns:x=\"http://schemas.microsoft.com/winfx/2006/xaml\">\r\n\r\n" +
        "    <TextBlock Text=\"Hello, Windows 8!\"\r\n" +
        "               FontSize=\"48\" />\r\n\r\n" +
        "</Page>";
```

```
            editBox.IsModified = false;
            loadedStorageFile = null;
            filenameText.Text = "";
        }
        ...
        void OnEditBoxSelectionChanged(object sender, RoutedEventArgs args)
        {
            DisplayLineAndColumn();
        }

        void DisplayLineAndColumn()
        {
            int line, col;
            editBox.GetPositionFromIndex(editBox.SelectionStart, out line, out col);
            lineColText.Text = String.Format("Line {0} Col {1}", line + 1, col + 1);

            if (editBox.SelectionLength > 0)
            {
                editBox.GetPositionFromIndex(editBox.SelectionStart + editBox.SelectionLength - 1,
                                             out line, out col);
                lineColText.Text += String.Format(" - Line {0} Col {1}", line + 1, col + 1);
            }
        }
        ...
    }
```

The constructor begins by fixing a little bug involving the *editBox* and *resultContainer* fields. The XAML parser definitely creates these fields during compilation, but they are not set by the *InitializeComponent* call at run time.

The remainder of the constructor sets a fixed-pitch font in the *TabbableTextBox* based on the predefined fonts available from the *LanguageFontGroup* class. This is apparently the only way to get actual font family names from the Windows Runtime. (In Chapter 15, "Going Native," I demonstrate how to use DirectWrite to obtain the collection of fonts installed on the system.)

The remaining initialization occurs in the *Loaded* event handler. The *DataContext* of the page is set to the *AppSettings* instance, as you probably anticipated from the data bindings in the MainPage.xaml file.

The *OnLoaded* method continues by loading a previously saved file or (if it doesn't exist) setting a default piece of XAML in the *TabbableTextBox* and calling *ParseText* to parse it. (You'll see how this works soon.) The *TabbableTextBox* is assigned keyboard input focus, and *OnLoaded* concludes by displaying the initial line and column, which is then updated whenever the *TextBox* selection changes.

You might wonder why *SetDefaultXamlFile* is defined as *async* and returns *Task* when it does not actually contain any asynchronous code. You'll see later that this method is used as an argument to another method in the file I/O logic, and that's the sole reason I had to define it oddly. The compiler generates a warning message because it doesn't contain any *await* logic.

Parsing the XAML

The major job of XamlCruncher is to pass a piece of XAML to *XamlReader.Load* and get out an object. A property of the *AppSettings* class named *AutoParsing* allows this to happen with every keystroke, or the program waits until you press the Refresh button on the application bar.

If *XamlReader.Load* encounters an error, it raises an exception, and the program then displays that error in red in the status bar at the bottom of the page and also colors the text in the *TabbableTextBox* red.

Project: XamlCruncher | **File:** MainPage.xaml.cs (excerpt)

```
public sealed partial class MainPage : Page
{
    Brush textBlockBrush, textBoxBrush, errorBrush;
    ...
    public MainPage()
    {
        ...
        // Set brushes
        textBlockBrush = Resources["ApplicationForegroundThemeBrush"] as SolidColorBrush;
        textBoxBrush = Resources["TextBoxForegroundThemeBrush"] as SolidColorBrush;
        errorBrush = new SolidColorBrush(Colors.Red);
        ...
    }

    ...

    void OnRefreshAppBarButtonClick(object sender, RoutedEventArgs args)
    {
        ParseText();
        this.BottomAppBar.IsOpen = false;
    }
    ...
    void OnEditBoxTextChanged(object sender, RoutedEventArgs e)
    {
        if (appSettings.AutoParsing)
            ParseText();
    }

    void ParseText()
    {
        object result = null;

        try
        {
            result = XamlReader.Load(editBox.Text);
        }
        catch (Exception exc)
        {
            SetErrorText(exc.Message);
            return;
        }

        if (result == null)
        {
            SetErrorText("Null result");
        }
        else if (!(result is UIElement))
        {
            SetErrorText("Result is " + result.GetType().Name);
        }
        else
        {
             resultContainer.Child = result as UIElement;
             SetOkText();
             return;
        }
    }

    void SetErrorText(string text)
    {
        SetStatusText(text, errorBrush, errorBrush);
    }

    void SetOkText()
    {
        SetStatusText("OK", textBlockBrush, textBoxBrush);
    }

    void SetStatusText(string text, Brush statusBrush, Brush editBrush)
    {
        statusText.Text = text;
        statusText.Foreground = statusBrush;
        editBox.Foreground = editBrush;
    }
}
```

It could be that a chunk of XAML successfully passes *XamlReader.Load* with no errors but then raises an exception later on. This can happen particularly when XAML animations are involved because the animation doesn't start up until the visual tree is loaded.

The only real solution is to install a handler for the *UnhandledException* event defined by the *Application* object, and that's done in the conclusion of the *Loaded* handler:

Project: XamlCruncher | **File:** MainPage.xaml.cs (excerpt)

```
async void OnLoaded(object sender, RoutedEventArgs args)
{
    ...
    Application.Current.UnhandledException += (excSender, excArgs) =>
        {
            SetErrorText(excArgs.Message);
            excArgs.Handled = true;
        };
}
```

The problem with something like this is that you want to make sure that the program isn't going to have some other kind of unhandled exception that isn't a result of some errant code.

Also, when Visual Studio is running a program in its debugger, it wants to snag the unhandled exceptions so that it can report them to you. Use the Exceptions dialog from the Debug menu to indicate which exceptions you want Visual Studio to intercept and which should be left to the program.

XAML Files In and Out

Whenever I approach the code involved in loading and saving documents, I always think it's going to be easier than it turns out to be. Here's the basic problem: Whenever a New or Open command occurs, you need to check if the current document has been modified without being saved. If that's the case, a message box should be displayed asking whether the user wants to save the file. The options are Save, Don't Save, and Cancel.

The easy answer is Cancel. The program doesn't need to do anything further. If the user selects the Don't Save option, the current document can be abandoned and the New or Open command can proceed.

If the user answers Save, the existing document needs to be saved under its file name. But that file name might not exist if the document wasn't loaded from a disk file or previously saved. At that point, the Save As dialog box needs to be displayed. But the user can select Cancel from that dialog box as well, and the New or Open operation ends. Otherwise, the existing file is first saved.

Let's first look at the methods involved in saving documents. The application button has Save and Save As buttons, but the Save button needs to invoke the Save As dialog box if it doesn't have a file name for the document:

Project: XamlCruncher | **File:** MainPage.xaml.cs (excerpt)

```
async void OnSaveAsAppBarButtonClick(object sender, RoutedEventArgs args)
{
     StorageFile storageFile = await GetFileFromSavePicker();

     if (storageFile == null)
         return;

     await SaveXamlToFile(storageFile);
}

async void OnSaveAppBarButtonClick(object sender, RoutedEventArgs args)
{
    Button button = sender as Button;
    button.IsEnabled = false;

    if (loadedStorageFile != null)
    {
        await SaveXamlToFile(loadedStorageFile);
    }
    else
    {
        StorageFile storageFile = await GetFileFromSavePicker();

        if (storageFile != null)
        {
            await SaveXamlToFile(storageFile);
        }
    }
    button.IsEnabled = true;
}

async Task<StorageFile> GetFileFromSavePicker()
{
    FileSavePicker picker = new FileSavePicker();
    picker.DefaultFileExtension = ".xaml";
    picker.FileTypeChoices.Add("XAML", new List<string> { ".xaml" });
    picker.SuggestedSaveFile = loadedStorageFile;
    return await picker.PickSaveFileAsync();
}

async Task SaveXamlToFile(StorageFile storageFile)
{
    loadedStorageFile = storageFile;
    string exception = null;

    try
    {
        await FileIO.WriteTextAsync(storageFile, editBox.Text);
    }
    catch (Exception exc)
    {
        exception = exc.Message;
    }

    if (exception != null)
    {
        string message = String.Format("Could not save file {0}: {1}",
                                        storageFile.Name, exception);
        MessageDialog msgdlg = new MessageDialog(message, "XAML Cruncher");
        await msgdlg.ShowAsync();
    }
    else
    {
        editBox.IsModified = false;
        filenameText.Text = storageFile.Path;
    }
}
```

For the Save button, the handler disables the button and then enables it when it's completed. I'm worried that the button might be re-pressed during the time the file is being saved and there might even be a reentrancy problem if the handler tries to save it again when the first save hasn't completed.

In the final method, the *FileIO.WriteTextAsync* call is in a *try* block. If an exception occurs while saving the file, the program wants to use *MessageDialog* to inform the user. But asynchronous methods such as *ShowAsync* can't be called in a *catch* block, so the exception is simply saved for checking afterward.