Object Oriented Programming II

Topics Covered

- Creating a WPF Application
- Planning the User Interface
- Refactoring to Enhance and Simplify
- Maintaining a Responsive UI using Async Methods
- Wildcard Searches using Regex
- Inheritance and Composition
  - *Is a* vs. *Has a*
- OOP
  - Interface Simplicity
  - Immutability / Statefulness
  - Performance
  - Recap

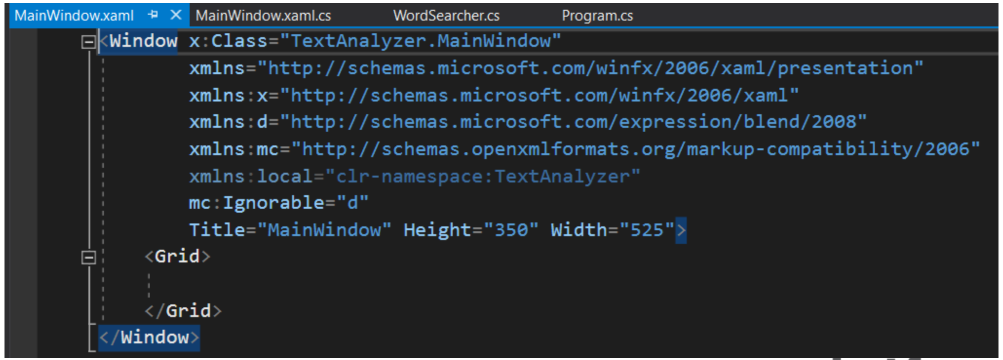
A Windowed Application

To illustrate more of the capabilities of our WordSearcher class, we need a more powerful UI. Let's create a windowed application.

In Solution Explorer, right-click on the solution and select Add → New Project. In the Add New Project Dialog, select "WPF App (.NET Framework)". Name the project "TextAnalyzer".

Visual Studio will create new WPF Application with a single class named MainWindow.

You will see a screen like that on the right. This is an XML-based language called XAML – Extended Application Markup Language. It is used to define our UI.



```
<Window x:Class="TextAnalyzer.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:TextAnalyzer"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525">
    <Grid>

    </Grid>
</Window>
```

Resources: TextAnalysis.zip

This presentation dives into some fairly complicated code, including advanced features of the C# language.

For this course you will not be expected to <u>write</u> C# using all of these features, but you should become familiar with the concepts.

Please download TextAnalysis.zip to follow along with this presentation.

Planning the UI

We need:

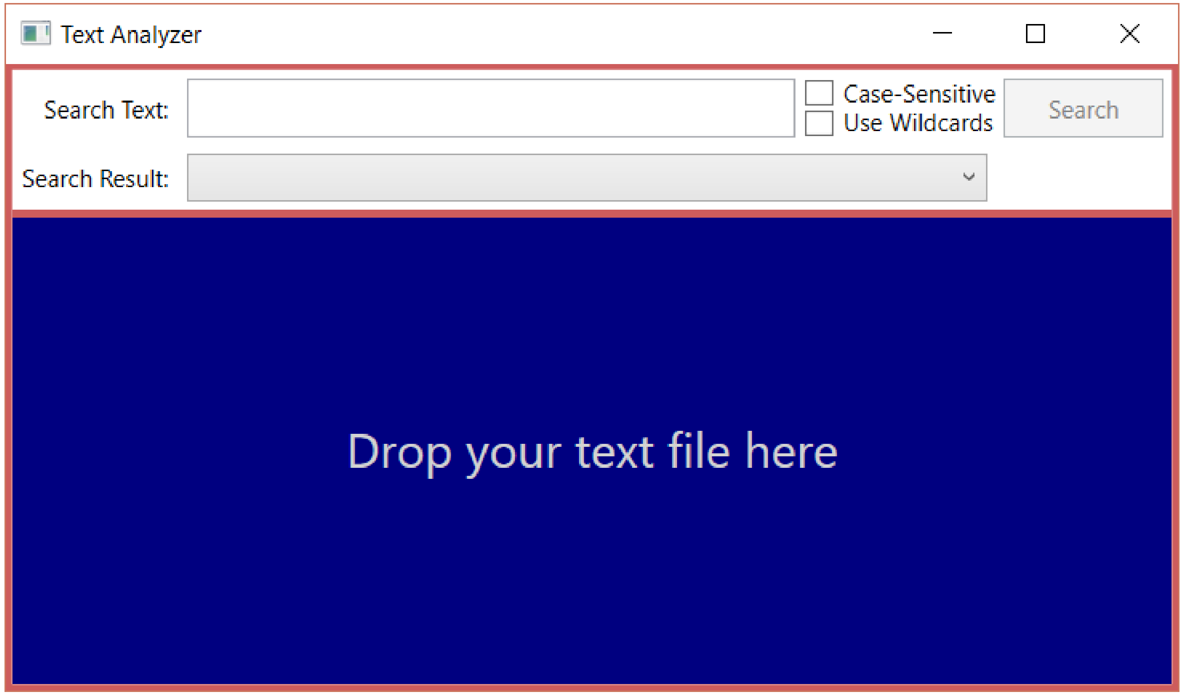
- A TextBox where we can enter search text.
- A Check Box to indicate whether our search is case-sensitive.
- A TextBox where we can display the current document. This will double as an area where we can drag a file to open it.
- A Button to trigger the search.
- A ComboBox to select the current search result.
- Labels to indicate what the different UI components are for.

A complete XAML file meeting these needs is provided. A full discussion of XAML is beyond the scope of this course, but feel free to tinker and explore the provided code.

The TextAnalyzer UI

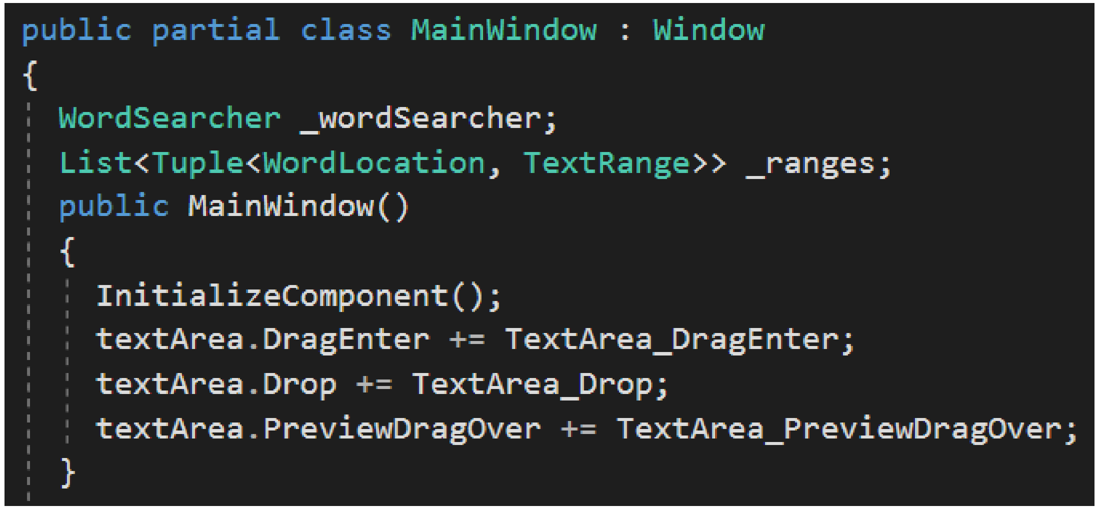
Here’s the idea:

When we search for a word, the ComboBox will be populated with results. When we select a result, the TextBox containing our document will automatically scroll to that location and highlight the found word.



XAML and Code-Behind

Our MainWindow.xaml file defines our user interface, and an associated .cs file, MainWindow.xaml.cs, defines the Window class which composes this interface. Here we can write code to handle events emitted by our UI components.



```
public partial class MainWindow : Window
{
    WordSearcher _wordSearcher;
    List<Tuple<WordLocation, TextRange>> _ranges;
    public MainWindow()
    {
        InitializeComponent();
        textArea.DragEnter += TextArea_DragEnter;
        textArea.Drop += TextArea_Drop;
        textArea.PreviewDragOver += TextArea_PreviewDragOver;
    }
```

This is an example of event-driven software. UI applications are event-driven because they respond to actions initiated by the user.

The details of the UI code are an advanced topic. This presentation will focus on how we use our WordSearcher class.

Using *all* of the Information

Let’s look again at WordSearcher.GetWordCount. During it’s execution, we locate every instance of the requested word, and then the method only returns a summary – the word count. We’ve thrown most of the information away!

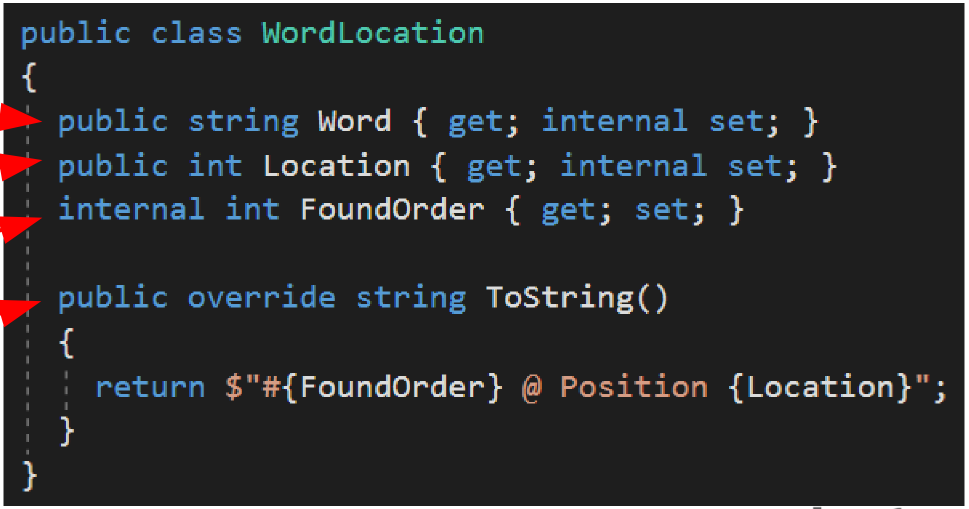
Let’s refactor WordSearcher to let us obtain the location of each word. First, create a new class in the class library named **WordLocation**:

The Word that was searched

The Location in the Document

The order in which it was found

A string representation of this object
This method is inherited from Object.



```
public class WordLocation
{
    public string Word { get; internal set; }
    public int Location { get; internal set; }
    internal int FoundOrder { get; set; }

    public override string ToString()
    {
        return $"#{FoundOrder} @ Position {Location}";
    }
}
```

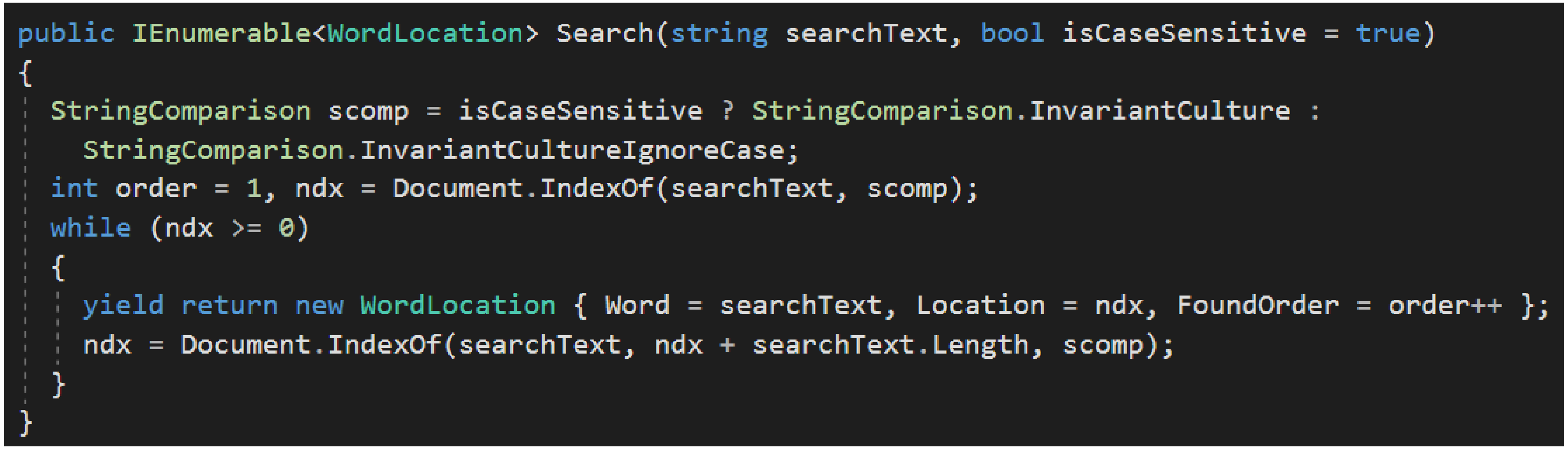
Note the use of the internal access modifier.

Access Modifiers

Name	Description
public	Class/member is visible and accessible everywhere.
private	Class/member is visible and accessible only to the declaring class or nested classes.
internal	Class/member is visible to all classes in the same assembly.
protected	Member is visible only to the current and inheriting classes.
protected internal	Member is visible to all classes in the same assembly and to inheriting classes.

Access modifiers are essential to encapsulation. Access modifiers allow us to hide the inner workings ("implementation details") of the class and expose only what consumers of the class need to know.

The WordSearch.Search Method

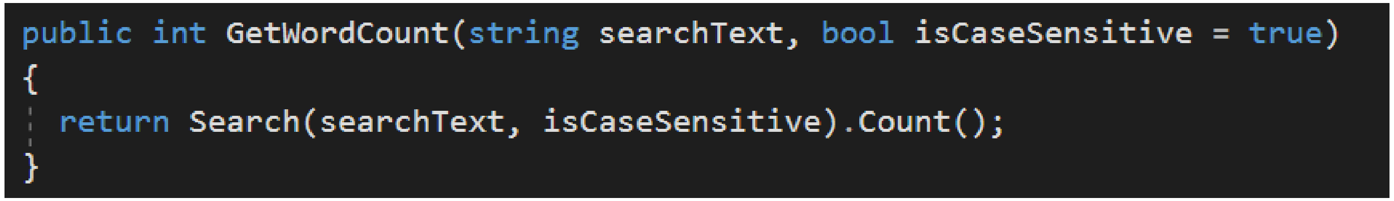


```
public IEnumerable<WordLocation> Search(string searchText, bool isCaseSensitive = true)
{
  StringComparison scomp = isCaseSensitive ? StringComparison.InvariantCulture :
    StringComparison.InvariantCultureIgnoreCase;
  int order = 1, ndx = Document.IndexOf(searchText, scomp);
  while (ndx >= 0)
  {
    yield return new WordLocation { Word = searchText, Location = ndx, FoundOrder = order++ };
    ndx = Document.IndexOf(searchText, ndx + searchText.Length, scomp);
  }
}
```

WordSearcher.Search returns an enumeration of WordLocation objects. It is similar to GetWordCount, but each time we locate a word, we use the yield keyword to return 1 element of an enumeration. This temporarily yields control to a calling foreach loop that processes the value.

Simplify GetWordCount

With our new Search method, we can now simplify GetWordCount – we simply call Search and return the number of elements enumerated:



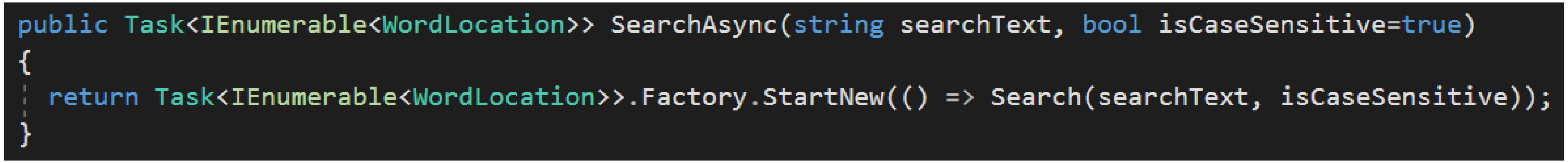
```
public int GetWordCount(string searchText, bool isCaseSensitive = true)
{
    return Search(searchText, isCaseSensitive).Count();
}
```

Refactoring always involves re-arranging code as we have done here. The objective is to maximize the functionality of the class while keeping the code as simple as possible.

Keeping a Responsive UI

Once we start creating windowed applications, it is important that the user interface (UI) remain responsive. For an extremely large document, our Search method might run slowly, and if we called it directly in response to clicking a button, it would freeze the UI.

To work around this, we will employ an asynchronous pattern. A (possibly) slow method such as Search has a companion method named SearchAsync. The code looks like this:

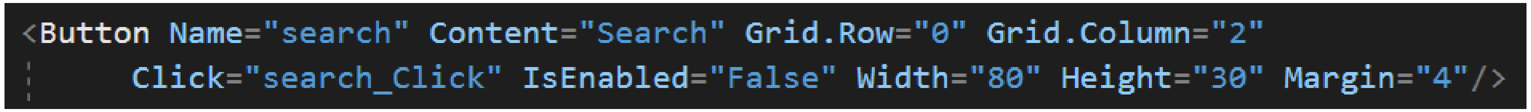


```
public Task<IEnumerable<WordLocation>> SearchAsync(string searchText, bool isCaseSensitive=true)
{
    return Task<IEnumerable<WordLocation>>.Factory.StartNew(() => Search(searchText, isCaseSensitive));
}
```

The Task class allows us to create methods which do not block the UI. We do not have to write any new Search code – we simply reuse what we already have.

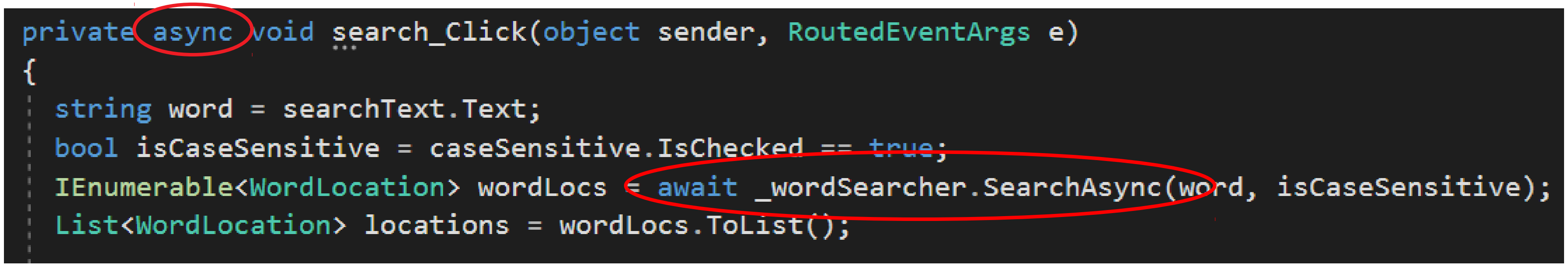
Calling SearchAsync

In the XAML, we added an event handler for the button’s Click event:



```
<Button Name="search" Content="Search" Grid.Row="0" Grid.Column="2"
        Click="search_Click" IsEnabled="False" Width="80" Height="30" Margin="4"/>
```

When we click the button, the handler code runs:



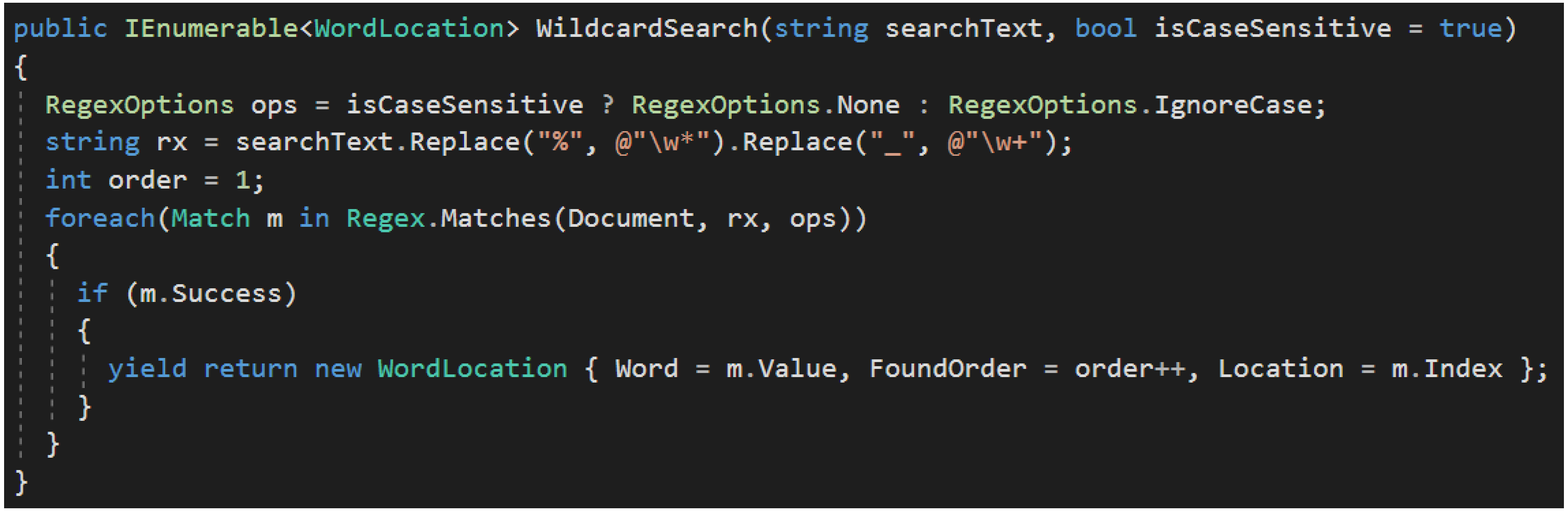
```
private async void search_Click(object sender, RoutedEventArgs e)
{
    string word = searchText.Text;
    bool isCaseSensitive = caseSensitive.IsChecked == true;
    IEnumerable<WordLocation> wordLocs = await _wordSearcher.SearchAsync(word, isCaseSensitive);
    List<WordLocation> locations = wordLocs.ToList();
```

The method is declared with the async keyword. This allows us to use await, which causes the awaited method to run in the background while the UI remains responsive.

Wildcard Searches

Let’s take our WordSearch a step further and implement wildcard searches. Let’s use the same wildcard characters that we use with SQL – % means 0 or more of any character, and _ means one of any character.

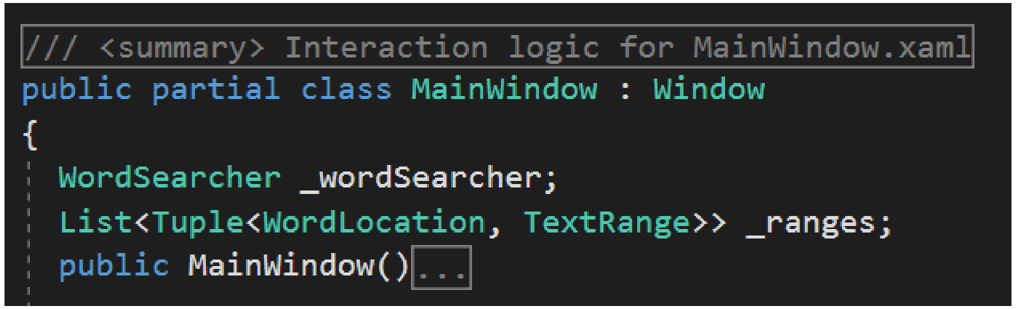
We need to add one more CheckBox to our UI – we can stack it below the Case Sensitive CheckBox. We can implement it using a Regular Expression and the Regex class:



```
public IEnumerable<WordLocation> WildcardSearch(string searchText, bool isCaseSensitive = true)
{
    RegexOptions ops = isCaseSensitive ? RegexOptions.None : RegexOptions.IgnoreCase;
    string rx = searchText.Replace("%", @"\w*").Replace("_", @"\w+");
    int order = 1;
    foreach(Match m in Regex.Matches(Document, rx, ops))
    {
        if (m.Success)
        {
            yield return new WordLocation { Word = m.Value, FoundOrder = order++, Location = m.Index };
        }
    }
}
```

MainWindow: Inheritance and Composition

Let’s look briefly at our MainWindow class.



```
/// <summary> Interaction logic for MainWindow.xaml
public partial class MainWindow : Window
{
    WordSearcher _wordSearcher;
    List<Tuple<WordLocation, TextRange>> _ranges;
    public MainWindow()...
```

The partial keyword indicates that part of the class is defined in another file. For our MainWindow, this file is generated by the compiler from the XAML.

It inherits from Window, which is to say it **is a** Window. Using the XAML, we place controls like our TextBox inside the window, which is to say that MainWindow **has a** TextBox. We further endow the window with search functionality using the fields shown above.

While *inheritance* (**is a**) gets much of the attention, most of the power of OOP comes from *composition* (**has a**), where we compose sophisticated capabilities by bringing diverse objects together.

WordSearcher: Object Oriented Ponderances

WordSearch currently has 7 methods and a property:

- A Constructor
- A Factory Method
- A Read-Only Document Property
- Search / SearchAsync Methods
- WildcardSearch / WildcardSearchAsync Methods
- A GetWordCount Method

The class is *immutable* – there are no methods or properties that actually make changes to the Document. Is this a good thing?

Are we happy with this? Is it well *encapsulated*? Is it easy to use? What happens when we add additional Search methods, like maybe a subsequence matching search?

How about GetWordCount, where we create an enumeration of WordLocation objects and throw them away? In a very large document, this creates memory churn. Can it be simplified?

OOP Characteristics: Recap

Object Oriented Programming promotes:

- **Re-usability**: Well-designed classes can be used and re-used in multiple contexts.
- **Extensibility**: Through inheritance, functionality of base classes can be extended, refined and re-purposed.
- **Polymorphism**: Objects in an inheritance hierarchy can be used based on their common characteristics and behaviors.
- **Encapsulation**: Each class specializes in one task and exposes just enough information to optimize its utility. Exposing too much information (“implementation details) is a distraction and a risk.
- **Composition**: Applications can bring together the capabilities of multiple classes to compose sophisticated functionality.

Topics Covered

- Creating a WPF Application
- Planning the User Interface
- Refactoring to Enhance and Simplify
- Maintaining a Responsive UI using Async Methods
- Wildcard Searches using Regex
- Inheritance and Composition
  - *Is a* vs. *Has a*
- OOP
  - Interface Simplicity
  - Immutability / Statefulness
  - Performance
  - Recap

OOP Exercises

1. Refactor WordSearchOptions to remove the performance concerns around GetWordCount. Hint: use a private method that enumerates only the integer word locations.

2. Create an enum named WordSearchOptions. The enum will have the [Flags] attribute so that it behaves like a bitmask, and enum values will be None, CaseSensitive, and UseWildcards.

3. Refactor WordSearch so that it has just two public Search methods (Search and SearchAsync). Each method accepts the word to search for and a WordSearchOptions value. [Note: You will still use several different private methods].