|  |
| --- |
|  |
| C# Events & Properties |
| Raising & Handling |
|  |
|  |
|  |

# 

# C# Events & Properties

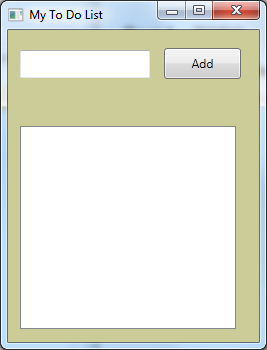
## Objectives

In this lab we’ll build a simple TO DO list using WPF. After completing this lab you should understand how to to do the following:

* Subscribe and react to an event
* Implement automatic properties and properties with backing fields
* Raise an event

## Part 1 – TODO

1. Open the **ToDo.sln** file underneath the \before\Todo directory for this lab.
2. Press **Ctrl+F5** to run the application. Verify there are no errors.



Although the application shell exists, the application does not let you add new To Do items. We’ll implement this functionality in the following steps.

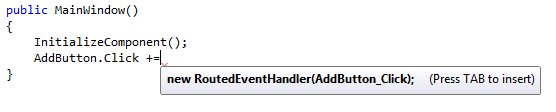
Our first step will be to wire up an event handler to respond when the user clicks the “Add“ button. This button is a part of our MainWindow class – it is assigned to a field named AddButton. You won’t see this field defined in the source code because it is in hidden, generated code.

1. Click next to MainWindow.xaml to expand the node and open the “code-behind” – **MainWindow.xaml.cs**.

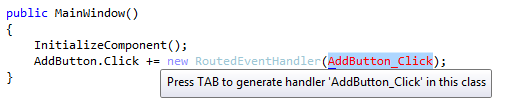


1. Inside the MainWindow constructor, and after the call to InitializeComponent, add an event handler for the AddButton’s **Click** **event**.

You can add the event handler by first typing “AddButton.Click +=”. At this point Visual Studio should prompt you to hit the Tab key to insert the subscription code.



After hitting Tab once, you’ll be prompted to hit Tab again to generate the event handling method.



The end result should look like the following.

public partial class MainWindow : Window  
{  
    public MainWindow()  
    {  
        InitializeComponent();  
        AddButton.Click += new RoutedEventHandler(AddButton\_Click);  
    }  
  
    void AddButton\_Click(object sender, RoutedEventArgs e)  
    {  
        throw new System.**NotImplementedException**();  
    }  
}

Note: it’s more common to wire up events from the XAML files that define WPF windows and controls. However, since we want to take a close look at events we’ll let ourselves wire up this event in the code file.

1. **Remove** the line of code inside the AddButton\_Click method.

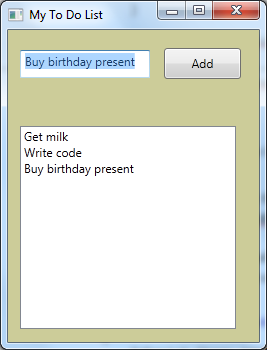
In the next step we’ll work with the other two controls in the application – the AddTextBox (the text box) and the ToDoList (a ListBox that displays the list of todo items).

1. Add a line of code that will take the **Text** property of the **AddTextBox** control and place it in the **Items** collection of the **ToDoList**.

void AddButton\_Click(object sender, RoutedEventArgs e)  
{  
    ToDoList.Items.Add(AddTextBox.Text);  
}

1. Press Ctrl+F5 to **run** the application.

You should be able to add items into the to-do list. Notice that the TextBox control doesn’t clear itself after adding an item – we’ll fix this next.



1. Close the application and return to MainWindow.xaml.cs
2. In the button click event, set the **Text** property of **AddTextBox** to an **empty** **string** after inserting into the list.

void AddButton\_Click(object sender, RoutedEventArgs e)  
{  
    ToDoList.Items.Add(AddTextBox.Text);  
    AddTextBox.Text = "";  
}

1. Run the application again and ensure the new functionality is working properly.

## Part II – Using a ViewModel

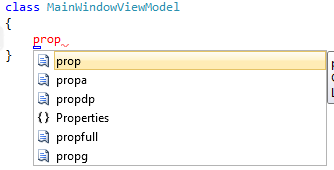
A popular approach to UI programming with WPF is to use a ViewModel. A ViewModel is a plain C# class that is easy to create and test. The view model is designed to separate the UI controls from your logic, but still allow controls to interact with the model by using data binding. A view model is overkill for this application, but has advantages when a UI becomes more complex. We’ll build a simplified ViewModel and wire it to the view in this section.

1. Open MainWindow.xaml.cs and remove all logic from AddButton\_Click.

void AddButton\_Click(object sender, RoutedEventArgs e)  
{  
      
}

1. Right-click the project and select Add -> **New Class**. Add the class **MainWindowViewModel**.
2. Give the view model a property with the name **ToDoText** of type **string**.

An easy way to add a property is to use the *prop* code snippet. Inside the class, type “prop”



Now press the Tab key class to expand the code snippet. You can now Tab through replaceable sections of the property and type in the type and name of the property.

class MainWindowViewModel  
{  
    public string ToDoText { get; set; }  
}

1. Add a using statement at the top of the file for System.Collections.ObjectModel.

using System.Collections.ObjectModel;

1. Give the view model another property named **ToDoItems** of type **ObservableCollection<string>**.

ObservableCollection<string> is a generic type – we’ll cover this topic in a later module. For now think of it as a list or collection of strings. The list will allow us to add, remove, and count the string objects it maintains. This is similar to using an array, however, a List makes it easy to add as many new elements as we need whereas an array has a fixed size.

class MainWindowViewModel  
{  
    public string ToDoText { get; set; }  
**public ObservableCollection<String> ToDoItems { get; set; }**  
}

1. Add a constructor to the view model that initializes the ToDoItems property to a new instance of ObservableCollection<string>.

public MainWindowViewModel()  
{  
    ToDoItems = new ObservableCollection<string>();  
}

1. Add a public **method** to the class named **AddToDoItem**. The method returns void and takes no parameters.
2. The implementation of the method should **Add** the **ToDoText** to the **ToDoItems**, then **empty** the **ToDoText**.

public void AddToDoItem()  
{  
    ToDoItems.Add(ToDoText);  
    ToDoText = "";  
}

1. Open **MainWindow.xaml.cs**.
2. Add a new private field to the MainWindow class named **\_model** and of type **MainWindowViewModel**.
3. In the constructor, and after the call to InitializeComponent, set the DataContext property of the MainWindow to the \_model object.

public partial class MainWindow : Window  
{  
    public MainWindow()  
    {  
        InitializeComponent();  
**DataContext = \_model;**        AddButton.Click += new RoutedEventHandler(AddButton\_Click);  
    }  
  
    void AddButton\_Click(object sender, RoutedEventArgs e)  
    {  
          
    }  
  
**MainWindowViewModel \_model = new MainWindowViewModel();**  
}

The DataContext allows us to data bind control properties in the UI to properties on the view model (or whatever object is assigned to the DataContext). The context is available to all controls in the window. We’ll see the TextBox in the window will automatically populate the view model’s ToDoText property with the value the user has typed into the control because it is bound to the property with the following markup (this is **not** code you have to add – it’s just here for background information about WPF).

<TextBox Text="{Binding ToDoText}" … />

1. Inside the AddButton\_Click event handler, invoke the \_model’s AddToDoItem method.

void AddButton\_Click(object sender, RoutedEventArgs e)  
{  
**\_model.AddToDoItem();**  
}

1. Press F5 to run the application and add a few items into the TODO list.

You’ll notice the TextBox control does **not** clear itself when we add an item to the list. This used to work, but we are missing something important when we switched to using a view model. We’ll fix bug this in the next section.

## Part III – Raising Property Changed Events

Our view model is emptying the ToDoText property when adding a new item to the ToDoItems collection, but WPF doesn’t know this. When we change a property value we need to “announce” the change via an event – a PropertyChangedEvent. The ObservableCollection we are using already raises this event when an item is added or deleted from the collection (technically it raises a CollectionChangedEvent). In this section we’ll implement the proper events so WPF knows when something changes.

1. Open MainWindowViewModel.cs
2. Add a using statement for System.ComponentModel

using System.ComponentModel;

1. Derive MainWindowViewModel from the INotifyPropertyChanged interface.

class MainWindowViewModel : INotifyPropertyChanged

1. Implement the INotifyPropertyChanged interface by adding a public event to the view model. The event is named PropertyChanged and the type is PropertyChangedEventHandler.

public event PropertyChangedEventHandler PropertyChanged;

1. Add a private method to the view model named RaisePropertyChanged. The method returns void and takes a string parameter “propertyName”.
2. Inside the method, add an if statement to see if PropertyChanged is not null.

A null value in PropertyChanged means no one has subscribed to the event, so if the value is null we don’t have to raise the event.

private void RaisePropertyChanged(string propertyName)  
{  
    if(PropertyChanged != null)  
    {  
          
    }  
}

1. Inside the if statement, create a new instance of PropertyChangedEventArgs passing the incoming propertyName value to the constructor.
2. Invoke PropertyChanged passing *this* as the sender and the newly constructed *args* as the event arguments.

private void RaisePropertyChanged(string propertyName)  
{  
    if(PropertyChanged != null)  
    {  
        PropertyChangedEventArgs args =   
            new PropertyChangedEventArgs(propertyName);  
        PropertyChanged(this, args);  
    }  
}

1. Change the ToDoText property from an automatic property to a property backed with a field named \_toDoText.

We need to use a backing field to detect changes and raise the PropertyChanged event. Note this change doesn’t change the public API of the object, only the internal implementation details.

private string \_toDoText;  
public string ToDoText  
{  
    get { return \_toDoText; }  
    set { \_toDoText = value; }  
}

1. Inside the set operation, wrap the existing line of code inside an *if* statement. The if statement should check if the incoming value is different than the existing value in \_toDoText.
2. If the value is different, and after setting the new value, invoke RaisePropertyChanged passing “ToDoText” as the property name.

private string \_toDoText;  
public string ToDoText  
{  
    get { return \_toDoText; }  
    set  
    {  
        if(\_toDoText != value)  
        {  
            \_toDoText = value;  
            RaisePropertyChanged("ToDoText");  
        }  
    }  
}

1. Implement the same pattern for the ToDoItems property (if the property changes – raise an event).

Although the ObservableCollection already raises events when we add items to the collection, we need to raise an event if the entire collection changes. Although we aren’t doing this in the current code, we should faithfully raise an event for any public property that changes.

private ObservableCollection<string> \_toDoItems;  
public ObservableCollection<String> ToDoItems  
{  
    get { return \_toDoItems; }  
    set  
    {  
        if (\_toDoItems != value)  
        {  
            \_toDoItems = value;  
            RaisePropertyChanged("ToDoItems");  
        }  
    }  
}

1. Press F5 to run the application again. Make sure the TextBox is cleared every time you add a new item to the to-do list.

## Summary

Congratulations! You’ve worked with events and properties in a WPF application. If you feel like you want some additional challenge then see if you can implement Delete functionality for the application.