[orks & Libraries](https://www.codeproject.com/Chapters/8/Platforms-Frameworks-Libraries.aspx) » [Windows Presentation Foundation](https://www.codeproject.com/KB/WPF/) » [General](https://www.codeproject.com/KB/WPF/#General)

[https://codeproject.global.ssl.fastly.net/App_Themes/CodeProject/Img/bookmark24.png](https://www.codeproject.com/script/Bookmarks/Add.aspx?obid=274982&obtid=2&bio=true&bis=medium) [https://codeproject.global.ssl.fastly.net/images/print24.png](https://www.codeproject.com/Articles/274982/Commands-in-MVVM?display=Print)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Article**  [Browse Code](https://www.codeproject.com/script/Articles/ViewDownloads.aspx?aid=274982)  [Stats](https://www.codeproject.com/script/Articles/Statistics.aspx?aid=274982)  [Revisions (6)](https://www.codeproject.com/script/Articles/ListVersions.aspx?aid=274982)  [Alternatives](https://www.codeproject.com/script/Articles/ListAlternatives.aspx?aid=274982)  [Comments (99)](https://www.codeproject.com/Articles/274982/WebControls/#_comments)  [Add your own alternative version](https://www.codeproject.com/script/Articles/SubmissionWizard.aspx?at=1&apid=274982)  **Tagged as**  [C#](https://www.codeproject.com/Tags/C--hash--)  [Windows](https://www.codeproject.com/Tags/Windows)  [.NET](https://www.codeproject.com/Tags/.NET)  [Architect](https://www.codeproject.com/Tags/Architect)  [Dev](https://www.codeproject.com/Tags/Dev)  [XAML](https://www.codeproject.com/Tags/XAML)  [WPF](https://www.codeproject.com/Tags/WPF)  [Beginner](https://www.codeproject.com/Tags/Beginner)  [Silverlight](https://www.codeproject.com/Tags/Silverlight)  **Stats**  279.4K views  14.7K downloads  268 bookmarked  Posted 29 Oct 2011  [CPOL](http://www.codeproject.com/info/cpol10.aspx) | Top of Form  Commands in MVVM  https://www.codeproject.com/script/Membership/ProfileImages/%7Bdc01949d-03a6-437b-8348-e1b83be25075%7D.jpg  [**Dave Kerr**](https://www.codeproject.com/script/Membership/View.aspx?mid=4259528), 3 Dec 2012   |  |  | | --- | --- | | https://codeproject.global.ssl.fastly.net/script/Ratings/Images/stars-fill-md.png  https://codeproject.global.ssl.fastly.net/script/Ratings/Images/stars-empty-md.png | 4.97 (101 votes) | |  | |  |  |  | | --- | --- | --- | | Rate: | [vote 1vote 2vote 3vote 4vote 5](https://www.codeproject.com/Articles/274982/Commands-in-MVVM) |  | |   A consistent approach to Commands, Asynchronous Commands, and Events-to-Commands for WPF, Silverlight, and WP7.  https://www.codeproject.com/images/warning48.png**Is your email address OK?** You are signed up for our newsletters but your email address is either unconfirmed, or has not been reconfirmed in a long time. Please **click**[**here**](https://www.codeproject.com/script/Membership/SendConfirmRequest.aspx?rp=%2fArticles%2f274982%2fCommands-in-MVVM)**to have a confirmation email sent** so we can confirm your email address and start sending you newsletters again. Alternatively, you can [**update your subscriptions**](https://www.codeproject.com/script/Membership/Subscribe.aspx?rp=%2fArticles%2f274982%2fCommands-in-MVVM).   * [**Download source code - 194 KB**](https://www.codeproject.com/KB/WPF/consistentmvvmcommands/Source_Code.zip)**(Note, the examples are named 'CommandingSample' and are in the '*Samples*' folder of the WPF, Silverlight, and WP7 folders.)** * [**Download example applications - 200 KB**](https://www.codeproject.com/KB/WPF/consistentmvvmcommands/Example_Applications.zip) * [**Download Apex binaries - 54.9 KB**](https://www.codeproject.com/KB/WPF/consistentmvvmcommands/Apex_Binaries.zip)   Contents   1. [Example 1 - A Simple Command](https://www.codeproject.com/Articles/274982/Commands-in-MVVM#example1) 2. [Example 2 - A Simple Command Using Lambda Functions](https://www.codeproject.com/Articles/274982/Commands-in-MVVM#example2) 3. [Example 3 - A Parameterized Command](https://www.codeproject.com/Articles/274982/Commands-in-MVVM#example3) 4. [Example 4 - Enabling or Disabling a Command](https://www.codeproject.com/Articles/274982/Commands-in-MVVM#example4) 5. [Example 5 - Commands that Fire Events](https://www.codeproject.com/Articles/274982/Commands-in-MVVM#example5) 6. [Example 6 - An Asynchronous Command](https://www.codeproject.com/Articles/274982/Commands-in-MVVM#example6) 7. [Example 7 - An Asynchronous Command that Updates the UI](https://www.codeproject.com/Articles/274982/Commands-in-MVVM#example7) 8. [Example 8 - An Asynchronous Command that can be Cancelled](https://www.codeproject.com/Articles/274982/Commands-in-MVVM#example8) 9. [Example 9 - Binding an Event to a Command](https://www.codeproject.com/Articles/274982/Commands-in-MVVM#example9) 10. [How Does it Work - The Command Class](https://www.codeproject.com/Articles/274982/Commands-in-MVVM#hdiw1) 11. [How Does it Work - The Asynchronous Command Class](https://www.codeproject.com/Articles/274982/Commands-in-MVVM#hdiw2) 12. [How Does it Work - The Event Binding Class](https://www.codeproject.com/Articles/274982/Commands-in-MVVM#hdiw3)   Introduction  In this article I am going to describe how Commands work for projects that use MVVM (Model View View-Model). The approach I am describing works in exactly the same way whether you are using WPF, Silverlight, or Windows Phone 7. First, I will show a variety of ways that we can use commands in any MVVM project - this section works as a quick-reference for Commanding. Each example is then described in detail later on in the article - I will show exactly how the code works.  The code used is part of my Apex MVVM library, however as each section is described with full code examples, you can easily integrate these methods into existing projects or libraries - or simply add Apex as a reference and get started straight away.  ***Screenshot 1: Commanding in WPF***  consistentmvvmcommands/WPF.png  ***Screenshot 2: Commanding in Silverlight***  consistentmvvmcommands/SL.png  ***Screenshot 3: Commanding in Windows Phone 7***  consistentmvvmcommands/WP7.png  What is a Command?  This is a really critical concept in MVVM. Here's what you need to know.  **A Command is an *object*that is bound to. It provides a separation between a user interface and logic.**  This is the core concept. In a bit more detail we can describe a command as follows:   1. A command is an object that implements the ICommand interface. 2. Generally, it is associated with a function in some code. 3. User interface elements bind to commands - when they are activated by the user the command is fired - which calls the associated function. 4. Commands know if they are enabled or not. 5. A function can disable the command object - automatically disabling any user interface elements associated with it.   Actually, there's a whole lot more to Commands. We can use commands to deal with asynchronous functionality, to provide logic that can be tested with or without a user interface, and more.  Commanding Examples  First we'll take a look at a set of scenarios where we will need to use commands. Each scenario has an associated details section later on in the article showing how it works.  Important note: Every View Model in this set of examples contains an observable collection of strings named 'Messages' - each example application shows these messages in a list so that we can see what is happening. The base View Model is shown below:  Hide   Copy Code  public class MainViewModel : ViewModel  {  public MainViewModel()  {  }  private ObservableCollection<string> messages = new ObservableCollection<string>();  public ObservableCollection<string> Messages  {  get { return messages; }  }  }  My View Model is derived from 'ViewModel' which implements INotifyPropertyChanged, but you can use any View Model type or base that you like.  Example 1: A Simple Command Example  Requirement: I need to invoke a function of the View Model when a user interface element is pressed or activated.  The most simple example of a command is shown below. First, we add a 'Command' object to our View Model - with an associated function that will be called when the command is invoked:  Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code  public class MainViewModel : ViewModel  {  public MainViewModel()  {  *// Create the simple command - calls DoSimpleCommand.*  simpleCommand = new Command(DoSimpleCommand);  }  /// *<summary>*  /// *The SimpleCommand function.*  /// *</summary>*  private void DoSimpleCommand()  {  *// Add a message.*  Messages.Add("Calling 'DoSimpleCommand'.");  }  /// *<summary>*  /// *The simple command object.*  /// *</summary>*  private Command simpleCommand;  /// *<summary>*  /// *Gets the simple command.*  /// *</summary>*  public Command SimpleCommand  {  get { return simpleCommand; }  }  }  Now we bind the command to the 'Command' property of a button (or other UI element).  Hide   Copy Code  <Button Content="Simple Command" Command="{Binding SimpleCommand}" />  This is the most trivial example of a Command - we bind the Command object to the Command Dependency Property of an interface element. When the element is activated, the Command is invoked. Under the hood, this will cause the 'DoSimpleCommand' function to be invoked.  [Example 1: How Does it Work?](https://www.codeproject.com/Articles/274982/Commands-in-MVVM#hdiw1)  Example 2: A Simple Command Example Using Lambda Expressions  Requirement: I need to invoke a function of the View Model when a user interface element is pressed or activated. However, it's such a simple function that I'd rather not have to write it out explicitly - can I use a lambda expression?  For all of the examples in this article, you can explicitly create a function, just as in Example 1, or you can use a lambda expression in-line. For small functions, a lambda expression can be a little tidier.  Hide   Copy Code  public MainViewModel()  {  *// Create the lambda command, no extra function necessary.*  lambdaCommand = new Command(  () =>  {  Messages.Add("Calling the Lamba Command - no explicit function necessary.");  });  }  /// *<summary>*  /// *The command object.*  /// *</summary>*  private Command lambdaCommand;  /// *<summary>*  /// *Gets the command.*  /// *</summary>*  public Command LambdaCommand  {  get { return lambdaCommand; }  }  Now we bind the command to the 'Command' property of a button (or other UI element).  Hide   Copy Code  <Button Content="Lambda Command" Command="{Binding LambdaCommand}" />  What have we learnt? Anywhere we define a command, we can use either a named function or a lambda expression.  [Example 2: How Does it Work?](https://www.codeproject.com/Articles/274982/Commands-in-MVVM#hdiw1)  Example 3: A Command with a Parameter  Requirement: When I invoke a command, I need to pass a parameter which has been set by a binding.  Anywhere we use a Command object (or an AsynchronousCommand object as we'll see later), we can optionally use a parameter with the function.  Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code  public class MainViewModel : ViewModel  {  public MainViewModel()  {  *// Create the parameterized command.*  parameterizedCommand = new Command(DoParameterisedCommand);  }  /// *<summary>*  /// *The Command function.*  /// *</summary>*  private void DoParameterisedCommand(object parameter)  {  Messages.Add("Calling a Parameterised Command - the Parameter is '" +  parameter.ToString() + "'.");  }  /// *<summary>*  /// *The command object.*  /// *</summary>*  private Command parameterizedCommand;  /// *<summary>*  /// *Gets the command.*  /// *</summary>*  public Command ParameterisedCommand  {  get { return parameterizedCommand; }  }  }  Now we bind the command to the 'Command' property of a button (or other UI element), but also bind a parameter:  Hide   Copy Code  <Button Content="Parameterized Command"  Command="{Binding ParameterizedCommand}" CommandParameter={Binding SomeObject} />  Anywhere we use a command, we can pass a parameter. When creating a command, we can use either an Action (a command function without any parameters) or an Action<object> (a command function that takes a single parameter of type 'object'). We can define the command function used above with a lambda expression if preferred:  Hide   Copy Code  *// Create the parameterized command.*  parameterizedCommand = new Command(  (parameter) =>  {  Messages.Add("Calling a Parameterised Command - the Parameter is '" +  parameter.ToString() + "'.");  });  [Example 3: How Does it Work?](https://www.codeproject.com/Articles/274982/Commands-in-MVVM#hdiw1)  Example 4: Disabling or Enabling a Command  Requirement: I need to be able to disable or enable a command - either with code or with XAML.  Every command has the property 'CanExecute'. When this property is set to true, the command is enabled. When this property is false, the command is disabled and the UI is updated accordingly - for example, a button bound to the command will become disabled.  Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code  public class MainViewModel : ViewModel  {  public MainViewModel()  {  *// Create the enable/disable command, initially disabled.*  enableDisableCommand = new Command(  () =>  {  Messages.Add("Enable/Disable command called.");  }, false);  }  private void DisableCommand()  {  *// Disable the command.*  EnableDisableCommand.CanExecute = false;  }  private void EnableCommand()  {  *// Disable the command.*  EnableDisableCommand.CanExecute = true;  }    /// *<summary>*  /// *The command object.*  /// *</summary>*  private Command enableDisableCommand;  /// *<summary>*  /// *Gets the command.*  /// *</summary>*  public Command EnableDisableCommand  {  get { return enableDisableCommand; }  }  }  Now we bind the command to the 'Command' property of a button (or other UI element). We can also bind the 'CanExecute' property of the command.  Hide   Copy Code  <CheckBox IsChecked="{Binding EnableDisableCommand.CanExecute, Mode=TwoWay}" Content="Enabled" />  <Button Content="Enable/Disable Command" Command="{Binding EnableDisableCommand}" />  We can enable or disable commands in code by setting the CanExecute property. We can also bind to the property to manipulate it from XAML.  Whenever we create a command object (as shown in the constructor), we can pass a boolean as an optional second parameter - this is the initial CanExecute value. By default, it is set to false. In the example above, we set the command to initially disabled.  [Example 4: How Does it Work?](https://www.codeproject.com/Articles/274982/Commands-in-MVVM#hdiw1)  Example 5: Commands that Fire Events  Requirement: I need to know when the command has been executed or is about to be executed.  Every Command has two events - Executed, which is called when the Command has been executed, and Executing, which is called when the Command is about to be executed. The Executing event allows the command to be cancelled.  **Important note**: There is a particular scenario where the events below are *very* useful. Imagine that you want to pop a message box when the command is about to run to ask the user whether they want to continue - where do you do this? In the command code? That is a bad idea - it forces you to have UI code in you View-Model, which clutters the command but also means that it cannot be run from a unit test. You should do it in the view - and with these events, you can. As another example, what if you wanted to set the focus to a specific control after a command is executed? You cannot do it in the View Model, it doesn't have access to the controls, but by subscribing to the Executed event in the View, we can do this without any problems.  Hide   Copy Code  public class MainViewModel : ViewModel  {  public MainViewModel()  {  *// Create the events command.*  eventsCommand = new Command(  () =>  {  Messages.Add("Calling the Events Command.");  });  Now we bind the command to the 'Command' property of a button (or other UI element).  Hide   Copy Code  <Button Content="Events Command" Command="{Binding EventsCommand}" />  So far we have nothing different from the first example. However, we will now subscribe to some events in the View. Note: In my View, my DataContext is named 'viewModel'.  Hide   Copy Code  /// *<summary>*  /// *Interaction logic for MainWindow.xaml*  /// *</summary>*  public partial class MainWindow : Window  {  public MainWindow()  {  InitializeComponent();  viewModel.EventsCommand.Executing +=  new Apex.MVVM.CancelCommandEventHandler(EventsCommand\_Executing);  viewModel.EventsCommand.Executed +=  new Apex.MVVM.CommandEventHandler(EventsCommand\_Executed);  }  void EventsCommand\_Executed(object sender, Apex.MVVM.CommandEventArgs args)  {  viewModel.Messages.Add("The command has finished - this is the View speaking!");  }  void EventsCommand\_Executing(object sender, Apex.MVVM.CancelCommandEventArgs args)  {  if (MessageBox.Show("Cancel the command?",  "Cancel?",  MessageBoxButton.YesNo) == MessageBoxResult.Yes)  args.Cancel = true;  }  }  We are starting to get a hint of the power of this Command implementation. We can subscribe to the Executedand Executing events in the View (or even in another ViewModel or another object) and know when they're fired. The Executing event passes a CancelCommandEventArgs object - this has a property called 'Cancel'. If it is set to true, the command will not execute. Both the CommandEventArgs and CancelCommandEventArgshave one more property - Parameter. This is the parameter that has been passed to the Command (if there is one).  [Example 5: How Does it Work?](https://www.codeproject.com/Articles/274982/Commands-in-MVVM#hdiw1)  Example 6: An Asynchronous Command  Requirement: My Command function will take a long time - I need it to run asynchronously and not block the UI thread.  Now in this circumstance, we would generally have to do something along the lines of create a background worker and run it in the command function. But we have issues:   * What if we want to update the ViewModel in the threaded function? We can't, without invoking actions on the UI thread. * How do we make sure that the Command isn't pressed twice in short succession, running the thread more than once? * How do we keep the View Model clean when we have lots of commands that must be threaded? * How do we keep it consistent for WP7 and Silverlight, where we have different options for threading?   The AsynchronousCommand object (which derives from Command) will deal with all of this - and more. Here's how we can use it:  Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code  public class MainViewModel : ViewModel  {  public MainViewModel()  {  *// Create the async command.*  asyncCommand1 = new AsynchronousCommand(  () =>  {  for (int i = 1; i <= 10; i++)  {  *// Report progress.*  asyncCommand1.ReportProgress(() => { Messages.Add(i.ToString()); });  System.Threading.Thread.Sleep(200);  }  });  }    /// *<summary>*  /// *The command object.*  /// *</summary>*  private AsynchronousCommand asyncCommand1;  /// *<summary>*  /// *Gets the command.*  /// *</summary>*  public AsynchronousCommand AsyncCommand1  {  get { return asyncCommand1; }  }  }  Now we bind the asynchronous command to the 'Command' property of a button (or other UI element).  Hide   Copy Code  <Button Content="Asynchronous Command" Command="{Binding AsyncCommand1}" />  As soon as the command is invoked, the function we have provided (via the lambda expression in the constructor) will be called - on a new thread (from the thread-pool).  If we need to do anything with the View Model objects (which may be bound to UI elements), we can do so via the ReportProgress function:  Hide   Copy Code  asyncCommand1.ReportProgress(() => { Messages.Add(i.ToString()); });  ReportProgress will ensure that the code that is passed to it will be run on the correct thread (the UI thread). This provides a remarkably easy way for your commands to update the UI as they run.  [Example 6: How Does it Work?](https://www.codeproject.com/Articles/274982/Commands-in-MVVM#hdiw2)  Example 7: Changing the UI While an Asynchronous Command is Running  Requirement: My asynchronous command takes a while to run, I'd like to show some progress in the UI.  AsynchronousCommand has a property named 'IsExecuting'. If this is set to true, then the command is running. AsynchronousCommand implements INotifyPropertyChanged, which means that we can bind to this property - and keep our UI fully up to date while the command is running.  Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code  public class MainViewModel : ViewModel  {  public MainViewModel()  {  *// Create the async command.*  asyncCommand2 = new AsynchronousCommand(  () =>  {  for (char c = 'A'; c <= 'Z'; c++)  {  *// Report progress.*  asyncCommand2.ReportProgress(() => { Messages.Add(c.ToString()); });  System.Threading.Thread.Sleep(100);  }  });  }    /// *<summary>*  /// *The command object.*  /// *</summary>*  private AsynchronousCommand asyncCommand2;  /// *<summary>*  /// *Gets the command.*  /// *</summary>*  public AsynchronousCommand AsyncCommand2  {  get { return asyncCommand2; }  }  }  Now we bind the asynchronous command to the 'Command' property of a button (or other UI element). We also bind other UI elements to the IsExecuting property of the command.  Hide   Copy Code  <Button Content="Asynchronous Command" Command="{Binding AsyncCommand2}"  Visibility="{Binding AsyncCommand2.IsExecuting,  Converter={StaticResource BooleanToVisibilityConverter},  ConverterParameter=Invert}" />  <StackPanel Visibility="{Binding AsyncCommand2.IsExecuting,  Converter={StaticResource BooleanToVisibilityConverter}}">  <TextBlock Text="The command is running!" />  <ProgressBar Height="20" Width="120" IsIndeterminate="True" />  </StackPanel>  In this example, as soon as the command starts to run, the button disappears and a text block and progress bar appears. Note that we are binding to the IsExecuting property of the command.  Hide   Copy Code  asyncCommand1.ReportProgress(() => { Messages.Add(i.ToString()); });  ReportProgress will ensure that the code that is passed to it will be run on the correct thread (the UI thread). This provides a remarkably easy way for your commands to update the UI as they run.  Note: The 'Invert' parameter can be passed to the BooleanToVisilityConverter because it is an extended version of the standard BooleanToVisibilityConverter that is defined in Apex.Converters. It inverts the result, very useful in these circumstances.  [Example 7: How Does it Work?](https://www.codeproject.com/Articles/274982/Commands-in-MVVM#hdiw2)  Example 8: Allowing a User to Cancel an AsynchronousCommand  Requirement: I need to let the user cancel an asynchronous command.  We have a couple of features of AsynchronousCommand that we can use here. Every AsynchronousCommandobject contains **another** Command - named CancelCommand. This can be bound to a UI element (for example, a button named 'Cancel') or invoked via code. When this command is invoked, the 'IsCancellationRequested' property of the AsynchronousCommand is set to true (note that this is a property that uses INotifyPropertyChanged, so you can also bind to it). As long as you periodically call the CancelIfRequested function and return if it returns true, then the cancellation functionality is enabled.  Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code  public class MainViewModel : ViewModel  {  public MainViewModel()  {  *// Create the cancellable async command.*  cancellableAsyncCommand = new AsynchronousCommand(  () =>  {  for(int i = 1; i <= 100; i++)  {  *// Do we need to cancel?*  if(cancellableAsyncCommand.CancelIfRequested())  return;  *// Report progress.*  cancellableAsyncCommand.ReportProgress( () => { Messages.Add(i.ToString()); } );  System.Threading.Thread.Sleep(100);  }  });  }    /// *<summary>*  /// *The command object.*  /// *</summary>*  private AsynchronousCommand cancellableAsyncCommand;  /// *<summary>*  /// *Gets the command.*  /// *</summary>*  public AsynchronousCommand CancellableAsyncCommand  {  get { return cancellableAsyncCommand; }  }  }  Now we bind the asynchronous command to the 'Command' property of a button (or other UI element). We also bind other UI elements to the IsExecuting property of the command.  Hide   Copy Code  <Button Content="Cancellable Async Command"  Command="{Binding CancellableAsyncCommand}"  Visibility="{Binding CancellableAsyncCommand.IsExecuting,  Converter={StaticResource BooleanToVisibilityConverter},  ConverterParameter=Invert}" />  <StackPanel Visibility="{Binding CancellableAsyncCommand.IsExecuting,  Converter={StaticResource BooleanToVisibilityConverter}}">  <TextBlock Margin="4" Text="The command is running!" />  <ProgressBar Margin="4" Height="20"  Width="120" IsIndeterminate="True" />  **<Button Margin="4" Content="Cancel"**  **Command="{Binding CancellableAsyncCommand.CancelCommand}" />**  </StackPanel>  In this example, we show a button named 'Cancel' when the command is executing. This button binds to the CancellableAsyncCommand.CancelCommand property. Because we use the CancelIfRequested function in our asynchronous function, we can gracefully cancel the asynchronous command.  **Note**: When an asynchronous command is cancelled - Executed is **not** called. An event named Cancelled is called instead, which takes the same parameters.  [Example 8: How Does it Work?](https://www.codeproject.com/Articles/274982/Commands-in-MVVM#hdiw2)  Example 9: Binding an Event to a Command  Requirement: I have a command that must be invoked, however the UI element which should invoke it does not have a 'Command' property - it only has an event.  In the case of forcing an event to fire a command, we can use an Attached Property named 'EventBindings'. This property and the associated classes are in the Apex.Commands namespace. EventBindings takes an EventBindingCollection, which is a simple collection of EventBinding objects. Each EventBinding object takes the name of the event and the command to fire.  Hide   Copy Code  public class MainViewModel : ViewModel  {  public MainViewModel()  {  *// Create the event binding.*  EventBindingCommand = new Command(  () =>  {  Messages.Add("Command called by an event.");  });  }    /// *<summary>*  /// *The command object.*  /// *</summary>*  private Command eventBindingCommand;  /// *<summary>*  /// *Gets the command.*  /// *</summary>*  public Command EventBindingCommand  {  get { return eventBindingCommand; }  }  }  Now we can bind to this command through an event as below:  Hide   Copy Code  <Border Margin="20" Background="Red">  *<!-- Bind the EventBindingCommand to the MouseLeftButtonDown event. -->*  <apexCommands:EventBindings.EventBindings>  <apexCommands:EventBindingCollection>  <apexCommands:EventBinding EventName="MouseLeftButtonDown"  Command="{Binding EventBindingCommand}" />  </apexCommands:EventBindingCollection>  </apexCommands:EventBindings.EventBindings>    <TextBlock VerticalAlignment="Center"  HorizontalAlignment="Center" Text="Left Click on Me"  FontSize="16" Foreground="White" />  </Border>  By using the EventBindings attached property, we can bind any event to a command.  [Example 9: How Does it Work?](https://www.codeproject.com/Articles/274982/Commands-in-MVVM#hdiw9)  How Does it Work?  Every example above has an associated description of how it works under the hood.  The Command Class: Used in Examples 1, 2, 3, 4, and 5  These examples use the 'Command' class, which is described section by section below:  Hide   Copy Code  /// *<summary>*  /// *The ViewModelCommand class - an ICommand that can fire a function.*  /// *</summary>*  public class Command : ICommand  {  /// *<summary>*  /// *Initializes a new instance of the <see cref="Command"/> class.*  /// *</summary>*  /// *<param name="action">The action.</param>*  /// *<param name="canExecute">if set to <c>true</c> [can execute].</param>*  public Command(Action action, bool canExecute = true)  {  *// Set the action.*  this.action = action;  this.canExecute = canExecute;  }  /// *<summary>*  /// *Initializes a new instance of the <see cref="Command"/> class.*  /// *</summary>*  /// *<param name="parameterizedAction">The parameterized action.</param>*  /// *<param name="canExecute">if set to <c>true</c> [can execute].</param>*  public Command(Action<object> parameterizedAction, bool canExecute = true)  {  *// Set the action.*  this.parameterizedAction = parameterizedAction;  this.canExecute = canExecute;  }  First of all, we have our two constructors that create the command - passing in an Action or an Action<object>. Action<object> is used for parameterised commands.  Next we have our action (or our parameterized action) as well as the 'canExecute' flag. Note that when the canExecute flag is changed, we must call canExecuteChanged.  Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code  /// *<summary>*  /// *The action (or parameterized action) that will be called when the command is invoked.*  /// *</summary>*  protected Action action = null;  protected Action<object> parameterizedAction = null;  /// *<summary>*  /// *Bool indicating whether the command can execute.*  /// *</summary>*  private bool canExecute = false;  /// *<summary>*  /// *Gets or sets a value indicating whether this instance can execute.*  /// *</summary>*  /// *<value>*  /// *<c>true</c> if this instance can execute; otherwise, <c>false</c>.*  /// *</value>*  public bool CanExecute  {  get { return canExecute; }  set  {  if (canExecute != value)  {  canExecute = value;  EventHandler canExecuteChanged = CanExecuteChanged;  if (canExecuteChanged != null)  canExecuteChanged(this, EventArgs.Empty);  }  }  }  Following this, we implement the ICommand interface:  Hide   Copy Code  /// *<summary>*  /// *Defines the method that determines whether the command can execute in its current state.*  /// *</summary>*  /// *<param name="parameter">Data used by the command.*  /// *If the command does not require data to be passed,*  /// *this object can be set to null.</param>*  /// *<returns>*  /// *true if this command can be executed; otherwise, false.*  /// *</returns>*  bool ICommand.CanExecute(object parameter)  {  return canExecute;  }  /// *<summary>*  /// *Defines the method to be called when the command is invoked.*  /// *</summary>*  /// *<param name="parameter">Data used by the command.*  /// *If the command does not require data to be passed,*  /// *this object can be set to null.</param>*  void ICommand.Execute(object parameter)  {  this.DoExecute(parameter);  }  We haven't seen the 'DoExecute' function yet - we'll save that till last.  Hide   Copy Code  /// *<summary>*  /// *Occurs when can execute is changed.*  /// *</summary>*  public event EventHandler CanExecuteChanged;  /// *<summary>*  /// *Occurs when the command is about to execute.*  /// *</summary>*  public event CancelCommandEventHandler Executing;  /// *<summary>*  /// *Occurs when the command executed.*  /// *</summary>*  public event CommandEventHandler Executed;  Remember the examples with the events? Well, here they are - as well as the CanExecuteChanged event required by ICommand.  Before we move onto the DoExecute function, we create an Invoke function for each event, so that we can invoke them in derived classes.  Hide   Copy Code  protected void InvokeAction(object param)  {  Action theAction = action;  Action<object> theParameterizedAction = parameterizedAction;  if (theAction != null)  theAction();  else if (theParameterizedAction != null)  theParameterizedAction(param);  }  protected void InvokeExecuted(CommandEventArgs args)  {  CommandEventHandler executed = Executed;  *// Call the executed event.*  if (executed != null)  executed(this, args);  }  protected void InvokeExecuting(CancelCommandEventArgs args)  {  CancelCommandEventHandler executing = Executing;  *// Call the executed event.*  if (executing != null)  executing(this, args);  }  Note that InvokeAction has to be a bit clever - we invoke either the action or the parameterized action, whichever has been set. Finally, we have DoExecute:  Hide   Copy Code  /// *<summary>*  /// *Executes the command.*  /// *</summary>*  /// *<param name="param">The param.</param>*  public virtual void DoExecute(object param)  {  *// Invoke the executing command, allowing the command to be cancelled.*  CancelCommandEventArgs args =  new CancelCommandEventArgs() { Parameter = param, Cancel = false };  InvokeExecuting(args);  *// If the event has been cancelled, bail now.*  if (args.Cancel)  return;  *// Call the action or the parameterized action, whichever has been set.*  InvokeAction(param);  *// Call the executed function.*  InvokeExecuted(new CommandEventArgs() { Parameter = param });  }  DoExecute is very straightforward - it calls the appropriate events (cancelling if required!) and that's all there is to it.  This class fully implements ICommand and provides the functionality described in Examples 1, 2, 3, 4, and 5.  The AsynchronousCommand Class: Used in Examples 6, 7, and 8  Examples 6, 7, and 8 use the AsynchronousCommand class, which derives from Command and is described blow-by-blow below.  First we will define the class and the constructors (which call the base):  Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code  /// *<summary>*  /// *The AsynchronousCommand is a Command that runs on a thread from the thread pool.*  /// *</summary>*  public class AsynchronousCommand : Command, INotifyPropertyChanged  {  /// *<summary>*  /// *Initializes a new instance of the <see cref="AsynchronousCommand"/> class.*  /// *</summary>*  /// *<param name="action">The action.</param>*  /// *<param name="canExecute">if set*  /// *to <c>true</c> the command can execute.</param>*  public AsynchronousCommand(Action action, bool canExecute = true)  : base(action, canExecute)  {  *// Initialise the command.*  Initialise();  }  /// *<summary>*  /// *Initializes a new instance of the <see cref="AsynchronousCommand"/> class.*  /// *</summary>*  /// *<param name="parameterizedAction">The parameterized action.</param>*  /// *<param name="canExecute">if set to <c>true</c> [can execute].</param>*  public AsynchronousCommand(Action<object> parameterizedAction, bool canExecute = true)  : base(parameterizedAction, canExecute)  {  *// Initialise the command.*  Initialise();  }  We implement INotifyPropertyChanged so that we can notify when the 'IsExecuting' variable changes. Both constructors call 'Initialise', let's look at this now:  Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code  /// *<summary>*  /// *The cancel command.*  /// *</summary>*  private Command cancelCommand;  /// *<summary>*  /// *Gets the cancel command.*  /// *</summary>*  public Command CancelCommand  {  get { return cancelCommand; }  }  /// *<summary>*  /// *Gets or sets a value indicating whether this instance is cancellation requested.*  /// *</summary>*  /// *<value>*  /// *<c>true</c> if this instance is cancellation requested; otherwise, <c>false</c>.*  /// *</value>*  public bool IsCancellationRequested  {  get  {  return isCancellationRequested;  }  set  {  if (isCancellationRequested != value)  {  isCancellationRequested = value;  NotifyPropertyChanged("IsCancellationRequested");  }  }  }  /// *<summary>*  /// *Initialises this instance.*  /// *</summary>*  private void Initialise()  {  *// Construct the cancel command.*  cancelCommand = new Command(  () =>  {  *// Set the Is Cancellation Requested flag.*  IsCancellationRequested = true;  }, true);  }  There's rather a lot here. We've got a Command object (just as in the examples) and all it does is set the 'IsCancellationRequested' flag to true. Initialise creates this object and we have a property to access it. We also have the 'IsCancellationRequested' property, which notifies when it is changed. As we will also want to know when we're executing, let's add a property for that:  Hide   Copy Code  /// *<summary>*  /// *Flag indicating that the command is executing.*  /// *</summary>*  private bool isExecuting = false;  /// *<summary>*  /// *Gets or sets a value indicating whether this instance is executing.*  /// *</summary>*  /// *<value>*  /// *<c>true</c> if this instance is executing; otherwise, <c>false</c>.*  /// *</value>*  public bool IsExecuting  {  get  {  return isExecuting;  }  set  {  if (isExecuting != value)  {  isExecuting = value;  NotifyPropertyChanged("IsExecuting");  }  }  }  Not too bad. As we allow cancellation, we'll need a Cancelled event and the PropertyChanged event (from INotifyPropertyChanged):  Hide   Copy Code  /// *<summary>*  /// *The property changed event.*  /// *</summary>*  public event PropertyChangedEventHandler PropertyChanged;  /// *<summary>*  /// *Occurs when the command is cancelled.*  /// *</summary>*  public event CommandEventHandler Cancelled;  Aside from the standard wiring up of NotifyPropertyChanged and the cancellation flag, the real interest is in DoExecute:  Hide   Copy Code  /// *<summary>*  /// *Executes the command.*  /// *</summary>*  /// *<param name="param">The param.</param>*  public override void DoExecute(object param)  {  *// If we are already executing, do not continue.*  if (IsExecuting)  return;  *// Invoke the executing command, allowing the command to be cancelled.*  CancelCommandEventArgs args =  new CancelCommandEventArgs() { Parameter = param, Cancel = false };  InvokeExecuting(args);  *// If the event has been cancelled, bail now.*  if (args.Cancel)  return;  *// We are executing.*  IsExecuting = true;  We never run the command if it is already executing. Following this, we allow the command to be cancelled and set the executing flag:  Hide   Copy Code  *// Store the calling dispatcher.*  #if !SILVERLIGHT  callingDispatcher = Dispatcher.CurrentDispatcher;  #else  callingDispatcher = System.Windows.Application.Current.RootVisual.Dispatcher;  #endif  We must store the dispatcher that the command is executed on. Why? So that when we report progress, we report it on the appropriate dispatcher. We have to be careful how we store the calling dispatcher, as it works a little different between Silverlight and WPF.  Hide   Copy Code  *// Run the action on a new thread from the thread pool*  *// (this will therefore work in SL and WP7 as well).*  ThreadPool.QueueUserWorkItem(  (state) =>  {  *// Invoke the action.*  InvokeAction(param);  *// Fire the executed event and set the executing state.*  ReportProgress(  () =>  {  *// We are no longer executing.*  IsExecuting = false;  *// If we were cancelled,*  *// invoke the cancelled event - otherwise invoke executed.*  if(IsCancellationRequested)  InvokeCancelled(new CommandEventArgs() { Parameter = param });  else  InvokeExecuted(new CommandEventArgs() { Parameter = param });  *// We are no longer requesting cancellation.*  IsCancellationRequested = false;  }  );  }  );  }  Now the clever stuff. Using the thread pool, we queue the InvokeAction function - this will call the command function (on a new thread). Now remember - ReportProgress runs on the calling dispatcher, so that's where we must change properties and call Executed etc. So on the calling dispatcher (once the action is finished), we clear the 'IsExecuting' flag and invoke either Cancelled or Executed depending on whether the event was cancelled. That's it. We're only left with ReportProgress:  Hide   Copy Code  /// *<summary>*  /// *Reports progress on the thread which invoked the command.*  /// *</summary>*  /// *<param name="action">The action.</param>*  public void ReportProgress(Action action)  {  if (IsExecuting)  {  if (callingDispatcher.CheckAccess())  action();  else  callingDispatcher.BeginInvoke(((Action)(() => { action(); })));  }  }  With ReportProgress, we invoke the provided action on the calling dispatcher, or just invoke it if we **are** the calling dispatcher.  **A Note on Unit Testing and the AsynchronousCommand**  If you are having any difficulty unit testing AsynchronousCommand objects, it is because when unit testing, there is no Dispatcher Frame. There is an explanation on dealing with this here: [consistentmvvmcommands.aspx?msg=4084944#xx4084944xx](https://www.codeproject.com/KB/WPF/consistentmvvmcommands.aspx?msg=4084944" \l "xx4084944xx).  The EventBindings: Used in Example 9  The EventBindings code has a few more complexities due to the differences between WPF and Silverlight. In WPF, EventBindingsCollection is a FreezableCollection. This is so that the children of the collection (the event bindings) inherit the data context of the parent element. In Silverlight, we do not have a FreezableCollection, so we must set the datacontext of each event binding ourselves. Here's how the EventBindings property is defined:  Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code  public static class EventBindings  {  /// *<summary>*  /// *The Event Bindings Property.*  /// *</summary>*  private static readonly DependencyProperty EventBindingsProperty =  DependencyProperty.RegisterAttached("EventBindings",  typeof(EventBindingCollection), typeof(EventBindings),  new PropertyMetadata(null, new PropertyChangedCallback(OnEventBindingsChanged)));  /// *<summary>*  /// *Gets the event bindings.*  /// *</summary>*  /// *<param name="o">The o.</param>*  /// *<returns></returns>*  public static EventBindingCollection GetEventBindings(DependencyObject o)  {  return (EventBindingCollection)o.GetValue(EventBindingsProperty);  }  /// *<summary>*  /// *Sets the event bindings.*  /// *</summary>*  /// *<param name="o">The o.</param>*  /// *<param name="value">The value.</param>*  public static void SetEventBindings(DependencyObject o,  EventBindingCollection value)  {  o.SetValue(EventBindingsProperty, value);  }  /// *<summary>*  /// *Called when event bindings changed.*  /// *</summary>*  /// *<param name="o">The o.</param>*  /// *<param name="args">The <see*  */// cref="System.Windows.DependencyPropertyChangedEventArgs"/>*  /// *instance containing the event data.</param>*  public static void OnEventBindingsChanged(DependencyObject o,  DependencyPropertyChangedEventArgs args)  {  *// Cast the data.*  EventBindingCollection oldEventBindings =  args.OldValue as EventBindingCollection;  EventBindingCollection newEventBindings =  args.NewValue as EventBindingCollection;  *// If we have new set of event bindings, bind each one.*  if (newEventBindings != null)  {  foreach (EventBinding binding in newEventBindings)  {  binding.Bind(o);  #if SILVERLIGHT  *// If we're in Silverlight we don't inherit the*  *// data context so we must set this helper variable.*  binding.ParentElement = o as FrameworkElement;  #endif  }  }  }  }  As we can see, the only function of real interest is EventBinding.Bind, which is listed below:  Hide   Copy Code  public void Bind(object o)  {  try  {  *// Get the event info from the event name.*  EventInfo eventInfo = o.GetType().GetEvent(EventName);  *// Get the method info for the event proxy.*  MethodInfo methodInfo = GetType().GetMethod("EventProxy",  BindingFlags.NonPublic | BindingFlags.Instance);  *// Create a delegate for the event to the event proxy.*  Delegate del = Delegate.CreateDelegate(eventInfo.EventHandlerType, this, methodInfo);  *// Add the event handler. (Removing it first if it already exists!)*  eventInfo.RemoveEventHandler(o, del);  eventInfo.AddEventHandler(o, del);  }  catch (Exception e)  {  string s = e.ToString();  }  }  This is a standard approach to this requirement, I have seen it in other articles. However, we must be a bit more careful in our event proxy (this I haven't seen before):  Hide   Copy Code  /// *<summary>*  /// *Proxy to actually fire the event.*  /// *</summary>*  /// *<param name="o">The object.</param>*  /// *<param name="e">The <see*  */// cref="System.EventArgs"/> instance*  /// *containing the event data.</param>*  private void EventProxy(object o, EventArgs e)  {  #if SILVERLIGHT  *// If we're in Silverlight, we have NOT inherited the data context*  *// because the EventBindingCollection is not a framework element and*  *// therefore out of the logical tree. However, we can set it here*  *// and update the bindings - and it will all work.*  DataContext = ParentElement != null ? ParentElement.DataContext : null;  var bindingExpression = GetBindingExpression(EventBinding.CommandProperty);  if(bindingExpression != null)  bindingExpression.UpdateSource();  bindingExpression = GetBindingExpression(EventBinding.CommandParameterProperty);  if (bindingExpression != null)  bindingExpression.UpdateSource();  #endif  if (Command != null)  Command.Execute(CommandParameter);  }  Final Thoughts  In the example applications I have provided, the ViewModel class is the same in every project - this code works exactly the same in WPF, Silverlight, and WP7.  I hope you find this article useful, please let me know via the Comments section if you notice any problems.  To keep up to date on Apex development, you can see my blog: [www.dwmkerr.com](http://www.dwmkerr.com/) or the Apex CodePlex page: [apex.codeplex.com](http://apex.codeplex.com/).  License  This article, along with any associated source code and files, is licensed under [The Code Project Open License (CPOL)](http://www.codeproject.com/info/cpol10.aspx)  Share  Bottom of Form |

v