Mishra, Siddharth

Beckman Coulter  [Company address]

WPF Concepts

Table of Contents

[WPF Command Model 2](#_Toc394989373)

[Commands 2](#_Toc394989374)

[Command Bindings 2](#_Toc394989375)

[Command Sources 2](#_Toc394989376)

[Command Targets 2](#_Toc394989377)

[ICommand Interface 2](#_Toc394989378)

[Execute() 2](#_Toc394989379)

[CanExecute() 2](#_Toc394989380)

[Event CanExecuteChanged 2](#_Toc394989381)

[Example: 3](#_Toc394989382)

[RoutedCommand 4](#_Toc394989383)

[Custom Routed Events 5](#_Toc394989384)

[RoutedUICommands 5](#_Toc394989385)

[The Command Library 5](#_Toc394989386)

# WPF Command Model

The WPF command model consists of a surprising number of moving parts. Altogether, it has 4 key ingredients:

## Commands

A command represents an application task and keeps track of whether it can be executed. However, commands don’t actually contain the code that performs the application task.

## Command Bindings

Command binds to User Interface. You can use same command with different command bindings.

## Command Sources

A command source triggers a command. For example, Button is a command source.

## Command Targets

A command target is the element on which the command is being performed. For example, a Paste command might insert text into a TextBox, and an OpenFile command might pop a document into a DocumentViewer.

# ICommand Interface

The heart of the WPF command model is System.Windows.Input.ICommand interface that defoines how commands work. this interface includes two methods and an event:

|  |
| --- |
| public interface ICommand  {      void Execute(object parameter);        bool CanExecute(object parameter);      event EventHandler CanExecuteChanged;  } |

## Execute()

The Execute() method would contain the application task logic ( for example printing the document or opening another window). It accepts an additional parameter which user can use to pass along any extra information he needs.

## CanExecute()

The CanExecute() method will return state of the command. It will return True if it is enabled and false if it is disabled. It accepts an additional parameter which user can use to pass along any extra information he needs.

## Event CanExecuteChanged

This event is raised when the state changes. This is a signal to any control using the command that they should call the CanExecute() method to check the command’s state. This is part of the glue that allows command sources (Button or MenuItems) to automatically enable themselves when the command is available and to disable when it is not.

## Example:

<http://blogs.msdn.com/b/mikehillberg/archive/2009/03/20/icommand-is-like-a-chocolate-cake.aspx>

Given an instance of an ICommand, you just call Execute, and it does whatever it’s supposed to do. Except you shouldn’t call it if it’s CanExecute is false. If you want to know when CanExecute might be willing to give you a different answer, listen to the CanExecuteChanged event.

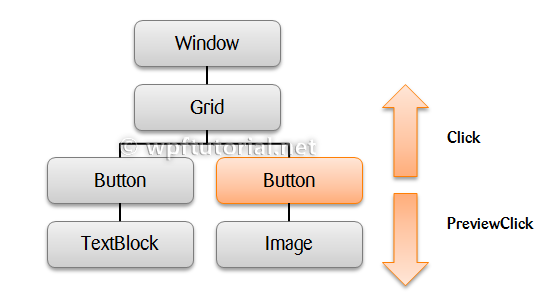
|  |
| --- |
| public class HelloWorldCommand : ICommand    {      #region ICommand Members        public void Execute(object parameter)      {        MessageBox.Show(parameter.ToString(), "Hello World Command", MessageBoxButton.OK, MessageBoxImage.Information);      }        public bool CanExecute(object parameter)      {        return true;      }      public event EventHandler CanExecuteChanged;        #endregion    } |

Once you have an ICommand instance handy, you can give it to a Button (on the Button.Command property), and Button knows what to do with it. As the simplest example, you can do this with the previous command:

|  |
| --- |
| <Window x:Class="WPFTutorial.XAMLTutorial"          xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"          xmlns:cmd="clr-namespace:WPFTutorial.Command"          Title="XAMLTutorial"          Width="300"          Height="300">    <Window.Resources>      <cmd:HelloWorldCommand x:Key="HelloWorldCommand" />    </Window.Resources>    <Grid>  <Button x:Name="HelloWorldCommand"              Grid.Row="0"              Grid.Column="1"              Margin="5"              Command="{StaticResource HelloWorldCommand}"              CommandParameter="Hey, This is Hello world command."              Content="Hello World" />    </Grid>  </Window> |

# RoutedCommand

* The RoutedCommand class is the only class in WPF which implements ICommand interface.
* All WPF provided commands are instances of RoutedCommand.
* RoutedCommand class doesn’t contain any application logic. It simply represents a command.
* RoutedCommand class provides an infrastructure for event tunneling and bubbling.
* ICommand interface encapsulate the idea of command but RoutedCommand modifies the command so that it can bubbly through WPF element hierarchy to get to the right event handler.
* RoutedCommand calss implements the ICommand interface privately and then adds different version of its methods. The most obvious change you will notice is that the Execute() and CanExecute() methods take an extra parameter.



Routed events are events which navigate up or down the visual tree according to their RoutingStrategy. The routing strategy can be **bubble, tunnel** or **direct**. You can hook up event handlers on the element that raises the event or also on other elements above or below it by using the attached event syntax: Button.Click="Button\_Click".

Routed events normally appear as pair. The first is a tunneling event called PreviewMouseDown and the second is the bubbling called MouseDown. They don't stop routing if the reach an event handler. To stop routing then you have to set **e.Handled = true**;

* **Tunneling** The event is raised on the root element and navigates down to the visual tree until it reaches the source element or until the tunneling is stopped by marking the event as handled. By naming convention it is called Preview... and appears before corresponding bubbling event.
* **Bubbling** The event is raised on the source element and navigates up to the visual tree until it reaches the root element or until the bubbling is stopped by marking the event as handled. ***The bubbling event is raised after the tunneling event***.
* **Direct** The event is raised on the source element and must be handled on the source element itself. This behavior is the same as normal .NET events.

## Custom Routed Events

|  |
| --- |
| // Register the routed event  **public** **static** **readonly** RoutedEvent SelectedEvent =  EventManager.RegisterRoutedEvent( "Selected", RoutingStrategy.Bubble,  [**typeof**](http://www.google.com/search?q=typeof+msdn.microsoft.com)(RoutedEventHandler), [**typeof**](http://www.google.com/search?q=typeof+msdn.microsoft.com)(MyCustomControl));    // .NET wrapper  **public** **event** RoutedEventHandler Selected  {  add { AddHandler(SelectedEvent, value); }  remove { RemoveHandler(SelectedEvent, value); }  }    // Raise the routed event "selected"  RaiseEvent([**new**](http://www.google.com/search?q=new+msdn.microsoft.com) RoutedEventArgs(MyCustomControl.SelectedEvent)); |

# RoutedUICommands

Most of the commands we deal with won’t be RoutedCommand objects but it will be instances of the RoutedUICommand class, which derives from RoutedCommand.

In fact, all the ready-made commands provided by WPF are RoutedUICommand objects. RoutedUICommand is intended for commands with text that is displayed somewhere in the user interface. For example. The text of a menu item or the tooltip for a toolbar button.

The RoutedDUICommand class adds a single property “Text” which is the display text for that command.

The advanctage of defining the command text witht the command (rather than directly on control) is that you can perform your localization in one place.

If we don’t use command text then the command will behave like RoutedCommand only.

# The Command Library