Commands

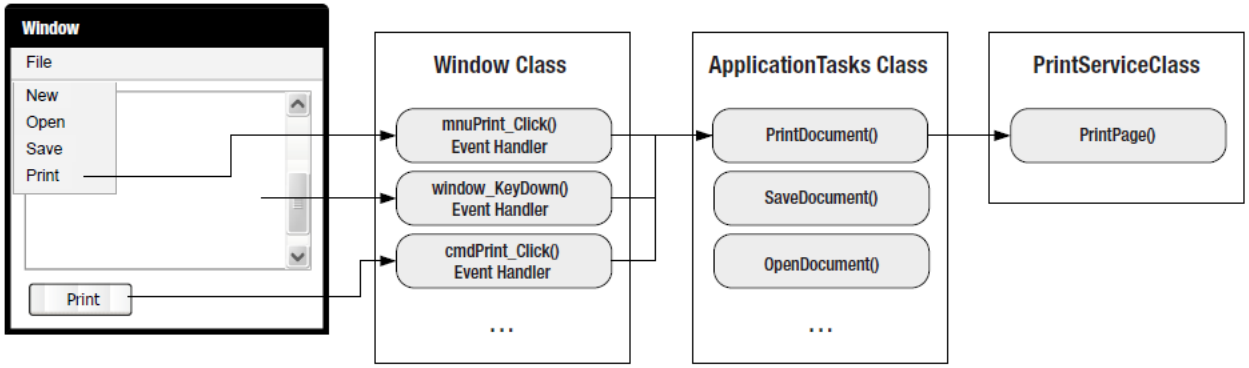
Agenda

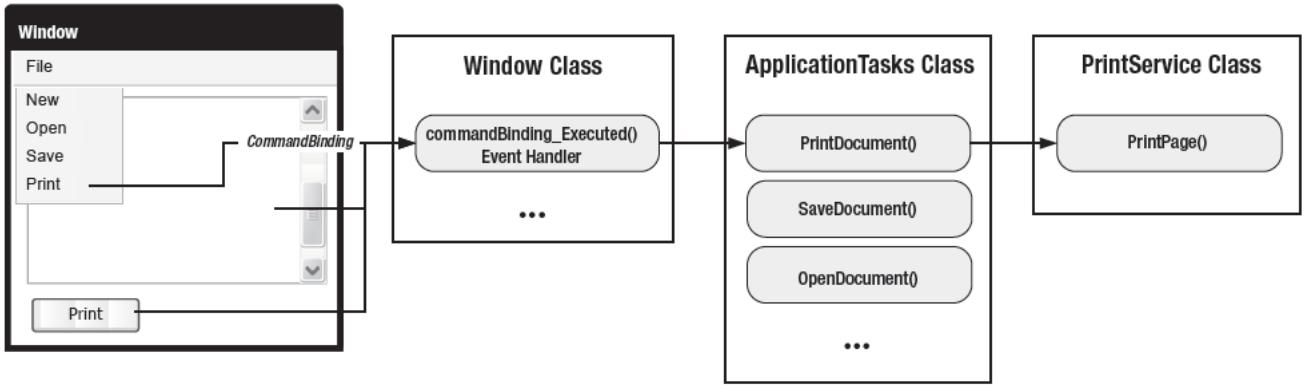
- The WPF Command Model
- Custom Commands

Commands

- The input events we've examined give us a detailed view of user input directed at individual elements
- However, it is often helpful to focus on what the user wants our application to do
  - rather than how she asked us to do it
- WPF supports this through the ***command*** abstraction
  - a command is an action the application performs at the user's request
- The way in which a command is invoked isn't usually important
  - Whether the user:
    - presses Ctrl-C,
    - selects the Edit - Copy menu item, or
    - clicks the Copy button on the toolbar
  - the application's response should be the same in each case

Mapping Event Handlers To A Task





Associating A Menuitem With A Command

- The command system lets a UI element provide a single handler for a command
  - reducing clutter and improving the clarity of your code

```
<DockPanel>
   <Menu DockPanel.Dock="Top">

   <MenuItem Header="_Edit">
   <MenuItem Header="Cu_t" Command="ApplicationCommands.Cut" />
   <MenuItem Header="_Copy" Command="ApplicationCommands.Copy" />
   <MenuItem Header="_Paste" Command="ApplicationCommands.Paste" />
   </MenuItem>
   </Menu>
   <ToolBarTray DockPanel.Dock="Top">
      <ToolBar>
      <Button Command="Cut" Content="Cut" />
      <Button Command="Copy" Content="Copy" />
      <Button Command="Paste" Content="Paste" />
      </ToolBar>
   </ToolBarTray>
   <TextBox />
</DockPanel>
```

Basic command handling

```
<!– XAML –>
<Window ...>
  <Grid>
    <Button Command="ApplicationCommands.Properties“
            Content="_Properties"/> </Grid>
</Window>

// Codebehind
public partial class Window1 : Window {
  public Window1( ) {
    InitializeComponent( );
    InputBinding ib = new InputBinding(
                      ApplicationCommands.Properties,
                      new KeyGesture(Key.Enter, ModifierKeys.Alt));
    this.InputBindings.Add(ib);
    CommandBinding cb = new CommandBinding(ApplicationCommands.Properties);
    cb.Executed += new ExecutedRoutedEventHandler(cb_Executed);
    this.CommandBindings.Add(cb);
  }
  void cb_Executed(object sender, ExecutedRoutedEventArgs e) {
    MessageBox.Show("Properties");
  }
}
```

The Command System

- **Command object**
  - An object identifying a particular command, such as copy or paste
- **Input binding**
  - An association between a particular input (e.g., Ctrl-C) and a command (e.g., Copy)
- **Command source**
  - The object that invoked the command, such as a Button, or an input binding
- **Command target**
  - The UI element that will be asked to execute the command—typically the control that had the keyboard focus when the command was invoked
- **Command binding**
  - A declaration that a particular UI element knows how to handle a particular command

WPF Commands Shortcomings

- WPF's RoutedCommand has one major shortcoming:
  - It can only route between objects in the visual tree
  - You can't send a WPF command to an object in the Model layer (BLL)

Custom Commands

- The following three major types represent commands in WPF:
  - ICommand
  - RoutedCommand
  - RoutedUICommand
- ICommand is the base interface for all commands and is the interface you'd implement to support your own non-UI related back-end commanding systems
- Josh Smith and Laurent Bugnion have done this for you, and created the RelayCommand and RelayCommand<T> classes for you to use (download from: http://mvvmfoundation.codeplex.com/)
- **With RelayCommand it is very easy to make custom commands!**
- *Or you can use **DelegateCommands***

RelayCommand Example

```
class DeviceViewModel
{
   public RelayCommand ConfigureDeviceCommand { get; private set; }

   public DeviceViewModel() {
      ConfigureDeviceCommand = new RelayCommand(
         () => ConfigureDevice(),
         () => CanConfigureDevice);
      }

   private void ConfigureDevice() {
      DeviceInfo workingCopy = new DeviceInfo(_selectedDevice);
      var win = new NewDeviceWizard(workingCopy);
      }

   protected bool CanConfigureDevice  {
      get { return (_selectedDevice != null); }
      }
```

```
<Button Grid.Row="4" Grid.Column="1"
        TabIndex="1"
        Content="_Configure device"
        Command="{Binding ConfigureDeviceCommand}"
        />
```

RelayCommand – Simple Usage

- No initializing in constructor

```
ICommand _PreviusCommand;
public ICommand PreviusCommand {
    get { return _PreviusCommand ??
          (_PreviusCommand = new RelayCommand(
           PreviusCommandExecute, PreviusCommandCanExecute)); }
}

private void PreviusCommandExecute() {
    if (CurrentIndex > 0)
        --CurrentIndex;
}

private bool PreviusCommandCanExecute() {
    if (CurrentIndex > 0)
         return true;
    else
         return false;
}
```

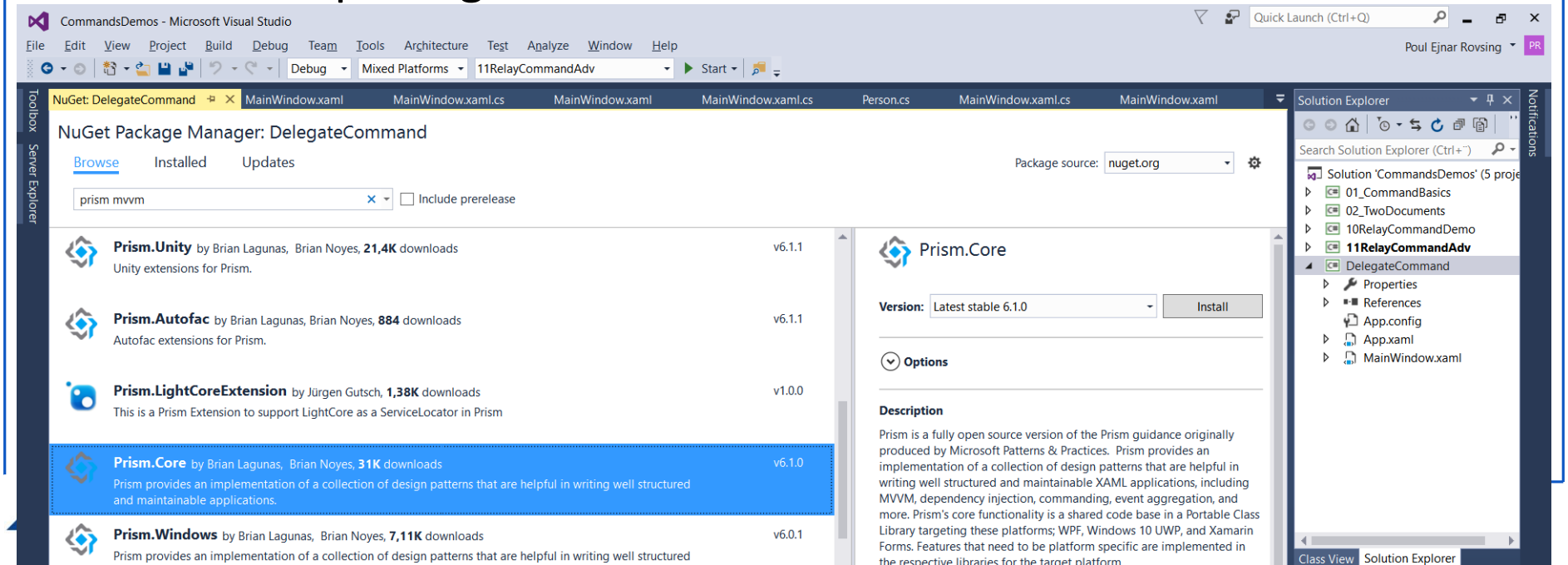
RelayCommand – With Lambda Expressions

- No initializing in constructor
- And inline cammandhandler with use of lambda expression

```
ICommand _nextCommand;
public ICommand NextCommand
{
  get
  {
    return _nextCommand ?? (_nextCommand = new RelayCommand(
           () => ++CurrentIndex,
           () => CurrentIndex < (Count - 1)));
  }
}
```

DelegateCommand

- DelegateCommand is another implementatino of the Icommand interface – very similar to RelayCommand
- DelegateCommand was originally developed by Microsoft's Pattern & Practices group but is now maintained by an open source project: https://github.com/PrismLibrary/Prism
- From ver. 6.0 of the Prism library DelegateCommand is distributed in a NuGet package called Prism.Core



Use of DelegateCommand

- Another name, another namespace
  - Otherwise business as usual

```
using Prism.Commands;
using System.ComponentModel;
using System.Windows;
using System.Windows.Input;

namespace DelegateCommandDemo
{
  public class Person : INotifyPropertyChanged
  {
    public ICommand BirthdayCommand { get; private set; }

    public Person()
    {
      BirthdayCommand = new DelegateCommand(BdayCmdHandler);
    }
```

Event To Command

- MVVM Light has an `EventToCommand` helper

```
xmlns:cmd="http://www.galasoft.ch/mvvmlight"

<cmd:EventToCommand Command="{Binding ScrollCommand}"
  PassEventArgsToCommand="True"/>
```

- And PRISM has `InvokeCommandAction`

```
 <i:Interaction.Triggers>
      <i:EventTrigger EventName="SelectionChanged">
         <prism:InvokeCommandAction
          Command="{Binding CustomersSelectedCommand}" />
      </i:EventTrigger>
   </i:Interaction.Triggers>
```

References

- MacDonald chapter 9 Commands

- Understanding Routed Commands
  by Josh Smith
  http://joshsmithonwpf.wordpress.com/2008/03/18/understanding-routed-commands/

- Using **RelayCommands** in Silverlight and WPF
  by Laurent Bugnion
  http://blog.galasoft.ch/archive/2009/09/26/using-relaycommands-in-silverlight-and-wpf.aspx

- Prism5forWPF.pdf: Commands page 87 to 90