The Core Components

Laurent Bugnion
@LBugnion
http://www.galasoft.ch

Outline

- What is MVVM Light and What is it not?
- The ObservableObject and the ViewModelBase
- Simplifying Commands with RelayCommand
- Sending messages with the Messenger
- Dispatching to the UI thread with the DispatcherHelper
- Summary

What is MVVM Light?
What is it Not?

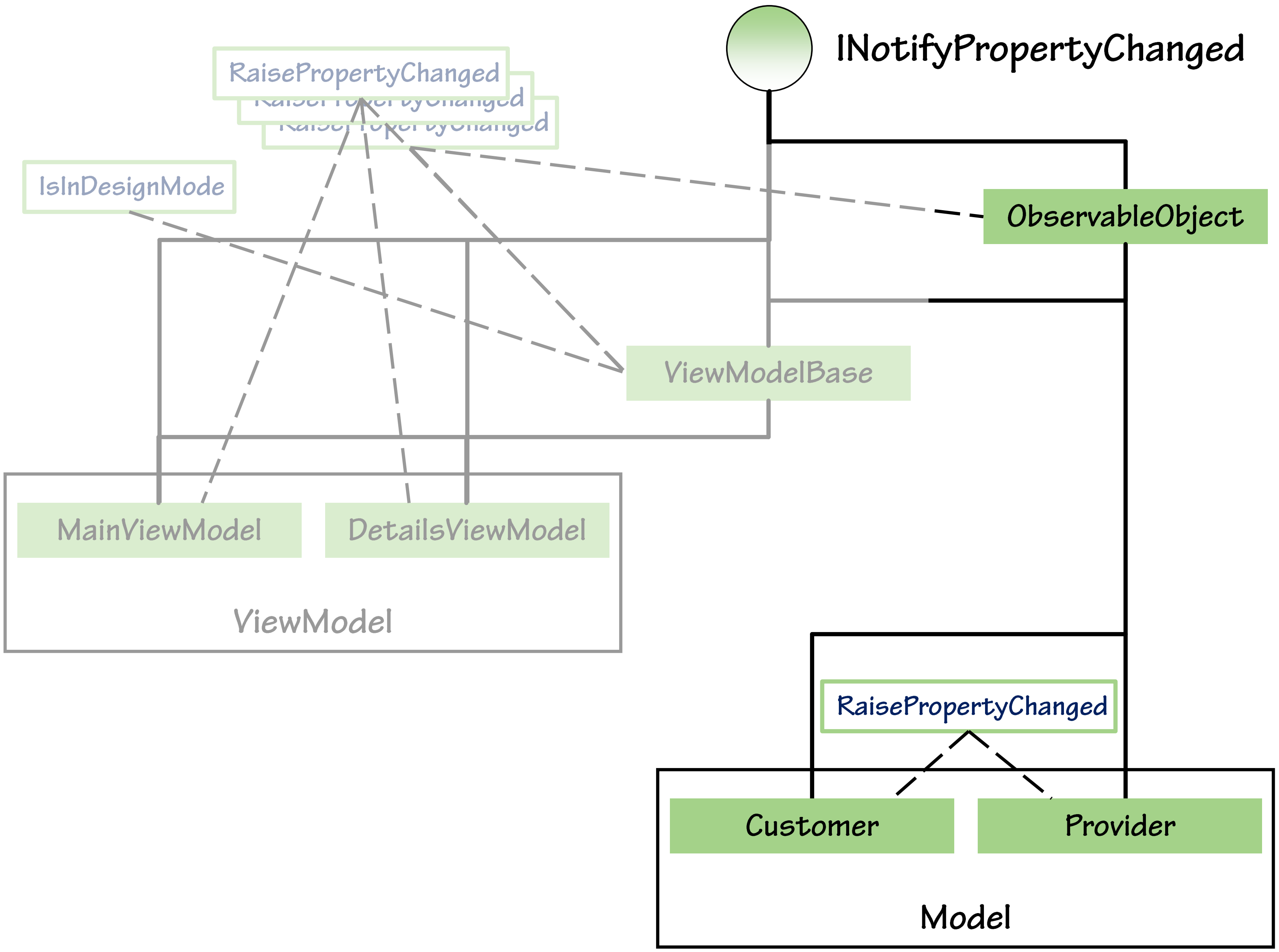
pluralsight
hardcore developer training

- **MVVM Light is a toolkit, a suite of tools**
  - Two DLLs, Project templates, Item templates, Code snippets

- **Helpers to help you code faster**
- **Helpers to avoid repetition and basic opreations**
- **Not only useful for XAML**

- **MVVM Light is not a framework**
  - Doesn’t require you to follow a specific architecture
  - Pick what you like, leave what you don’t

The ObservableObject and the ViewModelBase


INotifyPropertyChanged
RaisePropertyChanged
RaisePropertyChanged
RaisePropertyChanged
IsInDesignMode
ObservableObject
ViewModelBase
MainViewModel
DetailsViewModel
ViewModel
RaisePropertyChanged
Customer
Provider
Model

Raising the PropertyChanged Event

- Event raise (ObservableObject and ViewModelBase)

```
RaisePropertyChanged("MyProperty");
// "Classic" way.
// Typically used with a constant for the property's name.

RaisePropertyChanged(() => MyProperty);
// Supports Intellisense and automatic refactoring.
// Very, very small performance impact.

Set("MyProperty", ref _myProperty, value);

Set(() => MyProperty, ref _myProperty, value);
// Set method takes care of checking if event must be raised.
// Returns true if event was raised.
```

Raising the PropertyChanged Event

- Event raise and broadcasting through Messenger (ViewModelBase)

```
RaisePropertyChanged("MyProperty", oldValue, value, true);

RaisePropertyChanged(() => MyProperty, oldValue, value, true);

Set("MyProperty", ref _myProperty, value, true);

Set(() => MyProperty, ref _myProperty, value, true);
```

- Sends a PropertyChangedMessage (see module about Messenger)

Raising the PropertyChanging Event

- Event raise (ObservableObject and ViewModelBase)

```
RaisePropertyChanging("MyProperty");
// "Classic" way.
// Typically used with a constant for the property's name.

RaisePropertyChanging(() => MyProperty);
// Supports Intellisense and automatic refactoring.
// Very, very small performance impact.
```

- PropertyChanging is automatically raised by the Set method

The IsInDesignMode Property

- Different ways to check for design mode
- Silverlight, Windows Phone

```
_isInDesignMode = DesignerProperties.IsInDesignTool;
```

- Windows Store:

```
_isInDesignMode = DesignMode.DesignModeEnabled;
```

- WPF:

```
var prop = DesignerProperties.IsInDesignModeProperty;
_isInDesignMode = (bool)DependencyPropertyDescriptor
    .FromProperty(prop, typeof(FrameworkElement))
    .Metadata.DefaultValue;
```

- MVVM Light (all XAML frameworks)

```
public bool IsInDesignMode
public static bool IsInDesignModeStatic
```

The RelayCommand

The ICommand Interface

- **ICommand interface**
  - Execute method
  - CanExecute method
  - CanExecuteChanged event

- **Lots of (unnecessary) work to implement for each functionality we want to expose**

- **Solution: The RelayCommand**

The RelayCommand: Summary

- **An ICommand implementation**
  - Takes a delegate for the Execute method (compulsory).
  - Takes a delegate for the CanExecute method (optional).
  - Has a RaiseCanExecuteChanged method.

- **Removes the need for an explicit implementation of ICommand**

- **“Relays” the execution of the command to some local methods**

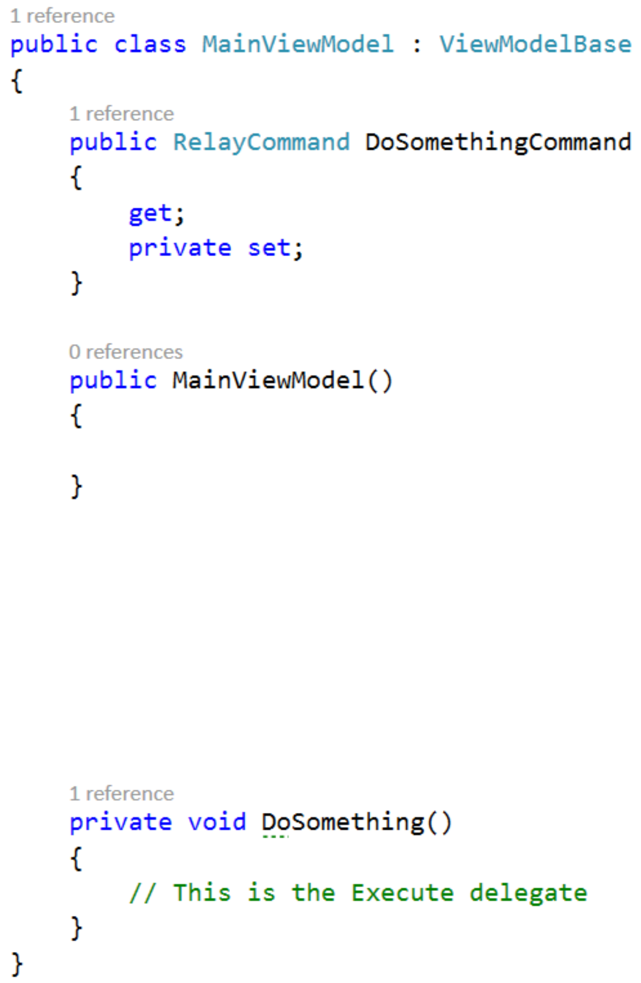
The RelayCommand

- **Constructor with a delegate for the Execute method**

```
1 reference
public class MainViewModel : ViewModelBase
{









}
```

The RelayCommand

- Constructor with two delegates for the Execute and CanExecute methods



```
1 reference
public class MainViewModel : ViewModelBase
{
    1 reference
    public RelayCommand DoSomethingCommand
    {
        get;
        private set;
    }

    0 references
    public MainViewModel()
    {

    }







    1 reference
    private void DoSomething()
    {
        // This is the Execute delegate
    }
}
```

The RelayCommand

- Both delegates can be lambda expressions

```
0 references
public MainViewModel()
{



















}
```

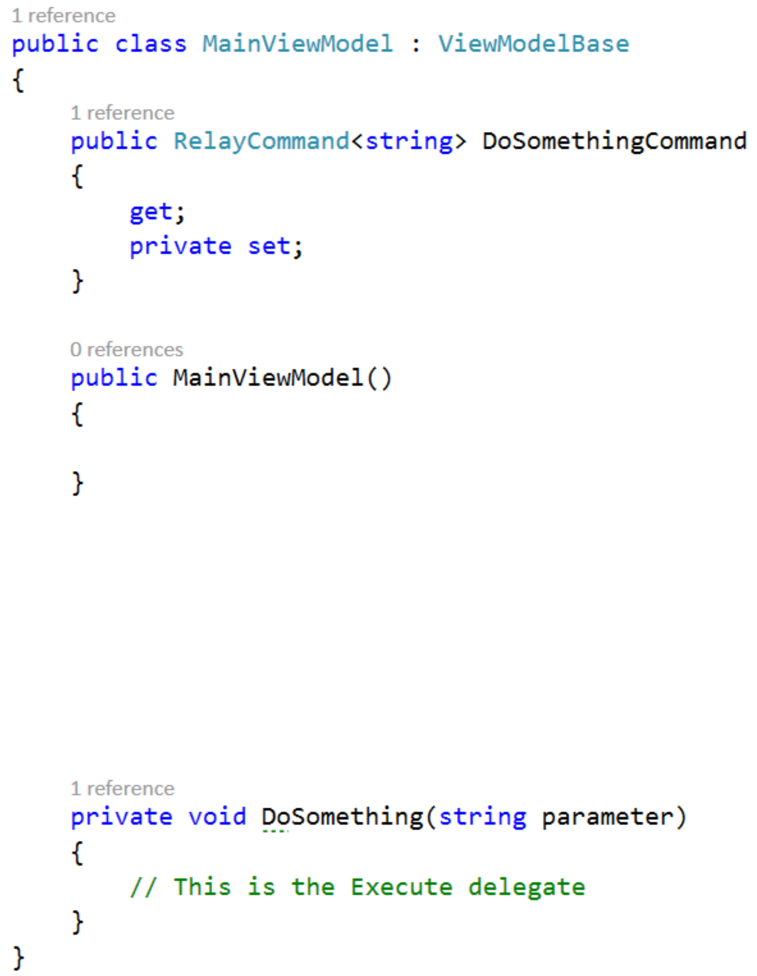
The RelayCommand<T>

- Constructor with a delegate for the Execute method

```
1 reference
public class MainViewModel : ViewModelBase
{
















}
```

The RelayCommand<T>

- **Constructor with two delegates for the Execute and CanExecute methods**



```
1 reference
public class MainViewModel : ViewModelBase
{
    1 reference
    public RelayCommand<string> DoSomethingCommand
    {
        get;
        private set;
    }

    0 references
    public MainViewModel()
    {

    }







    1 reference
    private void DoSomething(string parameter)
    {
        // This is the Execute delegate
    }
}
```

The RelayCommand<T>

- Both delegates can be lambda expressions

```
0 references
public MainViewModel()
{









}
```

Sending Messages and Loose Event Handling with the Messenger

The Issues with Conventional Event Handling

- **Attaching an event handler to a non-static method creates a strong link.**
  - If you forget, or cannot unregister the event, risk of memory leak.
- **Registering:**
  ```
  button.Click += ButtonClick;
  ```
- **Unregistering:**
  ```
  button.Click -= ButtonClick;
  ```
- **Another difficulty: Finding the right instance to register.**
  - Sometimes we don’t know who raises the event.
  - For instance: plug in scenarios.
  - Especially an issue in decoupled apps.

What does the Messenger Do?

- **It is an “event bus”.**
- **A message distribution system.**
  - One object broadcasts a message.
  - Other objects register to receive these messages.
  - The sender doesn’t know who receives the messages.
  - The receiver doesn’t know who sent the messages.
- **One default instance (Messenger.Default)**
  - but it is also possible to create as many Messenger instances as needed.

Registering for a Message

- **The receiver registers for a message type.**
  - Any types (even simple values) are supported

- **In addition, a “private channel” can be opened (see special cases).**

- **Typical registration:**

```
public MessageRecipient()
{
    Messenger.Default.Register<MyMessageType>(
        this, HandleMessage);
}

private void HandleMessage(MyMessageType message)
{
    // Do something
}
```

Registering for Messages (Examples)

- Using named methods or lambdas:

```
public MessageReceiver()
{








}
```

Sending a Message

- The receiver sends any object.

- Typical send:
```
var myInstance = new MyMessageType();
Messenger.Default.Send(myInstance);
```

- `with`
```
public class MyMessageType
{
    // Can be anything
}
```

Sending Messages (Examples)

- Anything can be sent:

The Weak Reference Issue

- Sometimes it is hard to unregister from the Messenger.
- The Messenger is optimized for these scenarios…
- …but there is only so much we can do!
- In some cases, you should really unregister!

Method
Static
Public
Internal
Private
Lambdas

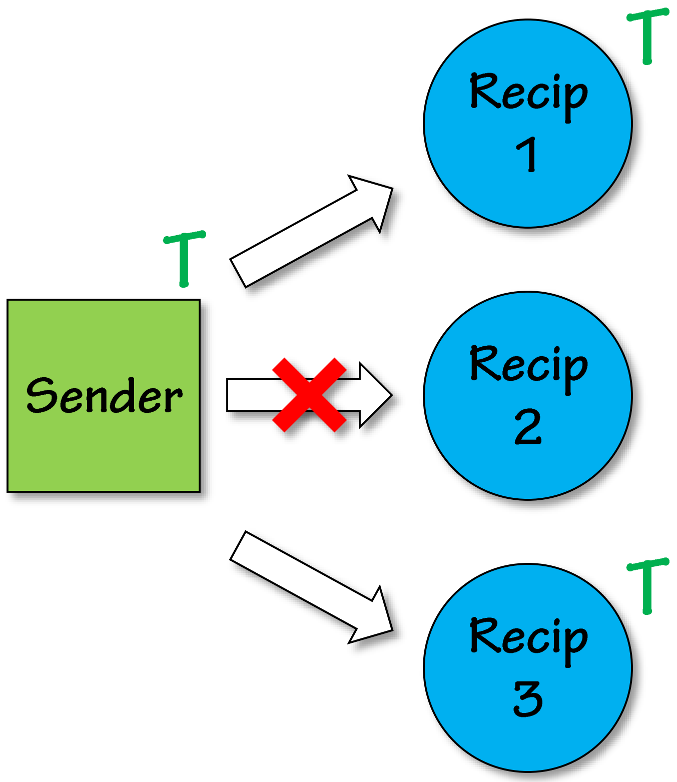
Unregistering (Examples)

- Always a good clean policy (if possible):

```
Messenger.Default.Unregister(this);

Messenger.Default.Unregister<IMessage>(this, HandleMessage);

Messenger.Default.Unregister<IMessage>(this, Token);

Messenger.Default.Unregister<IMessage>(this, Token, HandleMessage);
```

Special Cases

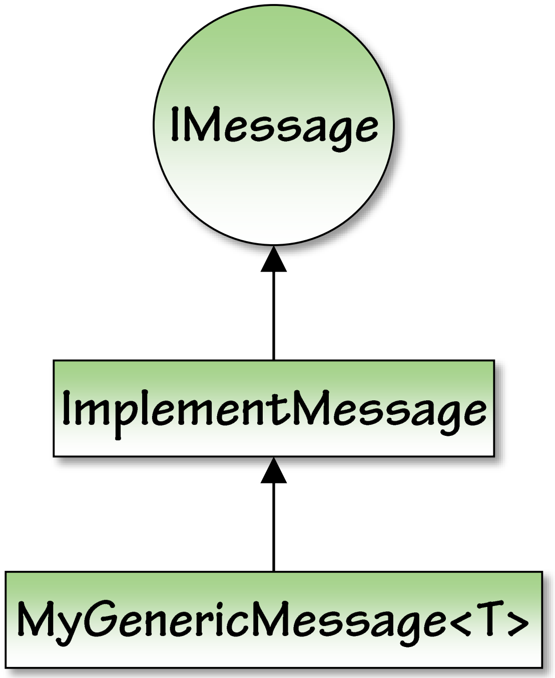
- Sending with a token.



Special Cases

- Registering for base classes.

```
Messenger.Default.Register<IMessage>(
    this,
    true,
    HandleMessage);
with
private void HandleMessage(IMessage message)
{
}


Messenger.Default.Send(new ImplementMessage());


Messenger.Default.Send(new MyGenericMessage<string>());
```



The Dangers of the Messenger

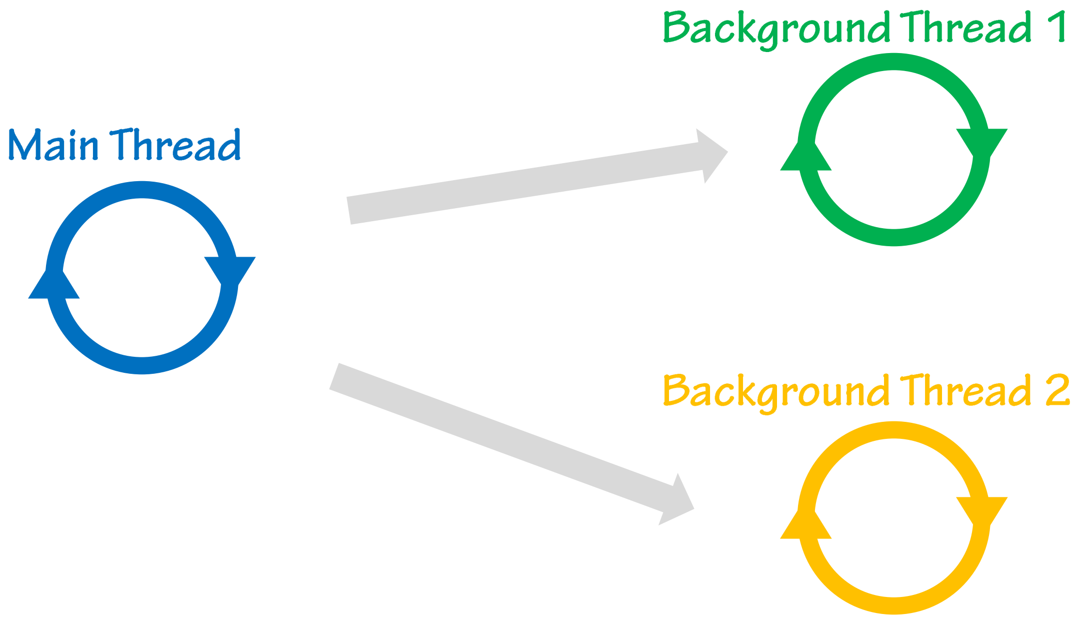
- **It can be tempting to overuse the Messenger.**
- **Can lead to confusing code.**
- **Be reasonable.**
  - Event handlers are OK sometimes too.
  - Often the Messenger can be replaced by a service (such as DialogService, NavigationService, etc)

- **Test your code for possible memory leaks.**
  - If unsure, Unregister!

Threading Made Easier with the DispatcherHelper

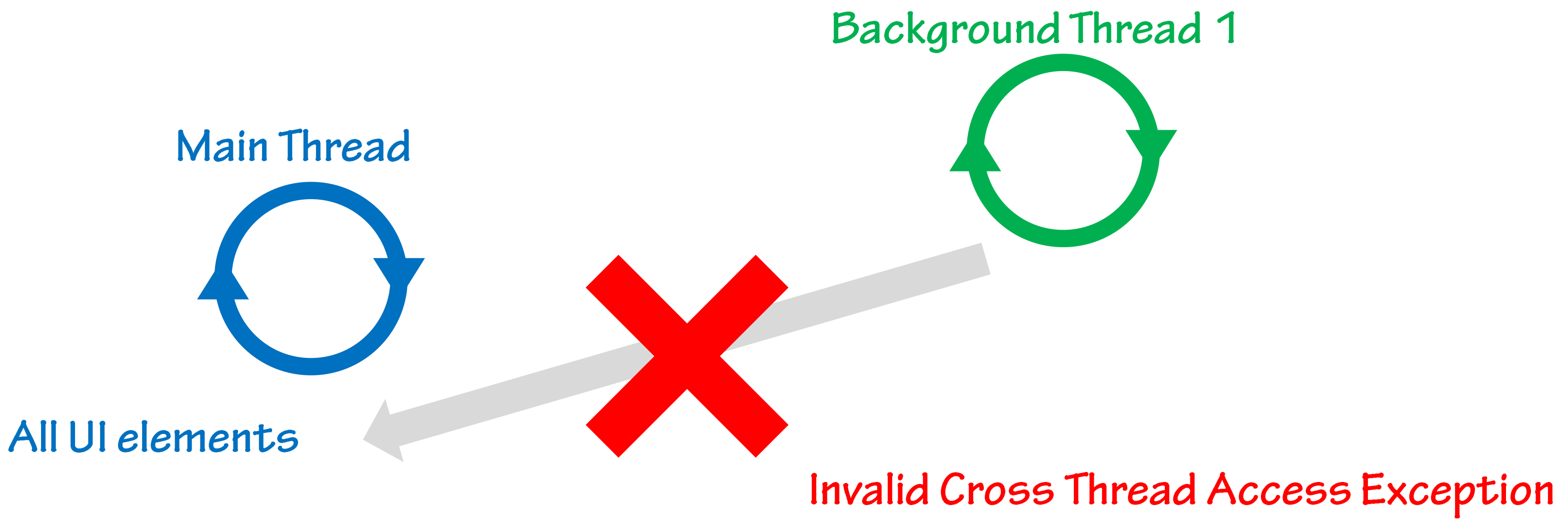
pluralsight
hardcore developer training

Threading in XAML Frameworks?



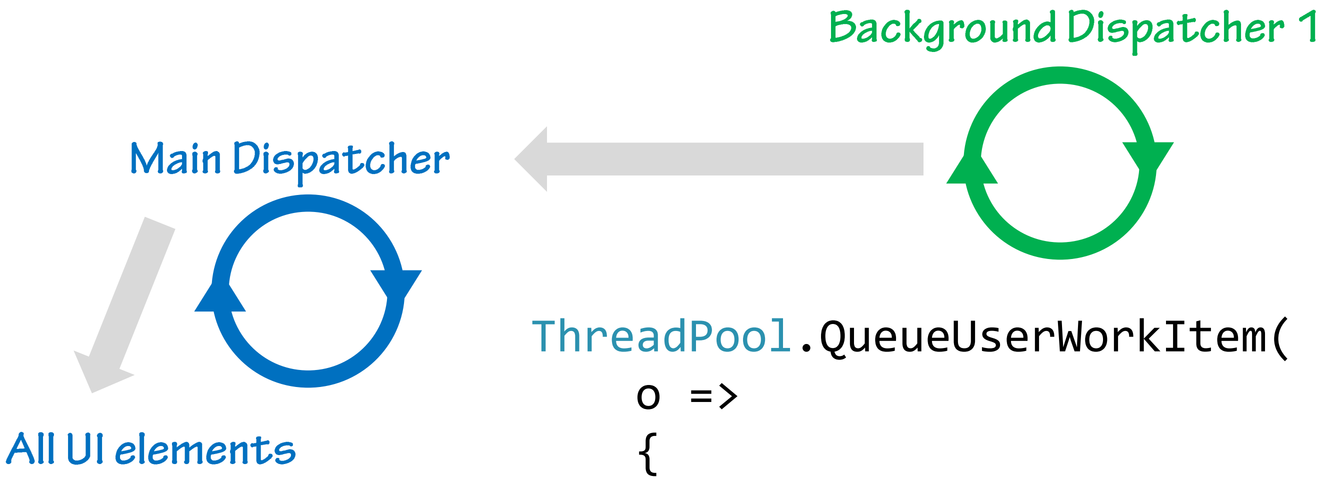
```
ThreadPool.QueueUserWorkItem(…)
(new BackgroundWorker()).RunWorkerAsync()
Sensors, web request...
```

Getting Back to the Main Thread



```
ThreadPool.QueueUserWorkItem(
    o =>
    {
        // Do something...

        StatusTextBlock.Text = "Done"; // CRASH
    });
```

Getting Back to the Main Thread



```
ThreadPool.QueueUserWorkItem(
    o =>
    {
        // Do something...

        Dispatcher.BeginInvoke(
            new Action<string>(UpdateStatus),
            "Done",
            DispatcherPriority.Normal);
    });
with
private void UpdateStatus(string message)
{
    StatusTextBlock.Text = message;
}
```

A Few Difficulties

- **The Dispatcher property is only available in the view.**
  - The ViewModel does not have direct access to the Dispatcher.
- **The syntax is cumbersome!**
  - And different in other XAML frameworks.
  - Windows 8 does not dispatch like Windows Phone, for instance
- **Solution: Using the DispatcherHelper**

The DispatcherHelper

- **Stores an instance of the main Dispatcher (UI Dispatcher).**
- **Needs to be initialized:**
  `DispatcherHelper.Initialize();`
- **Checking if dispatching needs to be done.**
  - Executes immediately if on main Thread
  - Executes with dispatching if on background thread

```
ThreadPool.QueueUserWorkItem(
    o =>
    {
        // Do something




    });
```

Summary

- What is MVVM Light and What is it not?
- The ObservableObject and the ViewModelBase
- Simplifying Commands with RelayCommand
- Sending messages with the Messenger
- Dispatching to the UI thread with the DispatcherHelper