DEVHOL202 – Curing the asynchronous blues with the Reactive Extensions for .NET (C#)

Introduction

This Hands-on-Lab (HOL) familiarizes the reader with the Reactive Extensions for .NET (Rx). By exploring the framework through a series of incremental samples, the reader gets a feel for Rx's compositional power used to write asynchronous applications, based on the concept of observable collections.

Prerequisites

We assume the following intellectual and material prerequisites in order to complete this lab successfully:

- Active knowledge of .NET and the C# programming language.
- Feeling for the concept of asynchronous programming and related complexities.
- Visual Studio 2010 and .NET 4 (prior versions can be used but the lab is built for VS2010) installation.
- Installation of Rx for .NET 4 from MSDN Data Developer Center at http://msdn.microsoft.com/en-us/data/gg577609.

What is Rx?

Rx can be summarized in the following sentence which can also be read on the Data Developer Center homepage:

Rx is a library for composing asynchronous and event-based programs using observable collections.

Three core properties are reflected in here, all of which will be addressed throughout this lab:

- **Asynchronous and event-based** – As reflected in the title, the bread and butter of Rx's mission statement is to simplify those programming models. Everyone knows what stuck user interfaces look like, both on the Windows platform and on the web. And with the cloud around the corner, asynchrony becomes quintessential. Low-level technologies like .NET events, the asynchronous pattern, tasks, AJAX, etc. are often too hard.
- **Composition** – Combining asynchronous computations today is way too hard. It involves a lot of plumbing code that has little to nothing to do with the problem being solved. In particular, the data flow of the operations involved in the problem is not clear at all, and code gets spread out throughout event handlers, asynchronous callback procedures, and whatnot.
- **Observable collections** – By looking at asynchronous computations as data sources, we can leverage the active knowledge of LINQ's programming model. That's right: your mouse is a database of mouse moves and clicks. In the world of Rx, such asynchronous data sources are composed using various operators in the LINQ sense, allowing things like filters, projections, joins, time-based operations, etc.

Lab flow

In this lab, we'll explore Rx in a gradual manner. First, we'll have a look at the **core interfaces** of Rx which ship in .NET 4's Base Class Library. Once we understand those interfaces (and their relationship with IEnumerable<T>), we'll move on to show how the Rx libraries allow for creating **simple observable sequences** using factory methods. This allows for some

basic experimentation. As we proceed, we'll introduce how to **bridge with existing asynchronous event sources** such as .NET events and the asynchronous pattern. Showing **more query operators** as we move along, we'll end up at a point where we start to **compose multiple asynchronous sources**, unleashing the true power of Rx.

Exercise 1 – Getting started with Rx interfaces and assemblies

Objective: Acquiring basic knowledge of the IObservable<T> and IObserver<T> interfaces that ship in the .NET 4 BCL (Base Class Library) and the role of the Rx release System.Reactive assembliy.

1. Open Visual Studio 2010 and go to File, New, Project... to create a new Console Application project in C#. Make sure the .NET Framework 4 target is set in the dropdown box at the top of the dialog.
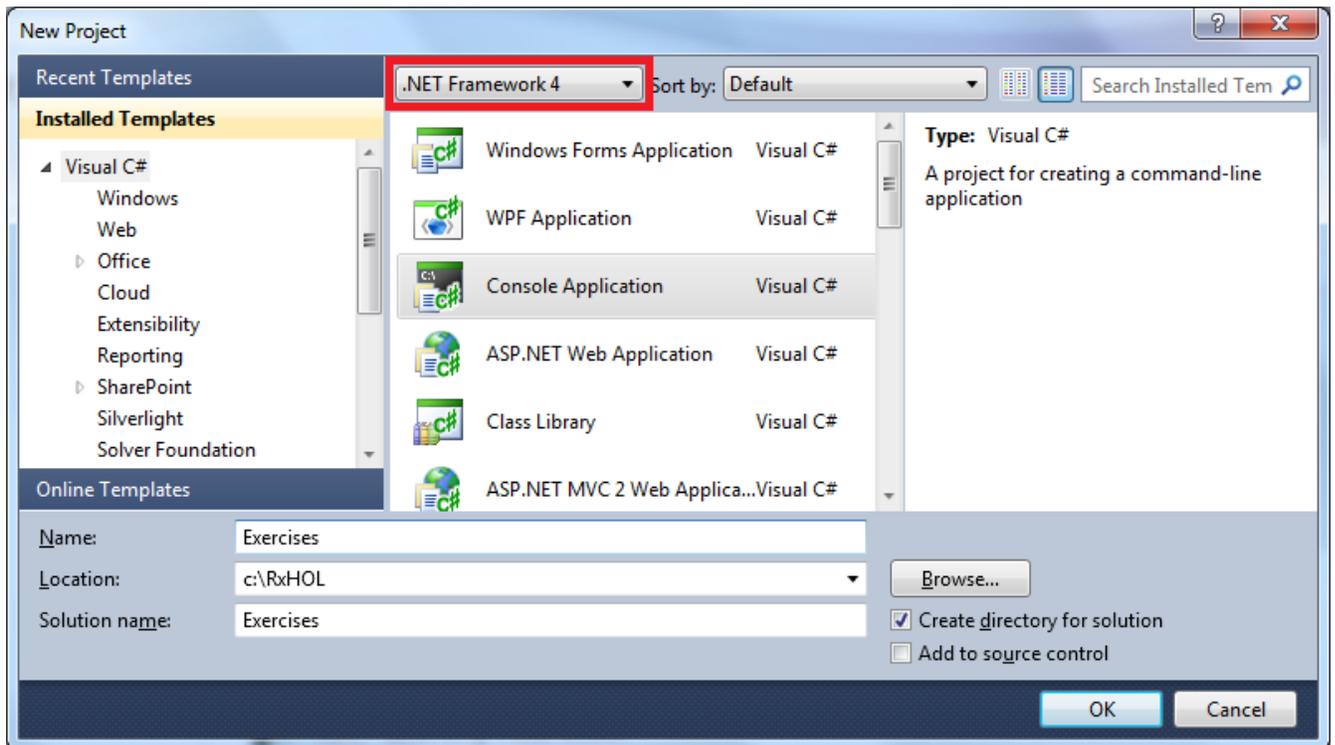
2. In the Program.Main method in the Program.cs source file, explore the System namespace by typing the following fragment of code:

```
System.IObs
```

This shows the two core interfaces around which Rx is built. Starting from .NET 4, those interfaces are built in to the Base Class Library's mscorlib.dll assembly.

Note: On other platforms supported by Rx (including .NET 3.5 SP1 and Silverlight), a separate assembly called System.Observable.dll has to be included in the project in order to get access to those two interfaces. This assembly gets installed in the same location as the other assemblies we'll talk about further on in this first exercise.

3. To start our exploration of those interfaces, we should have a look at them. Use the Object Browser (available through the View menu) or use both interfaces in a piece of code and use Go to Definition from the context menu (or press F12). We'll follow the latter route here as it allows to indicate the role of both interfaces in an informal manner using variable names:

```csharp
IObservable<int> source;
IObserver<int> handler;
```

The interfaces should look as shown below:

```csharp
public interface IObservable<out T>
{
    IDisposable Subscribe(IObserver<T> observer);
}

public interface IObserver<in T>
{
    void OnCompleted();
    void OnError(Exception error);
    void OnNext(T value);
}
```

Both interfaces serve a complementary role. The IObservable<T> interface acts as a data source that can be *observed*, meaning it can send data to everyone who's interested to hear about it. Those interested parties are represented by the IObserver<T> interface.

In order to receive notifications from an observable collection, one uses the Subscribe method to hand it an IObserver<T> object. In return for this observer, the Subscribe method returns an IDisposable object that acts as a handle for the subscription. Calling Dispose on this object will detach the observer from the source such that notifications are no longer delivered. Similarities with the += and -= operators used for .NET events are not accidental, but the Subscribe method provides more flexibility. In particular, an unsubscribe action can result in quite some bookkeeping, all of which is handled by the Rx framework.

Observers support three notifications, reflected by the interface's methods. OnNext can be called zero or more times, when the observable data source has data available. For example, an observable data source used for mouse move events could send out a Point object every time the mouse has moved. The other two methods are used to signal successful or exceptional termination.

Background: Those two interfaces are the *dual to IEnumerable<T> and IEnumerator<T>*. While this deep duality may sound frightening, it's a source of much beauty. First of all, the property of duality between those interfaces can be explained in two simple English words: *push versus pull*. Where an observable data source pushes data at its observers, enumerable data sources are being pulled by an enumerator to receive data (typically using the foreach language construct). As a result of this duality, all of the LINQ operators that apply in one world have a corresponding use in the other world. We'll see this illustrated in later exercises.

4. Even though we've shown those interfaces, we'll resist the temptation to implement them. Instead, the next exercise will explain how to create observable sequences using functionality in the Rx library. To recap the usage, we'll start by writing a bit of non-functional code (indeed, this <u>won't work</u> just yet):

```csharp
IDisposable subscription = source.Subscribe(handler);

Console.WriteLine("Press ENTER to unsubscribe...");
Console.ReadLine();

subscription.Dispose();
```
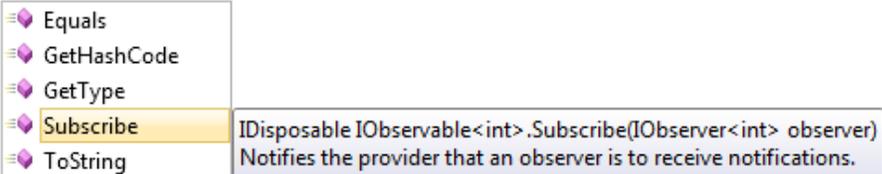
5. While the above shows a typical use pattern of observable sequences, you may have noticed there's little one can do with an IObservable<T> object, other than subscribing to it:
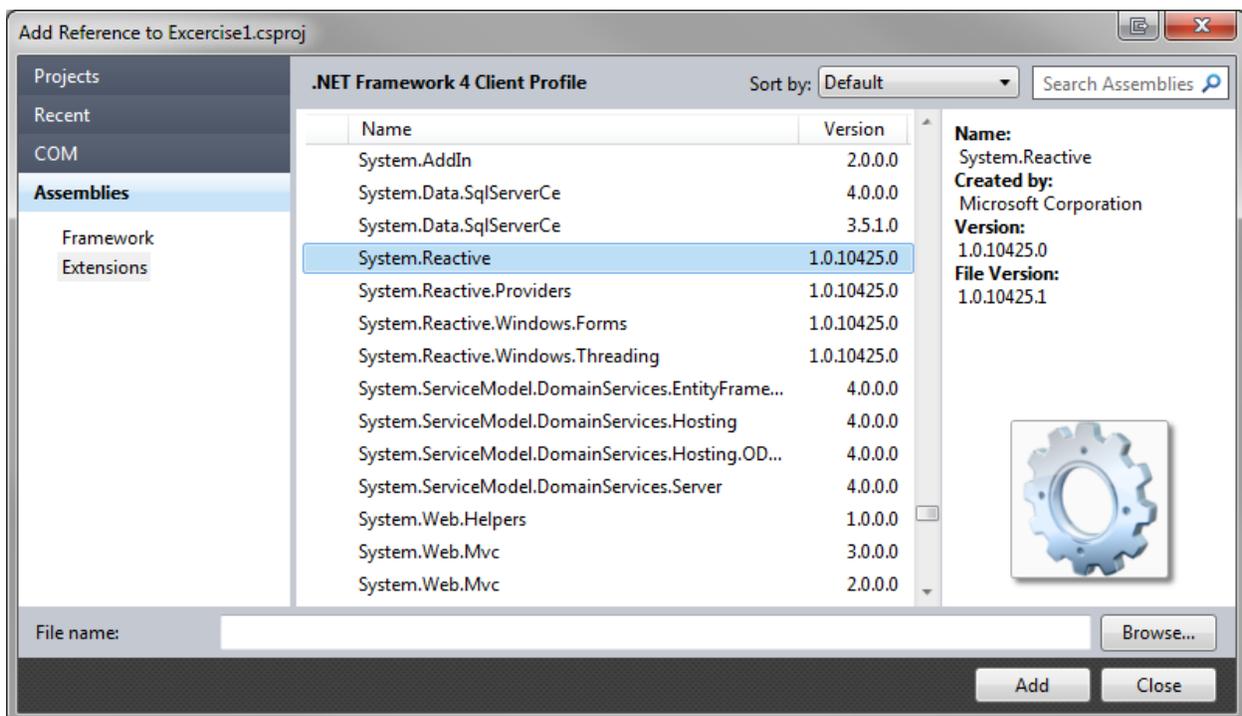
```csharp
IDisposable subscription = source.|
                                    ≡♦ Equals
Console.WriteLine("Press ENTER to     ≡♦ GetHashCode
Console.ReadLine();                    ≡♦ GetType
                                       ≡♦ Subscribe        IDisposable IObservable<int>.Subscribe(IObserver<int> observer)
subscription.Dispose();                ≡♦ ToString         Notifies the provider that an observer is to receive notifications.
```

6. In order to make observable sequences more useful, the Rx library provides a whole bunch of operators to apply operations to observable sequences and to compose them. So, while the .NET 4 BCL ships bare bones interfaces, Rx provides rich functionality for them. In order to leverage those, go to the Solution Explorer, right click on the project's node, choose Add Reference… In there, you should find System.Reactive under the Assemblies\Extensions menu or by using the Search Assemblies options.

Add Reference to Excercise1.csproj				
Projects	**.NET Framework 4 Client Profile**	Sort by: Default ▼	Search Assemblies 🔎	
Recent	Name	Version	**Name:** System.Reactive	
COM	System.AddIn	2.0.0.0	**Created by:** Microsoft Corporation	
Assemblies	System.Data.SqlServerCe	4.0.0.0	**Version:** 1.0.10425.0	
Framework	System.Data.SqlServerCe	3.5.1.0	**File Version:** 1.0.10425.1	
Extensions	System.Reactive	1.0.10425.0		
	System.Reactive.Providers	1.0.10425.0		
	System.Reactive.Windows.Forms	1.0.10425.0		
	System.Reactive.Windows.Threading	1.0.10425.0		
	System.ServiceModel.DomainServices.EntityFrame...	4.0.0.0		
	System.ServiceModel.DomainServices.Hosting	4.0.0.0		
	System.ServiceModel.DomainServices.Hosting.OD...	4.0.0.0		
	System.ServiceModel.DomainServices.Server	4.0.0.0		
	System.Web.Helpers	1.0.0.0		
	System.Web.Mvc	3.0.0.0		
	System.Web.Mvc	2.0.0.0		
File name:			Browse...	
			Add	Close

Add the System.Reactive assembly reference to your project. In this lab, we won't cover the System.Reactive.Providers, System.Reactive.Windows.Forms and System.Reactive.Windows.Threading

assemblies.

7. The System.Reactive assembly is what contains the operators for observable sequences, implemented as extension methods as we shall see in just a moment. Since many of those operators need to introduce concurrency (a necessary evil in the face of asynchronous programming), the notion of a **scheduler** was introduced. To achieve architectural layering, target specific scheduler functionality for Windows Forms and WPF resides in target specific assemblies: System.Reactive.Windows.Forms and System.Reactive.Windows.Threading. Later on, we'll see operators like ObserveOn that use schedulers.

8. With the assembly reference added, we should now see more methods on IObservable<T> objects. Since those are extension methods, a little experiment will reveal two buckets of additional methods. Eliminate all of the namespace imports other than System and observe the IntelliSense auto-completion list on IObservable<T>:

```csharp
Excercise1.Program
1    using System;
2
3    namespace Excercise1
4    {
5        class Program
6        {
7            static void Main(string[] args)
8            {
9                IObservable<int> source;
10               IObserver<int> handler;
11
12               IDisposable subscription = source.S
13                                                        ≡♦ Equals
14               Console.WriteLine("Press ENTER to     ≡♦ GetHashCode
15               Console.ReadLine();                   ≡♦ GetType
16                                                     ≡♦ Subscribe
17               subscription.Dispose();               ♦ Subscribe<>
18           }                                         ≡♦ ToString
19       }
20   }
21
```

Notice the Subscribe extension methods being added through the System namespace. Those overloads allow one to avoid implementing the IObserver<T> interface at all, since one can specify any of the three handler methods (OnNext, OnError, OnCompleted) using delegates. For example:

```csharp
IDisposable subscription = source.Subscribe(
```

```
        (int x) => {
            Console.WriteLine("Received {0} from source.", x);
        },
        (Exception ex) => {
            Console.WriteLine("Source signaled an error: {0}.", ex.Message);
        },
        () => {
            Console.WriteLine("Source said there are no messages to follow anymore.");
        }
    );
```

Exercise: It's left as an exercise for the reader to eliminate any excessive syntax in the sample above, such as types than can be inferred, redundant curly braces, etc.

9. To see operators that can be applied to IObservable<T> objects, add a **using** directive to import the **System.Reactive.Linq** namespace. "Dot into" our source object again, this time revealing a whole bunch of operators. For those familiar with LINQ, try finding some of your favorite operators such as Where, Select, etc.

Conclusion: The IObservable<T> and IObserver<T> interfaces represent a data source and a listener, respectively. In order to observe an observable sequence's notifications, one gives it an observer object using the Subscribe method, receiving an IDisposable object that be used to unsubscribe. While those interfaces are in the .NET 4 BCL, they're only available through a System.Observable assembly on other platforms. To enable rich functionality over observable sequences (as we'll discuss thoroughly in what follows), System.Reactive provides a series of extension methods that can be imported through the System and System.Reactive.Linq namespaces.

Exercise 2 – Creating observable sequences

Objective: Observable sequences are rarely provided by implementing the IObservable<T> interface directly. Instead a whole set of factory methods exist that create primitive observable sequences. Those factory methods provide a good means for initial exploration of the core notions of observable sources and observers.

1. Ensure the project setup of Exercise 1 is still intact, i.e. the reference to System.Reactive is in place and both the System and System.Reactive.Linq namespaces are imported in your Program.cs file. Also ensure the following skeleton code is still present:

   ```
   IObservable<int> source = /* We'll explore a set of factory methods here */;
   IDisposable subscription = source.Subscribe(
       x  => Console.WriteLine("OnNext:  {0}", x),
       ex => Console.WriteLine("OnError: {0}", ex.Message),
       () => Console.WriteLine("OnCompleted")
   );
   ```

2. In the above, we'll substitute the comment for various primitive sources and observe their behavior. We'll also contrast those with enumerable sequences. Since observable sequences sometimes introduce concurrency to pump out their notifications (required for asynchrony), we should prevent the program from quitting while the subscription is active. We'll hold the main thread by using Console.ReadLine to do so prior to calling Dispose on the subscription:

   ```
   Console.WriteLine("Press ENTER to unsubscribe...");
   Console.ReadLine();

   subscription.Dispose();
   ```

3. We'll start by looking at the Empty factory method:

   ```
   IObservable<int> source = Observable.Empty<int>();
   ```

   Running this code produces the following output:

   ```
   OnCompleted
   ```

   In order words, the empty sequence simply signals completion to its observers by calling OnCompleted. This is very similar to LINQ to Object's Enumerable.Empty or an empty array (e.g. new int[0]). For those enumerable sequences, the first call to the enumerator's MoveNext method will return false, signaling completion.

   Background: One may wonder when observable sequences start running. In particular, what's triggering the empty sequence to fire out the OnCompleted message to its observers? The answer differs from sequence to sequence. Most of the sequences we're looking at in this exercise are so-called *cold observables* which means they start running upon subscription. This is different from *hot observables* such as mouse move events which are flowing even before a subscription is active (there's no way to keep the mouse from moving after all…).

4. Besides the OnCompleted message, OnError is also a terminating notification, in a sense no messages can follow it. Where Empty is the factory method creating an observable sequence that immediately triggers completion, the Throw method creates an observable sequence that immediately triggers an OnError message to observers:

    ```
    IObservable<int> source = Observable.Throw<int>(new Exception("Oops"));
    ```

    Running this code produces the following output:

    ```
    OnError: Oops
    ```

Background: The OnError message is typically used by an observable sequence (not as trivial as the one simply returned by a call to Throw) to signal an error state which could either originate from a failed computation or the delivery thereof. Following the semantic model of the CLR's exception mechanism, errors in Rx are always terminating and exhibit a fail-fast characteristic, surfacing errors through observer handlers. More advanced mechanisms to deal with errors exist in the form of handlers called Catch, OnErrorResumeNext and Finally. We won't discuss those during this HOL, but their role should be self-explanatory based on corresponding language constructs in various managed languages.

5. One final essential factory method or primitive constructor is called Return. Its role is to represent a single-element sequence, just like a single-cell array would be in the world of IEnumerable sequences. The behavior observed by subscribed observers is two messages: OnNext with the value and OnCompleted signaling the end of the sequence has been received:

    ```
    IObservable<int> source = Observable.Return(42);
    ```

    Running this code produces the following output:

    ```
    OnNext: 42
    OnCompleted
    ```

Background: Return plays an essential role in the theory behind LINQ, known as *monads*. Together with an operator called SelectMany (which we'll learn about more later on), they form the primitive functions needed to leverage the power of monadic computation. More information can be found by searching the web using monad and functional as the keywords.

6. At this point, we've seen the most trivial observable sequence constructors that are intimately related to an observer's triplet of methods. While those are of interest in certain cases, more meaty sequences are worth to explore as well. The Range operator is just one operator that generates such a sequence. Symmetric to the same operator on Enumerable, Range does return a sequence with 32-bit integer values given a starting value and a length:

    ```
    IObservable<int> source = Observable.Range(5, 3);
    ```

    Running this code produces the following output:

```
OnNext: 5
OnNext: 6
OnNext: 7
OnCompleted
```

Note: As with all the sequences mentioned in this exercise, Range is a *cold observable*. To recap, this simply means that it starts producing its results to an observer upon subscription. Another property of cold observable sequence is that every subscription will cause such reevaluation. Thus, if two calls to Subscribe are made, both of the observers passed to those calls will receive the messages from the observable. It's not because the data observation has run to completion for one observer that other observers won't run anymore. Whether or not the produced data is the same for every observer depends on the characteristics of the sequence that's being generated. For deterministic and "purist functional" operators like Return and Range, the messages delivered to every observer will be the same. However, one could imagine other kinds of observable sequences that depend on side-effects and thus deliver different results for every observer that subscribes to them.

7. To generalize the notion of sequence creation in terms of a generator function, the Generate constructor function was added to Rx. It closely resembles the structure of a for-loop as one would use to generate an enumerable sequence using C# iterators (cf. the "yield return" and "yield break" statements). To do so, it takes a number of delegate-typed parameters that expect function to check for termination, to iterate one step and to emit a result that becomes part of the sequence and is sent to the observer:

```
IObservable<int> source = Observable.Generate(0, i => i < 5, i => i + 1, i => i * i);
```

Running this code produces the following output:

```
OnNext: 0
OnNext: 1
OnNext: 4
OnNext: 9
OnNext: 16
OnCompleted
```

Note: A sister function called GenerateWithTime exists that waits a computed amount of time before moving on to the next iteration. We'll look at this one in just a moment.

8. One operator that may seem interesting from a curiosity point of view only is called Never. It creates an observable sequence that will never signal any notification to a subscribed observer:

```
IObservable<int> source = Observable.Never<int>();
```

Running this code shouldn't produce any output till the end of mankind.

Background: One reason this operator has a role is to reason about *composition* with other sequences, e.g. what would it mean to concatenate a finite and an infinite sequence? Should the result also exhibit an infinite characteristic? Those answers are essential to ensuring all of the Rx semantics make sense and are consistent throughout various operators. Besides the operator's theoretical use, there are real use cases for it as well. For example, you may use it to ensure a sequence never terminates by avoiding an OnCompleted handler getting

triggered. Another use is to test whether an application does not hang in the presence of non-termination.

9. Where the Subscribe method has an asynchronous characteristic, other blocking operators exist to observe a sequence in a synchronous manner. One such operator is called **ForEach**.

First of all, let's formalize "asynchronous". It's near to impossible to say something is asynchronous in itself without mentioning two parties. In particular, the thread calling the Subscribe method on an observable sequence is not blocked till the sequence runs to completion, which may happen on another thread. That is, Subscribe is asynchronous in that the caller is not blocked till the observation of the sequence completes.

For demos and testing it's often useful to perform a "blocking subscribe" such that the caller is blocked till the observable sequence triggers OnError or OnCompleted to the observer. That's what ForEach is all about:

```csharp
IObservable<int> source = Observable.Range(0, 10);
source.ForEach(
    x  => Console.WriteLine("OnNext:  {0}", x),
    ex => Console.WriteLine("OnError: {0}", ex.Message),
    () => Console.WriteLine("OnCompleted")
);
```

In here, we won't get to the next statement till the sequence completes gracefully or exceptionally.

10. Finally, let's inspect the behavior of an observable sequence by looking at it through the lenses of the debugger in Visual Studio. We'll use the following fragment to do so:

```csharp
IObservable<int> source = Observable.Generate(
        0, i => i < 5,
        i => i + 1,
        i => i * i,
        i => TimeSpan.FromSeconds(i)
);

using (source.Subscribe(
        x  => Console.WriteLine("OnNext:  {0}", x),
        ex => Console.WriteLine("OnError: {0}", ex.Message),
        () => Console.WriteLine("OnCompleted")
    ))
{
    Console.WriteLine("Press ENTER to unsubscribe...");
    Console.ReadLine();
}
```
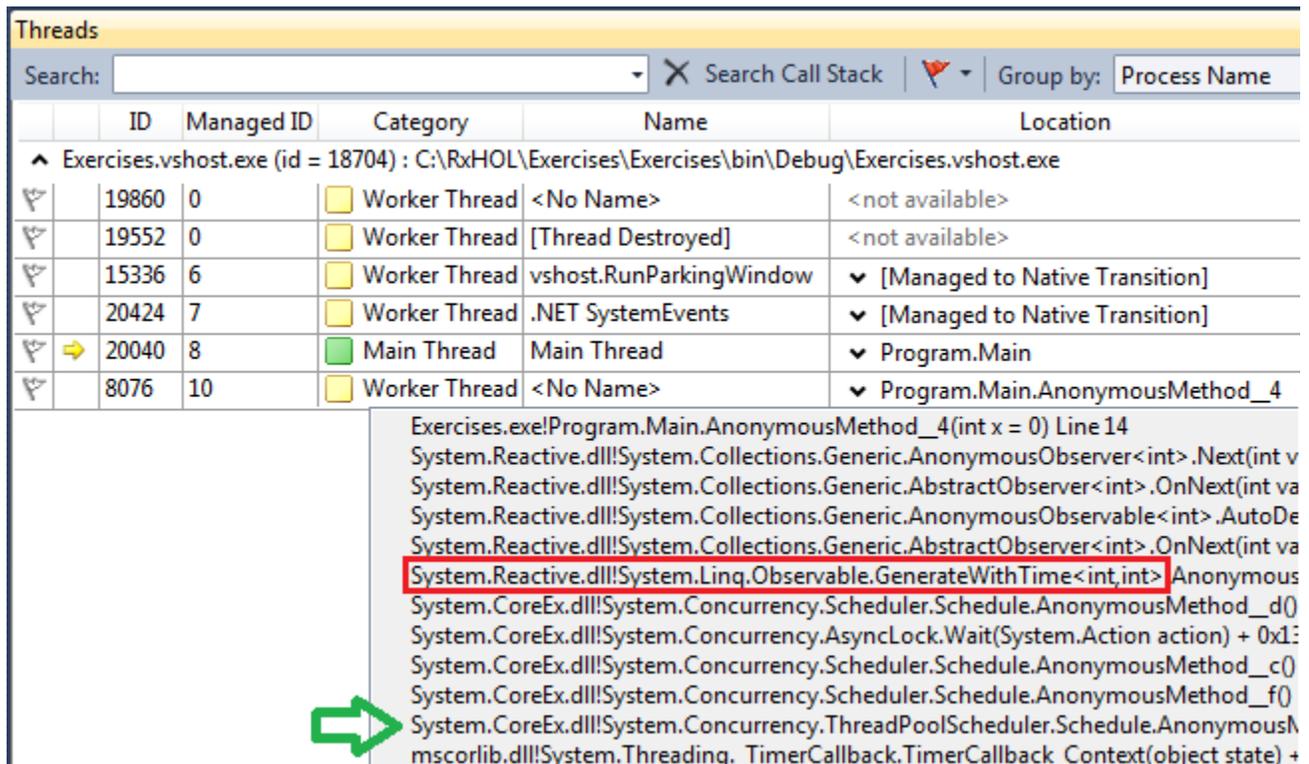
This sample uses a slightly different operator called Generate that allows specifying iteration time between producing results, dependent on the loop variable. In this case, 0 will be produced upon subscription, followed by 1 a second later, then 4 two seconds later, 9 three seconds later and 16 four seconds later. Notice how the notion of time – all important in an asynchronous world – is entering the picture here.

Note: Typically you won't immediately dispose a subscription by means of a using block. In this sample case it works fine since the code block blocks for user input. In most cases you'll store the IDisposable object in order to dispose it at a later time, or even don't bother to dispose it (e.g. for infinite sequences like timers).

a. Set a breakpoint on the highlighted lambda expression body using F9. Notice you need to be inside the lambda body with the cursor in the editor in order for the breakpoint to be set on the body and not the outer method call to Subscribe.

```
            using (source.Subscribe(
                    x => Console.WriteLine("OnNext:    {0}", x),
                    ex => Console.WriteLine("OnError: {0}", ex.Message),
                    () => Console.WriteLine("OnCompleted")
                ))
            {
                Console.WriteLine("Press ENTER to unsubscribe...");
```

b. Start running the program by pressing F10 and step down in the code using F10. You'll not see the breakpoint being hit just yet and will effectively reach the Console.ReadLine call:

```
            using (source.Subscribe(
                    x => Console.WriteLine("OnNext:    {0}", x),
                    ex => Console.WriteLine("OnError: {0}", ex.Message),
                    () => Console.WriteLine("OnCompleted")
                ))
            {
                Console.WriteLine("Press ENTER to unsubscribe...");
                Console.ReadLine();
            }
```

c. What's happening here is that the call to Subscribe started a background thread to pump out the observable sequence's values based on a timer. We'll learn more about the concurrency aspects of Rx later on, but for now let's verify this hypothesis by looking at the Threads window in the debugger (Debug, Windows, Threads or CTRL+D,T):

What can be seen from the above is the worker thread is clearly driven by a System.Threading timer running code in the ThreadPoolScheduler from System.Concurrency. Rx always uses primitives in this namespace to create concurrency when it needs to do so.

d. Now hit F5 to continue execution. At this time, we end up hitting our breakpoint which runs on the background thread as can be seen from the debugger:

```
            using (source.Subscribe(
                    x => Console.WriteLine("OnNext:   {0}", x),
The process or thread has changed since last step. WriteLine("OnError: {0}", ex.Message),
                    () => Console.WriteLine("OnCompleted")
                ))
            {
                Console.WriteLine("Press ENTER to unsubscribe...");
                Console.ReadLine();
            }
```

100 % ▾ ◂

Call Stack

Name
⬤ Exercises.exe!Program.Main.AnonymousMethod_4(int x = 1) Line 14
System.Reactive.dll!System.Collections.Generic.AnonymousObserver<int>.Next(int value) + 0xd bytes
System.Reactive.dll!System.Collections.Generic.AbstractObserver<int>.OnNext(int value) + 0x11 bytes
System.Reactive.dll!System.Collections.Generic.AnonymousObservable<int>.AutoDetachObserver.Next(int value)
System.Reactive.dll!System.Collections.Generic.AbstractObserver<int>.OnNext(int value) + 0x11 bytes
System.Reactive.dll!System.Linq.Observable.GenerateWithTime<int,int>.AnonymousMethod_471(System.Action·

Notice the main thread is still in Console.ReadLine (the gray adorner reveals a different thread) while our call stack reflects the timer-based background thread pumping out an OnNext message for x = 1.

e. The reader should feel free to hit F5 a couple more times to see the breakpoint getting hit for every subsequent OnNext message flowing out of the observable sequence. Setting a breakpoint on the OnCompleted will show the same multi-threaded behavior for the Generate sequence:

```
file:///C:/RxHOL/Exercises/Exercises/bin/Debug/Exercises.EXE

Press ENTER to unsubscribe...
OnNext:   0
OnNext:   1
OnNext:   4
OnNext:   9
OnNext:   16
```

```
            using (source.Subscribe(
                    x => Console.WriteLine("OnNext:   {0}", x),
                    ex => Console.WriteLine("OnError: {0}", ex.Message),
                    () => Console.WriteLine("OnCompleted")
                ))
            {
                Console.WriteLine("Press ENTER to unsubscribe...");
                Console.ReadLine();
            }
```

Whether or not a sequence pumps out messages to subscribed observers on a background thread depends on a number of factors. We won't detail this aspect at this point and will encounter the use of synchronization primitives in Exercise 3. Suffice to say that the developer can control this aspect by manually specifying an IScheduler object if ever needed.

Conclusion: Creating observable sequences does not require manual implementation of the IObservable<T> interface, nor does the use of Subscribe require an IObserver<T> implementation. For the former, a series of operators exist that create sequences with zero, one or more elements. For the latter, Subscribe extension methods exist that take various combinations of OnNext, OnError and OnCompleted handlers in terms of delegates.

Exercise 3 – Importing .NET events into Rx

Objective: Creating observable sequences out of nowhere using various factory "constructor" methods encountered in the previous exercise is one thing. Being able to bridge with existing sources of asynchrony in the framework is even more important. In this exercise we'll look at the FromEventPattern operator that allows "importing" a .NET event into Rx as an observable collection. Every time the event is raised, an OnNext message will be delivered to the observable sequence.

Background: Rx doesn't aim at replacing existing asynchronous programming models such as .NET events, the asynchronous pattern or the Task Parallel Library. Those concepts or frameworks are often perfectly suited for direct use, e.g. using event handlers in C#. However, once composition enters the picture, using those low-level concepts tends to be a grueling experience where one has to deal with resource maintenance (e.g. when to unsubscribe) and often has to reinvent the wheel (e.g. how do you "filter" an event?). Needless to say, all of this can be very error prone and distracts the developer from the real problem being solved. In this sample and the ones that follow, we'll show where the power of Rx comes in: composition of asynchronous data sources.

1. Windows Forms is a good sample of a framework API that's full of events. To reduce noise and be in absolute control, we won't create a Windows Forms project but will start from a new Console Application project. Once it's been created, add a reference to System.Windows.Forms and System.Drawing as well as System.Reactive:
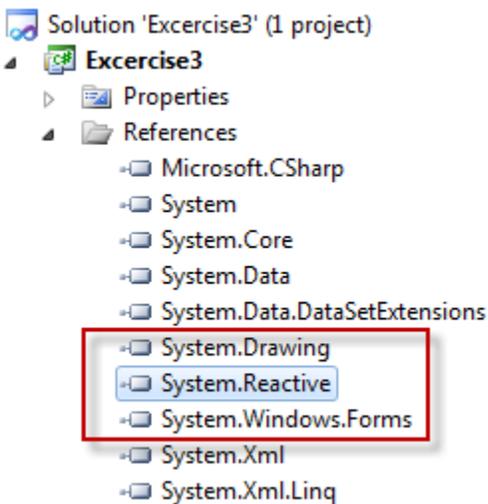
2. Enter the following code to create a new form and start running it by calling Application.Run:

```csharp
using System;
using System.Reactive.Linq;
using System.Windows.Forms;

class Program
{
    static void Main()
    {
        var frm = new Form();
        Application.Run(frm);
    }
}
```

Running the code above should simply show a form and cause the program to quit upon closing the form.

3. In order to contrast the world of Rx from classic .NET events, let's start by showing the well-understood way to deal with the latter. For our running sample, we'll be dealing with the MouseMove event:

```
var lbl = new Label();
var frm = new Form {
    Controls = { lbl }
};

frm.MouseMove += (sender, args) => {
    lbl.Text = args.Location.ToString();  // This has become a position-tracking label.
};

Application.Run(frm);

// We're sloppy and "forget" to detach the handler... This is harder than you may
// expect due to the use of a lambda expression (see point 4 below).
```

While this works great, there are a number of limitations associated with classic .NET events:

- Events are hidden data sources. It requires looking at the handler's code to see this. Did you ever regard the MouseMove event as a collection of Point values? In the world of Rx, we see events as just another concrete form of observable sequences: your mouse is a database of Point values!
- Events cannot be handed out, e.g. an event producing Point values cannot be handed out to a GPS service. The deeper reason for this is that events are not first-class objects. In the world of Rx, each observable sequence is represented using an object that can be passed around or stored.
- Events cannot be composed easily. For example, you can't hire a mathematician to write a generic filter operator that will filter an event's produced data based on a certain criterion. In the world of Rx, due to the first-class object nature of observables, we can provide generic operators like Where.
- Events require manual handler maintenance which requires you to remember the delegate that was handed to it. It's like keeping your hands on the money you paid for your newspaper subscription in order to be able to unsubscribe. In the world of Rx, you get an IDisposable handle to unsubscribe.

4. Now let's see how things look when using Rx. To import an event into Rx, we use the FromEventPattern operator, which we tell the EventArgs objects that will be raised by the event being bridged. Two overloads exist, one that takes a pair of functions used to attach and detach a handler, and one that uses reflection to find those add/remove methods on your behalf. We'll go for the latter approach here:

```
var lbl = new Label();
var frm = new Form {
    Controls = { lbl }
};

var moves = Observable.FromEventPattern<MouseEventArgs>(frm, "MouseMove");
using (moves.Subscribe(evt => {
        lbl.Text = evt.EventArgs.Location.ToString();
    }))
{
    Application.Run(frm);
}

// Proper clean-up just got a lot easier...
```
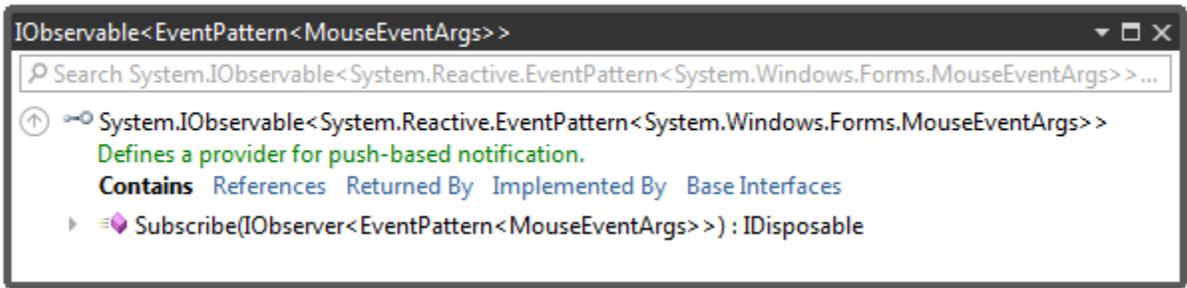
We'll now explain this fragment in depth to drive home the points we made before:

- The FromEventPattern operator turns the given event in an observable sequence with an EventPattern<T> element type that captures both the sender and the event arguments. Hovering over

the var keyword reveals the type inferred for the moves local variable:

```
var moves = Observable.FromEventPattern<MouseEventArgs>(frm, "MouseMove");
```

```
IObservable<EventPattern<MouseEventArgs>>                                    ▼ □ ✕

 🔍 Search System.IObservable<System.Reactive.EventPattern<System.Windows.Forms.MouseEventArgs>>…

 ⬆  ⟶ System.IObservable<System.Reactive.EventPattern<System.Windows.Forms.MouseEventArgs>>
        Defines a provider for push-based notification.
        Contains  References  Returned By  Implemented By  Base Interfaces
     ▸  ⬖ Subscribe(IObserver<EventPattern<MouseEventArgs>>) : IDisposable
```

When calling Subscribe, a handler is attached to the underlying event. For every time the event gets raised, the sender and arguments are stowed away in an EventPattern<MouseEventArgs> object that's sent to all of the observable's subscribers.

- Inside our OnNext handler, due to the strong typing provided by generics, we can "dot into" the event arguments' Location property. We'll see later how we can leverage built-in operators to shake off the need to traverse this object graph in the handler.

- Clean-up of the event handler is taken care of by the IDisposable object returned by the FromEventPattern operator. Calling Dispose – here achieved by reaching the end of the using-block – will automatically get rid of the underlying event handler.

5. To master the technique a bit further, let's have a look at another Windows Forms event. First add a TextBox control to the form, which we can conveniently do using C# 3.0 object initializer syntax:

```
var txt = new TextBox();

var frm = new Form
{
    Controls = { txt }
};
```

Note: Alternatively, feel free to use the proper Windows Forms designer to build the UI and set up observable sequences from the Form_Load event.

6. Restructure the code to look as follows in order to have both the Form's MouseMove and the TextBox's TextChanged event. This time, we'll print to the console as a means to do logging. In Exercise 5 we'll learn about a specialized operator (called Do) that can be used for this purpose.

```
var moves = Observable.FromEventPattern<MouseEventArgs>(frm, "MouseMove");
var input = Observable.FromEventPattern(txt, "TextChanged");

var movesSubscription = moves.Subscribe(evt =>
    {
        Console.WriteLine("Mouse at: " + evt.EventArgs.Location);
    });
var inputSubscription = input.Subscribe(evt =>
    {
        Console.WriteLine("User wrote: " + ((TextBox)evt.Sender).Text);
    });

using (new CompositeDisposable(movesSubscription, inputSubscription))
{
```

```
            Application.Run(frm);
    }
```

At this point it may seem we haven't gained too much yet. Things are just "different". What really matters though is that we've put us in the world of IObservable<T> sequences, over which a lot of operators are defined that we'll talk about in a moment.

For one thing, notice all the hoops one has to go through in order to get the text out of a TextChanged event. As mentioned before, classic .NET events don't explicitly exhibit a data-oriented nature. This particular event is a great example of this observation: from the event handler of a TextChanged event one doesn't immediately get the text after the change has occurred, even though that's what 99% of uses of the event are about (the other 1% may be justified by "state invalidation handling", e.g. to enable "Do you want to save changes?" behavior).

Finally, notice the way we can group the IDisposable subscription handlers together using one of the types in the System.Reactive.Disposables namespace. CompositeDisposable simply creates an IDisposable object that will dispose all of its contained IDisposable objects upon calling Dispose. We'll omit further exploration of this namespace in this hands-on lab.

```
using (new CompositeDisposable(moveSubscription, inputSubscription))
{
    Application.Run(frm)        Open in .NET Reflector
}                               Resolve              ▶    ⟨⟩  using System.Reactive.Disposables;
                                Refactor             ▶        System.Reactive.Disposables.CompositeDisposable
                                Generate             ▶
```
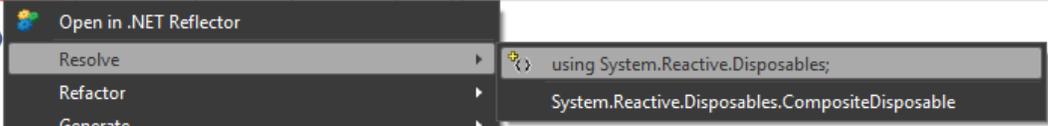
7. A fragment of sample output is shown below:
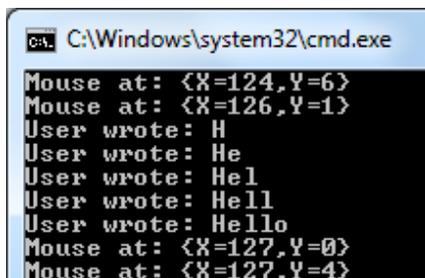
```
C:\Windows\system32\cmd.exe
Mouse at: {X=124,Y=6}
Mouse at: {X=126,Y=1}
User wrote: H
User wrote: He
User wrote: Hel
User wrote: Hell
User wrote: Hello
Mouse at: {X=127,Y=0}
Mouse at: {X=127,Y=4}
```

Conclusion: .NET events are just one form of asynchronous data sources. In order to use them as observable collections, Rx provides the FromEventPattern operator. In return one gets EventPattern objects containing the sender and event arguments.

Exercise 4 – A first look at some Standard Query Operators

Objective: Looking at observable sequences as asynchronous data sources is what enables them to be queried, just like any other data source. Who says querying in the context of C# programming nowadays, immediately thinks LINQ. In this exercise we'll show how to use the LINQ syntax to write queries over observable collections.

1. Continuing with the previous exercise's code, let's have a look back at the code we wrote to handle various UI-related events brought into Rx using the FromEventPattern operator:

```
var moves = Observable.FromEventPattern<MouseEventArgs>(frm, "MouseMove");
var input = Observable.FromEventPattern(txt, "TextChanged");

var movesSubscription = moves.Subscribe(evt =>
    {
        Console.WriteLine("Mouse at: " + evt.EventArgs.Location);
    });
var inputSubscription = input.Subscribe(evt =>
    {
        Console.WriteLine("User wrote: " + ((TextBox)evt.Sender).Text);
    });
```

Recall the type of the moves and input collections, both of which are IObservable<IEvent<TEventArgs>> objects, where TEventArgs is obtained from the generic parameter on FromEvent. Quite often we're not interested in all of the information an IEvent captures and want to "shake off" redundant stuff.

2. In a classic .NET event world – and in any other programming model for asynchronous data sources before the advent of Rx for that matter – such data-intensive operations often led to imperative code. For events, many of you will likely have written code like this:

```
private void frm_MouseMove(object sender, MouseEventArgs e)
{
    Point position = e.Location;
    if (position.X == position.Y)
    {
        // Only want to respond to events for mouse moves
        // over the first bisector of the form.
    }
}

private void txt_TextChanged(object sender, EventArgs e)
{
    string text = ((TextBox)sender).Text;
    // And now we can forget about sender and e parameters...
}
```

What we've really done here is mimicking data-intensive "sequence operators" in an imperative way. The first sample shows a *filter* using an if-statement; the second one embodies a *projection* using another local variable. In doing so, we've lost an important property though. The parts omitted by green comments no longer directly operate on an event but are lost in a sea of imperative code. In other words, it's not possible to filter an event and get another event back.

3. In the brave new Rx world, we can do better than this. Since observable sequences have gotten a first-class status by representing them as IObservable<T> *objects*, we can provide operators over them by providing a

whole set of *(generic extension) methods*. By ensuring some of those methods have the right signature, LINQ syntax works out of the box. Let's have a look at how we'd revamp the imperative event handler code above using those query operators over observable sequences. First, let's turn the event-based input sequences into what we wish they'd look like:
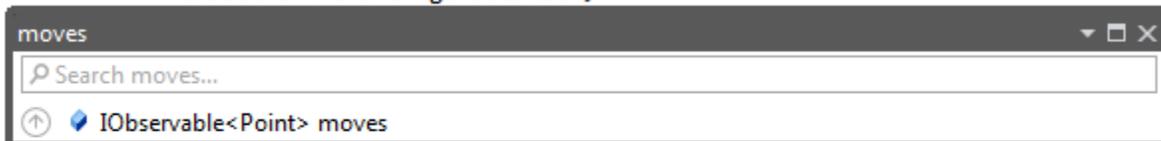
```csharp
var moves = from evt in Observable.FromEventPattern<MouseEventArgs>(frm, "MouseMove")
            select evt.EventArgs.Location;

var input = from evt in Observable.FromEventPattern(txt, "TextChanged")
            select ((TextBox)evt.Sender).Text;

var movesSubscription = moves.Subscribe(pos => Console.WriteLine("Mouse at: "   + pos));
var inputSubscription = input.Subscribe(inp => Console.WriteLine("User wrote: " + inp));
```

Using a query expression in C#, we project away the EventPattern<MouseEventArgs> and EventPattern<EventArgs> data type in favor of a Point and a string, respectively. As a result both the moves and input sequences now are observable sequences of a meaningful data type that just captures what we need:

```csharp
var moves = from evt in Observable.FromEventPattern<MouseEventArgs>(frm, "MouseMove")
            select evt.EventArgs.Location;
```

moves	▾ ▢ ✕
🔍 Search moves...	
⬆ ◆ IObservable<Point> moves	

```csharp
var input = from evt in Observable.FromEventPattern(txt, "TextChanged")
            select ((TextBox)evt.Sender).Text;
```

input	▾ ▢ ✕
🔍 Search input...	
⬆ ◆ IObservable<string> input	
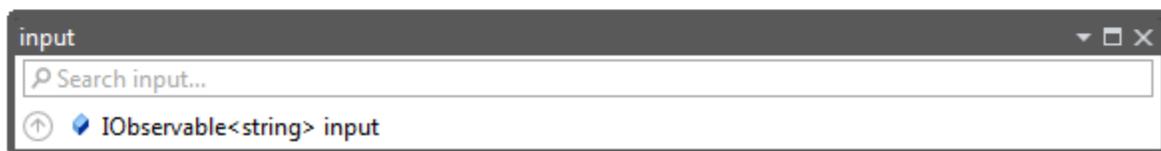
The query expression syntax is simply shorthand syntax for query operator methods. The above corresponds to the following equivalent code:

```csharp
var moves = Observable.FromEventPattern<MouseEventArgs>(frm, "MouseMove")
            .Select(evt => evt.EventArgs.Location);

var input = Observable.FromEventPattern(txt, "TextChanged")
            .Select(evt => ((TextBox)evt.Sender).Text);

var movesSubscription = moves.Subscribe(pos => Console.WriteLine("Mouse at: "   + pos));
var inputSubscription = input.Subscribe(inp => Console.WriteLine("User wrote: " + inp));
```

Background: The ability to represent asynchronous data sources as first-class objects is what enables operators like this to be defined. Being able to produce an observable sequence that operates based on input of one or more sequences is not just interesting from a data point of view. Equally important is the ability to control the lifetime of subscriptions. Consider someone subscribes to, say, the input sequence. What really happens here is that the Select operator's result sequence gets a request to subscribe an observer. On its turn, it propagates this request to its source, which happens to be a sequence produced by FromEventPattern. Ultimately, the event-wrapping source sequence hooks up an event handler. Disposing a subscription is propagated in a similar manner.

4. With the projections in place to reduce the noise on input sequences, we can now easily filter the mouse moves to those over the first bisector (where x and y coordinates are equal). How do you perform a filter in LINQ? Use the where keyword:

```
var overFirstBisector = from pos in moves
                        where pos.X == pos.Y
                        select pos;

var movesSubscription = overFirstBisector.Subscribe(pos =>
                            Console.WriteLine("Mouse at: " + pos));
```

   The type for both moves and overFirstBisector will be IObservable<Point>.

5. A sample of the output is shown below. All of the emitted mouse move messages satisfy the filter constraint we specified in the query:
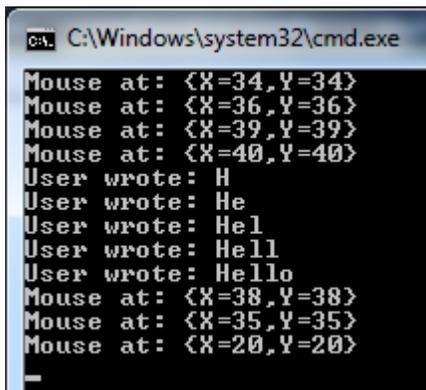
Conclusion: The first-class nature of observable sequences as IObservable<T> objects is what enables us to provide generic operators (sometimes referred to as operators) to be defined over them. The majority of those operators produce another observable sequence. This allows continuous "chaining" of operators to manipulate an asynchronous data source's emitted results till the application's requirements are met. LINQ syntax provides a convenient way to carry out some of those common operators. Others will be discussed further on.

Exercise 5 – More query operators to tame the user's input

Objective: Observable sequences are often not well-behaved for the intended usage. We may get data presented in one form but really want it in another shape. For this, simple projection can be used as shown in the previous exercise. But there are far more cases of ill-behaved data sources. For example, duplicate values may come out. But there's more beyond the perspective of observable sequences as "just data sources". In particular, asynchronous data sources have an intrinsic notion of timing. What if a source goes too fast for consumers to deal with their data? We'll learn how to deal with those situations in this exercise.

From this exercise on, we'll be floating on a common theme of the typical "dictionary suggest" sample for asynchronous programming. The idea is to let the user type a term in a textbox and show all the words that start with the term, by consulting a web service. To keep the UI from freezing, we got to do this kind of stuff in an asynchronous manner. Rx is a great fit for this kind of composition. But first things first, let's see how our form's TextBox control is behaving.

1. In what follows, we won't need the MouseMove event anymore, so let's stick with just a single TextBox control and its (projected) TextChanged event:

    ```
    var input = from evt in Observable.FromEventPattern(txt, "TextChanged")
                select ((TextBox)evt.Sender).Text;

    using (input.Subscribe(inp => Console.WriteLine("User wrote: " + inp)))
    {
        Application.Run(frm);
    }
    ```

    Note: UI designer fanatics should feel free to start from a clean slate with a Windows Forms (or WPF) designer and put code equivalent to the above in a classic (which is perfectly fine for this purpose) Load event handler. In such a scenario, you'll likely store the subscription IDisposable object in a field to dispose it when the form or window is unloaded or closed.

2. Now, let's carry out a few experiments. Start the application and type "reactive" (without the quotes) in the textbox. Obviously you should see no less than eight events being handled through the observer. However, notice what happens if you select a single letter in the word and overwrite it by the same letter:

    r e a c t|i v e → (SHIFT-LEFT ARROW) r e a c t i v e → (type t) r e a c t|i v e

    The screenshot below shows the corresponding output. What's this duplicate message at the end about? Didn't we ask for Text*Changed* events? Yes, but it turns out UI frameworks do not keep track of the last value entered by the user and may raise false positives based on internal state changes.
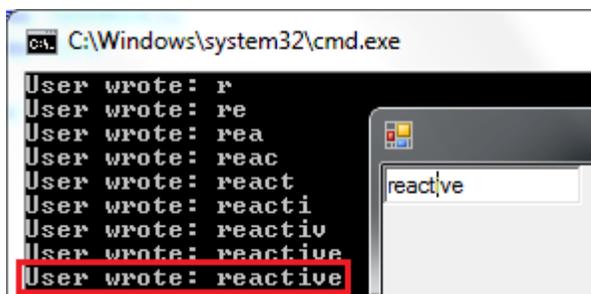
    Notice the same "issue" appears when pasting the same text over the entire selection of a TextBox (e.g. CTRL-A,

CTRL-C, CTRL-V to exhibit the quirk).

3. Assume for a moment we take the user input and feed it to a dictionary suggest web service (as we will later on) which charges the user or application vendor for every request made to the service. Do you really want to pay twice the price to lookup "reactive" because of some weird behavior in the UI framework? Likely not.

So how would you solve the issue in an Rx-free world? Well, you'd keep some state somewhere to keep track of the last value seen and only propagate the input through in case it differs from the previous input. All of this clutters the code significantly with things like a private field, an if-statement and additional assignment in the event handler, etc. But worse, all the logic goes in an event handler which lacks composition: at no point we have an asynchronous data source, free of duplicates, we can put our hands on (e.g. to hand it to another component in the system).

In Rx, thanks to the power of composition, we get away with a single operator call that does all the comparison and state maintenance on our behalf. This operator is called DistinctUntilChanged:

```
var input = (from evt in Observable.FromEventPattern(txt, "TextChanged")
             select ((TextBox)evt.Sender).Text)
            .DistinctUntilChanged();

using (input.Subscribe(inp => Console.WriteLine("User wrote: " + inp)))
{
    Application.Run(frm);
}
```

Overloads exist that accept an IEqualityComparer<T> object to carry out the check for equality between the current and previous elements.

With this fix in place, runs of identical values will only cause the first such value to be propagated. If the value received from the source is different the very first value or different from the previous one, it goes out at that very moment. If it's the same as the previous value, it's muted and attached observers don't get to see it.

Background: It's essential to understand how data flows through a "network" of interconnected observable sequences. In the sample above, there are three sequences in the mix. First, there's FromEventPattern that listens on a classic .NET event and emits its values to subscribed observers. Next, the Select operator takes care of carrying out a projection by receiving values, transforming them and sending them out. Finally, DistinctUntilChanged receives output from Select, filters out consecutive duplicates and propagates the results to its observer. The figure below illustrates how a subscription is set up and how data flows through the operators.
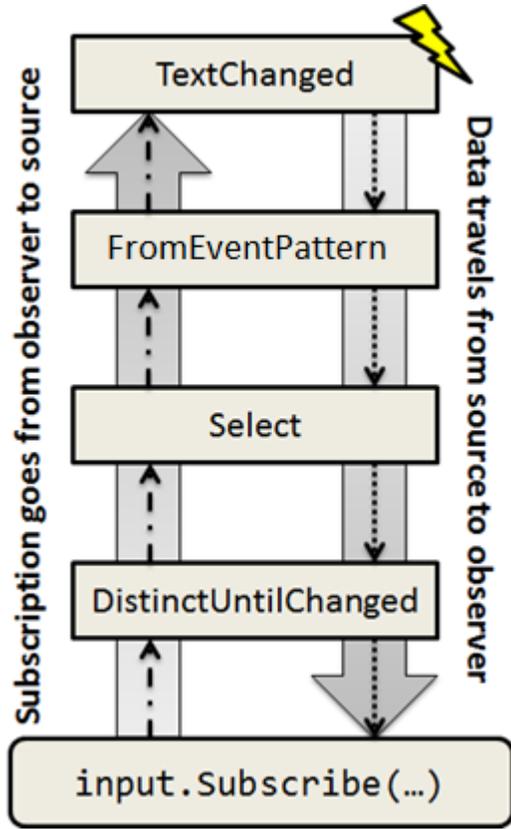
Each operator is a little black box that knows how to propagate subscriptions to its source sequence(s), as well as how to take the data it receives and transform it to send it along (if desired). All of this works nice till the point you see some data coming out in the observer and wonder where it comes from. To figure that out, a handy operator is called Do, which allows to log the data that's flowing through a "wire":

Cut along dotted line
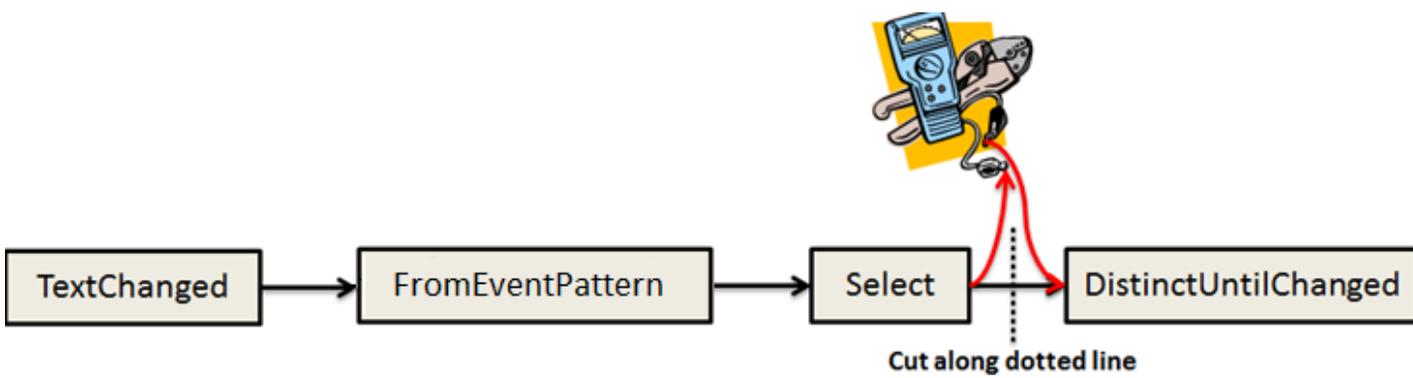
This is the Rx form of "printf debugging". The Do-operator has several overloads that are similar to Subscribe's and Run's. Either you give it an observer to monitor the data that's been propagated down through the operator or you can give it a set of handlers for OnNext, OnError and OnCompleted. Below is a sample of the operator's use to see DistinctUntilChanged filtering out the duplicate values it receives:

```
var input = (from evt in Observable.FromEventPattern(txt, "TextChanged")
             select ((TextBox)evt.Sender).Text)
            .Do(inp => Console.WriteLine("Before DistinctUntilChanged: " + inp))
            .DistinctUntilChanged();
```

```
using (input.Subscribe(inp => Console.WriteLine("User wrote: " + inp)))
{
    Application.Run(frm);
}
```

With this in place, the output looks as follows. Here we produced the quirky duplicate input four times and yet the "User wrote" message only appears for the first such input.

Note: Feel free to attach Do probes to other operators as well. When using query expression syntax, you'll have to revert to regular method calls (such as Where, Select) in order to insert the probing Do operator call.

4. Back to our running sample, there's yet another problem with the user's input. Since we'll ultimately feed it to a web service (which may be, as we said before, "pay for play" on a per-request basis), it's unlikely we want to send requests for every substring the user wrote while entering input. Stated otherwise, we need to protect the web service against fast typists.

Rx has an operator that can be used to "calm down" an observable sequence, called Throttle:

```
var input = (from evt in Observable.FromEventPattern(txt, "TextChanged")
             select ((TextBox)evt.Sender).Text)
            .Throttle(TimeSpan.FromSeconds(1))  // Exercise: what operator order is better?
            .DistinctUntilChanged();
```

The way this works is a timer is used to let an incoming message age for the specified duration, after which it can be propagated further on. If during this timeframe another message comes in, the original message gets dropped on the floor and substituted for the new one that effectively also resets the timer. For our sample, if the user types "reactive" without hiccups (i.e. no two consecutive changes are further apart than 1 second), no intermediate substrings will be propagated. When the user stops typing (after hitting 'e', causing the last changed event higher up), it takes one second before the input "reactive" is propagated down. Later one we'll feed this entire sequence to a web service which now cannot be called excessively due to a typist gone loose.

To illustrate the operator's effect, let's use the Do operator in conjunction with two specialized projection operators called Timestamp and Select. The former takes an IObservable<T> and turns it into an IObservable<Timestamped<T>>, where the latter does the opposite. Those operators simply add or remove a timestamp at the point a message is received. This allows us to visualize timing information:

```
var input = (from evt in Observable.FromEventPattern(txt, "TextChanged")
```

```
    select ((TextBox)evt.Sender).Text)

    .Timestamp()
    .Do(inp => Console.WriteLine(  "I: " + inp.Timestamp.Millisecond
                                  + " - " + inp.Value))
    .Select(x => x.Value)

    .Throttle(TimeSpan.FromSeconds(1))

    .Timestamp()
    .Do(inp => Console.WriteLine(  "T: " + inp.Timestamp.Millisecond
                                  + " - " + inp.Value))
    .Select(x => x.Value)

    .DistinctUntilChanged();
```

Note: We need to remove and reapply the timestamp around the Throttle operator call. If we wouldn't do so, *Throttle would simply propagate the original timestamped value. This technique allows us to see the real delta* between entering Throttle and leaving it, which should be about 1 second (give or take a few milliseconds).

5. As we've used the same three operators twice, let's extract the pattern into a specialized operator. Creating your own operators shouldn't be hard as you can see:

```
public static IObservable<T> LogTimestampedValues<T>(this IObservable<T> source,
                                               Action<Timestamped<T>> onNext)
{
    return source.Timestamp().Do(onNext).Select(x => x.Value);
}
```

Now we can rewrite the code to use our operator like this:

```
    var input = (from evt in Observable.FromEventPattern(txt, "TextChanged")
                 select ((TextBox)evt.Sender).Text)

        .LogTimestampedValues(x => Console.WriteLine("I: " + x.Timestamp.Millisecond
                                                    + " - " + x.Value))

        .Throttle(TimeSpan.FromSeconds(1))

        .LogTimestampedValues(x => Console.WriteLine("T: " + x.Timestamp.Millisecond
                                                    + " - " + x.Value))

        .DistinctUntilChanged();
```
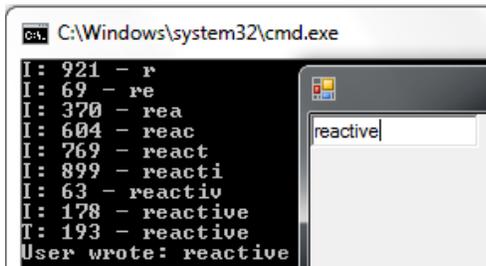
Below is the output for the sample of typing "reactive" with a mild hiccup after "re" and after "reac", both of which were in the sub-second range, hence not causing propagation to beyond the Throttle operator. However, when the user stopped typing it took 1015ms before the "reactive" string was observed in the Do operator after the Throttle operator.

```
C:\Windows\system32\cmd.exe
I: 921 - r
I: 69 - re
I: 370 - rea
I: 604 - reac
I: 769 - react
I: 899 - reacti
I: 63 - reactiv
I: 178 - reactive
T: 193 - reactive
User wrote: reactive
```

Note: It's strongly encouraged to brainstorm for a moment how you'd write a Throttle operator on classic .NET events taking all complexities of timers, subscriptions, resource management, etc. into account. One of the core properties of Rx is that it allows reusable operators to be written, operating on a wide range of asynchronous data sources. This improves signal-to-noise ratio of user code significantly. In this particular sample, just two operators had to be added in order to tame the input sequence both for its data and for its timing behavior.

Conclusion: Thanks to the first-class nature of observable sequences we were able to apply operators to the TextBox data sources to tame it. We learned how to filter out consecutive duplicate values and how to calm down an event stream using the Throttle operator. We also introduced debugging techniques using Do and Timestamp in this exercise. Manned with a well-behaved asynchronous input sequence from the TextBox control, we're now ready to walk up to the dictionary suggest web service, ask for word suggestions and present them to the user. It goes without saying that Rx will once more be the protagonist in this composition play. But before we do so, let's talk about synchronization.

Exercise 6 – Rx's concurrency model and synchronization support

Objective: Rx employs a free world with regards to concurrency of data sources and operators that act upon those. This means that an observer's On methods can be called from any "context", which typically boils down to threads. As an example, since an event in a UI framework is raised on the UI thread, FromEventPattern's call to OnNext with the event's data will also happen on that thread. In contrast, when a timer from the System.Threading namespace is used (e.g. through the use of the GenerateWithTime operator), its messages will come out on a different (threadpool worker) thread. Rx has ways to introduce concurrency known as schedulers and has operators that deal with synchronization onto a specific scheduler. We'll have a look at those now.

1. In the previous example, we've been calming down a TextBox's input using the time-based Throttle method. In order for this operator to send out messages, it needs to start a timer that's used to age the incoming messages in the way we described before. Under the debugger it's pretty straightforward to see those messages are indeed received on a worker thread:

```
19                 var input = (from evt in Observable.FromEventPattern<EventArgs>(txt, "TextChanged")
20                             select ((TextBox)evt.Sender).Text)
21                             .Throttle(TimeSpan.FromSeconds(1))
22                             .DistinctUntilChanged();
23
24                 using (input.Subscribe(inp => Console.WriteLine("User wrote: " + inp)))
25                 {
26                     Application.Run(frm);
27                 }
28
29             }
30         }
31 }
```
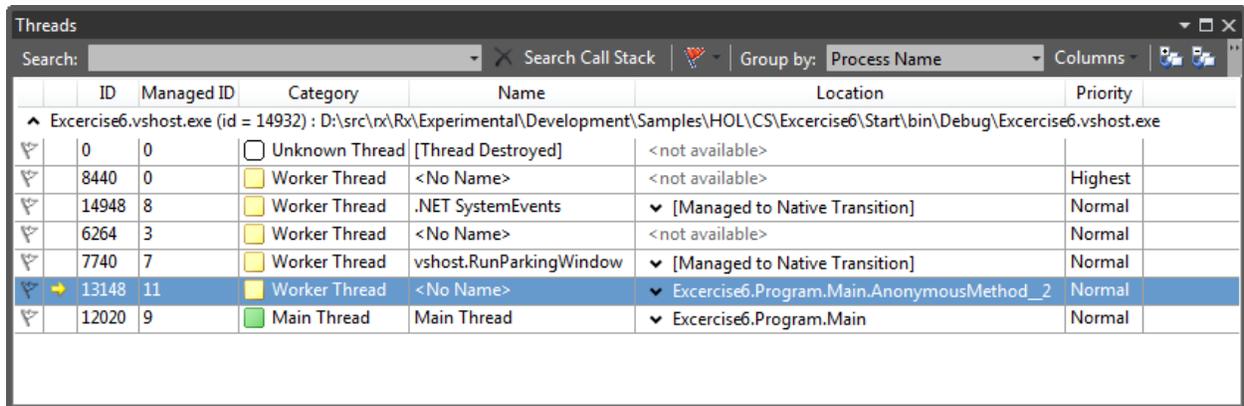
	ID	Managed ID	Category	Name	Location	Priority	
^ Excercise6.vshost.exe (id = 14932) : D:\src\rx\Rx\Experimental\Development\Samples\HOL\CS\Excercise6\Start\bin\Debug\Excercise6.vshost.exe							
	0	0	Unknown Thread	[Thread Destroyed]	<not available>		
	8440	0	Worker Thread	<No Name>	<not available>	Highest	
	14948	8	Worker Thread	.NET SystemEvents	✓ [Managed to Native Transition]	Normal	
	6264	3	Worker Thread	<No Name>	<not available>	Normal	
	7740	7	Worker Thread	vshost.RunParkingWindow	✓ [Managed to Native Transition]	Normal	
→	13148	11	Worker Thread	<No Name>	✓ Excercise6.Program.Main.AnonymousMethod_2	Normal	
	12020	9	Main Thread	Main Thread	✓ Excercise6.Program.Main	Normal	

2. As UI-savvy readers know, updating the UI from a thread other than the UI thread is a big no-no. To illustrate this point in the context of Rx, let's introduce a Label control on the form and update the handler to update the label with what the user just wrote (after being throttled and filtered for adjacent duplicates):

```
var txt = new TextBox();
var lbl = new Label { Left = txt.Width + 20 };
var frm = new Form {
    Controls = { txt, lbl }
};

var input = (from evt in Observable.FromEventPattern(txt, "TextChanged")
            select ((TextBox)evt.Sender).Text)
            .Throttle(TimeSpan.FromSeconds(1))
```
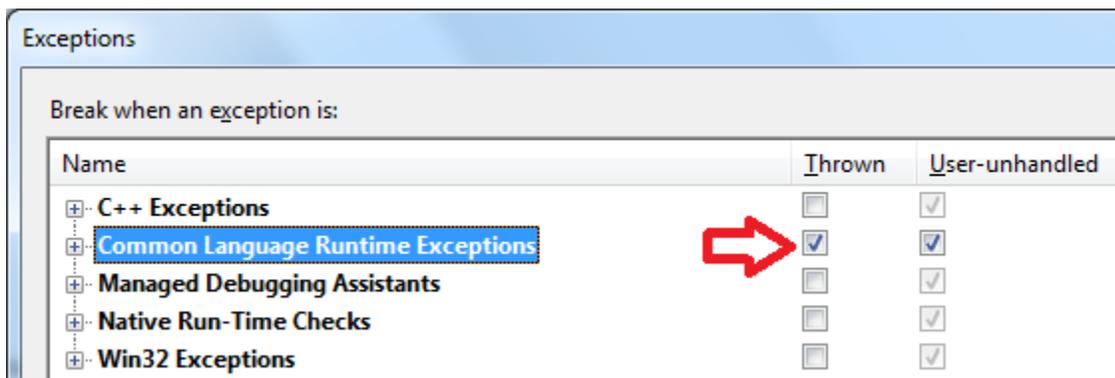
```
            .DistinctUntilChanged();

    using (input.Subscribe(inp => lbl.Text = inp))
        Application.Run(frm);
```

Running this piece of code will cause it to fail upon trying to assign to the label's Text property since that code is run from a thread other than the UI thread.

Note: In Windows Forms, behavior depends on whether or not a debugger is attached to the process. To reproduce this behavior, make sure to run under the debugger (i.e. start with F5, not CTRL-F5).

Start by enabling first-chance exceptions from the Debug, Exceptions… dialog (CTRL+D,E), as illustrated below. This will break in the debugger before the exception gets propagated and gets a chance to terminate the process:

Trying to run the executable now, entering a term in the TextBox control (and waiting for 1 second for it to be propagated beyond the Throttle operator) produces the following result:
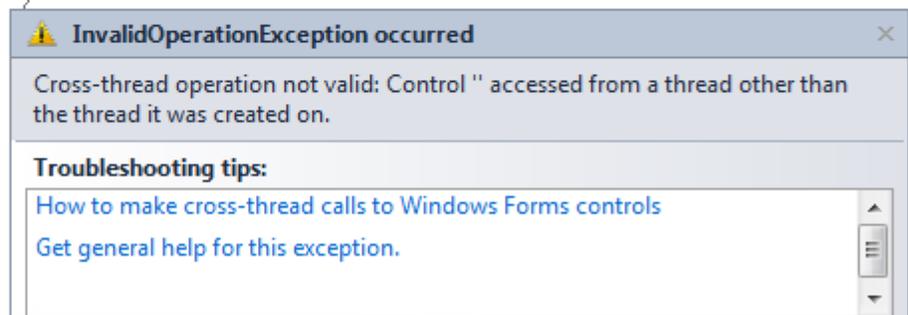
Background: If you'd have a peek at the call stack, you'd immediate see what we noticed in the previous step: the OnNext action is getting invoked from a background thread, with System.Concurrency near the bottom of the stack. This reveals Rx's source of concurrency in so-called IScheduler primitives. Whenever Rx needs to introduce concurrency to make an operator do its job, it uses this namespace to call into a scheduler. All of the operators that deal with concurrency have overloads with a parameter that lets the user specify an IScheduler in case the default is not what's desired. This said, the defaults were carefully chosen, so typically one doesn't
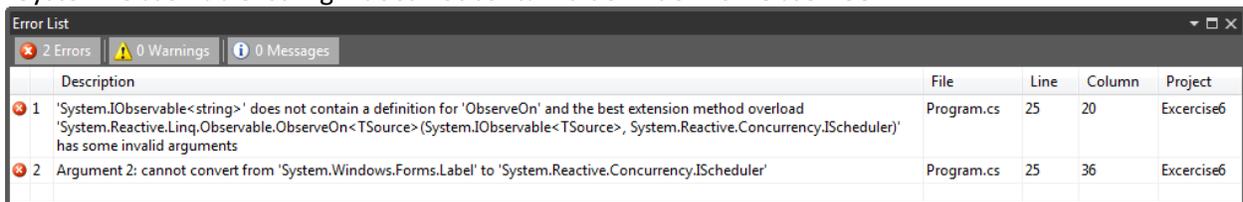
need to bother about those at all.

3. So, how do we solve this issue? Obviously one could click the link "How to make cross-thread calls to Windows Forms controls" as highlighted above, to learn the Invoke method is all you need to solve the issue. However, the deeper construct needed here is a way to jump between schedulers, in this case from some background thread onto the UI thread. Thanks to the powerful first-class object nature of observable sequences, we can provide the user with an operator that does precisely this. Also, this solution is more generic than a fix that just overcomes the single-threaded UI limitation, as it allows one to jump between any two IScheduler "contexts". This operator is called ObserveOn and is shown below:

```
var input = (from evt in Observable.FromEventPattern(txt, "TextChanged")
             select ((TextBox)evt.Sender).Text)
            .Throttle(TimeSpan.FromSeconds(1))
            .DistinctUntilChanged();

using (input.ObserveOn(lbl).Subscribe(inp => lbl.Text = inp))
    Application.Run(frm);
```

You will notice when you try to compile the new version of the code it will fail with the error message: "System.IObservable<string>' does not contain a definition for 'ObserveOn'… ".

	Description	File	Line	Column	Project
❌ 1	'System.IObservable<string>' does not contain a definition for 'ObserveOn' and the best extension method overload 'System.Reactive.Linq.Observable.ObserveOn<TSource>(System.IObservable<TSource>, System.Reactive.Concurrency.IScheduler)' has some invalid arguments	Program.cs	25	20	Excercise6
❌ 2	Argument 2: cannot convert from 'System.Windows.Forms.Label' to 'System.Reactive.Concurrency.IScheduler'	Program.cs	25	36	Excercise6

The most generic overload of ObserveOn takes an IScheduler. For convenience, an overload is provided that takes in a Windows Forms Control. This extension method is tied to the System.Windows.Forms namespace and resides inside the System.Reactive.Windows.Forms assembly. Make the code resolve correctly we need to add an additional assembly reference to the System.Reactive.Windows.Forms assembly.
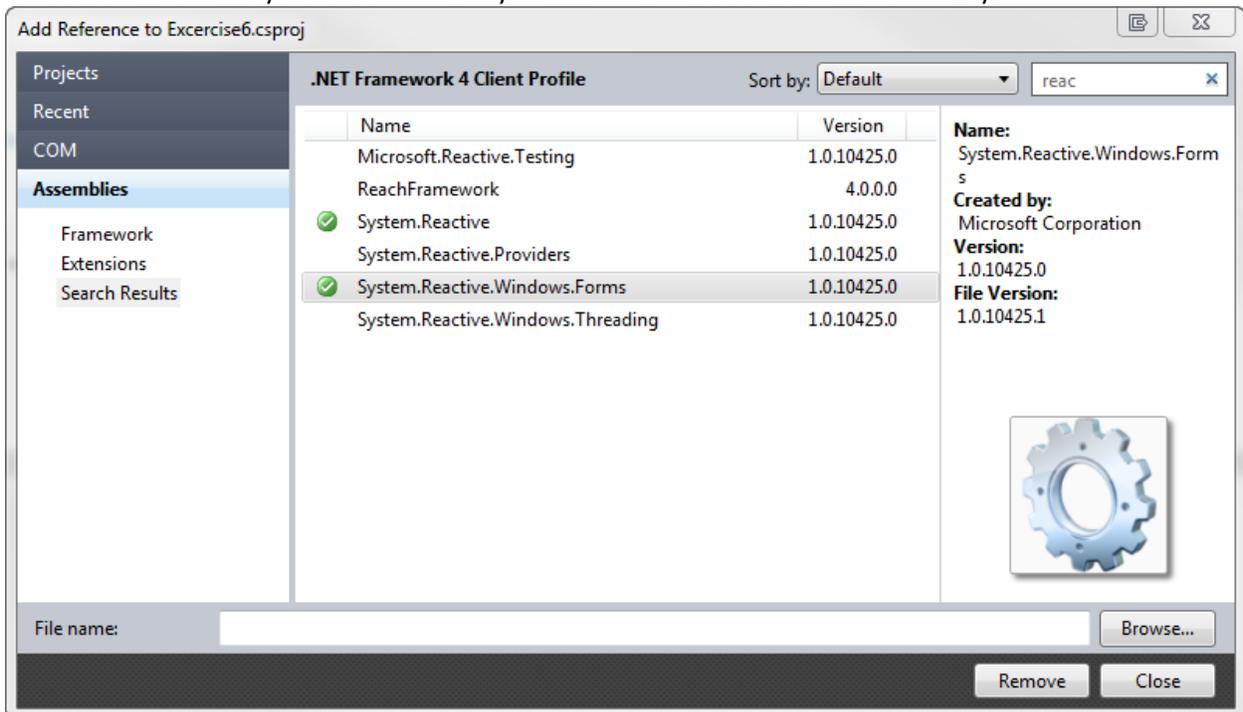
WPF users can use the ObserveOnDispatcher method to use the current dispatcher, which resides in the

System.Reactive.Windows.Threading assembly.

Background: In past designs of Rx, the team experimented with other ways to express the thread affinity of messages generated by observable sequences. One was to have a global context property and a related one was to follow the F# model where a SynchronizationContext for the UI thread is used to synchronize every single message on. Those approaches severely hampered performance and scalability of Rx. The use of IScheduler led to much more flexibility on behalf of the user and operators like ObserveOn flow nicely like any other operator. In this world, the philosophy is to defer any synchronization tasks till the last point in time, i.e. right before some UI binding is made. This approach offers the best performance characteristics.

Conclusion: Switching between threads for the messages received by observers is simply a matter of using another operator, called ObserveOn. Another sister operator, SubscribeOn, exists to synchronize the invocation of subscriptions with a given scheduler. The most generic overloads take an IScheduler but for UI programming convenience overloads exist. No longer has one to deal with low-level techniques like the WPF Dispatcher's or Windows Form's Invoke method: determining where messages are sent out is achieved using just another operator.

Exercise 7 – Bridging the asynchronous method pattern with Rx

Objective: In exercise 3, we learned how to bring classic .NET events to Rx by means of the FromEventPattern operator. Other asynchronous data sources exist in the .NET Framework, one of the most notable being the plethora of asynchronous method patterns. In this design pattern, two methods are provided. One method is used to start the computation and returns an IAsyncResult "handle" that's fed into the second method to acquire the result of the computation. All of this is pretty cumbersome to use in a manual fashion and often leads to spaghetti code with the initial call in one place and the receive logic in another spot. We'll now explore how to expose such asynchronous data sources as observables.

1. In our running sample, we've building up a simple dictionary suggest application. Upon the user entering a search term, the application will fire off a call to a web service to get word suggestions back. Since we don't want to block the UI, we'll want to keep the communication with the dictionary service asynchronous too.

   The online dictionary we'll be using can be found at http://www.dict.org. One of its disadvantages for direct consumption is the (seemingly arcane) Dictionary Server Protocol (RFC 2229) being used. To leverage our existing toolset, we'll be much better off with a web service wrapper around it which can be found at http://services.aonaware.com/DictService.

   Note: It'd be a good exercise to provide a native wrapper for the underlying Dictionary Server Protocol using Rx. Since TCP primitives in the System.Net namespace expose asynchronous operations as well, wrapping those should be not any harder than using a web service as we're doing here. Obviously dealing with byte arrays (which the author really likes) received from a TCP socket obviously leads to more clutter.

   As a little experiment with the service, choose the MatchInDict web method and enter wn for the dictionary

identifier (for WordNet ® 2.0, which can be found through the DictionaryList method), reac for the word, and prefix for the strategy. Verify that hitting Invoke produces a bunch of words starting with "reac":

MatchInDict

Look for matching words in the specified dictionary using the given strategy

Test

To test the operation using the HTTP POST protocol, click the 'Invoke' button.

Parameter	Value
dictId:	wn
word:	reac
strategy:	prefix
	Invoke

2. Keeping our existing code with the TextBox, we'll first focus on bridging with the web service. To perform those experiments, comment out your current code to have a clean Main method playground. The first thing to do is to add a reference to the web service using the Add Service Reference… entry in Solution Explorer when right-clicking the project or References node underneath it. This will create a new WCF service proxy.

Note: Classic web service client proxies could be used as well but do have a different asynchronous invocation mechanism based one classic .NET events. The new WCF approach – using the asynchronous method pattern – fits better in our sample to illustrate another Rx bridge operator.
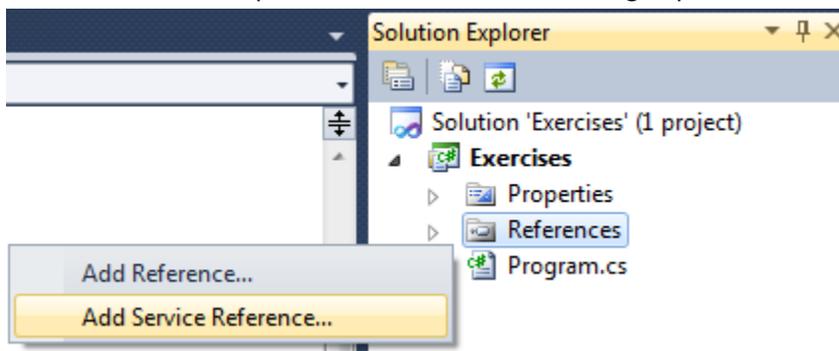
In the dialog that appears, enter http://services.aonaware.com/DictService/DictService.asmx for the service address and click Go. Change the Namespace field to DictionarySuggestService. Don't click OK yet.
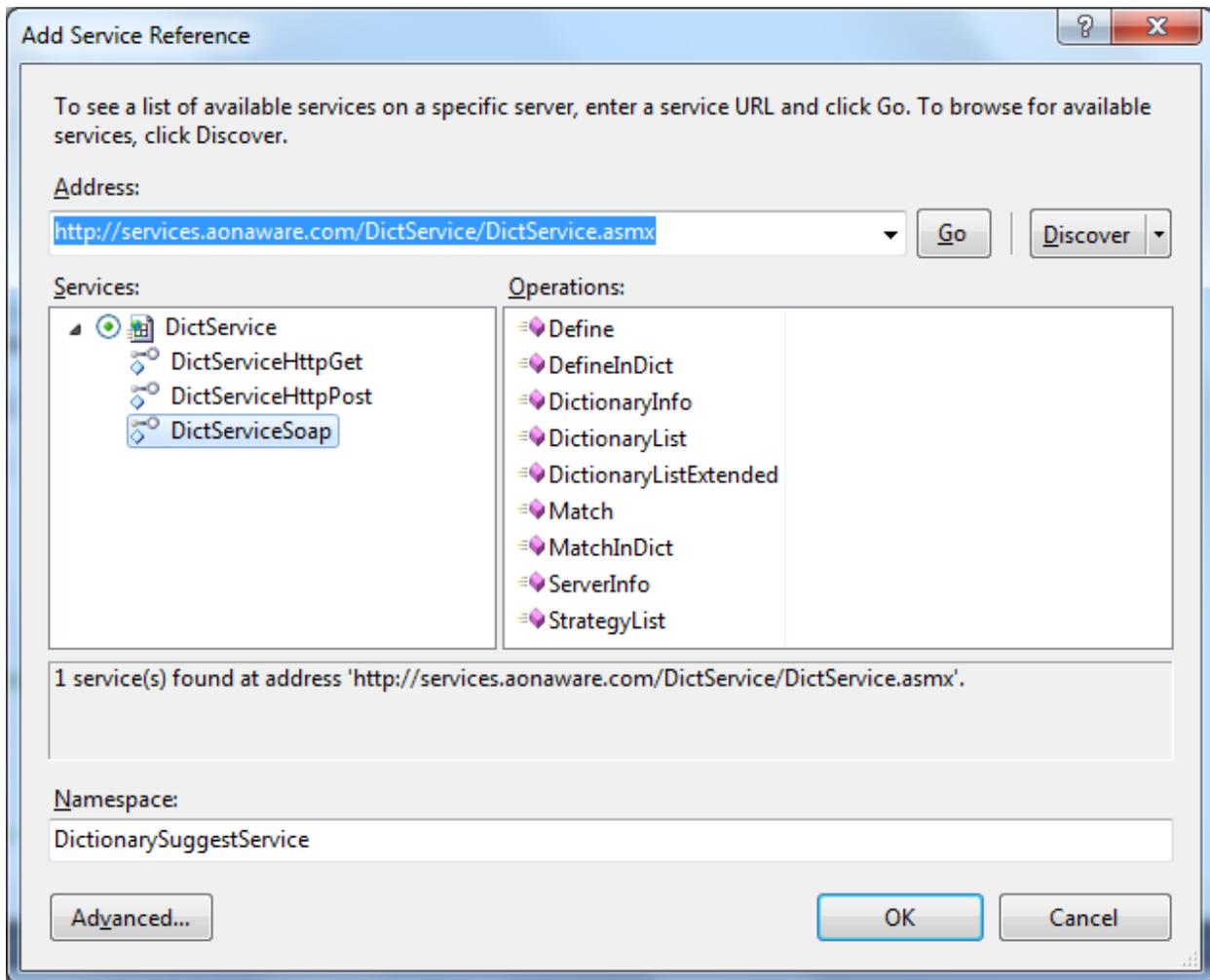
Before clicking OK, go to the Advanced… dialog and tick the "Generate asynchronous operations" option:
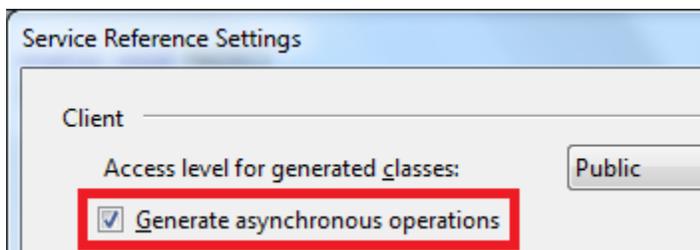
3. With our service client generated, we're ready to make web service calls. Ignore any Custom tool warnings for the References.svcmap. Despite those warnings, the generated proxy should work fine for our purposes. Let's first have a look at the code involved in manually invoking the service in an asynchronous manner:

```
var svc = new DictServiceSoapClient("DictServiceSoap");
svc.BeginMatchInDict("wn", "react", "prefix",
    iar => {
        var words = svc.EndMatchInDict(iar);
        foreach (var word in words)
            Console.WriteLine(word.Word);
    },
    null
```

```
    );

    Console.ReadLine();
```

The BeginMatchInDict method is the method that starts a web service call. Besides taking all of the parameters to be passed to the web method, it also takes an AsynCallback delegate. This delegate gets invokes when the service's response has been received. This code is quite clumsy and the data aspect of the asynchronous call not being immediately apparent. Furthermore, composition with other asynchronous data sources (such as our TextBox) becomes quite hard. It's also not clear how one can cancel outstanding requests, such that the callback procedure is guaranteed not to be called anymore. Dealing with error cases becomes hard too. Those complexities closely resemble the ones we called out for .NET events...

For completeness, here are the results you should expect to get back:
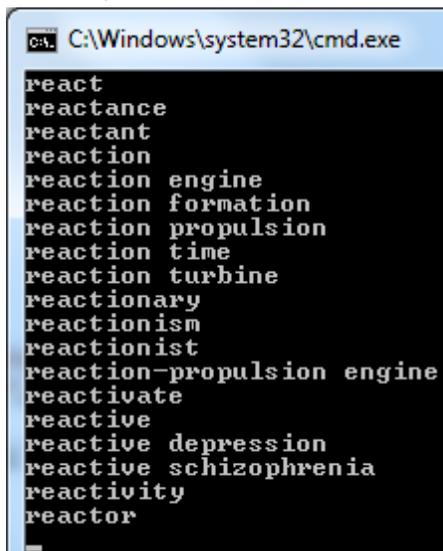
4. Converting the above fragment to Rx isn't very hard using the FromAsyncPattern method which takes a whole bunch of generic overloads for various Begin* method parameter counts. The generic parameters passed to this bridge method are the types of the Begin* method parameters, as well as the return type of the corresponding End* method. For the parameters to FromAsyncPattern, delegates to those Begin* and End* methods are passed:

```
    var svc = new DictServiceSoapClient("DictServiceSoap");
    var matchInDict = Observable.FromAsyncPattern<string, string, string, DictionaryWord[]>
        (svc.BeginMatchInDict, svc.EndMatchInDict);
```

The result of this bridging is a Func delegate that takes the web service parameters and produces an observable sequence that will receive the results:

```
    var res = matchInDict("wn", "react", "prefix");
    var subscription = res.Subscribe(words => {
            foreach (var word in words)
                Console.WriteLine(word.Word);
        });

    Console.ReadLine();
```

To make this more clear, let's get explicit about the types inferred for all of the calls shown above:

```
var matchInDict = Observable.FromAsyncPattern<string, string, string, DictionaryWord[]>
```
delegate System.Func<in T1,in T2,in T3,out TResult>
Encapsulates a method that has three parameters and returns a value of the type specified by the TResult parameter.

T1 is System.String
T2 is System.String
T3 is System.String
TResult is IObservable<DictionaryWord[]>

Contrast to .NET events (which are not parameterized), asynchronous method calls need input to operate on. To make a bridge to such a Begin-End method pair reusable, the return type is a function that accepts parameters to the underlying Begin* method.

Notice the bridge exhibits beneficial properties such as the explicit data-intensive nature reflected in the return type of the function (here an IObservable *of* a DictionaryWord-array). Other advantages include reusability and the option to unsubscribe from the asynchronous call. Since the result of calling the delegate returned from the FromAsyncPattern method returns an observable sequence, it can be used for further composition (as we shall see in a moment).

5. Since we'll always use our service with the same dictionary and strategy, let's simplify the function that acts as our entry-point to the web service. This is a trivial bit of functional programming where we simply wrap the function returned from FromAsyncPattern (which takes 3 parameters) in a function (with only one parameter, i.e. the term searched by the user) that supplies the two fixed parameter values:

```
var svc = new DictServiceSoapClient("DictServiceSoap");
var matchInDict = Observable.FromAsyncPattern<string, string, string, DictionaryWord[]>
    (svc.BeginMatchInDict, svc.EndMatchInDict);

Func<string, IObservable<DictionaryWord[]>> matchInWordNetByPrefix =
    term => matchInDict("wn", term, "prefix");

var res = matchInWordNetByPrefix("react");
var subscription = res.Subscribe(words => {
        foreach (var word in words)
            Console.WriteLine(word.Word);
    });

Console.ReadLine();
```

Running this piece of code should produce the same results as shown in step 3.
6. Since web services can easily fail, we should say a word or two on error handling. Try running the service with a single letter as its input, e.g. "r". Due to the data volume returned by the server, the System.ServiceModel layer triggers an error stating quota have been exceeded.
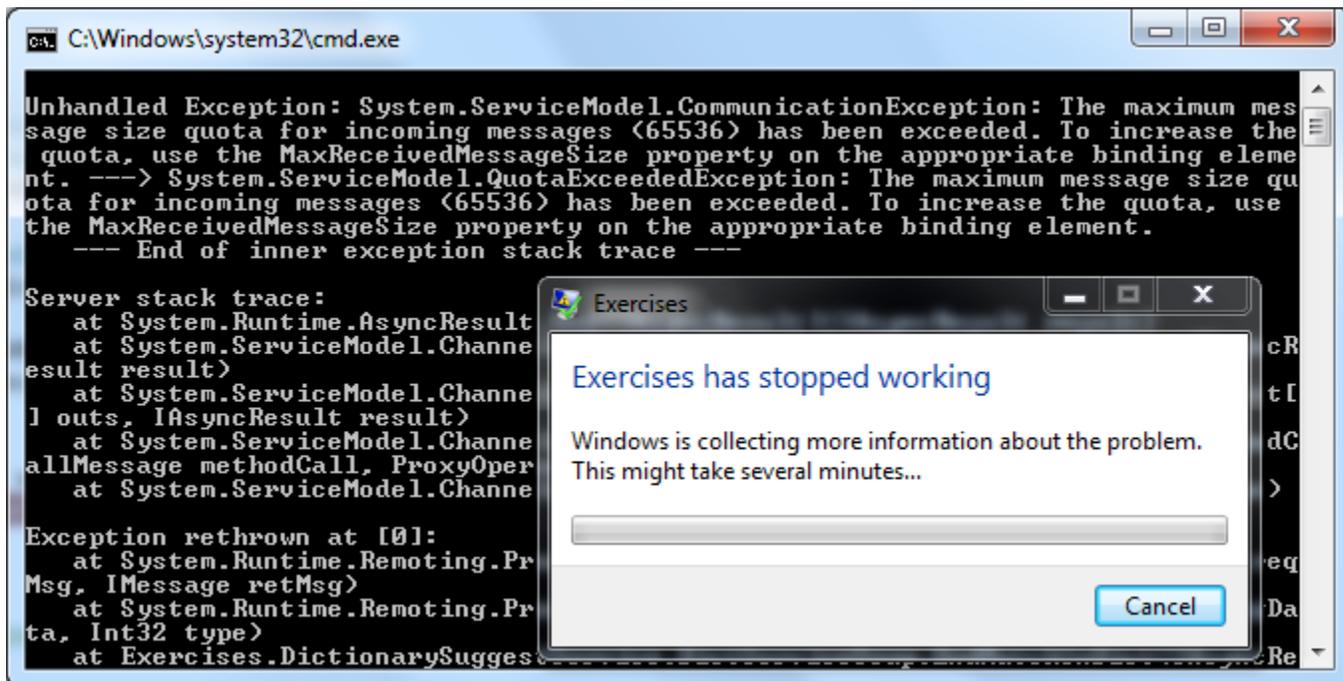
We don't really care about the specifics of this error but it should be common wisdom that in the world of distributed and asynchronous programming errors are not that exceptional. Rx is particularly good at dealing with errors due to the separate observer's OnError channel to signal those. If we were to change our sample as shown below, the error would be handled by the OnError function that's part of the observer:

```
var res = matchInWordNetByPrefix("react");
var subscription = res.Subscribe(
    words =>
    {
        foreach (var word in words)
            Console.WriteLine(word.Word);
    },
    ex =>
    {
        Console.Error.WriteLine(ex.Message);
    }
);
```

Note: Rx has exception handling operators such as Catch, Finally, OnErrorResumeNext and Retry which allow taking a compositional approach to error handling. Related operators include Materialize and Dematerialize which allow turning an IObservable<T> into an IObservable<Notification<T>> and vice versa. Such a notification represents an observer's possible messages as data, i.e. an OnNext<T> object, an OnError object or an OnCompleted one. We won't elaborate on the rich exception handling operators present in Rx and will keep things simple by using an OnError handler passed to Subscribe. If you want to make the application more robust in the presence of short input strings, you could either pre-filter the user input observable sequence (using a *where* clause on the length of the string) or fix up the underlying WCF setting as mentioned in the error text.

7. One general issue with distributed programming we should call out is the potential for out-of-order arrival of responses. In the next exercise, we'll compose the input sequence from the TextBox control with web service

calls, so multiple requests may be running at the same time. For example, if the user types "reac", waits one second (for Throttle to forward the string to its observers), then proceeds with typing "reactive" and waits another second, both web service calls will be in flight. The response to the second call may arrive before the call to the first one does. This is not too far-fetched even for this simple sample: as there'll be more words starting with "reac" than with "reactive", the first request will be more network-intensive than the second one.

We can mimic this situation quite easily by starting a couple of web service requests for "incremental strings" and observe the order answers come back in:

```
var input = "reactive";
for (int len = 3; len <= input.Length; len++)
{
    var req = input.Substring(0, len);
    matchInWordNetByPrefix(req).Subscribe(
        words => Console.WriteLine(req + " --> " + words.Length)
    );
}
```

If you run the fragment above a couple of times, you should be (un)lucky enough to hit an out-or-order arrival situation. While requests for "rea", "reac", "react", "reacti", "reactiv" and "reactive" are started in that order, results may come back in a different order as shown below:
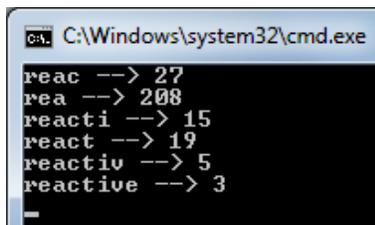

```
C:\Windows\system32\cmd.exe
reac --> 27
rea --> 208
reacti --> 15
react --> 19
reactiv --> 5
reactive --> 3
```

Conclusion: Wrapping the omnipresent but hard to use asynchronous method pattern in an observable sequence is easy with the FromAsyncPattern method. Specifying the Begin and End method pair, one gets back a function that can be called to obtain an observable sequence providing the results in an asynchronous manner. Using this technique, we illustrated how to wrap a WCF proxy's asynchronous method to call a dictionary service. Finally, we pinpointed a couple of asynchronous computing caveats such as errors and timing issues. Both of those will be tacked in the next exercise.

Exercise 8 – SelectMany: the Zen of composition

Objective: With a tamed input sequence and a bridge function for an asynchronous web service call, we're ready to glue together both pieces into a single application. For every term entered by the user, we want to reach out to the service to obtain word suggestions. In doing so, we'll have to make sure to deal with potential out-of-order arrival of results as exposed in the previous exercise.

1. First, we'll extend the application's UI to include a ListBox control that will be populated with the results that come back from the web service. Depending on the reader's preferences, a UI designer can be used or the following code can be pasted in the Main method:

```
var txt = new TextBox();
var lst = new ListBox { Top = txt.Height + 10 };
var frm = new Form {
    Controls = { txt, lst }
};
```

Assuming the Application.Run call is still in place, the following UI should be displayed:
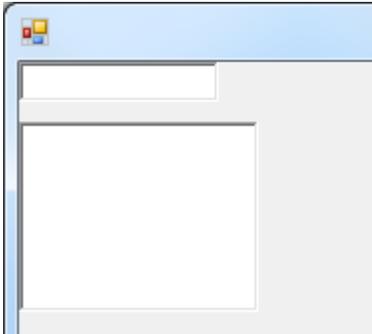
2. Also restore the following piece of code that's used to obtain user input in a throttled and free-of-duplicates manner:

```
var input = (from evt in Observable.FromEvent(txt, "TextChanged")
             select ((TextBox)evt.Sender).Text)
            .Throttle(TimeSpan.FromSeconds(1))
            .DistinctUntilChanged()
            .Do(x => Console.WriteLine(x));
```

Different from previous exercises, we're not going to subscribe to the input sequence directly. The Do operator added at the end will be used to visualize the requests that will be sent to the web service further on.

Note: This use of the Do method doesn't need to take in a lambda expression but sometimes can take a method group instead. In fact, this form of assigning to a delegate has been available since C# 1.0. In the above we could simply .Do(Console.WriteLine). Now that we have lambda expressions, people tend to forget about this syntax and write .Do(x => Console.WriteLine(x)) all the time. Shortening the lambda form to the method group is in fact rooted in a well-known concept in functional programming theory, known as *eta reduction*.

3. Next, put the Rx-based web service wrapper in place:

```
var svc = new DictServiceSoapClient("DictServiceSoap");
var matchInDict = Observable.FromAsyncPattern<string, string, string, DictionaryWord[]>
    (svc.BeginMatchInDict, svc.EndMatchInDict);

Func<string, IObservable<DictionaryWord[]>> matchInWordNetByPrefix =
```

```
term => matchInDict("wn", term, "prefix");
```

4. At this point we got two things: an observable sequence of input strings and a function that takes a string and produces an observable sequence containing the corresponding words. How do we glue those two together? The answer to this question lies in the incredibly powerful SelectMany operator. One of its overloads is shown below:

```
Observable.SelectMany(
```

▲ 2 of 5 ▼ (extension) IObservable<TResult> Observable.SelectMany<TSource,TResult>(**this IObservable<TSource> source**, Func<TSource,IObservable<TResult>> selector)
    Projects each value of an observable sequence to an observable sequence and flattens the resulting observable sequences into one observable sequence.

It turns out we got all the ingredients SelectMany needs to do its magic: we got an IObservable<TSource> which in our case contains strings. We also got a function mapping those strings on an IObservable<TResult>. What the SelectMany operator can do using those inputs is *bind* them together.

Most readers will be familiar with this operator in a possibly unconscious manner. Every time you query some database traversing a relationship between tables, you're really dealing with SelectMany. For example, assume you want to get all the suppliers across all the products in a store. Starting from a sequence of Product objects and a way to map a Product onto a Supplier (e.g. a function retrieving a Product's SuppliedBy property), the operator can give us a flattened list of Supplier objects across all Product objects. The following figure illustrates this binding operation for our scenario of dictionary suggest:
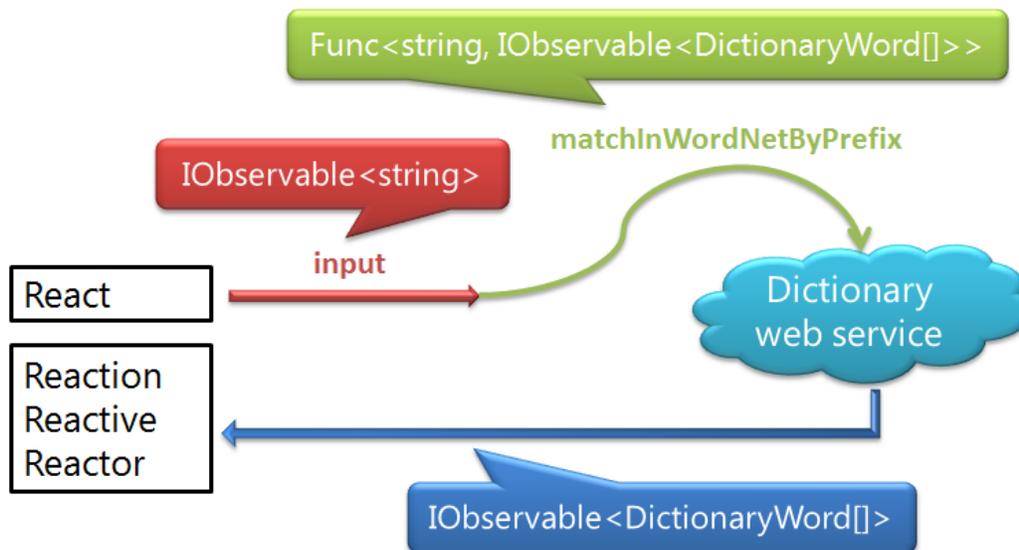
Note: The cardinality of the web service output may be a bit confusing at first. Being an IObservable sequence containing an array of DictionaryWord objects, subscribing means you'll get zero or more DictionaryWord arrays in an asynchronous manner. In this particular scenario, we'll receive a single array which will contain matching words (possibly empty). If a service would exist that notifies the client about results matching the query as they are found during a dictionary scan, the resulting type could be an IObservable<DictionaryWord>.

5. Let's turn SelectMany into motion now. Instead of using the operator as a method call, we can use C#'s query expression syntax where the use of SelectMany is triggered by the presence of multiple from clauses. This gets translated into one of the SelectMany overloads:

```
var res = from term in input
          from words in matchInWordNetByPrefix(term)
          select words;
```

This is all we need to do to compose the two asynchronous computations, the result still being asynchronous by itself as well. Herein lays the power of Rx: retaining the asynchronous nature of computations in the presence of rich composition operators (or "combinators").

Background: SelectMany is one of the most powerful operators of Rx. Its power is rooted in the underlying theory of monads, leveraged by LINQ in general. While monads may sound like a scary disease, they really are a rather simple concept. In the world of monads, the SelectMany operation is called *bind*. Its purpose in life is to take an object "in the monad", apply a function to it to end up with another object "in the monad". It's basically some kind of function application on steroids, threading a concern throughout the computation.

To put it more concrete, take the case of LINQ to Objects. Here the monad is IEnumerable<T>. Given such a sequence and a function to map an element T onto another IEnumerable<R> sequence, we can get a flattened IEnumerable<R> sequence across all of the original sequence's T elements. Feel free to think products and suppliers if it helps. In Rx, the only difference is that we're now dealing with IObservable sequences. No matter which monad we choose, the characteristic of SelectMany stays the same: start in the monad, perform some function, and you're still in the monad. An everyday sample is clock arithmetic. Starting with the second hand in the range 0 through 59, applying any function (e.g. add 10 seconds) will continue to provide a result that falls within the clock range. You can't fall off the clock (read: you can't leave the monad)!

6. The next step is to bind the results to the UI. In order to receive the results, we got to subscribe the resulting observable sequence. No matter how complex the composition is we're talking about, the result is still lazy. In this case, the SelectMany use has returned an IObservable<DictionaryWord[]> which won't do anything till we make a call to Subscribe. From that point on, throttled user input will trigger web service calls whose results are sent to the observer passed to Subscribe:

```
using (res.Subscribe(words =>
{
    lst.Items.Clear();
    lst.Items.AddRange((from word in words select word.Word).ToArray());
}))
    Application.Run(frm);
```

In this piece of code, we receive the words in the OnNext handler. After clearing the elements in the ListBox control, we use regular LINQ to Objects over the array of DictionaryWord objects to get the Word property (which is of type string) and pass an array of words to AddRange. This additional step is needed since the dictionary service hands back DictionaryWord objects which do not only contain the word itself, but also provide the dictionary the word was found in (usable information when searching multiple dictionaries using other web methods).

7. One problem that lurks around the corner is the thread the OnNext handler is called on. Since we're receiving data from the web service in an asynchronous manner, a threadpool thread will be used by WCF to deliver the results to our code.

```
var res = from term in input
          from word in matchInWordNetByPrefix(term)
          select word;

using (res.Subscribe(words =>
{
    lst.Items.Clear();
    lst.Items.AddRange((from w
}))
    Application.Run(frm);
```

⚠ **InvalidOperationException was unhandled by user code**

Cross-thread operation not valid: Control '' accessed from a thread other than the thread it was created on.

Troubleshooting tips:

How to make cross-thread calls to Windows Forms controls
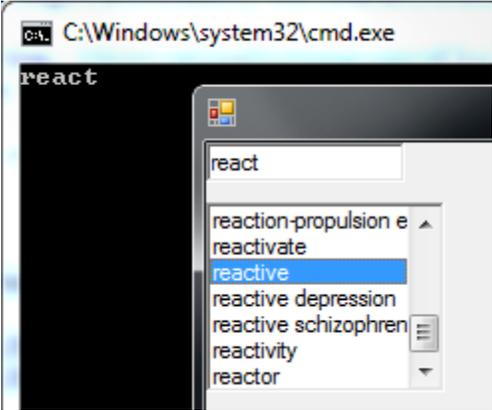
Get general help for this exception.

By now, the reader should know how to fix this issue: use ObserveOn.

```
using (res.ObserveOn(lst).Subscribe(words =>
{
    …
```

With this fix in place, the result of running the code is shown below. Notice the output of the Do operator:

8. However, there's two more problems waiting to get us. We pointed those out in the previous exercise. Let's start with the simpler of the two: what if the service returns an error? In such a case, Rx's SelectMany operator will propagate the exception down to the observer's OnError handler. Since we don't have such a handler in our code (yet), the exception will bring down the application. To solve this issue, we add an OnError handler:

```
using (res.ObserveOn(lst).Subscribe(
    words =>
    {
        lst.Items.Clear();
        lst.Items.AddRange((from word in words select word.Word).ToArray());
    },
    ex =>
    {
        MessageBox.Show(
            "An error occurred: " + ex.Message, frm.Text,
            MessageBoxButtons.OK, MessageBoxIcon.Error
        );
    }))
{
    Application.Run(frm);
}
```

Entering a single letter, resulting in an excessive return data volume, triggers an exception in the WCF stack,

which now gets handled by the error handler code passed to Subscribe:
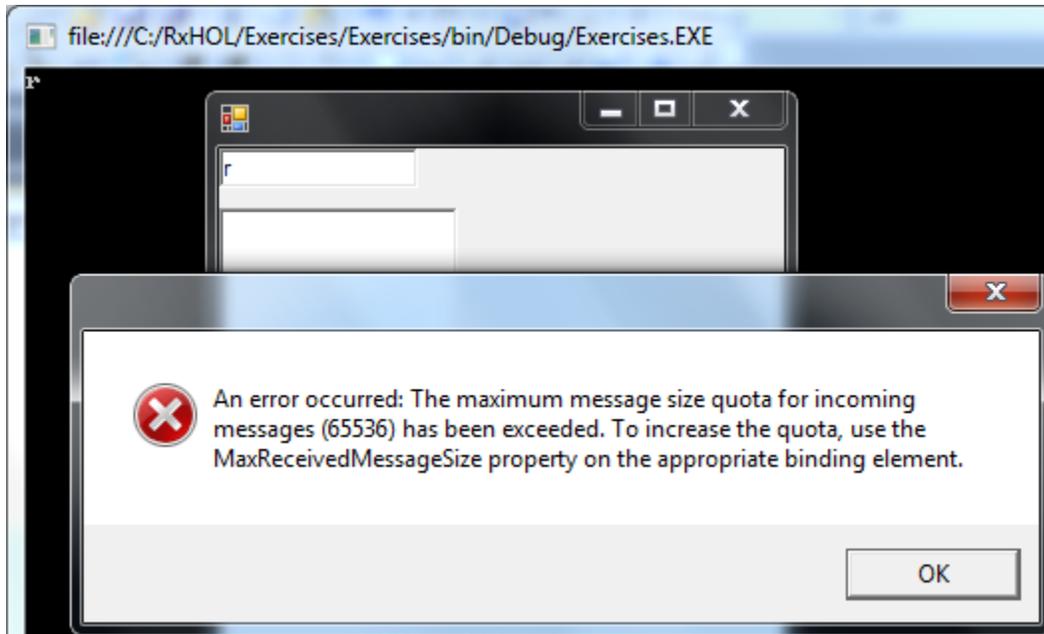
Note: After having received the exception, you are no longer receiving any other notifications.

9. A seemingly tougher problem is that of out-of-order arrival as mentioned at the end of the previous exercise. To understand why this can happen at all in the context of our composition using SelectMany, we need to elaborate on the working of the operator. Whenever the SelectMany operator receives an input on its source, it evaluates the selector function to obtain an observable sequence that will provide data for the operator's output. Essentially it flattens all of the observable sequences obtained in this manner into one flat resulting sequence.

For example, if the user types "react" and idles out for at least one second, SelectMany receives the string from the Throttle operator preceding it. Execution of the selector function - matchInWordNetByPrefix – causes a web service call to be started. While this call is in flight, the user may enter "reactive" which may end up triggering another web service call in a similar manner (one second throttle delay, application of the selector function). At that point, two parallel calls are happening, which could provide results out-of-order compared to the input. The figure below illustrates the issue that can arise due to this behavior:
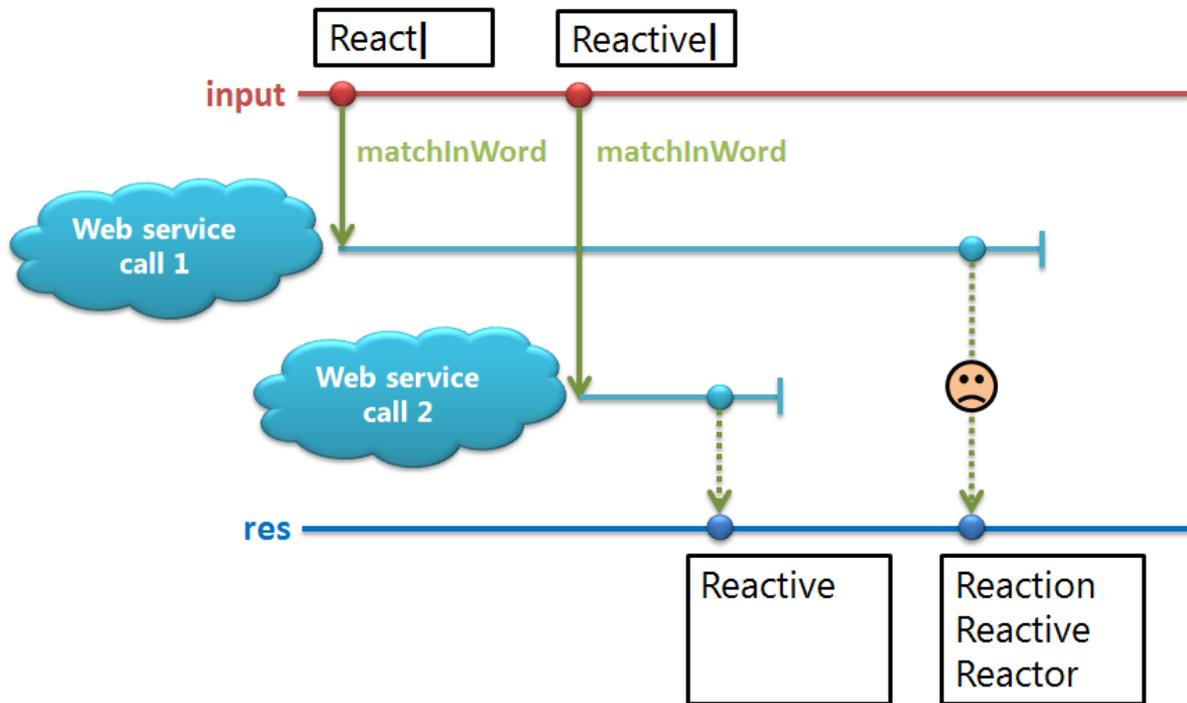
Since throttling adds a one second delay, you have to be a bit (un)lucky to hit this issue. However, if it remains unfixed, it'd likely come and get you the first time you demo the application to your boss. To show the issue's presence, take out the Throttle operator and type the word "reactive" without hesitation. In order to avoid hitting the message size limit, filter out short terms as well:

```
var input = (from evt in Observable.FromEventPattern(txt, "TextChanged")
             select ((TextBox)evt.Sender).Text)
            .Where(term => term.Length >= 3)
            //.Throttle(TimeSpan.FromSeconds(1))
            .DistinctUntilChanged()
            .Do(Console.WriteLine);
```

After a few attempts you should an out-of-order response issue, such as the one shown below:
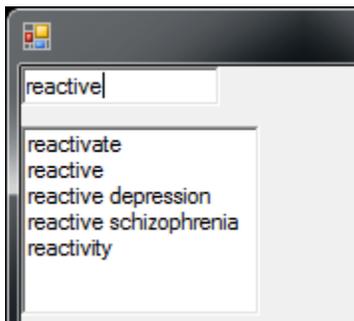
In order to solve this issue, we need to cancel out existing web service requests as soon as the user enters a new term, indicating there's no further interest in the previous search term. What we want to achieve is illustrated in the figure on the next page. The essence of the fix is "crossing out" or "muting" in-flight requests when a new one is received. This is illustrated as the red cross on the line for the first web service call. It turns out applying this fix is incredibly easy using Rx operators; moreover, there are a couple of ways to solve the issue. We'll show two different approaches here.
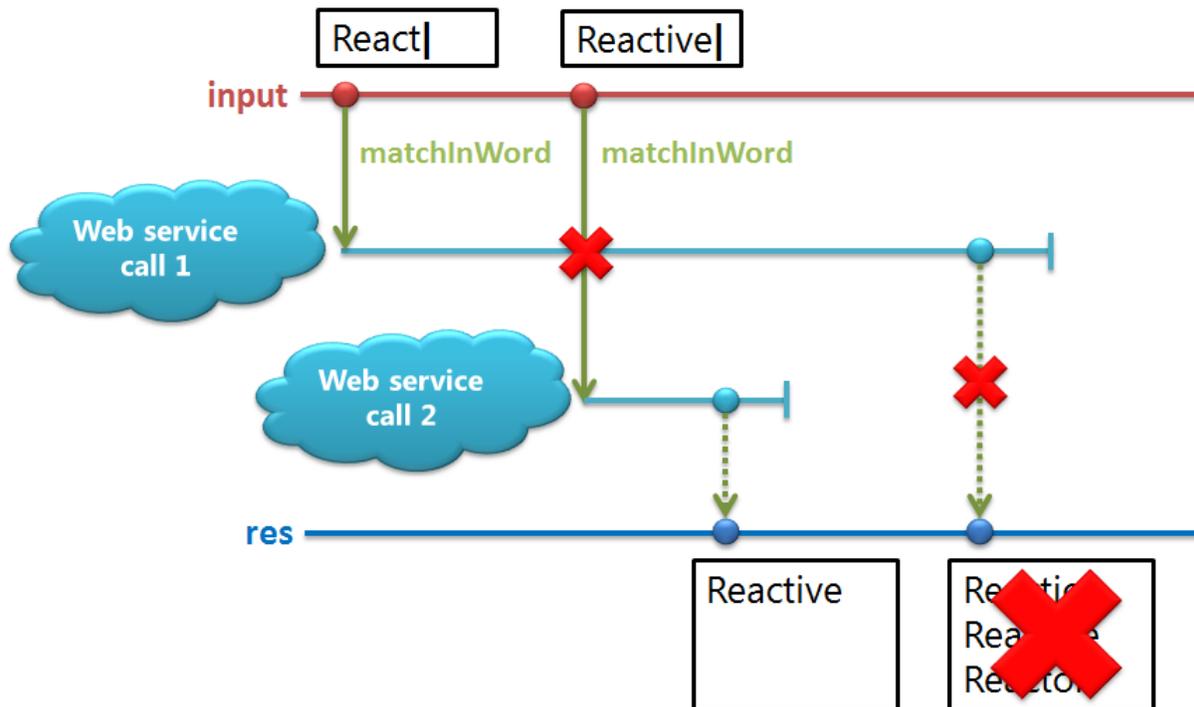
Note: At some point in the design of Rx, there was a special SelectMany operator with cancellation behavior: whenever an object was received on the source, the previous (if any) inner observable was unsubscribed before calling the selector function to get a new inner observable. As you may expect, this led to numerous debates which of the two behaviors was desired. Having two different operator flavors for SelectMany was not a good thing for various reasons. For one thing, only one flavor could be tied to the C# and Visual Basic LINQ syntax. The ultimate solution was to decouple the notion of cancellation from the concept of "monadic bind". As a result, new cancellation operators arose, which do have their use in a lot of other scenarios too. A win-win situation!

10. The realization of this cancellation behavior can be achieved using a single operator called TakeUntil whose behavior is to take elements from an observable sequence until another "signal" observable tells it to stop. One can compare it to a blowout preventer on an oil pipeline. While distasteful jokes about oil could be made, we'll retain ourselves from doing so. Suffice to say our operator *does* work and has been well-tested.
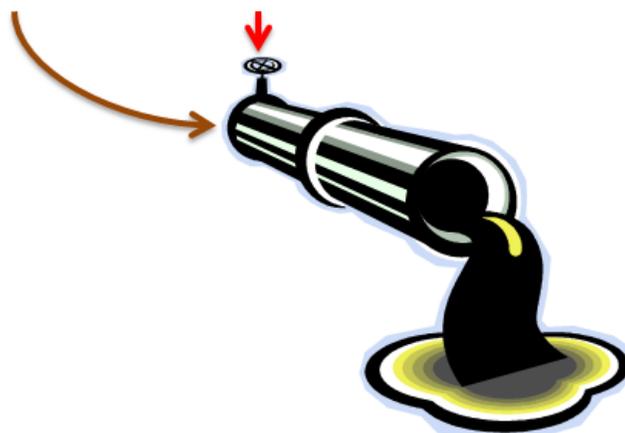
What should be the valve to shut down receiving results from an ongoing web service request in our case? The user entering a new input string is what we need. So, all we have to do is stick a TakeUntil call on the result of

calling the web service method. Let's put it to the test and tweak our code as shown below:

```
var res = from term in input
          from word in matchInWordNetByPrefix(term).TakeUntil(input)
          select word;
```

Recapping what this code does, the input sequence produces terms based on (throttled) user input. When a term is received from the input, the SelectMany operator hidden behind the second from keyword kicks in and executes the selector function passing it the received term:
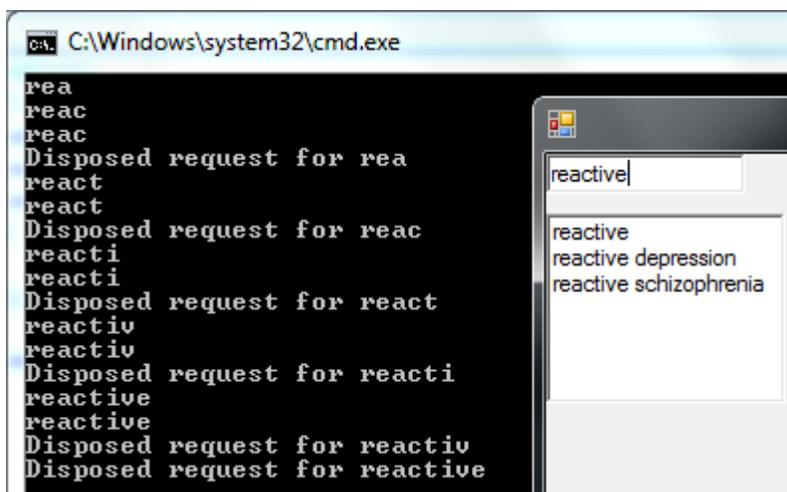
```
term => matchInWordNetByPrefix(term).TakeUntil(input)
```

In here, the resulting observable sequence is provided by the TakeUntil operator which on its turn consumes input from a web service call. It's important to notice the result of calling the selector function is an observable sequence which hasn't been subscribed to yet, so the web service call is not taking place yet. However, the very next thing the SelectMany operator does is subscribing to the observable obtained by the selector function. At that point, TakeUntil does its job. It subscribes to both its source (which is the web service call getting made, ticking away to produce its array of DictionaryWord objects) and the "valve" observable. From this point on, the TakeUntil operator will pass on any of the data it receives until the valve is signaled. In this case, the valve simply corresponds to user input. As a result, the next time the user types something, the valve is closed for the web service request that's in flight. When SelectMany also sees the user's new input, it repeats the steps we mentioned here. And so on…

Visualizing the cancellation behavior can be achieved using another operator. Where Do monitors data flowing through an observable "pipeline", the Finally operator can be used to perform an action when a disposal of the subscription happens. While it's normally used for clean-up operations, it comes in handy for the kind of logging we need here:

```
var res = from term in input
          from word in matchInWordNetByPrefix(term)
                       .Finally(() => Console.WriteLine("Disposed request for " + term))
                       .TakeUntil(input)
          select word;
```

The result of the above is shown below. When TakeUntil shuts down the valve it gets rid of the source by disposing the subscription it holds to it. This causes Finally to log the message:

Note: Why are we seeing the Do-based logging for input twice now? The answer is pretty straightforward: both

the SelectMany operator (in the first from clause) and the TakeUntil operator are subscribed to the input sequence. When side-effects are involved with subscriptions (e.g. paying a fee), you may want to share a single underlying subscription. Operators such as Publish exist for this purpose but fall outside the scope of this HOL.

The resulting code for the entire application fits on less than a page. Imagine doing the same composition of events with asynchronous web services in a concise manner, taking care of all the edge cases we discussed:

```csharp
var txt = new TextBox();
var lst = new ListBox { Top = txt.Height + 10 };
var frm = new Form {
    Controls = { txt, lst }
};

// Turn the user input into a tamed sequence of strings.
var textChanged = from evt in Observable.FromEventPattern(txt, "TextChanged")
                  select ((TextBox)evt.Sender).Text;

var input = textChanged
            .Throttle(TimeSpan.FromSeconds(1))
            .DistinctUntilChanged();

// Bridge with the web service's MatchInDict method.
var svc = new DictServiceSoapClient("DictServiceSoap");
var matchInDict = Observable.FromAsyncPattern<string, string, string, DictionaryWord[]>
    (svc.BeginMatchInDict, svc.EndMatchInDict);

Func<string, IObservable<DictionaryWord[]>> matchInWordNetByPrefix =
    term => matchInDict("wn", term, "prefix");

// The grand composition connecting the user input with the web service.
var res = from term in input
          from word in matchInWordNetByPrefix(term).TakeUntil(input)
          select word;

// Synchronize with the UI thread and populate the ListBox or signal an error.
using (res.ObserveOn(lst).Subscribe(
    words => {
        lst.Items.Clear();
        lst.Items.AddRange((from word in words select word.Word).ToArray());
    },
    ex => {
        MessageBox.Show("An error occurred: " + ex.Message, frm.Text,
                        MessageBoxButtons.OK, MessageBoxIcon.Error);
    }))
{
    Application.Run(frm);
} // Proper disposal happens upon exiting the application.
```
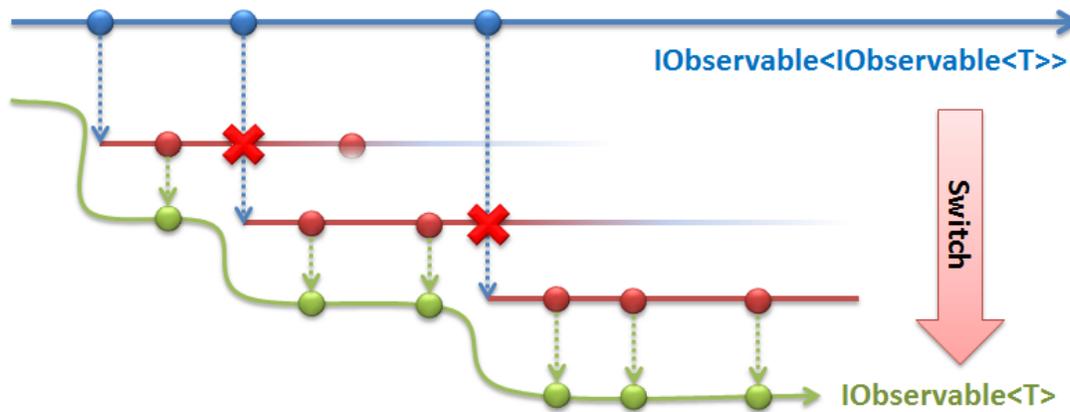
11. Since this issue is quite common, a specialized operator called Switch was introduced. Given a sequence of sequences (yes, that's not a typo) it hops from one sequence to another as they come in. Using a line diagram this can be made clear in an easy manner:

When the topmost observable "outer" sequence produces a new observable "inner" sequence, an existing inner sequence subscription is disposed and the newly received sequence is subscribed to. Results produced by the current inner sequence are propagated to the Switch operator's output.

Use of this operator in our scenario proceeds as follows. First we map user input on web service requests using a simple Select operator use. Since every web service request returns an IObservable<DictionaryWord[]>, the result of this projection is an IObservable<IObservable<DictionaryWord[]>>. Applying Switch over this nested sequence causes the behavior described above. For every request submitted by the user, the web service is contacted. If a new request is made, Switch cancels out the existing one's subscription and hops to the new one:

```
var res = (from term in input
           select matchInWordNetByPrefix(term))
          .Switch();
```

Which approach one choses is solely dependent on personal taste. Both are equally good at solving the issue.

Conclusion: Composition of multiple asynchronous data sources is one of the main strengths of Rx. In this exercise we looked at the SelectMany operator that allows "binding" one source to another. More specially, we turned user entries of terms (originating from .NET events) – into asynchronous web service calls (brought to Rx using FromAsyncPattern). To deal with errors and out-of-order arrival only a minimal portion of the code had to be tweaked. While we didn't mention operators other than SelectMany, TakeUntil and Switch that deal with multiple sources, suffice it to say that a whole bunch of those exist awaiting your further exploration.

Exercise 9 – Testability and mocking made easy

Objective: Testing asynchronous code is a hard problem. Representing asynchronous data sources as first-class objects implementing a common generic interface, Rx is in a unique position to help to simplify this task as well. We'll learn how easy it is to mock observable sequences to create solid tests.

1. Each observable sequence can be regarded as a testable unit. In our dictionary suggest sample, two essential sequences are being used. One is the user input sequence; the other is its composition with the web service. Since all of those are first-class objects, we can simply swap them out for a sequence "mock".

   One particularly useful operator in Rx is ToObservable defined as an extension method on IEnumerable<T>. It's a pull-to-push adapter that enumerates (pulls) the given sequence and exposes it as an observable sequence that notifies (pushes) its observers of the sequence's elements. As an example, replace the input sequence for an array-based mock observable as shown below:

   ```
   //var input = (from evt in Observable.FromEventPattern(txt, "TextChanged")
   //             select ((TextBox)evt.Sender).Text)
   //            .DistinctUntilChanged()
   //            .Throttle(TimeSpan.FromSeconds(1));
   var input = new[] { "reac", "reactive", "bing" }.ToObservable();
   ```

Background: The deep duality between IEnumerable<T> and IObservable<T> has yielded a lot of great results in the development of Rx. The ability to go back and forth between both models using the ToObservable and the ToEnumerable operators is just one sample of this. Because of the intrinsically concurrent nature of observable sequences, those operators exhibit some interesting properties. ToObservable –as shown here – introduces more concurrency since it needs to pull the enumerable sequence without blocking the caller, in order to feed the elements to the resulting observable. ToEnumerable on the other hand reduces concurrency as all the observable sequence's elements are tunneled onto the enumeration thread. An implication of the push-to-pull conversion realized by ToEnumerable is that it may have to buffer (enumeration may proceed at a different pace from the observable producer), while ToObservable never has to buffer. True mirror images: duality at its best!

2. Now when we try to run the application, our web service will be fed "reac", "reactive" and "bing" virtually at the same time since there's no delay between the elements in the input. If the TakeUntil or Switch operator does its job correctly, only the results for the "bing" request should appear on the screen.
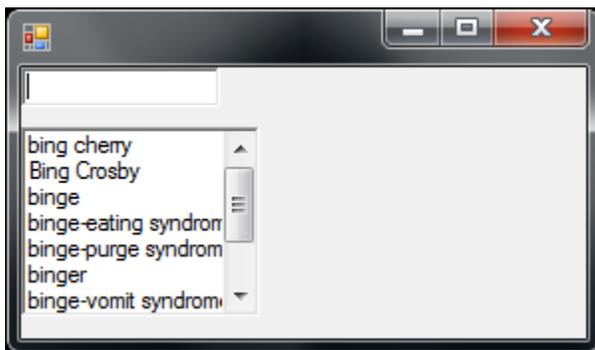
   It's a worthy experiment to take out the out-of-order arrival prevention mechanism from the previous exercise (i.e. TakeUntil or Switch) and observe responses coming back. With a bit of luck, you'll see the issue cropping up again. A sequence with a higher likelihood of reproducing the issue is the following, using LINQ to Objects:

   ```
   var input = (from len in Enumerable.Range(3, 8)
                select "reactive".Substring(0, len)) // rea, reac, react, reacti, reactiv, reactive
   ```

```
        .ToObservable();
```

3. Thanks to the various time-based operators, we can provide more realistic input. One thing we could do to mock user input in a more faithful way is to use time-based operators to mimic typing. Generate is a suitable candidate for our next experiment. Let's generate a sequence of incremental substrings for a given term, with random delays between them.

```csharp
const string INPUT = "reactive";

var rand = new Random();

var input = Observable.Generate(
            3,
            len => len <= INPUT.Length,
            len => len + 1,
            len => INPUT.Substring(0, len),
            _ => TimeSpan.FromMilliseconds(rand.Next(200, 1200))
        )
        .ObserveOn(txt)
        .Do(term => txt.Text = term)
        .Throttle(TimeSpan.FromSeconds(1));
```

GenerateWithTime uses a random number generator to simulate typing speed variations between 200 and 1200 milliseconds (such that Throttle will sometimes let a substring through). Before we throttle the sequence, we use the side-effecting Do operator to put the substring in the TextBox control to really simulate the user typing in a visual manner. To be able to do this, we have to use ObserveOn to be on the right UI context.

Note: While typing the code above you may notice a Subscribe method in the IntelliSense list for the input string. This extension method on IEnumerable<T> is yet another sample of the dual treatment of enumerable and observable sequences and is closely related to ToEnumerable and ToObservable. Similarly, a GetEnumerator method is defined on IObservable<T>.

```
len => INPUT.Sub|
          ┌────────────────────────────┐
          │ ◆ Subscribe<>              │
          │ ◆ Substring                │
          └────────────────────────────┘
```

Adding another Do logger to see Throttle passing on a search term to the web service is left as an exercise for the reader.

4. Similarly, we could mock the web service by replacing the matchInWordNetByPrefix function. Obviously one will have to come up with some output to go with the input term, maybe from a local dictionary or based on some dummy output generation. Below is a sample web service mock:

```csharp
Func<string, IObservable<DictionaryWord[]>> matchInWordNetByPrefix =
    //term => matchInDict("wn", term, "prefix");
    term =>
        Observable.Return(
            (from i in Enumerable.Range(0, rand.Next(0, 50))
             select new DictionaryWord { Word = term + i })
            .ToArray()
        ).Delay(TimeSpan.FromSeconds(rand.Next(1, 10)));
```

Reading the code inside-out, we first generate a range of 0 to 50 numbers which we append to the service's input using a projection. At this point, we're using the established LINQ to Objects functionality. As a result, we end up with an array of some number of outputs. For example, given "rea", we may get { "rea0", "rea1", "rea2" } back. Since the web service contract is to return an observable of such an array, we use Observable.Return to

create a single-element observable sequence with the generated array. Finally, we use the Delay operator that's defined for observable sequences to add a random 1-to-10 second delay in sending back the response.

Running the sample again without the out-of-order prevention, it should be plain easy to hit the issue we have described in much detail before. Assume such an issue has been uncovered in your code; it's incredibly simple to create a test case for it using mocks like those.
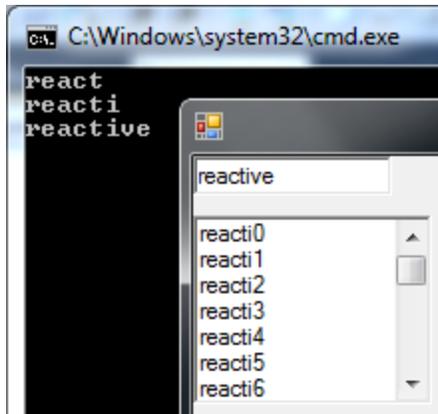
Conclusion: The first class object nature of observable sequences makes it easy to replace them, contrast to various asynchronous technologies like .NET events. This allows for smooth testing of asynchronous programs using mock input sequences, e.g. based on enumerable sequences turned into observable ones using ToObservable.