AN EVALUATION OF THE REACTIVE EXTENSIONS (RX) LIBRARY AS A TOOL IN DESIGNING AND CODING ASYNCHRONOUS APPLICATIONS

Authors
Samay Mahajan
Basab Das Gupta
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Executive Summary</td>
<td>4</td>
</tr>
<tr>
<td>Intended Audience</td>
<td>4</td>
</tr>
<tr>
<td>What Do Reactive Extensions Offer?</td>
<td>5</td>
</tr>
<tr>
<td>Benefits of Using Rx</td>
<td>5</td>
</tr>
<tr>
<td>Usage Scenarios Where Rx is Highly Relevant</td>
<td>6</td>
</tr>
<tr>
<td>Visualizing and Designing the Asynchronous Paradigm with Rx</td>
<td>6</td>
</tr>
<tr>
<td>A Look at Rx API Design</td>
<td>8</td>
</tr>
<tr>
<td>Real-Life Problem</td>
<td>13</td>
</tr>
<tr>
<td>Limitations of Rx</td>
<td>14</td>
</tr>
<tr>
<td>Conclusion</td>
<td>14</td>
</tr>
<tr>
<td>Appendix A: Getting Started with Rx</td>
<td>15</td>
</tr>
<tr>
<td>Appendix B: Understanding Asynchronous Programming</td>
<td>22</td>
</tr>
<tr>
<td>Appendix C: Rx Extensibility</td>
<td>23</td>
</tr>
<tr>
<td>Downloads</td>
<td>24</td>
</tr>
<tr>
<td>References</td>
<td>24</td>
</tr>
</tbody>
</table>
EXECUTIVE SUMMARY

As businesses face the increasing need to respond to and trigger actions based on data events, technologies have evolved to execute these events in an asynchronous fashion. Multiple technologies have been available to execute asynchronous tasks or programs; Component Object Model (COM) is one of the earliest examples. More recently, Reactive Extensions (commonly known as “Rx”), an offering from Microsoft DevLabs, can be leveraged to execute asynchronous tasks or programs. The Rx library accelerates time to market for asynchronous and event-based programs that have robust architectures. Rx can be used across the .NET platform, including WPF, Silverlight and Windows Phone. It also exposes a port to JavaScript.

The purpose of this white paper is to evaluate the Rx library as a tool for developing applications for financial markets. With the help of some use cases, this white paper addresses:
- Evolution of the Rx object model to implement event-driven applications
- Key architectural concepts of Rx to implement Rx applications
- Demonstration example: File monitoring mechanism using Rx
- Relevant usage scenarios of Rx

INTENDED AUDIENCE

This white paper is intended for .NET developers and architects who are already familiar with the following concepts:
- Observer design pattern
- Multi-threading in .NET
- Iterator pattern in .NET (IEnumerable/IEnumerator), Lambda Expressions and LINQ
- Event-driven application development
- Asynchronous application development using .NET
LINQ is often used during application development. A LINQ-based sequence operator offers extensive flexibility to filter, query and identify data conditions. However, it does not offer built-in capabilities for event-based mechanisms to push or trigger events. This requirement for a mechanism that would recognize events and respond appropriately in programming is what led to the implementation of Rx for .NET.

A superset of LINQ, Rx implements a push-based mechanism represented by the new interfaces IObserver<T> and IObservable<T> for asynchronous and event-based computations. These interfaces are mathematically dual of the pull-based IEnumerable<T> and IEnumerator<T>. Similar to LINQ, Rx provides an application programming interface (API) for handling events by allowing operators to aggregate, filter and query event flows in a concise and familiar way.

Over enumerable sequences, LINQ allows developers to consume and transform values from a wide range of pull-based sequences, such as in-memory collections, database tables and XML documents. In the same way, developers can frame complex event processing queries over push-based sequences, such as .NET events, Task<T>-based computations, Windows Phone 7 sensors, the Windows 7 Sensor and Location APIs, F# first-class events, APM-based (“IAsyncResult”) computations, SQL StreamInsight temporal event streams and asynchronous workflows by implementing Rx over observable sequences.

Apart from the push-based programming using Rx for .NET, there is push-based programming in browser-based events, as well. Rx for Javascript (RxJS) is a new feature introduced in the Rx library. RxJS provides observable collections to use with DOM, XmlHttpRequest and jQuery events. In addition, it provides easy-to-use conversions from existing DOM, XmlHttpRequest and jQuery events to observable collections, thereby allowing users to seamlessly plug RxJS into existing JavaScript-based web sites.

**WHAT DO REACTIVE EXTENSIONS OFFER?**

- **Simpler, more expressive asynchronous implementations.** The Rx APIs allow a uniform approach to structure asynchronous flows using the observer pattern. Developers can do so by exposing a LINQ-like library around the interface pair IObserver<T> and IObservable<T>. By providing cleaner and more expressive asynchronous code, this library wraps up existing .NET infrastructures used in asynchronous programming—including events, thread-pool and Task Parallel Library (TPL).

- **Reduced learning curve.** .NET developers who are familiar with LINQ can easily apply their knowledge along with the observer pattern to understand how to compose observables using Rx libraries. (Composing observables in Rx libraries is similar to composing collections using LINQ library.)

- **Easier method for designing asynchronous flows.** While designing asynchronous applications, such features as events, threads and tasks can be implemented asynchronously. However, the level of care and attention that should be taken when implementing these features is very high. Rx simplifies this thought process by using observables and observers as the common nomenclatures and internally implementing thread-safe data sharing.

**BENEFITS OF USING RX**

- **Simpler, more expressive asynchronous implementations.** The Rx APIs allow a uniform approach to structure asynchronous flows using the observer pattern. Developers can do so by exposing a LINQ-like library around the interface pair IObserver<T> and IObservable<T>. By providing cleaner and more expressive asynchronous code, this library wraps up existing .NET infrastructures used in asynchronous programming—including events, thread-pool and Task Parallel Library (TPL).

- **Reduced learning curve.** .NET developers who are familiar with LINQ can easily apply their knowledge along with the observer pattern to understand how to compose observables using Rx libraries. (Composing observables in Rx libraries is similar to composing collections using LINQ library.)

- **Easier method for designing asynchronous flows.** While designing asynchronous applications, such features as events, threads and tasks can be implemented asynchronously. However, the level of care and attention that should be taken when implementing these features is very high. Rx simplifies this thought process by using observables and observers as the common nomenclatures and internally implementing thread-safe data sharing.
**USAGE SCENARIOS WHERE RX IS HIGHLY RELEVANT**

- **Using Rx as organized asynchronous and event-based computations.** Imagine a scenario where more than one asynchronous execution has to be triggered for a particular event. In such cases, it is extremely difficult to code asynchronous executions. Developers can use Rx to build a state machine to deal with the frequent ordering of events. A classic example for a similar scenario in technical terms is an auto-complete textbox.

- **Using Rx to consume continuous asynchronous streams of data.** Conventional asynchronous operations in the .NET platform are relevant and very useful when an operator returns a single message and it needs to be worked on. Rx is primarily designed where continuous streams of data are being delivered in the lifetime of the operator and events are processed for the operator. Reading trading data and responding to events are classic examples where Rx can be relevant.

- **Eliminating blocking in web services.** In a cloud-based scenario, each web service request needs to be designed as a non-blocking request; this avoids waiting time from the calling thread. Though one web service can make a call to the second and the second to the third, there will be a delay in the process. Therefore, a request-response pair can be chosen. Accordingly, the web service registers an incoming request and moves on to the next request. When the registered request has a response ready, it notifies the same with a response as in the payload. Using Rx, the first web service could thus be an observer of the second web service as well as an observable for another web service that consumes it.

- **Leveraging Rx in concurrent programming.** Content Addition, so find below the new content for this point
  Leveraging Rx in concurrent programming. In concurrent programming scenarios, composite event notifications may be an interesting addition. It helps to optimize the resources for multi-threaded and asynchronous architecture applications. Suppose that two threads are spawned to do some concurrent processing from the main thread of execution and that each of these threads raises an event to mark the completion. In this case, one expects a notification about this completion as a single event to the main thread to spawn yet another thread to do more concurrent processing. When this thread notifies completion, the main thread takes over to do the final processing.

**VISUALIZING AND DESIGNING THE ASYNCHRONOUS PARADIGM WITH RX**

Rx offers a new paradigm for how asynchronous event-based programs are visualized, designed and delivered. To leverage Rx efficiently, it is important to understand some key design paradigms, such as the concept of duality.

**Concept of Duality**

The concept of duality is mathematical in origin but is used in diverse fields—from formal logic (De Morgan’s law) to calculus and analytics (limit and colimit of functions). In a nutshell, the principle of duality conveys that the same result can be arrived at using two different ways because they logically complement each other.

Rx has used duality to infer its core APIs (IObservable<T> and IObserver<T>) and to introduce a push-based, asynchronous approach to dealing with collections and streams as opposed to the existing pull-APIs (IEnumerable<T> and IEnumerator<T>). This, in turn, has helped in introducing into Rx the LINQ operators designed around the IEnumerable interface. In fact, the dual nature has caused some newly introduced Rx operators to also be introduced back into LINQ. Duality helps developers with LINQ experience to use the same query language in Rx, thus reducing
the learning curve. Also, duality enables a project refactored to have asynchronous behavior to refactor the LINQ queries to Rx queries on a one-to-one basis. These are some of the positive effects that duality has rendered to the design of Rx.

Another place where duality is realized in Rx API is with the IAsyncEnumerable and IAsyncEnumerator interfaces. Though pull based, these interfaces introduce concurrency by taking advantage of tasks in framework 4.0, thus making them more efficient than their synchronous counterparts. This interface pair is not a substitute of IObservable and IObserver.

**Visualizing Event Flows as a Marble Diagram**

While using Rx to build applications, sequences of events based on IObservable will be created. One of the key artifacts recommended to be prepared for designing an Rx application is a marble diagram. A sample marble diagram is illustrated below (see Figure 1).

![Marble Diagram](image1)

When observable event streams are composed, it is easy to visualize and discuss the streams using marble diagrams. Marble diagrams are also helpful in understanding the resulting stream behavior of different observable operators provided in the Rx. The event stream should visualize the following:

- OnNext
- OnError
- OnCompleted

A marble diagram can be understood as follows:

- Each horizontal line represents the timeline of a single IObservable stream.
- The ellipse represents a single occurrence of a new value event (OnNext).
- The vertical line represents the end of the stream sequence (OnComplete).
- The X sign represents exception (OnError).

A marble diagram can have more than one horizontal line representing individual streams. Below is a marble diagram of the zip operator:

![Zip Operator Diagram](image2)

The zip operator here combines two streams: xStream and yStream. The zip operation results in the observable stream. When the first event has occurred on both of the input streams, the OnNext event is fired on the result stream.

*Image courtesy—marble diagram figure 1:*

*Image courtesy—marble diagram figure 2:*
http://blogs.microsoft.co.il/blogs/bnaya/archive/2010/03/05/rx-for-beginners-part-7-zip-expression.aspx
A LOOK AT RX API DESIGN

Before orchestrating asynchronous events using the Rx library, developers need to address three questions:
1. What is to be observed?
   - The Rx library is used to compose an observable for the same.
2. Who are the observers—that is, who will observe it and get the notifications?
   - These objects should subscribe to the observable.
3. On which thread it is observed?
   - This is specifically important with respect to asynchronous operations dealing with concurrency. The Rx Scheduler is used to manage context-switches. Schedulers will be discussed in more detail under the section “Rx Scheduler & Concurrency Management.”

Some Important Rx Operators

• Projection Operators. Select and SelectMany operators are projection operators used to project another observable stream from a source observable stream. The resulting observable may be processing the source observable items.

```csharp
var oneNumberPerSecond = Observable.Interval(TimeSpan.FromSeconds(1));
var numbersTimesTwo = from n in oneNumberPerSecond
    select n * 2;
```

Restriction Operator (Where). The where operator restricts/filters push items of a source observable from the resultant observable.

Asynchronous Background Operation (Start). The start operator is used to run a piece of code asynchronously on the ThreadPool thread.

ToAsync: This operator is used to run any method asynchronously.

ForkJoin. The ForkJoin operator combines the last values of each concurrent observable stream and generates an OnNext message on the resulting IObservable. There is also a possibility of having input observable streams with different data types.

```csharp
var o = Observable.ForkJoin(Observable.Start(() =>
{
    Console.WriteLine("Executing 1st on Thread: {0}",
        Thread.CurrentThread.ManagedThreadId); return "Result A";
}, Observable.Start(() =>
{
    Console.WriteLine("Executing 2nd on Thread: {0}",
        Thread.CurrentThread.ManagedThreadId); return "Result B";
}, Observable.Start(() =>
{
    Console.WriteLine("Executing 3rd on Thread: {0}",
        Thread.CurrentThread.ManagedThreadId); return "Result C";
})).Finally(() => Console.WriteLine("Done!")));

foreach (string r in o.First())
    Console.WriteLine(r);
```
Result
Executing 1st on Thread: 3
Executing 2nd on Thread: 4
Executing 3rd on Thread: 3
Done!
Result A
Result B
Result C

**FromAsyncPattern.** This operator takes a pair of begin/end methods to be used in an asynchronous operation and creates observables on ThreadPool. Once the operation is over, there will be only one OnNext push. It can be combined with While Operator to multiply the number of item pushes. Often the ObserveOn operator needs to be used to context-switch between ThreadPool threads and the UI thread and to get the notifications on the UI thread.

**Time Related.** Different operators—including Timeout, BufferWithTime, Delay, Defer and Throttle—are available to control the periodicity of the events in the timeline of the observable stream.

Throttle is an operator that will ignore items occurring less frequently than the throttle periodicity. Throttle is useful, especially when an expensive operation like a web-service call needs to be done in the back end as an OnNext action of an observable, which is pumping push events at a very rapid rate. An observable to a mouse-move event is an example. If we use the throttle operation on this observable with a value of 100 milliseconds, the OnNext will be fired every time the mouse is not moved for 100 milliseconds using the last value, which, in this case, will be the last mouse position.

**Combination.** There are many ways to compose observables out of multiple observable streams. Some important combinations are discussed here:

- The **SelectMany** operator allows easy chaining of multiple observable streams. Output of one stream becomes input for another stream, and that stream is observed. In some scenarios, values from one stream need some translation before being passed to another stream. It is not possible with the select operator as it will return IObservable of IObservable (IObservable<IObservable<T>>) and not a flattened IObservable(IObservable<T>), which is actually required.
- The **Zip Operation** gives a resultant stream, where an OnNext event occurs once an OnNext event has occurred on both the input streams.
- In **Merge Operation**, multiple IObservable streams are merged into single IObservable streams. The OnNext operation on the resultant stream occurs with every OnNext occurring on any of the input streams.
- The **Publish Operation** is used to share a subscription with multiple observers. To make an observable of type publish, the publish() operator should be called. This returns an observable having a shared subscription. Additionally, the connect method needs to be called to start receiving notifications. This operator is useful, specifically when there are observables on other threads. Such observables may run until completion as soon as the first subscribe is done; hence, the second subscribe does not get a chance. The facility to subscribe multiple times and then connect avoids this behavior.
• **Switch.** In asynchronous calls, there are often scenarios in which the current stream of notifications has to be cancelled and a new stream will publish the notifications. However, the observer itself is unaware of the need to do all of these. In Rx, this type of scenario can be handled by the switch operator. As seen in the marble diagram below, the observable returned by the switch operator hops from one sequence to another as new sequences get created and maintain a smooth flow of OnNext notifications (the green timeline) based on the OnNext occurring on the live sequence (the red timelines).

![Marble diagram](http://go.microsoft.com/fwlink/?LinkId=208528)

**Rx Scheduler & Concurrency Management**

Generally, any concurrency support infrastructure has a scheduler to schedule task items on threads. The scheduler tackles two decisions:

1. When will a scheduled task be executed?
2. On which thread will it be executed?

For example, in .NET, ThreadPool scheduler has a queue from which it takes tasks and executes them on a ThreadPool thread. Task Parallel Library in 4.0 also has a similar scheduler to schedule tasks on threads.

Rx also has its own scheduler types to manage concurrency and to make the two decisions described above. The main interface to implement a scheduler in Rx is the IScheduler interface. It has the schedule method, which takes the action or job to be scheduled. Rx provides a few out-of-the-box schedulers that implement IScheduler. These schedulers wrap the most used scheduling infrastructure available in .NET and help in giving a uniform treatment to scheduling via the IScheduler interface.

- **Scheduler.Dispatcher.** This interface will ensure the performance of actions on the dispatcher thread. This is very useful for WPF and Silverlight applications. The implementation for Scheduler.Dispatcher would delegate any calls to Schedule(Action) straight to Dispatcher.BeginInvoke(Action).
- **Scheduler.NewThread.** It will schedule all actions onto a NewThread.
- **Scheduler.ThreadPool.** In this, all actions are scheduled onto the ThreadPool.
- **Scheduler.TaskPool.** It is available only in Silverlight 4 and .NET 4.0 and is used to schedule actions onto the TaskPool.
- **Scheduler.Immediate.** It ensures the action is not scheduled but executed immediately.
- **Scheduler.CurrentThread.** It ensures that the actions are performed on the thread that made the original call. This is different from Immediate, as CurrentThread will queue the action to be performed.

*Image courtesy—line diagram: http://go.microsoft.com/fwlink/?LinkId=208528*
In Rx, either of the two given below would be scheduled:

1. The invocation of a subscription
2. The publishing of notifications

If the observable stream and the observer are on the same thread, the subscribe method on the observable is called to connect the observer and provide the OnNext, OnCompleted and OnError callbacks. But suppose the observable is in a ThreadPool thread and the observer is on the UI thread. In this case, the subscription needs to happen on the ThreadPool thread and the push notifications on the UI thread. Therefore, instead of calling subscribe directly, an action on Scheduler.ThreadPool should be scheduled. There are two extension methods on IObservable to help us with this context-switching, which are mentioned below.

```
public static IObservable<TSource> SubscribeOn<TSource>(this IObservable<TSource> source, IScheduler scheduler)
public static IObservable<TSource> ObserveOn<TSource>(this IObservable<TSource> source, IScheduler scheduler)
```

In order to subscribe on the observable in the ThreadPool thread, the following needs to be done.

```
Observable.Range(1, 15)
    .SubscribeOn(Scheduler.ThreadPool) //Subscribe on ThreadPool
    .Subscribe(i => textbox1.Text = i);
```

The SubscribeOn method returns a proxy observable. When the Subscribe method is called on this proxy, the proxy simply calls the Schedule method of IScheduler and schedules the Subscribe operation. But the OnNext expression, i => textbox1.Text = i, will not work. Since OnNext will be called on ThreadPool thread, the UI will not be updated. To context-switch the OnNext notification to the UI thread, the ObserveOn operator is used. So the code above will become:

```
Observable.Range(1, 15)
    .SubscribeOn(Scheduler.ThreadPool) //Subscribe on ThreadPool
    .Subscribe(Scheduler.Dispatcher) //Subscribe on Dispatcher
    .Subscribe(i => textbox1.Text = i);
```

Another use case for ObserveOn operator: composing an observable out of multiple observables running on different threads. For example, the merge operator is used to compose the resulting observable. As expected, the resulting observable using the ObserveOn will send all notifications on the thread where the observer exists.

The SubscribeOn and ObserveOn method encapsulates a lot of threads/tasks related infrastructure code. Thus, the code looks much more clean and to the point.

However, asynchronous programming using multi-threading brings associated problems, such as deadlocks and race conditions. Therefore, the developer must be assured of what is required and what concurrency and synchronization is used to achieve it. The following scenario will illustrate the kind of problems that emerge when multi-threading:
The observable is on a ThreadPool thread, and multiple observers on other threads need to subscribe. As soon as the first observer subscribes, the observable gets evaluated and completes by calling `OnCompleted` even before other subscribers could subscribe. To hold the observable from evaluating unless all the subscribers have subscribed, a special observable called `IConnectableObservable`, which defers the observable from evaluating unless its `Connect` method, is called.

```csharp
var x = new NumberSource().ObserveOn(Scheduler.Dispatcher).Publish();
x.Subscribe(n => Console.WriteLine("A" + n));
x.Subscribe(n => Console.WriteLine("B" + n));
x.Connect();
```

In the above code, in order to get an `IConnectableObservable` the `Publish()` operator is used. When all subscriptions are done, the `Connect` method is called, which will trigger the observable to evaluate.

The main intent of having a connectable observable is to control the Start/Pause/Resume on the observable—Start/Resume by calling `Connect` and Stop by calling the `Dispose` returned by the `Connect` method.

Some other operators to be considered with respect to Concurrency are Start, Defer, Sample, While, FromAsyncPattern and ForkJoin.

**If you want to start with Rx, Please refer Appendix A: Getting Started with Rx.**
REAL-LIFE PROBLEM

Assume that fictitious broker firm LucreWise Inc. has an enterprise system that generates trade orders in milliseconds with hundreds of traders using the system on clients’ behalf. The system puts trade-related information in a standard file format and stores such files in configured file system folders where other applications can pick files for further processing.

Requirement
The firm needs a monitoring application that can generate logs on the activities happening on these system folders (file creates, updates and deletes) for analysis and audit purposes. The first iteration of the application should be able to monitor the file system and show a running log of the changes happening to these trade order files. Also, the user should be able to add new file system locations to be monitored at runtime. These locations also start getting monitored automatically without restarting the monitor application.

Use Cases
1. Collect running logs
   a. User starts the application.
   b. User is able to see the running logs for the file activities happening in folders already configured for the system.
   c. User closes the application.

2. Add a new file system location to monitor
   a. User starts the application.
   b. User adds a valid folder location to monitor.
   c. User is able to view the file activities of the new location in the running logs section.
   d. User closes the application.

Solution Design: Class Diagram
Composition of the Observable

- The FileSystemWatcher class may watch a directory and send create, rename and delete notifications on file and directory changes within that directory.
- For each directory added:
  - FileSystemWatcher is created
  - Rx APIs are used with the following procedure:
    - An observable from each of the notification events (create, update and delete) of the FileSystemWatcher is created.
    - These observables are merged to get a single observable stream that watches all the three events for a single directory.
    - The above observable is merged to the parent observable stream, which is already watching other directories.
    - The parent observable is subscribed.
    - By using ObserveOnDispatcher method, the running log is updated on WPF based UI.

Limitations of Rx

- **Learning Curve.** A decent working knowledge of Delegates, Generics, Lambda Expressions, LINQ and Iterators and Asynchronous Programming Models (APM) are necessary to understand Rx. However, once these concepts are known, Rx is easy to learn.
- **Debugging.** Those new to Rx might face challenges in debugging the Rx section of the code. Debugging does not have sequential flow due to anonymous methods used with subscribe.
- **Alignment with Requirements.** For very small implementation scenarios, Rx might be overkill, as it requires the developer to understand many concepts.

Conclusion

Rx gives the .NET developer a simple yet powerful way to understand and design solutions using the asynchronous programming model. With the rich set of extension operations in lines of LINQ, Rx offers the developer most of the asynchronous behaviors available out of the box, thereby improving productivity and quality in implementing most of the use cases. It can also be extended to deal with more domain-specific scenarios with the set of interfaces it exposes. The online community support around Rx is well developed and growing, which offers valuable “from-the-trenches” resources for developers. While every technology has its “take-and-break” areas, the Rx development story has reached the critical point where it provides a well-ordered conceptual design and APIs for developers.
APPENDIX A: GETTING STARTED WITH RX

Understanding the methods of using Rx is important before including it in the development lifecycle. This section offers a comprehensive, step-by-step approach describing how to get started with Rx.

Step 1: Adding Reference to Rx Interfaces and Assembly

In order to use Rx, the namespace System.Reactive and System.Observable should be referred as illustrated in the figure below.

- Rx interfaces and assemblies

![Add Reference](image)

Step 2: Creating Observable Sequences

Code to create an observable sequence is given below:

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

namespace RxDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            IObservable<string> source = Observable.Return<string>("Hello World");
        }
    }
}
```
Subscribe is an extension method, which is part of the system namespace. The overloads avoid implementation of the IObserver<T> interface every time an observer is needed since any of the three handler methods (OnNext, OnError, OnCompleted) can be specified using delegates on IObservable<T> Object.

```csharp
IDisposable subscription = source.Subscribe(inp =>
    Console.WriteLine("OnNext: {0}", inp),
    ex => Console.WriteLine("OnError: {0}", ex),
    () => Console.WriteLine("OnCompleted");

Console.WriteLine("Press ENTER Any Key to Unsubscribe");
Console.ReadLine();
}
```

The observable pushes the event stream items via the OnNext callback. Once the stream completes pushing all the items, OnCompleted handler is called back. Conversely, if there is an exception during pushing of the items, OnError is called back. The observable stops any further OnNext pushes once OnCompleted/OnError is called.

**Sample Problem**

**1. Creating Observable Sequences**

Code to create an observable sequence follows:

```csharp
using System;
using System.Linq;
namespace RxDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            IObservable<int> source = /* We’ll explore some factory methods here */;
            IDisposable subscription = source.Subscribe(
                x => Console.WriteLine("OnNext: {0}", x),
                ex => Console.WriteLine("OnError: {0}", ex.Message),
```
```csharp
(() => Console.WriteLine("OnCompleted"));

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

Console.ReadLine();
subscription.Dispose();
```

*Note: For the above code, System.CoreEx and System.ReactiveDls should be referenced.*

There is no need to implement an IObservable<T> interface directly. Instead, a whole set of factory methods that can create primitive observable sequences are available. Those factory methods make it convenient to create observable sources and observers. Some of these factory methods are discussed below:

i. **Empty Method.** The empty sequence simply signals completion to its observers by calling OnCompleted.

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

On running this code, output will be:

OnCompleted

ii. **Throw Method.** Empty is the factory method, which creates an observable sequence that immediately triggers completion. By contrast, the Throw method creates an observable sequence that immediately triggers an OnError message to observers:

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

The output will be shown as:

OnError: Oops

iii. **Return Method.** Its role is to represent a single element sequence, such as a single-cell array representing the world of IEnumerable sequences.

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

The output will be:

OnNext: 42
OnCompleted
iv. **RangeMethod.** The range operator is just one operator that generates a sequence.

```csharp
IObservable<int> source = Observable.Range(0, 10);
```

The output will be:
```
OnNext: 5
OnNext: 6
OnNext: 7
OnNext: 8
OnNext: 9
OnNext: 10
OnCompleted
```

v. **GenerateMethod.** This method closely resembles the structure of a for-loop that a developer would use to generate an enumerable sequence using C# iterators.

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

The output will be:
```
OnNext: 0
OnNext: 1
OnNext: 4
OnNext: 9
OnNext: 16
OnCompleted
```

**GenerateWithTime.** This sister function of “Generate” waits a computed amount of time before moving on to the next iteration.

```csharp
IObservable<int> source = Observable.GenerateWithTime<int, int>(0, i => i <= 10,
                                                                 i => i + 2, i => i * i,
                                                                 i => TimeSpan.FromSeconds(3));
```

vi. **Never.** It creates an observable sequence that will never signal any notifications to a subscribed observer.

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

It will never generate any output.

2. **Blocking Operator**

This subscribe method is asynchronous in nature to observe a sequence in synchronous manner. Therefore, we use “Run” operator.
source.Run(x => Console.WriteLine("OnNext: {0}", x),
    ex => Console.WriteLine("OnError: {0}",
        ex.Message),
    () => Console.WriteLine("OnCompleted"));

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

On running this code, output will be:
OnNext: 0
OnNext: 4
OnNext: 16
OnNext: 36
OnNext: 64
OnNext: 100
OnCompleted
Press ENTER to unsubscribe...

Note: Unlike the previous example, "Press ENTER to unsubscribe..." is blocked and appears in the end rather than the beginning.

3. .NET Events
This WinForms sample shows how to create an Rx observable for the GUI events. Every time an event is raised, an OnNext message will be delivered to the observable sequence.

   a. Create a new console project.

   b. Create a new form and run it. It will show a form.

using System.Line;
using System.Windows.Forms;

namespace RxDemo
{
    class Program
    {
        static void Main()
        {
            var frm = new Form();
            Application.Run(frm);
        }
    }
}
Suppose that when the below code is executed for a mouse move event, the form title will show the position of mouse point.

```csharp
var frm = new Form();
frm.MouseMove += (s, e) =>
{
    frm.Text = e.Location.ToString();
};
Application.Run(frm);
```

By using Rx to implement the same functionality, the code will become as shown below.

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

namespace RxDemo
{
    class MouseMoveDemo
    {
        static void Main()
        {
            var frm = new Form();

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

            Application.Run(frm);
        }
    }
}
If an event’s occurrence needs to be filtered and only some instances need to be processed conventionally, the codes will be:

```csharp
static void Main()
{
    var frm = new Form();
    frm.MouseDown += (s, e) =>
    {
        Point pos = e.Location;
        if (pos.X >= pos.Y)
        {
            frm.Text = e.Location.ToString();
        }
    };

    Application.Run(frm);
}
```

Using Rx restriction “where” and projection “select” operators, the same filtering will be done.

```csharp
static void Main()
{
    var frm = new Form();
    var moves = from mousePos
                 in Observable.FromEvent<MouseEventArgs>(frm, "MouseMove")
                 .Do(x =>
                     Console.WriteLine("Mouse Position on Form {0}",
                                     x.EventArgs.Location))
                 .Select(mousePos.EventArgs.Location);                                                

    using (moves.Subscribe(evt =>
    {
        frm.Text = evt.ToString();
    }))
    {
        Application.Run(frm);
    }
}
```

In this code snippet, yellow highlighted code shows one more operator “Do.” This operator is handy in debugging Rx and logging data that is flowing through observable streams.
Asynchronous programming is the practice of designing code in an event-driven fashion. A simple example is the case of Windows forms programming, where a button-click event is assigned an event handler during initialization code. The event handler is called back when the button is actually clicked. All such events in Windows forms create a flow, which is outside the sequential flow of the main program and makes the GUI more responsive. Such event flow is termed as asynchronous.

**Does an asynchronous call always create or use a new thread?**

There is some confusion about whether or not an asynchronous program will always require multi-threading. While many scenarios call for a multi-threaded solution, it is a means to an end and not the end itself. Asynchronous flows are events occurring independently of the main program flow.

For more understanding on asynchronous programming, refer to these resources:
- [http://codingndesign.com/blog/?p=189](http://codingndesign.com/blog/?p=189)
### APPENDIX C: RX EXTENSIBILITY

**IQbservable and Reactive Providers**

New LINQ providers can be implemented using IQueryable interface exposed by LINQ, which provides the same set of operators as for IEnumerable. This interface can be used to introduce new LINQ providers to query a different data source similar to LINQ to SQL and LINQ to Entities.

Following the same pattern, the Rx library exposes the IQbservable interface and provides the same set of operators it provides for IObservable interface. Rx can be extended by writing providers to the IQbservable interface for different event sources. Message Queue Updates, Cache Updates and syndication services such as RSS and live feeds (such as Bloomberg Feeds) are some of the event sources. Rx providers can be written for them.

To write Rx push-based providers for such event sources, the developer can implement IQbservable and IQbservableProvider. These are in lines with the IQueryable and IQueryProvider interfaces in LINQ. Thus, IEnumerable and IQueryable are there for the pull world, IObservable and IQbservable for the push world. It is often concluded that with these four interfaces, developers can realize “LINQ anything and everything”—more so because there are extension methods to reach any one of these interfaces from any other interface (see Figure below).

The video link given below discusses in length the above diagram concepts and the IQbservable interface.


*Image courtesy—IQbservable interface diagram:*

REFERENCES

- Rx Design Guidelines: http://go.microsoft.com/fwlink/?LinkId=205219
- Rx Hands on Lab: http://go.microsoft.com/fwlink/?LinkId=208528
- Rx Examples: http://rxwiki.wikidot.com/101samples
- http://zoltanarvai.com/tag/reactive-extensions/
About Sapient Global Markets
Sapient Global Markets, a division of Sapient® (NASDAQ: SAPE), is a leading provider of services to today's evolving financial and commodity markets. We provide a full range of capabilities to help our clients grow and enhance their businesses, create robust and transparent infrastructure, manage operating costs, and foster innovation throughout their organizations. We offer services across Advisory, Analytics, Technology, and Process, as well as unique methodologies in program management, technology development, and process outsourcing. Sapient Global Markets operates in key financial and commodity centers worldwide, including Boston, Chicago, Houston, New York, Calgary, Toronto, London, Amsterdam, Düsseldorf, Geneva, Munich, Zurich, and Singapore, as well as in large technology development and operations outsourcing centers in Bangalore, Delhi, and Noida, India.

For more information, visit www.sapientglobalmarkets.com.

© 2012 Sapient Corporation.
Trademark Information: Sapient and the Sapient logo are trademarks or registered trademarks of Sapient Corporation or its subsidiaries in the U.S. and other countries. All other trade names are trademarks or registered trademarks of their respective holders.
Global Offices

Amsterdam
Prins Bernhardplein 200
1097 JB Amsterdam
Netherlands
Tel: +31 (0) 20 330 0011

Bangalore
Salarpuria GR Tech Park
6th Floor, "VAYU" Block
#137, Bengaluru 560066
Karnataka
India
Tel: +91 (080) 410 47 000

Boston
131 Dartmouth Street
3rd Floor
Boston, MA 02116
Tel: +1 (617) 621 0200

Calgary
888 3rd Street SW
Suite 1000
Calgary, Alberta T2P 5C5
Canada
Tel: +1 (403) 444 5574

Chicago
30 West Monroe,
12th floor
Chicago, IL 60603
Tel: +1 (312) 458 1800

Delhi
Towers D & E,
DLF Cyber Greens
DLF Phase III
Sector 25-A
Gurgaon 122002
Haryana
India
Tel: +91 (124) 416 7000

Düsseldorf
Speditionstrasse 21
40221 Düsseldorf
Germany
Tel: +49 (0) 211 540 34 0

Geneva
Succursale Genève
c/o Florence Thiébaud, avocate
rue du Cendrier 15
1201 Geneva
Switzerland
Tel: +41 (0) 58 206 06 00

Houston
Heritage Plaza
1111 Bagby Street Suite 1950
Houston, TX 77002
Tel: +1 (713) 493 6880

London
Eden House
8 Spital Square
London, E1 6DU
United Kingdom
Tel: +44 (0) 207 786 4500

Los Angeles
1601 Cloverfield Blvd.
Suite 400 South
Santa Monica, CA 90404
Tel: +1 (310) 264 6900

Munich
Arnulfstrasse 60
80335 München
Germany
Tel: +49 (0) 89 552 987 0

New York
40 Fulton Street
22nd Floor
New York, NY 10038
Tel: +1 (212) 206 1005

Singapore
Air View Building #02-01
2 Peck Seah Street
Singapore 079305
Republic of Singapore
Tel: +65 (3) 104 5308

Toronto
129 Spadina Avenue
Suite 500
Toronto, Ontario M5V 2L3
Canada
Tel: +1 (416) 645 1500

Zürich
Seefeldstrasse 35
8008 Zürich
Switzerland
Tel: +41 (58) 206 06 00