Rx

Bookmarks:

* Concurrent Web Service Calls using Zip: <http://zoltanarvai.com/2011/05/04/concurrent-web-service-calls-from-windows-phone-7-using-rx-4/>
* Reactive Extentions Extensions (Rxx): <http://rxx.codeplex.com/>
* MSDN Doc: <http://msdn.microsoft.com/en-us/library/hh242985(v=VS.103).aspx>
* Jesse Liberty: <http://jesseliberty.com/Topics/reactive/>

Install: use NuGet. Search for Rx- . You will see Rx-Main, Rx-Silverlight/WPF/WinForm etc…

Overview

* IObservable<T> - Observable sequence, events stream, it can be viewed as a collection of elements of type T.
  + Rx Implmention.: Observable. It maintains a list of dependent IObserver<T> implementation as listeners.
* Composing: 1+ IObservable<T> event streams together using LINQ operators like SelectMany, Merge, … e.g. Drag and Drop by observing MouseDown, MouseMove, MouseUp events streams.
* IObservable is a 1st class object compare to .NET event field. Can be passed around through methods param / return value etc…

**Unit**: represents **void**.

**IObserver:** IObserver<T> got 3 publication **events**. OnNext(T value), OnCompleted(), OnError(Exception error). OnNext get called 0+ times.

**ISubject** inherits both IObservable and IObserver. Do both publish & subscribe.

* It is used as a “**Proxy**”. Support **1 data source**(Observable<T>) **to Many subscribers**(IObserver<T>(s)) e.g.

Subject<T> subject = new Subject<T>();

someObservableSource**.Subscribe(subject)**;

subject**.Subscribe(observer1)**;

subject**.Subscribe(observer2)**;

* Can be used to Replace .NET event field. e.g.
  + subject.Subscribe(observer); // Observer or Action<T> OnNext to “Receive” event
  + subject.OnNext(someValue); // “Fire” event
* Example [here](http://msdn.microsoft.com/en-us/library/hh242970(v=VS.103).aspx).
* More Subject Types: [RelaySubject, BehaviorSubject, AsyncSubject](http://msdn.microsoft.com/en-us/library/hh242970(v=VS.103).aspx).

**Subscribe():** add 1 IObserver<T> listener per call

* **Overloads:**
  + **Observer:** Manually create **Observer.Create**(Action<T> OnNext, Action<Exception> OnError, Action OnCompleted)
  + **Action<T> OnNext, Action<Exception> OnError, Action OnCompleted**: So no need to implement **IObserver**<T>
* **return IDisposable:** to **cancel/unsubscribe** the Observer from the IObservable stream by calling .Dispose(); using **-=** ... **Pain**!

**Bridge** – Observable factory methods to convert existing .NET patterns to IObservable<EventPattern<T>>. Every time an event is raised, an OnNext message will be delivered to the observable sequence.

* .NET event: e.g. MouseMove and its MouseMoveEventArgs

IObservable<EventPattern<someEventArgs>> source = Observable**.FromEventPattern**<someEventArgs>(this, “someEvent”);

source.Subscribe(ep => ep.Sender / ep.EventArgs …

* Asynchronous Method Pattern: typically with 2 methods (BeginX & EndX)

Func<T1, T2,…, IObservable<TResult>> call = Observable**.FromAsyncPattern**<T1,T2,…,TResult>(obj.BeginX, obj.EndX);

IObservable<TResult> asynSource = call(T1Value, T2Value, …); // TxValue can be [in/out/ref] i.e. result can be found in one of the TxValue

asyncSource.Subscribe( TResultValue => TResultValue. );

* + .**FromAsyncPattern** returns a **delegate Func**<T1, T2, …, IObservable<TResult>> //note the last Type TResult, like MOST of the Observable Operations, it **transforms returning TResult into returning IObservable<TResult>**
  + You can then call the above **delegate Func** by appending **(T1 p1, T2, p2, … )** which will return IObservable<TResult>
* **FromEvent**: use this when event is **NOT** the using void EventHandler<TEventArgs>(object **sender**, TEventArgs **e**) pattern
  + .FromEvent implicit create an event handler and pass it as ev below

            IObservable<xyzEventArgs> os = Observable.FromEvent**<EventHandler<xyzEventArgs>, xyzEventArgs>**(

ev => xyzEvent **+= ev**,  ev => xyzEvent -**= ev**);

* **ToObservable**
  + IObservable<T> os = someList<T>.**ToObservable**(); // **generic**
  + IObservable<T> os = someNonGenericArray.**Cast**<T>().**ToObservable**(); // **non-generic** require .Cast first
* **ToEnumerable**

**Hot & Cold Observables** (<http://leecampbell.blogspot.com/2010/08/rx-part-7-hot-and-cold-observables.html> \*difficult stuff)

* <http://blogs.microsoft.co.il/blogs/bnaya/archive/2010/03/13/rx-for-beginners-part-9-hot-vs-cold-observable.aspx>
* **Cold:**
  + e.g. Observable.Interval, on demand stream, Rx Async call, subscribe to a queue.
  + Sequence created **ONLY upon subscription**.
* **Hot:**
  + e.g. MouseMove, timer event
  + LIVE somewhere! Even before Subscribe.
  + Sample using Timer.

          System.Timers.Timer timer = new System.Timers.Timer(1000);

            timer.Elapsed += (s, e) => Console.WriteLine("------->: " + e.SignalTime); // **conventional, will see timer starts b4 Subscribe.**

            timer.Start(); // timer starts spitting out to the above event handler after this line.

            Thread.Sleep(5000); // 1st 5 secs will not be subscribed to IObservable.

            var tick = Observable.**FromEventPattern**(timer, "Elapsed");

            tick.Subscribe(n => Console.WriteLine(string.Format("onNext A {0} ", (n.EventArgs as ElapsedEventArgs).SignalTime)));

* + - Experiment: add a 2nd Subscribe and add .Publish/.Connect to see multicast.

**Operators**

* **Create**
  + IObservable<TResult> **Generate**<TState, TResult>(TState initialState,Func<TState,bool> condition, Func<TState,TState> iterate, Func<TState, TResult> resultSelector)
    - Imagine this as a method to create a **“For loop”** which return a TResult based on the iterated TState.
  + IObservable<**TValue**> **Defer**(**Func**<**IObservable**<**TValue**>>)**:** defer the creation of the sequence until Subscribe() is called, and the elements inside the sequence will based on the lastest value at the time that Subscribe() is called.
  + **Range**(int **start**, int **count**): generate an observable sequence of Int.
* **Primitives**
  + **Empty<T>**(): returns an empty IObservable<T>, after Subscribe() **onNext** will **NEVER** called but **onComplete** **WILL**.
  + **Never<T>**(): returns an empty and never teminated IObservable<T>, after Subscribe() **NONE** of **onNext, onComplete, onError** will get called.
  + **Return<T>(T value)**: returns an IObservable<T> sequence with a **SINGLE** element with specified value and after Subscribe() **onNext** get called **ONCE**, followed by **onComplete**
  + **Throw<T>(Exception e):** returns an empty IObservable<T> which called **onError** with the specified **Exception object**.
* **Combine Sequences**: both source1 and source2 runs concurrent.
  + **Concat**: source1.Concat(source1): resultSource accept source1’s push first and only after source1 finished pushing then start accept source2’s push. Source1 & Source 2 must be same type.
    - source1**.Catch**(source2): if source1 have Error, ONLY will source2 starts pushing. It behaves like Catch after source1 got Error
    - source1.**OnErrorResumeNext**(source2): It behaves like .Concat when source1 **Completed** **OR** **Error**. If source1 have Error OR Complete, source2 starts pushing.
  + **Merge**: source1.Merge(source2): resultSource accepts both source1 & source2’s pushes at the same time. If either source throws exception, resultant sequence will be terminated immediately. Source1 & Source 2 must be same type. View as 2 lanes merge to 1 lane at highway.
  + **Zip**: source1[.**Zip**](http://msdn.microsoft.com/en-us/library/hh211957(v=VS.103).aspx)(source2, (l,r)=> use l & r to do something ): A binary operation, the 1st element has to wait for the 2nd element before they will be put together into the new sequence. An output element will be created ONLY when BOTH 1st and 2nd are present. Source1 & Source 2 don’t have to be same type. View it as a 2-seat ski lift on a budget slope, lift won’t move until it is full.
    - A **slanted** combine operation. **NEITHER a Merge NOR a Concat**!
    - Static IObservable<TResult> Zip(this IObservable<TFirst> first, IObservable<TSecond> second, Func<TFirst,TSecond,TResult> resultSelector)
  + **Repeat**(count): When source sequence completed, it will subscribe to the source sequence again until reaches “count” or indefinitely if w/o argument. If Error before Complete, it will stop immediately.
    - **Retry**(count): View as doing Repeat BUT when Error happens, it will subscribe to the source sequence again “count”. Repeat<->Retry is similar to Concat<->Catch.
  + **StartWith**(params TSource[] values): **prepends** a sequence of elements to a observable sequence.
  + Amb
* **Functional**
  + **IConnectableObservable**<TSource> **Publish**():
    - Internally created a proxy to multicast **WHEN Connect()** is called. i.e. **NOT WHEN Subscribe** is called.
    - Turn a non-shared observable sequence **to a shared “connectable” Observable**.

var sharedConnectableOS = **nonSharedOS.Publish();**

sharedConnectableOS.**Subscribe**(action1); sharedConnectableOS.**Subscribe**(action2); // **Subscribe 1+ times**

sharedConnectableOS.**Connect**(); // NO Marbles until Connect() is called

* + **IConnectableObservable**<TSource> **Replay**(): returns a “Connectable” Obsrervable sequence to replay all notifications.
  + **Let**
* **Exception Handling**
  + **Finally(Action):** specify **Action** that executes whether exception thrown or not.
  + **Catch:** continue to the specified observable sequence **ONLY WHEN** current observable sequence got **exception**.
  + **OnErrorResumeNext:** continue to the specified observable sequence **WHETHER OR NOT** terminate **normally** **OR** /w **exception**.
* **Projection**: typical LINQ projection stuff
  + source1.**SelectMany**(source2):
    - for each projected element pushed by source1, ALL source2 elements of current source1 element are projected and .Concat to the previous source2 projection (i.e. if the 1st source2 elements are 1, 2, 3 and the 2nd source2 elements are 4,5,6 then after projected the resultant sequence becomes elements: 1,2,3,4,5,6).
    - You can view this as a nested outer/inner forloop. This is also equivalent to from y in z from x in y select x => x…; Without the help of SelectMany, you will need to use 2 forloop to get to x. [LINQ Pseudo-code](http://www.hookedonlinq.com/SelectManyOperator.ashx). [LINQ Example](http://msdn.microsoft.com/en-us/library/bb534336.aspx).
    - Implemented by composing the Select & Merge operators.
* **Filtering**: typical LINQ filtering stuff
  + Take, TakeUntil/TakeWhile, Skip, SkipUntil/SkipWhile
* **Time-based**
  + IObservable<**IList<TSource>**> **Buffer**(int count|skip/TimeSpan timeSpan|timeShift):
    - **Returns ALL** elements found inside the buffer.
    - Buffer is released when it is full “count” OR amount of time has elapsed “timeSpan” and interval between creation of consecutive buffers. The buffer is flushed every time after used
    - **IList<TSource>** is the source values held by the Buffer.

var seq = Observable.Interval(TimeSpan.FromSeconds(1));

var bufSeq = seq.Buffer(5);

bufSeq.Subscribe(values => Console.WriteLine(values.Sum()));

// The printout will be 10,35,60,… in which 10=0+1+2+3+4, 35=5+6+7+8+9, 60=10+11+12+13+14 etc…

* + IObservable<***long***> **Interval**(Timespan period): Observable.Interval(TimeSpan.FromSeconds(1)); // p.s. wrapper of .Timer()
  + IObservable<***TSource***> **Sample**(TimeSpan interval): Take the **last** **element** of the observable sequence at **each interval**.
  + IObservable<***Timestamped*<*TSource*>**> **.Timestamp():** wrap each element of the observable sequence with a **struct Timestamped** which has 2 properties: T **Value;** & DateTimeOffset **Timestamp;**
  + IObservable<***TimeInterval*<*TSource*>**>**TimeInterval()**: similar to TimeStamp() except **struct** is **TimeInterval** and property Timestamp ***replaced by*** **TimeSpan Interval;** which records the time interval between consecutive elements
  + IObservable<***TSource***> **Throttle**(TimeSpan dueTime): if time differences between adjacent elements < dueTime, ignores it.
  + IObservable<**TSource**> **Timeout**(TimeSpan dueTime): return the same sequence OR if pass dueTime throws **TimeoutException**
  + IObservable<**TSource**> **Delay**(TimeSpan dueTime): delay the output of the sequence elements by dueTime
  + IObservable<**long**>**Timer**(TimeSpan dueTime, TimeSpan period):
* **Aggregate**
  + **Aggregate**, **Count**, **Min**, **Max**, **Sum**
* **Misc: Window, Scan**

**Scheduler**

* 1. **Queue** Tasks 2. **Where** tasks got executed (ThreadPool, Current Thread, …) 3. **When** tasks got executed **relative to current Scheduler** (Scheduler.Now)
* **static** Scheduler.**CurrentThread/Immediate/NewThread/TaskPool/ThreadPool**/**Now** – These are **static** properties **NOT Enum,** each return an IScheduler impl. except the .Now which return DateTimeOffset.
  + TaskPool/ThreadPool for short running task. ThreadPool will be used on Timer based operators
  + NewThread for long running
  + CurrentThread/Immediate both run on current thread.
    - CurrentThread is for large to infinite messages
    - Immediate for small finite messages.
  + \*\*\* **Dispatcher** runs on UI thread: // only available in Silverlight when Rx-Silverlight is installed
    - Use IObservable<T>**.ObserveOnDispatcher()/SubscribeOnDispatcher()** …
    - OR explicitly use DispatcherScheduler.Instance
* **ObserverOn**[Dispatcher](someScheduler): Observer got notified using the specified Scheduler
* **SubscribeOn**[Dispatcher](someScheduler): The Observable Source runs using the specified Scheduler. **Barely** use since each Observable **operator** which returns IObservable<T> have an **overload** that takes **IScheduler**.
* Performance tips: Put ObserveOn at the end of the query will cause the least impact on performance

**Debugging**

* **.Do**(x => Console.WriteLine(x.ToString()) to **spy on each item** of the observable sequence at **ANY level** without affecting ANY sequences.
* **.Timestamp**().Subscribe(x => x.**Value**/x.**Timestamp**) to append the time on each item of the observable sequence.
* Set **breakpoints** inside the lambda expression **after the =>** (goes to) .

**Create Custom Observable**:

* Partial Impl. (Preferred way)
  + IObservable<T> Observable.**Create**(Func<IObserver<TSource>, **IDisposable**> **SubscribeImpl**);
    - .Create() implement and provides built-in Observer,
    - BUT, you must implement IObservable<T>.[Subscribe](http://msdn.microsoft.com/en-us/library/dd782981.aspx) by providing a Func delegate in the Observable.Create() method. [example here](http://msdn.microsoft.com/en-us/library/hh229114(v=VS.103).aspx)
    - IObservable<TSource> Observable**.Create**<TSource>(o => someIObservable<TSource>.Subscribe(o));
      * **IObserver<TSource> o** is implicity provided by the .Create method’s internally.
      * **i.e. you can provide custom sequence with custom subscribe without impl. your own Observer**
* From Subject: ??? new Subject<T>.AsObservable; [http://rxwiki.wikidot.com/101samples#toc48](http://rxwiki.wikidot.com/101samples%23toc48)
* Full Impl. Example: <http://msdn.microsoft.com/en-us/library/dd990377.aspx>

**Sync to Async**

* **Start:** invoke **Async** operation **IMMEDIATELY** **w/o** requiring calling **Subscribe.**
  + IObservable<**Unit**> Observable.**Start**(**Action** action);
  + IObservable<**TSource**> Observable.**Start<TSource>**(**Func**<**TSource**> function);
* Wrap **Synchronous** to **Async** and then make it **Observable**!
  + **Start:** invoke **Async** operation **IMMEDIATELY** **w/o** requiring calling **Subscribe.**
    - IObservable<**Unit**> Observable.**Start**(**Action** action);
    - IObservable<**TSource**> Observable.**Start<TSource>**(**Func**<**TSource**> function);
  + **ToAsync**: to **wrap a Sync** method like this **TResult** someSyncMethodName(**T1 p1, T2 p2, …** ) **to Async** and **return IObservable<TResult>**
    - Observable.**ToAsync**<T1, T2, …, TResult>(someSyncMethodName) returns a **Func**<T1, T2, …, **IObservable<TResult>**>
      * like MOST of the Observable Operations, it **TRANSFORM** returning **TResult** **INTO** **IObservable<TResult>**
    - **Call the Func** by appending **(T1 p1, T2, p2, … )** which will **return** **IObservable<TResult>**
    - **Start** only accept **Action & Func<TSource> vs ToAsync has MUCH MORE choices.**

**Application**

* Async WCF proxy calls:
  + **FromAsyncPattern** using auto generated proxy **Interface**.

Service1 iproxy = new Service1Client() as Service1;

var dowork = Observable.FromAsyncPattern<int, int>(iproxy.BeginDoWork1, iproxy.EndDoWork1);

var os = dowork(111);

var d = os.ObserveOnDispatcher().Subscribe(x => textBlock1.Text = x.ToString());

* + **FromEventPattern** using the **auto generated proxy**. Don’t like this one bcos the **hard-coded string**

Service1Client proxy = new Service1Client();

var dowork = Observable.FromEventPattern<DoWork1CompletedEventArgs>(proxy, "DoWork1Completed")

.ObserveOnDispatcher().Subscribe(x => textBlock1.Text = x.EventArgs.Result.ToString());

proxy.DoWork1Async(111);

**Chaining WCF Async calls**. Equilvalent to local calls: int textBlock1.Text = Foo2(Foo1(100));

* You learn: 1. Use Rx to make WCF Async call using **Observable.FromAsyncPattern**, 2. **Chaining WCF calls** using BOTH **2A**. Extension Methods & **2B**. LINQ Query Expression. e.g.

Service1 iproxy = new Service1Client() **as** **Service1**;

//**2A**.  **Extension Method**: Project inner element only

Observable.**FromAsyncPattern**<int, int>(iproxy.BeginFoo1, iproxy.EndFoo1)(100)

    .**SelectMany**(x => Observable.**FromAsyncPattern**<int, int>(iproxy.BeginFoo2, iproxy.EndFoo2)(x))

    .**ObserveOnDispatcher**().**Subscribe**(x => textBlock1.Text = x.ToString());

// **2A**. **Extension Method**: Project **BOTH outer & inner** elements

int xTemp = 0;

Observable.FromAsyncPattern<int, int>(iproxy.BeginFoo1, iproxy.EndFoo1)(100)

    .SelectMany(x => { xTemp = x; return Observable.FromAsyncPattern<int, int>(iproxy.BeginFoo2, iproxy.EndFoo2)(x); })

**.Select(x => string.Format("{0} : {1}", xTemp, x))**

    .ObserveOnDispatcher().Subscribe(x => textBlock1.Text = x.ToString());

// **2B**. **LINQ Query**: Project inner element only

(**from** i1 in Observable.**FromAsyncPattern**<int, int>(iproxy.BeginFoo1, iproxy.EndFoo1)(100)

**from** i2 in Observable.**FromAsyncPattern**<int, int>(iproxy.BeginFoo2, iproxy.EndFoo2)(i1)

**select i2**)

    .**ObserveOnDispatcher**().**Subscribe**(x => textBlock1.Text = x.ToString());

// **2B**. **LINQ Query**: Project **BOTH outer & inner** elements

(from i1 in Observable.FromAsyncPattern<int, int>(iproxy.BeginFoo1, iproxy.EndFoo1)(100)

    from i2 in Observable.FromAsyncPattern<int, int>(iproxy.BeginFoo2, iproxy.EndFoo2)(i1)

**select string.Format("{0} : {1}", i1, i2))**

    .ObserveOnDispatcher().Subscribe(x => textBlock1.Text = x.ToString());

**Concurrent WCF Async calls:**

* **Wait ALL**: until ALL message returns
  + Use .**Zip** which allow us to have a <TResult> which is **different** from the <TSource>

Observable.**FromAsyncPattern**<int, int>(iproxy.BeginFoo1, iproxy.EndFoo1)(100)

    .**Zip**(Observable.**FromAsyncPattern**<int, int>(iproxy.BeginFoo2, iproxy.EndFoo2)(200), **(l, r) => new { ResultFoo1 = l,ResultFoo2 = r}**)

    .**ObserveOnDispatcher**().**Subscribe**(x => textBlock1.Text = string.Format("{0} : {1}", **x.ResultFoo1, x.ResultFoo2**));

* + <http://zoltanarvai.com/tag/reactive-extensions/>
* **Wait ANY**: Receive and continue to the next message
  + Use .Merge which ALL have to be the same <TResult>

Observable.FromAsyncPattern<int, int>(iproxy.BeginFoo1, iproxy.EndFoo1)(100)

    .Merge(Observable.FromAsyncPattern<int, int>(iproxy.BeginFoo2, iproxy.EndFoo2)(200))

    .ObserveOnDispatcher().Subscribe(x => textBlock1.Text = x.ToString());

**Async WebRequest**

var httpWR = (HttpWebRequest)WebRequest.Create("http://localhost:3678/test.htm");

Observable.**FromAsyncPattern**<WebResponse>(httpWR.BeginGetResponse, httpWR.EndGetResponse)()

.ObserveOnDispatcher().Subscribe(resp => textBlock1.Text = new StreamReader(resp.GetResponseStream()).ReadToEnd());

**Async WebClient**

var client = new WebClient();

Observable.**FromEventPattern**<DownloadStringCompletedEventArgs>(client, "DownloadStringCompleted")

    .ObserveOnDispatcher().Subscribe(s => textBlock1.Text = s.EventArgs.Result);

client.DownloadStringAsync(new Uri("http://localhost:3678/test.htm"));