- Welcome page
- What is a Wiki Site?
- How to edit pages?
- How to join this site?
- Site members
- Recent changes
- List all pages
- Page Tags
- Site Manager

Page tags

reactive-extensions

Add a new page

new page

edit this panel

101 Rx Samples - a work in progress

You!
Yes, you, the one who is still scratching their head trying to figure out this Rx thing.
As you learn and explore, please feel free add your own samples here (or tweak existing ones!)
Anyone can (and should!) edit this page. (edit button is at the bottom right of each page)

(and sorry for the years of spam pages there - they should be gone now. Thanks for marking them. -Rob)

Fold

Table of Contents

Asynchronous Background Operations

Start - Run Code Asynchronously

```
public static async void StartBackgroundWork()
    Console.WriteLine("Shows use of Start to s
    var o = Observable.Start(() =>
    {
        //This starts on a background thread.
        Console.WriteLine("From background thr
        Console.WriteLine("Calculating...");
        Thread.Sleep(1000);
        Console.WriteLine("Background work com
    });
    await o.FirstAsync();   // subscribe and w
    Console.WriteLine("Main thread completed."
}
```

Run a method asynchronously on demand

Execute a long-running method asynchronously. The method does not start running until there is a subscriber. The method is started every time the observable is created and subscribed, so there could be more than one running at once.

```
// Synchronous operation
public DataType DoLongRunningOperation(string
{
    ...
}

public IObservable<DataType> LongRunningOperat
{
    return Observable.Create<DataType>(
        o => Observable.ToAsync<string,DataTyp
    );
}
```

CombineLatest - Parallel Execution

Merges the specified observable sequences into one observable sequence by emitting a list with the latest source elements whenever any of the observable sequences produces an element.

```
var o = Observable.CombineLatest(
    Observable.Start(() => { Console.WriteLine
    Observable.Start(() => { Console.WriteLine
    Observable.Start(() => { Console.WriteLine
).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

Note Was ForkJoin which is no longer supported. CombineLatest gives the same result.)

Create With Disposable & Scheduler - Canceling an asynchronous operation

This sample starts a background operation that generates a sequence of integers until it is canceled by the main thread. To start the background operation new the Scheduler class is used and a CancellationTokenSource is indirectly created by a Observable.Create.
Please check out the MSDN documentation on System.Threading.CancellationTokenSource to learn more about cancellation source.

```
IObservable<int> ob =
    Observable.Create<int>(o =>
        {
            var cancel = new CancellationDispo
            NewThreadScheduler.Default.Schedul
                {
                    int i = 0;
                    for (; ; )
                    {
                        Thread.Sleep(200);  //
                        if (!cancel.Token.IsCa
                            o.OnNext(i++);
                        else
                        {
                            Console.WriteLine(
                            o.OnCompleted();
                            return;
                        }
                    }
                }
            );

            return cancel;
        }
    );

IDisposable subscription = ob.Subscribe(i => C
Console.WriteLine("Press any key to cancel");
Console.ReadKey();
subscription.Dispose();
Console.WriteLine("Press any key to quit");
Console.ReadKey();  // give background thread
```

Observation Operators

Observing an Event - Simple

```
class ObserveEvent_Simple
{
    public static event EventHandler SimpleEve
    static void Main()
    {
        // To consume SimpleEvent as an IObser
        IObservable<EventPattern<EventArgs>> e
        ev => SimpleEvent += ev,
        ev => SimpleEvent -= ev);
    }
}
```

Alternately, you can use EventArgs:

```
public static event EventHandler<EventArgs> Si

private static void Main(string[] args) {
```

```
    IObservable<EventPattern<EventArgs>> event
        (ev => SimpleEvent += ev,
         ev => SimpleEvent -= ev);
 }
```

Observing an Event - Simple (expanded)

```
class ObserveEvent_Simple
{
    public static event EventHandler SimpleEve

    private static void Main()
    {
        Console.WriteLine("Setup observable");
        // To consume SimpleEvent as an IObser
        IObservable<EventPattern<EventArgs>> e
                ev => SimpleEvent += ev,
                ev => SimpleEvent -= ev);

        // SimpleEvent is null until we subscr
        Console.WriteLine(SimpleEvent == null

        Console.WriteLine("Subscribe");
        //Create two event subscribers
        var s = eventAsObservable.Subscribe(ar
        var t = eventAsObservable.Subscribe(ar

        // After subscribing the event handler
        Console.WriteLine(SimpleEvent == null

        Console.WriteLine("Raise event");
        if (null != SimpleEvent)
        {
            SimpleEvent(null, EventArgs.Empty)
        }

        // Allow some time before unsubscribin
        Thread.Sleep(100);

        Console.WriteLine("Unsubscribe");
        s.Dispose();
        t.Dispose();

        // After unsubscribing the event handl
        Console.WriteLine(SimpleEvent == null

        Console.ReadKey();
    }
}
```

Observing MouseMove in Silverlight

```
var mouseMove = Observable.FromEventPattern<Mo
mouseMove.ObserveOnDispatcher()
```

```
            .Subscribe(args => Debug.WriteLine(ar
```

Note that a reference to System.Reactive.Windows.Threading is required for ObserveOnDispatcher which is in Nuget as Reactive Extensions - Silverlight Helpers.

Observing an Event - Generic

```
class ObserveEvent_Generic
{
    public class SomeEventArgs : EventArgs { }
    public static event EventHandler<SomeEvent

    static void Main()
    {
        // To consume GenericEvent as an IObse
        IObservable<EventPattern<SomeEventArgs
            ev => GenericEvent += ev,
            ev => GenericEvent -= ev );
    }
}
```

Observing an Event - Non-Generic

```
class ObserveEvent_NonGeneric
{
    public class SomeEventArgs : EventArgs { }
    public delegate void SomeNonGenericEventHa
    public static event SomeNonGenericEventHan

    static void Main()
    {
        // To consume NonGenericEvent as an IO
        // In this case, it is SomeEventArgs.
        IObservable<IEvent<SomeEventArgs>> eve
            (EventHandler<SomeEventArgs> ev) =
            ev => NonGenericEvent += ev,
            ev => NonGenericEvent -= ev);
    }
}
```

Observing an Asynchronous Operation

```
class Observe_IAsync
{
    static void Main()
    {
        // We will use Stream's BeginRead and
        Stream inputStream = Console.OpenStand
```

```
        // To convert an asynchronous operatio
        // For the generic arguments, specify
        // If the End* method returns a value,
        var read = Observable.FromAsyncPattern

        // Now, you can get an IObservable ins
        byte[] someBytes = new byte[10];
        IObservable<int> observable = read(som
    }
}
```

Be aware that while the code above formally provides an observable, this is not enough for most intended uses. For more information, see Creating an observable sequence and c# - What is the proper way to create an Observable which reads a stream to the end - Stack Overflow.

Observing a Generic IEnumerable

```
class Observe_GenericIEnumerable
{
    static void Main()
    {
        IEnumerable<int> someInts = new List<i

        // To convert a generic IEnumerable in
        IObservable<int> observable = someInts
    }
}
```

Observing a Non-Generic IEnumerable - Single Type

```
class Observe_NonGenericIEnumerableSingleType
{
    static void Main()
    {
        IEnumerable someInts = new object[] {

        // To convert a non-generic IEnumerabl
        // first use Cast<> to change the non-
        // then use ToObservable.
        IObservable<int> observable = someInts
    }
}
```

Observing a Non-Generic IEnumerable - Multiple Types

Observing the Passing of Time

```
class Observe_Time
{
    static void Main()
    {
        // To observe time passing, use the Ob
        // It will notify you on a time interv

        // 0 after 1s, 1 after 2s, 2 after 3s,
        IObservable<long> oneNumberPerSecond =
        IObservable<long> alsoOneNumberPerSeco
    }
}
```

Restriction Operators

Where - Simple

```
class Where_Simple
{
    static void Main()
    {
        var oneNumberPerSecond = Observable.In

        var lowNums = from n in oneNumberPerSe
                      where n < 5
                      select n;

        Console.WriteLine("Numbers < 5:");

        lowNums.Subscribe(lowNum =>
        {
            Console.WriteLine(lowNum);
        });

        Console.ReadKey();
    }
}
```

Result
Numbers < 5:
0 *(after 1s)*
1 *(after 2s)*
2 *(after 3s)*
3 *(after 4s)*
4 *(after 5s)*

Where - Drilldown

```
class Where_DrillDown
{
    class Customer
    {
```

```
        public Customer() { Orders = new Obser
        public string CustomerName { get; set;
        public string Region { get; set; }
        public ObservableCollection<Order> Ord
    }

    class Order
    {
        public int OrderId { get; set; }
        public DateTimeOffset OrderDate { get;
    }

    static void Main()
    {
        var customers = new ObservableCollecti

        var customerChanges = Observable.FromE
            (EventHandler<NotifyCollectionChan
               => new NotifyCollectionChangedE
            ev => customers.CollectionChanged
            ev => customers.CollectionChanged

        var watchForNewCustomersFromWashington
            from c in customerChanges
            where c.EventArgs.Action == Notify
            from cus in c.EventArgs.NewItems.C
            where cus.Region == "WA"
            select cus;

        Console.WriteLine("New customers from

        watchForNewCustomersFromWashington.Sub
        {
            Console.WriteLine("Customer {0}:",

            foreach (var order in cus.Orders)
            {
                Console.WriteLine("Order {0}:
            }
        });

        customers.Add(new Customer
        {
            CustomerName = "Lazy K Kountry Sto
            Region = "WA",
            Orders = { new Order { OrderDate =
        });

        Thread.Sleep(1000);
        customers.Add(new Customer
        {
            CustomerName = "Joe's Food Shop",
            Region = "NY",
            Orders = { new Order { OrderDate =
        });

        Thread.Sleep(1000);
        customers.Add(new Customer
        {
            CustomerName = "Trail's Head Gourm
            Region = "WA",
```

```
            Orders = { new Order { OrderDate =
        });

        Console.ReadKey();
    }
}
```

Result
New customers from Washington and their orders:
Customer Lazy K Kountry Store: *(after 0s)*
Order 1: 11/20/2009 11:52:02 AM -06:00
Customer Trail's Head Gourmet Provisioners: *(after 2s)*
Order 3: 11/20/2009 11:52:04 AM -06:00

Projection Operators

Select - Simple

```
class Select_Simple
{
    static void Main()
    {
        var oneNumberPerSecond = Observable.In

        var numbersTimesTwo = from n in oneNum
                              select n * 2;

        Console.WriteLine("Numbers * 2:");

        numbersTimesTwo.Subscribe(num =>
        {
            Console.WriteLine(num);
        });

        Console.ReadKey();
    }
}
```

Result
Numbers * 2:
0 *(after 1s)*
2 *(after 2s)*
4 *(after 3s)*
6 *(after 4s)*
8 *(after 5s)*

Select - Transformation

```
class Select_Transform
{
    static void Main()
    {
        var oneNumberPerSecond = Observable.In
```

```
        var stringsFromNumbers = from n in one
                                 select new st

        Console.WriteLine("Strings from number

        stringsFromNumbers.Subscribe(num =>
        {
            Console.WriteLine(num);
        });

        Console.ReadKey();
    }
}
```

Result
Strings from numbers:
(after 0s)
* *(after 1s)*
** *(after 2s)*
*** *(after 3s)*
**** *(after 4s)*
***** *(after 5s)*
****** *(after 6s)*

Select - Indexed

```
class Where_Indexed
{
    class TimeIndex
    {
        public TimeIndex(int index, DateTimeOf
        {
            Index = index;
            Time = time;
        }
        public int Index { get; set; }
        public DateTimeOffset Time { get; set;
    }

    static void Main()
    {
        var clock = Observable.Interval(TimeSp
            .Select((t, index) => new TimeInde

        clock.Subscribe(timeIndex =>
        {
            Console.WriteLine(
                "Ding dong.  The time is now {
                timeIndex.Time,
                timeIndex.Index);
        });

        Console.ReadKey();
    }
}
```

Result
Ding dong. The time is now 1:55:00 PM. This is event number 0. *(after 0s)*
Ding dong. The time is now 1:55:01 PM. This is event number 1. *(after 1s)*
Ding dong. The time is now 1:55:02 PM. This is event number 2. *(after 2s)*
Ding dong. The time is now 1:55:03 PM. This is event number 3. *(after 3s)*
Ding dong. The time is now 1:55:04 PM. This is event number 4. *(after 4s)*
Ding dong. The time is now 1:55:05 PM. This is event number 5. *(after 5s)*

Grouping

Group By - Simple

This example counts how many time you press each key as you furiously hit the keyboard. :)

```
class GroupBy_Simple
{
    static IEnumerable<ConsoleKeyInfo> KeyPres
    {
        for (; ; )
        {
            var currentKey = Console.ReadKey(t

            if (currentKey.Key == ConsoleKey.E
                yield break;
            else
                yield return currentKey;
        }
    }
    static void Main()
    {
        var timeToStop = new ManualResetEvent(
        var keyPresses = KeyPresses().ToObserv

        var groupedKeyPresses =
            from k in keyPresses
            group k by k.Key into keyPressGrou
            select keyPressGroup;

        Console.WriteLine("Press Enter to stop

        groupedKeyPresses.Subscribe(keyPressGr
        {
            int numberPresses = 0;

            keyPressGroup.Subscribe(keyPress =
            {
                Console.WriteLine(
                    "You pressed the {0} key {
                    keyPress.Key,
                    ++numberPresses);
```

```
            },
            () => timeToStop.Set());
        });

        timeToStop.WaitOne();
    }
}
```

Result
Depends on what you press! But something like:
Press Enter to stop. Now bang that keyboard!
You pressed the A key 1 time(s)!
You pressed the A key 2 time(s)!
You pressed the B key 1 time(s)!
You pressed the B key 2 time(s)!
You pressed the C key 1 time(s)!
You pressed the C key 2 time(s)!
You pressed the C key 3 time(s)!
You pressed the A key 3 time(s)!
You pressed the B key 3 time(s)!
You pressed the A key 4 time(s)!
You pressed the A key 5 time(s)!
You pressed the C key 4 time(s)!

Time-Related Operators

Buffer - Simple

Buffer has a strange name, but a simple concept. Imagine an email program that checks for new mail every 5 minutes. While you can receive mail at any instant in time, you only get a batch of emails at every five minute mark.
Let's use Buffer to simulate this.

```
class Buffer_Simple
{
    static IEnumerable<string> EndlessBarrageO
    {
        var random = new Random();
        var emails = new List<String> { "Here
        for (; ; )
        {
            // Return some random emails at ra
            yield return emails[random.Next(em
            Thread.Sleep(random.Next(1000));
        }
    }
    static void Main()
    {
        var myInbox = EndlessBarrageOfEmail().

        // Instead of making you wait 5 minute
        var getMailEveryThreeSeconds = myInbox

        getMailEveryThreeSeconds.Subscribe(ema
```

```
        {
            Console.WriteLine("You've got {0}
            foreach (var email in emails)
            {
                Console.WriteLine("> {0}", ema
            }
            Console.WriteLine();
        });

        Console.ReadKey();
    }
}
```

Result
You've got 5 new messages! Here they are! *(after 3s)*
> Here is an email!
> Another email!
> Here is an email!
> Another email!
> Here is an email!

You've got 6 new messages! Here they are! *(after 6s)*
> Another email!
> Another email!
> Here is an email!
> Here is an email!
> Another email!
> Another email!

Delay - Simple

```
class Delay_Simple
{
    static void Main()
    {
        var oneNumberEveryFiveSeconds = Observ

        // Instant echo
        oneNumberEveryFiveSeconds.Subscribe(nu
        {
            Console.WriteLine(num);
        });

        // One second delay
        oneNumberEveryFiveSeconds.Delay(TimeSp
        {
            Console.WriteLine("...{0}...", num
        });

        // Two second delay
        oneNumberEveryFiveSeconds.Delay(TimeSp
        {
            Console.WriteLine("......{0}......
        });

        Console.ReadKey();
    }
```

```
}
```

Result
0 *(after 5s)*
...0... *(after 6s)*
.....0..... *(after 7s)*
1 *(after 10s)*
...1... *(after 11s)*
.....1..... *(after 12s)*

Interval - Simple

```
internal class Interval_Simple
{
    private static void Main()
    {
        IObservable<long> observable = Observa

        using (observable.Subscribe(Console.Wr
        {
            Console.WriteLine("Press any key t
            Console.ReadKey();
        }

        Console.WriteLine("Press any key to ex
        Console.ReadKey();
    }
}
```

Result
0 *(after 1s)*
1 *(after 2s)*
2 *(after 3s)*
3 *(after 4s)*
...

Sample - Simple

```
internal class Sample_Simple
{
    private static void Main()
    {
        // Generate sequence of numbers, (an i
        IObservable<long> observable = Observa

        // Sample the sequence every second
        using (observable.Sample(TimeSpan.From
            x => Console.WriteLine("{0}: {1}",
        {
            Console.WriteLine("Press any key t
            Console.ReadKey();
        }

        Console.WriteLine("Press any key to ex
```

```
            Console.ReadKey();
        }
    }
```

Result
15: 24/11/2009 15:40:45 *(after 1s)*
31: 24/11/2009 15:40:46 *(after 2s)*
47: 24/11/2009 15:40:47 *(after 3s)*
64: 24/11/2009 15:40:48 *(after 4s)*
…

Throttle - Simple

Throttle stops the flow of events until no more events are produced for a specified period of time. For example, if you throttle a TextChanged event of a textbox to .5 seconds, no events will be passed until the user has stopped typing for .5 seconds. This is useful in search boxes where you do not want to start a new search after every keystroke, but want to wait until the user pauses.

```
SearchTextChangedObservable = Observable.FromE
_currentSubscription = SearchTextChangedObserv
```

Here is another example:

```
internal class Throttle_Simple
{
    // Generates events with interval that alt
    static IEnumerable<int> GenerateAlternatin
    {
        int i = 0;

        while(true)
        {
            if(i > 1000)
            {
                yield break;
            }
            yield return i;
            Thread.Sleep( i++ % 10 < 5 ? 500 :
        }
    }

    private static void Main()
    {
        var observable = GenerateAlternatingFa
        var throttled = observable.Throttle(Ti

        using (throttled.Subscribe(x => Consol
        {
            Console.WriteLine("Press any key t
            Console.ReadKey();
        }
```

```
        Console.WriteLine("Press any key to ex
        Console.ReadKey();
    }
}
```

Result
5: <timestamp>
6: <timestamp>
7: <timestamp>
8: <timestamp>
9: <timestamp>
15: <timestamp>
16: <timestamp>
17: <timestamp>
18: <timestamp>
19: <timestamp>
…etc

Interval - With TimeInterval() - Simple

```
internal class TimeInterval_Simple
{
    // Like TimeStamp but gives the time-inter
    private static void Main()
    {
        var observable = Observable.Interval(T

        using (observable.Subscribe(
            x => Console.WriteLine("{0}: {1}",
        {
            Console.WriteLine("Press any key t
            Console.ReadKey();
        }

        Console.WriteLine("Press any key to ex
        Console.ReadKey();
    }
}
```

Result
0: 00:00:00.8090459 *(1st value)*
1: 00:00:00.7610435 *(2nd value)*
2: 00:00:00.7650438 *(3rd value)*
…

Interval - With TimeInterval() - Remove

```
internal class TimeInterval_Remove
{
    private static void Main()
    {
        // Add a time interval
        var observable = Observable.Interval(T
```

```
        // Remove it again
        using (observable.RemoveTimeInterval()
        {
            Console.WriteLine("Press any key t
            Console.ReadKey();
        }

        Console.WriteLine("Press any key to ex
        Console.ReadKey();
    }
}
```

Result
0
1
2
...

Timeout - Simple

```
internal class Timeout_Simple
{
    private static void Main()
    {
        Console.WriteLine(DateTime.Now);

        // create a single event in 10 seconds
        var observable = Observable.Timer(Time

        // raise exception if no event receive
        var observableWithTimeout = Observable

        using (observableWithTimeout.Subscribe
            x => Console.WriteLine("{0}: {1}",
            ex => Console.WriteLine("{0} {1}",
        {
            Console.WriteLine("Press any key t
            Console.ReadKey();
        }

        Console.WriteLine("Press any key to ex
        Console.ReadKey();
    }
}
```

Result
02/12/2009 10:13:00
Press any key to unsubscribe
The operation has timed out. 02/12/2009 10:13:09
...

Timer - Simple

Observable.Interval is a simple wrapper around Observable.Timer.

```
internal class Timer_Simple
{
    private static void Main()
    {
        Console.WriteLine(DateTime.Now);

        var observable = Observable.Timer(Time


        // or, equivalently
        // var observable = Observable.Timer(D
        //

        using (observable.Subscribe(
            x => Console.WriteLine("{0}: {1}",
        {
            Console.WriteLine("Press any key t
            Console.ReadKey();
        }

        Console.WriteLine("Press any key to ex
        Console.ReadKey();
    }
}
```

Result
02/12/2009 10:02:29
Press any key to unsubscribe
0: 02/12/2009 10:02:34*(after 5s)*
1: 02/12/2009 10:02:35 *(after 6s)*
2: 02/12/2009 10:02:36 *(after 7s)*
…

Timestamp - Simple

Adds a TimeStamp to each element using the system's local time.

```
internal class Timestamp_Simple
{
    private static void Main()
    {
        var observable = Observable.Interval(T

        using (observable.Subscribe(
            x => Console.WriteLine("{0}: {1}",
        {
            Console.WriteLine("Press any key t
            Console.ReadKey();
        }

        Console.WriteLine("Press any key to ex
        Console.ReadKey();
    }
```

```
}
```

Result
0: 24/11/2009 15:40:45 *(after 1s)*
1: 24/11/2009 15:40:46 *(after 2s)*
2: 24/11/2009 15:40:47 *(after 3s)*
3: 24/11/2009 15:40:48 *(after 4s)*
…

Timestamp - Remove

```
internal class Timestamp_Remove
{
    private static void Main()
    {
        // Add timestamp
        var observable = Observable.Interval(T

        // Remove it
        using (observable.RemoveTimestamp().Su
        {
            Console.WriteLine("Press any key t
            Console.ReadKey();
        }

        Console.WriteLine("Press any key to ex
        Console.ReadKey();
    }
}
```

Result
0 *(after 1s)*
1 *(after 2s)*
2 *(after 3s)*
3 *(after 4s)*
…

Window and Joins

Window

Divides a stream into "Windows" of time. For example, 5 five second window would contain all elements pushed in that five second interval.

```
IObservable<long> mainSequence = Observable.In
IObservable<IObservable<long>> seqWindowed = m
    {
        IObservable<long> seqWindowControl = O
        return seqWindowControl;
    });

seqWindowed.Subscribe(seqWindow =>
```

```
    {
        Console.WriteLine("\nA new window into
                          DateTime.Now.ToStr
        seqWindow.Subscribe(x => { Console.Wri
    });

Console.ReadLine();
```

GroupJoin - Joins two streams matching by one of their attributes

```
var leftList = new List<string[]>();
leftList.Add(new string[] { "2013-01-01 02:00:
leftList.Add(new string[] { "2013-01-01 03:00:
leftList.Add(new string[] { "2013-01-01 04:00:

var rightList = new List<string[]>();
rightList.Add(new string[] { "2013-01-01 01:00
rightList.Add(new string[] { "2013-01-01 02:00
rightList.Add(new string[] { "2013-01-01 03:00

var l = leftList.ToObservable();
var r = rightList.ToObservable();

var q = l.GroupJoin(r,
    _ => Observable.Never<Unit>(), // windows
    _ => Observable.Never<Unit>(), // windows
    (left, obsOfRight) => Tuple.Create(left, o

// e is a tuple with two items, left and obsOf
q.Subscribe(e =>
{
    var xs = e.Item2;
    xs.Where(
     x => x[0] == e.Item1[0]) // filter only w
     .Subscribe(
     v =>
     {
        Console.WriteLine(
           string.Format("{0},{1} and {2},{3}
           e.Item1[0],
           e.Item1[1],
           v[0],
           v[1]
        ));
     });
});
```

Range

Generates a Range of values. Useful for testing purposes.

Range - Prints from 1 to 10.

```
IObservable<int> source = Observable.Range(1,
IDisposable subscription = source.Subscribe(
x => Console.WriteLine("OnNext: {0}", x),
ex => Console.WriteLine("OnError: {0}", ex.Mes
() => Console.WriteLine("OnCompleted"));
Console.WriteLine("Press ENTER to unsubscribe.
Console.ReadLine();
subscription.Dispose();
```

Generate

There are several overloads for Generate.

Generate - simple

A simple use is to replicate Interval but have the sequence stop.

```
internal class Generate_Simple
{
    private static void Main()
    {
        var observable =
            Observable.Generate(1, x => x < 6,
                                        x=>Ti

        using (observable.Subscribe(x => Conso
        {
            Console.WriteLine("Press any key t
            Console.ReadKey();
        }

        Console.WriteLine("Press any key to ex
        Console.ReadKey();
    }
}
```

Result
1: 24/11/2009 15:40:45 (after 1s)
2: 24/11/2009 15:40:46 (after 2s)
3: 24/11/2009 15:40:47 (after 3s)
4: 24/11/2009 15:40:48 (after 4s)
5: 24/11/2009 15:40:49 (after 5s)

ISubject<T> and ISubject<T1, T2>

There are several implementations for ISubject.

Ping Pong Actor Model with ISubject<T1, T2>

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace RxPingPong
{
    /// <summary>Simple Ping Pong Actor model
    /// <remarks>
    /// You'll need to install the Reactive Ex
    /// You can get the installer from <see hr
    /// </remarks>
    class Program
    {
        static void Main(string[] args)
        {
            var ping = new Ping();
            var pong = new Pong();

            Console.WriteLine("Press any key t

            var pongSubscription = ping.Subscr
            var pingSubscription = pong.Subscr

            Console.ReadKey();

            pongSubscription.Dispose();
            pingSubscription.Dispose();

            Console.WriteLine("Ping Pong has c
        }
    }

    class Ping : ISubject<Pong, Ping>
    {
        #region Implementation of IObserver<Po

        /// <summary>
        /// Notifies the observer of a new val
        /// </summary>
        public void OnNext(Pong value)
        {
            Console.WriteLine("Ping received P
        }

        /// <summary>
        /// Notifies the observer that an exce
        /// </summary>
        public void OnError(Exception exceptio
        {
            Console.WriteLine("Ping experience
        }

        /// <summary>
        /// Notifies the observer of the end o
        /// </summary>
        public void OnCompleted()
        {
            Console.WriteLine("Ping finished."
        }
```

```
        #endregion

        #region Implementation of IObservable<

        /// <summary>
        /// Subscribes an observer to the obse
        /// </summary>
        public IDisposable Subscribe(IObserver
        {
            return Observable.Interval(TimeSpa
                .Where(n => n < 10)
                .Select(n => this)
                .Subscribe(observer);
        }

        #endregion

        #region Implementation of IDisposable

        /// <summary>
        /// Performs application-defined tasks
        /// </summary>
        /// <filterpriority>2</filterpriority>
        public void Dispose()
        {
            OnCompleted();
        }

        #endregion
    }

    class Pong : ISubject<Ping, Pong>
    {
        #region Implementation of IObserver<Pi

        /// <summary>
        /// Notifies the observer of a new val
        /// </summary>
        public void OnNext(Ping value)
        {
            Console.WriteLine("Pong received P
        }

        /// <summary>
        /// Notifies the observer that an exce
        /// </summary>
        public void OnError(Exception exceptio
        {
            Console.WriteLine("Pong experience
        }

        /// <summary>
        /// Notifies the observer of the end o
        /// </summary>
        public void OnCompleted()
        {
            Console.WriteLine("Pong finished."
        }

        #endregion
```

```
        #region Implementation of IObservable<

        /// <summary>
        /// Subscribes an observer to the obse
        /// </summary>
        public IDisposable Subscribe(IObserver
        {
            return Observable.Interval(TimeSpa
                .Where(n => n < 10)
                .Select(n => this)
                .Subscribe(observer);
        }

        #endregion

        #region Implementation of IDisposable

        /// <summary>
        /// Performs application-defined tasks
        /// </summary>
        /// <filterpriority>2</filterpriority>
        public void Dispose()
        {
            OnCompleted();
        }

        #endregion
    }
}
```

Result
1: Ping received Pong.
2: Pong received Ping.
3: Ping received Pong.
4: Pong received Ping.
5: Ping received Pong.

Combination Operators

Merge

The Merge operator combine two or more sequences. In the following example, the two streams are merged into one so that both are printed with one subscription. Also note the use of "using" to wrap the Observable, thus ensuring the subscription is Disposed.

```
class Merge
{
    private static IObservable<int> Xs
    {
        get { return Generate(0, new List<int>
    }

    private static IObservable<int> Ys
    {
```

```
        get { return Generate(100, new List<in
    }

    private static IObservable<int> Generate(i
    {
        // work-around for Observable.Generate
        intervals.Add(0);

        return Observable.Generate(initialValu
                                   x => x < in
                                   x => x + 1,
                                   x => x,
                                   x => TimeSp
    }

    private static void Main()
    {
        Console.WriteLine("Press any key to un

        using (Xs.Merge(Ys).Timestamp().Subscr
            z => Console.WriteLine("{0,3}: {1}
            () => Console.WriteLine("Completed
        {
            Console.ReadKey();
        }

        Console.WriteLine("Press any key to ex
        Console.ReadKey();
    }
}
```

result
0: 11/12/2009 12:17:44
100: 11/12/2009 12:17:45
1: 11/12/2009 12:17:46
101: 11/12/2009 12:17:47
2: 11/12/2009 12:17:48
102: 11/12/2009 12:17:49
3: 11/12/2009 12:17:50
103: 11/12/2009 12:17:51
4: 11/12/2009 12:17:52
104: 11/12/2009 12:17:53

Publish - Sharing a subscription with multiple Observers

```
class Publish
{
    private static void Main()
    {
        var unshared = Observable.Range(1, 4);

        // Each subscription starts a new sequ
        unshared.Subscribe(i => Console.WriteL
        unshared.Subscribe(i => Console.WriteL

        Console.WriteLine();
```

```
        // By using publish the subscriptions
        var shared = unshared.Publish();
        shared.Subscribe(i => Console.WriteLin
        shared.Subscribe(i => Console.WriteLin
        shared.Connect();

        Console.WriteLine("Press any key to ex
        Console.ReadKey();
    }
}
```

result
Unshared Subscription #1: 1
Unshared Subscription #1: 2
Unshared Subscription #1: 3
Unshared Subscription #1: 4
Unshared Subscription #2: 1
Unshared Subscription #2: 2
Unshared Subscription #2: 3
Unshared Subscription #2: 4

Shared Subscription #1: 1
Shared Subscription #2: 1
Shared Subscription #1: 2
Shared Subscription #2: 2
Shared Subscription #1: 3
Shared Subscription #2: 3
Shared Subscription #1: 4
Shared Subscription #2: 4

Zip

```
class Zip
{
    // same code as above for Merge...

    private static void Main()
    {
        Console.WriteLine("Press any key to un

        using (Xs.Zip(Ys, (x, y) => x + y).Tim
            z => Console.WriteLine("{0,3}: {1}
            () => Console.WriteLine("Completed
        {
            Console.ReadKey();
        }

        Console.WriteLine("Press any key to ex
        Console.ReadKey();
    }
}
```

result
100: 11/12/2009 12:17:45

```
102: 11/12/2009 12:17:47
104: 11/12/2009 12:17:49
106: 11/12/2009 12:17:51
108: 11/12/2009 12:17:53
```

CombineLatest

```
class CombineLatest
{
    // same code as above for Merge...

    private static void Main()
    {
        Console.WriteLine("Press any key to un

        using (Xs.CombineLatest(Ys, (x, y) =>
            z => Console.WriteLine("{0,3}: {1}
            () => Console.WriteLine("Completed
        {
            Console.ReadKey();
        }

        Console.WriteLine("Press any key to ex
        Console.ReadKey();
    }
}
```

result

```
100: 11/12/2009 12:17:45
101: 11/12/2009 12:17:46
102: 11/12/2009 12:17:47
103: 11/12/2009 12:17:48
104: 11/12/2009 12:17:49
105: 11/12/2009 12:17:50
106: 11/12/2009 12:17:51
107: 11/12/2009 12:17:52
108: 11/12/2009 12:17:53
```

Concat - cold observable

```
class ConcatCold
{
    private static IObservable<int> Xs
    {
        get { return Generate(0, new List<int>
    }

    private static IObservable<int> Ys
    {
        get { return Generate(100, new List<in
    }

    // same Generate() method as above for Mer

    private static void Main()
    {
```

```
        Console.WriteLine("Press any key to un

        Console.WriteLine(DateTime.Now);

        using (Xs.Concat(Ys).Timestamp().Subsc
            z => Console.WriteLine("{0,3}: {1}
            () => Console.WriteLine("Completed
        {
            Console.ReadKey();
        }

        Console.WriteLine("Press any key to ex
        Console.ReadKey();
    }
}
```

result
0: 11/12/2009 12:17:45
1: 11/12/2009 12:17:46
2: 11/12/2009 12:17:47
100: 11/12/2009 12:17:48
101: 11/12/2009 12:17:49
102: 11/12/2009 12:17:50

Concat - hot observable

```
class ConcatHot
{
    private static IObservable<int> Xs
    {
        get { return Generate(0, new List<int>
    }

    private static IObservable<int> Ys
    {
        get { return Generate(100, new List<in
    }

    // same Generate() method as above for Mer

    private static void Main()
    {
        Console.WriteLine("Press any key to un

        Console.WriteLine(DateTime.Now);

        using (Xs.Concat(Ys).Timestamp().Subsc
            z => Console.WriteLine("{0,3}: {1}
            () => Console.WriteLine("Completed
        {
            Console.ReadKey();
        }

        Console.WriteLine("Press any key to ex
        Console.ReadKey();
    }
}
```

result
0: 11/12/2009 12:17:45
1: 11/12/2009 12:17:46
2: 11/12/2009 12:17:47
102: 11/12/2009 12:17:48

Make your class native to IObservable<T>

If you are about to build new system, you could consider using just IObservable<T>.

Use Subject<T> as backend for IObservable<T>

```
class UseSubject
{
    public class Order
    {
        private DateTime? _paidDate;

        private readonly Subject<Order> _paidS
        public IObservable<Order> Paid { get {

        public void MarkPaid(DateTime paidDate
        {
            _paidDate = paidDate;
            _paidSubj.OnNext(this); // Raise P
        }
    }

    private static void Main()
    {
        var order = new Order();
        order.Paid.Subscribe(_ => Console.Writ

        order.MarkPaid(DateTime.Now);
    }
}
```

reactive-extensions