

---

**THIS DOCUMENT COVERS**

- ◆ Introduction
-

# Risk and Pricing Solutions

## Introduction

The Reactive Extensions API defines a framework for managing and co-ordinating asynchronous streams of data events. Because the elements of the stream are delivered as and when they are ready, Rx is ideal for modelling infinite streams of data. The Rx framework provides a set of operators for filtering, combining and transforming data streams.

Unlike LINQ which is a pull-based API, Reactive Extensions for .NET works with push-based sources. The core interface to an event source is `IObservable<T>` and the RX operators work on instances of this type in the same way LINQ operators work on instances of `IEnumerable<T>`

### Listing 1 `IObserver<T>` and `IObservable<T>`

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

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

## Risk and Pricing Solutions

### A Simple IObservable<T> Implementation

#### Listing 2 SimpleObservable<T>

```
public class SimpleObservable<T> : IObservable<T>
{
    ❶ public IDisposable Subscribe(IObserver<T> observer)
    {
        Console.WriteLine($"Subscribe");
        _observers.Add(observer);

        return new ActionDisposable(() =>
        {
            Console.WriteLine($"Dispose");
            _observers.Remove(observer);
        });
    }

    ❷ public void Publish(T val)
    {
        foreach (var observer in _observers)
        {
            observer.OnNext(val);
        }
    }

    private readonly List<IObserver<T>>
        _observers = new List<IObserver<T>>();
}
```

When clients invoke `Subscribe` ❶ we add the given `IObserver` to our list. We also define a `Publish` ❷ method (not part of `IObservable<T>`) which iterates the list and invokes `OnNext` on each subscribed observer. We use a special implementation of `IDisposable` that takes an `Action` in its constructor and invokes this action when its `Dispose` method is invoked.

# Risk and Pricing Solutions

## Basic Lifecycle

```
// (1) Create an instance of an Observable
SimpleObservable<int> observable = new SimpleObservable<int>();

// (2) Create an instance of an IObservable
IObservable<int> observer = new SimpleObserver<int>();

// (3) Subscribe the observer to the observable
IDisposable disposable = observable.Subscribe(observer);

// (4) Observable delivers some events
observable.Publish(1);
observable.Publish(2);

// (5) After disposal the Observer no longer delivers
// events to the disposed observer
disposable.Dispose();
observable.Publish(3);
```

## Risk and Pricing Solutions

### Delegate Based Observers

We rarely ever explicitly implement `IObserver<T>` because the Reactive Framework combines a delegate-based implementation of `IObserver` with extension methods that enable us to subscribe for updates using delegates and lambdas. If the framework didn't already do it for us we could do this ourselves as follows.

#### Listing 3 Delegate Based Observer

```
public class DelegateBasedObserver<T> : IObserver<T>
{
    public DelegateBasedObserver(Action<T> nextAction)
    {
        _nextAction = nextAction;
    }

    public DelegateBasedObserver(Action<T> nextAction,
        Action completedAction) : this(nextAction)
    {
        _completedAction = completedAction;
    }

    public DelegateBasedObserver(Action<T> nextAction,
        Action<Exception> exceptAction, Action completedAction) :
        this(nextAction, completedAction)
    {
        _exceptAction = exceptAction;
    }

    public void OnNext(T value)
    {
        _nextAction?.Invoke(value);
    }

    public void OnError(Exception error)
    {
        _exceptAction?.Invoke(error);
    }

    public void OnCompleted()
    {
        _completedAction?.Invoke();
    }

    private readonly Action<T> _nextAction;
    private readonly Action _completedAction;
    private readonly Action<Exception> _exceptAction;
}
```

## Risk and Pricing Solutions

Now we need to write a static extension method that takes an instance of `IObservable<T>` and an `Action`. The extension method then creates an instance of our `DelegateBasedObserver` type and subscribes it to updates from the observable.

### Listing 4 Subscribe Extension Method

```
public static class MyObservable
{
    public static IDisposable Subscribe<T>(this IObservable<T> obs,
        Action<T> action)
    {
        return obs.Subscribe(new DelegateBasedObserver<T>(action));
    }
}
```

We can then subscribe for notification by passing in a delegate as follows

### Listing 5 Subscribing using delegates

```
SimpleObservable<int> observable = new SimpleObservable<int>();
observable.Subscribe( i => Console.WriteLine(i));
observable.Publish(5);
```

# Risk and Pricing Solutions

## Scheduling

### Default Scheduling

```
// 1. Create the observable
var observable = new SimpleObservable<int>();

// 2. Create the observer
IObserver<int> observer = new SimpleObserver<int>();

// 3. Register the observer with the observable
var disposable = observable.Subscribe(observer);

// 4. Publish a value
observable.Publish(1);

// 5. Dispose the observer
disposable.Dispose();
```

### Schedulers

The core interface we must implement if we want to define a reactive scheduler is `IScheduler`. The following code shows how to specify the scheduler on which `OnNext` methods are invoked. We can create our own very simple Scheduler in

## Risk and Pricing Solutions

Listing 6 Custom Scheduler. We can then instruct our code to observe or subscribe on this custom scheduler.



# Risk and Pricing Solutions

## Listing 6 Custom Scheduler

```
public class SingleThreadedScheduler : IScheduler
{
    public SingleThreadedScheduler(string threadName)
    {
        Thread t = new Thread(() =>
        {
            while (true)
            {
                try
                {
                    Action a = _queue.Take();
                    a();
                }
                catch (Exception)
                {
                    "SingleThreadedScheduler".Log();
                }
            }
        })
        { Name = threadName, IsBackground = false };
        t.Start();
    }

    public IDisposable Schedule<TState>(TState state, Func<IScheduler, TState, IDisposable> action)
    {
        var d = new SingleAssignmentDisposable();

        _queue.TryAdd(() =>
        {
            if (d.IsDisposed)
                return;

            d.Disposable = action(this, state);
        });

        return d;
    }

    public IDisposable Schedule<TState>(TState state, TimeSpan dueTime, Func<IScheduler, TState, IDisposable> action)
        => Disposable.Empty;

    public IDisposable Schedule<TState>(TState state, DateTimeOffset dueTime, Func<IScheduler, TState, IDisposable> action)
        => Disposable.Empty;

    public DateTimeOffset Now { get; }

    private readonly BlockingCollection<Action> _queue
        = new BlockingCollection<Action>(new ConcurrentQueue<Action>());
}
```

# Risk and Pricing Solutions

## Explicit Subscription

### Listing 7 Explicit Subscription scheduling

```
// 1. Log out the calling thread
"ExplicitMultiThreadedSubscription".Log();
Console.WriteLine("Current Thread {0}",
Thread.CurrentThread.ManagedThreadId);

// 2. Create a scheduler with its own thread
IScheduler scheduler = new SingleThreadedScheduler("KennyScheduler");

// 3. Create the observable
var observable = new SimpleObservable<int>();

// 4. Create the observer
IObserver<int> observer = new SimpleObserver<int>();

// 5. Register the observer with the observable
var disposable = observable.SubscribeOn(scheduler).Subscribe(observer);

// Make sure the publish does not happen before the subscription as
// subscription is running on a separate thread
Thread.Sleep(100);

// 6. Publish a value
observable.Publish(1);

// 7. Dispose the observer
disposable.Dispose();
```

### LISTING 8 OUTPUT

```
ExplicitMultiThreadedSubscription - Thread Main Query Thread 11
Current Thread 11
Subscribe on Thread "KennyScheduler:17"
OnNext(1) thread Main Query Thread:11
Dispose on Thread KennyScheduler:17
```

# Risk and Pricing Solutions

## Explicit Observation

### Listing 9 Explicit Observation

```
// 1. Log out the calling thread
"ExplicitMultiThreadedObservation".Log();

// 2. Create a scheduler with its own thread and a
//     wait handle to prevent premature completion
IScheduler scheduler = new SingleThreadedScheduler("KennysScheduler");
AutoResetEvent handle = new AutoResetEvent(false);

// 3. Create observable tell it we want to observer
//     on our explicit scheduler
var observable = new SimpleObservable<int>();
var disposable = observable
    .ObserveOn(scheduler)
    .Subscribe(i => i.ToString().Log(), () => handle.Set());

// 4. Publish 2 messages and then complete
observable.Publish(1);
observable.Publish(2);
observable.Complete();

handle.WaitOne();
disposable.Dispose();
```

### LISTING 10 OUTPUT

```
ExplicitMultiThreadedObservation - Thread Main Query Thread 10
Subscribe on Thread "Main Query Thread:10"
1 - Thread KennysScheduler 11
2 - Thread KennysScheduler 11
Dispose on Thread KennysScheduler:11
```

## Risk and Pricing Solutions

### Implementing Observable

Although the `IObservable<T>` looks simple, a full compliant implementation is actually rather tricky. It has to handle disposals and work correctly in multithreaded scenarios introduced by different schedulers. Consider the following implementations of `IObservable` and `IObserver`

#### Listing 11 Implementing IObservable

```
public class MyObservable<T> : IObservable<T>
{
    private readonly List<IObserver<T>> _observers = new List<IObserver<T>>();

    public IDisposable Subscribe(IObserver<T> observer)
    {
        _observers.Add(observer);
        return Disposable.Empty;
    }

    public void Publish(T m) => _observers.ForEach(obs => obs.OnNext(m));
}

public class MyObserver<T> : IObserver<T>
{
    public void OnNext(T value) => Console.WriteLine(value);
    public void OnError(Exception error) => Console.WriteLine("Error");
    public void OnCompleted() => Console.WriteLine("Completed");
}
```

We then write code to subscribe an instance of `MyObserver` to `MyObservable`. Finally, we publish a value from `MyObservable`, dispose the observer and publish another value through the `MyObservable`.

```
MyObservable<int> observable = new MyObservable<int>();
MyObserver<int> observer = new MyObserver<int>();
IDisposable disposable = observable.Subscribe(observer);

observable.Publish(1);
disposable.Dispose();
observable.Publish(2);
```

As we might expect disposing the observable has no effect as our `MyObservable` returns an empty disposable from its `Subscribe` method. It has no logic to do the unsubscription.

The output from our code becomes

1  
2

## Risk and Pricing Solutions

This highlights the first implicit behaviour we need to support when creating RX sources, namely unsubscribing observers when they are disposed.

### Observable.Create

We can fix the code from the previous section such that it stops delivering events to IObservable instances that have been unsubscribed by using the static `Observable.Create` method.

```
MyObservable<int> observable = new MyObservable<int>();
MyObserver<int> observer = new MyObserver<int>();

var dec = Observable.Create<int>(o => observable.Subscribe(o));
var disposable = dec.Subscribe(observer);

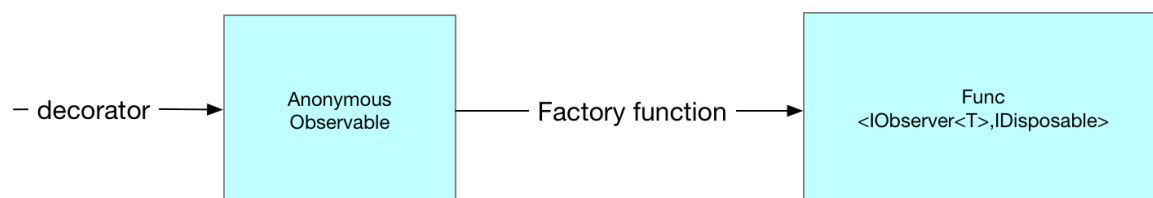
observable.Publish(1);
disposable.Dispose();
observable.Publish(2);
```

Now the output from running this code is what we want. The first publish causes the observer to write 1 to the console but the second publish after the dispose call is suppressed.

1

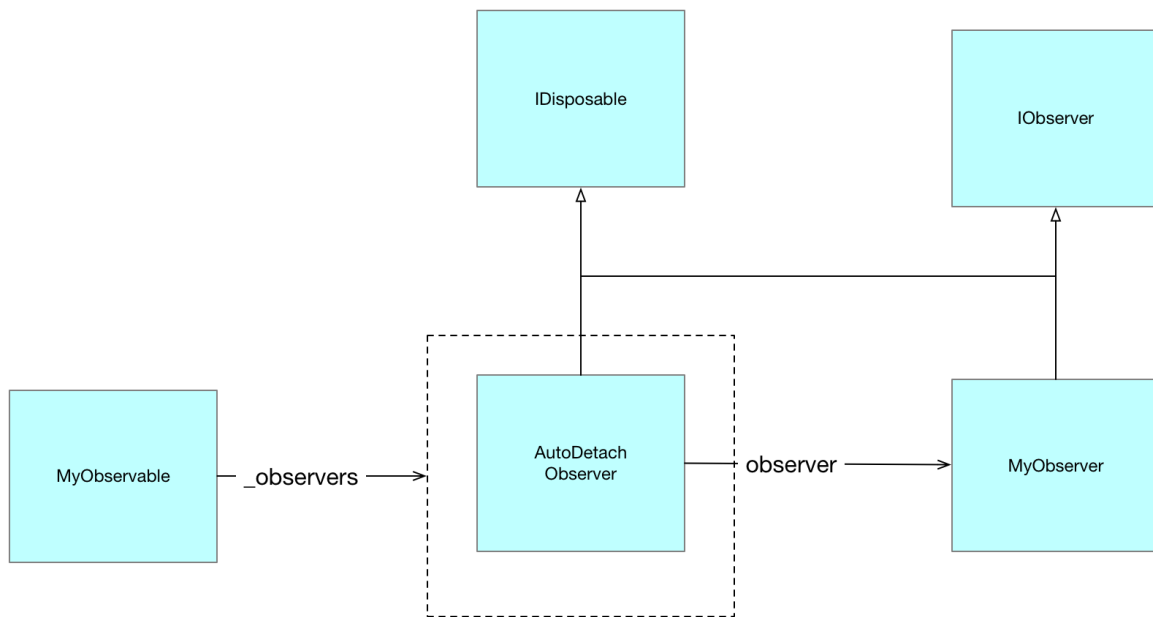
What happens here is that when we call `Observable.Create` it creates an instance of a framework type called `AnonymousObservable` which stores a reference to the factory method we pass into it.

#### Listing 12 AnonymousObservable



Then when we call `Subscribe` on our observable and pass in our `MyObserver` instance we are actually calling `Subscribe` on the `AnonymousObservable` instance. This in turn creates an instance of a type called `AutoDetachObserver` which it gives to the factory method we passed into `Observable.Create`. The `AutoDetachObserver` holds a reference to our actual `MyObserver` creating an extra layer of indirection.

### Listing 13 AutoDetachObserver



Notice how the **AutoDetachObserver** implements **IDisposable**. It is the **AutoDetachObserver** which is returned to the client when it invokes `subscribe` on the instance of **AnonymousObserver**. This enables the **AutoDetachObserver** to stop delivering updates to the **MyObserver** after `dispose` has been invoked on it.

**Observable.Create** is the preferred means of creating observable sequences as internally it has been carefully coded to ensure correct order of subscriptions and notification in multi threaded scenarios

## Risk and Pricing Solutions

### Hot and Cold Observables

Rx makes the distinction between hot and cold observables. A cold observable only ever produces values at the point an observer subscribes and each subscriber is given its own set of data. A subscriber can never miss out on data by subscribing late. A hot observable on the other hand is always producing data irrespective of whether any observers are subscribed. With a hot observable a subscriber can miss out on earlier values if it subscribes late. We consider cold and hot observables in turn.

#### Cold Observable

The basic characteristic of a cold observable is that nothing is done until a subscription is made and each subscription gets different values

#### Listing 14 Basic Cold Observable

```
// We use a factory method together with Observable.Create to create
// an IObservable which delivers completely different values to each
// Subscription. This is the basic property of a ColdObservable.
// Nothing is delivered until a subscription is made and each
// subscription gets a different value.
int x = 0;

// Define a factory method that when invoked directly calls OnNext
IDisposable FactMeth(IObserver<int> observer)
{
    observer.OnNext(x++);
    return Disposable.Empty;
}

// Use Observable.Create to turn our factory method into an IObservable
var observable = Observable.Create((Func<IObserver<int>,
IDisposable>)FactMeth);

// Perform two different subscriptions. Each IOBserver
// get different values to the nature of a cold observable
observable.Subscribe(i => Console.WriteLine($"A {i}"));
observable.Subscribe(i => Console.WriteLine($"B {i}"));
```

#### LISTING 15 BASIC COLD OBSERVABLE OUTPUT

```
A 0
B 1
```

## Risk and Pricing Solutions

### Connectable Observable

#### Listing 16 Connectable Observable

```
// As per the previous sample we use a factory method together
// with Observable.Create to create an IObservable which delivers
// completely different values from each subscription. The key
// difference is that we wrap this Observable
// with a ConnectableObservable
// using the Publish extension method. This extra layer allows us to
// share the values published from the originating Observable as the
// Connectable wrapper performs the multiplexing
int x = 0;

// Define a factory method that when invoked directly calls OnNext
IDisposable FactMethod(IObserver<int> observer)
{
    observer.OnNext(x++);
    return Disposable.Empty;
}

// Use the Observable.Create to turn our factory method into an
IObservable
IObservable<int> observable = Observable.Create((Func<IObserver<int>,
IDisposable>)FactMethod);

// Wrap the source Observable in a Connectable observable
IConnectableObservable<int> connectableObservable = observable.Publish();

// Even though we subscribe twice the connectable observable
// will make sure
// there is only one underlying Observable
// with the ConnectableObservable
// providing multi-plexing
connectableObservable.Subscribe(i => Console.WriteLine($"A {i}"));
connectableObservable.Subscribe(i => Console.WriteLine($"B {i}"));

// The subscription is now carried out and multiplexed out to the
// registered observers
connectableObservable.Connect();
```

#### LISTING 17 CONNECTABLE OBSERVABLE OUTPUT

```
A 0
B 0
```



# Risk and Pricing Solutions

## Hot Observable

### Listing 18 Hot Observable

```
// In this example we wrap our observable in a ConnectableObservable
// and connect it before we make any subscriptions. By doing this we are
// creating what is known as a Hot Observable. This Observable is still
// publishing values even though it has no subscriptions.
SimpleObservable<int> sourceObservable = new SimpleObservable<int>();

IConnectableObservable<int> hotObservable = sourceObservable
    .Do(i => Console.WriteLine("Source({0}) thread {1}", i,
Thread.CurrentThread.ManagedThreadId))
    .Publish();

// Connecting to the IConnectableObservable causes it to subscribe on
// the sourceObservable thereby setting up a hot observable which will
// publish out even when the connectableObservable has no observers
IDisposable disposable = hotObservable.Connect();

// Tis logged via the Do call even though we have no observer on
// the connectableObservable
sourceObservable.Publish(1);

// now we subscribe on the connectableObservable
hotObservable.Subscribe(i => Console.WriteLine("OnNext({0}) thread {1}",
i, Thread.CurrentThread.ManagedThreadId));

// this is now delivered to the IObservable
sourceObservable.Publish(2);

// Disposing of the connectableObservable turns off the publishing
disposable.Dispose();
sourceObservable.Publish(3);

// we can now reconnect to the same IConnectableObservable and once again
// messages are delivered
disposable = hotObservable.Connect();
sourceObservable.Publish(4);
```

### LISTING 19 HOT OBSERVABLE OUTPUT

```
Subscribe on Thread "Main Query Thread:12"
Source(1) thread 12
Source(2) thread 12
OnNext(2) thread 12
Dispose on Thread Main Query Thread:12
Subscribe on Thread "Main Query Thread:12"
Source(4) thread 12
OnNext(4) thread 12
```

# Risk and Pricing Solutions

## Creating Sequences

Sequences can be created in three ways

- ◆ Factory methods
- ◆ Functional unfolds
- ◆ Transitioning from other entities (delegates, tasks, events)

## Factory Methods

### OBSERVABLE.CREATE

```
IObservable<int> s =  
    Observable.Create<int>(observer =>  
    {  
        observer.OnNext(1);  
        observer.OnNext(2);  
        observer.OnNext(3);  
        observer.OnCompleted();  
        return Disposable.Empty;  
    });  
  
s.Subscribe(i =>  
    WriteLine($"OnNext({i})"),  
    () => WriteLine("OnCompleted"));
```

### OBSERVABLE.EMPTY

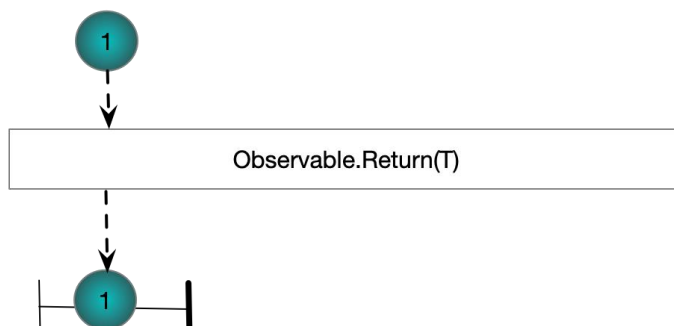
Returns an empty stream which simply invokes OnCompleted to end the sequence

```
IObservable<int> s =  
    Observable.Empty<int>();  
  
s.Subscribe(i => WriteLine($"OnNext({i})"),  
    () => WriteLine("OnCompleted"));
```

### OBSERVABLE.RETURN<T>

Returns a stream consisting of a single value.

```
Observable.Return(1)  
    .Subscribe(WriteLine, () => WriteLine("OnCompleted"));
```



# Risk and Pricing Solutions

## OBSERVABLE.THROW

Returns a stream which calls `OnError` and then returns

```
IObservable<int> s =  
    Observable.Throw<int>(new Exception("An exception"));  
s.Subscribe(i => WriteLine($"OnNext({i})"),  
    exception => WriteLine("OnException"),  
    () => WriteLine("OnCompleted"));
```

## Functional Unfolds

An unfold can be used to produce a possibly infinite sequence. `Generate` is the root Rx unfold method. The other unfold methods such as `Range`, `Interval` and `Timer` could be produced using `Generate`.

## OBSERVABLE.GENERATE

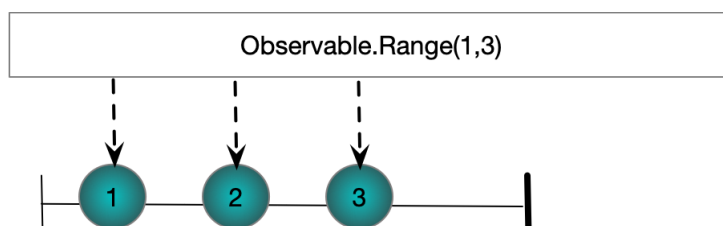
Supports the creation of more complex sequences of event.

```
IObservable<int> s =  
    Observable.Generate(0, i => i < 5, i => i + 1, i => i);  
  
s.Subscribe(i => WriteLine($"OnNext({i})"),  
    exception => WriteLine("OnException"),  
    () => WriteLine("OnCompleted"));
```

## OBSERVABLE.RANGE(INT,INT)

Returns an observable over a sequence of consecutive integers

```
Observable.Range(1, 3)  
    .Subscribe(WriteLine, () => WriteLine("OnCompleted"));
```



## Risk and Pricing Solutions

### **OBSERVABLE.INTERVAL(TIMESPAN)**

Produces incrementally increasing integers. The gap between elements is defined by the given Timespan

```
Observable
    .Interval(TimeSpan.FromSeconds(0.25))
    .Take(4)
    .ObserveOn(Scheduler.Default)
    .Subscribe(l => WriteLine($"{l}"),
        () => tcs.SetResult(null));
```

### **OBSERVABLE.TIMER(TIMESPAN)**

Takes a timespan and creates a stream of one value. The single value is delivered after the specified TimeSpan has elapsed

```
TaskCompletionSource<string> tcs = new TaskCompletionSource<string>();

Observable
    .Timer(TimeSpan.FromSeconds(1))
    .ObserveOn(Scheduler.Default)
    .Subscribe(l => WriteLine($"{l}"), () => tcs.SetResult(null));
```

### **OBSERVABLE.TIMER(TIMESPAN, TIMESPAN)**

Takes two TimeSpan objects. The first determines how much time should be allowed to elapse before the first element is delivered. The second TimeSpan determines the interval between subsequent events.

```
TaskCompletionSource<string> tcs = new TaskCompletionSource<string>();

Observable
    .Timer(TimeSpan.FromSeconds(2), TimeSpan.FromSeconds(0.1))
    .Take(4)
    .ObserveOn(Scheduler.Default)
    .Subscribe(l => WriteLine($"{l}"), () => tcs.SetResult(null));
```

## Transitioning from Other APIs

We can use the methods of Rx to create Observables from other .NET types.

### **OBSERVABLE.START(ACTION)**

Creates a single value observable from an Action delegate

```
Action a = () => { };
Observable
    .Start(a)
    .Subscribe(unit => WriteLine($"OnNext({unit})"),
        () => WriteLine("OnCompleted"));
```

# Risk and Pricing Solutions

## **OBSERVABLE.START(Func<T>)**

Creates a single value observable from a Func<T>. The value is the value returned by the Func<T>

```
Action a = () => { };
Observable
    .Start(a)
    .Subscribe(unit => WriteLine($"OnNext({unit})"),
               () => WriteLine("OnCompleted"));
```

## **OBSERVABLE.FROMEVENTPATTERN**

Creates Observables from standard .NET events

```
IObservable<EventPattern<EventArgs>> obs
    = Observable
        .FromEventPattern<EventArgs>(h => MyEvent += h, h => MyEvent -= h);

obs.Subscribe(x => WriteLine(x.EventArgs));
MyEvent?.Invoke(this, new EventArgs());
```

## **TASK.TO OBSERVABLE**

Creates Observables from standard tasks

```
TaskCompletionSource<string> tcs = new TaskCompletionSource<string>();
tcs.SetResult("Value");
tcs
    .Task
    .ToObservable()
    .Subscribe(x => WriteLine($"OnNext({x})"), () =>
WriteLine("OnCompleted"));
WriteLine("Subscribed to already completed task\n");

Task
    .Run(() => "Value")
    .ToObservable()
    .Subscribe(x => WriteLine($"OnNext({x})"), () =>
WriteLine("OnCompleted"));
WriteLine("Subscribed to running task");
```

## **IEnumerable<T>.TOOBSERVABLE**

```
Enumerable
    .Range(0, 3)
    .ToObservable()
    .Subscribe(x => WriteLine($"OnNext({x})"), () =>
WriteLine("OnCompleted"));
```

## Risk and Pricing Solutions

### Composing Sequences

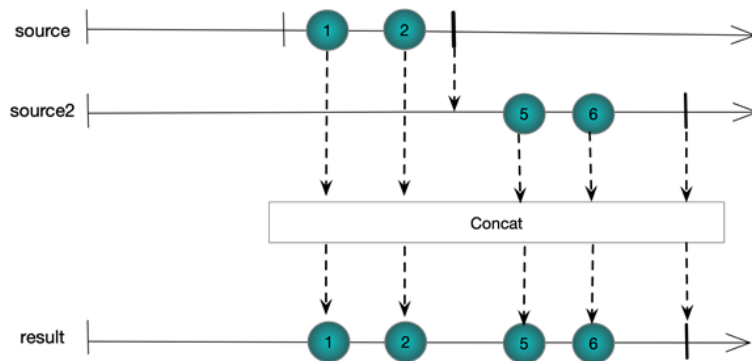
Sequences can be combined using three classes of compositors

1. Sequential composition
2. Concurrent compositions
3. Pairwise composition

### Sequential Composition

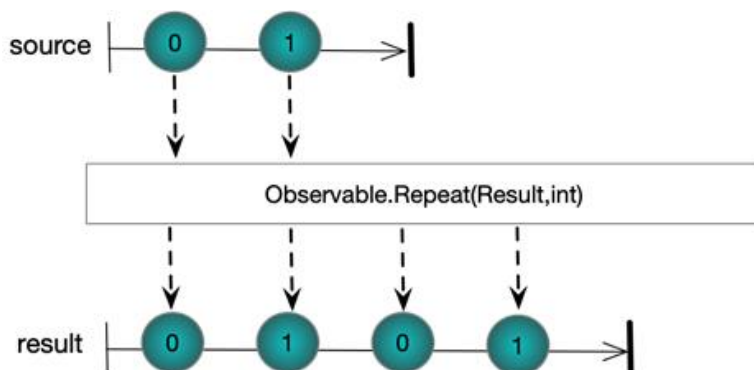
#### **OBSERVABLE.CONCAT(IObservable<TSource>)**

```
Observable  
    .Range(0, 2)  
    .Concat(Observable.Range(5, 2))  
    .Subscribe(WriteLine);
```



#### **OBSERVABLE.REPEAT(INT REPS)**

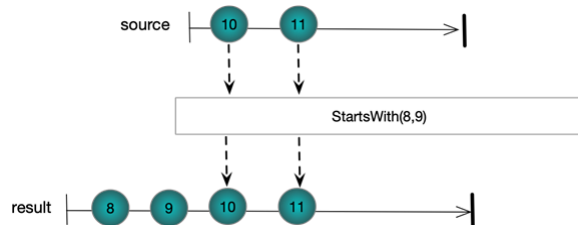
```
Observable  
    .Range(0, 2)  
    .Repeat(2)  
    .Subscribe(WriteLine, () => WriteLine("OnCompleted\n"));
```



# Risk and Pricing Solutions

## OBSERVABLE.STARTSWITH

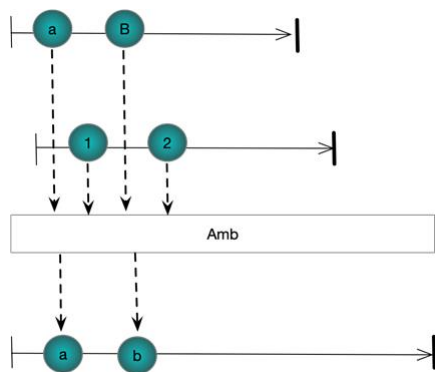
```
Observable  
  .Range(10, 2)  
  .StartWith(8, 9)  
  .Subscribe(WriteLine);
```



## Concurrent Composition

### OBSERVABLE.AMB

```
Subject<string> a = new Subject<string>();  
Subject<string> b = new Subject<string>();  
  
Observable.Amb(a,b)  
  .Subscribe(WriteLine);  
  
a.OnNext("a");  
b.OnNext("1");  
a.OnNext("b");  
b.OnNext("2");
```



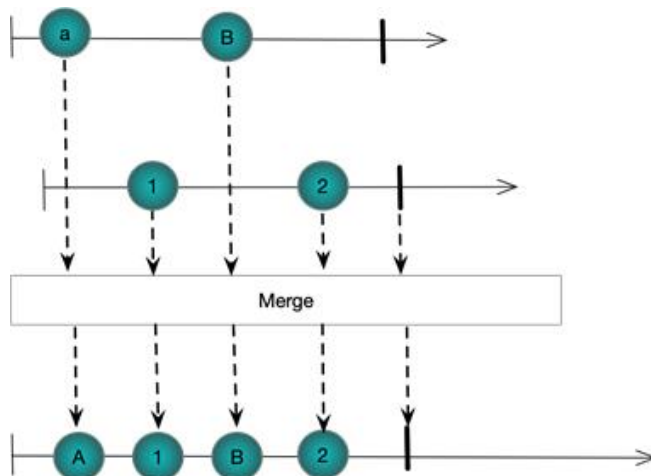
# Risk and Pricing Solutions

## OBSERVABLE.MERGE

```
Subject<string> a = new Subject<string>();  
Subject<string> b = new Subject<string>();
```

```
Observable.Merge(a, b)  
    .Subscribe(WriteLine);
```

```
a.OnNext("a");  
b.OnNext("1");  
a.OnNext("b");  
b.OnNext("2");
```





## Risk and Pricing Solutions

### OBSERVABLE.SWITCH

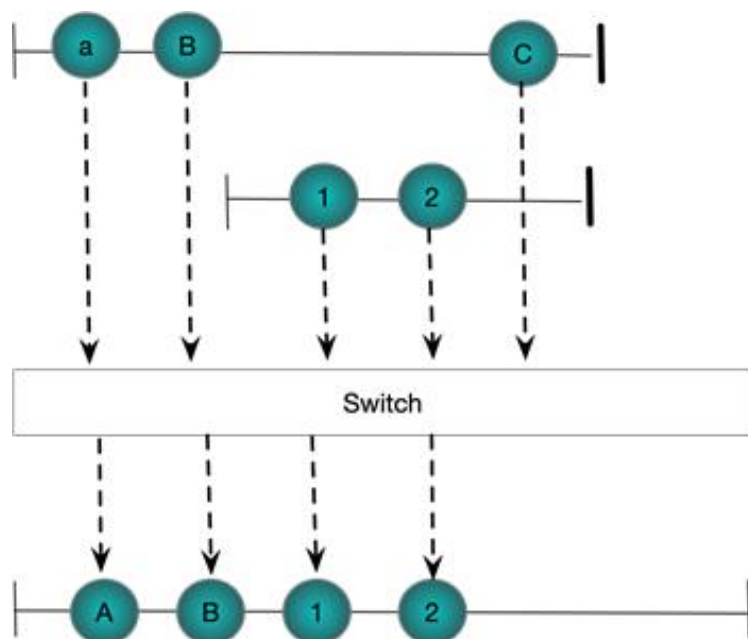
Works on a stream of streams. When the first stream starts publishing, its events are published into the result stream until the second stream starts publishing. At which point the first stream is unsubscribed

```
Subject<string> a = new Subject<string>();
Subject<string> b = new Subject<string>();

Subject<Subject<string>> master = new Subject<Subject<string>>();

master.Switch()
    .Subscribe(WriteLine);

master.OnNext(a);
a.OnNext("a");
a.OnNext("b");
master.OnNext(b);
b.OnNext("1");
b.OnNext("2");
a.OnNext("c");
```



## Pariwise Composition

### OBSERVABLE.COMBINELATEST

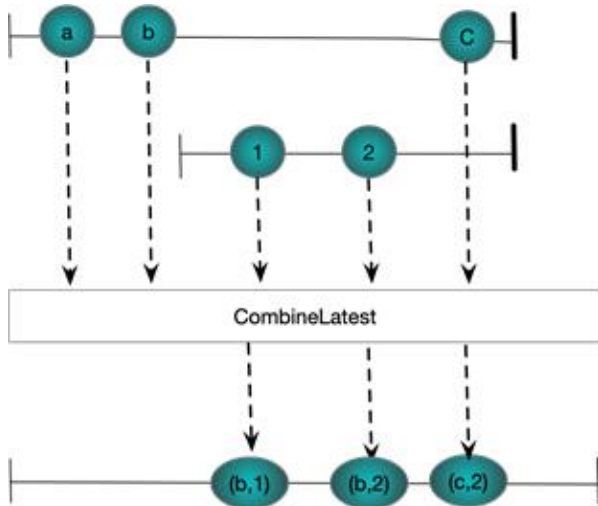
Combines the latest value from two streams as each stream produces new values. Requires that each stream has at least one value before anything is published to the result

```
Subject<string> a = new Subject<string>();
Subject<string> b = new Subject<string>();

Observable.CombineLatest(a,b,(s, s1) => $"({s},{s1})")
    .Subscribe(WriteLine);
```

## Risk and Pricing Solutions

```
a.OnNext("a");  
a.OnNext("b");  
b.OnNext("1");  
b.OnNext("2");  
a.OnNext("c");
```

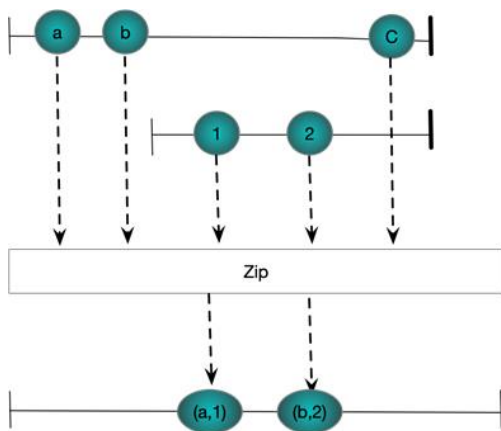


# Risk and Pricing Solutions

## OBSERVABLE.ZIP

Pairs together values from two streams.

```
Subject<string> a = new Subject<string>();  
Subject<string> b = new Subject<string>();  
  
Observable.Zip(a,b, (s, s1) => $"({s},{s1})")  
    .Subscribe(WriteLine);  
  
a.OnNext("a");  
a.OnNext("b");  
b.OnNext("1");  
b.OnNext("2");  
a.OnNext("c");
```



# Risk and Pricing Solutions

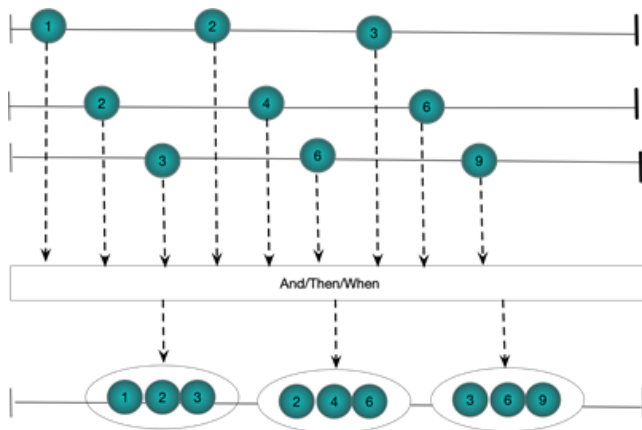
## AND/THEN/WHEN

Pairs together values from ,multiple streams.

```
IObservable<int> a = Observable.Range(1, 3);
IObservable<int> b = Observable.Range(1, 3).Select(x => x * 2);
IObservable<int> c = Observable.Range(1, 3).Select(x => x * 3);

Observable
    .When(a
        .And(b)
        .And(c)
        .Then((x, y, z) => (x, y, z)))
    .Subscribe(x => WriteLine(x));

// Verbose form to show what is happening
Pattern<int, int> pattern1 = a.And(b);
Pattern<int, int, int> pattern2 = pattern1.And(c);
Plan<(int, int, int)> then = pattern2.Then((i, i1, i2) => (i, i1, i2));
IObservable<(int, int, int)> observable = Observable.When(then);
observable.Subscribe(x => WriteLine(x));
```



## Risk and Pricing Solutions

### Time Shifting

#### Buffering

The various overloads of Buffer enable one to group together elements from an input stream. The resulting groups are called buffers. The basic idea then is to take a stream  $\text{IObservable}<\text{T}>$  and produce a stream  $\text{IObservable}<\text{IList}<\text{T}>$ . Buffers can be perfectly contiguous with every source element existing in one and only one result buffer; skipping with some source elements being completely left out of the result buffers or overlapping where some source elements make it into more than one destination buffer.

#### **BUFFERING BEHAVIOUR**

- ◆ Perfectly contiguous
- ◆ Overlapping
- ◆ Skipping

Buffers can be defined by source element count, time interval or via open and close signals.

#### **BUFFER DEFINITION**

- ◆ Source element count
- ◆ time interval
- ◆ opening and closing signals

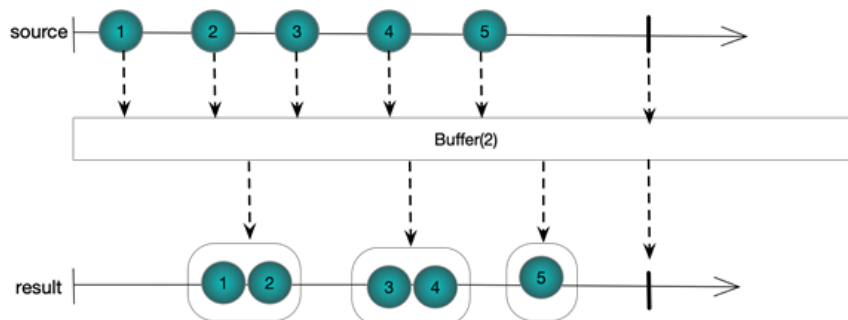
## Risk and Pricing Solutions

### BUFFER(INT COUNT)

Supports perfectly contiguous buffers in the result stream

```
Observable  
  .Range(0, 5)  
  .Buffer(2)  
  .Subscribe(ints => WriteLine(string.Join(", ", ints)));
```

The following figure shows graphically what is happening. Each buffer contains at most count items. Notice the final buffer has less than two items because the source stream completes



## Risk and Pricing Solutions

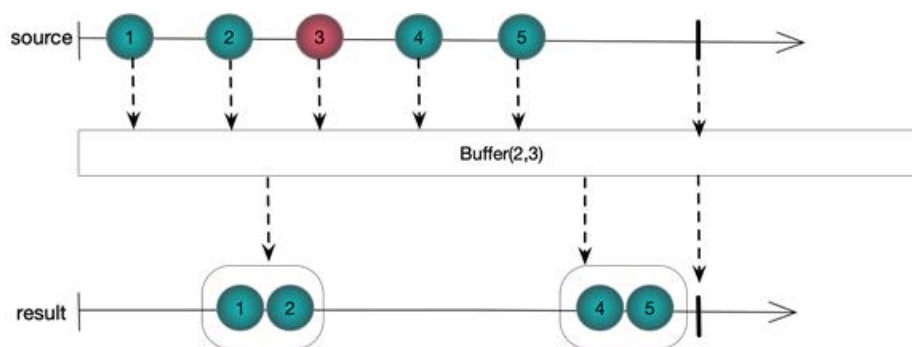
### BUFFER(INT COUNT,INT SKIP)

By taking two integer arguments this overload of Buffer supports non-contiguous result buffers. Skip defines the number of elements between each buffer opening and count defines the maximum number of elements in each buffer. If skip is more than count then we miss out some source elements from the destination. If skip is less than count then we have overlapping buffers with some elements making it into more than one buffer. Of course if skip is equal to count we have the exact same behaviour as Buffer(int count), that is to say perfectly contiguous buffers. Let us look at each case in turn

#### Skipping Elements

We set the count to 2 and the skip to 3. A new buffer is started every three elements and hold at most 2 elements.

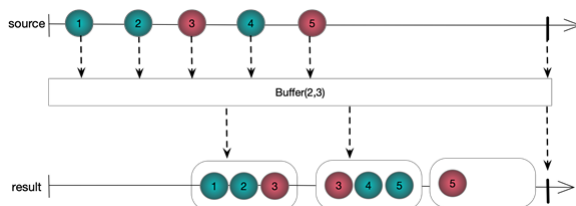
```
Observable.Range(1, 5)
    .Buffer(2, 3)
    .Subscribe(ints => WriteLine(string.Join(", ", ints)));
```



#### Overlapping Elements

We set the count to 3 and the skip to 2. A new buffer is started every two elements and hold at most three elements. The last element from each buffer is also duplicated as the first element in the next buffer

```
Observable.Range(1, 5)
    .Buffer(3, 2)
    .Subscribe(ints => WriteLine(string.Join(", ", ints)));
```



## Risk and Pricing Solutions

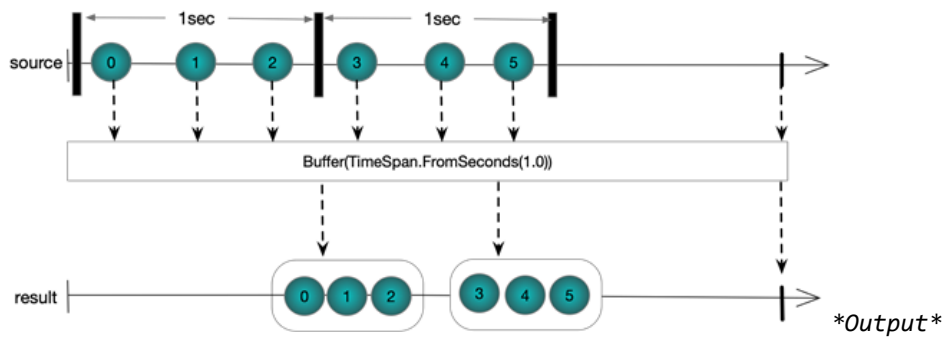
### **BUFFER(TIMESPAN TIMESPAN)**

Partitions each element of a source stream into contiguous buffers in the result stream. The buffers in the result stream are defined by the timing of events in the source stream.

```
EventWaitHandle ewh = new AutoResetEvent(false);

Observable
    .Interval(TimeSpan.FromSeconds(0.3))
    .Take(6)
    .Buffer(TimeSpan.FromSeconds(1.0))
    .Subscribe(ints => WriteLine(string.Join(", ", ints)), ()=>ewh.Set());

ewh.WaitOne();
```





# Risk and Pricing Solutions

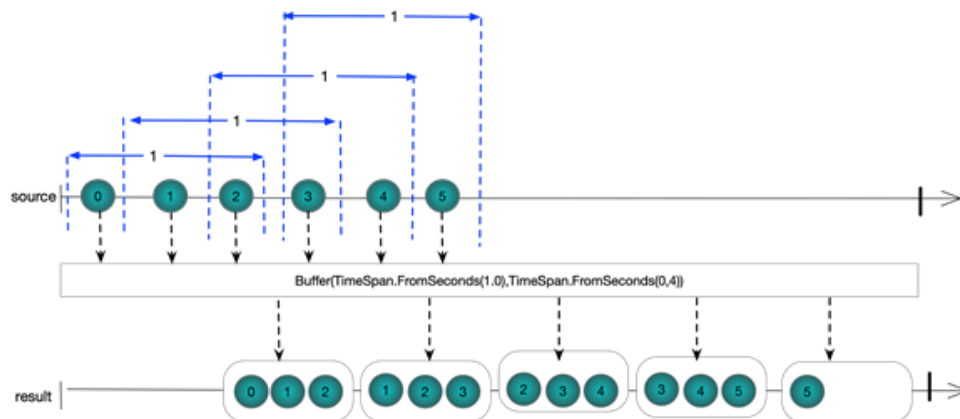
## BUFFER(TIMESPAN TIMESPAN, TIMESPAN TIMESHIFT)

As with using Buffer(int,int) this overloads support overlapping and skipping elements by specifying two TimeSpans. We consider each in turn

### Overlapping Elements

```
EventWaitHandle ewh = new AutoResetEvent(false);
Observable
    .Interval(TimeSpan.FromSeconds(0.3))
    .Take(6)
    .Buffer(TimeSpan.FromSeconds(1.0), TimeSpan.FromSeconds(0.4))
    .Subscribe(ints => WriteLine(string.Join(",", ints)), () => ewh.Set());

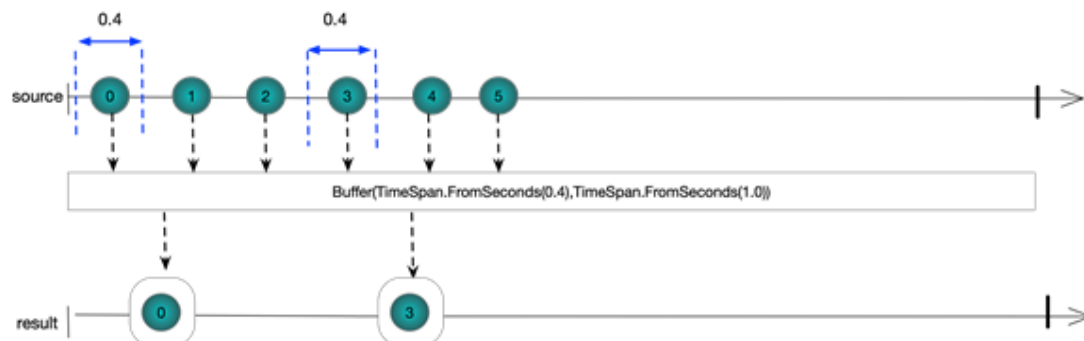
ewh.WaitOne();
```



### Skipping Elements

```
EventWaitHandle ewh = new AutoResetEvent(false);
Observable
    .Interval(TimeSpan.FromSeconds(0.3))
    .Take(6)
    .Buffer(TimeSpan.FromSeconds(0.4), TimeSpan.FromSeconds(1.0))
    .Subscribe(ints => WriteLine(string.Join(",", ints)), () => ewh.Set());

ewh.WaitOne();
```



## Risk and Pricing Solutions

### **BUFFER(FUNC<IOBSERVABLE<TCLOSINGSELECTOR>> CLOSINGSELECTOR)**

This form uses a separate observable sequence that produces values in order to flush buffers. Each time a element is published by this observable the current buffer is flushed through

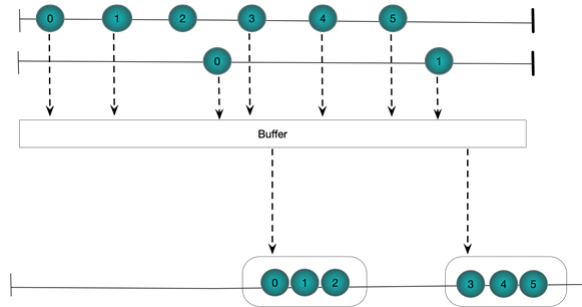
```
EventWaitHandle latch = new AutoResetEvent(false);

var obs = Observable
    .Interval(TimeSpan.FromSeconds(0.3))
    .Take(10);

var closing = Observable
    .Interval(TimeSpan.FromSeconds(1.0))
    .Take(2);

obs
    .Buffer(closing)
    .Subscribe(ints => WriteLine(string.Join(",", ints)), ()=>latch.Set());

latch.WaitOne();
```



## Risk and Pricing Solutions

**BUFFER(IObservable<TBufferOpening> BUFFEROPENINGS, FUNC<TBufferOpening, IObservable<TBufferOpening>> BUFFERCLOSINGSELECTOR)**

The bufferOpenings selector is used to publish events which determine when buffers are opened. The bufferClosingSelector takes the value from the buffer opening event and uses it to generate the buffer closing event.

```
EventWaitHandle latch = new AutoResetEvent(false);

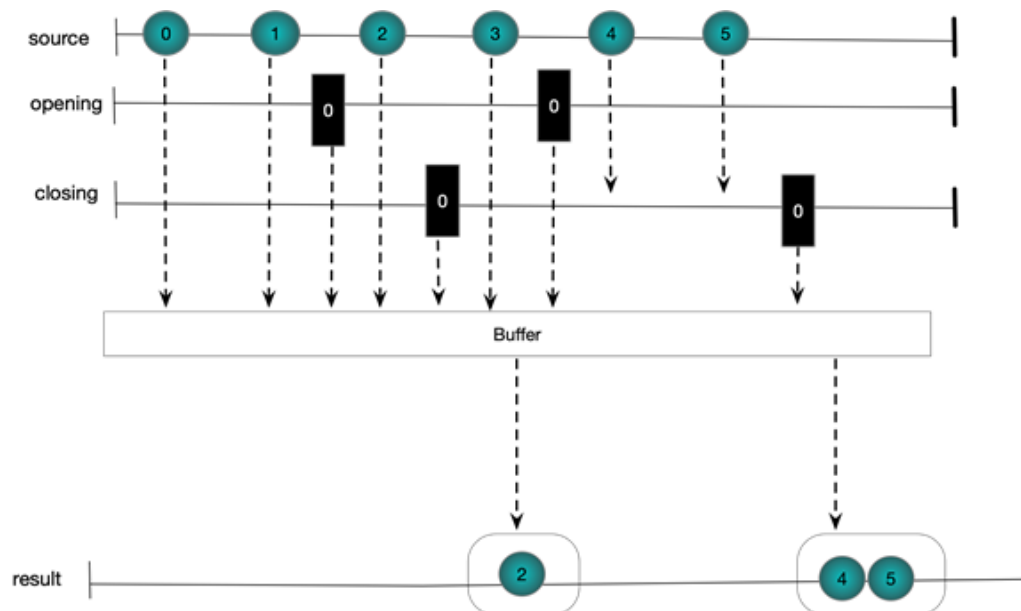
var obs = Observable
    .Interval(TimeSpan.FromSeconds(0.3))
    .Take(10);

var opening = Observable
    .Interval(TimeSpan.FromSeconds(0.7))
    .Take(2);

var closing = Observable
    .Timer(TimeSpan.FromSeconds(0.5))
    .Take(2);

obs
    .Buffer(opening, i => closing)
    .Subscribe(ints => WriteLine($"({string.Join(",", ints)})"), () =>
        latch.Set());

latch.WaitOne();
```



## Risk and Pricing Solutions

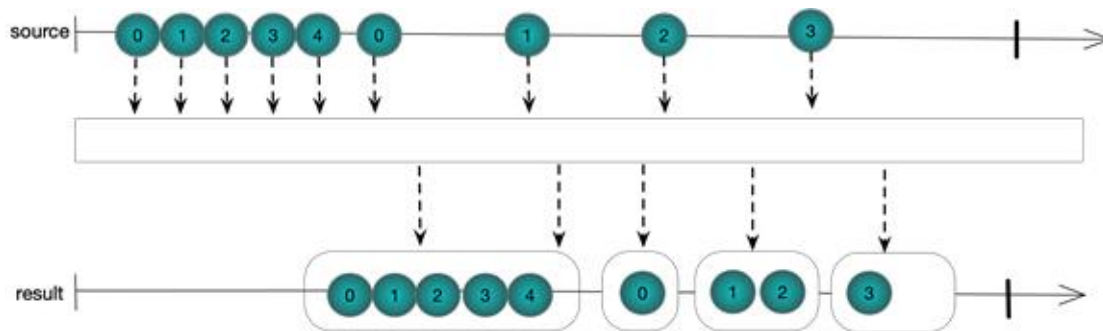
### **BUFFER(TIMESPAN TIMESPAN, INT COUNT)**

Sometimes it is useful to be able to buffer by both count and timeframe. A buffer is closed and a new one started when either the maximum buffer size is reached or the timespan elapses. This means the buffer does not get too big and also data never gets stale

```
EventWaitHandle latch = new AutoResetEvent(false);

Observable
    .Interval(TimeSpan.FromSeconds(0.1))
    .Take(5)
    .Concat(Observable.Interval(TimeSpan.FromSeconds(0.5)).Take(4))
    .Buffer(TimeSpan.FromSeconds(1.0), 5)
    .Subscribe(ints => WriteLine($"{string.Join(", ", ints)}"), () =>
        latch.Set());

latch.WaitOne();
```



# Risk and Pricing Solutions

## Delay

### DELAY (TIMESPAN)

Simply delays a sequence by a specified TimeSpan. The times between elements remain the same.

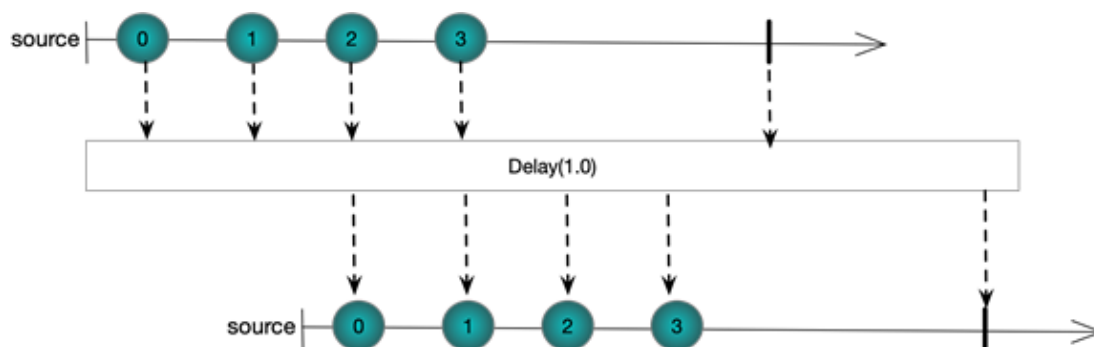
```
DateTime now = DateTime.Now;
EventWaitHandle latch = new AutoResetEvent(false);

var source = Observable.Interval(TimeSpan.FromSeconds(0.5)).Take(4);
var delays = source.Delay(TimeSpan.FromSeconds(1.0));

source.Subscribe(l => WriteLine($"Original {l}    {(DateTime.Now -
now).TotalSeconds}"));
delays.Subscribe(l => WriteLine($"Delayed {l}    {(DateTime.Now -
now).TotalSeconds}"), () => latch.Set());

latch.WaitOne();
```

```
Original 0    0.5671419
Original 1    1.0641751
Original 2    1.5644976
Delayed 0     1.6014948
Original 3    2.0791571
Delayed 1     2.0985018
Delayed 2     2.6119969
Delayed 3     3.1114934
```



# Risk and Pricing Solutions

## Sample

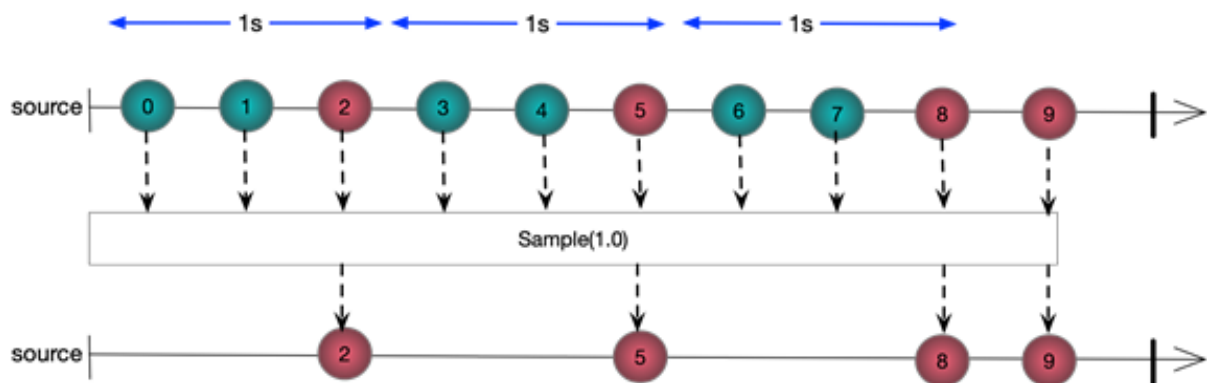
### SAMPLE (TIMESPAN)

Returns the last element emitted by a source sequence in a buffer interval specified by a `TimeSpan`

```
DateTime now = DateTime.Now;
EventWaitHandle waitHandle = new AutoResetEvent(false);

Observable
    .Interval(TimeSpan.FromSeconds(0.3))
    .Take(10)
    .Sample(TimeSpan.FromSeconds(1.0))
    .Subscribe(l => Console.WriteLine($"Delayed {l}    {(DateTime.Now -
now).TotalSeconds}"), () => waitHandle.Set());

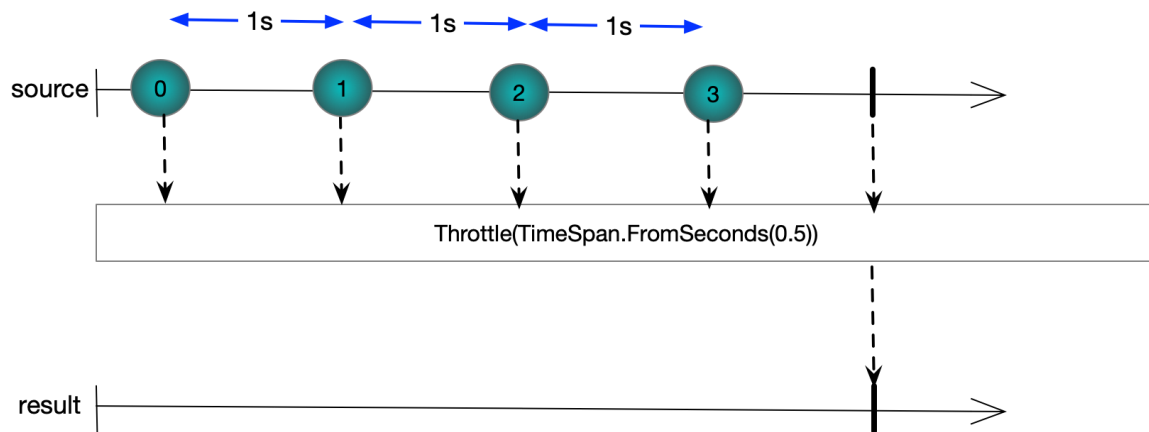
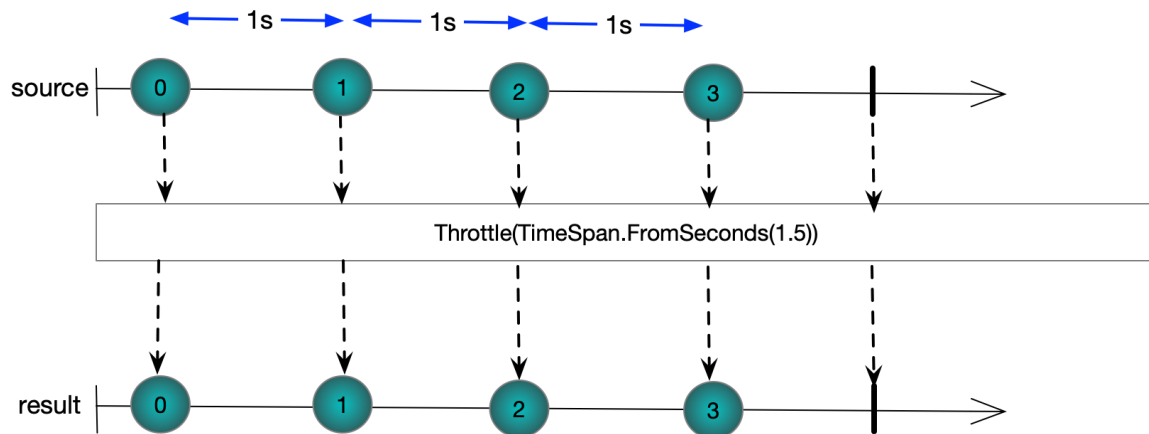
waitHandle.WaitOne();
```



```
Delayed 2    1.1075141
Delayed 5    2.1179985
Delayed 8    3.1327291
Delayed 9    4.1504981
```

# Risk and Pricing Solutions

## More Examples



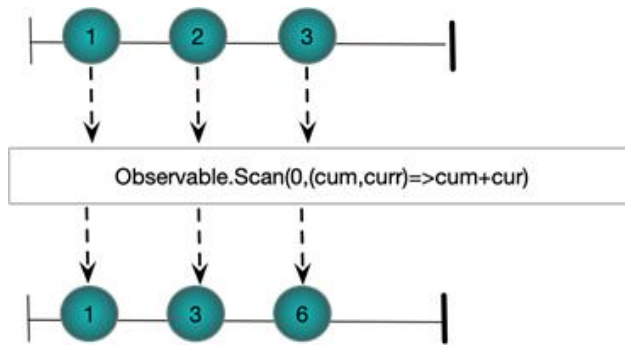
## Risk and Pricing Solutions

### Aggregating

#### **SCAN(TACCUMULATE, FUNC<TACCUMULATE, TSource, TACCUMULATE>)**

An accumulator which produces a sequence of accumulated values

```
Observable.Range(1, 3)
    .Scan(0, (cum, i1) => cum+i1)
    .Subscribe(WriteLine, () => WriteLine("OnCompleted\n"));
```





# Risk and Pricing Solutions

## Transforming Streams

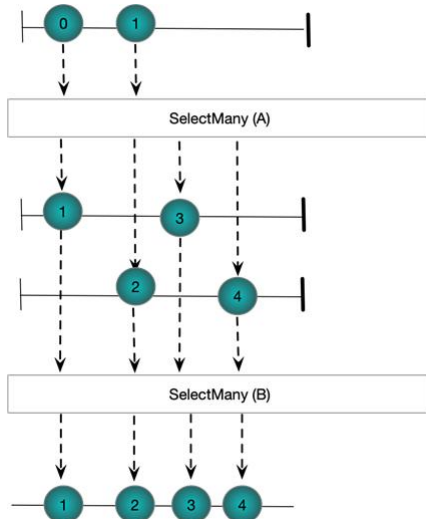
### SELECTMANY

SelectMany seems to work by taking an observable sequence and using each value from that sequence to generate another sequence. The generated sequences are then merged (flattened) together

```
Subject<int>[] subs = new Subject<int>[]  
{  
    new Subject<int>(),  
    new Subject<int>()  
};
```

```
Observable  
    .Range(0, 2)  
    .SelectMany(i => subs[i])  
    .Subscribe(WriteLine);
```

```
subs[0].OnNext(1);  
subs[1].OnNext(2);  
subs[0].OnNext(3);  
subs[1].OnNext(4);
```



## Questions

### INTRODUCTION

#### What is RX?

*API for asynchronous streams of events.*

*Provides a set of operators that filter, combine and transform those streams.*

#### Compare Rx to LINQ?

*LINQ is a pull-based API*

*Rx works with push-based sources of data*

*IObservable<T> is Rx equivalent of IEnumerable<T>*

#### What are the core interfaces of RX?

*IObservable<T> and IOObserver<T>*

#### Why is it unlikely we will ever explicitly implement IOObserver<T>?

*Because the Rx library provides a set of extension methods that take actions and internally create instances of a special IOObserver<T> implementation that invokes the provided delegates when the IObservable<T> calls OnNext, OnError and OnCompleted*

#### What is the single method of IObservable<T>

*IDisposable Subscribe(IOObserver<T>)*

## Risk and Pricing Solutions

**What are the three methods of IObservable<T>**

*Void OnNext(T)*

*Void OnError(Exception)*

*Void OnCompleted()*

**What is the implicit contract a stream should obey?**

1. *An Observable delivers 0..N items via OnNext followed by either OnError or OnCompleted*
2. *Rx calls cannot interleave from a single source. The source must wait for any methods it invokes to complete before invoking the next one*

**What are the two ways a sequence can be terminated?**

*By calling OnCompleted or OnError*

**What are the advantages of Rx versus events?**

1. *Rx sources are first class objects and can be passed as arguments to methods and stored in fields and properties*
2. *An event source's items are delivered in a well-defined order in the presence of multiple threads*
3. *Well defined mechanism for delivering errors*
4. *Well defined mechanism for notifying the end of the sequence*

**What are the difficulties of implementing IObservable<T>**

*A source needs to play nicely with RX Schedulers and multi-threaded scenarios.*

**What is the preferred way of creating observables?**

*Using Observable.Create*

**What does observable.Create do?**

*We pass it a factory method and it creates and returns an instance of AnonymousObservable*

*When we invoke subscribe on AnonymousObservable it creates an AutoDetachObserver that decorates our observer*

*This autodetachobserver is then passed to our factory method.*

*AutoDetachObserver does not process items once disposed*

## Risk and Pricing Solutions

**What are the advantages of using Observable.Create to create sources?**

*Deals with stopping sending messages to disposed subscribers so you don't have to.*

**How does one define a reactive scheduler?**

*Implement IScheduler*

# Risk and Pricing Solutions

## HOT AND COLD

### **Describe cold observables?**

*Cold observables only produce values at the point an observer subscribes*

*Each subscriber given own set of data*

*Subscriber can never miss data by subscribing late*

### **Describe hot observables?**

*Always producing data irrespective of whether observers are subscribed*

*Observer can miss out on earlier values*

### **Given a single cold observable how can we ensure all observers get the same values?**

*Using Observable.Publish to create a IConnectableObservable*

*Multiple observers can subscribe on the connectable observable*

*Once connect is called on connectable observable the subscription is carried out on original observable and multiplexed out to all observers*

### **How can we create a hot observable from a cold observable?**

*Create connectable observable and connect to it before subscribing any observers*

*Values are then being produced before a subscriber get them*

# Risk and Pricing Solutions

## OPERATORS

### What are the factory methods?

*Observable.Create* – daddy method can be used to create the others

*Observable.Empty* – return empty stream and *OnCompleted*

*Observable.Return* – single value and *OnCompleted*

*Observable.Throw* – simply returns *OnError*

### What are the functional unfolds methods?

*Observable.Generate* – root unfold method can produce the others. Take seed and accumulator function

*Observable.Range* – sequence of consecutive integers

*Observable.Interval* – produce incrementally increasing integers, gap between specified by interval

*Observable.Timespan* – delivers single value after given timespan

*Observable.Timer*

### What are the transition methods?

*Observable.Start(Action)*

*Observable.FromEventPattern*

*Task.ToObservable*

*IEnumerable<T>.ToObservable*

### What are the composition operators?

*Concat* – sequential composition

*Repeat*

*StartsWith* – prepends

*Merge*

*Switch* – publish first stream to result until second stream starts publishing

*CombineLatest*

*Zip* – pairs together values

## Risk and Pricing Solutions

*And/Then/When – pair together values from multiple streams*

## Risk and Pricing Solutions

**Write a delegate based implementation of `IObserver<T>` as mentioned in the previous section**

```
public class DelegateBasedObserver<T> : IObserver<T>
{
    public DelegateBasedObserver(Action<T> nextAction)
    {
        _nextAction = nextAction;
    }

    public DelegateBasedObserver(Action<T> nextAction,
        Action completedAction) : this(nextAction)
    {
        _completedAction = completedAction;
    }

    public DelegateBasedObserver(Action<T> nextAction,
        Action<Exception> exceptAction, Action completedAction) :
        this(nextAction, completedAction)
    {
        _exceptAction = exceptAction;
    }

    public void OnNext(T value)
    {
        _nextAction?.Invoke(value);
    }

    public void OnError(Exception error)
    {
        _exceptAction?.Invoke(error);
    }

    public void OnCompleted()
    {
        _completedAction?.Invoke();
    }

    private readonly Action<T> _nextAction;
    private readonly Action _completedAction;
    private readonly Action<Exception> _exceptAction;
}
```



## Risk and Pricing Solutions

**Write an extension method that uses the created `IObserver<T>` to enable one to subscribe using action delegates. You need only deal with `OnNext` actions and not `OnError` and `OnCompleted`**

```
public static class MyObservable
{
    public static IDisposable Subscribe<T>(this IObservable<T> obs,
        Action<T> action)
    {
        return obs.Subscribe(new DelegateBasedObserver<T>(action));
    }
}
```

*in decorators which block messages to disposed subscriptions*

# Risk and Pricing Solutions

## SEQUENCE COMPOSITION

### SEQUENCE CREATION

**What are 3 kind of method for creating sequences?**

*Factory methods*

*Functional unfolds*

*Transitioning from events, delegates and tasks*

**What is the preferred means of creating observables and why?**

*Observable.Create because it is designed to ensure correct ordering of subscriptions and notifications in multi-threaded scenarios.*

**What are the five factory methods**

*Observable.Create*

*Observable.Empty*

*Observable.Throw*

*Observable.Return*

*Observable.Never*

**Which of these methods can be used to create the other four?**

*Observable.Create*

**Write code that produces the single value “hello world”**

*Observable.Return("Hello World")*

**Write code that produces the integers from one to ten?**

*Observable.Range(1,10)*

**What are the advantages of Observable.Create**

*Lazily evaluated when a client subscribes*

## Risk and Pricing Solutions

**What are 4 basic functional unfold methods provided?**

*Observable.Generate*

*Observable.Range*

*Observable.Interval*

*Observable.Timer*

## HOT AND COLD OBSERVABLES

**Compare and contrast hot and cold observables?**

1. *Hot sources produce values even when there are no observers*
2. *Mouse events, keyboard presses and stock ticks would typical examples of hot observables*
3. *Each subscriber to a cold source gets separate values*