Risk and Pricing Solutions

Reactive Extensions

THIS DOCUMENT COVERS

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);
}
```

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.

Basic Lifecycle

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

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

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

// (4) Observerable 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);
```

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;
}
```

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);
```

Questions - Introduction

What is the Reactive Extensions API?

A framework for managing and co-ordinating asynchronous streams of data events

What is RX ideal for?

Managing infinite streams of data

Why?

The events are delivered as and when they are ready

What does RX provide?

A set of operators for filtering, transforming and combining data streams

What are the benefits of RX?

Data can be buffered, throttled, and delayed as needed.

Compare Rx to LINQ?

Whereas LINQ is a pull-based API, RX is a push-based API

What are the core interfaces of RX?

IObservable<T> and IObserver<T>

What are the LINQ based equivalents of these interfaces?

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

IObserver<T> is the Rx equivalent of IEnumerator<T>

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

RX combines a delegate-based implementation of IObserver with extension methods that enable us to subscribe for updates using delegates and lambdas

What is the single method of IObservable<T>?

IDisposable Subscribe(IObserver<T>)

What are the three method of IObserver<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 form a single source. The source must wait for any methods it invokes to complete before invoking the net 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 sources 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 a returns an instance of AnonymousObservable

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

The AutoDetachObserver is then passed to our factory method.

AutoDetachObserver does not process items once disposed

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

How does one unsubscribe from an observable?

Subscribe returns an instance of IDisposable

Calling dispose on it unsubscribes

Why was it done like this?

Language support via using

Why should one always dispose subscriptions?

The IDisposable an observable returns will not have a finalizer and so if you never dispose it you can create memory leaks

What is the exception?

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

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

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>());
    }
```

Explicit Subscription

Listing 7 Explicit Subscription scheduling

```
// 1. Log out the calling thread
"ExplicitMultiThreadedSubsciption".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

```
ExplicitMultiThreadedSubsciption - 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
```

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
```

Questions - Scheduling

What is the core interface we must implement to create a custom scheduler in Rx?

IScheduler

What are the schedulers that come with RX?

EventLoopScheduler Allows user to specify a specific thread for the scheduler

Scheduler.Default Scheduler work on platforms default scheduler (ThreadPool?)

DispatcherScheduler Actions marshalled to Dispatchers BeginInvoke

Scheduler.Immediate Single Threaded scheduler which executes immediately

Scheduler.Current Single Threaded using a message queue

How can we use difference schedulers?

Use the ObserveOn, SubscribeOn and overloads of other methods

What is the difference between Scheduler.Current and Scheduler.Immediate?

Both execute on the current thread. Current places the action on a queue and it executes once the current action is complete. Immediate executes it in line in the current action

What is Scheduler.Default?

Platform default scheduler. Probably the thread pool

How does one scheduler on the UI thread?

DispatcherScheduler

How does one specify a specific thread ?

EventLoopScheduler

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
```

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 IObserver instances that have been unsubscribed by using the static `Observable.Create` method.

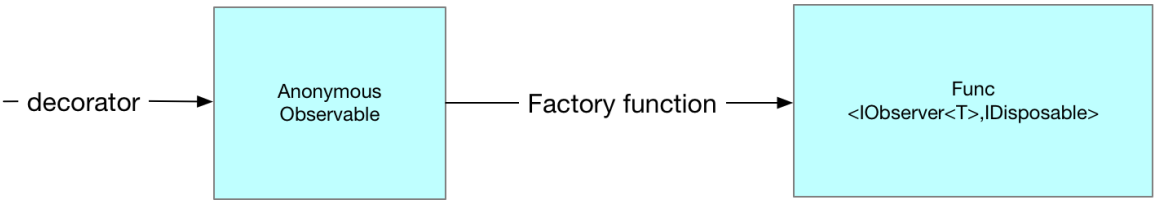
```
MyObservable<int> observable = new MyObservable<int>();
MyObserver<int> observer = new MyObserver<int>();

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
```

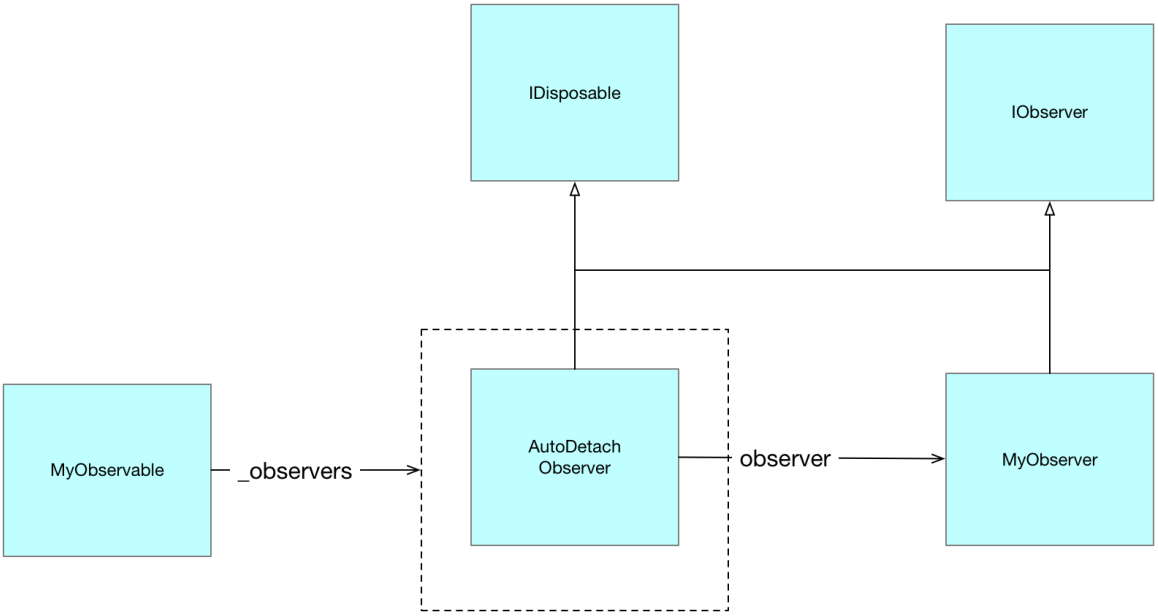
What happens here is that when we call Observable.Create it creates an instance of a framework type called `AnonymousObservable` which stores a reference to the factory method we pass into it.

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 is 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 implement IDisposable. It is the AutoDetachObserver which is returned to the client when it invokes subscribe on the instance of Anonymous Observer. This enables the AutoDetachObserver to stop delivering updates to the MyObserver after dispose has been invoked on it.

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

Questions – Implementing Observable

What is implementing IObservable<T> tricky?

It has to handle disposals and work correctly in multithreaded scenarios introduced by different schedulers

What is the recommended way of creating Observables?

Using the Observable.Create factory method

What does this method do?

Creates an instance of AnonymousObservable whose subscribe method delegates to the given factory method. AnonymousObservable wraps any observers in AutoDetachObserver enabling it to avoid deliverable events to disposed subscriptions.

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 util 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
```

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 Observale Output

```
A 0
B 0
```

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 IObserver
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
```

Questions – Hot and Cold Observables

What are the defining features of a cold observable?

Values only produced at the point of subscription

Each subscriber gets their own set of values

Subscriber can never miss out on data by subscribing late

What are the defining feature of a hot observable?

Always producing values irrespective of whether there are any subscribers

Subscribers can miss values by subscribing late

Given a cold observable how can one create a new observable such that all subscribers get the same value?

Create a connectable observable by calling Publish method on the original

Subscribe multiple observers

Call connect on the connectable observable

Given a cold observable how can one create a hot observable

Create a connectable observable by calling Publish method on the original

Call connect on the connectable observable

Subscribe subscribers

Creating Sequences

Sequences can be created in three ways

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

Factory Methods

TABLE 1 FACTORY METHOD LIST

Empty<T>	Create an Observable<T> that delivers no values and call OnCompleted()
Return<T>(T obj)	Create an Observable<T> that delivers a single item of type T and calls OnCompleted()
Throw<T>(Exception)	Create an Observable<T> that calls OnError with the provided exception and then calls OnCompleted()
Create	Various overloads to create Observable sequences. Has logic to deal well with multi-threaded scheduling and disposed subscriptions

OBSERVABLE.CREATE

Contains various overloads to create observable sequences that behave well in multi-threaded schedulers.

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

```
1
2
3
OnCompleted()
```

OBSERVABLE.EMPTY

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

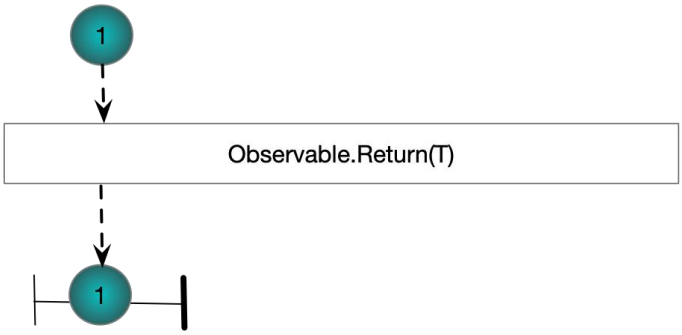
```
Observable
 .Empty<int>()
 .Subscribe(e => {},() => WriteLine("OnCompleted()"));
```

```
OnCompleted()
```

OBSERVABLE.RETURN<T>

Returns a stream consisting of a single value.

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



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

TABLE 2 FUNCTIONAL UNFOLDS LIST

Interval(TimeSpan period)	Delivers successive integers. The elapsed time between each element is defined by period
Timer(TimeSpan dueTime)	Delivers a single integer after dueTime
Timer(TimeSpan dueTime, TimeSpan period)	Delivers the first integer after dueTime and successive integers after each period
Range(int start, int count)	Return an observable range
Generate	Many overloads for generating sequences

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

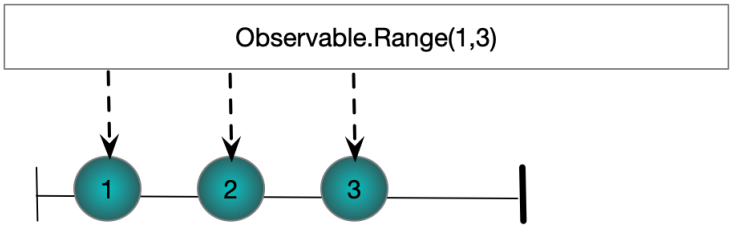
Supports the creation of more complex sequences of event.

```
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"));
```



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

TABLE 3 TRANSITIONING METHODS

Start	Delivers a single Unit after the delegate completes. If the delegate is an action delivers a IObservable<Unit> and if the delegate is Func<T> returns IObservable<T>
FromEvent	Transitions from a standard .NET event to an observable. On observable subscription the event handler add is called and on disposal the remove is called
Task.ToObservable extension	Transitions from a task
IEnumerable<T>.ToObservable	Transition from enumerable

We can use the methods of Rx to create Observables form 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"));
```

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.TOOBSERVABLE

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"));
```

Questions – Creating Sequences

What are the three categories of creational methods?

Factory methods

Functional unfolds

Transitioning from other entities (delegates, tasks, events)

What are the factory methods?

Empty<T>	Create an Observable<T> that delivers no values and call OnCompleted()
Return<T>(T obj)	Create an Observable<T> that delivers a single item of type T and calls OnCompleted()
Throw<T>(Exception)	Create an Observable<T> that calls OnError with the provided exception and then calls OnCompleted()
Create	Various overloads to create Observable sequences. Has logic to deal well with multi-threaded scheduling and disposed subscriptions

Which of the previous methods can be used to generate the other three?

Observable.Create

What are the functional unfolds?

Interval(TimeSpan period)	Delivers successive integers. The elapsed time between each element is defined by period
Timer(TimeSpan dueTime)	Delivers a single integer after dueTime
Timer(TimeSpan dueTime, TimeSpan period)	Delivers the first integer after dueTime and successive integers after each period
Range(int start, int count)	Return an observable range
Generate	Many overloads for generating sequences

What are the means of transitioning from other APIs?

Start	Delivers a single Unit after the delegate completes. If the delegate is an action delivers a IObservable<Unit> and if the delegate is Func<T> returns IObservable<T>
FromEvent	Transitions from a standard .NET event to an observable. On observable subscription the event handler add is called and on disposal the remove is called
Task.ToObservable extension	Transitions from a task
IEnumerable<T>.ToObservable	Transition from enumerable

What should one consider before using IEnumerable<T>.ToObservable?

The blocking pull nature of IEnumerable does not always fit well with the asynchronous push nature of IObservable.

Consider passing IObservable<IEnumerable<T>>

Reducing Sequences

TABLE 4 REDUCTION METHODS

Where	Apply a filter function T=>bool
Distinct	Only propagate values not seen yet in the sequence
DistinctUntilChanged	Filter out a value if it is the same as the previous value
IgnoreElements	Filter out all elements
Skip	Skip the first n elements
Take	Take the fitst n elements
SkipWhile	Skip while a condition is true
TakeWhile	Take while a condition is true
SkipUntil	Skip until a value is produced be another specified observable
TakeUntil	Take until a value is produced by another specied observable

Questions – Reducing Sequences

What are the methods for reducing a sequence?

Where	Apply a filter function T=>bool
Distinct	Only propagate values not seen yet in the sequence
DistinctUntilChanged	Filter out a value if it is the same as the previous value
IgnoreElements	Filter out all elements
Skip	Skip the first n elements
Take	Take the fitst n elements
SkipWhile	Skip while a condition is true
TakeWhile	Take while a condition is true
SkipUntil	Skip until a value is produced be another specified observable
TakeUntil	Take until a value is produced by another specied observable

What overloads are provided for Distinct?

T=>TKey Key Selector Function

IEqualityComparer<T>

What is the difference between Distinct and DistinctUntilChanged?

Distinct filters out values that have been seen anywhere previously in the seq

DistinctUntilChanged filters out values that are the same as the previous element

What does IgnoreElements do?

Only delivers OnCompleted or OnError. No OnNext is ever delivered

Examining Sequences

TABLE 5 SEQUENCE EXAMINATION METHODS

Any()	Returns IObservable<bool> indicating if the sequence is not empty
Any(T=>bool)	Returns IObservable<bool> indicating if any element in the sequence conforms to the predicate
All(T=>bool)	Returns IObservable<bool> indicating if all elements in the sequence conform to the predicate
Contains(T)	Returns IObservable<bool> indicating if the secquence contains the specified value
DefaultIfEmpty	Returns IObservable<T> with a single value of default(T) if the sequence is empty
ElementAt	Returns IObservable<T> with nth value in the sequence
SequenceEqual	Return IObservable<bool> with single value indicating if the two sequences are equal

Questions – Examining Sequences

What are the methods for examining a sequence?

Any()	Returns IObservable<bool> indicating if the sequence is not empty
Any(T=>bool)	Returns IObservable<bool> indicating if any element in the sequence conforms to the predicate
All(T=>bool)	Returns IObservable<bool> indicating if all elements in the sequence conform to the predicate
Contains(T)	Returns IObservable<bool> indicating if the secquence contains the specified value
DefaultIfEmpty	Returns IObservable<T> with a single value of default(T) if the sequence is empty
ElementAt	Returns IObservable<T> with nth value in the sequence
SequenceEqual	Return IObservable<bool> with single value indicating if the two sequences are equal

Aggregating Sequences

TABLE 6 AGGREGATION TO SINGLE VALUE SEQUENCE

Count()	Returns IObservable<int> with single value
Min()	Returns IObservable<T> with single value
Max()	Returns IObservable<T> with single value
Average()	Returns IObservable<double> with the arithmetic mean

TABLE 7 AGGREGATION TO SINGLE VALUE

First	Return first element as T
FirstOrDefault	If sequence is empty return default(T) or we can provide our own default
Last	Return the last element
LastOrDefault	If sequence is empty return default(T) or we can provide our own default

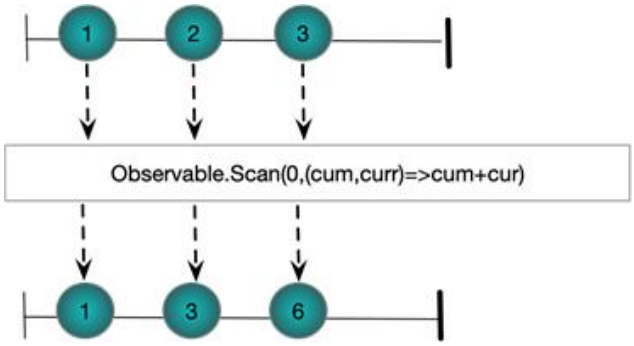
TABLE 8 AGGREGATION TO MULTI-VALUED SEQUENCE

Scan	Aggregates to a running sequence of accumulated values

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"));
```



Questions – Aggregating Sequences

Which functions aggregate to a single value observable?

Count()	Returns IObservable<int> with single value
Min()	Returns IObservable<T> with single value
Max()	Returns IObservable<T> with single value
Average()	Returns IObservable<double> with the arithmetic mean

What are the functional folds/catamorphisms?

First	Return first element as T
FirstOrDefault	If sequence is empty return default(T) or we can provide our own default
Last	Return the last element
LastOrDefault	If sequence is empty return default(T) or we can provide our own default

What happens if the input sequences call OnError?

They throw an exception

Why do we need to be careful with the catamorphisms?

They are blocking and can introduce deadlocks

What is the difference between scan and aggregate?

Aggregate produces a sequence with a single value

Scan produces a sequence with one element per input sequence element

Partitioning Sequences

TABLE 9 PARTITIONING OPERATORS

MinBy	Returns Observable<List<T>> of a single list of all elements with the minimum key value. Only provides a value when the input sequence completes
MaxBy	Returns Observable<List<T>> of a single list of all elements with the minimum key value. Only provides a value when the output sequence completes
GroupBy	Returns an observable sequence of observables. Each subsequence provides all values that match a given key

Questions – Partitioning Sequences

What are the partitioning operators?

MinBy	Returns Observable<List<T>> of a single list of all elements with the minimum key value. Only provides a value when the input sequence completes
MaxBy	Returns Observable<List<T>> of a single list of all elements with the minimum key value. Only provides a value when the output sequence completes
GroupBy	Returns an observable sequence of observables. Each subsequence provides all values that match a given key

What is the purpose of the GroupBy operator?

Partition data from one source so it can be shared to multiple sources.

Transforming Sequences

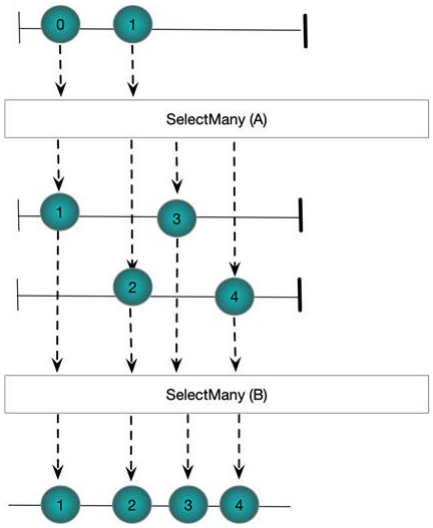
TABLE 10 TRANSFORMATION METHODS

Select	Simple transformation from IObservable<TSource> to IObservable<TResult>
Cast<T>	Take IObservable<Object> and given IObservable<T> If anything cannot be cast then OnError is called
OfType<T>	Take IObservable<Object> and given IObservable<T> If anything cannot be cast then it is excluded
Timestamp	Wraps each element in a struct with the element value and the current timestamp
TimeInterval	Wraps each element in a struct with the element value and the elapsed time since the previous element
Materialize	Transform IObservable<T> into a sequence of IObservable<Notification<T>> With debug notification
Dematerialize	Take a materialzed sequence and dematerialize it
SelectMany	Map one sequence to a sequence of sequences and flatten
ToArray, ToList,ToDict,ToLookup	Produce an observable with one value containing all the values from the input sequence. Only delivered once input sequence completes

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 – Transforming Sequences

What are the transformation functions?

Select	Simple transformation from IObservable<TSource> to IObservable<TResult>
Cast<T>	Take IObservable<Object> and given IObservable<T> If anything cannot be cast then OnError is called
OfType<T>	Take IObservable<Object> and given IObservable<T> If anything cannot be cast then it is excluded
Timestamp	Wraps each element in a struct with the element value and the current timestamp
TimeInterval	Wraps each element in a struct with the element value and the elapsed time since the previous element
Materialize	Transform IObservable<T> into a sequence of IObservable<Notification<T>> With debug notification
Dematerialize	Take a materialzed sequence and dematerialize it
SelectMany	Map one sequence to a sequence of sequences and flatten

Composing Sequences

TABLE 11 SEQUENTIAL COMPOSITION OPERATORS

Concat	Appends one sequence to another. The first sequence has to complete.
Repeat	Repeate a sequence n times. The sequence must complete
StartsWith	Prepend a value to a sequence.

TABLE 12 CONCURRENT COMPOSITION OPERATORS

Amb	First sequence wins
Merge	Merge values from two sequences
Switch	Propagate items from first stream to result stream until second stream starts publishing. At this point only items from the second stream are provided

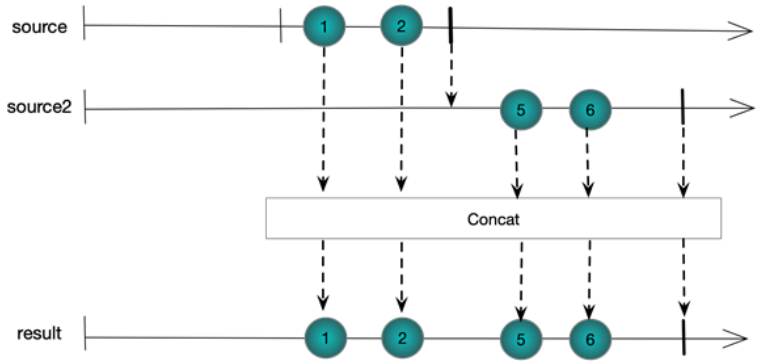
TABLE 13 PAIRWISE COMPOSITION OPERATORS

CombineLatest	Pairs latest value from first stream with latest value from second stream as each value is produced from either stream
Zip	Pairs together values from two streams
AndThenWhen	Pairs together values from multiple streams

Sequential Composition

OBSERVABLE.CONCAT(IOBSERVABLE<TSOURCE>)

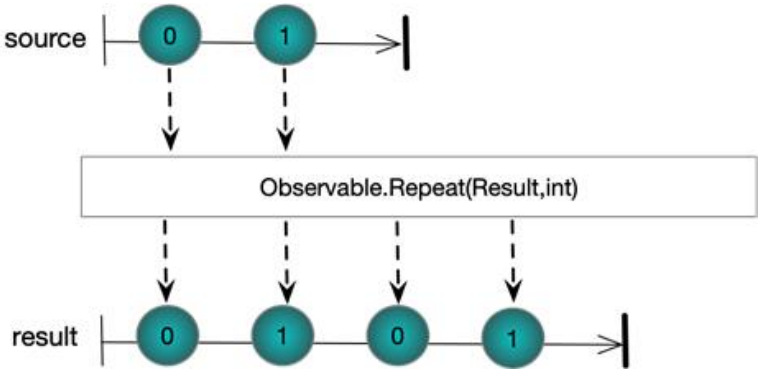
```
Observable
  .Range(0, 2)
  .Concat(Observable.Range(5, 2))
  .Subscribe(WriteLine);
```



OBSERVABLE.REPEAT(INT REPS)

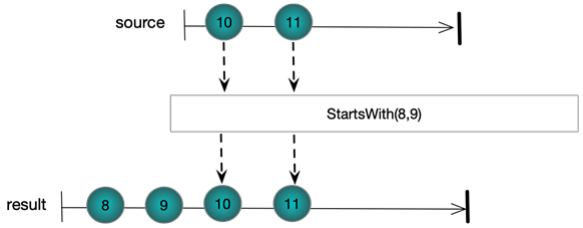
```
Observable
 .Range(0, 2)
```

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



OBSERVABLE.STARTSWITH

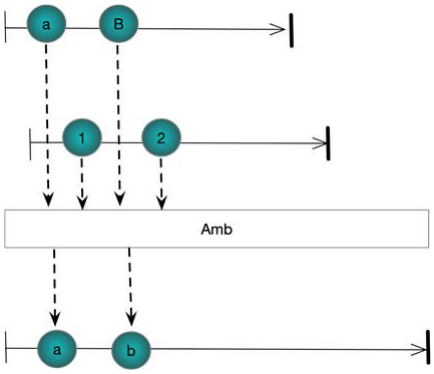
```
Observable
  .Range(10, 2)
  .StartWith(8, 9)
  .Subscribe(WriteLine);
```



Concurrent Composition

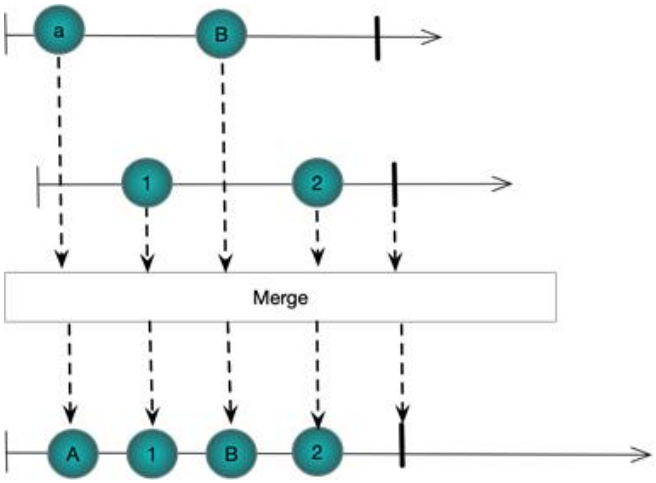
OBSERVABLE.AMB

```
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");
```


a
B
1
2
Amb
a
b

OBSERVABLE.MERGE

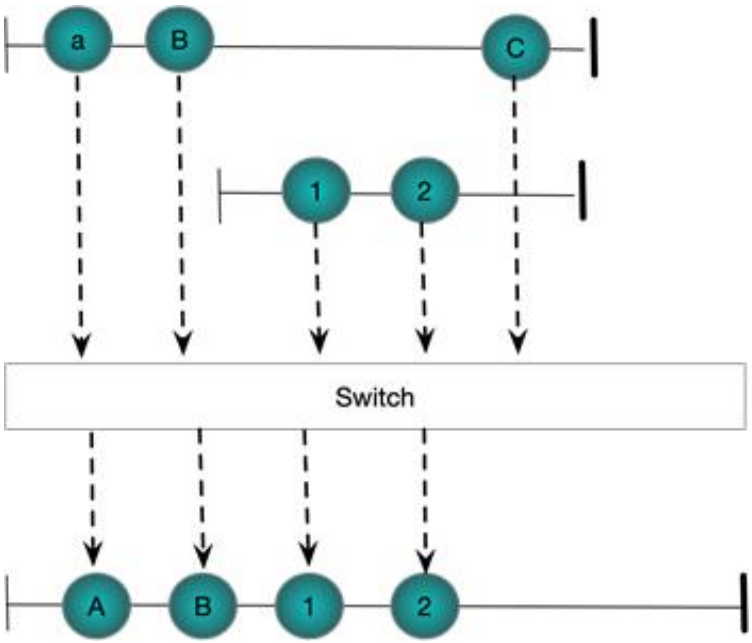
```
Subject<string> a = new Subject<string>();
Subject<string> b = new Subject<string>();

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

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



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");
```



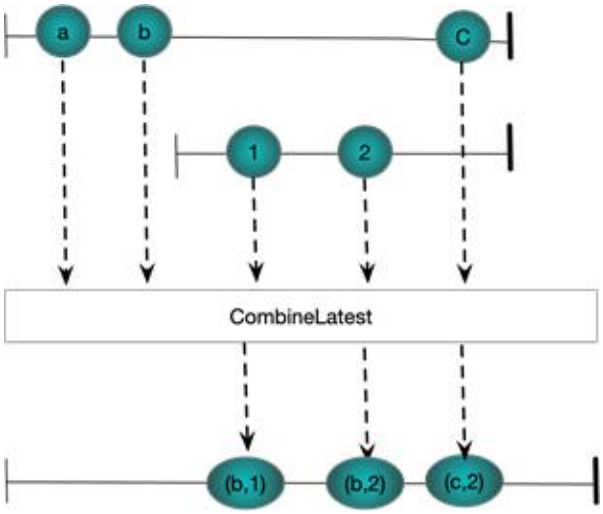
Pairwise Composition

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);
```

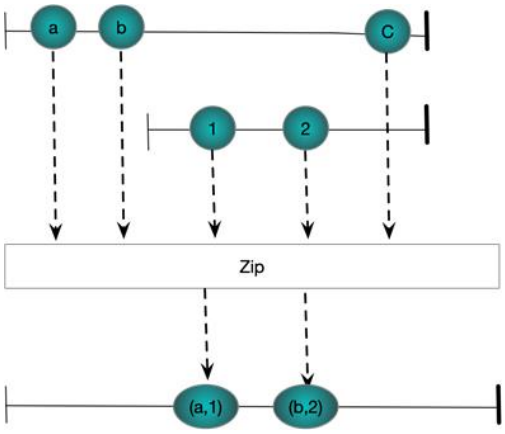
```
a.OnNext("a");
a.OnNext("b");
b.OnNext("1");
b.OnNext("2");
a.OnNext("c");
```



OBSERVABLE.ZIP

Pairs together values from two streams.

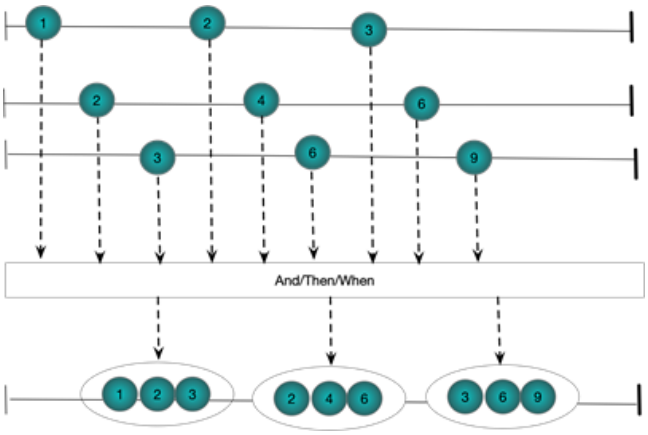
```
Subject<string> a = new Subject<string>();
Subject<string> b = new Subject<string>();

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");
```



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));
```



Questions – Composing Sequences

What are the means of composing sequences?

Sequential

Concurrent

Pairwise

What are the sequential composition operators?

Concat

Repeat

StartsWith

What does Concat do?

Concatenates two sequences

What must happen for Concat to work?

The first sequence most complete

What does Repeat do?

Repeats a sequence n times

What must happen for Repeat to work?

The sequence most complete

What does StartsWith do?

Prepends a value to a sequence.

What are the concurrent composition operators?

Amb

Merge

Switch

What does AMB do?

First sequence to start publishing wins

What does Switch do

Propagates items from one sequence until a second sequence starts publishing and then only publishes from the second sequece

What are the pairwise composition operators?

Zip

CombineLatest

AndThenWhen

What does Zip do?

Pairs together values from two streams

What does CombineLatest do?

Combine values from two streams as and when they become ready

Buffering Operators

Buffer(int count)	Builds perfectly contiguous buffers of size count
Buffer(int count, int skip)	Supports non contiguous buffers. If skip is greater than count we miss out elements. If skip is less than count we have overlapping buffers
Buffer(TimeSpan timespan)	Build perfectly contiguous buffers by timespan
Buffer(TimeSpan timeSpan, TimeSpan timeShift)	If timespan is greater than timeShift we get overlapping buffers. If timestap is less than timeShift we miss out elements
Buffer(Func<IObservable<TC>> closingSelector)	Each time a value is published from closing selector the current buffer is closed and flushed
Buffer(TimeSpan timeSpan, int count)	Buffer is closed when eiher count it reached or timespan elapses

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 IObservable<T> and produce a stream IObservable<IList<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 bugffers or overlapping where some source elements make it into more than once 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

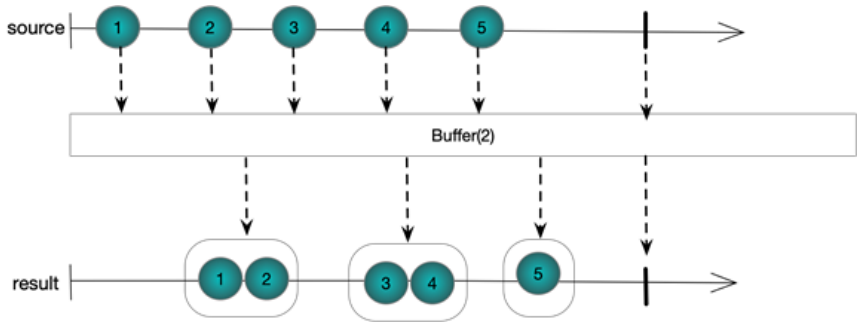
- Source element count
- time interval
- opening and closing signals

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



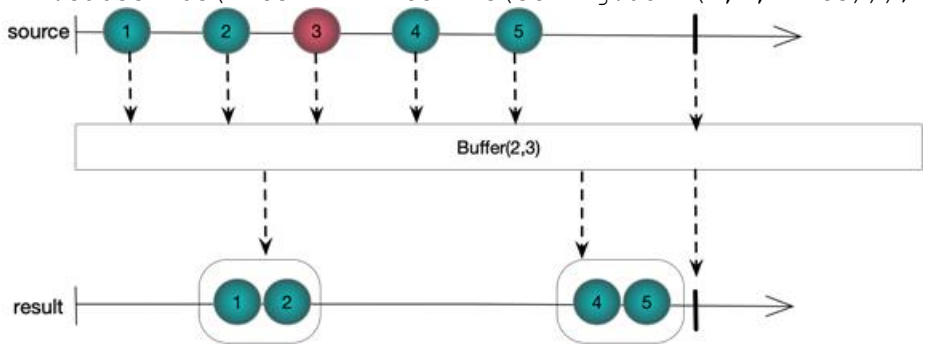
BUFFER(INT COUNT,INT SKIP)

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.

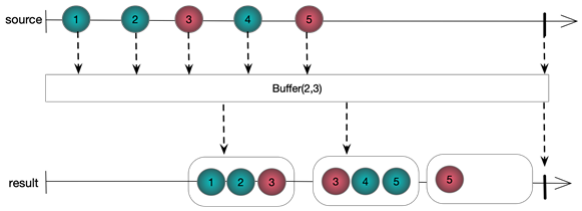
```
Observable.Range(1, 5)
  .Buffer(2, 3)
  .Subscribe(ints => WriteLine(string.Join(",", ints)));
```



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

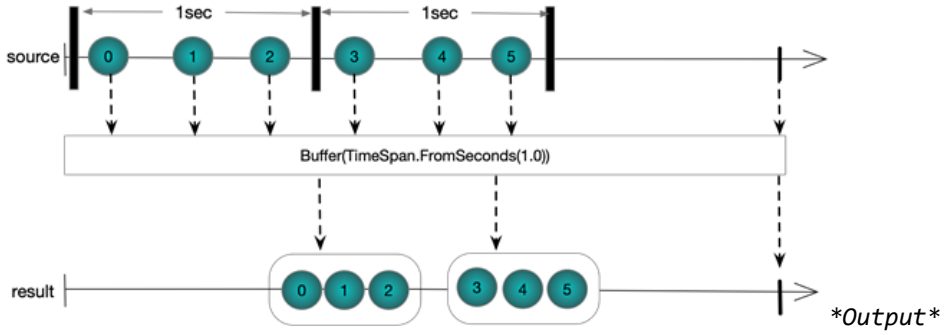
```
Observable.Range(1, 5)
  .Buffer(3, 2)
  .Subscribe(ints => WriteLine(string.Join(",", ints)));
```



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();
```

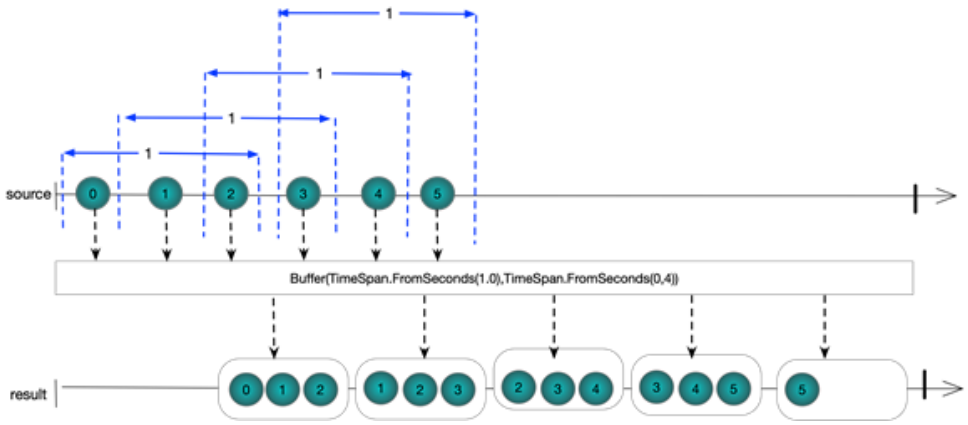


BUFFER(TIMESPAN TIMESPAN, TIMESPAN TIMESHIFT)

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

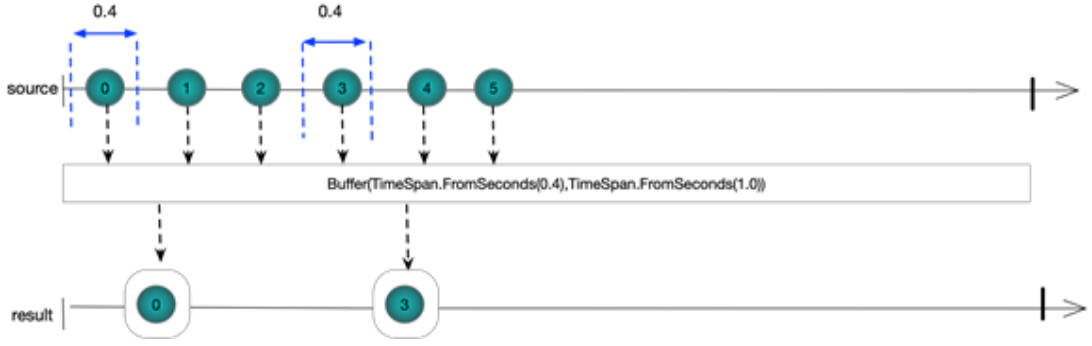
Overlapping Elements

```
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

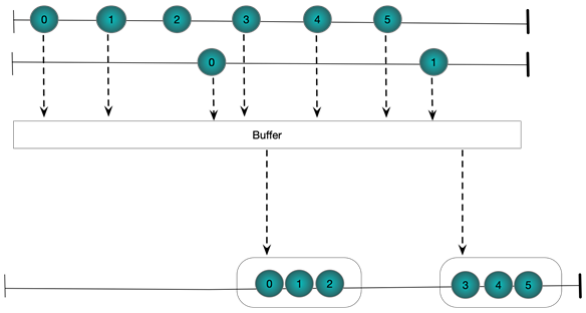
```
EventWaitHandle ewh = new AutoResetEvent(false);
Observable
 .Interval(TimeSpan.FromSeconds(0.3))
 .Take(6)
 .Buffer(TimeSpan.FromSeconds(0.4), TimeSpan.FromSeconds(1.0))
 .Subscribe(ints => WriteLine(string.Join(",", ints)), () => ewh.Set());

ewh.WaitOne();
```



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

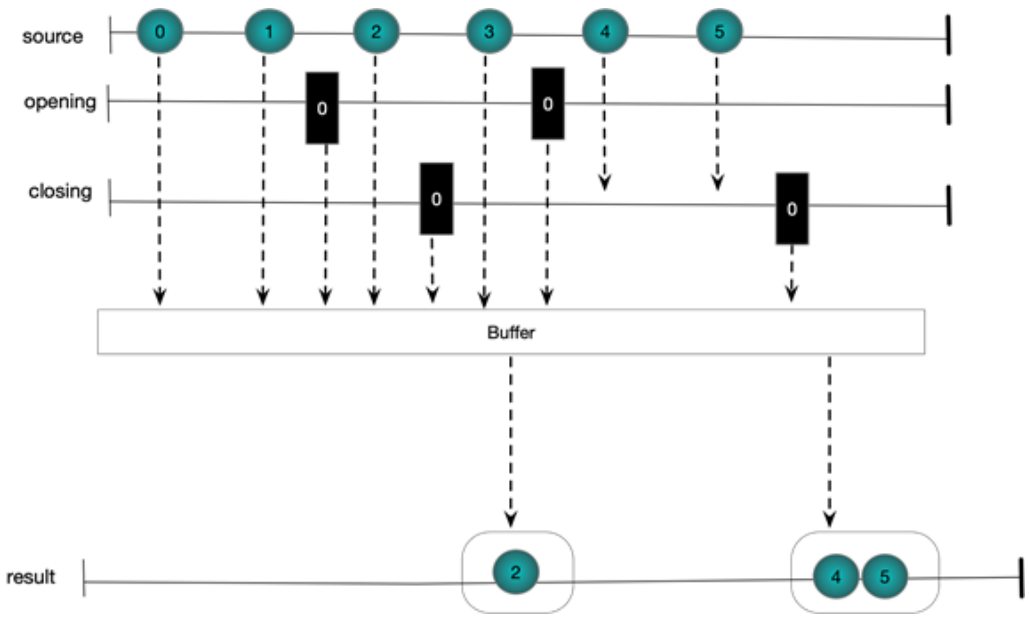
```
EventWaitHandle latch = new AutoResetEvent(false);

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();
```



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.

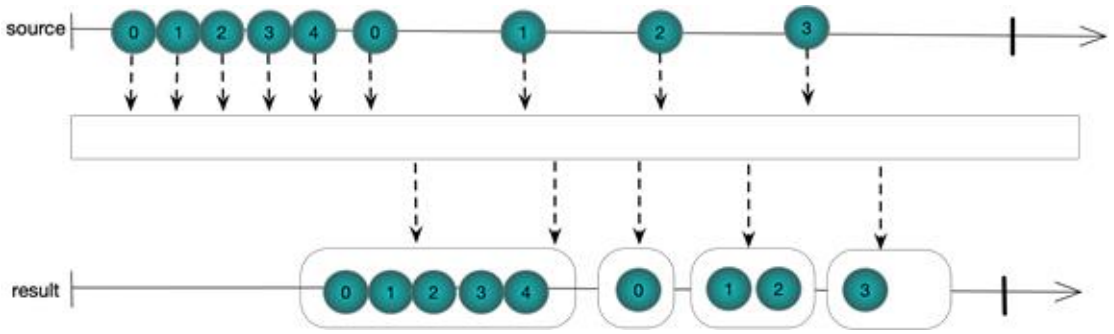
```
EventWaitHandle latch = new AutoResetEvent(false);

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();
```



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();
```



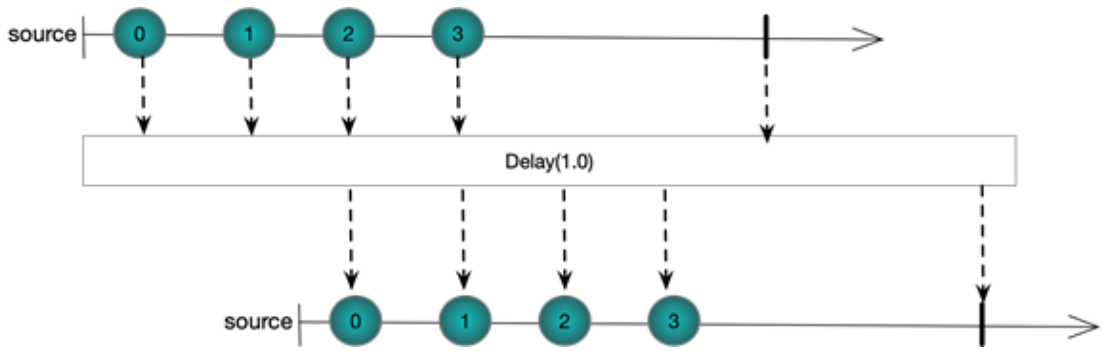
Questions - Buffering

Delay Operators

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
```



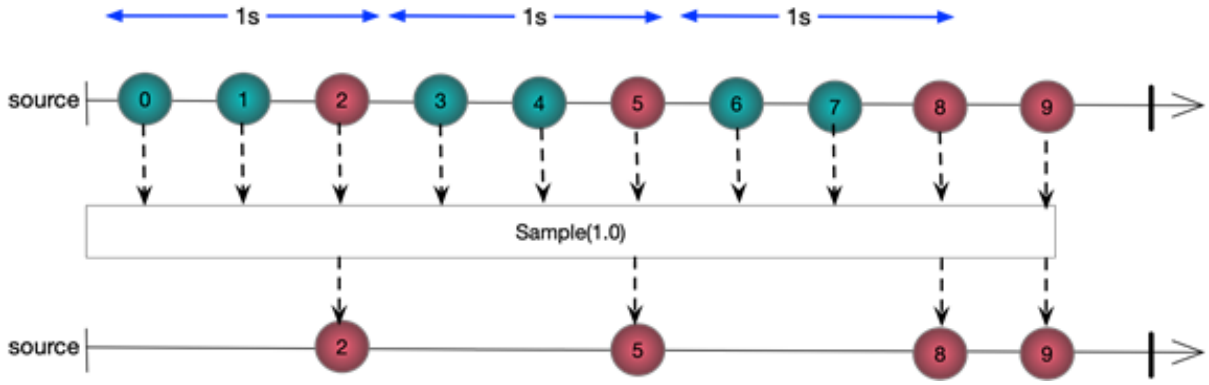
Questions – Delay Operators

Sample and Throttle

SAMPLE (TIMESPAN)

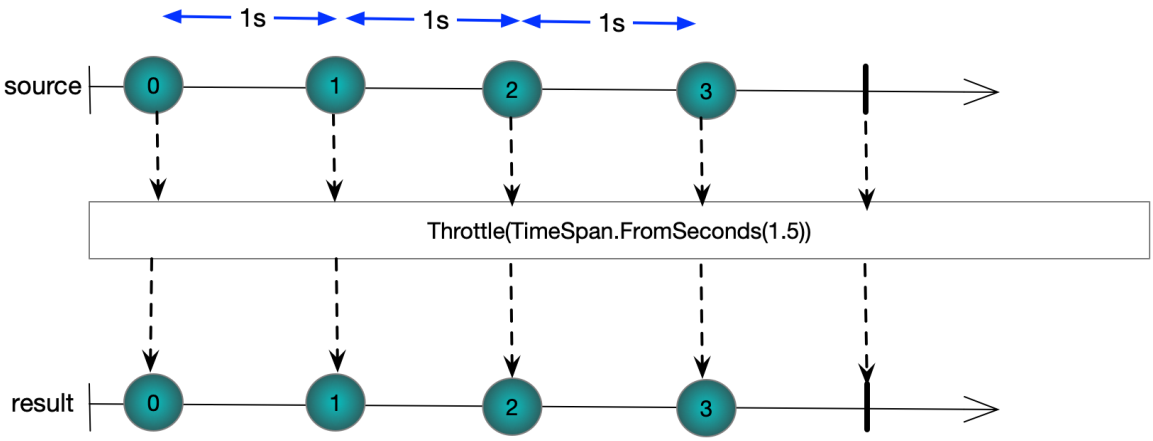
Returns the last element emitted by a source sequence in a buffer interval specified by a TimeSpan

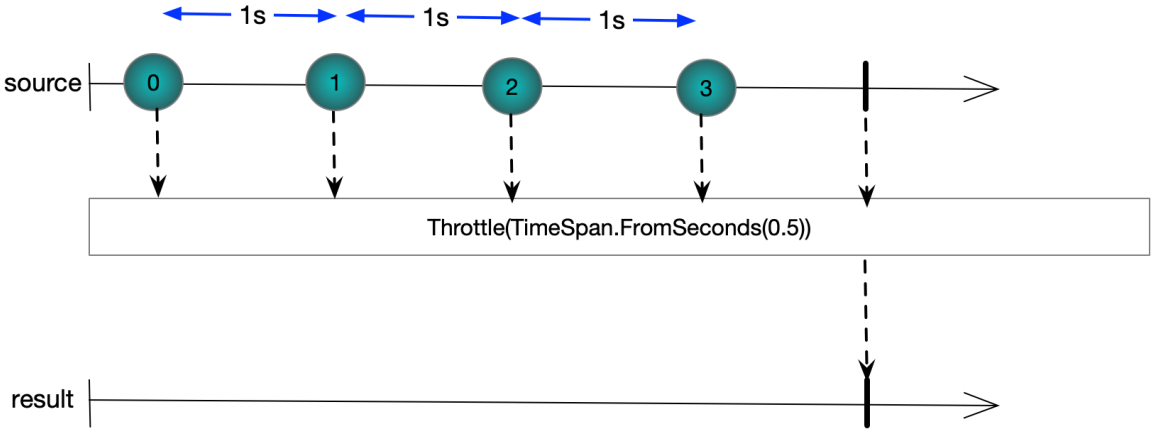
```
DateTime now = DateTime.Now;
EventWaitHandle waitHandle = new AutoResetEvent(false);

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
```

More Examples

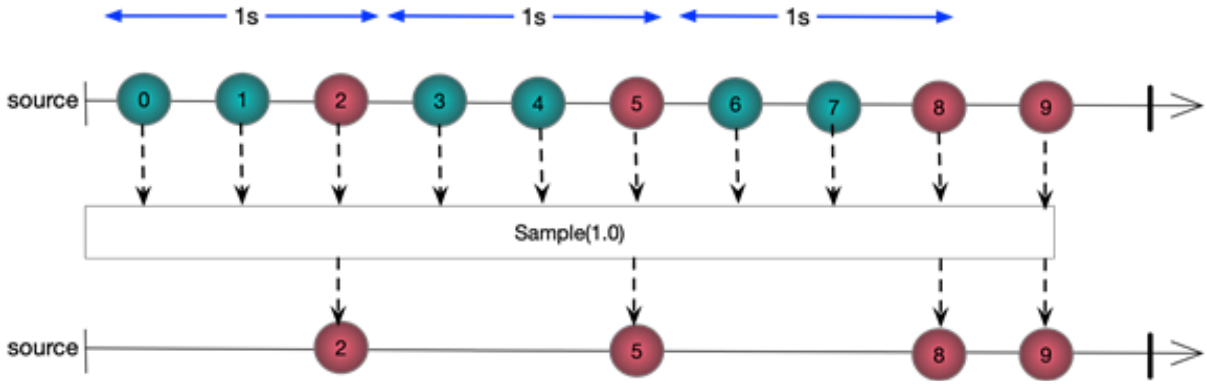




Questions – Sample and Throttle

What does sample do?

Returns the last element emitted by a source sequence in a given interval



What does buffer do?

Returns the last element emitted by a source sequence in a given interval. However, if an element is received inside the window specified by the timespan the window is reset. So if a source produces values in uniform intervals where the interval is less than the buffer length no elements will make it to the destination sequence.

Timeout

Timeout(TimeSpan dueTime)	Completes with OnError if a sequence does has longer than dueTime between elements
Timeout(DateTimeOffset dueTime)	Completes with OnError is sequence does not complete in the given dueTime
Timeout(TimeSpan dueTime,IObservable<T>)	Returns second sequence if first times out

Questions - Timeout

Testing Rx

Subject

Rx provides a type called `Subject<T>` that implements IObserver<T> and IObservable<T> that enables code to publish and subscribe from the same code

```
var s1 = new Subject<int>();
s1.Subscribe(o => WriteLine($"OnNext({o})"),() =>
WriteLine("OnCompleted()"));

s1.OnNext(1);
s1.OnCompleted();
```

OUTPUT

```
OnNext(1)
OnCompleted()
```

ReplaySubject

In addition to the basic subject, RX provides a `ReplaySubject<T>` that caches values so late subscribers still get them

```
var s1 = new ReplaySubject<int>();
s1.OnNext(1);
s1.OnNext(2);

s1.Subscribe(o => WriteLine($"OnNext({o})"),() =>
WriteLine("OnCompleted()"));
s1.OnCompleted();
```

OUTPUT

```
OnNext(1)
OnCompleted()
```

The replay subject can take a buffer size to limit the memory footprint

```
var s1 = new ReplaySubject<int>(2);
s1.OnNext(1);
s1.OnNext(2);
s1.OnNext(3);

s1.Subscribe(o =>
WriteLine($"OnNext({o})"),()=>WriteLine("OnCompleted()"));
s1.OnCompleted();
```

OUTPUT

```
OnNext(2)
OnNext(3)
OnCompleted()
```

BehaviourSubject

`BehaviorSubject<T>` remembers the last value published which it delivers to a late subscriber. Its constructor takes a default value that is delivered to a subscriber subscribes before any values are published. All subscribers get a value unless the sequence is completed.

```
var s1 = new BehaviorSubject<int>(1);

s1.Subscribe(o =>
WriteLine($"OnNext({o})"),()=>WriteLine("OnCompleted()"));
s1.OnCompleted();
```

OUTPUT

```
OnNext(1)
OnCompleted()
```

AsyncSubject

Keeps track of the last published value and deliver it and only it when sequence completes. If sequence does not complete never delivers any values.

```
var s1 = new AsyncSubject<int>();

s1.OnNext(1);
s1.OnNext(2);
s1.OnNext(3);
s1.OnCompleted();
s1.Subscribe(o => WriteLine($"OnNext({o})"),()=>WriteLine("OnCompleted()"));
```

OUTPUT

```
OnNext(3)
OnCompleted()
```

Questions – Testing Rx

What is a Subject<T>?

An object that implements both IObserver<T> and IObservable<T>

Enables client to easily publish to the Observable

What is a ReplaySubject<T>?

Caches values so late subscribers do not miss out

Supports cache size and cache interval to limit memory footprint

What is a BehaviourSubject<T>?

Caches last value published. Take default value

Any subscriber always gets a value unless sequence completes

Compare and Contract ReplaySubject<T>(1) and BehaviourSubject

If no values have ever been published a subscriber to BehaviourSubject gets a default value whereas this is not the case with ReplaySubject(1)

ReplaySubject(1) caches value after completion but BehaviourSubject does not.

If we want to test Rx code we can take advantage of the type `Microsoft.Reactive.Testing.TestScheduler` which provides a set of methods to simulate time moving forward. The following snippet shows how we might make this work.

```
        TestScheduler t = new TestScheduler();
        DoSubscription(t);
        t.Start();
    }

    public void DoSubscription(IScheduler scheduler)
    {
        IObservable<long> observable = Observable
        .Interval(TimeSpan.FromSeconds(1),scheduler)
        .Take(5);

        observable
              .ObserveOn(scheduler)
              .Subscribe(e => logger.Info(e));
    }
```

Note how we pass in an instance of `IScheduler` Typically we would use an interface such as ISchedulerProvider to enable us to inject in different implementations of our view model.

Best Practice

Always favour Observable.Create over writing one own IObservable<T> implementation

Always favour the factory methods that take functions over writing ones own IObserver<T> implementations

Threading should be defined by the subscriber so SubscribeOn and ObserveOm should generally only be seen before a Subscibe call

Subsciptions should be disposed

An OnError handler should be provided

If you return a sequence return empy sequences rather than null

Try not to use blocking operators that break the monad.