Reactive Extensions

Working with streaming data

# Introduction

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);

}

## Implementing IObservable<T>

Listing 2 SimpleObservable<T>

public class SimpleObservable<T> : IObservable<T>

{

➊ public IDisposable Subscribe(IObserver<T> observer)

{

Console.WriteLine($"Subscribe”);

\_observers.Add(observer);

return new ActionDisposable(() =>

{

Console.WriteLine($"Dispose”);

\_observers.Remove(observer);

});

}

➋ 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 ➊ we add the given IObserver to our list. We also define a Publish ➋method (not part of IObservable<T>) which iterates the list and invokes OnNext on each subscriber. 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);

# 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 Figure 4 Custom Scheduler. We can then instruct our code to observe or subscribe on this custom scheduler.

Listing 6 Custom Scheduler



### 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 9Explicit 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 10Output

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

## 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



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.



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.

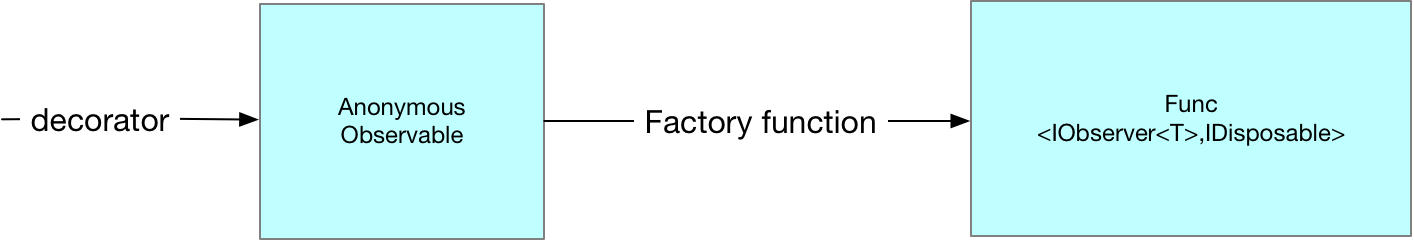


Now the output from running this code is what we want. The first publish causes the observer to write 1 to the console but the second publish after the dispose call is suppressed.

1

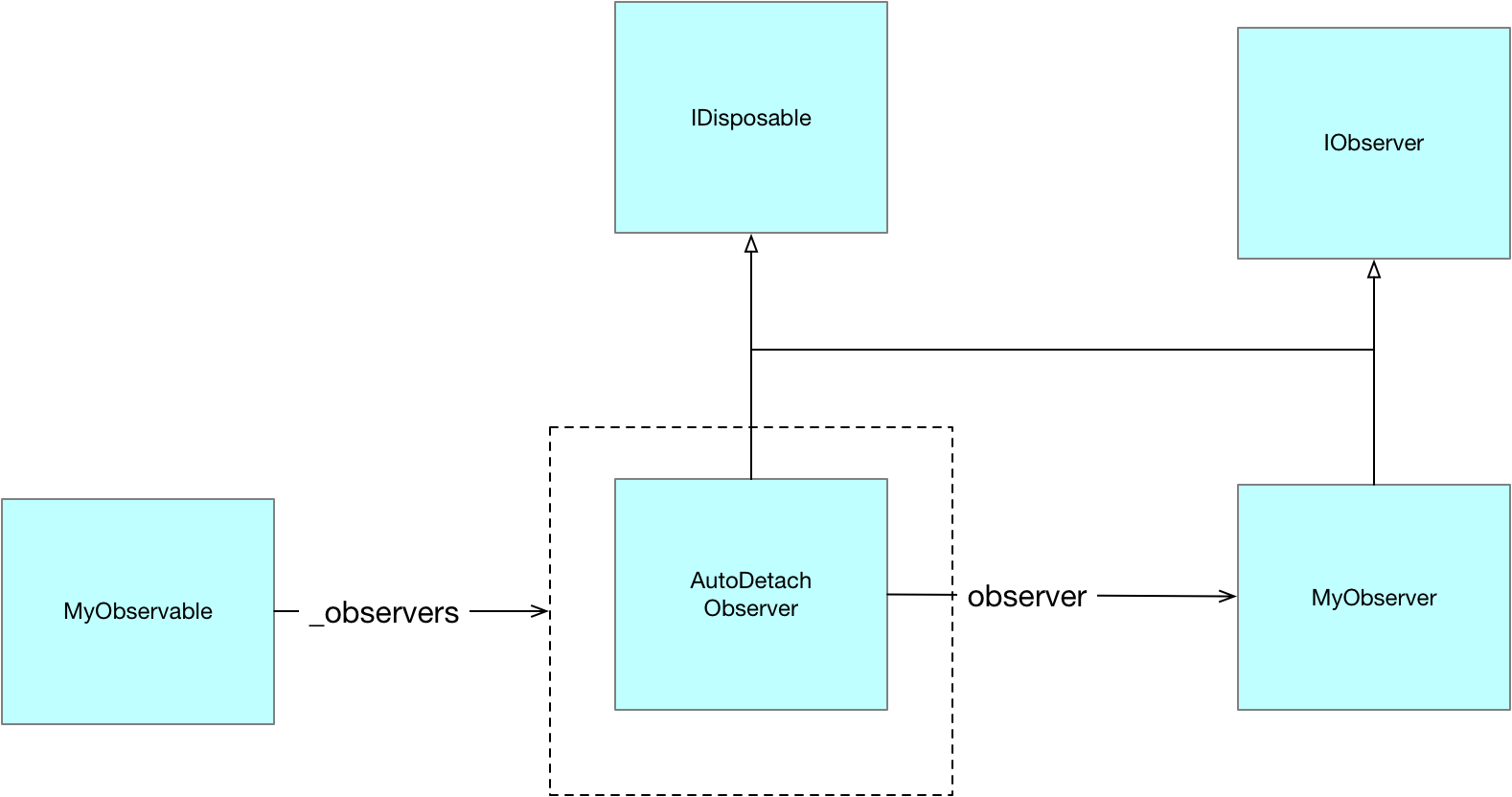
What happens here is that when we call Observable.Create it creates an instance of a framework type called AnonymousObservable which stores a reference to the factory method we pass into it.

Listing 11AnonymousObservable



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 12AutoDetachObserver



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

# 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 13Basic 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 14Basic Cold Observable Output

A 0  
B 1

## Connectable Observable

Listing 15Connectable 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 16Connectable Observale Output

A 0  
B 0

## Hot Observable

Listing 17Hot 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 18Hot 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

# Creating Sequences

Sequences can be created in three ways

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

Factory Methods

1. Observable.Return
2. Observable.Empty
3. Observable.Create
4. Observable. Throw
5. Observable.New

Obervable.Create can be used to write the other 4 methods

Functional Unfolds

1. Observable.Generate
2. Observable.Range
3. Observable.Interval
4. Observable.Timer

Transitioning From other .NET types

1. Delegates
2. Tasks
3. Events

# Composing Sequences

Sequences can be combined using three classes of compositors

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

Sequential composition

1. Concat
2. Repeat
3. StartsWith

Concurrent composition

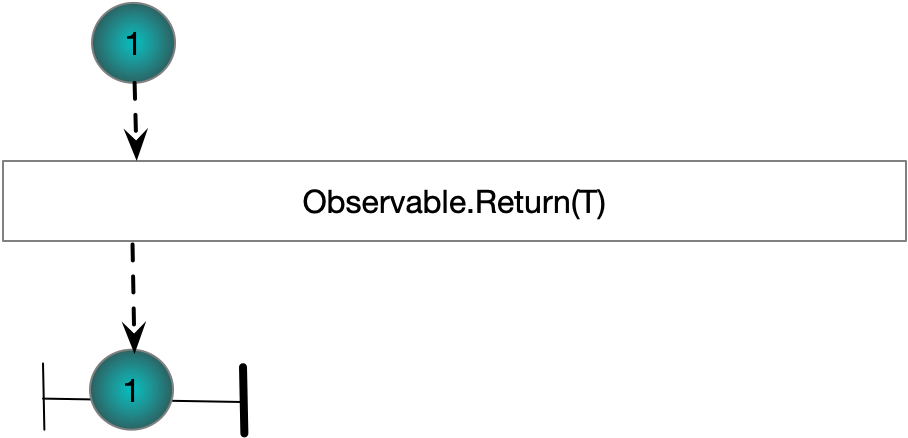
1. Amg
2. Merge
3. Switch

Pairwise composition

1. CombineLatest
2. Zip
3. And Then When

# Operations

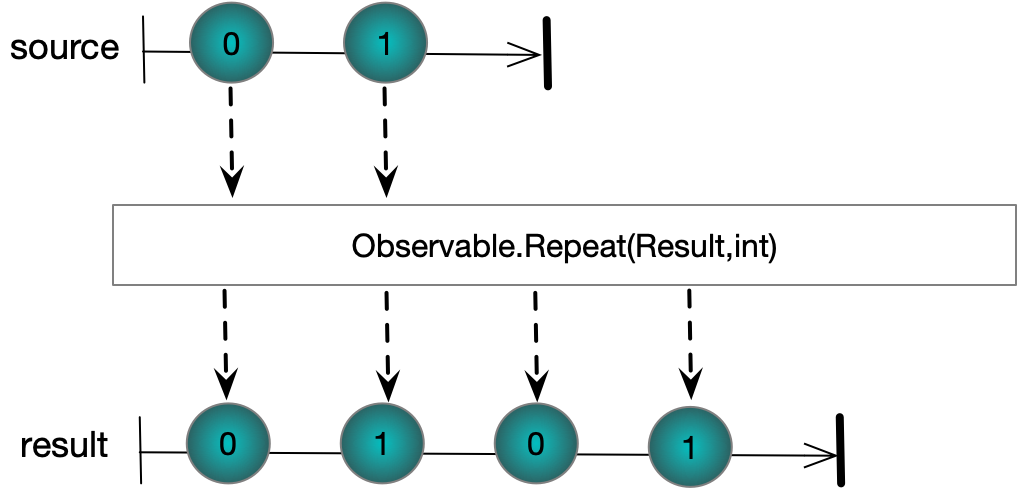
### Obvervable.Return(T)



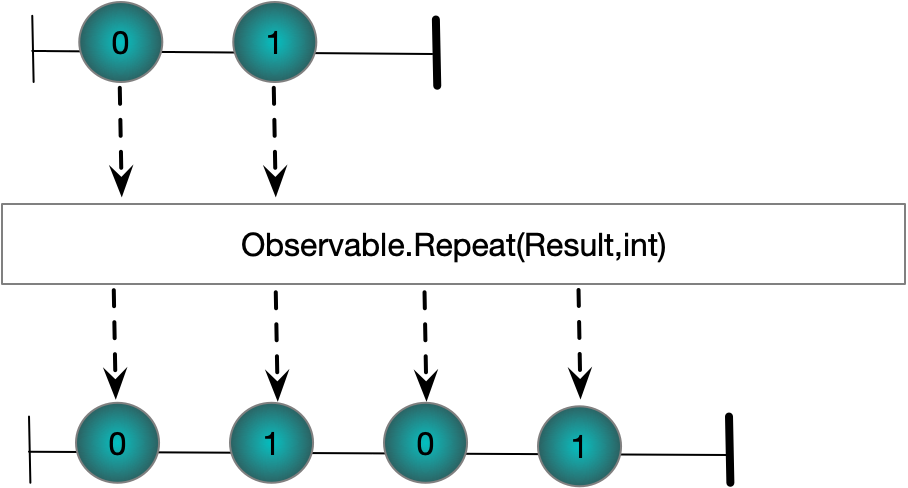
### Obvervable.Range(int,int)

# 

### Obvervable.Repeat(T,int)



### Obvervable.Repeat(Iobservable<T>,int)



### Obvervable.Scan

# 

# Obvervable.SelectMany

# 

# Obvervable.Concat

# 

# Obvervable.StartsWith

# 

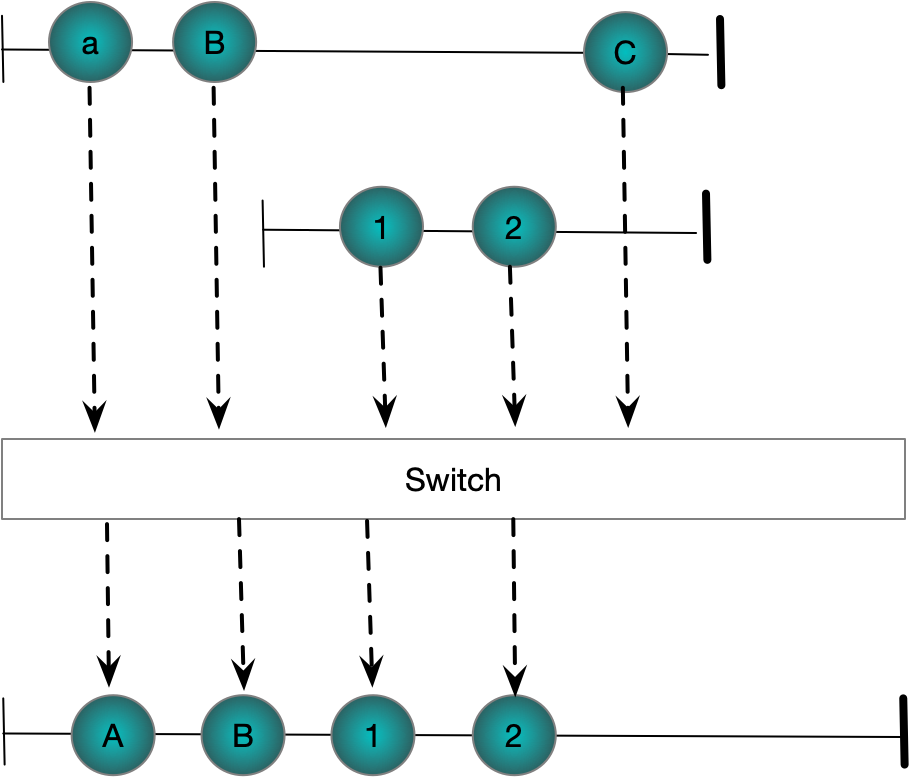
# Obvervable.Amb

# 

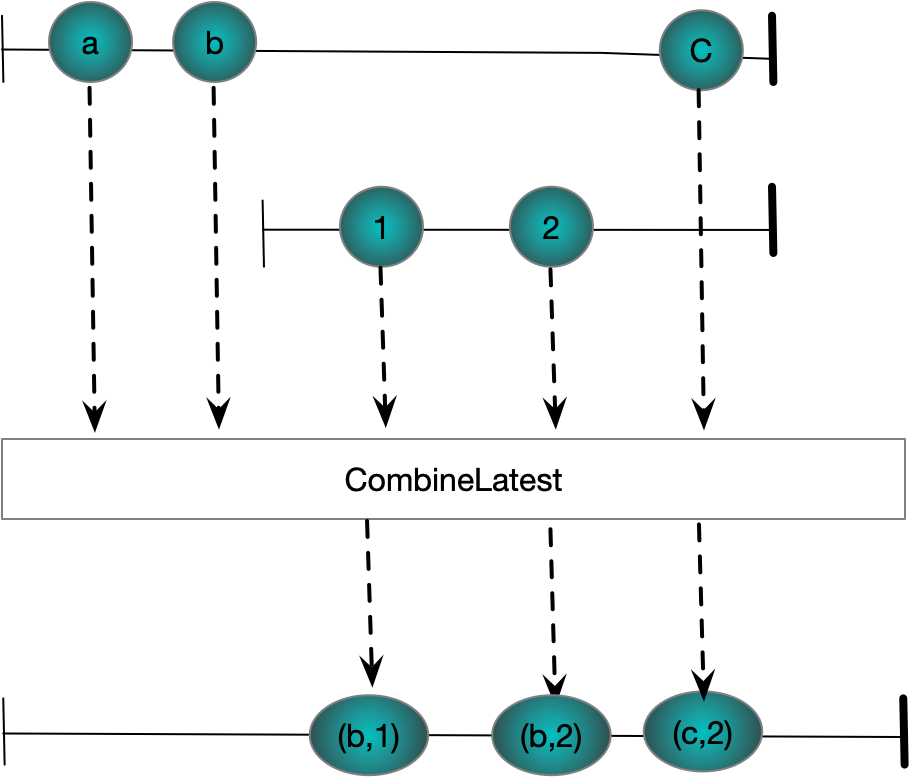
# Observable.Merge

# 

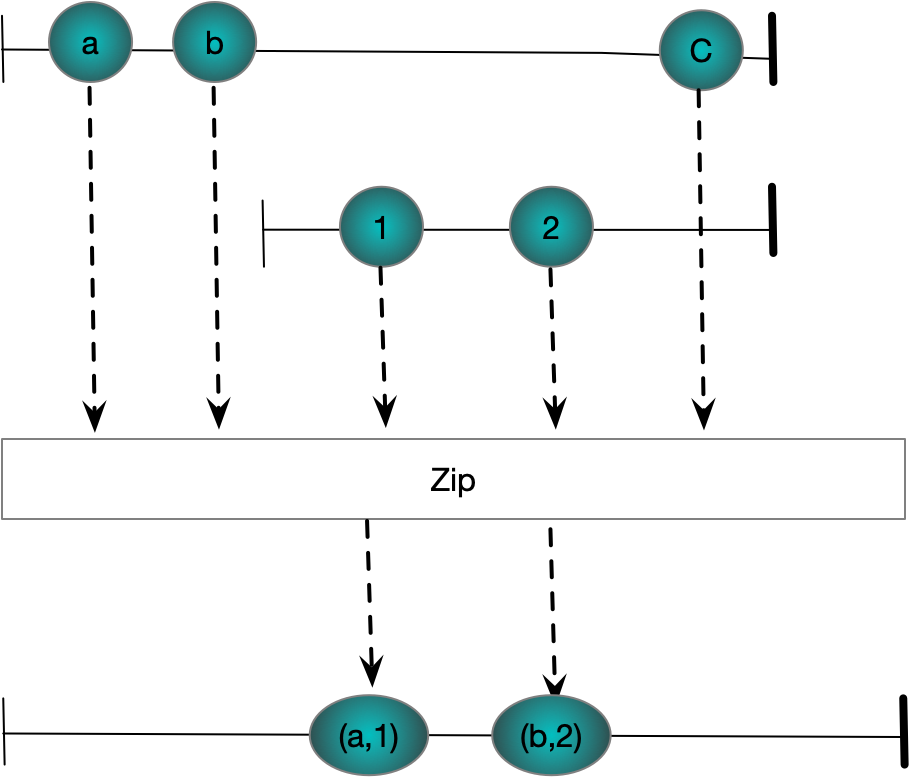
# Observable.Switch



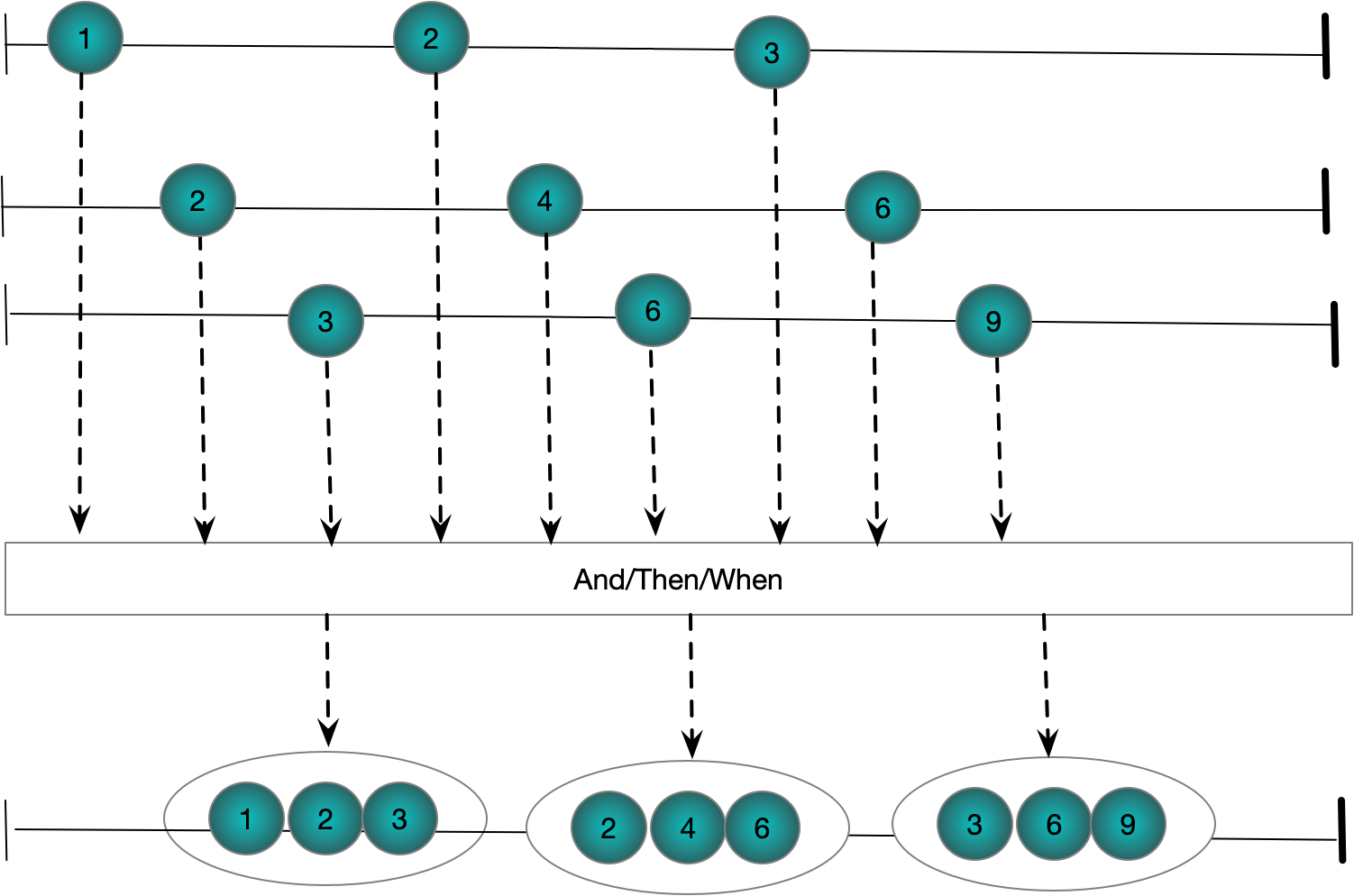
# Observable.CombineLatest



# Observable.Zip



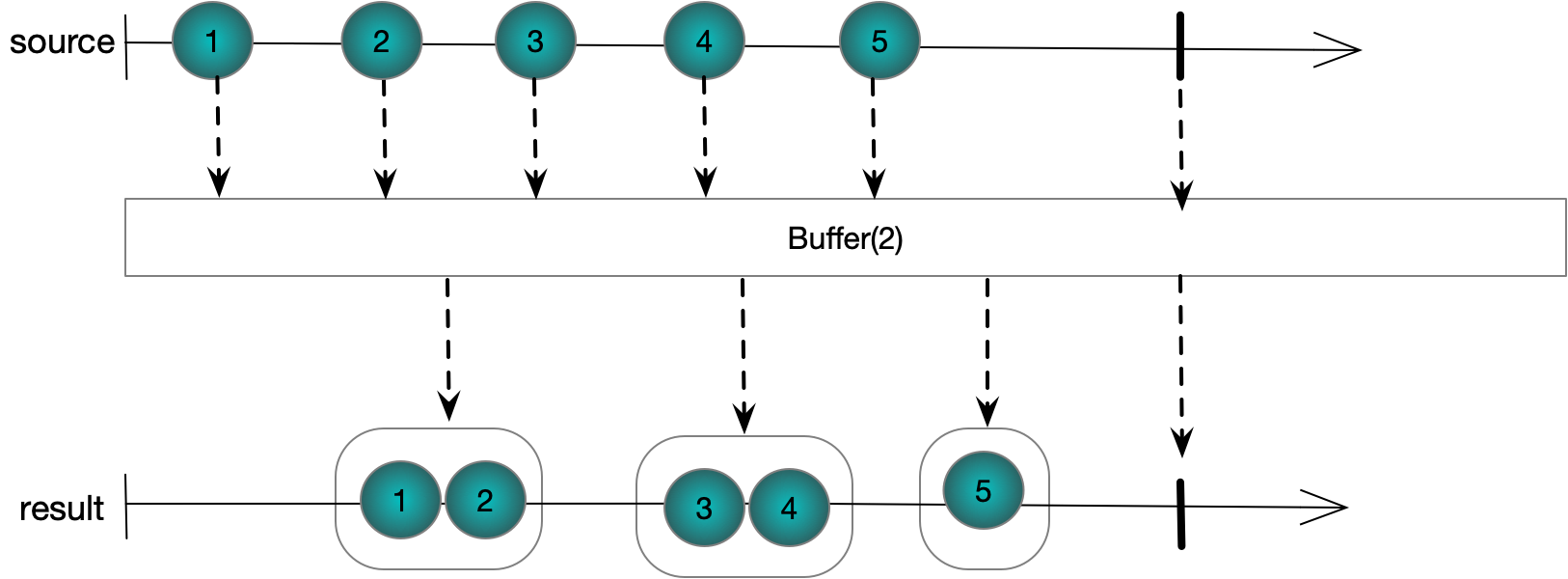
# And/Then/When



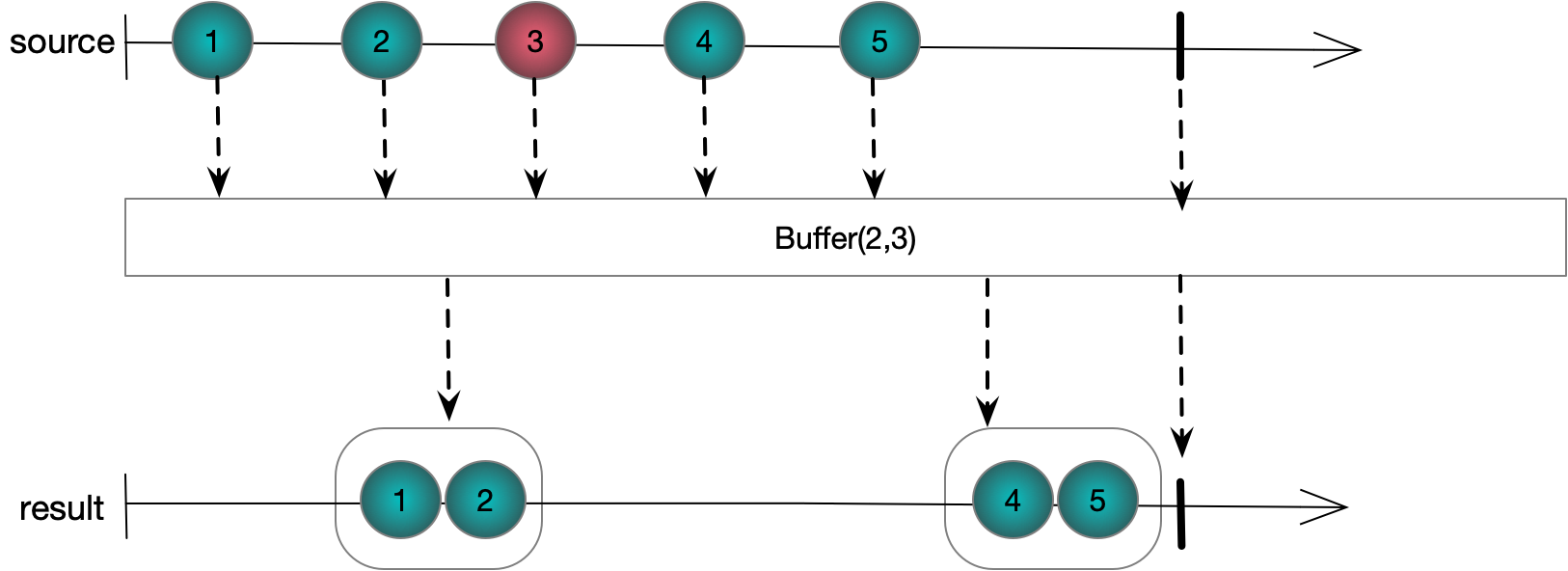
# Buffer

### Buffer(int Count)

Listing 19 Buffer(int)

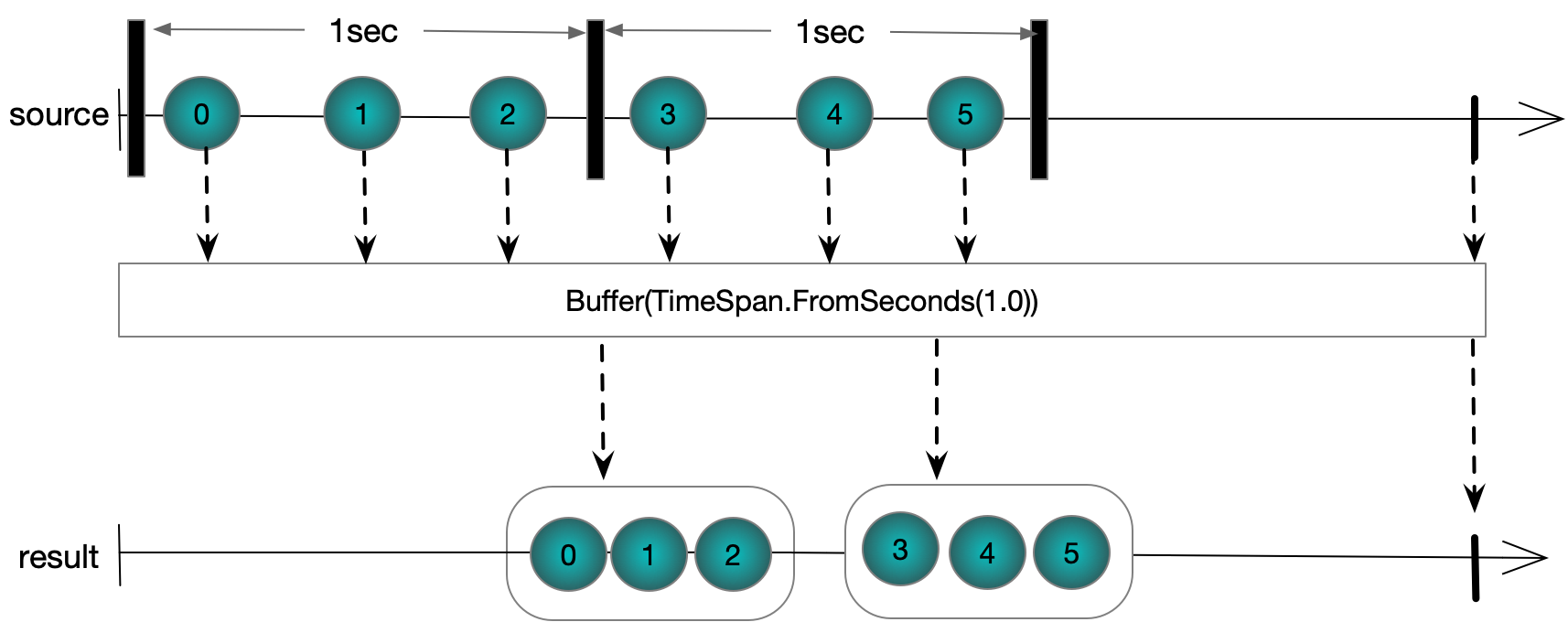


## Buffer(int Count, int Skip) – Skipping

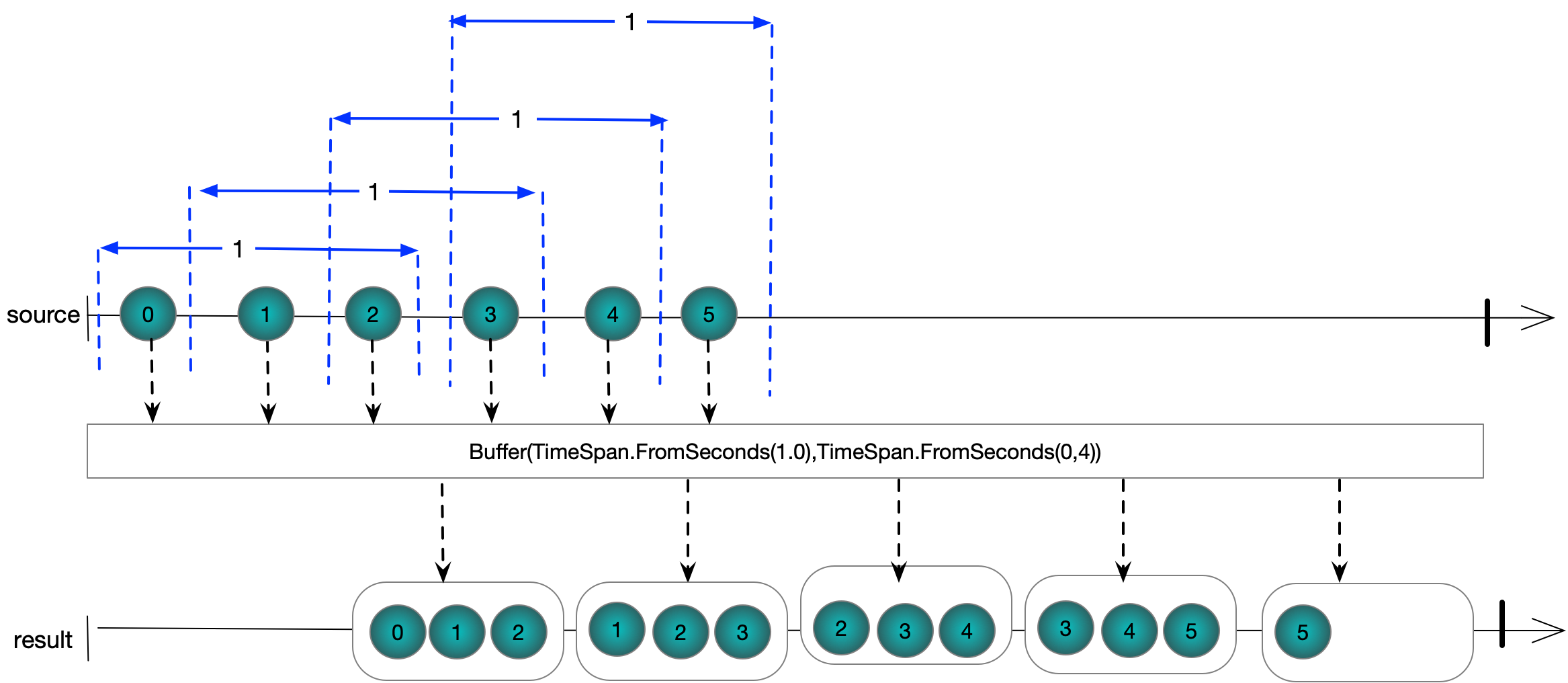


## Buffer(int Count, int Skip) - Overlapping

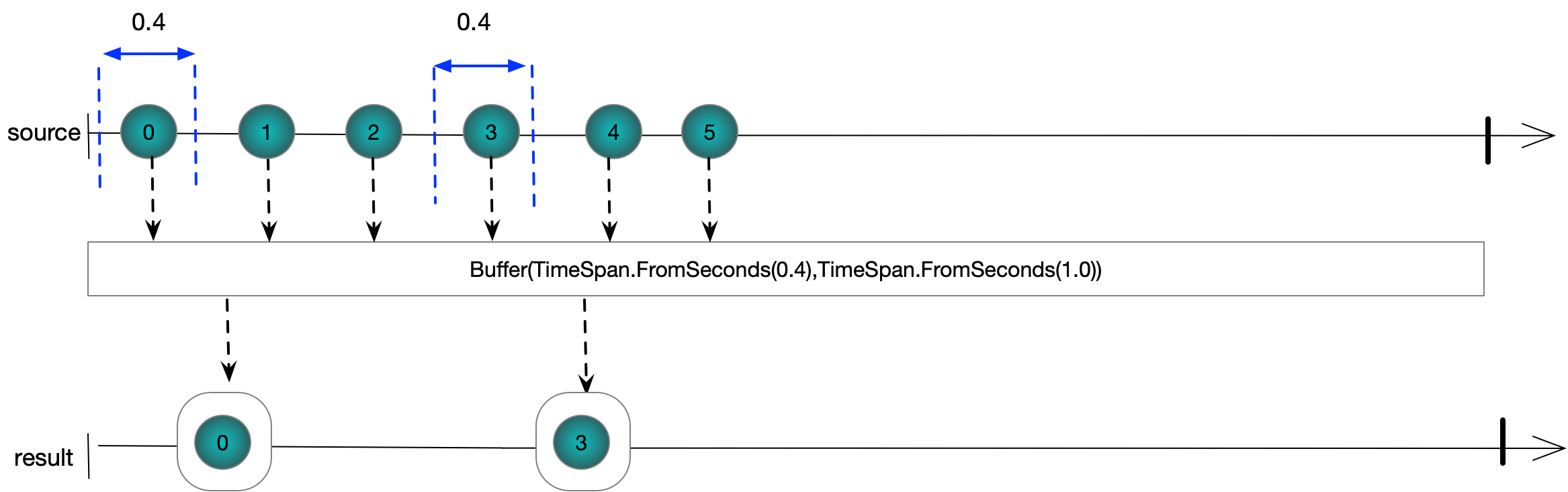
## Buffer(Timespan timeSpan)



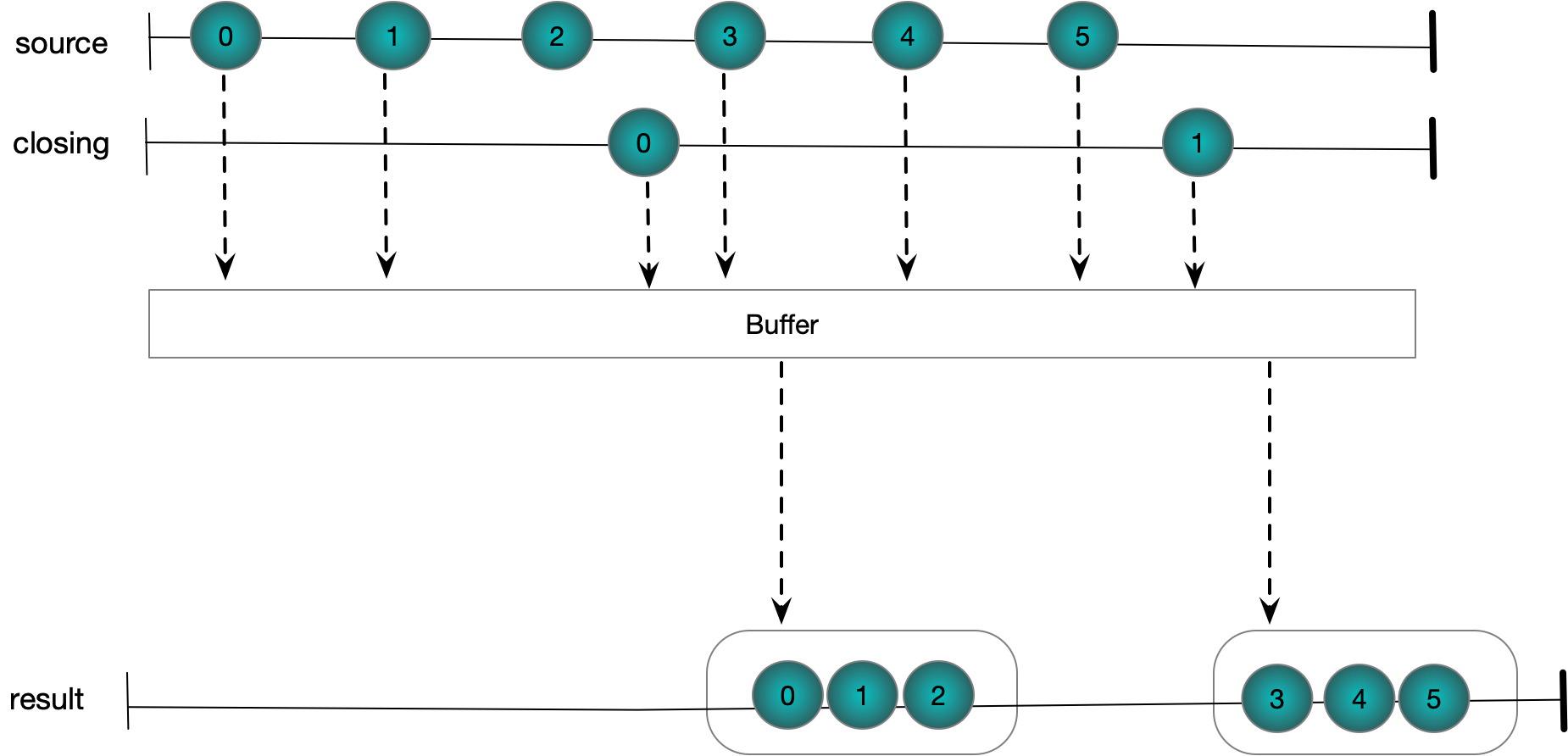
## Buffer(Timespan timeSpan, Timespan timeShift)



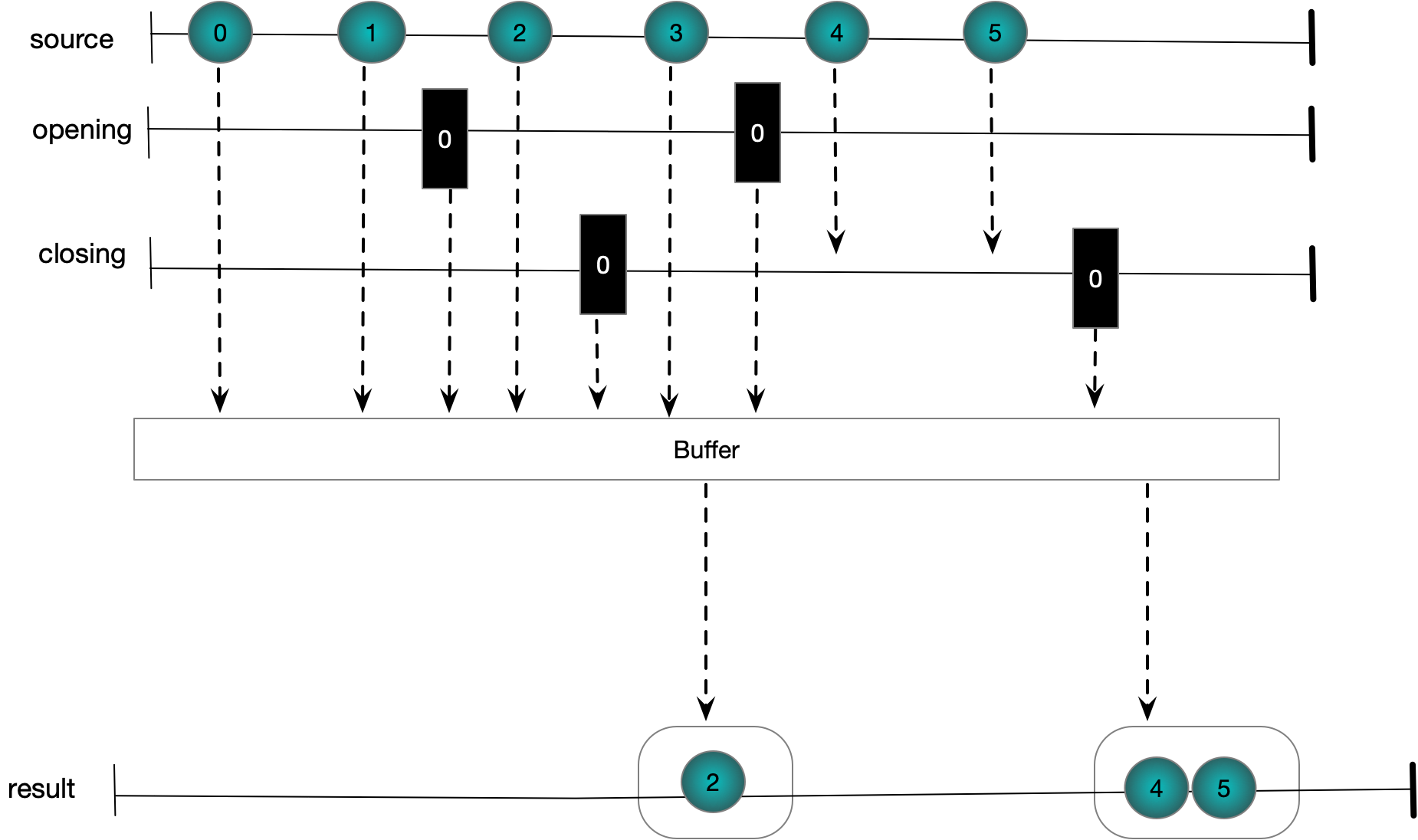
## Buffer(Timespan timeSpan, Timespan timeShift)



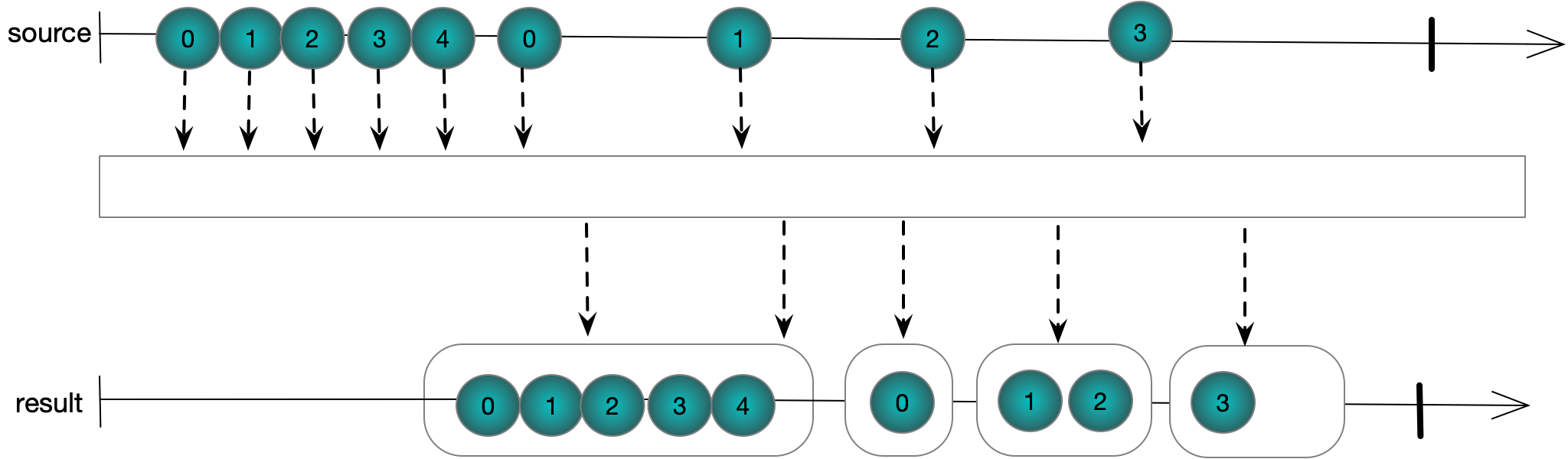
## Buffer(Iobservable closingSignals)



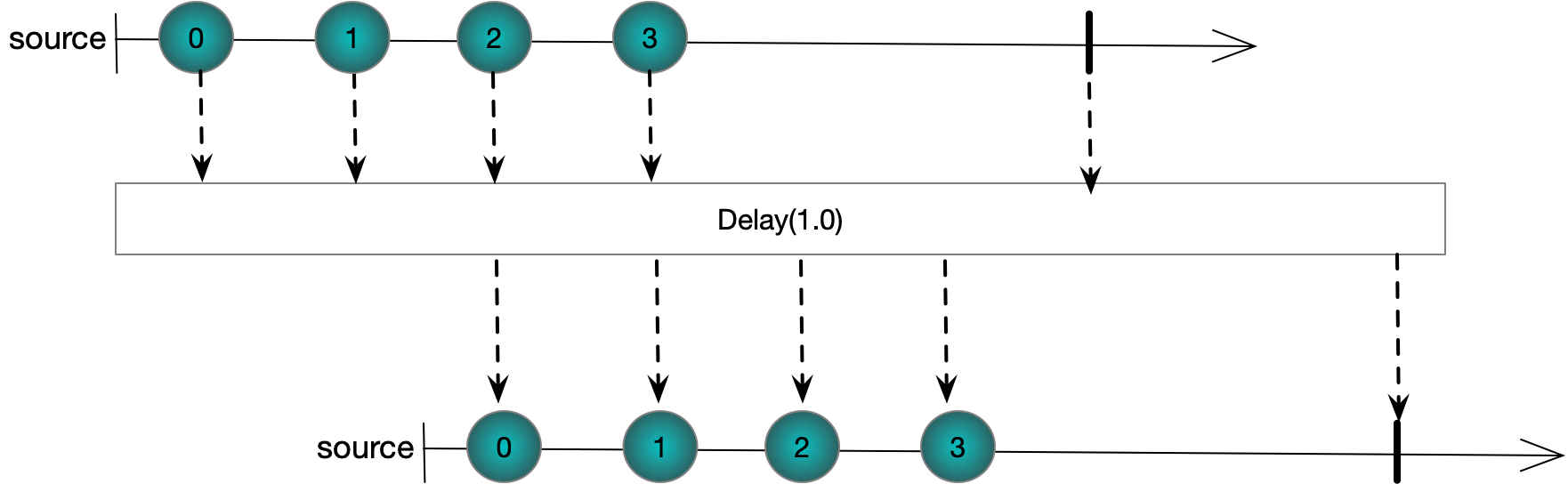
## Buffer(Iobservable openingSignals, Iobservable closingSignals)



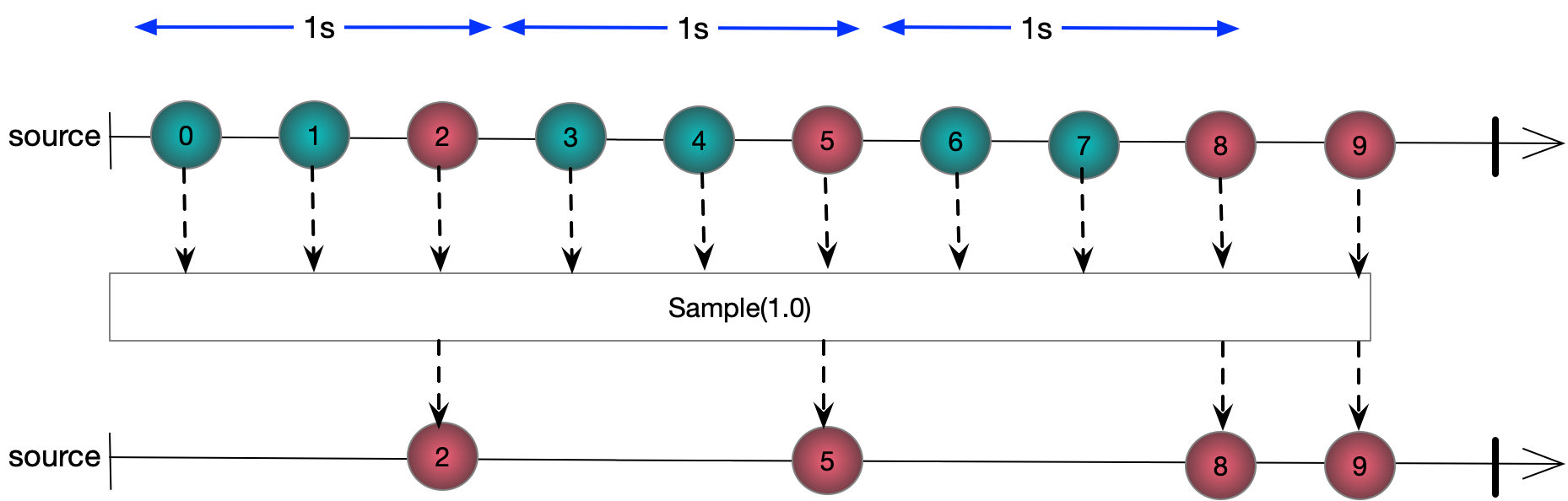
## Buffer(TimeSpan timeSpan, int count)



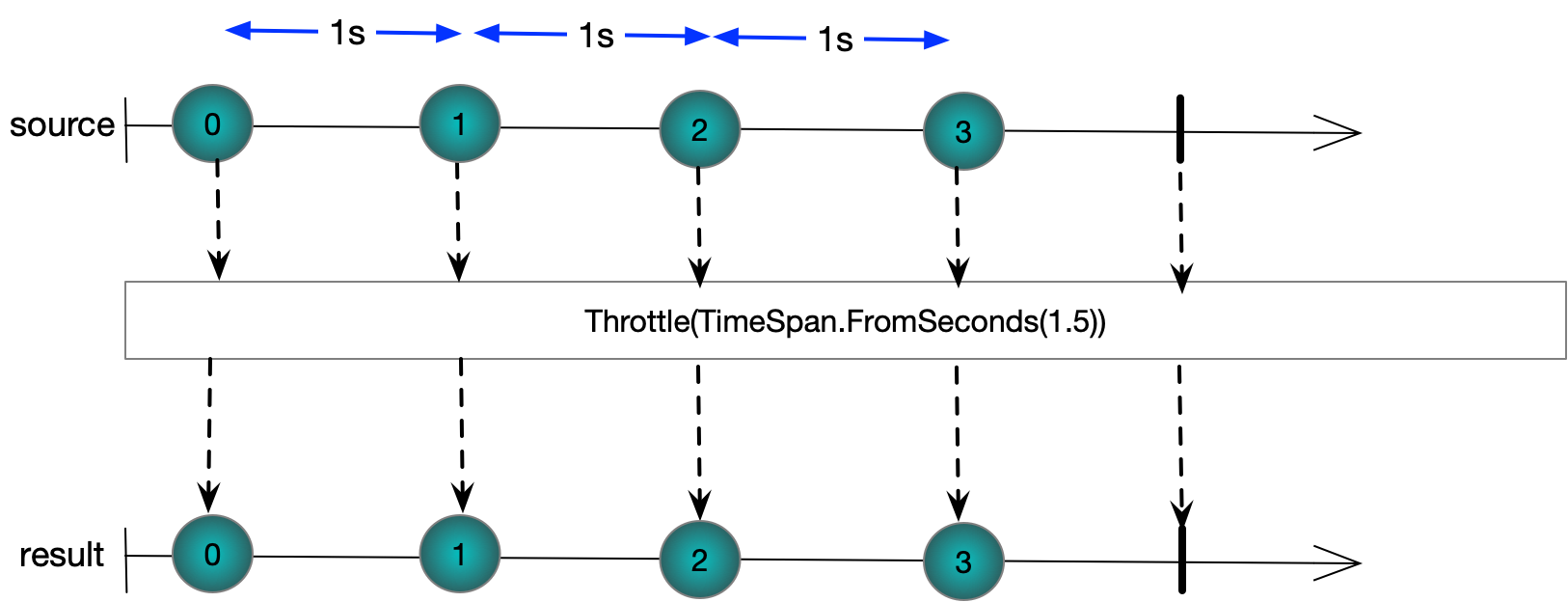
## Buffer(TimeSpan timeSpan, int count)

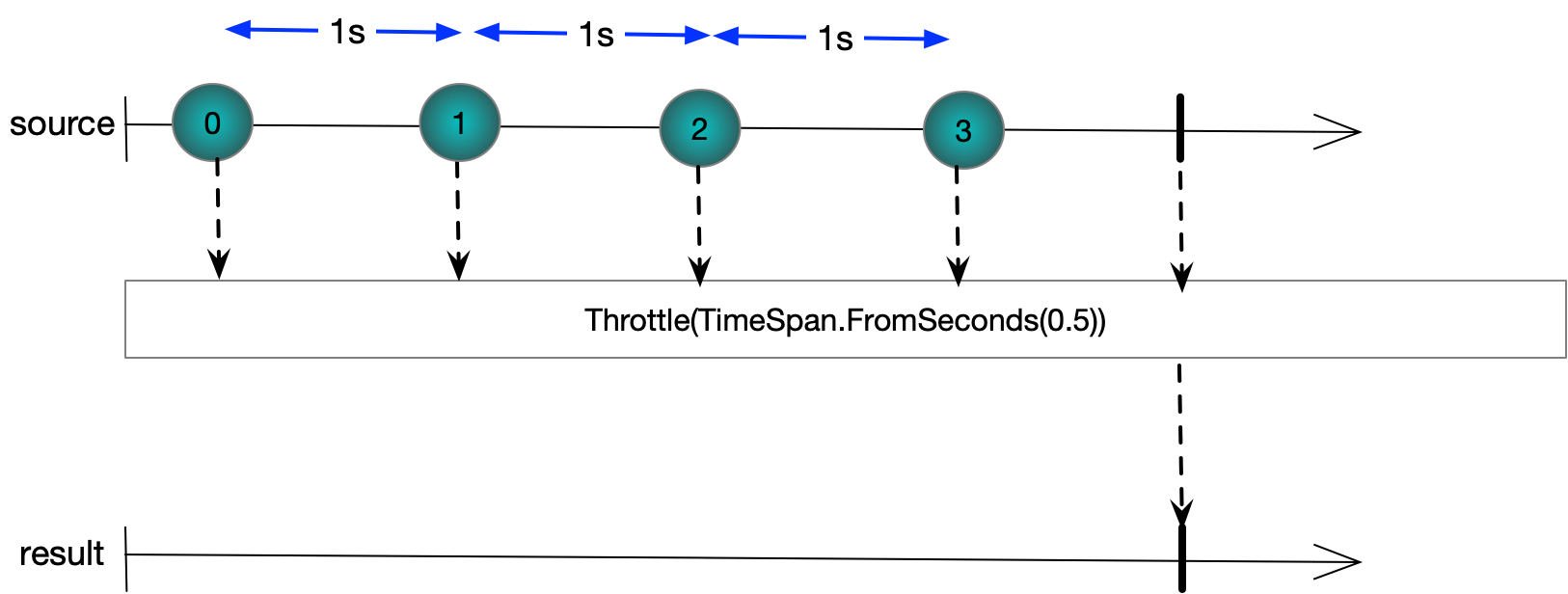


## Sample(TimeSpan timeSpan)



## Throttle(TimeSpan timeSpan)





Questions

Introduction

What is RX?

The Reactive Extensions API defines an interface for asynchronous streams of events and provides a set of operators that filter, combine and transform those 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>

What are the key parts of RX?

Two key interfaces that define an event source as first-class object

Large body of functions to create, transform, reduce and merge streams.

What are the core interfaces of RX?

IObservable<T> and IObserver<T>

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

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

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()

Rx provides an extension method on IObservable<T> which takes delegates. Internally this creates a type that implements IOberver<T> and calls the delegates as the observable calls its methods

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 events 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

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



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



What are the difficulties of implementing IObservable<T>

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

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. Creates decorators IObserver that wrap the actual IObserver and block messages on disposed subsriptions. Vv 6p

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

Observervable.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

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