



<http://www.clipcode.net/>

Clipcode Reference Library For The Rx API

Written By Eamon O'Tuathail

Last Update June 13th, 2011

Table of Contents

1: System.....	5
1.1: Overview.....	5
1.2: IObservable<T>.....	5
1.3: IObserver<T>.....	6
1.4: ObservableExtensions.....	8
2: System.Reactive.....	11
2.1: Overview.....	11
2.2: EventPattern.....	12
2.3: IEventPatternSource.....	13
2.4: IEventSource.....	13
2.5: Notification.....	14
2.6: Notification<T>.....	15
2.7: NotificationKind.....	16
2.8: Observer.....	16
2.9: TimeInterval.....	20
2.10: TimeStamped.....	21
2.11: Unit.....	21
3: System.Reactive.Concurrency.....	23
3.1: Overview.....	23
3.2: CurrentThreadScheduler.....	25
3.3: EventLoopScheduler.....	27
3.4: HistoricalScheduler.....	29
3.5: HistoricalSchedulerBase.....	30
3.6: ImmediateScheduler.....	31
3.7: IScheduledItem.....	33
3.8: IScheduler.....	33
3.9: NewThreadScheduler.....	35
3.10: Scheduler.....	37

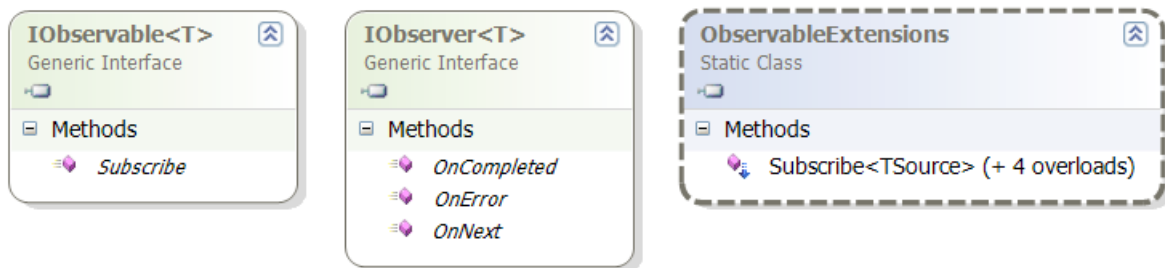
3.11: SynchronizationContextScheduler.....	40
3.12: TaskPoolScheduler.....	41
3.13: ThreadPoolScheduler.....	43
3.14: VirtualTimeScheduler.....	44
3.15: VirtualTimeSchedulerBase.....	47
4: System.Reactive.Disposables.....	51
4.1: Overview.....	51
4.2: BooleanDisposable.....	52
4.3: CancellationDisposable.....	53
4.4: CompositeDisposable.....	54
4.5: ContextDisposable.....	55
4.6: Disposable.....	55
4.7: MultipleAssignmentDisposable.....	56
4.8: RefCountDisposable.....	57
4.9: ScheduledDisposable.....	58
4.10: SerialDisposable.....	59
4.11: SingleAssignmentDisposable.....	60
5: System.Reactive.Joins.....	61
5.1: Pattern.....	62
5.2: Pattern<T1, T2>.....	62
5.3: Plan.....	63
6: System.Reactive.Linq.....	64
6.1: Overview.....	64
6.2: IGroupedObservable.....	65
6.3: Observable.....	65
7: System.Reactive.Subjects.....	134
7.1: Overview.....	134
7.2: AsyncSubject.....	135
7.3: BehaviorSubject.....	135
7.4: IConnectableObservable.....	135
7.5: ISubject<T1, T2>.....	136

7.6: ISubject<T>.....	137
7.7: ReplaySubject.....	137
7.8: Subject.....	138
7.9: Subject<T>.....	140
8: System.Threading.Tasks.....	142
8.1: Overview.....	142
8.2: TaskObservableExtensions.....	142

1: System

1.1: Overview

The types that are introduced by Reactive Extensions (Rx) into the System namespace can be divided into two categories: a pair of interfaces that are part of the .NET 4.0 installation (IObservable<T> and IObservable<T> in mscorlib) and the one static class that is part of the Rx deliverable (ObservableExtensions in System.Reactive.dll).



1.2: IObservable<T>

Represents an observable sequence.

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

Type Parameters

- T - The type of the element that is supplied to the IObservable. Note the out modifier on this type parameter means it is covariant, so you can use it or a more derived type

Remarks

Just as an enumerable represents a pull-based delivery of a sequence of elements (client calls `GetEnumerator` to get next available element - and blocks until one is available), an observable represents a push-based delivery, with the client initially passing in an observer with three methods (`OnNext`, `OnError` and `OnCompleted`), and then it is the observable that calls `OnNext` whenever the next element becomes available, and `OnCompleted` when finished, or `OnError` when an error is detected.

Quite often, observables will exhibit characteristics of a “cold observable”, which means each subscriber will see the entire set of elements in the sequence and each subscription results in a new cycle through the sequence (and a new

set of side-effects). In contrast, some observables will exhibit characteristics of a “hot observable”, which means a subscriber can subscribe to an “in-flight” sequence (think of a “live” data feed from the stock market), and will receive elements from that point onwards.

1.2.1: Subscribe

Subscribes an observer to the observable and returns an `IDisposable`, which may be used later to unsubscribe.

```
IDisposable Subscribe(IObserver<T> observer);
```

Parameters

- `observer (IObserver<T>)` - the observer to subscribe to the observable sequence

Return Values

- `IDisposable` - Call its `Dispose()` method to unsubscribe

Remarks

The observer must not be null.

To ensure observables and observers work as expected with Rx, above and beyond what is stated in the two interfaces, there are additional requirements known as the Rx contract. The expected order of calls to the observer is:

```
OnNext* (OnError|OnCompleted)?
```

(zero or more `OnNext` calls, followed by one of either `OnError` or `OnCompleted`, and nothing else). In particular, once either `OnError` or `OnCompleted` has been delivered, no further calls to the observer should be made by the observable.

A number of LINQ operators are provided to automate the creation of observables, and where possible, these should be used, instead of manually creating your own, to ensure that the Rx contract is adhered to.

1.3: IObserver<T>

Represents an observer of an observable sequence.

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

Type Parameters

- `T` - The type of the element that is supplied in the ensuing `OnNext` calls. Note the `in` modifier on this type parameter means it is contravariant, so

you can use it or a less derived type

Remarks

An observer (implements `IObserver`) observes an observable (implements `IObservable`). Whenever the observable has a new element, it makes it available by calling the `OnNext` of each observer that is subscribed to it.

The execution context (e.g. thread) on which calls to `IObserver` methods are made is dependent on which scheduler is used (this may or may not be on the same thread as `Subscribe` was called).

Note that all this type's method signatures return `void`.

A number of LINQ operators are provided to automate the creation of observers, and where possible, these should be used, instead of manually creating your own, to ensure that the Rx contract is adhered to.

1.3.1: OnCompleted

Called when the end of the sequence is determined.

```
void OnCompleted();
```

Remarks

Provided the observable adheres to the Rx contract, after this call, the observer can be sure that the end of the sequence has been reached, and no further calls to any of its methods will be made.

1.3.2: OnError

Called when an error has been detected.

```
void OnError(Exception error);
```

Parameters

- `error (Exception)` - The exception that was raised

Remarks

Provided the observable adheres to the Rx contract, after this call, the observer can be sure that no further calls to any of its methods will be made.

1.3.3: OnNext

Provides the next element in the sequence.

```
void OnNext(T value);
```

Parameters

- `values (T)` - the next element in the sequence

1.4: ObservableExtensions

Provides static methods to create an observer based on delegate parameters and subscribe it to an observable.

```
public static class ObservableExtensions {
    public static IDisposable Subscribe<TSource>(
        this IObservable<TSource> source);
    public static IDisposable Subscribe<TSource>(
        this IObservable<TSource> source,
        Action<TSource> onNext);
    public static IDisposable Subscribe<TSource>(
        this IObservable<TSource> source,
        Action<TSource> onNext,
        Action onCompleted);
    public static IDisposable Subscribe<TSource>(
        this IObservable<TSource> source,
        Action<TSource> onNext,
        Action<Exception> onError);
    public static IDisposable Subscribe<TSource>(
        this IObservable<TSource> source,
        Action<TSource> onNext,
        Action<Exception> onError,
        Action onCompleted);
}
```

Remarks

Observers and observables must comply with the Rx contract, which contains a number of rules in addition to simply implementing the interfaces. Hence it is recommended to use the supplied helper methods (such as these) to create them.

Note all these Subscribe methods return an IDisposable, which may be used to unsubscribe, but the observer itself is not returned.

1.4.1: Subscribe(source)

Create an observer that subscribes to the observable without having any methods called for OnNext/OnError/OnCompleted.

```
public static IDisposable Subscribe<TSource>(
    this IObservable<TSource> source);
```

Parameters

- source (TSource) - The non-null observable to subscribe to

Return Value

- IDisposable - used to unsubscribe

1.4.2: Subscribe(source, onNext)

Create an observer that subscribes to the observable with an OnNext method.

```
public static IDisposable Subscribe<TSource>(
    this IObservable<TSource> source,
```

```
Action<TSource> onNext);
```

Parameters

- source (TSource) - The non-null observable to subscribe to.
- onNext (Action<TSource>) - the delegate to call when a new element becomes available

Return Value

- IDisposable - used to unsubscribe

1.4.3: Subscribe(source, onNext, onCompleted)

Create an observer that subscribes to the observable with OnNext and OnCompleted methods.

```
public static IDisposable Subscribe<TSource>(
    this IObservable<TSource> source,
    Action<TSource> onNext,
    Action onCompleted);
```

Parameters

- source (TSource) - The non-null observable to subscribe to
- onNext (Action<TSource>) - the delegate to call when a new element becomes available
- onCompleted (Action) - the delegate to call when all elements in the sequence have been delivered

Return Value

- IDisposable - used to unsubscribe

1.4.4: Subscribe(source, onNext, onError)

Create an observer that subscribes to the observable with OnNext and OnError methods.

```
public static IDisposable Subscribe<TSource>(
    this IObservable<TSource> source,
    Action<TSource> onNext,
    Action<Exception> onError);
```

Parameters

- source (TSource) - The non-null observable to subscribe to
- onNext (Action<TSource>) - the delegate to call when a new element becomes available
- onError (Action<Exception>) - the delegate to call when an error occurs

Return Value

- `IDisposable` - used to unsubscribe

1.4.5: `Subscribe(source, onNext, onError)`

Create an observer that subscribes to the observable with `onNext`, `onError` and `onCompleted` methods.

```
public static IDisposable Subscribe<TSource>(
    this IObservable<TSource> source,
    Action<TSource> onNext,
    Action<Exception> onError,
    Action onCompleted);
```

Parameters

- `source (TSource)` - The non-null observable to subscribe to.
- `onNext (Action<TSource>)` - the delegate to call when a new element becomes available
- `onError (Action<Exception>)` - the delegate to call when an error occurs
- `onCompleted (Action)` - the delegate to call when all elements in the sequence have been delivered

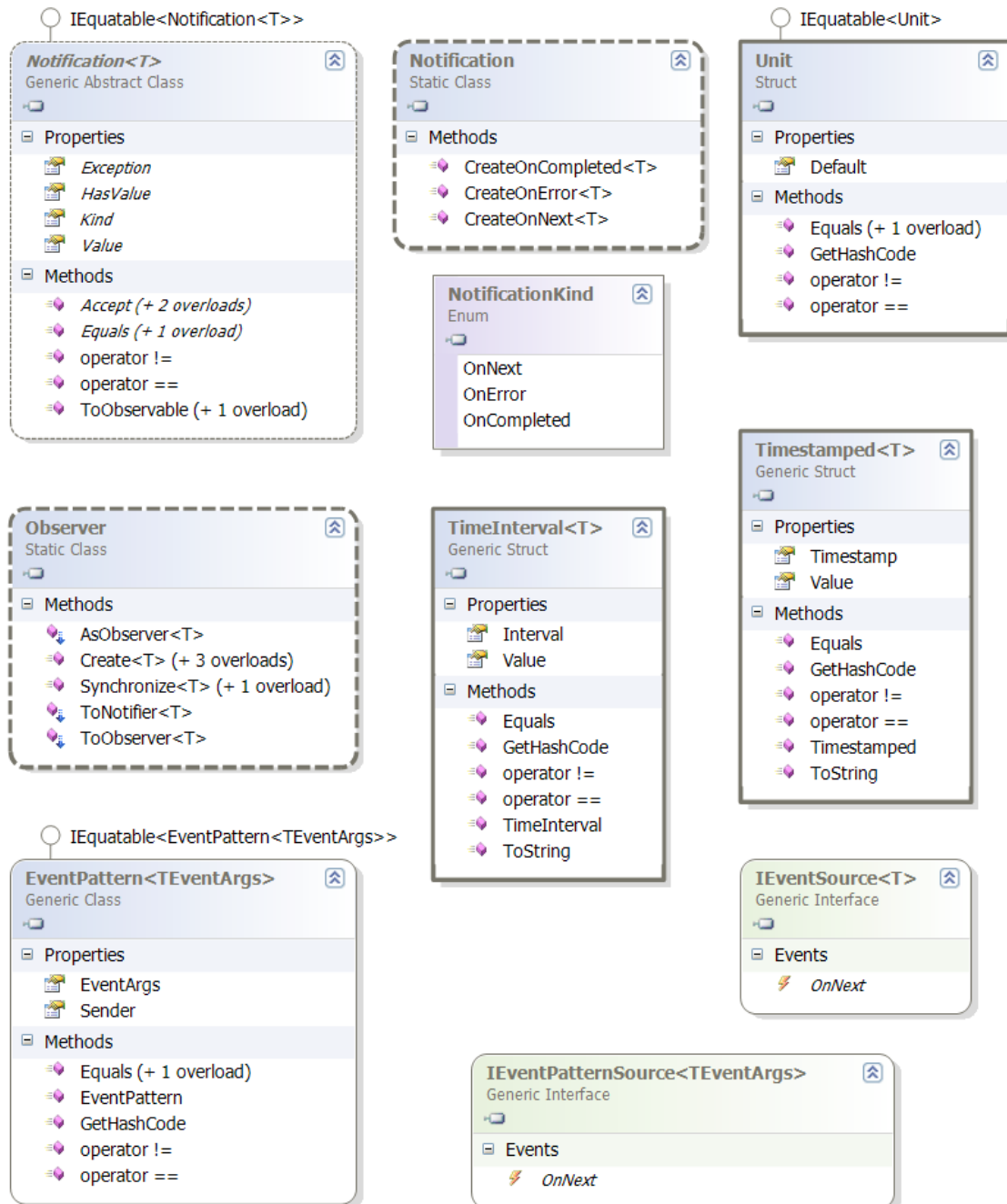
Return Value

- `IDisposable` - used to unsubscribe (calling `Dispose()` disposes of this observer's subscription to the observable [there will be one subscription for each subscribed observer] - the observable itself is not disposed)

2: System.Reactive

2.1: Overview

System.Reactive has the following types:



2.2: EventPattern

A class to represent a sender and event arg as a combined type.

```
public class EventPattern<TEventArgs> :
    IEquatable<EventPattern<TEventArgs>>
    where TEventArgs : global::System.EventArgs
{
    public EventPattern(object sender, TEventArgs e);
    public static bool operator !=(EventPattern<TEventArgs> first,
                                   EventPattern<TEventArgs> second);
    public static bool operator ==(EventPattern<TEventArgs> first,
                                   EventPattern<TEventArgs> second);
    public TEventArgs EventArgs { get; }
    public object Sender { get; }
    public bool Equals(EventPattern<TEventArgs> other);
    public override bool Equals(object obj);
    public override int GetHashCode();
}
```

Remarks

Note in the class definition that the generic parameter `TEventArgs` must derive from `System.EventArgs` in the global namespace.

The .NET event pattern is a recommendation for how to design event handlers. A sender (object) and event args (System.EventArgs-derived) parameters are passed in to the event handler.

This class is used to define the return values from the various `System.Reactive.Linq.Observable`'s `FromEventPattern` overloaded methods and it is used as an input parameter to `System.Reactive.Linq.Observable`'s `ToEventPattern` method. `System.Reactive.Linq.Observable` also has methods to work with events directly (i.e. those that might not follow the event pattern, and so might have different parameters).

Also see remarks for `IEventPatternSource`.

2.2.1: EventPattern Constructor

Creates an `EventPattern` instance using the supplied parameters.

```
public EventPattern(object sender, TEventArgs e);
```

Parameters

- sender (object) - the sender
- eventArgs (TEventArgs) - The strongly typed event args

2.2.2: EventArgs

A property with a getter only representing the event arguments.

```
public TEventArgs EventArgs { get; }
```

2.2.3: Sender

A property with a getter only representing the sender of the event.

```
public object Sender { get; }
```

2.3: IEventPatternSource

An interface representing an event handler for OnNext.

```
public interface IEventPatternSource<TEventArgs>
    where TEventArgs : global::System.EventArgs {
    event EventHandler<TEventArgs> OnNext;
}
```

2.3.1: Remarks

This interface is used to define the return value from the System.Reactive.Linq.Observable's ToEventPattern method.

There are three common asynchronous patterns used with .NET: the Asynchronous Programming Model (APM) pattern [BeginXXX/EndXXX and IAsyncResult], the Event-asynchronous pattern (EAP) [XXXEventHandler and XXXEventArgs] and the Task-based Asynchronous Pattern (TAP) [used with Async C#].

The Rx EventPattern class and IEventPatternSource interface are useful when used EAP and Rx together.

2.3.2: OnNext

Event raised for next element in stream.

```
event EventHandler<TEventArgs> OnNext;
```

2.4: IEventSource

An interface representing an Action for OnNext.

```
public interface IEventSource<T> {
    event Action<T> OnNext;
}
```

Remarks

The difference between IEventPatternSource and IEventSource is two-fold: for the former, the type parameter must derive from System.EventArgs whereas there is no such constraint with the latter, and secondly, the former has an event of type EventHandler whereas the latter has an event of type Action.

2.4.1: OnNext

Event raised for next element in stream.

```
event Action<T> OnNext;
```

2.5: Notification

A static class used to create Notification<T> instances.

```
public static class Notification {
    public static Notification<T> CreateOnCompleted<T>();
    public static Notification<T> CreateOnError<T>(Exception error);
    public static Notification<T> CreateOnNext<T>(T value);
}
```

Remarks

One could use this class to manually create notifications.

2.5.1: CreateOnCompleted

Creates an OnCompleted notification.

```
public static Notification<T> CreateOnCompleted<T>();
```

Return Value

- Notification<T> - The OnCompleted notification (Kind = NotificationKind.OnCompleted)

2.5.2: CreateOnError

Creates an OnError notification.

```
public static Notification<T> CreateOnError<T>(Exception error);
```

Parameters

- error (Exception) - The exception for the OnError message

Return Value

- Notification<T> - The OnError notification (Kind = NotificationKind.OnError)

2.5.3: CreateOnNext

Creates an OnNext notification.

```
public static Notification<T> CreateOnNext<T>(T value);
```

Parameters

- value (T) - The value of the OnNext message

Return Value

- Notification<T> - The OnNext notification (Kind = NotificationKind.OnNext and Value to value parameter)

2.6: Notification<T>

The abstract base class for Notification<T> notifications.

```
public abstract class Notification<T> : IEquatable<Notification<T>> {
    public static bool operator
        !=(Notification<T> left, Notification<T> right);
    public static bool operator
        ==(Notification<T> left, Notification<T> right);
    public abstract Exception Exception{get;}
    public abstract bool HasValue{get;}
    public abstract NotificationKind Kind{get;}
    public abstract T Value{ get; }
    public abstract TResult Accept<TResult>(IObserver<T,TResult> observer);
    public abstract void Accept(IObserver<T> observer);
    public abstract void Accept(
        Action<T> onNext,
        Action<Exception> onError,
        Action onCompleted);
    public abstract TResult Accept<TResult>(
        Func<T,TResult> onNext,
        Func<Exception,TResult> onError,
        Func<TResult> onCompleted);
    public abstract bool Equals(Notification<T> other);
    public override bool Equals(object obj);
}
```

Remarks

Notifications are used with materialization, which converts the results of observing an observable sequence (method calls to OnNext/OnError/OnCompleted) into a stream of notifications, and dematerialization, which does the reverse (see System.Reactive.Linq.Observable's Materialize and Dematerialize methods; also see System.Reactive.Observer's ToNotifier and ToObserver methods).

This is an abstract class and there are three concrete implementations of notifications (one each for OnNext/OnCompleted/OnError) and these are internal to Rx.

2.6.1: Exception

The Exception instance (for notification of kind NotificationKind.Exception).

```
public abstract Exception Exception{get;}

```

2.6.2: Kind

The kind of this notification.

```
public abstract NotificationKind Kind{get;}

```

2.6.3: Value

The value of the notification (if any).

```
public abstract T Value{ get; }

```

2.6.4: HasValue

Whether this notification has a value.

```
public abstract bool HasValue{get;}
```

2.6.5: Accept

Invokes a delegate relevant to the notification kind.

```
public abstract TResult Accept<TResult>(IObserver<T, TResult> observer);
public abstract void Accept(IObserver<T> observer);
public abstract void Accept(
    Action<T> onNext,
    Action<Exception> onError,
    Action onCompleted);
public abstract TResult Accept<TResult>(
    Func<T, TResult> onNext,
    Func<Exception, TResult> onError,
    Func<TResult> onCompleted);
```

Remarks

When a notification is received, a delegate from an observer or directly supplied may be called, relevant to the notification kind.

2.7: NotificationKind

An enumeration identifying the kind of the notification.

```
public enum NotificationKind {
    OnNext=0,
    OnError=1,
    OnCompleted=2,
}
```

2.8: Observer

A collection of extension and static methods for working with Observers.

```
public static class Observer {
    public static IObserver<T> AsObserver<T>(this IObserver<T> observer);

    public static IObserver<T> Create<T>(Action<T> onNext);
    public static IObserver<T> Create<T>(
        Action<T> onNext, Action onCompleted);
    public static IObserver<T> Create<T>(
        Action<T> onNext, Action<Exception> onError);
    public static IObserver<T> Create<T>(
        Action<T> onNext, Action<Exception> onError, Action onCompleted);

    public static IObserver<T> Synchronize<T>(IObserver<T> observer);
    public static IObserver<T> Synchronize<T>(
        IObserver<T> observer, object gate);
    public static Action<Notification<T>> ToNotifier<T>(
        this IObserver<T> observer);
    public static IObserver<T> ToObserver<T>(
        this Action<Notification<T>> handler);
}
```

Remarks

This static class provides helper methods to assist with working with observers, in areas such as creation, converting between IObservable and notifications, anonymous observers and synchronization.

2.8.1: AsObserver

Create an observer that is a wrapper for another, in essence hiding the wrapped observer's type and instance identity

```
public static IObservable<T> AsObserver<T>(this IObservable<T> observable);
```

Parameters

- observable (this IObservable<T>) - The observable to wrap

Return Values

- IObservable<T> - The wrapped observable

2.8.2: Create(Action<T>)

Creates an observable using the supplied action for OnNext.

```
public static IObservable<T> Create<T>(Action<T> onNext);
```

Parameters

- onNext (Action<T>) - The method to call for each received message

Return Value

- IObservable<T> - The observable with the onNext method

Remarks

OnCompleted and OnException have no effect.

2.8.3: Create(Action<T>, Action)

Creates an observable using the supplied action for OnNext and OnCompleted.

```
public static IObservable<T> Create<T>(
    Action<T> onNext, Action onCompleted);
```

Parameters

- onNext (Action<T>) - The method to call for each received message
- onCompleted (Action) - the method to call for OnCompleted

Return Value

- IObservable<T> - The observable with the onNext method

Remarks

OnException has no effect.

2.8.4: Create(Action<T>, Action<Exception>)

Creates an observer using the supplied action for OnNext and OnError.

```
public static IObservable<T> Create<T>(
    Action<T> onNext, Action<Exception> onError);
```

Parameters

- onNext (Action<T>) - The method to call for each received message
- onError (Action<Exception>) - The method to call for errors

Return Value

- IObservable<T> - The observer with the onNext and onError methods

Remarks

OnCompleted has no effect.

2.8.5: CreateAction<T>, Action<Exception>, Action)

Creates an observer using the supplied action for OnNext, OnError and OnCompleted.

```
public static IObservable<T> Create<T>(
    Action<T> onNext, Action<Exception> onError, Action onCompleted);
```

Parameters

- onNext (Action<T>) - The method to call for each received message
- onError (Action<Exception>) - The method to call for errors
- onCompleted (Action) - the method to call for OnCompleted

Return Value

- IObservable<T> - The observer with the onNext method

2.8.6: Synchronize

Creates an observer whose incoming messages are synchronized, regardless of whether messages to the parameter observer are synchronized.

```
public static IObservable<T> Synchronize<T>(IObservable<T> observer);
```

Parameters

- observer (IObservable<T>) - the observer whose incoming messages are to be synchronized

Return Value

- IObservable<T> - The synchronized observer

Remarks

As part of the Rx Contract, observers should be able to expect messages to be synchronized by the observable. Where this is not the case, this method may be used.

Note that `System.Reactive.Linq.Observable` also has a `Synchronize` method, and its role is to synchronize outgoing messages to a particular observer.

2.8.7: Synchronize

Creates an observer whose incoming messages are synchronized using the supplied gate object, regardless of whether messages to the parameter observer are synchronized.

```
public static IObservable<T> Synchronize<T>(
    IObservable<T> observable, object gate);
```

Parameters

- `observable` (`IObservable<T>`) - the observable whose incoming messages are to be synchronized
- `gate` (`Object`) - the synchronization object to be used as a gate

Return Value

- `IObservable<T>` - The synchronized observable

2.8.8: ToNotifier

Creates a notification handler which passes received notifications to the `observer` parameter.

```
public static Action<Notification<T>> ToNotifier<T>(
    this IObservable<T> observable);
```

Parameters

- `observer` (`this IObservable<T>`) - The observable which is to receive messages via the notification handler

Return Value

- `Action<Notification<T>>` - The notification handler which can be used for observables of materialized observables

Remarks

This method creates a notification handler which wraps an observable. When the notification handler is passed notifications, these are forwarded to the observable.

2.8.9: ToObserver

Constructs an observable that can be subscribed to observables, and that calls a

notification handler for each message received.

```
public static IObservable<T> ToObserver<T>(
    this Action<Notification<T>> handler);
```

Parameters

- handler (Action<Notification<T>>) - The handler to call for each received message

Return Value

- IObservable<T> - The created observer

Remarks

Note that the parameter to the action is Notification<T>, and not T.

2.9: TimeInterval

A wrapper for an instance of a type that also stores a timespan.

```
public struct TimeInterval<T> {
    public TimeInterval(T value, TimeSpan interval);
    public static bool operator !=(
        TimeInterval<T> first, TimeInterval<T> second);
    public static bool operator ==(
        TimeInterval<T> first, TimeInterval<T> second);
    public TimeSpan Interval { get; }
    public T Value { get; }
    public override bool Equals(object obj);
    public override int GetHashCode();
    public override string ToString();
}
```

Remarks

A helper struct used when there is a need to store a value and a (TimeSpan) time interval.

2.9.1: TimeInterval Constructor

Creates a new instance of a time interval value.

```
public TimeInterval(T value, TimeSpan interval);
```

Parameters

- value (T) - The instance to which the timestamp applies
- timestamp (DateTimeOffset) - the timestamp

2.9.2: Interval

The time interval for the value.

```
public TimeSpan Interval { get; }
```

2.9.3: Value

The instance to which the time interval applies.

```
public T Value { get; }
```

2.10: TimeStamped

A wrapper for an instance of a type that also stores a timestamp.

```
public struct TimeStamped<T> {
    public TimeStamped(T value, DateTimeOffset timestamp);
    public static bool operator !=(
        TimeStamped<T> first, TimeStamped<T> second);
    public static bool operator ==(
        TimeStamped<T> first, TimeStamped<T> second);
    public DateTimeOffset Timestamp { get; }
    public T Value { get; }
    public override bool Equals(object obj);
    public override int GetHashCode();
    public override string ToString();
}
```

Remarks

A helper struct used when there is a need to store a value and a (DateTimeOffset) timestamp.

2.10.1: TimeStamped Constructor

Creates a new instance of a timestamped value.

```
public TimeStamped(T value, DateTimeOffset timestamp);
```

Parameters

- value (T) - The instance to which the timestamp applies
- timestamp (DateTimeOffset) - the timestamp

2.10.2: Timestamp

The timestamp for the value.

```
public DateTimeOffset Timestamp { get; }
```

2.10.3: Value

The instance to which the timestamp applies.

```
public T Value { get; }
```

2.11: Unit

Some operators expect a type parameter, and where none is needed. Unit may be supplied. Consider it as similar in concept to void.

```
[Serializable]
public struct Unit: IEquatable<Unit>{
```

```
public static bool operator !=(Unit unit1, Unit unit2);  
public static bool operator ==(Unit unit1, Unit unit2);  
public static Unit Default { get; }  
public override bool Equals(object obj);  
public bool Equals(Unit other);  
public override int GetHashCode();  
}
```

2.11.1: Operator !=

Simply returns the false value.

```
public static bool operator !=(Unit unit1, Unit unit2);
```

2.11.2: Operator ==

Simply returns the true value.

```
public static bool operator ==(Unit unit1, Unit unit2);
```

2.11.3: Default

Returns a default unit.

```
public static Unit Default { get; }
```

2.11.4: Equals (Object)

If the Object parameter is an instance of Unit, return true, otherwise return false.

```
public override bool Equals(object obj);
```

2.11.5: Equals (Unit)

Simply returns the true value.

```
public bool Equals(Unit other);
```

2.11.6: GetHashCode

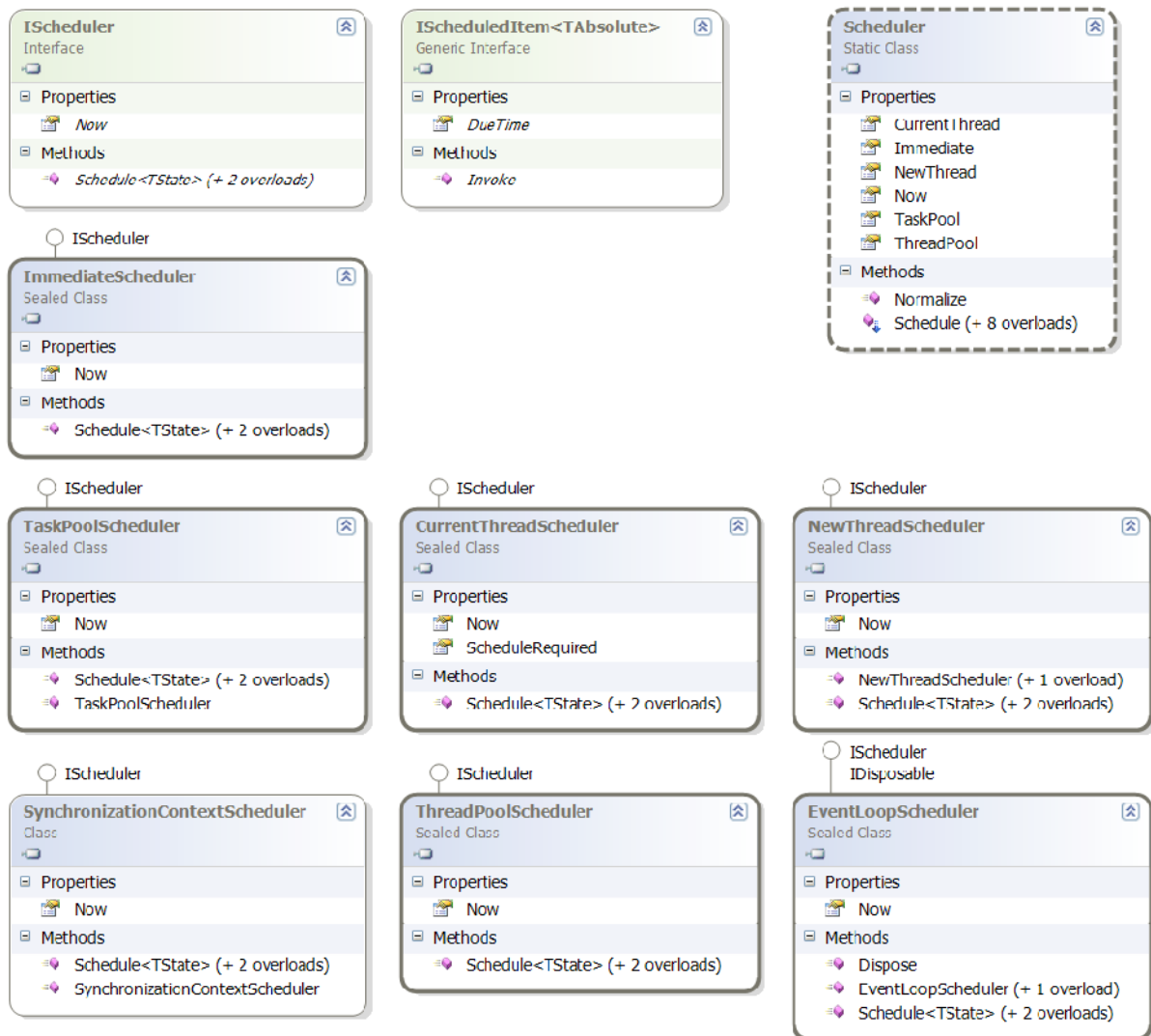
Returns a hash code.

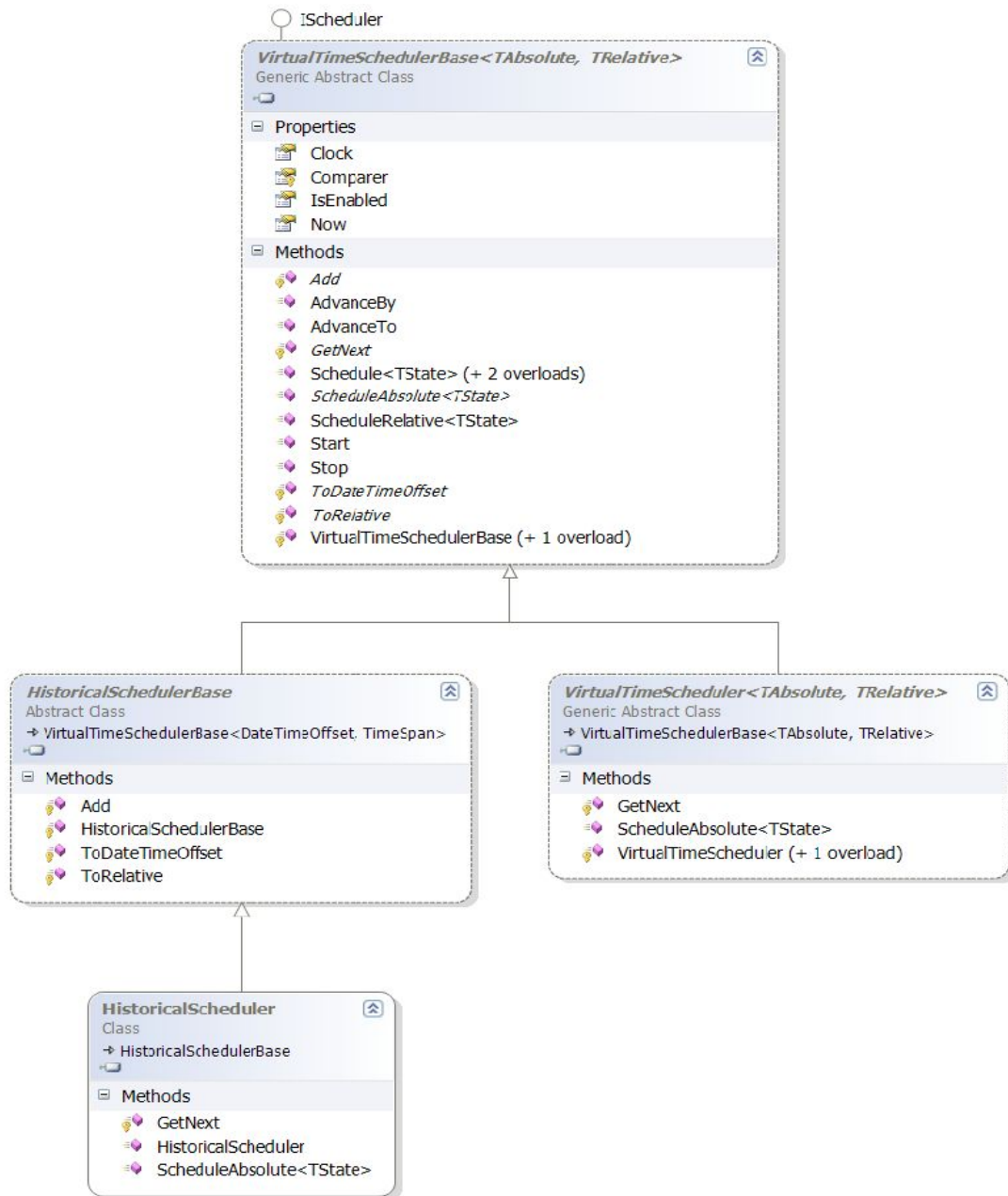
```
public override int GetHashCode();
```

3: System.Reactive.Concurrency

3.1: Overview

System.Reactive.Concurrency has the following types:





3.2: CurrentThreadScheduler

A trampoline-based scheduler which schedules work on the current thread using a priority queue based on dueTime.

```
public sealed class CurrentThreadScheduler : IScheduler {
    public DateTimeOffset Now { get; }
    public bool ScheduleRequired { get; }
    public IDisposable Schedule<TState>(TState state,
        Func<IScheduler, TState, IDisposable> action);
    public IDisposable Schedule<TState>(TState state,
        DateTimeOffset dueTime,
        Func<IScheduler, TState, IDisposable> action);
    public IDisposable Schedule<TState>(TState state, TimeSpan dueTime,
        Func<IScheduler, TState, IDisposable> action);
}
```

Remarks

A trampoline is a way of queuing up actions to be called, and executing them in a specific order. When using deep recursion, the call stack can get too large. A trampoline is a way to queue up calls that need to be made, but not have them all on the call stack at the same time.

CurrentThreadScheduler has no public constructors. To access a CurrentThreadScheduler instance for the current thread, use the CurrentThread property from the System.Concurrency.Scheduler static class.

This scheduler could be called a TrampolineScheduler, because that is its primary feature.

For CurrentThreadScheduler, a priority queue is used, sorted by dueTime. It can be very important when scheduled actions themselves need to schedule inner actions, and such recursion can lead to the stack growing very quickly and problems with locking between the inner and outer actions. By using a trampoline, the queue of actions can be executed sorted on dueTime and one action can run until completion (and release its locks) before the next action begins.

3.2.1: Now

Now for a CurrentThreadScheduler is the same as DateTimeOffset.Now.

```
public DateTimeOffset Now { get; }
```

3.2.2: ScheduleRequired

A boolean indicating whether there are queued elements awaiting scheduling.

```
public bool ScheduleRequired { get; }
```

3.2.3: Schedule(TState, Func)

Schedule an action for execution on the current thread.

```
IDisposable Schedule<TState>(TState state,
```

```
Func<IScheduler, TState, IDisposable> action);
```

Parameters

- state (TState) - the state to pass to the func
- action (Func<IScheduler, TState, IDisposable>) - the action to execute

Return Value

- IDisposable - represents the scheduled action

3.2.4: Schedule(TState, DateTimeOffset, Func)

Schedule an action for execution at or as soon as possible after dueTime.

```
IDisposable Schedule<TState>(TState state, DateTimeOffset dueTime,
    Func<IScheduler, TState, IDisposable> action);
```

Parameters

- state (TState) - the state to pass to the func
- action (Func<IScheduler, TState, IDisposable>) - the action to execute
- dueTime (DateTimeOffset) - the time offset to wait until executing the action

Return Value

- IDisposable - represents the scheduled action

Schedule(TState, TimeSpan, Func)

Schedule an action for execution at or as soon as possible after dueTime.

```
IDisposable Schedule<TState>(TState state, TimeSpan dueTime,
    Func<IScheduler, TState, IDisposable> action);
```

Parameters

- state (TState) - the state to pass to the func
- action (Func<IScheduler, TState, IDisposable>) - the action to execute
- DueTime (TimeSpan) - the time span to wait until executing the action.

Return Value

- IDisposable - represents the scheduled action

Remarks

Scheduling an action on the CurrentThreadScheduler involves use of a trampoline. A queue is maintained, and each scheduled action is placed on it. The queue is a priority queue, ordered by dueTime. Hence the order of execution of scheduled actions is not that of the order of scheduling (as it would be e.g.

with an ImmediateScheduler) but rather based on dueTime.

3.3: EventLoopScheduler

Creates a separate thread to run scheduled actions and maintains a priority queue which uses a System.Threading.ManualResetEvent as a lock.

```
public class EventLoopScheduler : IScheduler, IDisposable {
    public EventLoopScheduler();
    public EventLoopScheduler(Func<ThreadStart, Thread> threadFactory);
    public DateTimeOffset Now { get; }
    public void Dispose();
    public IDisposable Schedule<TState>(TState state,
        Func<IScheduler, TState, IDisposable> action);
    public IDisposable Schedule<TState>(TState state,
        DateTimeOffset dueTime,
        Func<IScheduler, TState, IDisposable> action);
    public IDisposable Schedule<TState>(TState state, TimeSpan dueTime,
        Func<IScheduler, TState, IDisposable> action);
}
```

Remarks

Note that “event loop” here has nothing to do with user interfaces, instead it refers to the ManualResetEvent from System.Threading.

This scheduler maintains a single private worker thread which is solely dedicated to executing actions scheduled via the EventLoopScheduler.

The EventLoopScheduler en-queues actions in a priority queue, and its private worker thread executes them.

EventLoopScheduler()

Constructor for an EventLoopScheduler that uses a default name for the thread.

```
public EventLoopScheduler();
```

EventLoopScheduler(Func)

Constructor for an EventLoopScheduler that uses the supplied thread factory.

```
public EventLoopScheduler(Func<ThreadStart, Thread> threadFactory);
```

Parameters

- threadfactory (Func<ThreadStart, Thread>) - The func used to create the thread

Remarks

Creates a worker thread using the func supplied in the parameter.

Now

Now for a EventLoopScheduler is the same as DateTimeOffset.Now.

```
public DateTimeOffset Now{get;}
```

Dispose

Causes the worker thread to exit.

```
public void Dispose();
```

3.3.1: Schedule(TState, Func)

Schedule an action for execution on the current thread.

```
IDisposable Schedule<TState>(TState state,  
                             Func<IScheduler, TState, IDisposable> action);
```

Parameters

- state (TState) - the state to pass to the func
- action (Func<IScheduler, TState, IDisposable>) - the action to execute

Return Value

- IDisposable - represents the scheduled action

3.3.2: Schedule(TState, DateTimeOffset, Func)

Schedule an action for execution at or as soon as possible after dueTime.

```
IDisposable Schedule<TState>(TState state, DateTimeOffset dueTime,  
                             Func<IScheduler, TState, IDisposable> action);
```

Parameters

- state (TState) - the state to pass to the func
- action (Func<IScheduler, TState, IDisposable>) - the action to execute
- dueTime (DateTimeOffset) - the time offset to wait until executing the action

Return Value

- IDisposable - represents the scheduled action

Schedule(TState, TimeSpan, Func)

Schedule an action for execution at or as soon as possible after dueTime.

```
IDisposable Schedule<TState>(TState state, TimeSpan dueTime,  
                             Func<IScheduler, TState, IDisposable> action);
```

Parameters

- state (TState) - the state to pass to the func
- action (Func<IScheduler, TState, IDisposable>) - the action to execute
- DueTime (TimeSpan) - the time span to wait until executing the action.

Return Value

- `IDisposable` - represents the scheduled action

3.4: HistoricalScheduler

A concrete class that implements a virtual time scheduler base with absolute time based on `DateTimeOffset` and relative time based on `TimeSpan`.

```
public class HistoricalScheduler : HistoricalSchedulerBase {
    public HistoricalScheduler();

    protected override IScheduledItem<DateTimeOffset> GetNext();
    public override IDisposable ScheduleAbsolute<TState>(
        TState state, DateTimeOffset dueTime,
        Func<IScheduler, TState, IDisposable> action);
}
```

Remarks

This is one of four classes related to virtual time and is the easiest to use since it is not abstract and hence there is no need to create a custom class with custom implementation of certain abstract methods.

Those seeking to use virtual time will often find what they need by using `HistoricalScheduler`.

3.4.1: HistoricalScheduler

Constructor for `HistoricalScheduler`.

```
public HistoricalScheduler();
```

3.4.2: GetNext

Returns the next scheduled item to invoke.

```
protected override IScheduledItem<DateTimeOffset> GetNext();
```

Return Value

- `IScheduledItem<DateTimeOffset>` - next scheduled item

3.4.3: ScheduleAbsolute

Schedule an action using absolute time.

```
public override IDisposable ScheduleAbsolute<TState>(
    TState state, DateTimeOffset dueTime,
    Func<IScheduler, TState, IDisposable> action);
```

Parameters

- `dueTime` (`DateTimeOffset`) - When the action should be executed
- `action` (`Func<IScheduler, TState, IDisposable>`) - the action whose execution is to be scheduled

Return Value

- `IDisposable` – may be used to cancel the scheduled item (if it has not already been executed)

3.5: HistoricalSchedulerBase

Provides an abstract base for a virtual time scheduler with absolute time represented by `DateTimeOffset` and relative time represented by `TimeSpan`.

```
public abstract class HistoricalSchedulerBase :
    VirtualTimeSchedulerBase<DateTimeOffset, TimeSpan> {

    protected HistoricalSchedulerBase();

    protected override DateTimeOffset Add(
        DateTimeOffset absolute, TimeSpan relative);
    protected override DateTimeOffset ToDateTimeOffset(
        DateTimeOffset absolute);
    protected override TimeSpan ToRelative(TimeSpan timeSpan);
}
```

Remarks

An abstract class with no type parameters and which is derived from `VirtualTimeSchedulerBase` with type parameters set to `DateTimeOffset` and `TimeSpan`. The two methods that still need to be implemented by derived classes are `GetNext` and `ScheduleAbsolute`.

3.5.1: HistoricalSchedulerBase()

Protected constructor for `HistoricalSchedulerBase`.

```
protected HistoricalSchedulerBase();
```

3.5.2: Add

Adds a relative time to an absolute time.

```
protected override DateTimeOffset Add(
    DateTimeOffset absolute, TimeSpan relative);
```

3.5.3: Parameters

- `absolute` (`DateTimeOffset`) – The absolute time
- `relative` (`TimeSpan`) – The relative time

Return Value

- `DateTimeOffset` – `absolute` + `relative` time

3.5.4: ToDateTimeOffset

Converts to a `DateTimeOffset` from an absolute time.

```
protected override DateTimeOffset ToDateTimeOffset(
    DateTimeOffset absolute);
```

3.5.5: ToRelative

Converts from a `TimeSpan` to a relative time.

```
protected override TimeSpan ToRelative(TimeSpan timeSpan);
```

Parameters

- timeSpan (TimeSpan) - the relative time to convert

Return Value

- TimeSpan - the result of the conversion

3.6: ImmediateScheduler

A scheduler which schedules work on the current thread for immediate execution (before Schedule() returns).

```
public sealed class ImmediateScheduler : IScheduler {
    public DateTimeOffset Now { get; }
    public IDisposable Schedule<TState>(TState state,
        Func<IScheduler, TState, IDisposable> action);
    public IDisposable Schedule<TState>(TState state,
        DateTimeOffset dueTime,
        Func<IScheduler, TState, IDisposable> action);
    public IDisposable Schedule<TState>(TState state, TimeSpan dueTime,
        Func<IScheduler, TState, IDisposable> action);
}
```

Remarks

ImmediateScheduler has no public constructors. To access an ImmediateScheduler instance for the current thread, use the Immediate property from the System.Concurrency.Scheduler static class.

ImmediateScheduler is the simplest of the schedulers as it just executes the action on the current thread and the Schedule call is blocking until the action has been executed.

Now

Now for an ImmediateScheduler is the same as System.DateTimeOffset.Now.

```
public DateTimeOffset Now { get; }
```

3.6.1: Schedule(TState, Func)

Schedule an action for immediate execution.

```
IDisposable Schedule<TState>(TState state,
    Func<IScheduler, TState, IDisposable> action);
```

Parameters

- state (TState) - the state to pass to the func
- action (Func<IScheduler, TState, IDisposable>) - the action to execute

Return Value

- `IDisposable` - represents the scheduled action (since the action has been executed by the time this `Schedule` returns, one cannot use this disposable to dispose of the scheduled action)

Schedule(TState, DateTimeOffset, Func)

Schedule an action for execution at or as soon as possible after `dueTime`.

```
IDisposable Schedule<TState>(TState state, DateTimeOffset dueTime,
                             Func<IScheduler, TState, IDisposable> action);
```

Parameters

- `state` (T State) - the state to pass to the fun
- `action` (`Func<IScheduler, TState, IDisposable>`) - the action to execute
- `dueTime` (`DateTimeOffset`) - the time offset to wait until executing the action

Return Value

- `IDisposable` - represents the scheduled action (since the action has been executed by the time this `Schedule` returns, one cannot use this disposable to dispose of the scheduled action)

Remarks

If `dueTime` is negative, it is treated as if it were 0, and so the action is executed immediately. If `dueTime > 0`, then the thread sleeps for `dueTime`, executes the action and returns.

Schedule(TState, TimeSpan, Func)

Schedule an action for execution at or as soon as possible after `dueTime`.

```
IDisposable Schedule<TState>(TState state, TimeSpan dueTime,
                             Func<IScheduler, TState, IDisposable> action);
```

Parameters

- `state` (TState) - the state to pass to the func
- `action` (`Func<IScheduler, TState, IDisposable>`) - the action to execute
- `DueTime` (`TimeSpan`) - the time span to wait until executing the action.

Return Value

- `IDisposable` - represents the scheduled action (since the action has been executed by the time this `Schedule` returns, one cannot use this disposable to dispose of the scheduled action)

Remarks

If `dueTime` is negative, it is treated as if it were 0, and so the action is executed immediately. If `dueTime > 0`, then the thread sleeps for `dueTime`, executes the action and returns.

3.7: IScheduledItem

A item that has been scheduled.

```
public interface IScheduledItem<TAbsolute> {
    TAbsolute DueTime { get; }
    void Invoke();
}
```

Type Parameters

- `TAbsolute` - The type to use for `DueTime` (usually `DateTimeOffset` or `TimeSpan`)

Remarks

`IScheduledItem` represents a scheduled item. It is used as the return value in `VirtualTimeSchedulerBase.GetNext()`.

3.7.1: DueTime

Property with getter only returning when the item is scheduled to be invoked.

```
TAbsolute DueTime { get; }
```

3.7.2: Invoke

Invokes the scheduled item.

```
void Invoke();
```

3.8: IScheduler

All concurrency activities in Rx flow through schedulers, as defined by this interface.

```
public interface IScheduler {
    DateTimeOffset Now { get; }
    IDisposable Schedule<TState>(TState state,
        Func<IScheduler, TState, IDisposable> action);
    IDisposable Schedule<TState>(TState state, DateTimeOffset dueTime,
        Func<IScheduler, TState, IDisposable> action);
    IDisposable Schedule<TState>(TState state, TimeSpan dueTime,
        Func<IScheduler, TState, IDisposable> action);
}
```

Remarks

Schedulers provide flexibility in how, where and when actions are executed using Rx. The two important questions schedulers answer is which thread to use

when executing an action and when to execute the action.

Three variations of the Schedule method are defined: the first parameter for all is state, the last parameter for all is the action to be scheduled (the same signature) and the difference is the middle parameter. For the first overload, there is no middle parameter, for the second it is a DateTimeOffset and for the third it is a TimeSpan.

Time is assumed to be monotonically increasing, but need not follow wall-clock time.

Note that IScheduler itself does not have a type parameter. But the various Scheduler methods are generic.

3.8.1: Now

A particular scheduler's understanding of what now is.

```
DateTimeOffset Now { get; }
```

Remarks

Different schedulers can vary the actual time intervals between actions and so Now for different schedulers might not be the same. This is specifically the case when using schedulers based on virtual time, where there is a desire to speed up or eliminate entirely the intervals between when actions must execute.

For schedulers not based on virtual time, Now usually is equivalent to calling DateTime.Now.

3.8.2: Schedule(TState, Func)

Schedule an action for execution.

```
IDisposable Schedule<TState>(TState state,  
                               Func<IScheduler, TState, IDisposable> action);
```

Parameters

- state (TState) - the state to pass to the func
- action (Func<IScheduler, TState, IDisposable>) - the action to execute

Return Value

- IDisposable - represents the scheduled func

3.8.3: Schedule(TState, DateTimeOffset, Func)

Schedule an action for execution at or as soon as possible after dueTime.

```
IDisposable Schedule<TState>(TState state, DateTimeOffset dueTime,  
                               Func<IScheduler, TState, IDisposable> action);
```

Parameters

- state (TState) - the state to pass to the func
- action (Func<IScheduler, TState, IDisposable>) - the action to execute
- dueTime (DateTimeOffset) - the time offset to wait until executing the action

Return Value

- IDisposable - represents the scheduled action

Remarks

DueTime may be a positive timespan or a negative timespan. Depending on the scheduler's implementation, negative dueTime may be normalized to 0, or may mean the scheduled action takes place before other scheduled actions.

3.8.4: Schedule(TState, TimeSpan, Func)

Schedule an action for execution at or as soon as possible after dueTime.

```
IDisposable Schedule<TState>(TState state, TimeSpan dueTime,
                             Func<IScheduler, TState, IDisposable> action);
```

Parameters

- state (TState) - the state to pass to the func
- action (Func<IScheduler, TState, IDisposable>) - the action to execute
- DueTime (TimeSpan) - the time span to wait until executing the action.

Return Value

- IDisposable - represents the scheduled action.

Remarks

DueTime may be a positive timespan or a negative timespan. Depending on the scheduler's implementation, negative dueTime may be normalized to 0, or may mean the scheduled action takes place before other scheduled actions.

3.9: NewThreadScheduler

A scheduler which schedules work on a new thread.

```
public sealed class NewThreadScheduler : IScheduler {
}
public class NewThreadScheduler : IScheduler {
    public NewThreadScheduler();
    public NewThreadScheduler(Func<ThreadStart, Thread> threadFactory);
    public DateTimeOffset Now { get; }

    public IDisposable Schedule<TState>(TState state,
                                       Func<IScheduler, TState, IDisposable> action);
}
```

```

    public IDisposable Schedule<TState>(TState state,
        DateTimeOffset dueTime,
        Func<IScheduler, TState, IDisposable> action);
    public IDisposable Schedule<TState>(TState state, TimeSpan dueTime,
        Func<IScheduler, TState, IDisposable> action);
}

```

Remarks

NewThreadScheduler has no public constructors. To access a NewThreadScheduler instance, use the property System.Concurrency.Scheduler.NewThread.

NewThreadScheduler creates a new thread for each scheduled action. For example, if you call schedule twice, two different new threads will be created and each scheduled action will execute in its own thread. For heavy-duty data processing this is not optimal, but can be useful when you are sure you need a new thread (e.g. for fine grained isolation of execution).

3.9.1: NewThreadScheduler()

Constructor for a NewThreadScheduler that uses a default thread factory.

```
public NewThreadScheduler();
```

NewThreadScheduler(Func)

Constructor for a NewThreadScheduler that uses the supplied thread factory.

```
public NewThreadScheduler(Func<ThreadStart, Thread> threadFactory);
```

Parameters

- threadfactory (Func<ThreadStart, Thread>) - The func used to create the thread

Remarks

Creates a worker thread using the func supplied in the parameter.

3.9.2: Now

Now for a NewThreadScheduler is the same as DateTimeOffset.Now.

```
public DateTimeOffset Now { get; }
```

3.9.3: Schedule(TState, Func)

Schedule an action for execution on a new thread.

```

IDisposable Schedule<TState>(TState state,
    Func<IScheduler, TState, IDisposable> action);

```

Parameters

- state (TState) - the state to pass to the func
- action (Func<IScheduler, TState, IDisposable>) - the action to execute

Return Value

- `IDisposable` - represents the scheduled action

3.9.4: `Schedule(TState, DateTimeOffset, Func)`

Schedule an action for execution at or as soon as possible after `dueTime`.

```
IDisposable Schedule<TState>(TState state, DateTimeOffset dueTime,
                             Func<IScheduler, TState, IDisposable> action);
```

Parameters

- `state` (`TState`) - the state to pass to the func
- `action` (`Func<IScheduler, TState, IDisposable>`) - the action to execute
- `dueTime` (`DateTimeOffset`) - the time offset to wait until executing the action

Return Value

- `IDisposable` - represents the scheduled action

3.9.5: `Schedule(TState, TimeSpan, Func)`

Schedule an action for execution at or as soon as possible after `dueTime`.

```
IDisposable Schedule<TState>(TState state, TimeSpan dueTime,
                             Func<IScheduler, TState, IDisposable> action);
```

Parameters

- `state` (`TState`) - the state to pass to the func
- `action` (`Func<IScheduler, TState, IDisposable>`) - the action to execute
- `DueTime` (`TimeSpan`) - the time span to wait until executing the action.

Return Value

- `IDisposable` - represents the scheduled action

Remarks

The newly created thread sleeps for `dueTime`, and if the scheduled action has not been canceled (via a call to `Dispose()` on the returned `IDisposable`, then it is executed.

3.10: Scheduler

A static class providing helper methods to schedule actions on schedulers and properties providing access to specific scheduler instances.

```
public static class Scheduler {
    public static CurrentThreadScheduler CurrentThread { get; }
    public static ImmediateScheduler Immediate { get; }
    public static NewThreadScheduler NewThread { get; }
}
```

```

public static TaskPoolScheduler TaskPool { get; }
public static ThreadPoolScheduler ThreadPool { get; }

public static DateTimeOffset Now { get; }
public static TimeSpan Normalize(TimeSpan timeSpan);

public static IDisposable Schedule(
    this IScheduler scheduler, Action<Action> action);
public static IDisposable Schedule(
    this IScheduler scheduler, Action action);
public static IDisposable Schedule(
    this IScheduler scheduler, DateTimeOffset dueTime,
    Action<Action<DateTimeOffset>> action);
public static IDisposable Schedule(this IScheduler scheduler,
    DateTimeOffset dueTime, Action action);
public static IDisposable Schedule(this IScheduler scheduler,
    TimeSpan dueTime, Action<Action<TimeSpan>> action);
public static IDisposable Schedule(this IScheduler scheduler,
    TimeSpan dueTime, Action action);
public static IDisposable Schedule<TState>(
    this IScheduler scheduler,
    TState state, Action<TState, Action<TState>> action);
public static IDisposable Schedule<TState>(
    this IScheduler scheduler,
    TState state, DateTimeOffset dueTime,
    Action<TState, Action<TState, DateTimeOffset>> action);
public static IDisposable Schedule<TState>(
    this IScheduler scheduler,
    TState state, TimeSpan dueTime,
    Action<TState, Action<TState, TimeSpan>> action);
}

```

3.10.1: CurrentThread

Provides the CurrentThreadScheduler for the current thread.

```
public static CurrentThreadScheduler CurrentThread { get; }
```

3.10.2: Immediate

Provides the ImmediateScheduler for the current thread.

```
public static ImmediateScheduler Immediate { get; }
```

3.10.3: NewThread

Provides a NewThreadScheduler.

```
public static NewThreadScheduler NewThread { get; }
```

3.10.4: TaskPool

Provides the default TaskPoolScheduler.

```
public static TaskPoolScheduler TaskPool { get; }
```

3.10.5: ThreadPool

Provides the ThreadPoolScheduler.

```
public static ThreadPoolScheduler ThreadPool { get; }
```

3.10.6: Normalize

Schedulers can only execute actions now or in the future, so Normalized is used to convert a TimeSpan to be Zero or positive.

```
public static TimeSpan Normalize(TimeSpan timeSpan);
```

Parameters

- timeSpan (TimeSpan) - The timespan to normalize

Return Value

- TimeSpan - the normalized (\geq Zero) timespan

3.10.7: Now

Returns DateTimeOffset.Now.

```
public DateTimeOffset Now { get; }
```

3.10.8: Schedule

Schedules an action to be executed using the specified scheduler.

```
public static IDisposable Schedule(
    this IScheduler scheduler, Action<Action> action);
public static IDisposable Schedule(
    this IScheduler scheduler, Action action);
public static IDisposable Schedule(
    this IScheduler scheduler, DateTimeOffset dueTime,
    Action<Action<DateTimeOffset>> action);
public static IDisposable Schedule(this IScheduler scheduler,
    DateTimeOffset dueTime, Action action);
public static IDisposable Schedule(this IScheduler scheduler,
    TimeSpan dueTime, Action<Action<TimeSpan>> action);
public static IDisposable Schedule(this IScheduler scheduler,
    TimeSpan dueTime, Action action);
public static IDisposable Schedule<TState>(
    this IScheduler scheduler,
    TState state, Action<TState, Action<TState>> action);
public static IDisposable Schedule<TState>(
    this IScheduler scheduler,
    TState state, DateTimeOffset dueTime,
    Action<TState, Action<TState, DateTimeOffset>> action);
public static IDisposable Schedule<TState>(
    this IScheduler scheduler,
    TState state, TimeSpan dueTime,
    Action<TState, Action<TState, TimeSpan>> action);
```

Parameters

- scheduler (IScheduler) - The scheduler to use
- dueTime - When to execute action
- action - the action to execute

Return Value

- IDisposable - Call Dispose() on this to cancel the action (if it has not already executed)

3.11: SynchronizationContextScheduler

A scheduler which schedules work using the synchronization context.

```
public sealed class SynchronizationContextScheduler : IScheduler {
    public SynchronizationContextScheduler(
        SynchronizationContext context);
    public DateTimeOffset Now { get; }

    public IDisposable Schedule<TState>(TState state,
        Func<IScheduler, TState, IDisposable> action);
    public IDisposable Schedule<TState>(TState state,
        DateTimeOffset dueTime,
        Func<IScheduler, TState, IDisposable> action);
    public IDisposable Schedule<TState>(TState state, TimeSpan dueTime,
        Func<IScheduler, TState, IDisposable> action);
}
```

3.11.1: SynchronizationContextScheduler

Creates a new SynchronizationContextScheduler based on the supplied SynchronizationContext.

```
public SynchronizationContextScheduler(SynchronizationContext context);
```

Parameters

- context (SynchronizationContext) - the context to use for this scheduler

3.11.2: Now

Now for a SynchronizationContextScheduler is the same as System.DateTimeOffset.Now.

```
public DateTimeOffset Now { get; }
```

3.11.3: Schedule(TState, Func)

Schedule an action for execution on the synchronization context.

```
IDisposable Schedule<TState>(TState state,
    Func<IScheduler, TState, IDisposable> action);
```

Parameters

- state (TState) - the state to pass to the func
- action (Func<IScheduler, TState, IDisposable>) - the action to execute

Return Value

- IDisposable - represents the scheduled action

3.11.4: Schedule(TState, DateTimeOffset, Func)

Schedule an action for execution on the synchronization context at or as soon as possible after dueTime.

```
IDisposable Schedule<TState>(TState state, DateTimeOffset dueTime,
    Func<IScheduler, TState, IDisposable> action);
```

Parameters

- state (TState) - the state to pass to the func
- action (Func<IScheduler, TState, IDisposable>) - the action to execute
- dueTime (DateTimeOffset) - the time offset to wait until executing the action

Return Value

- IDisposable - represents the scheduled action

Schedule(TState, TimeSpan, Func)

Schedule an action for execution on the synchronization context at or as soon as possible after dueTime.

```
IDisposable Schedule<TState>(TState state, TimeSpan dueTime,
    Func<IScheduler, TState, IDisposable> action);
```

Parameters

- state (TState) - the state to pass to the func
- action (Func<IScheduler, TState, IDisposable>) - the action to execute
- DueTime (TimeSpan) - the time span to wait until executing the action

3.12: TaskPoolScheduler

A scheduler which schedules work using a task pool.

```
public sealed class TaskPoolScheduler : IScheduler {
    public TaskPoolScheduler(TaskFactory taskFactory);
    public DateTimeOffset Now { get; }
    public IDisposable Schedule<TState>(TState state,
        Func<IScheduler, TState, IDisposable> action);
    public IDisposable Schedule<TState>(TState state,
        DateTimeOffset dueTime,
        Func<IScheduler, TState, IDisposable> action);
    public IDisposable Schedule<TState>(TState state, TimeSpan dueTime,
        Func<IScheduler, TState, IDisposable> action);
}
```

Remarks

TaskPoolScheduler contains a public constructor (that takes a TaskFactory) and also a default instance is provided via Scheduler.TaskPool (in System.Reactive.Concurrency), which uses the default TaskFactory.

3.12.1: TaskPoolScheduler

Creates a new task pool scheduler based on the supplied task factory.

```
public TaskPoolScheduler(TaskFactory taskFactory);
```

Parameters

- taskFactory (TaskFactory) - The task factory to use.

Now

Now for a TaskPoolScheduler is the same as System.DateTimeOffset.Now.

```
public DateTimeOffset Now { get; }
```

3.12.2: Schedule(TState, Func)

Schedule an action for execution on the task pool.

```
IDisposable Schedule<TState>(TState state,  
                             Func<IScheduler, TState, IDisposable> action);
```

Parameters

- state (TState) - the state to pass to the func
- action (Func<IScheduler, TState, IDisposable>) - the action to execute

Return Value

- IDisposable - represents the scheduled action

Remarks

Uses TaskFactory.StartNew to schedule execution of the action. Returns a CancellationDisposable which may be used to cancel the action.

3.12.3: Schedule(TState, DateTimeOffset, Func)

Schedule an action for execution on the task pool at or as soon as possible after dueTime.

```
IDisposable Schedule<TState>(TState state, DateTimeOffset dueTime,  
                             Func<IScheduler, TState, IDisposable> action);
```

Parameters

- state (TState) - the state to pass to the func
- action (Func<IScheduler, TState, IDisposable>) - the action to execute
- dueTime (DateTimeOffset) - the time offset to wait until executing the action

Return Value

- IDisposable - represents the scheduled action

Remarks

Uses TaskFactory.StartNew to schedule execution of the action. Returns a CancellationDisposable which may be used to cancel the action.

Schedule(TState, TimeSpan, Func)

Schedule an action for execution on the task pool at or as soon as possible after dueTime.

```
IDisposable Schedule<TState>(TState state, TimeSpan dueTime,
                             Func<IScheduler, TState, IDisposable> action);
```

Parameters

- state (TState) - the state to pass to the func
- action (Func<IScheduler, TState, IDisposable>) - the action to execute
- DueTime (TimeSpan) - the time span to wait until executing the action

Remarks

Uses TaskFactory.StartNew to schedule execution of the action. Returns a CancellationDisposable which may be used to cancel the action.

3.13: ThreadPoolScheduler

A scheduler which schedules work by queuing work items for execution on the thread pool.

```
public sealed class ThreadPoolScheduler : IScheduler {
    public DateTimeOffset Now { get; }
    public IDisposable Schedule<TState>(TState state,
                                       Func<IScheduler, TState, IDisposable> action);
    public IDisposable Schedule<TState>(TState state,
                                       DateTimeOffset dueTime,
                                       Func<IScheduler, TState, IDisposable> action);
    public IDisposable Schedule<TState>(TState state, TimeSpan dueTime,
                                       Func<IScheduler, TState, IDisposable> action);
}
```

Remarks

ThreadPoolScheduler has no public constructors. To access a ThreadPoolScheduler instance, use the property System.Reactive.Concurrency.Scheduler.ThreadPool.

3.13.1: Now

Now for a ThreadPoolScheduler is the same as System.DateTimeOffset.Now.

```
public DateTimeOffset Now { get; }
```

3.13.2: Schedule(TState, Func)

Schedule an action for execution on the thread pool.

```
IDisposable Schedule<TState>(TState state,
                             Func<IScheduler, TState, IDisposable> action);
```

Parameters

- state (TState) - the state to pass to the func
- action (Func<IScheduler, TState, IDisposable>) - the action to execute

Return Value

- IDisposable - represents the scheduled action

Remarks

Uses TaskFactory.StartNew to schedule execution of the action. Returns a CancellationDisposable which may be used to cancel the action.

3.13.3: Schedule(TState, DateTimeOffset, Func)

Schedule an action for execution on the thread pool at or as soon as possible after dueTime.

```
IDisposable Schedule<TState>(TState state, DateTimeOffset dueTime,
                             Func<IScheduler, TState, IDisposable> action);
```

Parameters

- state (TState) - the state to pass to the func
- action (Func<IScheduler, TState, IDisposable>) - the action to execute
- dueTime (DateTimeOffset) - the time offset to wait until executing the action

Return Value

- IDisposable - represents the scheduled action

Schedule(TState, TimeSpan, Func)

Schedule an action for execution on the thread pool at or as soon as possible after dueTime.

```
IDisposable Schedule<TState>(TState state, TimeSpan dueTime,
                             Func<IScheduler, TState, IDisposable> action);
```

Parameters

- state (TState) - the state to pass to the func
- action (Func<IScheduler, TState, IDisposable>) - the action to execute
- DueTime (TimeSpan) - the time span to wait until executing the action

3.14: VirtualTimeScheduler

A scheduler which works with virtual (non wall-clock) time.

```
public class VirtualScheduler<TAbsolute, TRelative> : IScheduler
```

```

        where TAbsolute : global::System.IComparable<TAbsolute> {
    protected VirtualScheduler();
    public DateTimeOffset Now{ get; }
    public TAbsolute Ticks{ get; }
    public abstract TRelative FromTimeSpan(TimeSpan timeSpan);
    public abstract TAbsolute Increment(
        TAbsolute absolute, TRelative relative);

    public void Run();
    public void RunTo(TAbsolute time);
    public IDisposable Schedule(Action action);
    public IDisposable Schedule(Action action, TimeSpan dueTime);
    public IDisposable Schedule(Action action, TRelative dueTime);
    public void Sleep(TRelative ticks);
    public abstract DateTimeOffset ToDateTimeOffset(TAbsolute absolute);
}

```

Remarks

This is an abstract base class upon which derived virtual scheduler implementations may be based.

Many other scheduler types work with real time, in the sense that an action scheduled to run in five minutes will run at or very shortly after five minutes has elapsed. If one wishes to use historical data (say stock market trades for the last five years for a particular stock), then one does not want to wait five years for the result. Instead, one would like to execute multiple scheduled actions using “virtual” or “simulated” time, so that between each significant event, time can be “speeded up” to the next event, without having to wait the wall-clock time.

VirtualTimeScheduler derives from VirtualTimeSchedulerBase and has three abstract methods - FromTimeSpan, Increment and ToDateTimeOffset. Unlike VirtualTimeSchedulerBase, VirtualTimeScheduler implements IScheduler and provides the infrastructure to manage scheduled items, namely the logic for ScheduleAbsolute and GetNext.

3.14.1: VirtualTimeScheduler

A protected constructor for a virtual scheduler.

```
protected VirtualScheduler();
```

3.14.2: Now

Return the virtual time scheduler's concept of time, which is usually different from wall-clock time.

```
public DateTimeOffset Now{ get; }
```

3.14.3: Ticks

Returns the numbers of ticks that have elapsed in virtual time.

```
public TAbsolute Ticks{ get; }
```

3.14.4: FromTimeSpan

An abstract method to convert from TimeSpan to TRelative.

```
public abstract TRelative FromTimeSpan(TimeSpan timeSpan);
```

Parameters

- `timeSpan (TimeSpan)` - The `TimeSpan` to convert

Return Value

- `TRelative` - The converted value

3.14.5: Increment

An abstract method to increment time by the relative amount.

```
public abstract TAbsolute Increment(  
    TAbsolute absolute, TRelative relative);
```

Parameters

- `absolute (TAbsolute)` - the previous absolute time
- `relative (TRelative)` - the amount to increment by

Return Value

- `TAbsolute` - the new absolute time

3.14.6: Run

Runs all scheduled (and non-disposed) actions.

```
public void Run();
```

Remarks

Until many other types of schedulers, with a virtual scheduler one must manually call `run` to progress time.

3.14.7: RunTo

Runs all scheduled (and non-disposed) action up until the specific time.

```
public void RunTo(TAbsolute time);
```

Parameters

- `time (TAbsolute)` - the time up until scheduled action should be run

3.14.8: Schedule(Action)

Schedule an action for execution.

```
public IDisposable Schedule(Action action);
```

Parameters

- `action (Action)` - the action to execute

Return Value

- `IDisposable` - Call its `Dispose()` to cancel the scheduled action

3.14.9: Schedule(Action, TimeSpan)

Schedule an action for execution after the dueTime.

```
public IDisposable Schedule(Action action, TimeSpan dueTime);
```

Parameters

- action (Action) - the action to execute
- dueTime (TimeSpan) - the delay before the action should be executed

Return Value

- IDisposable - Call its Dispose() to cancel the scheduled action

3.14.10: Schedule(Action, TRelative)

Schedule an action for execution after the dueTime.

```
public IDisposable Schedule(Action action, TRelative dueTime);
```

Parameters

- action (Action) - the action to execute
- dueTime (TRelative) - the delay before the action should be executed

Return Value

- IDisposable - Call its Dispose() to cancel the scheduled action

Sleep

Sleep for the ticks in relative time.

```
public void Sleep(TRelative ticks);
```

Parameters

- ticks (TRelative) - action

3.14.11: ToDateTimeOffset

Converts from absolute to DateTimeOffset.

```
public abstract DateTimeOffset ToDateTimeOffset(TAbsolute absolute);
```

Parameters

- absolute (TAbsolute) - action

Return Value

- DateTimeOffset - action

3.15: VirtualTimeSchedulerBase

An abstract base class for schedulers using virtual time.

```

public abstract class VirtualTimeSchedulerBase<TAbsolute, TRelative>
    : IScheduler {

    protected VirtualTimeSchedulerBase();
    protected VirtualTimeSchedulerBase(TAbsolute initialClock,
        IComparer<TAbsolute> comparer);
    public TAbsolute Clock { get; protected set; }
    protected IComparer<TAbsolute> Comparer { get; }
    public bool IsEnabled { get; }
    public DateTimeOffset Now { get; }
    protected abstract TAbsolute Add(
        TAbsolute absolute, TRelative relative);
    public void AdvanceBy(TRelative time);
    public void AdvanceTo(TAbsolute time);

    protected abstract IScheduledItem<TAbsolute> GetNext();

    public IDisposable Schedule<TState>(TState state,
        Func<IScheduler, TState, IDisposable> action);
    public IDisposable Schedule<TState>(TState state,
        DateTimeOffset dueTime,
        Func<IScheduler, TState, IDisposable> action);
    public IDisposable Schedule<TState>(TState state, TimeSpan dueTime,
        Func<IScheduler, TState, IDisposable> action);
    public abstract IDisposable ScheduleAbsolute<TState>(TState state,
        TAbsolute dueTime,
        Func<IScheduler, TState, IDisposable> action);
    public IDisposable ScheduleRelative<TState>(TState state,
        TRelative dueTime,
        Func<IScheduler, TState, IDisposable> action);

    public void Start();
    public void Stop();

    protected abstract DateTimeOffset ToDateTimeOffset(
        TAbsolute absolute);
    protected abstract TRelative ToRelative(TimeSpan timeSpan);
}

```

Remarks

Virtual time is time that does not reflect wall-clock time. If one has historical information for the past hundred years (e.g. a country's economics statistics) and there is a need to pass this through an observable (e.g. to detect patterns in the statistics) then one certainly does not want to wait a hundred years for the operation to complete. Instead, one uses a virtual time scheduler, and progress virtual time through all the data points and eliminate gaps in between (which have no relevance). The `Now` property indicates the current time as the scheduler sees it. For most schedulers (e.g. those not based on `VirtualTimeSchedulerBase`), `Now` is the current wall-clock time. For those based on `VirtualTimeSchedulerBase`, `Now` is the current virtual time.

It is noted that the abstract methods are `Add`, `GetNext`, `ScheduleAbsolute`, `ScheduleAbsolute`, `ToDateTimeOffset` and `ToRelative`.

The `VirtualTimeScheduler` generic abstract class and `HistoricalSchedulerBase` abstract class derive from `VirtualTimeSchedulerBase`. `HistoricalScheduler`, which derives from `HistoricalSchedulerBase`, is a concrete class.

`VirtualTimeSchedulerBase` does not implement `IScheduler` whereas the derived types `VirtualTimeScheduler` and `HistoricalSchedulerBase` do, so usually one uses the derived types.

To implement a concrete class based directly on `VirtualTimeSchedulerBase`, one usually needs to create three classes - the class that implements `VirtualTimeSchedulerBase` and `IScheduler`, a class to represent scheduled items (implements `IScheduledItem` and `IDisposable`) and a class to implement a comparer (`IComparer`). The class that implements `VirtualTimeSchedulerBase` needs to maintain a queue (e.g. `System.Collections.Generic.SortedSet`) of scheduled items, the comparer sorts scheduled items based on their due time and is the comparer for the queue. The implementation of `ScheduleAbsolute` creates a new scheduled item and adds it to the queue. The implementation of `GetNext` returns the top of the queue and importantly removes it from the queue.

3.15.1: ScheduleAbsolute

Schedule an action with an absolute time.

```
public abstract IDisposable ScheduleAbsolute<TState>(TState state,
    TAbsolute dueTime,
    Func<IScheduler, TState, IDisposable> action);
```

Parameters

- state (`TState`) - The state to pass to the action
- dueTime (`TAbsolute`) - When to execute the action
- action (`Func<IScheduler, TState, IDisposable>`) - The action to execute

Return Value

- `IDisposable` - Used to cancel scheduled action (if not already executed)

3.15.2: ScheduleRelative

Schedule an action with a relative time.

```
public IDisposable ScheduleRelative<TState>(TState state,
    TRelative dueTime,
    Func<IScheduler, TState, IDisposable> action);
```

Parameters

- state (`TState`) - The state to pass to the action
- dueTime (`TRelative`) - When to execute the action

- action (Func<IScheduler, TState, IDisposable>) - The action to execute

Return Value

- IDisposable - Used to cancel scheduled action (if not already executed)

3.15.3: Start

Starts the virtual scheduler.

```
public void Start();
```

Remarks

With schedulers not based on virtual time, the scheduled actions are executed according to scheduled time. With virtual time schedulers, there is a need to call this method so the scheduler knows when its time begins.

3.15.4: Stop

Stops the virtual scheduler.

```
public void Stop();
```

3.15.5: ToDateTimeOffset

Converts an absolute time to DateTimeOffset.

```
protected abstract DateTimeOffset ToDateTimeOffset(  
    TAbsolute absolute);
```

Parameters

- absolute (TAbsolute) - The absolute time value

Return value

- DateTimeOffset - The absolute time represented as a DateTimeOffset

3.15.6: ToRelative

Converts a timespan to relative time.

```
protected abstract TRelative ToRelative(TimeSpan timeSpan);
```

Parameters

- timeSpan (TimeSpan) - The TimeSpan to convert

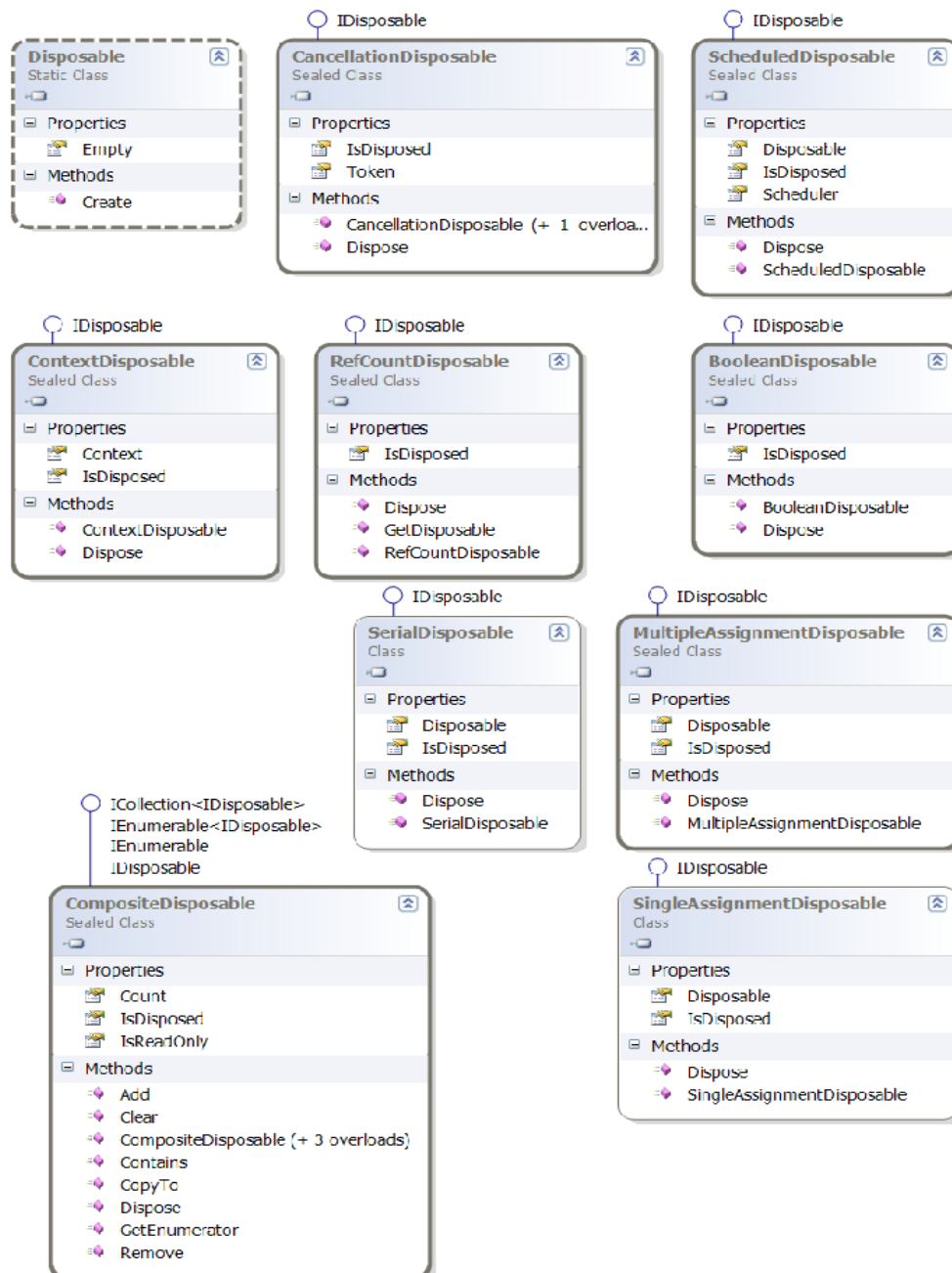
Return Value

- TRelative - The result of the conversion

4: System.Reactive.Disposables

4.1: Overview

System.Reactive.Disposables has the following types (most based on System.IDisposable):



Remarks

Where Rx needs to allow client code to unsubscribe (from an `Observable`) or disconnect (from a `System.Reactive.Subject.IConnectableObservable`) it uses the Disposable pattern.

The subscription (`Observable.Subscribe`) or connection (`IConnectableObservable.Connect`) or scheduling of an action (`IScheduler.Schedule`) call returns an `IDisposable`, which represents the subscription / connection / scheduling of an action, and when the client is no longer interested, it may call the disposable's `Dispose()` method.

The benefits of using `IDisposable` instead of creating additional custom types to represent subscriptions/connections is that it is simpler, application developers already understand them and they may be used with C#'s `using` construct.

The types in the `System.Disposables` namespace provide flexibility in the handling of disposables. One allows you to create an implementation of `IDisposable`, whose `Dispose()` method just calls a provided action, and all the others derive from `IDisposable`. There is also an enum for assignment behavior.

Note all disposables in this namespace are sealed.

4.2: BooleanDisposable

A disposable that indicates whether its `Dispose()` method has been called.

```
public sealed class BooleanDisposable : IDisposable {
    public BooleanDisposable();
    public bool IsDisposed { get; }
    public void Dispose();
}
```

Remarks

Initially, `IsDisposed` is set to false (`Dispose()` has not been called).

BooleanDisposable

Constructor to create a `BooleanDisposable`.

```
public BooleanDisposable();
```

IsDisposed

A property indicating whether the `Dispose()` method for this disposable has been called.

```
public bool IsDisposed { get; }
```

Dispose

Dispose of this disposable.

```
public void Dispose();
```

4.3: CancellationDisposable

A disposable with a cancellation token.

```
public sealed class CancellationDisposable : IDisposable
{
    public CancellationDisposable();
    public CancellationDisposable(CancellationTokenSource cts);
    public bool IsDisposed { get; }
    public CancellationToken Token { get; }
    public void Dispose();
}
```

Remarks

A cancellation token (.NET 4's System.Threading.CancellationToken structure) is a structure used to indicate that operations should be canceled and to execute register delegates when this happens. When Dispose is called, the cancellation token is canceled.

Note that the cancellation token is made available via a getter property only. It is constructed inside of CancellationDisposable and passed out to client code via this property.

To construct a cancellation token, a CancellationTokenSource instance is used. Client code may create an instance and pass in in via one of the constructors, or else CancellationDisposable can create one internally.

4.3.1: CancellationDisposable()

Creates a new CancellationDisposable (with an internal CancellationTokenSource).

```
public CancellationDisposable();
```

4.3.2: CancellationDisposable(CancellationTokenSource)

Creates a new CancellationDisposable with a custom CancellationTokenSource.

```
public CancellationDisposable(CancellationTokenSource cts);
```

Parameters

- cts (CancellationTokenSource) - the CancellationTokenSource

4.3.3: IsDisposed

A boolean indicating whether the disposable has been disposed.

```
public bool IsDisposed { get; }
```

4.3.4: Token

The cancellation token.

```
public CancellationToken Token{ get; }
```

4.3.5: Dispose

Initiates cancellation of the token.

```
public void Dispose();
```

4.4: CompositeDisposable

A group of disposables that act as one.

```
class CompositeDisposable : ICollection<IDisposable>,
    IEnumerable<IDisposable>, IEnumerable, IDisposable {

    public CompositeDisposable();
    public CompositeDisposable(IEnumerable<IDisposable> disposables);
    public CompositeDisposable(int capacity);
    public CompositeDisposable(params IDisposable[] disposables);

    public int Count{get;}
    public bool IsDisposed { get; }
    public bool IsReadOnly{get;}
    public void Add(IDisposable item);
    public void Clear();
    public bool Contains(IDisposable disposable);
    public void CopyTo(IDisposable[] array, int arrayIndex);
    public void Dispose();
    public IEnumerator<IDisposable> GetEnumerator();
    public bool Remove(IDisposable disposable);
}
```

Remarks

CompositeDisposable maintains a list of disposables, and when CompositeDisposable.Dispose() method is called, it in turn calls the Dispose() methods of all the disposables in its list.

4.4.1: CompositeDisposable

Constructors to create a composite disposable and, for some, to initialize its list to those supplied in the parameter.

```
public CompositeDisposable();
public CompositeDisposable(IEnumerable<IDisposable> disposables);
public CompositeDisposable(int capacity);
public CompositeDisposable(params IDisposable[] disposables);
```

Parameters

- disposables - the initial list of disposables
- capacity - the number of disposables that may be in the composite

4.4.2: Dispose

Calls Dispose() method of all the Disposables in the composite's list.

```
public void Dispose();
```

4.5: ContextDisposable

A thread-affine IDisposable which uses a synchronization context for disposal of an underlying disposable.

```
public sealed class ContextDisposable : IDisposable
{
    public ContextDisposable(SynchronizationContext context,
                            IDisposable disposable);

    public SynchronizationContext Context { get; }
    public bool IsDisposed { get; }
    public void Dispose();
}
```

4.5.1: ContextDisposable

The constructor for a ContextDisposable.

```
public ContextDisposable(SynchronizationContext context,
                        IDisposable disposable);
```

Parameters

- context (SynchronizationContext) - The context to use when disposing
- disposable (IDisposable) - The underlying disposable

4.5.2: Context

A getter property which returns the synchronization property.

```
public SynchronizationContext Context { get; }
```

4.5.3: IsDisposed

A boolean indicating whether the disposable has been disposed.

```
public bool IsDisposed { get; }
```

4.5.4: Dispose

Disposes of the underlying disposable using the supplied synchronization context.

```
public void Dispose();
```

4.6: Disposable

Provides a static method to create a Disposable based on a supplied Action, and a property which returns a Disposable whose Dispose() is empty.

```
public static class Disposable
{
    public static IDisposable Empty { get; }
    public static IDisposable Create(Action dispose);
}
```

4.6.1: Empty

Returns an IDisposable whose Dispose() method is a no-op.

```
public static IDisposable Empty { get; }
```

4.6.2: Create

Creates a Disposable, whose Dispose() method calls the supplied Action.

```
public static IDisposable Create(Action dispose);
```

Parameters

- dispose (Action) - The action to call during Dispose()

Return Value

- IDisposable - used to dispose of the disposable

4.7: MultipleAssignmentDisposable

MultipleAssignmentDisposable is a wrapper around an underlying disposable which can be changed for another underlying disposable.

```
public sealed class MultipleAssignmentDisposable : IDisposable {  
    public MutableDisposable();  
    public IDisposable Disposable { get; set; }  
    public bool IsDisposed { get; }  
    public void Dispose();  
}
```

4.7.1: MultipleAssignmentDisposable()

Constructor for MultipleAssignmentDisposable which initially has no wrapped disposable.

```
public MultipleAssignmentDisposable();
```

Remarks

The Disposable property setter may be called multiple times and in each case the new disposable property replaces the previous, but importantly, the Dispose() method of the previous is not called. Later, when the MultipleAssignmentDisposable.Dispose() is called, then the underlying disposable's Dispose() is called. Hence with this value, only the most recent set underlying disposable will have its Dispose() method called at some point.

4.7.2: Disposable

A property with a public getter and setter for the wrapped disposable.

```
public IDisposable Disposable { get; set; }
```

This is the wrapped disposable. When a new value is set and previously it was non-null, then Dispose is called for the previous wrapped disposable - regardless of whether MultipleAssignmentDisposable.Dispose() has been called.

If MultipleAssignmentDisposable.Dispose() has been called, then each time a non-null value is set using the setter, the new wrapped disposable will have its Dispose method called immediately.

4.7.3: Dispose

Disposes of the currently wrapped disposable (if any) and importantly, any future wrapped disposables.

```
public void Dispose();
```

4.8: RefCountDisposable

A disposable which wraps an underlying disposable, manages a reference count and only disposes of the underlying disposable when the reference count is 0.

```
public sealed class RefCountDisposable : IDisposable {  
    public RefCountDisposable(IDisposable underlyingDisposable);  
    public bool IsDisposed { get; }  
    public void Dispose();  
    public IDisposable GetDisposable();  
}
```

Remarks

For reference counting to work, there needs to be a way to increment the reference count and a way to decrement it. The reference count is incremented each time `GetDisposable()` is called, and decremented each time an `IDisposable` returned from `GetDisposable()` is disposed.

The reference count starts at 0, so if one created a new `RefCountDisposable` for an underlying disposable and immediately called `Dispose()` for the new `RefCountDisposable`, then `Dispose()` would be called for the underlying disposable immediately.

However, if one create a new `RefCountDisposable`, then calls its `GetDisposable`, and then calls the `RefCountDisposable`'s `Dispose()`, then the underlying disposable's `Dispose()` would not be called immediately. It will be later, when `Dispose()` is called for the `IDisposable` returned by `GetDisposable`.

4.8.1: RefCountDisposable

Creates a new `RefCountDisposable` that wraps an underlying disposable.

```
public RefCountDisposable(IDisposable underlyingDisposable);
```

Parameters

- `underlyingDisposable` (`IDisposable`) - The disposable to wrap.

4.8.2: IsDisposed

A boolean indicating whether the disposable has been disposed.

```
public bool IsDisposed { get; }
```

4.8.3: Dispose

Dispose of the underlying disposable when the reference is zero.

```
public void Dispose();
```

4.8.4: GetDisposable

Increments the reference count.

```
public IDisposable GetDisposable();
```

Return Value

- `IDisposable` – A “reference” disposable which must have its `Dispose()` method invoked (either before or after `Dispose` for the `RefCountDisposable` itself is called) before the underlying disposable called be disposed of.

Remarks

The reference count is incremented when this method is called. When `Dispose()` is called on the returned `IDisposable`, the reference count is decremented.

4.9: ScheduledDisposable

A wrapper for an underlying disposable which calls its `Dispose()` using the supplied scheduler.

```
public class ScheduledDisposable : IDisposable {
    public ScheduledDisposable(
        IScheduler scheduler, IDisposable disposable);
    public IDisposable Disposable {get;}
    public IScheduler Scheduler {get;}
    public void Dispose();
}
```

Remarks

The `Dispose()` call for the wrapped disposable happens via the scheduler, not (necessarily) on the thread where the call to `ScheduledDisposable.Dispose()` happened.

4.9.1: ScheduledDisposable

Constructor for a `ScheduledDisposable`.

```
public ScheduledDisposable(
    IScheduler scheduler, IDisposable disposable);
```

Parameters

- `scheduler` (`IScheduler`) – The scheduler to use for disposing of the underlying disposable.
- `disposable` (`IDisposable`) – The underlying disposable that is to be wrapper by the `ScheduledDisposable`.

4.9.2: Disposable

A getter property which returns the underlying disposable.

```
public IDisposable Disposable{get;}
```

4.9.3: Scheduler

A getter property which returns the scheduler.

```
public IScheduler Scheduler{get;}
```

4.9.4: IsDisposed

A boolean indicating whether the disposable has been disposed.

```
public bool IsDisposed { get; }
```

4.9.5: Dispose

Calls the Dispose method of the wrapped disposable using the provided scheduler.

```
public void Dispose();
```

4.10: SerialDisposable

SerialDisposable is a wrapper around an underlying disposable which can be changed for another underlying disposable.

```
public sealed class SerialDisposable : IDisposable {  
    public SerialDisposable();  
    public IDisposable Disposable{ get; set; }  
    public bool IsDisposed { get; }  
    public void Dispose();  
}
```

4.10.1: SerialDisposable()

Constructor for SerialDisposable which initially has no wrapped disposable.

```
public SerialDisposable();
```

Remarks

The Disposable property setter may be called multiple times and in each case the new disposable property replaces the previous, and the Dispose() method of the previous disposable is called. Later, when the SerialDisposable.Dispose() is called, then the underlying disposable's Dispose() is called. Hence with this value, all underlying disposables will have their Dispose() method called at some point.

4.10.2: Disposable

A property with a public getter and setter for the wrapped disposable.

```
public IDisposable Disposable{ get; set; }
```

This is the wrapped disposable. When a new value is set and previously it was non-null, then Dispose is called for the previous wrapped disposable - regardless of whether SerialDisposable.Dispose() has been called.

If SerialDisposable.Dispose() has been called, then each time a non-null value is set using the setter, the new wrapped disposable will have its Dispose method

called immediately.

4.10.3: Dispose

Disposes of the currently wrapped disposable (if any) and importantly, any future wrapped disposables.

```
public void Dispose();
```

4.11: SingleAssignmentDisposable

SingleAssignmentDisposable is a wrapper around an underlying disposable which can be changed for another underlying disposable.

```
public class SingleAssignmentDisposable : IDisposable {
    public SingleAssignmentDisposable();
    public IDisposable Disposable { get; set; }
    public bool IsDisposed { get; }
    public void Dispose();
}
```

4.11.1: SingleAssignmentDisposable()

Constructor for SingleAssignmentDisposable which initially has no wrapped disposable.

```
public SingleAssignmentDisposable();
```

Remarks

A SingleAssignment instance can only be assigned one underlying disposable (the setter for the SingleAssignment.Disposable property can only be called once). If it is called a second time, an InvalidOperationException is raised with the message set to "Disposable has already been assigned".

4.11.2: Disposable

A property with a public getter and setter for the wrapped disposable.

```
public IDisposable Disposable { get; set; }
```

This is the wrapped disposable. When a new value is set and previously it was non-null, then Dispose is called for the previous wrapped disposable - regardless of whether SingleAssignmentDisposable.Dispose() has been called.

If SingleAssignmentDisposable.Dispose() has been called, then each time a non-null value is set using the setter, the new wrapped disposable will have its Dispose method called immediately.

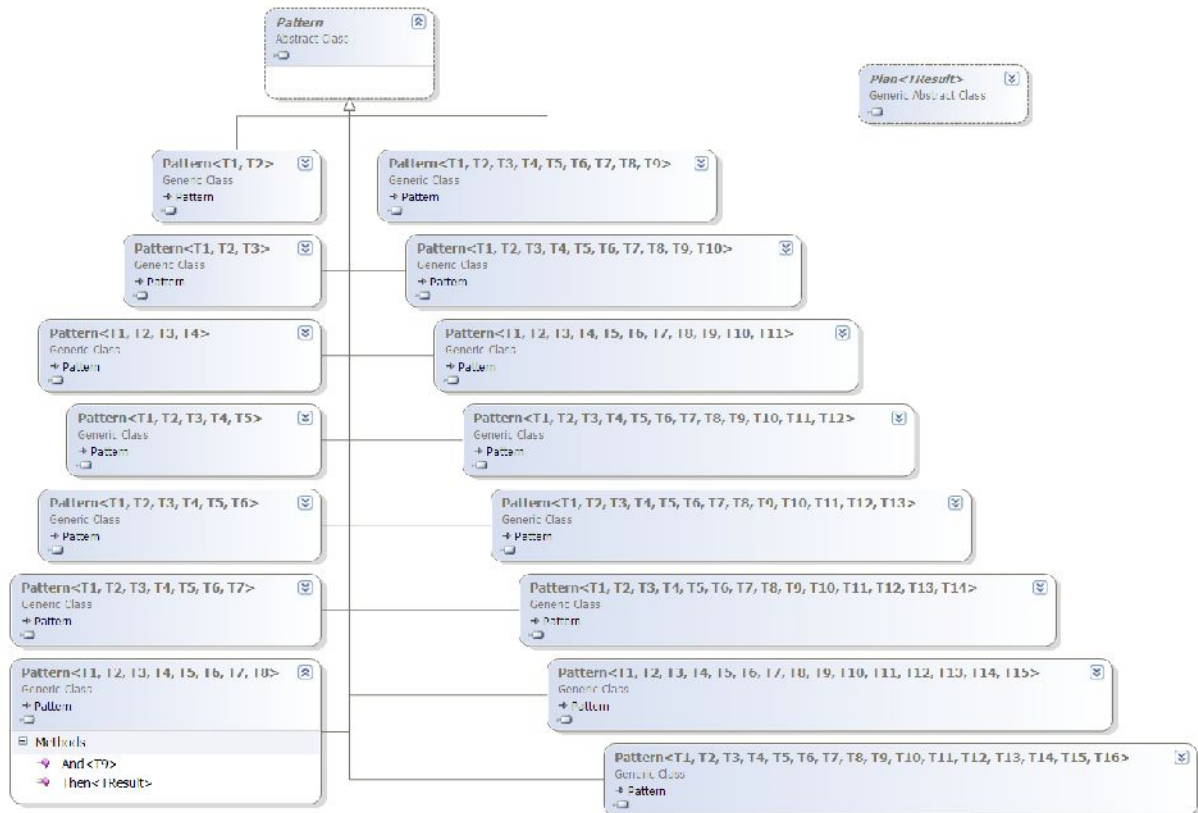
4.11.3: Dispose

Disposes of the currently wrapped disposable (if any) and importantly, any future wrapped disposables.

```
public void Dispose();
```

5: System.Reactive.Joins

The System.Reactive.Joins namespace (in System.Reactive.dll) has the following types:



Remarks

A pattern is a number of source observables that can be used to form a plan. A plan is a number of source observables and a selector function that projects elements from the source observables into a result.

The Pattern class is abstract and is the base for all other Pattern types.

There are a variety of Pattern types accepting a different number of type parameters - with between 2 and 16 type parameters. Each of these derive from Pattern, but not from each other. How they all work is the same. The only difference is that they manage between 2 and 16 observables.

5.1: Pattern

An abstract base class for all patterns.

```
public abstract class Pattern
{
}
```

Remarks

This class has no public interface. Its sole purpose is to serve as a common base for the Pattern classes that accept type parameters.

5.2: Pattern<T1, T2>

Represents a pattern that can be Anded to create another pattern and combined with a selector to form a plan.

```
public class Pattern<T1, T2> : Pattern {
    public Pattern<T1, T2, T3> And<T3>(IObservable<T3> other);
    public Plan<TResult> Then<TResult>(Func<T1, T2, TResult> selector);
}
```

Remarks

There are Pattern classes accepting varying numbers of type parameters, and all function the same as this one.

When one has a pattern, a new pattern may be constructed using And. To create a pattern from scratch, use System.Reactive.Linq.Observable.And().

5.2.1: And

Creates another Pattern instance by adding an additional observable to those already in the pattern.

```
public Pattern<T1, T2, T3> And<T3>(IObservable<T3> other);
```

Parameters

- other () - The additional observable

Return Value

- Pattern<T1, T2, T3> - The Pattern combining all three observables

Remarks

Note that the type parameters for all three observables may be different.

There is not public interface to access the constituent observables with a pattern.

5.2.2: Then

Constructs a plan based on the pattern and a supplied selector function.

```
public Plan<TResult> Then<TResult>(Func<T1, T2, TResult> selector);
```

Parameters

- selector (Func<T1, T2, TResult>) - A selector function that converts elements from each pattern observable to an output element

Return Value

- Plan<TResult> - A plan that when run with Observable.When processes the observables through the selector

5.3: Plan

An abstract class representing a plan.

```
public abstract class Plan<TResult>  
{  
}
```

Remarks

A plan can be created via System.Reactive.Linq.Observable.Then() or Pattern<T1, T2>.Then().

A plan can be executed via System.Reactive.Linq.Observable.When().

6: System.Reactive.Linq

6.1: Overview

Rx (System.Reactive.dll) adds the following types to System.Reactive.Linq:

The image shows a screenshot of Visual Studio with the `Observable` class expanded to show its methods. The methods are listed in two columns. The left column contains methods from `Observable` and `IObservable`, while the right column contains methods from `IObservable` and `IObservable<T>`.

Observable Methods (Left Column):

- Aggregate<TSource, TAccumulate> (+ 1 overload)
- All<TSource>
- Amb<TSource> (+ 2 overloads)
- And<TLeft, TRight>
- Any<TSource> (+ 1 overload)
- AsObservable<TSource>
- Average (+ 3 overloads)
- Buffer<TSource, TBufferOpening, TBufferClosing> (+ 9 overloads)
- Cast<TResult>
- Catch<TSource, TException> (+ 3 overloads)
- CombineLatest<TFirst, TSecond, TResult>
- Concat<TSource> (+ 3 overloads)
- Contains<TSource> (+ 1 overload)
- Count<TSource>
- Create<TSource> (+ 1 overload)
- DefaultIfEmpty<TSource> (+ 1 overload)
- Defer<TValues>
- Delay<TSource> (+ 3 overloads)
- Dematerialize<TSource>
- Distinct<TSource> (+ 3 overloads)
- DistinctUntilChanged<TSource, TKey> (+ 3 overloads)
- Do<TSource> (+ 4 overloads)
- ElementAt<TSource>
- ElementAtOrDefault<TSource>
- Empty<TResult> (+ 1 overload)
- Finally<TSource>
- First<TSource> (+ 1 overload)
- FirstOrDefault<TSource> (+ 1 overload)
- ForEach<TSource> (+ 5 overloads)
- FromAsyncPattern<TResult> (+ 29 overloads)
- FromEvent<TDelegate, TEventArgs> (+ 3 overloads)
- FromEventPattern (+ 7 overloads)
- Generate<TState, TResult> (+ 5 overloads)
- GetEnumerator<TSource>
- GroupBy<TSource, TKey, TElement> (+ 3 overloads)
- GroupByUntil<TSource, TKey, TElement, TDuration> (+ 3 overloads)
- GroupJoin<TLeft, TRight, TLeftDuration, TRightDuration, TResult>
- IgnoreElements<TSource>
- Interval (+ 1 overload)
- Join<TLeft, TRight, TLeftDuration, TRightDuration, TResult>
- Last<TSource> (+ 1 overload)
- LastOrDefault<TSource> (+ 1 overload)
- Lates<TSource>
- LongCount<TSource>
- Materialize<TSource>
- Max<TSource> (+ 11 overloads)
- MaxBy<TSource, TKey> (+ 1 overload)
- Merge<TSource> (+ 9 overloads)
- Min<TSource> (+ 11 overloads)
- MinBy<TSource, TKey> (+ 1 overload)
- MostRecent<TSource>
- Multicast<TSource, TResult> (+ 1 overload)
- Never<TResult>
- Next<TSource>
- ObserveOn<TSource> (+ 1 overload)
- OfType<TResult>
- OnErrorResumeNext<TSource> (+ 2 overloads)
- Publish<TSource> (+ 3 overloads)
- PublishLast<TSource> (+ 1 overload)
- Range (+ 1 overload)
- RefCount<TSource>

IObservable Methods (Right Column):

- RefCount<TSource> (+ 5 overloads)
- Replay<TSource> (+ 15 overloads)
- Retry<TSource> (+ 1 overload)
- Return<TResult> (+ 1 overload)
- Sample<TSource, TSample> (+ 7 overloads)
- Scan<TSource, TAccumulate> (+ 1 overload)
- Select<TSource, TResult> (+ 1 overload)
- SelectMany<TSource, IOther> (+ 5 overloads)
- SequenceEqual<TSource> (+ 1 overload)
- Single<TSource> (+ 1 overload)
- SingleOrDefault<TSource> (+ 1 overload)
- Skip<TSource>
- SkipLast<TSource>
- SkipUntil<TSource, TOther>
- SkipWhile<TSource> (+ 1 overload)
- Start<TSource> (+ 3 overloads)
- StartWith<TSource> (+ 1 overload)
- Subscribe<TSource> (+ 1 overload)
- SubscribeOn<TSource> (+ 1 overload)
- Sum (+ 9 overloads)
- Switch<TSource>
- Synchronize<TSource> (+ 1 overload)
- Take<TSource>
- TakeLast<TSource>
- TakeUntil<TSource, TOther>
- TakeWhile<TSource> (+ 1 overload)
- Then<TSource, TResult>
- Throttle<TSource> (+ 1 overload)
- Throw<TResult> (+ 1 overload)
- TimeInterval<TSource> (+ 1 overload)
- Timeout<TSource> (+ 7 overloads)
- Timer (+ 7 overloads)
- Timestamp<TSource> (+ 1 overload)
- ToArray<TSource>
- ToAsync<TResult> (+ 67 overloads)
- ToDictionary<TSource, TKey, TElement> (+ 3 overloads)
- ToEnumerable<TSource>
- ToEvent (+ 1 overload)
- ToEventPattern<TEventArgs>
- ToList<TSource>
- ToLookup<TSource, TKey, TElement> (+ 3 overloads)
- ToObservable<TSource> (+ 1 overload)
- Using<TSource, TResource>
- When<TResult> (+ 1 overload)
- Where<TSource> (+ 1 overload)
- Window<TSource, TWindowOpening, TWindowClosing> (+ 9 overloads)
- Zip<TFirst, TSecond, TResult> (+ 1 overload)

The image shows a screenshot of Visual Studio showing the `IGroupedObservable` interface. The interface is defined as `IGroupedObservable<TKey, TElement>` and inherits from `IObservable<TElement>`. It has a single property `Key` of type `TKey`.

6.2: IGroupedObservable

Used to group elements according to a key.

```
public interface IGroupedObservable<out TKey, out TElement> :
    IObservable<TElement>
{
    TKey Key {get;}
}
```

Remarks

IGroupedObservable is used in conjunction with the System.Linq.Observable's GroupBy and GroupByUntil operators.

The role of IGroupedObservable for an observable is similar to that of IGrouping for an IEnumerable.

6.2.1: Key

The key property.

```
TKey Key {get;}
```

6.3: Observable

Observable is a static class providing the Linq operators to work with IObservable.

```
public static class Observable
{
}
```

6.3.1: Aggregate

Passes elements from a sequence through an accumulator function and returns an observable with the result as its single element.

```
public static IObservable<TSource>
    Aggregate<TSource>(
        this IObservable<TSource> source,
        Func<TSource, TSource, TSource> accumulator);
```

Parameters

- source (IObservable<TSource>) - The sequence to accumulate
- accumulator (Func<TSource, TSource, TSource>) - The function to used for accumulation

Return Value

- IObservable<TSource> - A sequence with one element, which is the result of the accumulation

Remarks

The accumulator function returns TSource, and takes in two parameters, the merged value so far and the next element. The merged value so far and the next element are “accumulator” type instance based on the functionality needed, and the result returned.

6.3.2: Aggregate

Passes elements from a sequence through an accumulator function (which starts with a seed) and returns an observable with the result as its single element.

```
public static IObservable<TAccumulate>
    Aggregate<TSource, TAccumulate>(
        this IObservable<TSource> source,
        TAccumulate seed,
        Func<TAccumulate, TSource, TAccumulate> accumulator);
```

Parameters

- source (IObservable<TSource>) - The sequence to accumulate
- accumulator (Func<TAccumulate, TSource, TAccumulate>) - The accumulator function
- seed (TAccumulate) - The initial value

Return Value

- IObservable<TSource> - A sequence with one element, which is the result of the accumulation

Remarks

The accumulator function returns TSource, and takes in two parameters, the merged value so far and the next element. The merged value so far and the next element are “accumulator” type instances based on the functionality needed, and the result returned. The merged value initial value is set to seed.

6.3.3: All

Passes all elements in a sequence through a boolean predicate and returns whether all passed in an observable of type boolean with the result as its single element.

```
public static IObservable<bool> All<TSource>(
    this IObservable<TSource> source, Func<TSource, bool> predicate);
```

Parameters

- source (IObservable<TSource>) - The sequence whose elements are to be passed to the predicate
- predicate (Func<TSource, bool>) - A boolean function which is passed each element in turn

Return Value

- `IObservable<bool>` - An observable with one boolean element, the result of whether all elements in the input observable passed through the predicate successfully (i.e. returned true)

Remarks

All returns true if all elements in the input sequence, when passed in turn to the predicate, result in true being returned. All returns false if any predicate calls return false.

6.3.4: Amb

From a number of observables, return the first to be active.

```
public static IObservable<TSource> Amb<TSource>(
    params IObservable<TSource>[] sources);
public static IObservable<TSource> Amb<TSource>(
    this IEnumerable<IObservable<TSource>> sources);
public static IObservable<TSource> Amb<TSource>(
    this IObservable<TSource> first,
    IObservable<TSource> second);
```

Parameters

- `sources (params IObservable<TSource>)` - The source observables
- `sources (IEnumerable< IObservable<TSource> >)` - The source observables
- `first (IObservable<TSource>)` - The first source observable
- `second (IObservable<TSource>)` - The second source observable

Return Value

- `IObservable<TSource>` - The first to be active

Remarks

`Amb` (ambiguous) is used when one is interested in the first observable from among many to be active. One is not concerned which it is, apart from the fact that it is active.

It is noted that all source observables and the result observable all have the same type parameter for the element type.

6.3.5: And

Returns a pattern contains both observable sources.

```
public static Pattern<TLeft, TRight> And<TLeft, TRight>(
    this IObservable<TLeft> left,
    IObservable<TRight> right);
```

Parameters

- left (IObservable<TLeft>) - The left source
- right (IObservable<TLeft>) - The right source

Return Value

- Pattern<TLeft, TRight> - the matched pattern

Remarks

A pattern is a match when all sequences in the pattern have elements available.

Note that left and right have distinct type parameters.

This method creates a System.Reactive.Joins.Pattern.

To create a plan from a pattern, use System.Reactive.Joins.Pattern.Then(). To create a plan from an observable, use System.Reactive.Linq.Then().

To execute one or more plans, use System.Reactive.Linq.Observable.When().

To append additional observables to the pattern, call Pattern.And().

6.3.6: Any(this IObservable<TSource>)

Returns an observable with a single boolean element stating whether the source observable has elements

```
public static IObservable<bool> Any<TSource>(
    this IObservable<TSource> source);
```

Parameters

- source (this IObservable<TSource>) - The source to be examined

Return Value

- IObservable<bool> - An observable with one bool element stating whether the source has any elements

Remarks

This operator waits until it detects either one OnNext call or the OnCompleted call, before returning its result.

6.3.7: Any(this IObservable<TSource>, Func<TSource, bool>)

Returns an observable with a single boolean element stating whether the source observable has elements that matches a predicate

```
public static IObservable<bool> Any<TSource>(
    this IObservable<TSource> source,
    Func<TSource, bool> predicate);
```

Parameters

- source (this IObservable<TSource>) - The source to be examined
- predicate (Func<TSource, bool>) - A condition that the element must satisfy in order for Any to return true

Return Value

- IObservable<bool> - An observable with one bool element stating whether the source has any elements that match the predicate

Remarks

This operator waits until it detects either one OnNext call that matches the predicate (so it can return true) or the OnCompleted call before such a match (so it can return false), before returning its result.

6.3.8: AsObservable

Anonymizes an observable (hides its type by using a wrapper).

```
public static IObservable<TSource> AsObservable<TSource>(
    this IObservable<TSource> source);
```

Parameters

- source (IObservable<TSource>) - The source observable, whose type information is to be hidden

Return Value

- IObservable<TSource> - The anonymous observable

Remarks

This operator may be useful when a developer returns an instance of one type that implements IObservable in one version of a library, but plans to return an instance of a different type in future, and there is a desire to definitely ensure client code cannot cast the initial version from IObservable to its native type, which would cause problems in future.

6.3.9: Average

Determines an average of a sequence with numeric elements.

```
public static IObservable<decimal?> Average(
    this IObservable<decimal?> source);
public static IObservable<decimal> Average(
    this IObservable<decimal> source);
public static IObservable<double?> Average(
    this IObservable<double?> source);
public static IObservable<double> Average(
    this IObservable<double> source);
public static IObservable<float?> Average(
    this IObservable<float?> source);
public static IObservable<float> Average(
```

```

        this IObservable<float> source);
public static IObservable<double?> Average(
        this IObservable<int?> source);
public static IObservable<double> Average(
        this IObservable<int> source);
public static IObservable<double?> Average(
        this IObservable<long?> source);
public static IObservable<double> Average(
        this IObservable<long> source);

```

Parameters

- source (IObservable< *numeric type* >) - The source sequence

Return Value

- IObservable < *numeric type* > - The result in a single element observable

Remarks

This operator calculates the average of a sequence of numeric types.

Note the type of the returned sequence can be either float, float?, double, double?, decimal or decimal?. The type of the source sequence can be decimal, decimal?, double, double?, float, float?, int, int?, long, or long?.

6.3.10: Buffer

Sub-divides a sequence into a set of windows - based on time, a condition or a count - and returns each window as a single buffer when the window completes.

```

public static IObservable<IList<TSource>>
    Buffer<TSource, TBufferClosing>(
        this IObservable<TSource> source,
        Func<IObservable<TBufferClosing>> bufferClosingSelector);
public static IObservable<IList<TSource>> Buffer<TSource>(
        this IObservable<TSource> source, int count);
public static IObservable<IList<TSource>> Buffer<TSource>(
        this IObservable<TSource> source, TimeSpan timeSpan);
public static IObservable<IList<TSource>> Buffer<TSource>(
        this IObservable<TSource> source, int count, int skip);
public static IObservable<IList<TSource>>
    Buffer<TSource, TBufferOpening, TBufferClosing>(
        this IObservable<TSource> source,
        IObservable<TBufferOpening> bufferOpenings,
        Func<TBufferOpening, IObservable<TBufferClosing>>
            bufferClosingSelector);
public static IObservable<IList<TSource>> Buffer<TSource>(
        this IObservable<TSource> source, TimeSpan timeSpan, int count);
public static IObservable<IList<TSource>> Buffer<TSource>(
        this IObservable<TSource> source, TimeSpan timeSpan,
        IScheduler scheduler);
public static IObservable<IList<TSource>> Buffer<TSource>(
        this IObservable<TSource> source, TimeSpan timeSpan,
        TimeSpan timeShift);
public static IObservable<IList<TSource>> Buffer<TSource>(
        this IObservable<TSource> source, TimeSpan timeSpan,
        int count, IScheduler scheduler);
public static IObservable<IList<TSource>> Buffer<TSource>(

```

```
this IObservable<TSource> source, TimeSpan timeSpan,  
TimeSpan timeShift, IScheduler scheduler);
```

Parameters

- source (this IObservable<TSource>) - The source sequence
- bufferClosingSelector (Func<IObservable<TBufferClosing>>) - Emits an element when the window is to close
- count (int) - Number of elements to be placed in buffer before window closes
- timeSpan (TimeSpan) - The timespan to wait before closing the window
- skip (int) - Number of elements to skip between closing one window and opening another
- bufferOpenings (IObservable<TBufferOpening>) - When to open a window
- bufferClosingSelector (Func<TBufferOpening, IObservable<TBufferClosing>>) - Emits an element when the window is to close
- timeShift (TimeSpan) - The time shift
- scheduler (IScheduler) - The scheduler to use for the buffering workload

Return Value

- IObservable<IList<TSource>> - An observable of buffers, where each buffer consists of all source elements within the window

Remarks

There is an operator similar to Buffer called Window and it is important to understand the difference between them. Both are windowing operators in the sense they apply a window (a sub-division) to the source sequence (based on count, a condition, time, etc.). The difference is that Buffer waits until its window is closed and then emits an IList with the elements in the window, whereas Window emits an IObservable on the window itself (Window is an IObservable<IObservable<TSource>, whereas Buffer returns an IObservable<IList<TSource>>). In other words, the Window operator does not need to have any source elements before it can return its IObservable, whereas the Buffer operator needs to have received all its source elements so it can fill the IList and return it.

When the Buffer operator detects OnCompleted from the source, it closes the final window and returns it, and then returns OnCompleted itself. So a buffer operator told to return a window of count=4, would return OnNext with an IList

of 4 elements, followed by `OnNext` with an `IList` of 4 elements, followed by `OnNext` with an `IList` of 2 elements, followed by `OnCompleted`.

6.3.11: Cast

Creates an observable of a specific type from an observable of object using `cast`.

```
public static IObservable<TResult> Cast<TResult>(this
IObservable<object> source);
```

Parameters

- `source` (`IObservable<TSource>`) - The source observable (of objects)

Return Value

- `IObservable<TSource>` - A strongly typed observable

Remarks

This operator in effect casts the source elements into `TSource` elements.

Since the compiler has no way of inferring the type for `TSource`, it must be explicitly stated.

6.3.12: Catch

Catches an error and moves to the next observable.

```
public static IObservable<TSource> Catch<TSource>(
    params IObservable<TSource>[] sources);
public static IObservable<TSource> Catch<TSource>(
    this IEnumerable<IObservable<TSource>> sources);
public static IObservable<TSource> Catch<TSource, TException>(
    this IObservable<TSource> source, Func<TException,
    IObservable<TSource>> handler) where TException : Exception;
public static IObservable<TSource> Catch<TSource>(
    this IObservable<TSource> first, IObservable<TSource> second);
```

Parameters

- `source` (`IObservable<TSource>`) - The source observable

Return Value

- `IObservable<TSource>` - An observable consisting of those elements from the first source observable before it (potentially) raises an error, following by those for the second, etc.

Remarks

Much like `catch` in C# exception handling, this operator allows an exception to be caught and handles it by progressing to subsequent source observables.

6.3.13: CombineLatest

Combines the latest elements from first and second observables using a result

selector function.

```
public static IObservable<TResult>
    CombineLatest<TFirst, TSecond, TResult>(
        this IObservable<TFirst> first,
        IObservable<TSecond> second,
        Func<TFirst, TSecond, TResult> resultSelector);
```

Parameters

- first (IObservable<TSource>) - The first source observable
- second (IObservable<TSource>) - The second source observable
- resultSelector (Func<TFirst, TSecond, TResult>) - A function to select elements

Return Value

- IObservable<TResult> - A observable which is the combination of elements from first and second source observables based on the result selector

Remarks

Note that the type parameters - TFirst, TSecond and TResult - may all be different.

The role of the resultSelector is to decide what the resulting element is for each pair of elements accepted from first and second sources.

6.3.14: Concat

Concatenates multiple source observables.

```
public static IObservable<TSource> Concat<TSource>(
    params IObservable<TSource>[] sources);
public static IObservable<TSource> Concat<TSource>(
    this IEnumerable<IObservable<TSource>> sources);
public static IObservable<TSource> Concat<TSource>(
    this IObservable<IObservable<TSource>> sources);
public static IObservable<TSource> Concat<TSource>(
    this IObservable<TSource> first, IObservable<TSource> second);
```

Parameters

- sources (IObservable<TSource>[]) - The source observables
- sources (IEnumerable<TSource>>) - The source observables
- sources (IObservable<IObservable<TSource>>>) - The source observables

Return Value

- `IObservable<TSource>` - the concatenated observable

Remarks

This operator concates multiple source observables into one.

Contrast the Merge and Concat operators.

6.3.15: Contains

Determines if a source observable contains a specified value.

```
public static IObservable<bool> Contains<TSource>(
    this IObservable<TSource> source, TSource value);
public static IObservable<bool> Contains<TSource>(
    this IObservable<TSource> source, TSource value,
    IEqualityComparer<TSource> comparer);
```

Parameters

- source (`IObservable<TSource>`) - the source
- value (`TSource`) - The value to check for

Return Value

- `IObservable<bool>` - An observable containing one boolean element, stating if source contained value

6.3.16: Count

Counts the elements in an observable.

```
public static IObservable<int> Count<TSource>(this
    IObservable<TSource> source);
```

Parameters

- source (`IObservable<TSource>`) - The source observable

Return Value

- `IObservable<int>` - An element (of type `int`) containing the number of elements in the source

Remarks

The result observable only emits its element when it has completed cycling through the source.

There is also a `LongCount` operator.

6.3.17: Create

Creates an observable.

```
public static IObservable<TSource> Create<TSource>(
    Func<IObserver<TSource>, Action> subscribe);
```

```
public static IObservable<TSource> Create<TSource>(
    Func<IObserver<TSource>, IDisposable> subscribe);
```

Parameters

- subscribe (Func<IObserver<TSource>, Action>) - A function with one input parameter that returns an Action
- subscribe (Func<IObserver<TSource>, IDisposable>) - A function with one input parameter that returns an IDisposable

Return Value

- IObservable<TSource> - The create observable

Remarks

6.3.18: DefaultIfEmpty

The resulting observable has at least one element.

```
public static IObservable<TSource> DefaultIfEmpty<TSource>(
    this IObservable<TSource> source);
public static IObservable<TSource> DefaultIfEmpty<TSource>(
    this IObservable<TSource> source, TSource defaultValue);
```

Parameters

- source (IObservable<TSource>) - The source observable
- defaultValue (TSource) - The default value to use

Return Value

- IObservable<TSource> - An observable with at least one element

6.3.19: Defer

Uses an observable factory to create observables when new subscriptions are established.

```
public static IObservable<TValue> Defer<TValue>(
    Func<IObservable<TValue>> observableFactory);
```

Parameters

- observableFactory (Func<IObservable<TValue>>) - A function used to create observables when needed

Return Value

- IObservable<TValue> - An observable which will invoke the factory func when needed

Remarks

6.3.20: Delay

Inserts a delay before element generation for an observable starts.

```
public static IObservable<TSource> Delay<TSource>(
    this IObservable<TSource> source, DateTimeOffset dueTime);
public static IObservable<TSource> Delay<TSource>(
    this IObservable<TSource> source, TimeSpan dueTime);
public static IObservable<TSource> Delay<TSource>(
    this IObservable<TSource> source, DateTimeOffset dueTime,
    IScheduler scheduler);
public static IObservable<TSource> Delay<TSource>(
    this IObservable<TSource> source, TimeSpan dueTime,
    IScheduler scheduler);
```

Parameters

- source (IObservable<TSource>) - The source
- dueTime (DateTimeOffset) - The delay
- dueTime (TimeSpan) - The delay
- scheduler (IScheduler) - The scheduler to use for this operator

Return Value

- IObservable<TSource> - The time shifted observable

Remarks

The delay is only inserted once.

6.3.21: Dematerialize

Converts a notification element to a code invocation element, in the sense that Notification instances become OnNext/OnError/OnCompleted method calls.

```
public static IObservable<TSource> Dematerialize<TSource>(
    this IObservable<Notification<TSource>> source);
```

Parameters

- source (IObservable<Notification<TSource>>) - An observable of notifications

Return Value

- IObservable<TSource> - An observable of element

Remarks

Normally distinct methods are needed to handle OnNext/OnError/OnCompleted calls from a source. Notifications are a way of unifying this handling. There is a

common base notification type (System.Reactive) and it can be used to indicate which call has occurred.

Dematerialization converts from notifications to method calls.

Also look at Materialize(), which is the reverse.

6.3.22: Distinct

Returns an observable with the distinct elements in the source.

```
public static IObservable<TSource> Distinct<TSource>(
    this IObservable<TSource> source);
public static IObservable<TSource> Distinct<TSource, TKey>(
    this IObservable<TSource> source, Func<TSource, TKey> keySelector);
public static IObservable<TSource> Distinct<TSource>(
    this IObservable<TSource> source,
    IEqualityComparer<TSource> comparer);
public static IObservable<TSource> Distinct<TSource, TKey>(
    this IObservable<TSource> source, Func<TSource, TKey> keySelector,
    IEqualityComparer<TKey> comparer);
```

Parameters

- source (IObservable<TSource>) - The source observable
- keySelector (Func<TSource, TKey>) - A function to determine the key to use when deciding on distinctiveness

Return Value

- IObservable<TSource> - An observable with the distinct elements from the source

Remarks

The operator detects which elements in the source are distinct (possibly using the key selector function), and adds them to the resulting observable.

6.3.23: DistinctUntilChanged

Returns elements so long as they are not the same as the immediately previous element.

```
public static IObservable<TSource> DistinctUntilChanged<TSource>(
    this IObservable<TSource> source);
public static IObservable<TSource> DistinctUntilChanged<TSource, TKey>(
    this IObservable<TSource> source,
    Func<TSource, TKey> keySelector);
public static IObservable<TSource> DistinctUntilChanged<TSource>(
    this IObservable<TSource> source,
    IEqualityComparer<TSource> comparer);
public static IObservable<TSource>
    DistinctUntilChanged<TSource, TKey>(
    this IObservable<TSource> source,
    Func<TSource, TKey> keySelector,
    IEqualityComparer<TKey> comparer);
```

Parameters

- source (IObservable<TSource>) - The source
- keySelector (Func<TSource, TKey>) - The key selector function
- comparer (IEqualityComparer<TKey>) - The comparer to use

Return Value

IObservable<Source> - The resulting observable

Remarks

The difference between Distinct and DistinctUntilChanged is that the former will only emit a unique element once, whereas the later will emit element so long as they are different from its predecessor.

6.3.24: Do

Executes an additional function when cycling through a source.

```
public static IObservable<TSource> Do<TSource>(
    this IObservable<TSource> source, Action<TSource> onNext);
public static IObservable<TSource> Do<TSource>(
    this IObservable<TSource> source, IObservable<TSource> observer);
public static IObservable<TSource> Do<TSource>(
    this IObservable<TSource> source, Action<TSource> onNext,
    Action onCompleted);
public static IObservable<TSource> Do<TSource>(
    this IObservable<TSource> source, Action<TSource> onNext,
    Action<Exception> onError);
public static IObservable<TSource> Do<TSource>(
    this IObservable<TSource> source, Action<TSource> onNext,
    Action<Exception> onError, Action onCompleted);
```

Parameters

- source (IObservable<TSource>) - The source observable
- onNext (Action<TSource>) - An additional action to execute when the source passes an element to an observer
- onError (Action<Exception>) - An additional action to execute when the source passes an error to an observer
- onCompleted (Action) - An additional action to execute when the source completes

Return Value

- IObservable<TSource> - An observable, which, when observed, will carry execute the additional actions specified

Remarks

The action for the Do operator only execute when an observer is observing the returned observable.

Also examine ForEach operator.

6.3.25: ElementAt

Returns the element at a specific index (or throws an exception if not available).

```
public static IObservable<TSource> ElementAt<TSource>(
    this IObservable<TSource> source, int index);
```

Parameters

- source (IObservable<TSource>) - The source observable
- index (int) - The zero-based index of the element to return

Return Value

- IObservable<TSource> - An observable with one element, that which appear at index in the source, and if not available, calls OnError

Remarks

If there is no element at the index specified, OnError is called with an exception of ArgumentOutOfRangeException and an message of "Specified argument was out of the range of valid values".

6.3.26: ElementAtOrDefault

Returns the element at a specific index (or the default if not available).

```
public static IObservable<TSource> ElementAtOrDefault<TSource>(
    this IObservable<TSource> source, int index);
```

Parameters

- source (IObservable<TSource>) - The source observable
- index (int) - The zero-based index of the element to return

Return Value

- IObservable<TSource> - An observable with one element, that which appear at index in the source, or the default

Remarks

If there is no element at the index location, default(TSource) is returned.

6.3.27: Empty

Creates an empty observable.

```
public static IObservable<TResult> Empty<TResult>();
public static IObservable<TResult> Empty<TResult>(
    IScheduler scheduler);
```

Parameters

- scheduler (IScheduler) - The scheduler to use for this operator

Return Value

- IObservable<TResult> - An observable containing no elements

Remarks

This operator creates an observable which has no elements, and when subscribed to, simply calls OnCompleted.

6.3.28: Finally

The key property.

```
public static IObservable<TSource> Finally<TSource>(
    this IObservable<TSource> source, Action finallyAction);
```

Parameters

- source (IObservable<TSource>) - The source observable
- finallyAction (Action) - The action to execute, regardless of whether errors were generating during a subscription to the result

Return Value

- IObservable<TSource> - A result observable which, when subscribed to, will return elements from the source, and at the end (after either OnError or OnCompleted), will execute the finallyAction

Remarks

like a finally block is a C# exception handler, this operator ensure a finally action is executed for the source.

6.3.29: First

Returns the first element in the source.

```
public static TSource First<TSource>(this IObservable<TSource>
source);
public static TSource First<TSource>(this IObservable<TSource>
source, Func<TSource, bool> predicate);
```

Parameters

- source (IObservable<TSource>) - The source observable

Return Value

- TSource - The first element

Remarks

This operator returns the first element.

6.3.30: FirstOrDefault

The key property.

```
public static TSource FirstOrDefault<TSource>(this
IObservable<TSource> source);
public static TSource FirstOrDefault<TSource>(this
IObservable<TSource> source, Func<TSource, bool> predicate);
```

Parameters

- source (IObservable<TSource>) - The source observable

Return Value

- TSource - The first element or the default

Remarks

This operator returns the first element if the source has at least one, otherwise it return default(TSource).

6.3.31: ForEach

Cycles through the elements in a source observable and executing actions as configured via parameters.

```
public static void ForEach<TSource>(this IObservable<TSource>
source);
public static void ForEach<TSource>(this IObservable<TSource> source,
Action<TSource> onNext);
public static void ForEach<TSource>(this IObservable<TSource> source,
IObserver<TSource> observer);
public static void ForEach<TSource>(this IObservable<TSource> source,
Action<TSource> onNext, Action onCompleted);
public static void ForEach<TSource>(this IObservable<TSource> source,
Action<TSource> onNext, Action<Exception> onError);
public static void ForEach<TSource>(this IObservable<TSource> source,
Action<TSource> onNext, Action<Exception> onError, Action onCompleted);
```

Parameters

- source (IObservable<TSource>) - The source observable
- onNext () - The action to execute when a new element is available
- onError () - the action to execute when an error is reported
- onCompleted () - The action to execute when the source completes
- observer () - The observer to which to pass observable elements

Remarks

For each element, this operator does something. For the first implementation, what it does is only to wait for the element to arrive which includes the source

side-effect when generate that element, for the other implementations of ForEach, actions are executed.

Contrast ForEach with Do. ForEach has no return value, whereas Do returns an IObservable.

6.3.32: FromAsyncPattern

Converts from an APM async pattern to a function which returns an IObservable of Unit.

```
public static Func<IObservable<Unit>> FromAsyncPattern(
    Func<AsyncCallback, object, IAsyncResult> begin,
    Action<IAsyncResult> end);
public static Func<IObservable<TResult>> FromAsyncPattern<TResult>(
    Func<AsyncCallback, object, IAsyncResult> begin,
    Func<IAsyncResult, TResult> end);
public static Func<T1, IObservable<Unit>> FromAsyncPattern<T1>(
    Func<T1, AsyncCallback, object, IAsyncResult> begin,
    Action<IAsyncResult> end);
public static Func<T1, IObservable<TResult>>
    FromAsyncPattern<T1, TResult>(
    Func<T1, AsyncCallback, object, IAsyncResult> begin,
    Func<IAsyncResult, TResult> end);
public static Func<T1, T2, IObservable<Unit>>
    FromAsyncPattern<T1, T2>(
    Func<T1, T2, AsyncCallback, object, IAsyncResult> begin,
    Action<IAsyncResult> end);
public static Func<T1, T2, IObservable<TResult>>
    FromAsyncPattern<T1, T2, TResult>(
    Func<T1, T2, AsyncCallback, object, IAsyncResult> begin,
    Func<IAsyncResult, TResult> end);
public static Func<T1, T2, T3, IObservable<Unit>>
    FromAsyncPattern<T1, T2, T3>(
    Func<T1, T2, T3, AsyncCallback, object, IAsyncResult> begin,
    Action<IAsyncResult> end);
public static Func<T1, T2, T3, IObservable<TResult>>
    FromAsyncPattern<T1, T2, T3, TResult>(
    Func<T1, T2, T3, AsyncCallback, object, IAsyncResult> begin,
    Func<IAsyncResult, TResult> end);
public static Func<T1, T2, T3, T4, IObservable<Unit>>
    FromAsyncPattern<T1, T2, T3, T4>(
    Func<T1, T2, T3, T4, AsyncCallback, object, IAsyncResult> begin,
    Action<IAsyncResult> end);
public static Func<T1, T2, T3, T4, IObservable<TResult>>
    FromAsyncPattern<T1, T2, T3, T4, TResult>(
    Func<T1, T2, T3, T4, AsyncCallback, object, IAsyncResult> begin,
    Func<IAsyncResult, TResult> end);
public static Func<T1, T2, T3, T4, T5, IObservable<Unit>>
    FromAsyncPattern<T1, T2, T3, T4, T5>(
    Func<T1, T2, T3, T4, T5, AsyncCallback, object, IAsyncResult> begin,
    Action<IAsyncResult> end);
public static Func<T1, T2, T3, T4, T5, IObservable<TResult>>
    FromAsyncPattern<T1, T2, T3, T4, T5, TResult>(
    Func<T1, T2, T3, T4, T5, AsyncCallback, object, IAsyncResult> begin,
    Func<IAsyncResult, TResult> end);
public static Func<T1, T2, T3, T4, T5, T6, IObservable<Unit>>
    FromAsyncPattern<T1, T2, T3, T4, T5, T6>(
    Func<T1, T2, T3, T4, T5, T6, AsyncCallback, object, IAsyncResult>
```

```

        begin, Action<IAsyncResult> end);
public static Func<T1, T2, T3, T4, T5, T6, IObservable<TResult>>
    FromAsyncPattern<T1, T2, T3, T4, T5, T6, TResult>(
        Func<T1, T2, T3, T4, T5, T6, AsyncCallback, object, IAsyncResult>
        begin, Func<IAsyncResult, TResult> end);
public static Func<T1, T2, T3, T4, T5, T6, T7, IObservable<Unit>>
    FromAsyncPattern<T1, T2, T3, T4, T5, T6, T7>(
        Func<T1, T2, T3, T4, T5, T6, T7, AsyncCallback, object, IAsyncResult>
        begin, Action<IAsyncResult> end);
public static Func<T1, T2, T3, T4, T5, T6, T7, IObservable<TResult>>
    FromAsyncPattern<T1, T2, T3, T4, T5, T6, T7, TResult>(
        Func<T1, T2, T3, T4, T5, T6, T7, AsyncCallback, object,
        IAsyncResult> begin, Func<IAsyncResult, TResult> end);
public static Func<T1, T2, T3, T4, T5, T6, T7, T8, IObservable<Unit>>
    FromAsyncPattern<T1, T2, T3, T4, T5, T6, T7, T8>(
        Func<T1, T2, T3, T4, T5, T6, T7, T8, AsyncCallback, object,
        IAsyncResult> begin, Action<IAsyncResult> end);
public static Func<T1, T2, T3, T4, T5, T6, T7, T8, IObservable<TResult>>
    FromAsyncPattern<T1, T2, T3, T4, T5, T6, T7, T8, TResult>(
        Func<T1, T2, T3, T4, T5, T6, T7, T8, AsyncCallback, object,
        IAsyncResult> begin, Func<IAsyncResult, TResult> end);
public static Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, IObservable<Unit>>
    FromAsyncPattern<T1, T2, T3, T4, T5, T6, T7, T8, T9>(
        Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, AsyncCallback, object,
        IAsyncResult> begin, Action<IAsyncResult> end);
public static
    Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, IObservable<TResult>>
    FromAsyncPattern<T1, T2, T3, T4, T5, T6, T7, T8, T9, TResult>(
        Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, AsyncCallback, object,
        IAsyncResult> begin, Func<IAsyncResult, TResult> end);
public static
    Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, IObservable<Unit>>
    FromAsyncPattern<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10>(
        Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, AsyncCallback, object,
        IAsyncResult> begin, Action<IAsyncResult> end);
public static
    Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, IObservable<TResult>>
    FromAsyncPattern<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10,
    TResult>(Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, AsyncCallback,
    object, IAsyncResult> begin, Func<IAsyncResult, TResult> end);
public static
    Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, IObservable<Unit>>
    FromAsyncPattern<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11>(
        Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, AsyncCallback,
        object, IAsyncResult> begin, Action<IAsyncResult> end);
public static
    Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, IObservable<TResult>>
    FromAsyncPattern<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, TResult>(
        Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, AsyncCallback,
        object, IAsyncResult> begin, Func<IAsyncResult, TResult> end);
public static Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12,
    IObservable<Unit>> FromAsyncPattern<T1, T2, T3, T4, T5, T6, T7, T8,
    T9, T10, T11, T12>(
        Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12,
        AsyncCallback, object, IAsyncResult> begin, Action<IAsyncResult> end);
public static Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12,
    IObservable<TResult>> FromAsyncPattern<T1, T2, T3, T4, T5, T6, T7, T8,
    T9, T10, T11, T12, TResult>(
        Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12,
        AsyncCallback, object, IAsyncResult> begin,

```

```

        Func<IAsyncResult, TResult> end);
public static Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12,
T13, IObservable<Unit>> FromAsyncPattern<T1, T2, T3, T4, T5, T6, T7, T8,
T9, T10, T11, T12, T13>(
    Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13,
    AsyncCallback, object, IAsyncResult> begin,
    Action<IAsyncResult> end);
public static Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12,
T13, IObservable<TResult>> FromAsyncPattern<T1, T2, T3, T4, T5, T6, T7,
T8, T9, T10, T11, T12, T13, TResult>(
    Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13,
    AsyncCallback, object, IAsyncResult> begin,
    Func<IAsyncResult, TResult> end);
public static Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12,
T13, T14, IObservable<Unit>> FromAsyncPattern<T1, T2, T3, T4, T5, T6, T7,
T8, T9, T10, T11, T12, T13, T14>(
    Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14,
    AsyncCallback, object, IAsyncResult> begin,
    Action<IAsyncResult> end);
public static Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12,
T13, T14, IObservable<TResult>> FromAsyncPattern<T1, T2, T3, T4, T5, T6,
T7, T8, T9, T10, T11, T12, T13, T14, TResult>(
    Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14,
    AsyncCallback, object, IAsyncResult> begin,
    Func<IAsyncResult, TResult> end);

```

Parameters

- begin (Func<AsyncCallback, object, IAsyncResult>) - The begin handler
- end (Action<IAsyncResult>) - The end handler

Return Value

- Func<T1,..., IObservable<Unit>>) - A function which accepts a varying number of parameters and returns IObservable<Unit>

Remarks

Converts from an async pattern to an observable, thus allowing intergration between the world of Rx and APM.

6.3.33: FromEvent

Create a sequence of TEventArgs from add and remove event handlers.

```

public static IObservable<TEventArgs> FromEvent<TEventArgs>(
    Action<Action<TEventArgs>> addHandler,
    Action<Action<TEventArgs>> removeHandler);
public static IObservable<Unit> FromEvent(
    Action<Action> addHandler,
    Action<Action> removeHandler);
public static IObservable<TEventArgs> FromEvent<TDelegate, TEventArgs>(
    Action<TDelegate> addHandler,
    Action<TDelegate> removeHandler);
public static IObservable<TEventArgs> FromEvent<TDelegate, TEventArgs>(
    Func<Action<TEventArgs>, TDelegate> conversion,
    Action<TDelegate> addHandler, Action<TDelegate> removeHandler);

```

Parameters

- `addHandler (Action<Action<TEventArgs>>)` - Used to add handlers
- `removeHandler (Action<Action<TEventArgs>>)` - Used to remove handlers

Return Value

- `IObservable<TEventArgs>` - Sequence of event args

Remarks

6.3.34: FromEventPattern

Create a sequence of `EventPattern<TEventArgs>` from add and remove event handlers.

```
public static IObservable<EventPattern<TEventArgs>>
    FromEventPattern<TEventArgs>(
        Action<EventHandler<TEventArgs>> addHandler,
        Action<EventHandler<TEventArgs>> removeHandler)
    where TEventArgs : EventArgs;

public static IObservable<EventPattern<EventArgs>> FromEventPattern(
    Action<EventHandler> addHandler,
    Action<EventHandler> removeHandler);

public static IObservable<EventPattern<TEventArgs>>
    FromEventPattern<TDelegate, TEventArgs>(
        Action<TDelegate> addHandler, Action<TDelegate> removeHandler)
    where TEventArgs : EventArgs;

public static IObservable<EventPattern<EventArgs>> FromEventPattern(
    object target, string eventName);

public static IObservable<EventPattern<TEventArgs>>
    FromEventPattern<TEventArgs>(
        object target, string eventName) where TEventArgs : EventArgs;

public static IObservable<EventPattern<EventArgs>> FromEventPattern(
    Type type, string eventName);

public static IObservable<EventPattern<TEventArgs>>
    FromEventPattern<TEventArgs>(Type type, string eventName)
    where TEventArgs : EventArgs;

public static IObservable<EventPattern<TEventArgs>>
    FromEventPattern<TDelegate, TEventArgs>(
        Func<EventHandler<TEventArgs>, TDelegate> conversion,
        Action<TDelegate> addHandler, Action<TDelegate> removeHandler)
    where TEventArgs : EventArgs;
```

Parameters

- `addHandler (Action<EventHandler<TEventArgs>>)` - Used to add

handlers

- `removeHandler (Action<EventHandler<TEventArgs>>)` - Used to remove handlers

Return Value

- `IObservable<EventPattern<TEventArgs>>` - Sequence of event args

Remarks

6.3.35: Generate

Generate a sequence based on parameters - similar in concept to a for loop to generate elements.

```
public static IObservable<TResult> Generate<TState, TResult>(
    TState initialState, Func<TState, bool> condition,
    Func<TState, TState> iterate, Func<TState, TResult> resultSelector);
public static IObservable<TResult> Generate<TState, TResult>(
    TState initialState, Func<TState, bool> condition,
    Func<TState, TState> iterate, Func<TState, TResult> resultSelector,
    Func<TState, DateTimeOffset> timeSelector);
public static IObservable<TResult> Generate<TState, TResult>(
    TState initialState, Func<TState, bool> condition,
    Func<TState, TState> iterate, Func<TState, TResult> resultSelector,
    Func<TState, TimeSpan> timeSelector);
public static IObservable<TResult> Generate<TState, TResult>(
    TState initialState, Func<TState, bool> condition,
    Func<TState, TState> iterate, Func<TState, TResult> resultSelector,
    IScheduler scheduler);
public static IObservable<TResult> Generate<TState, TResult>(
    TState initialState, Func<TState, bool> condition,
    Func<TState, TState> iterate, Func<TState, TResult> resultSelector,
    Func<TState, DateTimeOffset> timeSelector, IScheduler scheduler);
public static IObservable<TResult> Generate<TState, TResult>(
    TState initialState, Func<TState, bool> condition,
    Func<TState, TState> iterate, Func<TState, TResult> resultSelector,
    Func<TState, TimeSpan> timeSelector, IScheduler scheduler);
```

Parameters

- `initialState (TState)` - the initial state
- `condition (Func<TState, bool>)` - The end condition to determine when generation is to be completed
- `iterate (Func<TState, TState>)` - The logic to iterate to the next element
- `resultSelector (Func<TState, TResult>)` - Determining the next element
- `timeSelector (Func<TState, TimeSpan>)` - The delay between emitting elements
- `scheduler (IScheduler)` - The scheduler to use for this operator

Return Value

- (IObservable<TResult>) - The generated sequence

Remarks

The easiest way to consider Generate is to think of it as a mirror of a for loop.

```
for (i=0; i<10; i++) return i * 2;
```

Generate (initial state, end condition, iterate) resultSelector

6.3.36: GetEnumerator

One can enumerate a source observable using the enumerator provided by this operator.

```
public static IEnumerable<TSource> GetEnumerator<TSource>(
    this IObservable<TSource> source);
```

Parameters

- source (IObservable<TSource>) - The source

Return Value

- IEnumerable<TSource> - The enumerator which can be used to enumerate elements in the source

6.3.37: GroupBy

```
public static IObservable<IGroupedObservable<TKey, TSource>>
GroupBy<TSource, TKey>(this IObservable<TSource> source, Func<TSource,
TKey> keySelector);
public static IObservable<IGroupedObservable<TKey, TElement>>
GroupBy<TSource, TKey, TElement>(this IObservable<TSource> source,
Func<TSource, TKey> keySelector, Func<TSource, TElement>
elementSelector);
public static IObservable<IGroupedObservable<TKey, TSource>>
GroupBy<TSource, TKey>(this IObservable<TSource> source, Func<TSource,
TKey> keySelector, IEqualityComparer<TKey> comparer);
public static IObservable<IGroupedObservable<TKey, TElement>>
GroupBy<TSource, TKey, TElement>(this IObservable<TSource> source,
Func<TSource, TKey> keySelector, Func<TSource, TElement> elementSelector,
IEqualityComparer<TKey> comparer);
```

6.3.38: GroupByUntil

```

    public static IObservable<IGroupedObservable<TKey, TSource>>
    GroupByUntil<TSource, TKey, TDuration>(this IObservable<TSource> source,
    Func<TSource, TKey> keySelector, Func<IGroupedObservable<TKey, TSource>,
    IObservable<TDuration>> durationSelector);
    public static IObservable<IGroupedObservable<TKey, TSource>>
    GroupByUntil<TSource, TKey, TDuration>(this IObservable<TSource> source,
    Func<TSource, TKey> keySelector, Func<IGroupedObservable<TKey, TSource>,
    IObservable<TDuration>> durationSelector, IEqualityComparer<TKey>
    comparer);
    public static IObservable<IGroupedObservable<TKey, TElement>>
    GroupByUntil<TSource, TKey, TElement, TDuration>(this
    IObservable<TSource> source, Func<TSource, TKey> keySelector,
    Func<TSource, TElement> elementSelector, Func<IGroupedObservable<TKey,
    TElement>, IObservable<TDuration>> durationSelector);
    public static IObservable<IGroupedObservable<TKey, TElement>>
    GroupByUntil<TSource, TKey, TElement, TDuration>(this
    IObservable<TSource> source, Func<TSource, TKey> keySelector,
    Func<TSource, TElement> elementSelector, Func<IGroupedObservable<TKey,
    TElement>, IObservable<TDuration>> durationSelector,
    IEqualityComparer<TKey> comparer);

```

Parameters

- source (IObservable<TSource>) - The source observable

Return Value

- s () -

Remarks**6.3.39: GroupJoin**

```

    public static IObservable<TResult> GroupJoin<TLeft, TRight,
    TLeftDuration, TRightDuration, TResult>(this IObservable<TLeft> left,
    IObservable<TRight> right, Func<TLeft, IObservable<TLeftDuration>>
    leftDurationSelector, Func<TRight, IObservable<TRightDuration>>
    rightDurationSelector, Func<TLeft, IObservable<TRight>, TResult>
    resultSelector);

```

Parameters

- source (IObservable<TSource>) - The source observable

Return Value

- s () -

Remarks

6.3.40: IgnoreElement

Returns an observable with no elements but which emits OnCompleted when the source emits OnCompleted.

```
public static IObservable<TSource> IgnoreElements<TSource>(
    this IObservable<TSource> source);
```

Parameters

- source (IObservable<TSource>) - The source observable

Return Value

- IObservable<TSource> - An observable that only emits OnCompleted

Remarks

This operator effectively ignores whatever elements the source has, and the observable it returns emits OnCompleted when the source does.

6.3.41: Interval

The resulting observable emits an element after every interval.

```
public static IObservable<long> Interval(TimeSpan period);
public static IObservable<long> Interval(TimeSpan period,
    IScheduler scheduler);
```

Parameters

- period (TimeSpan) - The period between elements
- scheduler (IScheduler) - The scheduler to use for this operator

Return Value

- IObservable<long> - An observable that emits element after every interval has elapsed

Remarks

The value of the long in the emitted element is a zero-based index of the of element issued.

6.3.42: Join

Joins two observables, based on duration selector functions, with the result also based on a selector function.

```
public static IObservable<TResult>
Join<TLeft, TRight, TLeftDuration, TRightDuration, TResult>(
    this IObservable<TLeft> left,
    IObservable<TRight> right,
    Func<TLeft, IObservable<TLeftDuration>> leftDurationSelector,
    Func<TRight, IObservable<TRightDuration>> rightDurationSelector,
```

```
Func<TLeft, TRight, TResult> resultSelector);
```

Parameters

- left (IObservable<TSource>) - The left source observable
- right (IObservable<TSource>) - The right source observable
- leftDurationSelector () -
- rightDurationSelector () -
- resultSelector () -

Return Value

- IObservable<TResult> - An observable contains the result of the correlation

Remarks

6.3.43: Last

Returns a single element, which is the last element the source observable emitted.

```
public static TSource Last<TSource>(
    this IObservable<TSource> source);
public static TSource Last<TSource>(
    this IObservable<TSource> source, Func<TSource, bool> predicate);
```

Parameters

- source (IObservable<TSource>) - the source
- predicate (Func<TSource, bool>) - A condition which the element must satisfy

Return Value

- TSource - The last element emitted by the source

Remarks

The element created by Last is not returned until the source completes, as only then is the last element known.

If the source observable is empty (e.g. created by Observable.Empty<TSource>()), then an exception is raised, of type InvalidOperationException, with the error message set to "Sequence contains no

elements”.

6.3.44: LastOrDefault

Returns a single element, which is the last element the source observable emitted, or the default value for the source type.

```
public static TSource LastOrDefault<TSource>(
    this IObservable<TSource> source);
public static TSource LastOrDefault<TSource>(
    this IObservable<TSource> source, Func<TSource, bool> predicate);
```

Parameters

- source (IObservable<TSource>) - the source
- predicate (Func<TSource, bool>) - A condition which the element must satisfy

Return Value

- TSource - The last element emitted by the source, or the default

Remarks

If the source observable is empty (e.g. created by Observable.Empty<TSource>()), then not exception is raised, and instead the default for the source type is returned.

6.3.45: Latest

Returns the most recent element from a source observable.

```
public static IEnumerable<TSource> Latest<TSource>(
    this IObservable<TSource> source);
```

Parameters

- source (IObservable<TSource>) - The source observable

Return Value

- IEnumerable<TSource> - The most recent element

Remarks

Extracts the most recent element from the observable.

6.3.46: LongCount

Counts the elements in an observable.

```
public static IObservable<long> LongCount<TSource>(
    this IObservable<TSource> source);
```

Parameters

- source (IObservable<TSource>) - The source observable

Return Value

- `IObservable<long>` - An element (of type `long`) containing the number of elements in the source

Remarks

The result observable only emits its element when it has completed cycling through the source.

There is also a `Count` operator.

6.3.47: Materialize

Converts a code invocation to a data element, in the sense that `OnNext/OnError/OnCompleted` method calls become `Notification` instances.

```
public static IObservable<Notification<TSource>>
    Materialize<TSource>(this IObservable<TSource> source);
```

Parameters

- `source (IObservable<TSource>)` - The source observable

Return Value

- `IObservable<Notification<TSource>>` - An observable of notifications

Remarks

Normally distinct methods are needed to handle `OnNext/OnError/OnCompleted` calls from a source. Notifications are a way of unifying this handling. There is a common base notification type (`System.Reactive`) and it can be used to indicate which call has occurred.

Also look at `DeMaterialize()`.

6.3.48: Max

Calculates the maximum numeric value in a source observable.

```
public static IObservable<decimal?> Max(
    this IObservable<decimal?> source);
public static IObservable<decimal> Max(
    this IObservable<decimal> source);
public static IObservable<double?> Max(
    this IObservable<double?> source);
public static IObservable<double> Max(
    this IObservable<double> source);
public static IObservable<float?> Max(
    this IObservable<float?> source);
public static IObservable<float> Max(this IObservable<float> source);
public static IObservable<int?> Max(this IObservable<int?> source);
public static IObservable<int> Max(this IObservable<int> source);
public static IObservable<long?> Max(this IObservable<long?> source);
public static IObservable<long> Max(this IObservable<long> source);
public static IObservable<TSource> Max<TSource>(
```

```

        this IObservable<TSource> source);
    public static IObservable<TSource> Max<TSource>(
        this IObservable<TSource> source, IComparer<TSource> comparer);

```

Parameters

- source (IObservable<TSource>) - The source observable

Return Value

- IObservable < *numeric type* > - The result in a single element observable

Remarks

This operator calculates the maximum of a sequence of numeric types.

Note the type of the returned sequence can be either int, int?, float, float?, double, double?, decimal or decimal?. The type of the source sequence can be decimal, decimal?, double, double?, float, float?, int, int?, long, or long?.

6.3.49: MaxBy

Returns an observable of a list of source elements, which contain elements that a selector function has deemed to be the maximum within the source.

```

    public static IObservable<IList<TSource>> MaxBy<TSource, TKey>(
        this IObservable<TSource> source, Func<TSource, TKey> keySelector);
    public static IObservable<IList<TSource>> MaxBy<TSource, TKey>(
        this IObservable<TSource> source, Func<TSource, TKey> keySelector,
        IComparer<TKey> comparer);

```

Parameters

- source (IObservable<TSource>) - The source observable
- keySelector (Func<TSource, TKey>) - A function which is passed in a source element and decide what key to use for it

Return Value

- IObservable<IList<TSource>> - A list of elements which are the maximum, as decided by the value returned by key selector

Remarks

This operator calculates max based on a key selector function. Multiple elements may have the maximum key returned by the selector, hence the return value is an observable of an IList.

6.3.50: Merge

Merges a number of source observables into a single output observable.

```

    public static IObservable<TSource> Merge<TSource>(
        params IObservable<TSource>[] sources);
    public static IObservable<TSource> Merge<TSource>(
        this IEnumerable<IObservable<TSource>> sources);

```

```

public static IObservable<TSource> Merge<TSource>(
    this IObservable<IObservable<TSource>> sources);
public static IObservable<TSource> Merge<TSource>(
    IScheduler scheduler, params IObservable<TSource>[] sources);
public static IObservable<TSource> Merge<TSource>(
    this IEnumerable<IObservable<TSource>> sources,
    int maxConcurrent);
public static IObservable<TSource> Merge<TSource>(
    this IEnumerable<IObservable<TSource>> sources,
    IScheduler scheduler);
public static IObservable<TSource> Merge<TSource>(
    this IObservable<IObservable<TSource>> sources,
    int maxConcurrent);
public static IObservable<TSource> Merge<TSource>(
    this IObservable<TSource> first, IObservable<TSource> second);
public static IObservable<TSource> Merge<TSource>(
    this IEnumerable<IObservable<TSource>> sources,
    int maxConcurrent, IScheduler scheduler);
public static IObservable<TSource> Merge<TSource>(
    this IObservable<TSource> first, IObservable<TSource> second,
    IScheduler scheduler);

```

Parameters

- sources (IObservable<TSource>[]) - The source observables to merge
- first (IObservable<TSource>) - The first observable to merge
- second (IObservable<TSource>) - The second observable to merge
- scheduler (IScheduler) - The scheduler to use for this operator
- maxConcurrent (int) - The maximum number of concurrent elements to be accepted

Return Value

- IObservable <TSource> - the merged observable

Remarks

This operator merges elements from multiple source.

Also consider Select and SelectMany.

6.3.51: Min

Calculates the minimum numeric value in a source observable.

```

public static IObservable<decimal?>
    Min(this IObservable<decimal?> source);
public static IObservable<decimal>
    Min(this IObservable<decimal> source);
public static IObservable<double?>
    Min(this IObservable<double?> source);
public static IObservable<double>
    Min(this IObservable<double> source);
public static IObservable<float?>
    Min(this IObservable<float?> source);

```

```

public static IObservable<float> Min(this IObservable<float> source);
public static IObservable<int?> Min(this IObservable<int?> source);
public static IObservable<int> Min(this IObservable<int> source);
public static IObservable<long?> Min(this IObservable<long?> source);
public static IObservable<long> Min(this IObservable<long> source);
public static IObservable<TSource>
    Min<TSource>(this IObservable<TSource> source);
public static IObservable<TSource>
    Min<TSource>(this IObservable<TSource> source,
                IComparer<TSource> comparer);

```

Parameters

- source (IObservable<TSource>) - The source observable

Return Value

- IObservable < *numeric type* > - The result in a single element observable

Remarks

This operator calculates the minimum of a sequence of numeric types.

Note the type of the returned sequence can be either int, int?, float, float?, double, double?, decimal or decimal?. The type of the source sequence can be decimal, decimal?, double, double?, float, float?, int, int?, long, or long?.

6.3.52: MinBy

Finds the minimum element based on a selector function and optionally comparer.

```

public static IObservable<IList<TSource>> MinBy<TSource, TKey>(
    this IObservable<TSource> source, Func<TSource, TKey> keySelector);
public static IObservable<IList<TSource>> MinBy<TSource, TKey>(
    this IObservable<TSource> source, Func<TSource, TKey> keySelector,
    IComparer<TKey> comparer);

```

Parameters

- source (IObservable<TSource>) - The source observable
- keySelector (Func<TSource, TKey>) - A function which is passed in a source element and decide what key to use for it
- comparer (IComparer<TKey>) - A comparer function used to determine what minimum means for the key

Return Value

- IObservable<IList<TSource>> - A list of elements which are the minimum, as decided by the value returned by key selector

Remarks

This operator calculates minimum based on a key selector function. Multiple elements may have the minimum key returned by the selector, hence the return value is an observable of an IList.

The difference between the MinBy operator and the Min operator is that the former operates on any element type and requires a key selector function, whereas the latter only operates on numeric types

6.3.53: MostRecent

Returns an IEnumerable which yields the most recent element emitted from a source (or a default value if none available yet).

```
public static IEnumerable<TSource> MostRecent<TSource>(
    this IObservable<TSource> source, TSource initialValue);
```

Parameters

- source (this IObservable<TSource>) - the source
- initialValue (TSource) - the initial value, if source does not immediately have an element

Return Value

- IEnumerable<TSource> - An enumerable yielding the most recent result of the source (or, if not available, the initial value)

6.3.54: Multicast

Creates a connectable observable which remains connected to the underlying observable while subscriptions exist to it.

```
public static IConnectableObservable<TResult>
    Multicast<TSource, TResult>(
        this IObservable<TSource> source,
        ISubject<TSource, TResult> subject);
public static IObservable<TResult>
    Multicast<TSource, TIntermediate, TResult>(
        this IObservable<TSource> source,
        Func<ISubject<TSource, TIntermediate>> subjectSelector,
        Func<IObservable<TIntermediate>, IObservable<TResult>> selector);
```

Parameters

- source (IObservable<TSource>) - The underlying observable
- subject (ISubject<TSource, TResult>) - The subject to which observers can subscribe and which itself subscribes to the underlying observable
- subjectSelector (Func<ISubject<TSource, TIntermediate>>) - a selector for the subject
- selector (Func<IObservable<TIntermediate>, IObservable<TResult>>) - A selector function

Return Value

- `IConnectableObservable<TResult>` - The connectable observable

6.3.55: Never

Creates an observable that never completes and never emits an element.

```
public static IObservable<TResult> Never<TResult>();
```

Return Value

- `IObservable<TResult>` - An observable that never returns

Remarks

This operator can be useful for testing purposes.

Since the compiler has no way of inferring the type of `TResult`, it must be explicitly set.

6.3.56: Next

Reports the next element from a source via a returned enumerable.

```
public static IEnumerable<TSource> Next<TSource>(
    this IObservable<TSource> source);
```

Parameters

- `source` (`IObservable<TSource>`) - the source

Return Value

- `IEnumerable<TSource>` - An enumerable which blocks until the next element is available from the source

6.3.57: ObserveOn

Observe source using the specified scheduler.

```
public static IObservable<TSource> ObserveOn<TSource>(this
IObservable<TSource> source, IScheduler scheduler);
public static IObservable<TSource> ObserveOn<TSource>(this
IObservable<TSource> source, SynchronizationContext context);
```

Parameters

- `source` (`IObservable<TSource>`) - the source observable

Return Value

- `IObservable<TSource>` - the observable which will deliver elements

Remarks

There is often a desire that elements from a source be observed via a specific scheduler and that is the purpose of this operator. Note there is a separate `SubscribeOn` operator for subscriptions.

6.3.58: OfType

Converts an observable of object to an observable of a specific type, discarding any elements not of that type.

```
public static IObservable<TResult> OfType<TResult>(
    this IObservable<object> source);
```

Parameters

- source (IObservable<object>) - An observable of object instances

Return Value

- IObservable<TResult> - The resulting observable all of whose instances share the same type.

6.3.59: OnErrorResumeNext

Accepts multiple source observables, and if one produces an error, resumes with the next one.

```
public static IObservable<TSource> OnErrorResumeNext<TSource>(
    params IObservable<TSource>[] sources);
public static IObservable<TSource> OnErrorResumeNext<TSource>(
    this IEnumerable<IObservable<TSource>> sources);
public static IObservable<TSource> OnErrorResumeNext<TSource>(
    this IObservable<TSource> first, IObservable<TSource> second);
```

Parameters

- sources (IObservable<TSource>[]) - The set of sources
- sources (IEnumerable<IObservable<TSource>[]>) - The set of sources
- first (IObservable<TSource>) - The first source
- second (IObservable<TSource>) - The second source

Return Value

IObservable<TSource> - The result

Remarks

Similar to the VB construct.

6.3.60: Publish

Publishes a single subscription to a source via the returned connectable observable to which multiple observers can subscribe.

```
public static IConnectableObservable<TSource> Publish<TSource>(
    this IObservable<TSource> source);
public static IObservable<TResult> Publish<TSource, TResult>(
    this IObservable<TSource> source,
    Func<IObservable<TSource>, IObservable<TResult>> selector);
public static IConnectableObservable<TSource> Publish<TSource>(
```

```
        this IObservable<TSource> source, TSource initialValue);  
public static IObservable<TResult> Publish<TSource, TResult>(  
    this IObservable<TSource> source,  
    Func<IObservable<TSource>, IObservable<TResult>> selector,  
    TSource initialValue);
```

Parameters

- source (IObservable<TSource>) - The underlying observable
- selector (Func<IObservable<TSource>, IObservable<TResult>>) - A selector function
- initialValue (TSource) - The initial value to emit

Return Value

- IConnectableObservable - An observable which maintains a single subscription to an underlying observable

Remarks

Publish can be used when one needs to share the same subscription to an underlying observable to multiple observers. The observers subscribe to the result of Publish, which internally subscribes to the underlying observable.

6.3.61: PublishLast

Creates an IConnectableObservable whose sole element is the last element from a source observable.

```
public static IConnectableObservable<TSource>  
    PublishLast<TSource>(this IObservable<TSource> source);  
public static IObservable<TResult> PublishLast<TSource, TResult>(this IObservable<TSource> source,  
    Func<IObservable<TSource>, IObservable<TResult>> selector);
```

Parameters

- source (this IObservable<TSource>) - the source whose last element is to be published
- selector (Func<IObservable<TSource>, IObservable<TResult>>) - A selector function

Return Value

- IConnectableObservable<TSource> - A connectable observable with the last element from the source

Remarks

Call Connect on the connectable observable in order to access the result.

6.3.62: Range

Creates an observable of integers with a specified number of elements comprising of a range of numbers from a start value incrementing by 1 each time.

```
public static IObservable<int> Range(int start, int count);  
public static IObservable<int> Range(int start, int count,  
    IScheduler scheduler);
```

Parameters

- start (int) - the value of the first element
- count (int) - The number of elements

Return Value

- IObservable<int> - The resulting observable of integers

Remarks

Note the second int parameter is a count of elements, not the upper limit to the range.

6.3.63: RefCount

Creates an observable that maintains a refcount of its subscriptions, and while it is non-zero, remains connected to the source observable.

```
public static IObservable<TSource> RefCount<TSource>(this IConnectableObservable<TSource> source);
```

Parameters

- source (this IConnectableObservable<TSource>) - A connection to a source observable

Return Value

- IObservable<TSource> - The resulting observable

Remarks

This operator uses a ref count to decide when to disconnect.

6.3.64: Repeat

Repeats the source elements, either without stopping, or a specified number of times.

```
public static IObservable<TSource> Repeat<TSource>(this IObservable<TSource> source);  
public static IObservable<TResult> Repeat<TResult>(TResult value);  
public static IObservable<TSource> Repeat<TSource>(this IObservable<TSource> source, int repeatCount);  
public static IObservable<TResult> Repeat<TResult>(TResult value, int repeatCount);
```

```

public static IObservable<TResult> Repeat<TResult>(
    TResult value, IScheduler scheduler);
public static IObservable<TResult> Repeat<TResult>(
    TResult value, int repeatCount, IScheduler scheduler);

```

Parameters

- source (IObservable<TSource>) - The source elements to repeat
- value (TResult) - A single element to repeat
- repeatCount - The number of times to repeat the source/value
- scheduler () - The scheduler to use for this operator

Return Value

- IObservable<TSource> - An observable of elements
- IObservable<TResult> - An observable of elements

Remarks

This operator repeats the source/value either continuously (and so never emits OnCompleted) or a repeatCount number of times and then emits OnCompleted.

6.3.65: Replay

Creates a connectable observable that maintains a single subscription to the source and replays its elements.

```

public static IConnectableObservable<TSource> Replay<TSource>(
    this IObservable<TSource> source);
public static IObservable<TResult> Replay<TSource, TResult>(
    this IObservable<TSource> source,
    Func<IObservable<TSource>, IObservable<TResult>> selector);
public static IConnectableObservable<TSource> Replay<TSource>(
    this IObservable<TSource> source, int bufferSize);
public static IConnectableObservable<TSource> Replay<TSource>(
    this IObservable<TSource> source, IScheduler scheduler);
public static IConnectableObservable<TSource> Replay<TSource>(
    this IObservable<TSource> source, TimeSpan window);
public static IObservable<TResult> Replay<TSource, TResult>(
    this IObservable<TSource> source, Func<IObservable<TSource>,
    IObservable<TResult>> selector, int bufferSize);
public static IObservable<TResult> Replay<TSource, TResult>(
    this IObservable<TSource> source,
    Func<IObservable<TSource>, IObservable<TResult>> selector,
    IScheduler scheduler);
public static IObservable<TResult> Replay<TSource, TResult>(
    this IObservable<TSource> source,
    Func<IObservable<TSource>, IObservable<TResult>> selector,
    TimeSpan window);
public static IConnectableObservable<TSource> Replay<TSource>(
    this IObservable<TSource> source, int bufferSize,
    IScheduler scheduler);
public static IConnectableObservable<TSource> Replay<TSource>(
    this IObservable<TSource> source, int bufferSize,
    TimeSpan window);

```

```
public static IConnectableObservable<TSource> Replay<TSource>(
    this IObservable<TSource> source, TimeSpan window,
    IScheduler scheduler);
public static IObservable<TResult> Replay<TSource, TResult>(
    this IObservable<TSource> source,
    Func<IObservable<TSource>, IObservable<TResult>> selector,
    int bufferSize,
    IScheduler scheduler);
public static IObservable<TResult> Replay<TSource, TResult>(
    this IObservable<TSource> source,
    Func<IObservable<TSource>, IObservable<TResult>> selector,
    int bufferSize, TimeSpan window);
public static IObservable<TResult> Replay<TSource, TResult>(
    this IObservable<TSource> source, Func<IObservable<TSource>,
    IObservable<TResult>> selector, TimeSpan window,
    IScheduler scheduler);
public static IConnectableObservable<TSource> Replay<TSource>(
    this IObservable<TSource> source, int bufferSize,
    TimeSpan window, IScheduler scheduler);
public static IObservable<TResult> Replay<TSource, TResult>(
    this IObservable<TSource> source,
    Func<IObservable<TSource>, IObservable<TResult>> selector,
    int bufferSize, TimeSpan window,
    IScheduler scheduler);
```

Parameters

- source (this IObservable<TSource>) - The source observable
- selector (Func<IObservable<TSource>, IObservable<TResult>>) - The selector function
- selector (IObservable<TResult>>) - An observable which emits an element to define the selector
- bufferSize (int) - The buffer size to maintain
- scheduler (IScheduler) - The scheduler to use for this element
- window (TimeSpan) - The timespan within which to repeat elements

Return Value

- IConnectableObservable<TSource> - A connectable observable which shares a single subscription to the source among its own (potentially many) observers
- IObservable<TResult> - An observable of result elements
- IObservable<TSource> - An observable of source elements

Remarks

This operator repeats elements it receives from the source.

When using the connectable observable, note its Connect() method must be

called.

Selection of repeated elements can be based on a timer, a buffer count or a selector function.

6.3.66: Retry

Keeps cycling through a source observable until it completes successfully.

```
public static IObservable<TSource> Retry<TSource>(this
IObservable<TSource> source);
public static IObservable<TSource> Retry<TSource>(this
IObservable<TSource> source, int retryCount);
```

Parameters

- source (this IObservable<TSource>) - The source observable
- retryCount (int) - The number of times to retry (default is unlimited)

Return Value

- IObservable<TSource> -

Remarks

When an observable emits an OnError message that may not always be the case (e.g. intermittent networking problem), then Retry can be used to try again to receive the elements.

6.3.67: Return

Returns a single value observable.

```
public static IObservable<TResult> Return<TResult>(TResult value);
public static IObservable<TResult> Return<TResult>(
TResult value, IScheduler scheduler);
```

Parameters

- value (TResult) - The value to insert into the observable at its first and only element
- scheduler (IScheduler) - The scheduler to use for this operator

Return Value

- IObservable<TResult> - An observable with a single element

Remarks

This operator provides an easy way to create an observable with a single element.

6.3.68: Sample

Sample elements from a source observable, with element selection based on

another observable or a `TimeSpan`.

```
public static IObservable<TSource> Sample<TSource, TSample>(
    this IObservable<TSource> source, IObservable<TSample> sampler);
public static IObservable<TSource> Sample<TSource>(
    this IObservable<TSource> source, TimeSpan interval);
public static IObservable<TSource> Sample<TSource>(
    this IObservable<TSource> source, TimeSpan interval,
    IScheduler scheduler);
```

Parameters

- source (`IObservable<TSource>`) -
- sampler (`IObservable<TSample>`) -
- interval (`TimeSpan`) -
- scheduler (`IScheduler`) - The scheduler to use for this operator

Return Values

- `IObservable<TSource>` - An observable containing the samples from the source

Remarks

This operator take a sample of elements from the source and creates an observable on the samples.

The choice of samples depending on which implementation to use. One takes an interval, in which case the element received nearest that time is emitted. Another takes an `IObservable<TSample>`, in which case when it emits an `OnNext` (value is ignored) that a sample from source is taken.

6.3.69: Scan

Runs an accumulator function over elements from a source and returns an observable which, for each source element, emits the accumulator's result.

```
public static IObservable<TSource> Scan<TSource>(
    this IObservable<TSource> source,
    Func<TSource, TSource, TSource> accumulator);
public static IObservable<TAccumulate> Scan<TSource, TAccumulate>(
    this IObservable<TSource> source, TAccumulate seed,
    Func<TAccumulate, TSource, TAccumulate> accumulator);
```

Parameters

- source (`IObservable<TSource>`) - The source to which the accumulator is to be applied
- accumulator (`Func<TSource, TSource, TAccumulate>`) - The accumulator function (when not using a seed)

- accumulator (Func<TSource, TSource, TAccumulate>) - The accumulator function (when using a seed)
- seed (TAccumulate) - A seed to initially add to the accumulator

Return Value

- IObservable<TSource> - A sequence of accumulator results

Remarks

Note that the accumulator is applied to each element, and the resulting observable contains the result for each element.

6.3.70: Select

Converts a source sequence to a result sequence using a projection function.

```
public static IObservable<TResult> Select<TSource, TResult>(
    this IObservable<TSource> source,
    Func<TSource, int, TResult> selector);
public static IObservable<TResult> Select<TSource, TResult>(
    this IObservable<TSource> source,
    Func<TSource, TResult> selector);
```

Parameters

- source (IObservable<TSource>) - The source elements
- selector (Func<TSource, TResult>) - The projection function
- selector (Func<TSource, int, TResult>) - The projection function which includes the element index

Return Value

- IObservable<TResult> - The projected elements

Remarks

The type for the result need not be the same as the type for the source.

6.3.71: SelectMany

```
public static IObservable<TResult> SelectMany<TSource, TResult>(
    this IObservable<TSource> source,
    Func<TSource, IEnumerable<TResult>> selector);
public static IObservable<TResult> SelectMany<TSource, TResult>(
    this IObservable<TSource> source,
    Func<TSource, IObservable<TResult>> selector);
public static IObservable<TOther> SelectMany<TSource, TOther>(
    this IObservable<TSource> source, IObservable<TOther> other);
public static IObservable<TResult>
    SelectMany<TSource, TCollection, TResult>(
```

```

        this IObservable<TSource> source,
        Func<TSource, IEnumerable<TCollection>> collectionSelector,
        Func<TSource, TCollection, TResult> resultSelector);
public static IObservable<TResult>
    SelectMany<TSource, TCollection, TResult>(
        this IObservable<TSource> source,
        Func<TSource, IObservable<TCollection>> collectionSelector,
        Func<TSource, TCollection, TResult> resultSelector);
public static IObservable<TResult> SelectMany<TSource, TResult>(
        this IObservable<TSource> source,
        Func<TSource, IObservable<TResult>> onNext,
        Func<Exception, IObservable<TResult>> onError,
        Func<IObservable<TResult>> onCompleted);

```

Parameters

- source (IObservable<TSource>) -
- other (IObservable<TSource>) -
- selector (Func<TSource, IEnumerable<TResult>>) - selector function
- collectionSelector (Func<TSource, IEnumerable<TResult>>) - collection selector function based on enumerable
- collectionSelector (Func<TSource, IObservable<TResult>>) - collection selector function based on observable
- resultSelector (Func<TSource, TCollection, TResult>) - result selector function
- onNext (Func<TSource, IObservable<TResult>>) - OnNext method
- onError (Func<Exception, IObservable<TResult>>) - OnError method
- onCompleted (Func<IObservable<TResult>>) - OnCompleted method

Return Value

- IObservable<TResult> - An observable of resulting elements

Remarks

This operator selects from the source based on a number of selector techniques.

6.3.72: SequenceEqual

Examines two observables and compares the next element from each. Returns an observable with true if all are equal, false otherwise.

```

public static IObservable<bool> SequenceEqual<TSource>(
    this IObservable<TSource> first, IObservable<TSource> second);
public static IObservable<bool> SequenceEqual<TSource>(
    this IObservable<TSource> first, IObservable<TSource> second,
    IEqualityComparer<TSource> comparer);

```

Parameters

- first (IObservable<TSource>) - The first source
- second (IObservable<TSource>) - The second source

Return Value

- IObservable<bool> - Contains an element which states if each pair of source elements match

Remarks

The comparer parameter can be used to supply a custom comparison function.

6.3.73: Single

For a source observable with exactly one element (possibly matching a predicate), return that element, otherwise, throw an exception.

```
public static TSource Single<TSource>(
    this IObservable<TSource> source);
public static TSource Single<TSource>(
    this IObservable<TSource> source, Func<TSource, bool> predicate);
```

Parameters

- source (IObservable<TSource>) - The source
- predicate (Func<TSource, bool>) - Defines a condition the element must match

Return Value

- TSource - The single element in the source observable

Remarks

The exception thrown is of type InvalidOperationException with a message of "Sequence contains no elements."

Note the return value is the element, not an IObservable of the element type.

6.3.74: SingleOrDefault

For a source observable with exactly element (possibly matching a predicate), return that element. If no element, return a default. If >1 elements, through exception.

```
public static TSource SingleOrDefault<TSource>(
    this IObservable<TSource> source);
public static TSource SingleOrDefault<TSource>(
    this IObservable<TSource> source, Func<TSource, bool> predicate);
```

Parameters

- source (IObservable<TSource>) - The source
- predicate (Func<TSource, bool>) - Defines a condition the element must

match

Return Value

- TSource - The single element in the source observable, or the default

Remarks

Note the return value is the element, not an IObservable of the element type.

6.3.75: Skip

Skips the first count of elements and returns the rest.

```
public static IObservable<TSource> Skip<TSource>(
    this IObservable<TSource> source, int count);
```

Parameters

- source (this IObservable<TSource>) - The source elements
- count (int) - The count of initial elements to ignore

Return Value

- IObservable<TSource> - The source elements, excluding the first count of elements

Remarks

As soon as this operator has received count+1 elements from the source, it can start emitting its elements.

Contrast the use of Skip/SkipLast/SkipUntil/SkipWhile operators with the Take/TakeUntil/TakeWhile/TakeLast operators.

6.3.76: SkipLast

Skips the last count of elements and returns all the previous elements.

```
public static IObservable<TSource> SkipLast<TSource>(
    this IObservable<TSource> source, int count);
```

Parameters

- source (this IObservable<TSource>) - The source elements
- count (int) - The count of last elements to ignore

Return Value

- IObservable<TSource> - The source elements, excluding the last count of elements

Remarks

This operator must ensure it has received at least count+1 elements, before it

emits its first element (so that it is in a position to ignore the remainder, if it receives OnCompleted from the source).

Contrast the use of Skip/SkipLast/SkipWhile operators with the Take/TakeUntil/TakeWhile/TakeLast operators.

6.3.77: SkipUntil

Skips elements until the other observable emits an element.

```
public static IObservable<TSource> SkipUntil<TSource, TOther>(
    this IObservable<TSource> source, IObservable<TOther> other);
```

Parameters

- source (this IObservable<TSource>) - The source elements
- other (IObservable<TOther>) - An observable which emits an element when the skipping of the source is to stop

Return Value

- IObservable<TSource> - The source elements, excluding the first count of elements

Remarks

This operator skips all received source elements until other emits an element.

Contrast the use of Skip/SkipLast/SkipUntil/SkipWhile operators with the Take/TakeUntil/TakeWhile/TakeLast operators.

6.3.78: SkipWhile

Skips source elements while a conditions holds.

```
public static IObservable<TSource> SkipWhile<TSource>(
    this IObservable<TSource> source, Func<TSource, bool> predicate);
public static IObservable<TSource> SkipWhile<TSource>(
    this IObservable<TSource> source,
    Func<TSource, int, bool> predicate);
```

Parameters

- source (this IObservable<TSource>) - The source elements
- other (IObservable<TOther>) - An observable which emits an element when the skipping of the source is to stop

Return Value

- IObservable<TSource> - The source elements, excluding the first count of elements
- predicate (Func<TSource, bool>) - A function which is passed in the element and returns a bool, stating whether the skipping of elements is to

continue

- predicate (Func<TSource, int, bool>) - A function which is passed in the element and the zero-based index, and returns a bool, stating whether the skipping of elements is to continue

Remarks

This operator skips all received source elements until one predicate calls returns false - after that, all later elements are returned.

Contrast the use of Skip/SkipLast/SkipUntil/SkipWhile operators with the Take/TakeUntil/TakeWhile/TakeLast operators.

6.3.79: Start

Asynchronous execution of an action or function.

```
public static IObservable<Unit> Start(Action action);
public static IObservable<TSource> Start<TSource>(
    Func<TSource> function);
public static IObservable<Unit> Start(
    Action action, IScheduler scheduler);
public static IObservable<TSource> Start<TSource>(
    Func<TSource> function, IScheduler scheduler);
```

Parameters

- action (Action) - The action to execute asynchronously
- function (Func<TSource>) - The function to execute asynchronously

Return Value

- IObservable<Unit> - The result of executing the action
- IObservable<Tsource> - The result of executing the function
- scheduler (IScheduler) - The scheduler to use for this operator

Remarks

This operator converts an action/function to run asynchronously. The action has no return value and when it finishes executing, the result observable emits one element (of type Unit) and then emits OnCompleted. The function returns a TSource instance and when it finishes executing, the result observable emits one TSource instance which is the return value of the function, and then emits OnCompleted.

6.3.80: StartWith

Adds elements to the start of a sequence.

```
public static IObservable<TSource> StartWith<TSource>(
```

```
        this IObservable<TSource> source, params TSource[] values);  
public static IObservable<TSource> StartWith<TSource>(  
    this IObservable<TSource> source, IScheduler scheduler,  
    params TSource[] values);
```

Parameters

- source (IObservable<TSource>) - The source observable
- values (params TSource[]) - The elements to prepend to the sequence
- scheduler (IScheduler) - The scheduler to use for this operator

Return Value

- IObservable<TSource> - A sequence starting with the prepending elements and continuing with the elements from the source

Remarks

Note that normally in Rx the scheduler parameter comes last, but since params are used here, the scheduler is the penultimate parameter.

6.3.81: Subscribe

Enables an observer to subscribe to an enumerable.

```
public static IDisposable Subscribe<TSource>(  
    this IEnumerable<TSource> source, IObservable<TSource> observer);  
public static IDisposable Subscribe<TSource>(  
    this IEnumerable<TSource> source, IObservable<TSource> observer,  
    IScheduler scheduler);
```

Parameters

- source (IEnumerable<TSource>) - The source enumerable
- observer (IObservable<TSource>) - The observer to subscribe to the enumerable
- scheduler (IScheduler) - The scheduler to use for this operator

Return Value

- IDisposable - The subscription, which can be disposed of

Remarks

Normally an observer subscribes to an observable but this operator allow an observer to subscribe to an enumerable. Note that the return value is an IDisposable, unlike most other operators, which return an IObservable.

6.3.82: SubscribeOn

Defines the scheduler/synchronization context to use for subscribing.

```
public static IObservable<TSource> SubscribeOn<TSource>(
```

```

        this IObservable<TSource> source, IScheduler scheduler);
    public static IObservable<TSource> SubscribeOn<TSource>(
        this IObservable<TSource> source, SynchronizationContext context);

```

Parameters

- source (IObservable<TSource>) - The source to subscribe to
- scheduler (IScheduler) - The scheduler to use
- context (SynchronizationContext) - The synchronization context to use

Return Value

- IObservable<TSource> - An observable to which subscriptions execute via the specified scheduler/context

Remarks

Note that the Subscribe and SubscriberOn operators are quite different. The former has an enumerable as the source, while the latter has an observable as the source.

6.3.83: Sum

Sums up numeric elements.

```

    public static IObservable<decimal?> Sum(this IObservable<decimal?>
source);
    public static IObservable<decimal> Sum(this IObservable<decimal>
source);
    public static IObservable<double?> Sum(this IObservable<double?>
source);
    public static IObservable<double> Sum(this IObservable<double>
source);
    public static IObservable<float?> Sum(this IObservable<float?>
source);
    public static IObservable<float> Sum(this IObservable<float> source);
    public static IObservable<int?> Sum(this IObservable<int?> source);
    public static IObservable<int> Sum(this IObservable<int> source);
    public static IObservable<long?> Sum(this IObservable<long?> source);
    public static IObservable<long> Sum(this IObservable<long> source);

```

Parameters

- source (IObservable< *numeric type* >) - The source sequence

Return Value

- IObservable < *numeric type* > - The result in a single element observable

Remarks

This operator calculates the sum of a sequence of numeric types.

Note the type of the returned sequence can be either float, float?, double, double?, decimal or decimal?. The type of the source sequence can be decimal,

decimal?, double, double?, float, float?, int, int?, long, or long?.

6.3.84: Switch

From a group of source observables, pick the last one, and emit its elements.

```
public static IObservable<TSource> Switch<TSource>(
    this IObservable<IObservable<TSource>> sources);
```

Parameters

sources (IObservable<IObservable<TSource>>) - An observable of observables of source elements

Return Value

IObservable<TSource> - The source elements of the last source observable

Remarks

Which observable that is picked is the one that is last to be emitted by the observable of observables. It is not dependent on the timing of element emitting from the individual observables themselves.

6.3.85: Synchronize

Synchronizes access to an observable when the Rx Contract is not adhered to.

```
public static IObservable<TSource> Synchronize<TSource>(
    this IObservable<TSource> source);
public static IObservable<TSource> Synchronize<TSource>(
    this IObservable<TSource> source, object gate);
```

Parameters

- source - The source observable (which might not synchronize its output)
- gate (object) - The object to use as a gate

Return Value

- IObservable<TSource>) - an observable which does synchronize its output

Remarks

The Rx Contract states that observables should synchronize its output. When this does not occur, the Synchronize operator may be used to force it.

6.3.86: Take

Returns an observable with a certain number of the first elements in the source.

```
public static IObservable<TSource> Take<TSource>(
    this IObservable<TSource> source, int count);
```

Parameters

- source (IObservable<TSource>) - The source of the elements

- count (int) - The number of elements from the source to return

Return Value

- IObservable<TSource> - A sequence of the first “count” elements from the source

Remarks

This observable returns elements from the beginning of the source.

To return elements from the end, use TakeLast. To take elements until another observable has an element, use TakeUntil. To take elements while a predicate condition holds, use TakeWhile.

6.3.87: TakeLast

Returns an observable with a certain number of the last elements in the source.

```
public static IObservable<TSource> TakeLast<TSource>(
    this IObservable<TSource> source, int count);
```

Parameters

- source (IObservable<TSource>) - The source of the elements
- count (int) - The number of elements from the source to return

Return Value

- IObservable<TSource> - A sequence of the last “count” elements from the source

Remarks

This observable returns elements from the end of the source. It does not know the end of the source until it receives OnCompleted from it, so it must wait until then before it emits its elements.

To return elements from the beginning, use Take. To take elements until another observable has an element, use TakeUntil. To take elements while a predicate condition holds, use TakeWhile.

6.3.88: TakeUntil

Takes elements until another observable has an element.

```
public static IObservable<TSource> TakeUntil<TSource, TOther>(
    this IObservable<TSource> source, IObservable<TOther> other);
```

Parameters

- source (IObservable<TSource>) - The source of the elements
- other (IObservable<TOther>) - The other observable

Return Value

- `IObservable<TSource>` - A sequence of the elements from the source, until the other observable emits an element

Remarks

This observable returns elements from the beginning of the source, until the other observable emits an element.

To return elements from the beginning, use `Take`. To return elements from the end, use `TakeLast`. To take elements while a predicate condition holds, use `TakeWhile`.

6.3.89: TakeWhile

Takes elements while a predicate condition holds.

```
public static IObservable<TSource> TakeWhile<TSource>(
    this IObservable<TSource> source,
    Func<TSource, bool> predicate);
public static IObservable<TSource> TakeWhile<TSource>(
    this IObservable<TSource> source,
    Func<TSource, int, bool> predicate);
```

Parameters

- `source` (`IObservable<TSource>`) - The source of the elements
- `predicate` (`Func<TSource, bool>`) - A function which is passed in the element and returns a `bool`, stating whether this element is to be taken - if false, this element, and all later ones are ignored
- `predicate` (`Func<TSource, int, bool>`) - A function which is passed in the element and the zero-based index, and returns a `bool`, stating whether this element is to be taken - if false, this element, and all later ones are ignored

Return Value

- `IObservable<TSource>` - A sequence of the first “count” elements from the source

Remarks

This observable returns elements while a predicate condition holds. There are two distinct predicate signatures - the difference is one has an `int` parameter, which starts at 0 for the first elements, and increments for each later element.

To return elements from the beginning, use `Take`. To return elements from the end, use `TakeLast`. To take elements until another observable has an element, use `TakeUntil`.

6.3.90: Then

Creates a plan based on a single source observable, which is matched when the source has an element and uses the selector.

```
public static Plan<TResult> Then<TSource, TResult>(this
    IObservable<TSource> source, Func<TSource, TResult> selector);
```

Parameters

- source (IObservable<TSource>) - The source of the elements
- selector (Func<TSource, TResult>) - The selector function which returns the projected result for the element

Return Value

- Plan<TResult> - A plan

Remarks

One can execute one or more plans by passing them to Observable.When() method.

One can create a plan based on more than one source observable by creating a pattern (using Observable.And) consisting of two observables which returns a Pattern instance, optionally adding additional observables by calling Pattern.And() for that instance, and then calling Pattern.Then for it.

6.3.91: Throttle

Throttles a sequence, so that an element received within dueTime of the next element is discarded.

```
public static IObservable<TSource> Throttle<TSource>(
    this IObservable<TSource> source, TimeSpan dueTime);
public static IObservable<TSource> Throttle<TSource>(
    this IObservable<TSource> source, TimeSpan dueTime,
    IScheduler scheduler);
```

Parameters

- source () - The sequence to be throttled
- dueTime (TimeSpan) - The period outside of which elements can be accepted
- scheduler (IScheduler) - The scheduler to use for this operator

Return Value

- IObservable<TResult> - An observable which will only have elements that have been received outside of dueTime of the succeeding element

Remarks

When there is only interest in fresh elements in a sequence, throttling can be used to eliminate elements that come too close to a later element.

This is not a sampling operator, in the sense that a sequence whose elements will be less than `dueTime` apart will return no elements when throttled at `dueTime` (if the sequence completes, the last element will be returned). There is a separate `Sample` operator.

6.3.92: Throw

Creates an observable, which will only observe one `OnError`, based on the exception parameter.

```
public static IObservable<TResult> Throw<TResult>(
    Exception exception);
public static IObservable<TResult> Throw<TResult>(
    Exception exception, IScheduler scheduler);
```

Parameters

- `Exception` - the exception to pass to `OnError`
- `scheduler (IScheduler)` - The scheduler to use for this operator

Return Value

- `IObservable<TResult>` - An observable which will call `OnError` once with no calls to `OnNext` or `OnCompleted`

Remarks

This type is useful for testing error handling code.

Note that the type parameter for the returned observable must be explicitly set when calling `Throw`, since the compiler has no way of inferring what it is.

6.3.93: TimeInterval

Detects the interval between source `OnNext` calls.

```
public static IObservable<TimeInterval<TSource>> TimeInterval<TSource>(
    this IObservable<TSource> source);
public static IObservable<TimeInterval<TSource>> TimeInterval<TSource>(
    this IObservable<TSource> source, IScheduler scheduler);
```

Parameters

- `source (IObservable<TSource>)` - The elements whose intervals are to be recorded
- `scheduler (IScheduler)` - The scheduler to use for this operator

Return Value

- `IObservable<TimeInteval<TSource>>` - An observable of `TimeInterval` instances based on source elements

Remarks

The `System.Reactive.TimeInterval<T>` struct maintains two properties - `Interval` (of type `TimeSpan`) and `Value` (of type `T`). It is used in association with `System.Reactive.Linq.Observable.TimeInterval` to record the interval times between received elements.

elements received from a source observable are used to create `TimeInterval` instances with the value set to the element and `Interval` set to the interval between `OnNext` calls, and a sequence of these is the return value.

This operator is useful when there is a need to know the timing between arrival of elements.

The difference between `TimeInterval` and `Timestamp` operators is that the former returns the intervals (`TimeSpan`) between source `OnNext` calls, whereas the latter returns the time (`DateTimeOffset`) of the source `OnNext` call itself.

6.3.94: Timeout

Returns a result observable with elements from the source if they are received within a timeout period, otherwise either throws an exception or returns elements from another observable.

```
public static IObservable<TSource> Timeout<TSource>(
    this IObservable<TSource> source, DateTimeOffset dueTime);
public static IObservable<TSource> Timeout<TSource>(
    this IObservable<TSource> source, TimeSpan dueTime);
public static IObservable<TSource> Timeout<TSource>(
    this IObservable<TSource> source, DateTimeOffset dueTime,
    IObservable<TSource> other);
public static IObservable<TSource> Timeout<TSource>(
    this IObservable<TSource> source, DateTimeOffset dueTime,
    IScheduler scheduler);
public static IObservable<TSource> Timeout<TSource>(
    this IObservable<TSource> source, TimeSpan dueTime,
    IObservable<TSource> other);
public static IObservable<TSource> Timeout<TSource>(
    this IObservable<TSource> source, TimeSpan dueTime,
    IScheduler scheduler);
public static IObservable<TSource> Timeout<TSource>(
    this IObservable<TSource> source, DateTimeOffset dueTime,
    IObservable<TSource> other, IScheduler scheduler);
public static IObservable<TSource> Timeout<TSource>(
    this IObservable<TSource> source, TimeSpan dueTime,
    IObservable<TSource> other, IScheduler scheduler);
```

Parameters

- `dueTime (DateTimeOffset)` - Delay until the first `OnNext` is called
- `dueTime (TimeSpan)` - Delay until the first `OnNext` is called
- `scheduler (IScheduler)` - The scheduler to use for this operator

- other (IObservable<TSource>) - Delay between OnNext calls

Return Value

- IObservable<long> - An observable of long, where the long value is a counter of expired periods - starts at 0

Remarks

There are 8 implementations of Timeout. One can divide them into two groups - those that take a scheduler parameter (specifies the scheduler to use) and those that don't (uses the default scheduler).

Of the 4 in each group, one can divide again into two that take a TimeSpan dueTime parameter, and two that take a DateTimeOffset dueTime parameter. For each pair, one takes an additional parameter, IObservable<TSource> other.

The returned observable is a different instance from the source observable. If elements from the source observable are received within the timeout period, these are made available in the returned observable. For the implementations that don't take an other parameter, if the elements from the source are received outside the timeout period, then the returned observable generates an OnError message with a System.TimeoutException exception, with the message stating "The operation has timed out." For the implementations that do take an other parameter, an element from it is used.

6.3.95: Timer

Creates a sequence which emits an element each dueTime.

```
public static IObservable<long> Timer(DateTimeOffset dueTime);
public static IObservable<long> Timer(TimeSpan dueTime);
public static IObservable<long> Timer(
    DateTimeOffset dueTime, IScheduler scheduler);
public static IObservable<long> Timer(
    DateTimeOffset dueTime, TimeSpan period);
public static IObservable<long> Timer(
    TimeSpan dueTime, IScheduler scheduler);
public static IObservable<long> Timer(
    TimeSpan dueTime, TimeSpan period);
public static IObservable<long> Timer(DateTimeOffset dueTime,
    TimeSpan period, IScheduler scheduler);
public static IObservable<long> Timer(TimeSpan dueTime,
    TimeSpan period, IScheduler scheduler);
```

Parameters

- dueTime (DateTimeOffset) - Delay until the first OnNext is called
- dueTime (TimeSpan) - Delay until the first OnNext is called
- scheduler (IScheduler) - The scheduler to use for this operator
- period (TimeSpan) - Delay between OnNext calls

Return Value

- `IObservable<long>` - An observable of long, where the long value is a counter of expired periods - starts at 0

Remarks

Similar to `timeout`, there are 8 implementations of timer. One can divide them into two groups - those that take a scheduler parameter (specifies the scheduler to use) and those that don't (uses the default scheduler).

Of the 4 in each group, one can divide again into two that take a `(TimeSpan)` period parameter, and two that don't. Each pair start with a `dueTime` parameter, one that is of type `dateTimeOffset`, and the other of type `TimeSpan`.

Timer returns an `IObservable<long>`. For those Timer implementations that do not take a period parameter, after `dueTime` elapses, the returned observable generates an `OnNext(0)` followed by an `OnComplete`.

For those Timer implementations that do take a period parameter, after `dueTime` elapses, it generates `OnNext(0)`, then waits `dueTime` again, then generates `OnNext(1)`, then waits `dueTime` again, then generates `OnNext(2)`, and so on. It never generates an `OnCompleted`.

If using a period, and there is a wish to have the first `OnNext` called immediately, set `dueTime` to `TimeSpan.Zero`.

6.3.96: Timestamp

Timestamps elements in the source observable.

```
public static IObservable<Timestamped<TSource>>
    Timestamp<TSource>(this IObservable<TSource> source);
public static IObservable<Timestamped<TSource>>
    Timestamp<TSource>(this IObservable<TSource> source,
        IScheduler scheduler);
```

Parameters

- `source (IObservable<TSource>)` - The elements to be timestamped
- `scheduler (IScheduler)` - The scheduler to use for this operator

Return Value

- `IObservable<Timestamped<TSource>>` - An observable of timestamped instances based on source elements

Remarks

The `System.Reactive.Timestamped<T>` struct maintains two properties - `TimeStamp` (of type `DateTimeOffset`) and `Value` (of type `T`). It is used in

association with `System.Reactive.Linq.Observable.Timestamped` to effectively timestamp received elements.

elements received from a source observable are used to create `Timestamped` instances with the value set to the element and `Timestamp` set to when the element was received, and a sequence of these is the return value.

This operator is useful when there is a need to know the timing of arrival of elements.

6.3.97: ToArray

Consumes all elements from the source until `OnCompleted` is received, and then produces an array with all consumed elements.

```
public static IObservable<TSource[]> ToArray<TSource>(
    this IObservable<TSource> source);
```

Parameters

- `source (IObservable<TSource>)` - The elements to be added to the array

Return Value

- `IObservable<TSource[]>` - An array of the source elements

Remarks

This operator returns an observable that only emits an element when the source sequence completes.

This operator creates an array, adds each source element to it, and when the source completes, emits one result element, the array of received source elements, and then emits `OnCompleted`.

6.3.98: ToAsync

Converts an input action or function parameter to an output function whose return value is `IObservable`.

```
public static Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12,
T13, T14, T15, T16, IObservable<Unit>> ToAsync<T1, T2, T3, T4, T5, T6,
T7, T8, T9, T10, T11, T12, T13, T14, T15, T16>(this Action<T1, T2, T3,
T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14, T15, T16> action);
public static Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12,
T13, T14, T15, IObservable<Unit>> ToAsync<T1, T2, T3, T4, T5, T6, T7, T8,
T9, T10, T11, T12, T13, T14, T15>(this Action<T1, T2, T3, T4, T5, T6, T7,
T8, T9, T10, T11, T12, T13, T14, T15> action);
public static Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12,
T13, T14, IObservable<Unit>> ToAsync<T1, T2, T3, T4, T5, T6, T7, T8, T9,
T10, T11, T12, T13, T14>(this Action<T1, T2, T3, T4, T5, T6, T7, T8, T9,
T10, T11, T12, T13, T14> action);
public static Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12,
T13, IObservable<Unit>> ToAsync<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10,
```



```

IScheduler scheduler);
    public static Func<T1, T2, T3, T4, T5, T6, T7, T8,
IObservable<TResult>> ToAsync<T1, T2, T3, T4, T5, T6, T7, T8,
TResult>(this Func<T1, T2, T3, T4, T5, T6, T7, T8, TResult> function,
IScheduler scheduler);
    public static Func<T1, T2, T3, T4, T5, T6, T7, IObservable<TResult>>
ToAsync<T1, T2, T3, T4, T5, T6, T7, TResult>(this Func<T1, T2, T3, T4,
T5, T6, T7, TResult> function, IScheduler scheduler);
    public static Func<T1, T2, T3, T4, T5, T6, IObservable<TResult>>
ToAsync<T1, T2, T3, T4, T5, T6, TResult>(this Func<T1, T2, T3, T4, T5,
T6, TResult> function, IScheduler scheduler);
    public static Func<T1, T2, T3, T4, T5, IObservable<TResult>>
ToAsync<T1, T2, T3, T4, T5, TResult>(this Func<T1, T2, T3, T4, T5,
TResult> function, IScheduler scheduler);
    public static Func<T1, T2, T3, T4, IObservable<TResult>> ToAsync<T1,
T2, T3, T4, TResult>(this Func<T1, T2, T3, T4, TResult> function,
IScheduler scheduler);
    public static Func<T1, T2, T3, IObservable<TResult>> ToAsync<T1, T2,
T3, TResult>(this Func<T1, T2, T3, TResult> function, IScheduler
scheduler);
    public static Func<T1, T2, IObservable<TResult>> ToAsync<T1, T2,
TResult>(this Func<T1, T2, TResult> function, IScheduler scheduler);
    public static Func<IObservable<TResult>> ToAsync<TResult>(this
Func<TResult> function, IScheduler scheduler);

```

Parameters

- action (Action<T1, ...>) - the input action
- function (this Func<T1, ..., TResult>) - The input function
- scheduler (IScheduler) - The scheduler to use for this operator

Return Value

- Func<T1, ..., IObservable<Unit>> - The result when the input is an action
- Func<T1, ..., IObservable<TResult>> - The result when the input is a function

Remarks

The return value of ToAsync is not an IObservable, rather it is a Func (whose return value is an observable).

6.3.99: ToDictionary

Consumes all elements from the source until OnCompleted is received, and then produces a dictionary with all consumed elements.

```

public static IObservable<IDictionary<TKey, TSource>>
ToDictionary<TSource, TKey>(this IObservable<TSource> source,
Func<TSource, TKey> keySelector);
public static IObservable<IDictionary<TKey, TElement>>
ToDictionary<TSource, TKey, TElement>(
this IObservable<TSource> source, Func<TSource, TKey> keySelector,
Func<TSource, TElement> elementSelector);
public static IObservable<IDictionary<TKey, TSource>>

```

```

    ToDictionary<TSource, TKey>(this IObservable<TSource> source,
    Func<TSource, TKey> keySelector,
    IEqualityComparer<TKey> comparer);
    public static IObservable<IDictionary<TKey, TElement>>
    ToDictionary<TSource, TKey, TElement>(
    this IObservable<TSource> source, Func<TSource, TKey> keySelector,
    Func<TSource, TElement> elementSelector,
    IEqualityComparer<TKey> comparer);

```

Parameters

- source (IObservable<TSource>) - The elements to be added to the dictionary
- keySelector (Func<TSource, TKey>) - A function to select the key
- elementSelector (Func<TSource, TElement>) - A function to select the element
- comparer (IEqualityComparer<TKey>) - A comparer for the keys

Return Value

- IObservable<Dictionary<TKey, TSource> - A dictionary of the source elements

Remarks

This operator returns an observable that only emits an element when the source sequence completes.

This operator creates a dictionary, passes each source element to the keyselector function to generate a key and inserts the element in the dictionary using that key, and when the source completes, emits one result element, the dictionary, and then emits OnCompleted.

6.3.100: ToEnumerable

Converts an observable to an enumerable.

```

    public static IEnumerable<TSource> ToEnumerable<TSource>(
    this IObservable<TSource> source);

```

Parameters

- source (IObservable<TSource>) - The elements to be added to the enumerable

Return Value

- IEnumerable<TSource> - the source elements as an enumerable

Remarks

There is an important difference between the operation of ToEnumerable in

contrast with `ToArray/ToDictionary`. With `ToEnumerable`, the elements are made available in the enumerable as soon as each is received from the source, whereas with `ToArray/ToDictionary`, all elements have to be received, and after that the array/dictionary is emitted.

6.3.101: ToEvent

Converts from an `IObservable` to an `IEventSource`.

```
public static IEventSource<TSource> ToEvent<TSource>(
    this IObservable<TSource> source);
public static IEventSource<Unit> ToEvent(
    this IObservable<Unit> source);
```

Parameters

- `source (IObservable<TSource>)` - The observable to be converted

Return Value

- `IEventSource<TSource>` - The result of the conversion

Remarks

A `System.Reactive.IEventSource` is an interface with one event, `OnNext`. When the source observable reports a new element with an `OnNext` method call, the event source raises the `OnNext` event. Client code can attach event handlers to the event source.

The difference between `ToEvent` and `ToEventPattern` is the type of resulting event - `ToEvent` is an event without any standard pattern, whereas `ToEventPattern` is an event which follows the `sender/EventArgs` pattern.

6.3.102: ToEventPattern

Converts from an `IObservable` to an `IEventPatternSource`.

```
public static IEventPatternSource<TEventArgs>
    ToEventPattern<TEventArgs>(
        this IObservable<EventPattern<TEventArgs>> source) where
        TEventArgs : EventArgs;
```

Parameters

- `source (IObservable<EventPattern<TEventArgs>)` - The observable of event patterns to be converted

Return Value

- `IEventPatternSource<TEventArgs>` - The result of the conversion

Remarks

A `System.Reactive.IEventPatternSource` is an interface with one event, `OnNext`. When the source observable calls `OnNext`, this event is raised, and it following

the recommended .NET event pattern.

6.3.103: ToList

The key property.

```
public static IObservable<IList<TSource>> ToList<TSource>(
    this IObservable<TSource> source);
```

Parameters

- source (IObservable<TSource>) - The elements to be added to the list

Return Value

- IObservable<IList<TSource>> - An observable of a list of source elements

Remarks

This operator returns an observable that only emits an element when the source sequence completes.

This operator creates a list, adds each source element to it, and when the source completes, emits one result element, the list of received source elements, and then emits OnCompleted.

6.3.104: ToLookup

The key property.

```
public static IObservable<ILookup<TKey, TSource>> ToLookup<TSource,
TKey>(this IObservable<TSource> source, Func<TSource, TKey> keySelector);
public static IObservable<ILookup<TKey, TElement>> ToLookup<TSource,
TKey, TElement>(this IObservable<TSource> source, Func<TSource, TKey>
keySelector, Func<TSource, TElement> elementSelector);
public static IObservable<ILookup<TKey, TSource>> ToLookup<TSource,
TKey>(this IObservable<TSource> source, Func<TSource, TKey> keySelector,
IEqualityComparer<TKey> comparer);
public static IObservable<ILookup<TKey, TElement>> ToLookup<TSource,
TKey, TElement>(this IObservable<TSource> source, Func<TSource, TKey>
keySelector, Func<TSource, TElement> elementSelector,
IEqualityComparer<TKey> comparer);
```

Parameters

- source (IObservable<TSource>) - The elements to be added to the lookup
- keySelector (Func<TSource, TKey>) - A function to select the key
- elementSelector (Func<TSource, TElement>) - A function to select the element
- comparer (IEqualityComparer<TKey>) - A comparer for the keys

Return Value

- IObservable<ILookup<TKey, TSource>> - A lookup of the source elements

Remarks

This operator returns an observable that only emits an element when the source sequence completes.

6.3.105: ToObservable

Converts an enumerable to an observable.

```
public static IObservable<TSource> ToObservable<TSource>(
    this IEnumerable<TSource> source);
public static IObservable<TSource> ToObservable<TSource>(
    this IEnumerable<TSource> source, IScheduler scheduler);
```

Parameters

- source (this IEnumerable<TSource>) - The source enumerable
- scheduler (IScheduler) - the scheduler to use for this operator

Return Value

- IObservable<TSource> - The observable whose elements are those from the enumerable

Remarks

When one has an array or a list or any other form of enumerable, and one wishes to make it into an observable, use this operator.

6.3.106: Using

Similar to the using keyword in C#, this operator allows use of an IDisposable-based resource, and ensures its Dispose() method is called when finished.

```
public static IObservable<TSource> Using<TSource, TResource>(
    Func<TResource> resourceFactory,
    Func<TResource, IObservable<TSource>> observableFactory)
    where TResource : IDisposable;
```

Parameters

- resourceFactory (Func<TResource>) - A function to create the IDisposable-based resource
- observableFactory (Func<TResource, IObservable<TSource>>) - A function whose input is the constructed resource and whose output is an observable of TSource

Return Value

- IObservable<TSource> - The observable to use

Remarks

When using instances whose types implement IDisposable, there is usually a

desire to call its `Dispose()` method when it is no longer needed. It can be done manually, but as one can see with “using” in C#, it is safer (more reliable) to have a construct whereby the instance to be disposed is automatically disposed when no longer needed.

The `using` operator here is similar to “using” in C#. Two factory functions are involved. The `resourceFactory` function is used to create a resource, and this is passed in to the `observableFactory` function where the observable is created. Once that observable has emitted its `OnCompleted`, the resource's `Dispose()` method will be called.

6.3.107: When

Executes a set of plans.

```
public static IObservable<TResult> When<TResult>(
    params Plan<TResult>[] plans);
public static IObservable<TResult> When<TResult>(
    this IEnumerable<Plan<TResult>> plans);
```

Parameters

- `plans (params Plan<TResult>[])` - The plans to use
- `plans (this IEnumerable<Plan<TResult>>)` - The plans to use

Return Value

- `IObservable<TResult>` - The result of executing the plans

Remarks

Plans can be generated by either calling `Observable.Then` or `System.Reactive.Joins.Pattern<>.Then()`.

6.3.108: Where

Runs elements through a predicate and only passes on elements that satisfy the condition.

```
public static IObservable<TSource> Where<TSource>(
    this IObservable<TSource> source, Func<TSource, bool> predicate);
public static IObservable<TSource> Where<TSource>(
    this IObservable<TSource> source,
    Func<TSource, int, bool> predicate);
```

Parameters

- `source (IObservable<TSource>)` -
- `predicate (Func<TSource, bool>)` -
- `predicate (Func<TSource, int, bool>)` -

Return Value

- `IObservable<TSource>` -

Remarks

Two variations of the predicate function are supported, one that takes the zero-based element index (int) and one that does not.

6.3.109: Window

A windowing operator that provides an `IObservable` on a window of source elements.

```
public static IObservable<IObservable<TSource>>
    Window<TSource, TWindowClosing>(this IObservable<TSource> source,
        Func<IObservable<TWindowClosing>> windowClosingSelector);
public static IObservable<IObservable<TSource>> Window<TSource>(
    this IObservable<TSource> source, int count);
public static IObservable<IObservable<TSource>> Window<TSource>(
    this IObservable<TSource> source, TimeSpan timeSpan);
public static IObservable<IObservable<TSource>> Window<TSource>(
    this IObservable<TSource> source, int count, int skip);
public static IObservable<IObservable<TSource>>
    Window<TSource, TWindowOpening, TWindowClosing>(
        this IObservable<TSource> source,
        IObservable<TWindowOpening> windowOpenings,
        Func<TWindowOpening, IObservable<TWindowClosing>>
            windowClosingSelector);
public static IObservable<IObservable<TSource>> Window<TSource>(
    this IObservable<TSource> source,
    TimeSpan timeSpan, int count);
public static IObservable<IObservable<TSource>> Window<TSource>(
    this IObservable<TSource> source, TimeSpan timeSpan,
    IScheduler scheduler);
public static IObservable<IObservable<TSource>> Window<TSource>(
    this IObservable<TSource> source, TimeSpan timeSpan,
    TimeSpan timeShift);
public static IObservable<IObservable<TSource>> Window<TSource>(
    this IObservable<TSource> source, TimeSpan timeSpan,
    int count, IScheduler scheduler);
public static IObservable<IObservable<TSource>> Window<TSource>(
    this IObservable<TSource> source, TimeSpan timeSpan,
    TimeSpan timeShift, IScheduler scheduler);
```

Parameters

- `source` (`IObservable<TSource>`) - The source to which the window operator should be applied
- `count` (int) - The maximum number of elements in a window
- `timeSpan` (`TimeSpan`) - The duration the window is to remain open
- `timeShift` (`TimeSpan`) - The timeshift to apply
- `windowOpenings` (`IObservable<TWindowOpening>`) - An observable which indicates when a window is to open
- `windowClosingSelector` (`Func<IObservable<TWindowClosing>>`) - A

function which indicates when the window closes

- scheduler (IScheduler) - The scheduler to use for this operator

Return Value

IObservable<IObservable<TSource>> - The window, which itself can be observed

Remarks

There is an operator similar to Window called Buffer and it is important to understand the difference between them. Both are windowing operators in the sense they apply a window (a sub-division) to the source sequence (based on count, a condition, time, etc.). The difference is that Buffer waits until its window is closed and then emits an IList with the elements in the window, whereas Window emits an IObservable on the window itself (Window is an IObservable<IObservable<TSource>, whereas Buffer returns an IObservable<IList<TSource>>). In other words, the Window operator does not need to have any source elements before it can return its IObservable, whereas the Buffer operator needs to have received all its source elements so it can fill the IList and return it.

6.3.110: Zip

Combines the elements from two observables based on a selector function.

```
public static IObservable<TResult> Zip<TFirst, TSecond, TResult>(
    this IObservable<TFirst> first, IEnumerable<TSecond> second,
    Func<TFirst, TSecond, TResult> resultSelector);
public static IObservable<TResult> Zip<TFirst, TSecond, TResult>(
    this IObservable<TFirst> first, IObservable<TSecond> second,
    Func<TFirst, TSecond, TResult> resultSelector);
```

Parameters

- first (this IObservable<TFirst>) - The first source observable
- second (IEnumerable<TSecond>) - An enumerable of TSecond elements
- second (IObservable<TSecond>) - An observable of TSecond elements
- resultSelector (Func<TFirst, TSecond, TResult>) - A selector function

Return Value

- IObservable<TResult> - The zipped result

Remarks

There are two implementations provided, in one the second parameter is an observable and in the other it is an enumerable. Note the types of the first (TFirst) observable and second (TSecond) source observable/enumerable may be

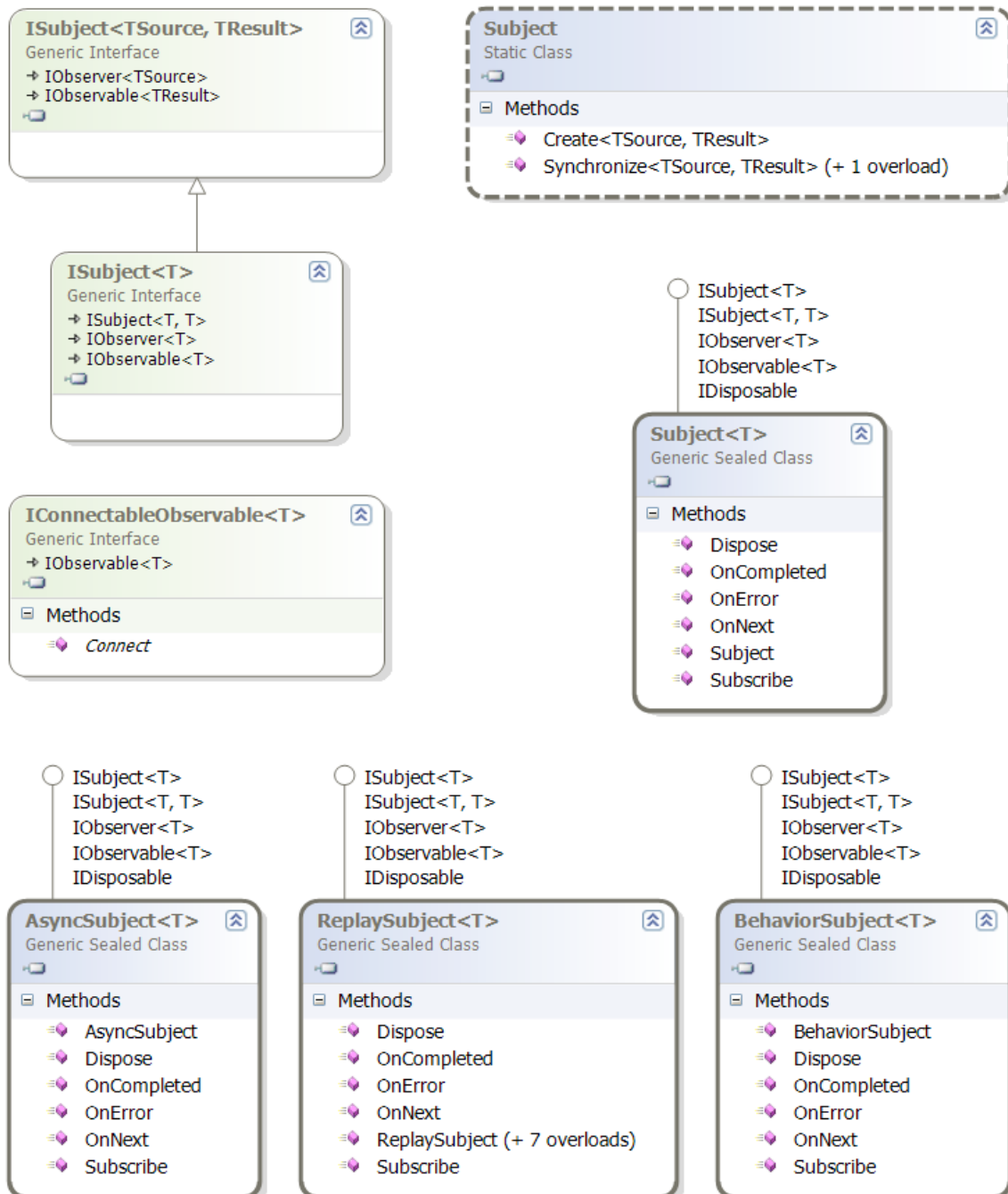
different and that the type of the returned observable is `TResult`, not `TFirst` or `TSecond`.

There is also a separate `Merge` operator, which similar merges elements from multiple observables into one, without using a selector.

7: System.Reactive.Subjects

7.1: Overview

System.Reactive.Subjects has the following types:



7.2: AsyncSubject

AsyncSubject emits its first (and only) event when it completes, to all observers that are still subscribed at the time of completion.

```
public sealed class AsyncSubject<T> : ISubject<T>, ISubject<T, T>,
IObserver<T>, IObservable<T>, IDisposable
{
    public AsyncSubject();

    public void Dispose();
    public void OnCompleted();
    public void OnError(Exception error);
    public void OnNext(T value);
    public IDisposable Subscribe(IObserver<T> observer);
}
```

Remarks

Before OnCompleted is called, no elements are passed to subscribed observers.

AsyncSubject remembers the most recent element passed to OnNext.

Once OnCompleted is called, all currently subscribed observers are passed the more recent element, and then their OnCompleted method is called.

7.3: BehaviorSubject

BehaviorSubject is similar to ReplaySubject with the difference being it retains just a single (the last) element.

```
public sealed class BehaviorSubject<T>
    : ISubject<T>, ISubject<T, T>, IObserver<T>,
    IObservable<T>, IDisposable
{
    public BehaviorSubject(T value);

    public void Dispose();
    public void OnCompleted();
    public void OnError(Exception error);
    public void OnNext(T value);
    public IDisposable Subscribe(IObserver<T> observer);
}
```

7.4: IConnectableObservable

Represents an observable to which may be connected to and disconnected from.

```
Public interface IConnectableObservable <out T> : IObservable<T> {
    IDisposable Connect();
}
```

Remarks

It is best to think of a connectable observable as an implementation of an

observer (on an underlying observable) and an observable (to which clients of the connectable observable can subscribe).

When one needs both an observer and an observable, typically one uses a subject. Hence implementations of `ICConnectableObservable` will often manage both a subject and a separate underlying observable, representing the source of the elements.

It is important to realize there are two relationships involved – one between client observers and the connectable observable (which are called subscriptions), and another between the connectable observable and the underlying observable (which are called connections).

Client observers can subscribe to the subject using `IObservable.Subscribe`. The return value is an `IDisposable`, and client observers can unsubscribe by disposing of this.

The connectable observable is “connected” to the underlying observable by calling `Connect()` and this returns an `IDisposable`, which represents the connection. This can be disposed of too.

Connections and subscriptions can be established and disposed of independently of each other. When subscriptions and a connection exists, then elements from the underlying observable are passed to the connectable observable which in turn passes them to the client observers.

7.4.1: Connect

Establish the connection.

```
IDisposable Connect();
```

Return Value

- `IDisposable` – Call `Dispose()` on this to disconnect

7.5: `ISubject<T1, T2>`

A subject is both an observer and an observable. This interface accepts distinctly specified type parameters for `IObserver` and `IObservable`.

```
Public interface ISubject<in T1, out T2>  
    : IObserver<T1>, IObservable<T2>{ }
```

Remarks

An implementation of `ISubject` could observe another observable, in some way process the elements received from it, and they pass out the processed elements to observers of the subject. However, in many cases there will not be any additional observable. Instead, element generating code will directly call the

subject's OnNext/OnError/OnCompleted methods, these calls are processed in some way and the results passed on to observers of the subject. All the subject implementations in System.Reactive.Subjects are of the latter form. You will note their constructors do not allow you to pass in an observable to observe.

Note another interface, ISubject<T>, exists which derives from ISubject<T1, T2> and it passes T as the type parameter for both T1 and T2.

7.6: ISubject<T>

A subject is both an observer and an observable. This interface accepts just one type parameter, which is used for both IObservable and IObservable.

```
public interface ISubject<T>
    : ISubject<T,T>, IObservable<T>, IObservable<T>{ }
```

Remarks

The System.Reactive.Subjects namespace includes a number of Subject implementations and they all derive from Subject<T> (e.g. the type parameter for the IObservable is the same as for the IObservable).

7.7: ReplaySubject

ReplaySubject retains a history of previous elements, so for each observer, the entire list is available.

```
public sealed class ReplaySubject<T> : ISubject<T>, ISubject<T, T>,
    IObservable<T>, IObservable<T>, IDisposable {

    public ReplaySubject();
    public ReplaySubject(int bufferSize);
    public ReplaySubject(IScheduler scheduler);
    public ReplaySubject(TimeSpan window);
    public ReplaySubject(int bufferSize, IScheduler scheduler);
    public ReplaySubject(int bufferSize, TimeSpan window);
    public ReplaySubject(TimeSpan window, IScheduler scheduler);
    public ReplaySubject(
        int bufferSize, TimeSpan window, IScheduler scheduler);

    public void Dispose();
    public void OnCompleted();
    public void OnError(Exception error);
    public void OnNext(T value);
    public IDisposable Subscribe(IObservable<T> observer);
}
```

Remarks

ReplaySubject remembers elements pass to its OnNext. The amount it retains can be a default, or based on a buffer size or based on a time window.

7.7.1: ReplaySubject Constructor

Constructors for ReplaySubject.

```

public ReplaySubject();
public ReplaySubject(int bufferSize);
public ReplaySubject(IScheduler scheduler);
public ReplaySubject(TimeSpan window);
public ReplaySubject(int bufferSize, IScheduler scheduler);
public ReplaySubject(int bufferSize, TimeSpan window);
public ReplaySubject(TimeSpan window, IScheduler scheduler);
public ReplaySubject(
    int bufferSize, TimeSpan window, IScheduler scheduler);

```

Parameters

- bufferSize (int) - The maximum number of elements to remember
- scheduler (IScheduler) - The scheduler to use for all operations
- window (TimeSpan) - The time window within which to remember elements

7.8: Subject

A collection of static helper methods which return ISubject instances.

```

public static class Subject {
    public static ISubject<TSource, TResult>
        Create<TSource, TResult>(
            IObservable<TSource> observable,
            IObservable<TResult> observable);
    public static ISubject<TSource, TResult>
        Synchronize<TSource, TResult>(
            ISubject<TSource, TResult> subject);
    public static ISubject<TSource, TResult>
        Synchronize<TSource, TResult>(
            ISubject<TSource, TResult> subject,
            IScheduler scheduler);
}

```

Remarks

Note there are two quite distinct types named Subject. This static class, and a concrete class called Subject<T> that implements ISubject<T>.

7.8.1: Create

Create a subject from the observable and observer parameters.

```

public static ISubject<TSource, TResult>
    Create<TSource, TResult>(
        IObservable<TSource> observable,
        IObservable<TResult> observable);

```

Parameters

- observer (IObservable<TSource>) - The observer to use

- observable (IObservable<TResult>) - The observable to use

Return Value

- ISubject<> - The created subject

Remarks

Use this method where there is a need to create a subject from distinct observer and observable. Often there is a need for interaction between the implementation of both interfaces and so a custom type is needed, but where they are separate, then this class will create the subject easily.

7.8.2: Synchronize(ISubject<TSource, TResult>)

Synchronize the calls made by the observable in the subject to subscribers.

```
public static ISubject<TSource, TResult>
    Synchronize<TSource, TResult>(
        ISubject<TSource, TResult> subject);
```

Parameters

- subject (ISubject<TSource, TResult>) - The observer to use

Return Value

- ISubject<> - The synchronized subject

Remarks

The output of observables is meant to be synchronized in order to comply with the Rx Contract.

Call this method to ensure subjects which are not compliant become compliant.

7.8.3: Synchronize(ISubject<TSource, TResult>, IScheduler)

Synchronize the calls made by the observable in the subject to subscribers using the supplied scheduler.

```
public static ISubject<TSource, TResult>
    Synchronize<TSource, TResult>(
        ISubject<TSource, TResult> subject,
        IScheduler scheduler);
```

Parameters

- subject (ISubject<TSource, TResult>) - The observer to use

Return Value

- ISubject<> - The synchronized subject

7.9: Subject<T>

A subject is both an observable and an observer.

```
public sealed class Subject<T> : ISubject<T>, ISubject<T, T>,
    IObservable<T>, IOObserver<T>, IDisposable {

    public Subject ();

    public void Dispose ();
    public void OnCompleted ();
    public void OnError (Exception error);
    public void OnNext (T value);
    public IDisposable Subscribe (IObserver<T> observer);
}
```

Remarks

Alternative Subject implementations in System.Reactive.Subjects perform other types of processing on the event stream.

7.9.1: Subject()

Constructor for subject.

```
public Subject ();
```

Parameters

- scheduler (IScheduler) - scheduler to use.

7.9.2: Dispose()

The subject stops acting as an observable and so does not pass on received messages to subscribed observers.

```
public void Dispose ();
```

7.9.3: OnCompleted

Calls OnCompleted for each subscribed observer.

```
public void OnCompleted ();
```

7.9.4: OnError

Calls OnError for each subscribed observer.

```
public void OnError (Exception exception);
```

Parameters

- exception (Exception) - The exception to pass to the OnError call(s)

7.9.5: OnNext

Calls OnNext for each subscribed observer.

```
public void OnNext (T value);
```

Parameters

- value (T) - The value to pass to the OnNext call(s)

7.9.6: Subscribe

Subscribes an observer to the subject.

```
public IDisposable Subscribe(IObserver<T> observer);
```

Parameters

- observer (IObserver<T>) - The observer doing the subscribing

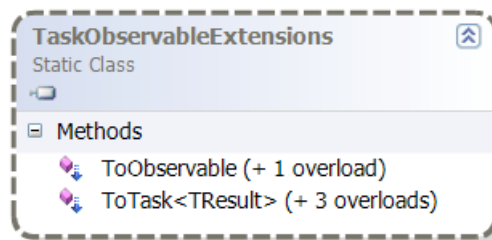
Return Value

- IDisposable - Call dispose() on this to end the subscription

8: System.Threading.Tasks

8.1: Overview

Rx adds one static type to System.Threading.Tasks:



8.2: TaskObservableExtensions

Extension methods to convert from observable to task and from task to observable with optional support for cancellation tokens.

```
public static class TaskObservableExtensions {  
  
    public static IObservable<TResult> ToObservable<TResult>(this Task<TResult> task);  
    public static IObservable<Unit> ToObservable(this Task task);  
  
    public static Task<TResult> ToTask<TResult>(this IObservable<TResult> observable);  
    public static Task<TResult> ToTask<TResult>(this IObservable<TResult> observable, Cancellation token);  
    public static Task<TResult> ToTask<TResult>(this IObservable<TResult> observable, object state);  
    public static Task<TResult> ToTask<TResult>(this IObservable<TResult> observable, Cancellation token, object state);  
  
}
```

8.2.1: ToObservable(Task<TResult>)

Converts a task with a result to an observable on that result.

```
public static IObservable<TResult> ToObservable<TResult>(this Task<TResult> task);
```

Parameters

- task (Task<TResult>) - The task to be converted to an observable

Return Value

- IObservable<TResult> - The observable

Remarks

The task must be run by the calling code (this method does not do that). The result of subscribing to the returned observable is that either `OnNext` will be called once and `OnCompleted` will be called once, or `OnError` will be called once if an error occurred. The return value of the task becomes the parameter for the `OnNext` call.

If the task completes before `ToObservable` is called, then the behavior does not change - the task remembers its return value and it is made available to observers of the observable later (thus avoiding race conditions).

`Task<TResult>` is one of the supported signatures for C#'s `async/await` functionality. The result of a call to an `async` method can be used to call `ToObservable`, in effect enabling observers to subscribe to `async` methods.

8.2.2: ToObservable(Task)

Converts a task with no result to an observable.

```
public static IObservable<Unit> ToObservable(this Task task);
```

Parameters

- `task (Task)` - The task which is to be converted to an observable

Return Value

- `IObservable<Unit>` - The observable (`Unit` is a type introduced by Reactive Extensions to mimic the behavior of `void`)

Remarks

The task must be run by the calling code (this method does not do that). The result of subscribing to the returned observable is that either `OnNext` will be called once and `OnCompleted` will be called once, or `OnError` will be called once if an error occurred. An instance of `Unit` is the parameter for the `OnNext` call.

`Task` is one of the supported signatures for C#'s `async/await` functionality. The result of a call to an `async` method can be used to call `ToObservable`, in effect enabling observers to subscribe to `async` methods.

8.2.3: ToTask(IObservable<TResult>)

Converts an observable to a task whose result value is the last element from the observable.

```
public static Task<TResult> ToTask<TResult>(
    this IObservable<TResult> observable);
```

Parameters

- `IObservable<TResult>` - The observable

Return Value

- `Task<TResult>` - The task whose result value is the last element from the observable

Remarks

There is no need to call `Task.Start`, as the computation is already running. If the task has not completed executing, then a call to `Task.Result` blocks until it has completed.

8.2.4: `ToTask(IObservable<TResult>, object)`

Converts an observable to a task whose result value is the last element from the observable and whose `AsyncState` value is passed as a parameter.

```
public static Task<TResult> ToTask<TResult>(
    this IObservable<TResult> observable, object state);
```

Parameters

- `IObservable<TResult>` - The observable
- `state (object)` - The `AsyncState` of the the task

Return Value

- `Task<TResult>` - The task whose result value is the last element from the observable

Remarks

There is no need to call `Task.Start`, as the computation is already running. The state parameter may be used to pass state information into the executing task.

State information passed as a parameter can be accessed via the task instance's `AsyncState` property.

8.2.5: `ToTask(IObservable<TResult>, CancellationToken)`

Converts an observable to a task whose result is the last element from the observable, and allows use of a cancellation token to cancel.

```
public static Task<TResult> ToTask<TResult>(
    this IObservable<TResult> observable,
    CancellationToken cancellationToken);
```

Parameters

- `IObservable<TResult>` - The observable
- `cancellationToken (CancellationToken)` - used to cancel

Return Value

- `Task<TResult>` - The task whose return value is the last element from the observable

Remarks

The cancellation token parameter may be created by first instantiating `System.Threading.CancellationTokenSource` and then passing its `Token` property (of type `CancellationToken`) to this `ToTask`.

The token may be canceled by calling `CancellationTokenSource`'s `Cancel` or `CancelAfter` methods. If this occurs and the task has not completed executing, an exception of type `AggregateException` is thrown, with one inner exception, of type `System.Threading.Tasks.TaskCanceledException`. If the cancellation token is canceled after the task has finished executing, it is ignored.

8.2.6: `ToTask(IObservable<TResult>, CancellationToken, object)`

Converts an observable to a task whose result is the last element from the observable, and allows use of a cancellation token to cancel and whose `AsyncState` is passed as a parameter.

```
public static Task<TResult> ToTask<TResult>(
    this IObservable<TResult> observable,
    CancellationToken cancellationToken, object state);
```

Parameters

- `IObservable<TResult>` - The observable
- `cancellationToken` (`CancellationToken`) - used to cancel
- `state` (`object`) - The `AsyncState` of the the task

Return Value

- `Task<TResult>` - The task whose return value is the last element from the observable

Remarks

The cancellation token parameter may be created by first instantiating `System.Threading.CancellationTokenSource` and then passing its `Token` property (of type `CancellationToken`) to this `ToTask`.

The token may be canceled by calling `CancellationTokenSource`'s `Cancel` or `CancelAfter` methods. If this occurs and the task has not completed executing, an exception of type `AggregateException` is thrown, with one inner exception, of type `System.Threading.Tasks.TaskCanceledException`. If the cancellation token is canceled after the task has finished executing, it is ignored.

State information passed as a parameter can be accessed via the task instance's `AsyncState` property.