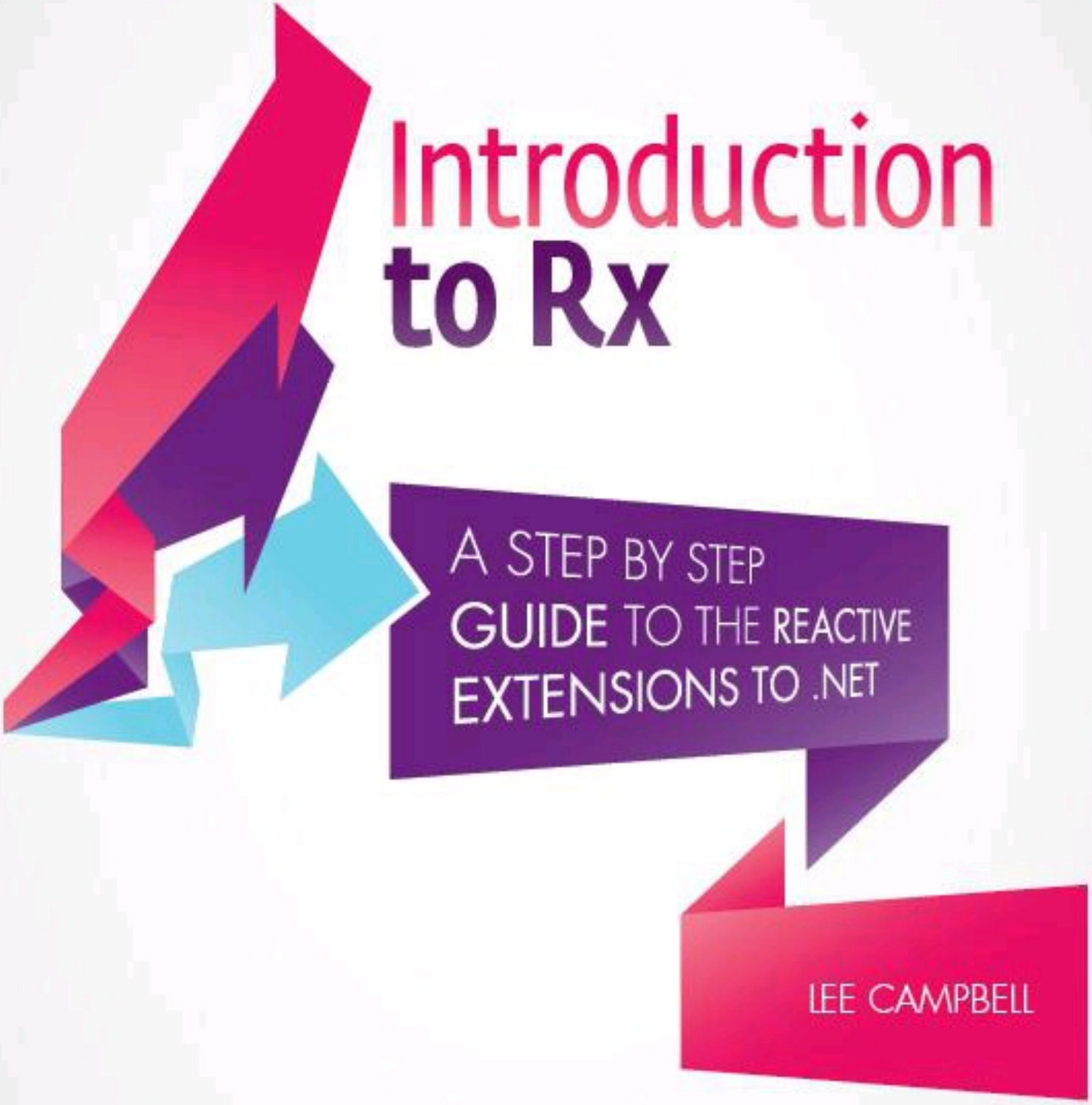

Introduction
to Rx
A STEP BY STEP
GUIDE TO THE REACTIVE
EXTENSIONS TO .NET
LEE CAMPBELL

TABLE OF CONTENTS

Introduction to Rx

Lee Campbell

Preface

Reactive programming is not a new concept. I remember studying my first Event Driven module for Visual Basic 5 in 2000. Even then the technology (Visual Basic 5) was already considered somewhat dated. Long before VB5 and the turn of the millennium, we have seen languages supporting events. Over time languages like Smalltalk, Delphi and the .NET languages have popularized reactive or event-driven programming paradigms. This not to say that events are passé: current trends such as CEP (Complex Event Processing), CQRS (Command Query Responsibility Segregation) and rich immersive GUIs, all have events as a fundamental part of their makeup.

The event driven paradigm allows for code to be invoked without the need for breaking encapsulation or applying expensive polling techniques. This is commonly implemented with the Observer pattern, events exposed directly in the language (e.g. C#) or other forms of callback via delegate registration. The Reactive Extensions extend the callback metaphor with LINQ to enable querying sequences of events and managing concurrency.

The Reactive Extensions are effectively a library of implementations of the *IObservable<T>* and *IObserver<T>* interfaces for .NET, Silverlight and Windows Phone 7. The libraries are also available in JavaScript. As a dynamic language, JavaScript had no need for the two interfaces so the JavaScript implementation could have been written long before .NET 4 was released. This book will introduce Rx via C#. Users of VB.NET, F# and other .NET languages hopefully will be able to extract the concepts and translate them to their particular language. JavaScript users should be able to gather the concepts from this book and apply them to their language. JavaScript users may however find some features are not supported, and some concepts, such as scheduling do not transcend platforms.

As Rx is just a library, the team at Microsoft delivering Rx was able to isolate themselves from the release schedule of the .NET Framework. This proved important as the libraries saw fairly constant evolution since late 2009 through to their official release in mid 2011. This evolution has been largely enabled by the openness of the team and their ability to take onboard criticisms, suggestions and feature requests from the brave community of pre-release users.

While Rx is *just a library*, it is a significant and bold move forward for the team at Microsoft and for any consumers of the library. Rx *will* change the way you design and build software for the following reasons:

- The way that it tackles the Observer pattern is a divorce from .NET events toward a Java-style interface pattern but far more refined.
- The way it tackles concurrency is quite a shift how many .NET developers would have done it before.
- The abundance of (extension) methods in the library.
- The way in which it integrates with LINQ to leverage LINQ's composability & declarative style, makes Rx very usable and discoverable to those already familiar with LINQ and *IEnumerable<T>*.

- The way it can help any .NET developer that works with event driven and/or asynchronous programs. Developers of Rich Clients, Web Clients and Services alike can all benefit from Rx.
- The future plans seem even grander, but that is a different book for some time in the future :-)

This book aims to teach you:

- about the new types that Rx will provide
- about the extension methods and how to use them
- how to manage subscriptions to "sequences" of data
- how to visualize "sequences" of data and sketch your solution before coding it
- how to deal with concurrency to your advantage and avoid common pitfalls
- how to compose, aggregate and transform streams
- how to test your Rx code
- some guidance on best practices when using Rx.

The best way to learn Rx is to use it. Reading the theory from this book will only help you be familiar with Rx, but will not really enable you to fully understand Rx. You can download the latest version of Rx from the Microsoft Data Developer site (http://msdn.microsoft.com/en-us/data/gg577609) or if you use NuGet you can just download Rx via that.

My experience with Rx is straight from the trenches. I worked on a team of exceptional developers on a project that was an early adopter of Rx (late 2009). The project was a financial services application that started off life as a Silverlight project then expanded into an integration project. We used Rx everywhere; client side in Silverlight 3/4, and server side in .NET 3.5/4.0. We used Rx eagerly and sometimes too eagerly. We were past leading edge, we were *bleeding* edge. We were finding bugs in the early releases and posting proposed fixes to the guys at Microsoft. We were constantly updating to the latest version. It cost the project to be early adopters, but in time the payment was worth it. Rx allowed us to massively simplify an application that was inherently asynchronous, highly concurrent and targeted low latencies. Similar workflows that I had written in previous projects were pages of code long; now with Rx were several lines of LINQ. Trying to test asynchronous code on clients (WPF/Win Forms/Silverlight) was a constant challenge, but Rx solved that too. Today if you ask a question on the Rx Forums, you will most likely be answered by someone from that team (or Dave Sexton).

Acknowledgements

I would like to take this quick pause to recognize the people that made this book possible. First is my poor wife for losing a husband to a dark room for several months. Her understanding and tolerance is much appreciated. To my old team "Alpha Alumni"; every developer on that team has helped me in some way to better myself as a developer. Specific mention goes to James Miles, Matt Barrett, John Marks, Duncan Mole, Cathal Golden, Keith Woods, Ray Booysen & Olivier DeHeurles for all the deep dive sessions, emails, forum banter, BBM exchanges, lunch breaks and pub sessions spent trying to get our heads around Rx. To Matt Davey for being brave enough to support us in using Rx back in 2009. To the team at Microsoft that did the hard work and brought us Rx; Jeffery Van Gogh, Wes Dyer, Erik Meijer & Bart De Smet. Extra special mention to Bart, there is just something about the content that Bart produces that clicks with me. Finally to the guys that helped edit the book; Joe Albahari and Gregory Andrien. Joe is a veteran author of books such as the C# in a nutshell, C# pocket reference and LINQ pocket reference, and managed to find time to help out on this project while also releasing the latest versions of these books. For Gregory and I, this was a first for both of us, as editor and author respectively. Gregory committed many late nights to helping complete this project. There is also some sweet irony in having a French person as the editor. Even though English is not his native tongue, he clearly has a better grasp of it than I.

It is my intention that from the experiences both good and bad, I can help speed up your understanding of Rx and lower that barrier to entry to using Rx. This will be a progressive step-by-step approach. It may seem slow in places, but the fundamentals are so important to have a firm grasp on the powerful features. I hope you will have the patience to join me all the way to the end.

The content of this book was originally posted as a series of blog posts at http://LeeCampbell.blogspot.com and has proved popular enough that I thought it warranted being reproduced as an e-book. In the spirit of other books such as Joe Albahari's Threading in C# and Scott Chacon's Pro Git books, and considering the blog was free, I have made the first version of this book free.

The version that this book has been written against is the .Net 4.0 targeted Rx assemblies version 1.0.10621.0 (NuGet: Rx-Main v1.0.11226).

The Table of Contents for the *Introduction to Rx* shows you all of the topics covered in this book. Kindle users can get to the table of contents by pressing the Menu button from any page. Move the 5-way down until you underline "Table of Contents" and press the 5-way to go to it.

So, fire up Visual Studio and let's get started.

PART 1 - Getting started

Why Rx?

Users expect real time data. They want their tweets now. Their order confirmed now. They need prices accurate as of now. Their online games need to be responsive. As a developer, you demand fire-and-forget messaging. You don't want to be blocked waiting for a result. You want to have the result pushed to you when it is ready. Even better, when working with result sets, you want to receive individual results as they are ready. You do not want to wait for the entire set to be processed before you see the first row. The world has moved to push; users are waiting for us to catch up. Developers have tools to push data, this is easy. Developers need tools to react to push data.

Welcome to Reactive Extensions for .NET (Rx). This book is aimed at any .NET developer curious about the *IObservable<T>* and *IObserver<T>* interfaces that have popped up in .NET 4. The Reactive Extensions libraries from Microsoft are the implementations of these interfaces that are quickly picking up traction with Server, Client and Web developers alike. Rx is a powerfully productive development tool. Rx enables developers to solve problems in an elegant, familiar and declarative style; often crucially with less code than was possible without Rx. By leveraging LINQ, Rx gets to boast the standard benefits of a LINQ implementation[1].

Integrated
: LINQ is integrated into the C# language.

Unitive
: Using LINQ allows you to leverage your existing skills for querying data at rest (LINQ to SQL, LINQ to XML or LINQ to objects) to query data in motion. You could think of Rx as LINQ to events. LINQ allows you to transition from other paradigms into a common paradigm. For example you can transition a standard .NET event, an asynchronous method call, a *Task* or perhaps a 3rd party middleware API into a single common Rx paradigm. By leveraging our existing language of choice and using familiar operators like *Select*, *Where*, *GroupBy* etc, developers can rationalize and communicate designs or code in a common form.

Extensible
: You can extend Rx with your own custom query operators (extension methods).

Declarative
: LINQ allows your code to read as a declaration of *what* your code does and leaves the *how* to the implementation of the operators.

Composable
: LINQ features, such as extension methods, lambda syntax and query comprehension syntax, provide a fluent API for developers to consume. Queries can be constructed with numerous operators. Queries can then be composed together to further produce composite queries.

Transformative
: Queries can transform their data from one type to another. A query might translate a single value to another value, aggregated from a sequence of values to a single average value or expand a single data value into a sequence of values.

When is Rx appropriate?

Rx offers a natural paradigm for dealing with sequences of events. A sequence can contain zero or more events. Rx proves to be most valuable when composing sequences of events.

Should use Rx

Managing events like these is what Rx was built for:

- UI events like mouse move, button click
- Domain events like property changed, collection updated, "Order Filled", "Registration accepted" etc.
- Infrastructure events like from file watcher, system and WMI events
- Integration events like a broadcast from a message bus or a push event from WebSockets API or other low latency middleware like Nirvana
- Integration with a CEP engine like StreamInsight or StreamBase.

Interestingly Microsoft's CEP product StreamInsight, which is part of the SQL Server family, also uses LINQ to build queries over streaming events of data.

Rx is also very well suited for introducing and managing concurrency for the purpose of *offloading*. That is, performing a given set of work concurrently to free up the current thread. A very popular use of this is maintaining a responsive UI.

You should consider using Rx if you have an existing *IEnumerable<T>* that is attempting to model data in motion. While *IEnumerable<T>can* model data in motion (by using lazy evaluation like `yield return`), it probably won't scale. Iterating over an *IEnumerable<T>* will consume/block a thread. You should either favor the non-blocking nature of Rx via either IObservable<T> or consider the `async` features in .NET 4.5.

Could use Rx

Rx can also be used for asynchronous calls. These are effectively sequences of one event.

- Result of a Task or Task<T>
- Result of an APM method call like *FileStream* BeginRead/EndRead

You may find the using TPL, Dataflow or `async` keyword (.NET 4.5) proves to be a more natural way of composing asynchronous methods. While Rx can definitely help with these scenarios, if there are other more appropriate frameworks at your disposal you should consider them first.

Rx can be used, but is less suited for, introducing and managing concurrency for the purposes of *scaling* or performing *parallel* computations. Other dedicated frameworks like TPL (Task Parallel Library) or C++ AMP are more appropriate for performing parallel compute intensive work.

See more on TPL, Dataflow, `async` and C++ AMP at [Microsoft's Concurrency homepage](#).

Won't use Rx

Rx and specifically *IObservable<T>* is not a replacement for *IEnumerable<T>*. I would not recommend trying to take something that is naturally pull based and force it to be push based.

- Translating existing *IEnumerable<T>* values to *IObservable<T>* just so that the code base can be "more Rx"
- Message queues. Queues like in MSMQ or a JMS implementation generally have transactionality and are by definition sequential. I feel *IEnumerable<T>* is a natural fit for here.

By choosing the best tool for the job your code should be easier to maintain, provide better performance and you will probably get better support.

Rx in action

Adopting and learning Rx can be an iterative approach where you can slowly apply it to your infrastructure and domain. In a short time you should be able to have the skills to produce code, or reduce existing code, to queries composed of simple operators. For example this simple ViewModel is all I needed to code to integrate a search that is to be executed as a user types.

```
public class MemberSearchViewModel : INotifyPropertyChanged
{
        //Fields removed...
        public MemberSearchViewModel(IMemberSearchModel memberSearchModel,
        ISchedulerProvider schedulerProvider)
        {
         memberSearchModel = memberSearchModel;
        //Run search when SearchText property changes
        this.PropertyChanges(vm => vm.SearchText)
        .Subscribe(Search);
        }
        //Assume INotifyPropertyChanged implementations of properties...
        public string SearchText { get; set; }
        public bool IsSearching { get; set; }
        public string Error { get; set; }
        public ObservableCollection<string> Results { get; }
        //Search on background thread and return result on dispatcher.
        private void Search(string searchText)
        {
        using (_currentSearch) { }
        IsSearching = true;
        Results.Clear();
        Error = null;
        _currentSearch = _memberSearchModel.SearchMembers(searchText)
        .Timeout(TimeSpan.FromSeconds(2))
        .SubscribeOn(_schedulerProvider.TaskPool)
        .ObserveOn(_schedulerProvider.Dispatcher)
        .Subscribe(
        Results.Add,
        ex =>
        {
                IsSearching = false;
                Error = ex.Message;
        },
        () => { IsSearching = false; });
        }
        ...
}
```

While this code snippet is fairly small it supports the following requirements:

- Maintains a responsive UI
- Supports timeouts
- Knows when the search is complete
- Allows results to come back one at a time
- Handles errors
- Is unit testable, even with the concurrency concerns
- If a user changes the search, cancel current search and execute new search with new text.

To produce this sample is almost a case of composing the operators that match the requirements into a single query. The query is small, maintainable, declarative and far less code than "rolling your own". There is the added benefit of reusing a well tested API. The less code *you* have to write, the less code *you* have to test, debug and maintain. Creating other queries like the following is simple:

- calculating a moving average of a series of values e.g. **service level agreements** for average latencies or downtime
- combining event data from multiple sources e.g.: **search results** from Bing, Google and Yahoo, or **sensor data** from Accelerometer, Gyro, Magnetometer or temperatures
- grouping data e.g. **tweets** by topic or user, or **stock prices** by delta or liquidity
- filtering data e.g. **online game servers** within a region, for a specific game or with a minimum number of participants.

Push is here. Arming yourself with Rx is a powerful way to meet users' expectations of a push world. By understanding and composing the constituent parts of Rx you will be able to make short work of complexities of processing incoming events. Rx is set to become a day-to-day part of your coding experience.

---

[1] Essential LINQ - Calvert, Kulkarni

Key types

There are two key types to understand when working with Rx, and a subset of auxiliary types that will help you to learn Rx more effectively. The *IObserver<T>* and *IObservable<T>* form the fundamental building blocks for Rx, while implementations of *ISubject<TSource, TResult>* reduce the learning curve for developers new to Rx.

Many are familiar with LINQ and its many popular forms like LINQ to Objects, LINQ to SQL & LINQ to XML. Each of these common implementations allows you query *data at rest*; Rx offers the ability to query *data in motion*. Essentially Rx is built upon the foundations of the Observer pattern. .NET already exposes some other ways to implement the Observer pattern such as multicast delegates or events (which are usually multicast delegates). Multicast delegates are not ideal however as they exhibit the following less desirable features;

- In C#, events have a curious interface. Some find the `+=` and `-=` operators an unnatural way to register a callback
- Events are difficult to compose
- Events don't offer the ability to be easily queried over time
- Events are a common cause of accidental memory leaks
- Events do not have a standard pattern for signaling completion
- Events provide almost no help for concurrency or multithreaded applications. e.g. To raise an event on a separate thread requires you to do all of the plumbing

Rx looks to solve these problems. Here I will introduce you to the building blocks and some basic types that make up Rx.

IObservable<T>

IObservable<T> is one of the two new core interfaces for working with Rx. It is a simple interface with just a Subscribe method. Microsoft is so confident that this interface will be of use to you it has been included in the BCL as of version 4.0 of .NET. You should be able to think of anything that implements *IObservable<T>* as a streaming sequence of *T* objects. So if a method returned an *IObservable<Price>* I could think of it as a stream of Prices.

```
//Defines a provider for push-based notification.
public interface IObservable<out T>
{
        //Notifies the provider that an observer is to receive notifications.
        IDisposable Subscribe(IObserver<T> observer);
}
```

.NET already has the concept of Streams with the type and sub types of *System.IO.Stream*. The *System.IO.Stream* implementations are commonly used to stream data (generally bytes) to or from an I/O device like a file, network or block of memory. *System.IO.Stream* implementations can have both the ability to read and write, and sometimes the ability to seek (i.e. fast forward through a stream or move backwards). When I refer to an instance of *IObservable<T>* as a stream, it does not exhibit the seek or write functionality that streams do. This is a fundamental difference preventing Rx being built on top of the *System.IO.Stream* paradigm. Rx does however have the concept of forward streaming (push), disposing (closing) and completing (eof). Rx also extends the metaphor by introducing concurrency constructs, and query operations like transformation, merging, aggregating and expanding. These features are also not an appropriate fit for the existing *System.IO.Stream* types. Some others refer to instances of *IObservable<T>* as Observable Collections, which I find hard to understand. While the observable part makes sense to me, I do not find them like collections at all. You generally cannot sort, insert or remove items from an *IObservable<T>* instance like I would expect you can with a collection. Collections generally have some sort of backing store like an internal array. The values from an *IObservable<T>* source are not usually pre-materialized as you would expect from a normal collection. There is also a type in WPF/Silverlight called an *ObservableCollection<T>* that does exhibit collection-like behavior, and is very well suited to this description. In fact *IObservable<T>* integrates very well with *ObservableCollection<T>* instances. So to save on any confusion we will refer to instances of *IObservable<T>* as **sequences**. While instances of *IEnumerable<T>* are also sequences, we will adopt the convention that they are sequences of *data at rest*, and *IObservable<T>* instances are sequences of *data in motion*.

IObserver<T>

IObserver<T> is the other one of the two core interfaces for working with Rx. It too has made it into the BCL as of .NET 4.0. Don't worry if you are not on .NET 4.0 yet as the Rx team have included these two interfaces in a separate assembly for .NET 3.5 and Silverlight users. *IObservable<T>* is meant to be the "functional dual of *IEnumerable<T>*". If you want to know what that last statement means, then enjoy the hours of videos on Channel9 where they discuss the mathematical purity of the types. For everyone else it means that where an *IEnumerable<T>* can effectively yield three things (the next value, an exception or the end of the sequence), so too can *IObservable<T>* via *IObserver<T>*'s three methods *OnNext(T)*, *OnError(Exception)* and *OnCompleted()*.

```
//Provides a mechanism for receiving push-based notifications.
public interface IObserver<in T>
{
        //Provides the observer with new data.
        void OnNext(T value);
        //Notifies the observer that the provider has experienced an error condition.
        void OnError(Exception error);
        //Notifies the observer that the provider has finished sending push-based notifications.
        void OnCompleted();
}
```

Rx has an implicit contract that must be followed. An implementation of *IObserver<T>* may have zero or more calls to *OnNext(T)* followed optionally by a call to either *OnError(Exception)* or *OnCompleted()*. This protocol ensures that if a sequence terminates, it is always terminated by an *OnError(Exception)*, **or** an *OnCompleted()*. This protocol does not however demand that an *OnNext(T)*, *OnError(Exception)* or *OnCompleted()* ever be called. This enables to concept of empty and infinite sequences. We will look into this more later.

Interestingly, while you will be exposed to the *IObservable<T>* interface frequently if you work with Rx, in general you will not need to be concerned with *IObserver<T>*. This is due to Rx providing anonymous implementations via methods like *Subscribe*.

Implementing IObserver<T> and IObservable<T>

It is quite easy to implement each interface. If we wanted to create an observer that printed values to the console it would be as easy as this.

```
public class MyConsoleObserver<T> : IObserver<T>
{
        public void OnNext(T value)
        {
        Console.WriteLine("Received value {0}", value);
        }
        public void OnError(Exception error)
        {
        Console.WriteLine("Sequence faulted with {0}", error);
        }
        public void OnCompleted()
        {
        Console.WriteLine("Sequence terminated");
        }
}
```

Implementing an observable sequence is a little bit harder. An overly simplified implementation that returned a sequence of numbers could look like this.

```
public class MySequenceOfNumbers : IObservable<int>
{
        public IDisposable Subscribe(IObserver<int> observer)
        {
        observer.OnNext(1);
        observer.OnNext(2);
        observer.OnNext(3);
        observer.OnCompleted();
        return Disposable.Empty;
        }
}
```

We can tie these two implementations together to get the following output

```
var numbers = new MySequenceOfNumbers();
var observer = new MyConsoleObserver<int>();
numbers.Subscribe(observer);
```

Output:

```
Received value 1
Received value 2
Received value 3
Sequence terminated
```

The problem we have here is that this is not really reactive at all. This implementation is blocking, so we may as well use an *IEnumerable<T>* implementation like a *List<T>* or an array.

This problem of implementing the interfaces should not concern us too much. You will find that when you use Rx, you do not have the need to actually implement these interfaces, Rx provides all of the implementations you need out of the box. Let's have a look at the simple ones.

Subject<T>

I like to think of the *IObserver<T>* and the *IObservable<T>* as the 'reader' and 'writer' or, 'consumer' and 'publisher' interfaces. If you were to create your own implementation of *IObservable<T>* you may find that while you want to publicly expose the IObservable characteristics you still need to be able to publish items to the subscribers, throw errors and notify when the sequence is complete. Why that sounds just like the methods defined in *IObserver<T>*! While it may seem odd to have one type implementing both interfaces, it does make life easy. This is what subjects can do for you. *Subject<T>* is the most basic of the subjects. Effectively you can expose your *Subject<T>* behind a method that returns *IObservable<T>* but internally you can use the *OnNext*, *OnError* and *OnCompleted* methods to control the sequence.

In this very basic example, I create a subject, subscribe to that subject and then publish values to the sequence (by calling `subject.OnNext(T)`).

```
static void Main(string[] args)
{
        var subject = new Subject<string>();
        WriteSequenceToConsole(subject);
        subject.OnNext("a");
        subject.OnNext("b");
        subject.OnNext("c");
        Console.ReadKey();
}
//Takes an IObservable<string> as its parameter.
//Subject<string> implements this interface.
static void WriteSequenceToConsole(IObservable<string> sequence)
{
        //The next two lines are equivalent.
        //sequence.Subscribe(value=>Console.WriteLine(value));
        sequence.Subscribe(Console.WriteLine);
}
```

Note that the `WriteSequenceToConsole` method takes an *IObservable<string>* as it only wants access to the subscribe method. Hang on, doesn't the *Subscribe* method need an *IObserver<string>* as an argument? Surely *Console.WriteLine* does not match that interface. Well it doesn't, but the Rx team supply me with an Extension Method to *IObservable<T>* that just takes an *Action<T>*. The action will be executed every time an item is published. There are other overloads to the Subscribe extension method that allows you to pass combinations of delegates to be invoked for *OnNext*, *OnCompleted* and *OnError*. This effectively means I don't need to implement *IObserver<T>*. Cool.

As you can see, *Subject<T>* could be quite useful for getting started in Rx programming. *Subject<T>* however, is a basic implementation. There are three siblings to *Subject<T>* that offer subtly different implementations which can drastically change the way your program runs.

ReplaySubject<T>

ReplaySubject<T> provides the feature of caching values and then replaying them for any late subscriptions. Consider this example where we have moved our first publication to occur before our subscription

```
static void Main(string[] args)
{
        var subject = new Subject<string>();
        subject.OnNext("a");
        WriteSequenceToConsole(subject);
        subject.OnNext("b");
        subject.OnNext("c");
        Console.ReadKey();
}
```

The result of this would be that 'b' and 'c' would be written to the console, but 'a' ignored. If we were to make the minor change to make subject a *ReplaySubject<T>* we would see all publications again.

```
var subject = new ReplaySubject<string>();
subject.OnNext("a");
WriteSequenceToConsole(subject);
subject.OnNext("b");
subject.OnNext("c");
```

This can be very handy for eliminating race conditions. Be warned though, the default constructor of the *ReplaySubject<T>* will create an instance that caches every value published to it. In many scenarios this could create unnecessary memory pressure on the application. *ReplaySubject<T>* allows you to specify simple cache expiry settings that can alleviate this memory issue. One option is that you can specify the size of the buffer in the cache. In this example we create the *ReplaySubject<T>* with a buffer size of 2, and so only get the last two values published prior to our subscription:

```
public void ReplaySubjectBufferExample()
{
        var bufferSize = 2;
        var subject = new ReplaySubject<string>(bufferSize);
        subject.OnNext("a");
        subject.OnNext("b");
        subject.OnNext("c");
        subject.Subscribe(Console.WriteLine);
        subject.OnNext("d");
}
```

Here the output would show that the value 'a' had been dropped from the cache, but values 'b' and 'c' were still valid. The value 'd' was published after we subscribed so it is also written to the console.

```
Output:
b
c
d
```

Another option for preventing the endless caching of values by the *ReplaySubject<T>*, is to provide a window for the cache. In this example, instead of creating a *ReplaySubject<T>* with a buffer size, we specify a window of time that the cached values are valid for.

```
public void ReplaySubjectWindowExample()
{
        var window = TimeSpan.FromMilliseconds(150);
        var subject = new ReplaySubject<string>(window);
        subject.OnNext("w");
        Thread.Sleep(TimeSpan.FromMilliseconds(100));
        subject.OnNext("x");
        Thread.Sleep(TimeSpan.FromMilliseconds(100));
        subject.OnNext("y");
        subject.Subscribe(Console.WriteLine);
        subject.OnNext("z");
}
```

In the above example the window was specified as 150 milliseconds. Values are published 100 milliseconds apart. Once we have subscribed to the subject, the first value is 200ms old and as such has expired and been removed from the cache.

```
Output:
x
y
z
```

BehaviorSubject<T>

BehaviorSubject<T> is similar to *ReplaySubject<T>* except it only remembers the last publication. *BehaviorSubject<T>* also requires you to provide it a default value of *T*. This means that all subscribers will receive a value immediately (unless it is already completed).

In this example the value 'a' is written to the console:

```
public void BehaviorSubjectExample()
{
        //Need to provide a default value.
        var subject = new BehaviorSubject<string>("a");
        subject.Subscribe(Console.WriteLine);
}
```

In this example the value 'b' is written to the console, but not 'a'.

```
public void BehaviorSubjectExample2()
{
        var subject = new BehaviorSubject<string>("a");
        subject.OnNext("b");
        subject.Subscribe(Console.WriteLine);
}
```

In this example the values 'b', 'c' & 'd' are all written to the console, but again not 'a'

```
public void BehaviorSubjectExample3()
{
        var subject = new BehaviorSubject<string>("a");
        subject.OnNext("b");
        subject.Subscribe(Console.WriteLine);
        subject.OnNext("c");
        subject.OnNext("d");
}
```

Finally in this example, no values will be published as the sequence has completed. Nothing is written to the console.

```
public void BehaviorSubjectCompletedExample()
{
        var subject = new BehaviorSubject<string>("a");
        subject.OnNext("b");
        subject.OnNext("c");
        subject.OnCompleted();
        subject.Subscribe(Console.WriteLine);
}
```

That note that there is a difference between a *ReplaySubject<T>* with a buffer size of one (commonly called a 'replay one subject') and a *BehaviorSubject<T>*. A *BehaviorSubject<T>* requires an initial value. With the assumption that neither subjects have completed, then you can be sure that the *BehaviorSubject<T>* will have a value. You cannot be certain with the *ReplaySubject<T>* however. With this in mind, it is unusual to ever complete a *BehaviorSubject<T>*. Another difference is that a replay-one-subject will still cache its value once it has been completed. So subscribing to a completed *BehaviorSubject<T>* we can be sure to not receive any values, but with a *ReplaySubject<T>* it is possible.

*BehaviorSubject<T>*s are often associated with class properties. As they always have a value and can provide change notifications, they could be candidates for backing fields to properties.

AsyncSubject<T>

AsyncSubject<T> is similar to the Replay and Behavior subjects in the way that it caches values, however it will only store the last value, and only publish it when the sequence is completed. The general usage of the *AsyncSubject<T>* is to only ever publish one value then immediately complete. This means that is becomes quite comparable to *Task<T>*.

In this example no values will be published as the sequence never completes. No values will be written to the console.

```
static void Main(string[] args)
{
        var subject = new AsyncSubject<string>();
        subject.OnNext("a");
        WriteSequenceToConsole(subject);
        subject.OnNext("b");
        subject.OnNext("c");
        Console.ReadKey();
}
```

In this example we invoke the *OnCompleted* method so the last value 'c' is written to the console:

```
static void Main(string[] args)
{
        var subject = new AsyncSubject<string>();
        subject.OnNext("a");
        WriteSequenceToConsole(subject);
        subject.OnNext("b");
        subject.OnNext("c");
        subject.OnCompleted();
        Console.ReadKey();
}
```

Implicit contracts

There are implicit contacts that need to be upheld when working with Rx as mentioned above. The key one is that once a sequence is completed, no more activity can happen on that sequence. A sequence can be completed in one of two ways, either by *OnCompleted()* or by *OnError(Exception)*.

The four subjects described in this chapter all cater for this implicit contract by ignoring any attempts to publish values, errors or completions once the sequence has already terminated.

Here we see an attempt to publish the value 'c' on a completed sequence. Only values 'a' and 'b' are written to the console.

```
public void SubjectInvalidUsageExample()
{
        var subject = new Subject<string>();
        subject.Subscribe(Console.WriteLine);
        subject.OnNext("a");
        subject.OnNext("b");
        subject.OnCompleted();
        subject.OnNext("c");
}
```

ISubject interfaces

While each of the four subjects described in this chapter implement the *IObservable<T>* and *IObserver<T>* interfaces, they do so via another set of interfaces:

```
//Represents an object that is both an observable sequence as well as an observer.
public interface ISubject<in TSource, out TResult>
        : IObserver<TSource>, IObservable<TResult>
{
}
```

As all the subjects mentioned here have the same type for both *TSource* and *TResult*, they implement this interface which is the superset of all the previous interfaces:

```
//Represents an object that is both an observable sequence as well as an observer.
public interface ISubject<T> : ISubject<T, T>, IObserver<T>, IObservable<T>
{
}
```

These interfaces are not widely used, but prove useful as the subjects do not share a common base class. We will see the subject interfaces used later when we discover hot and cold observables.

Subject factory

Finally it is worth making you aware that you can also create a subject via a factory method. Considering that a subject combines the *IObservable<T>* and *IObserver<T>* interfaces, it seems sensible that there should be a factory that allows you to combine them yourself. The *Subject.Create(IObserver<TSource>, IObservable<TResult>)* factory method provides just this.

```
//Creates a subject from the specified observer used to publish messages to the subject
//  and observable used to subscribe to messages sent from the subject
public static ISubject<TSource, TResult> Create<TSource, TResult>(
        IObserver<TSource> observer,
        IObservable<TResult> observable)
{...}
```

Subjects provide a convenient way to poke around Rx, however they are not recommended for day to day use. An explanation is in the Usage Guidelines in the appendix. Instead of using subjects, favor the factory methods we will look at in Part 2.

The fundamental types *IObserver<T>* and *IObservable<T>* and the auxiliary subject types create a base from which to build your Rx knowledge. It is important to understand these simple types and their implicit contracts. In production code you may find that you rarely use the *IObserver<T>* interface and subject types, but understanding them and how they fit into the Rx eco-system is still important. The *IObservable<T>* interface is the dominant type that you will be exposed to for representing a sequence of data in motion, and therefore will comprise the core concern for most of your work with Rx and most of this book.

Lifetime management

The very nature of Rx code is that you as a consumer do not know when a sequence will provide values or terminate. This uncertainty does not prevent your code from providing a level of certainty. You can control when you will start accepting values and when you choose to stop accepting values. You still need to be the master of your domain. Understanding the basics of managing Rx resources allow your applications to be as efficient, bug free and predictable as possible.

Rx provides fine grained control to the lifetime of subscriptions to queries. While using familiar interfaces, you can deterministically release resources associated to queries. This allows you to make the decisions on how to most effectively manage your resources, ideally keeping the scope as tight as possible.

In the previous chapter we introduced you to the key types and got off the ground with some examples. For the sake of keeping the initial samples simple we ignored a very important part of the *IObservable<T>* interface. The *Subscribe* method takes an *IObserver<T>* parameter, but we did not need to provide that as we used the extension method that took an *Action<T>* instead. The important part we overlooked is that both *Subscribe* methods have a return value. The return type is *IDisposable*. In this chapter we will further explore how this return value can be used to management lifetime of our subscriptions.

Subscribing

Just before we move on, it is worth briefly looking at all of the overloads of the *Subscribe* extension method. The overload we used in the previous chapter was the simple Overload to Subscribe which allowed us to pass just an *Action<T>* to be performed when *OnNext* was invoked. Each of these further overloads allows you to avoid having to create and then pass in an instance of *IObserver<T>*.

```
//Just subscribes to the Observable for its side effects.
// All OnNext and OnCompleted notifications are ignored.
// OnError notifications are re-thrown as Exceptions.
IDisposable Subscribe<TSource>(this IObservable<TSource> source);
//The onNext Action provided is invoked for each value.
//OnError notifications are re-thrown as Exceptions.
IDisposable Subscribe<TSource>(this IObservable<TSource> source,
        Action<TSource> onNext);
//The onNext Action is invoked for each value.
//The onError Action is invoked for errors
IDisposable Subscribe<TSource>(this IObservable<TSource> source,
        Action<TSource> onNext,
        Action<Exception> onError);
//The onNext Action is invoked for each value.
//The onCompleted Action is invoked when the source completes.
//OnError notifications are re-thrown as Exceptions.
IDisposable Subscribe<TSource>(this IObservable<TSource> source,
        Action<TSource> onNext,
        Action onCompleted);
//The complete implementation
IDisposable Subscribe<TSource>(this IObservable<TSource> source,
        Action<TSource> onNext,
        Action<Exception> onError,
        Action onCompleted);
```

Each of these overloads allows you to pass various combinations of delegates that you want executed for each of the notifications an *IObservable<T>* instance could produce. A key point to note is that if you use an overload that does not specify a delegate for the *OnError* notification, any *OnError* notifications will be re-thrown as an exception. Considering that the error could be raised at any time, this can make debugging quite difficult. It is normally best to use an overload that specifies a delegate to cater for *OnError* notifications.

In this example we attempt to catch error using standard .NET Structured Exception Handling:

```
var values = new Subject<int>();
try
{
        values.Subscribe(value => Console.WriteLine("1st subscription received {0}", value));
}
catch (Exception ex)
{
        Console.WriteLine("Won't catch anything here!");
}
values.OnNext(0);
//Exception will be thrown here causing the app to fail.
values.OnError(new Exception("Dummy exception"));
```

The correct way to way to handle exceptions is to provide a delegate for *OnError* notifications as in this example.

```
var values = new Subject<int>();
values.Subscribe(
        value => Console.WriteLine("1st subscription received {0}", value),
        ex => Console.WriteLine("Caught an exception : {0}", ex));
values.OnNext(0);
values.OnError(new Exception("Dummy exception"));
```

We will look at other interesting ways to deal with errors on a sequence in later chapters in the book.

Unsubscribing

We have yet to look at how we could unsubscribe from a subscription. If you were to look for an *Unsubscribe* method in the Rx public API you would not find any. Instead of supplying an Unsubscribe method, Rx will return an *IDisposable* whenever a subscription is made. This disposable can be thought of as the subscription itself, or perhaps a token representing the subscription. Disposing it will dispose the subscription and effectively *unsubscribe*. Note that calling *Dispose* on the result of a Subscribe call will not cause any side effects for other subscribers; it just removes the subscription from the observable's internal list of subscriptions. This then allows us to call *Subscribe* many times on a single *IObservable<T>*, allowing subscriptions to come and go without affecting each other. In this example we initially have two subscriptions, we then dispose of one subscription early which still allows the other to continue to receive publications from the underlying sequence:

```
var values = new Subject<int>();
var firstSubscription = values.Subscribe(value =>
        Console.WriteLine("1st subscription received {0}", value));
var secondSubscription = values.Subscribe(value =>
        Console.WriteLine("2nd subscription received {0}", value));
values.OnNext(0);
values.OnNext(1);
values.OnNext(2);
values.OnNext(3);
firstSubscription.Dispose();
Console.WriteLine("Disposed of 1st subscription");
values.OnNext(4);
values.OnNext(5);
```

Output:

```
1st subscription received 0
2nd subscription received 0
1st subscription received 1
2nd subscription received 1
1st subscription received 2
2nd subscription received 2
1st subscription received 3
2nd subscription received 3
Disposed of 1st subscription
2nd subscription received 4
2nd subscription received 5
```

The team building Rx could have created a new interface like *ISubscription* or *IUnsubscribe* to facilitate unsubscribing. They could have added an *Unsubscribe* method to the existing *IObservable<T>* interface. By using the *IDisposable* type instead we get the following benefits for free:

- The type already exists
- People understand the type
- *IDisposable* has standard usages and patterns
- Language support via the *using* keyword
- Static analysis tools like FxCop can help you with its usage
- The *IObservable<T>* interface remains very simple.

As per the *IDisposable* guidelines, you can call *Dispose* as many times as you like. The first call will unsubscribe and any further calls will do nothing as the subscription will have already been disposed.

OnError and OnCompleted

Both the *OnError* and *OnCompleted* signify the completion of a sequence. If your sequence publishes an *OnError* or *OnCompleted* it will be the last publication and no further calls to *OnNext* can be performed. In this example we try to publish an *OnNext* call after an *OnCompleted* and the *OnNext* is ignored:

```
var subject = new Subject<int>();
subject.Subscribe(
        Console.WriteLine,
        () => Console.WriteLine("Completed"));
subject.OnCompleted();
subject.OnNext(2);
```

Of course, you could implement your own *IObservable<T>* that allows publishing after an *OnCompleted* or an *OnError*, however it would not follow the precedence of the current Subject types and would be a non-standard implementation. I think it would be safe to say that the inconsistent behavior would cause unpredictable behavior in the applications that consumed your code.

An interesting thing to consider is that when a sequence completes or errors, you should still dispose of your subscription.

IDisposable

The *IDisposable* interface is a handy type to have around and it is also integral to Rx. I like to think of types that implement *IDisposable* as having explicit lifetime management. I should be able to say "I am done with that" by calling the *Dispose()* method.

By applying this kind of thinking, and then leveraging the C# `using` statement, you can create handy ways to create scope. As a reminder, the `using` statement is effectively a `try`/`finally` block that will always call *Dispose* on your instance when leaving the scope.

If we consider that we can use the *IDisposable* interface to effectively create a scope, you can create some fun little classes to leverage this. For example here is a simple class to log timing events:

```
public class TimeIt : IDisposable
{
        private readonly string _name;
        private readonly Stopwatch _watch;
        public TimeIt(string name)
        {
        _name = name;
        _watch = Stopwatch.StartNew();
        }
        public void Dispose()
        {
        _watch.Stop();
        Console.WriteLine("{0} took {1}", _name, _watch.Elapsed);
        }
}
```

This handy little class allows you to create scope and measure the time certain sections of your code base take to run. You could use it like this:

```
using (new TimeIt("Outer scope"))
{
        using (new TimeIt("Inner scope A"))
        {
        DoSomeWork("A");
        }
        using (new TimeIt("Inner scope B"))
        {
        DoSomeWork("B");
        }
        Cleanup();
}
```

Output:

```
Inner scope A took 00:00:01.0000000
Inner scope B took 00:00:01.5000000
Outer scope took 00:00:02.8000000
```

You could also use the concept to set the color of text in a console application:

```
//Creates a scope for a console foreground color. When disposed, will return to
//  the previous Console.ForegroundColor
public class ConsoleColor : IDisposable
{
        private readonly System.ConsoleColor _previousColor;
        public ConsoleColor(System.ConsoleColor color)
        {
        _previousColor = Console.ForegroundColor;
        Console.ForegroundColor = color;
        }
        public void Dispose()
        {
        Console.ForegroundColor = _previousColor;
        }
}
```

I find this handy for easily switching between colors in little *spike* console applications:

```
Console.WriteLine("Normal color");
using (new ConsoleColor(System.ConsoleColor.Red))
{
        Console.WriteLine("Now I am Red");
        using (new ConsoleColor(System.ConsoleColor.Green))
        {
        Console.WriteLine("Now I am Green");
        }
        Console.WriteLine("and back to Red");
}
```

Output:

```
Normal color
Now I am Red
Now I am Green
and back to Red
```

So we can see that you can use the *IDisposable* interface for more than just common use of deterministically releasing unmanaged resources. It is a useful tool for managing lifetime or scope of anything; from a stopwatch timer, to the current color of the console text, to the subscription to a sequence of notifications.

The Rx library itself adopts this liberal usage of the *IDisposable* interface and introduces several of its own custom implementations:

- Disposable
- BooleanDisposable
- CancellationDisposable
- CompositeDisposable
- ContextDisposable
- MultipleAssignmentDisposable
- RefCountDisposable
- ScheduledDisposable
- SerialDisposable
- SingleAssignmentDisposable

For a full rundown of each of the implementations see the Disposables reference in the Appendix. For now we will look at the extremely simple and useful *Disposable* static class:

```
namespace System.Reactive.Disposables
{
        public static class Disposable
        {
        // Gets the disposable that does nothing when disposed.
        public static IDisposable Empty { get {...} }
        // Creates the disposable that invokes the specified action when disposed.
        public static IDisposable Create(Action dispose)
        {...}
        }
}
```

As you can see it exposes two members: *Empty* and *Create*. The *Empty* method allows you get a stub instance of an *IDisposable* that does nothing when `Dispose()` is called. This is useful for when you need to fulfil an interface requirement that returns an *IDisposable* but you have no specific implementation that is relevant.

The other overload is the *Create* factory method which allows you to pass an *Action* to be invoked when the instance is disposed. The *Create* method will ensure the standard Dispose semantics, so calling `Dispose()` multiple times will only invoke the delegate you provide once:

```
var disposable = Disposable.Create(() => Console.WriteLine("Being disposed."));
Console.WriteLine("Calling dispose...");
disposable.Dispose();
Console.WriteLine("Calling again...");
disposable.Dispose();
```

Output:

```
Calling dispose...
Being disposed.
```

```
Calling again...
```

Note that "Being disposed." is only printed once. In a later chapter we cover another useful method for binding the lifetime of a resource to that of a subscription in the Observable.Using method.

Resource management vs. memory management

It seems many .NET developers only have a vague understanding of the .NET runtime's Garbage Collector and specifically how it interacts with Finalizers and *IDisposable*. As the author of the Framework Design Guidelines points out, this may be due to the confusion between 'resource management' and 'memory management':

> Many people who hear about the Dispose pattern for the first time complain that the GC isn't doing its job. They think it should collect resources, and that this is just like having to manage resources as you did in the unmanaged world. The truth is that the GC was never meant to manage resources. It was designed to manage memory and it is excellent in doing just that. - Krzysztof Cwalina from Joe Duffy's blog

This is both a testament to Microsoft for making .NET so easy to work with and also a problem as it is a key part of the runtime to misunderstand. Considering this, I thought it was prudent to note that *subscriptions will not be automatically disposed of*. You can safely assume that the instance of *IDisposable* that is returned to you does not have a finalizer and will not be collected when it goes out of scope. If you call a *Subscribe* method and ignore the return value, you have lost your only handle to unsubscribe. The subscription will still exist, and you have effectively lost access to this resource, which could result in leaking memory and running unwanted processes.

The exception to this cautionary note is when using the *Subscribe* extension methods. These methods will internally construct behavior that will *automatically detach* subscriptions when the sequence completes or errors. Even with the automatic detach behavior; you still need to consider sequences that never terminate (by *OnCompleted* or *OnError*). You will need the instance of *IDisposable* to terminate the subscription to these infinite sequences explicitly.

> You will find many of the examples in this book will not allocate the *IDisposable* return value. This is only for brevity and clarity of the sample. Usage guidelines and best practice information can be found in the appendix.

By leveraging the common *IDisposable* interface, Rx offers the ability to have deterministic control over the lifetime of your subscriptions. Subscriptions are independent, so the disposable of one will not affect another. While some *Subscribe* extension methods utilize an automatically detaching observer, it is still considered best practice to explicitly manage your subscriptions, as you would with any other resource implementing *IDisposable*. As we will see in later chapters, a subscription may actually incur the cost of other resources such as event handles, caches and threads. It is also best practice to always provide an *OnError* handler to prevent an exception being thrown in an otherwise difficult to handle manner.

With the knowledge of subscription lifetime management, you are able to keep a tight leash on subscriptions and their underlying resources. With judicious application of standard disposal patterns to your Rx code, you can keep your applications predictable, easier to maintain, easier to extend and hopefully bug free.

PART 2 - Sequence basics

So you want to get involved and write some Rx code, but how do you get started? We have looked at the key types, but know that we should not be creating our own implementations of *IObserver<T>* or *IObservable<T>* and should favor factory methods over using subjects. Even if we have an observable sequence, how do we pick out the data we want from it? We need to understand the basics of creating an observable sequence, getting values into it and picking out the values we want from them.

In Part 2 we discover the basics for constructing and querying observable sequences. We assert that LINQ is fundamental to using and understanding Rx. On deeper inspection, we find that *functional programming* concepts are core to having a deep understanding of LINQ and therefore enabling you to master Rx. To support this understanding, we classify the query operators into three main groups. Each of these groups proves to have a root operator that the other operators can be constructed from. Not only will this deconstruction exercise provide a deeper insight to Rx, functional programming and query composition; it should arm you with the ability to create custom operators where the general Rx operators do not meet your needs.

Creating a sequence

In the previous chapters we used our first Rx extension method, the *Subscribe* method and its overloads. We also have seen our first factory method in `Subject.Create()`. We will start looking at the vast array of other methods that enrich *IObservable<T>* to make Rx what it is. It may be surprising to see that there are relatively few public instance methods in the Rx library. There are however a large number of public static methods, and more specifically, a large number of extension methods. Due to the large number of methods and their overloads, we will break them down into categories.

> Some readers may feel that they can skip over parts of the next few chapters. I would only suggest doing so if you are very confident with LINQ and functional composition. The intention of this book is to provide a step-by-step introduction to Rx, with the goal of you, the reader, being able to apply Rx to your software. The appropriate application of Rx will come through a sound understanding of the fundamentals of Rx. The most common mistakes people will make with Rx are due to a misunderstanding of the principles upon which Rx was built. With this in mind, I encourage you to read on.

It seems sensible to follow on from our examination of our key types where we simply constructed new instances of subjects. Our first category of methods will be *creational* methods: simple ways we can create instances of *IObservable<T>* sequences. These methods generally take a seed to produce a sequence: either a single value of a type, or just the type itself. In functional programming this can be described as *anamorphism* or referred to as an '*unfold*'.

Simple factory methods

Observable.Return

In our first and most basic example we introduce *Observable.Return<T>(T value)*. This method takes a value of T and returns an IObservable<T> with the single value and then completes. It has *unfolded* a value of *T* into an observable sequence.

```
var singleValue = Observable.Return<string>("Value");
//which could have also been simulated with a replay subject
var subject = new ReplaySubject<string>();
subject.OnNext("Value");
subject.OnCompleted();
```

Note that in the example above that we could use the factory method or get the same effect by using the replay subject. The obvious difference is that the factory method is only one line and it allows for declarative over imperative programming style. In the example above we specified the type parameter as *string*, this is not necessary as it can be inferred from the argument provided.

```
singleValue = Observable.Return<string>("Value");
//Can be reduced to the following
singleValue = Observable.Return("Value");
```

Observable.Empty

The next two examples only need the type parameter to unfold into an observable sequence. The first i s *Observable.Empty<T>()*. This returns an empty *IObservable<T>* i.e. it just publishes an *OnCompleted* notification.

```
var empty = Observable.Empty<string>();
//Behaviorally equivalent to
var subject = new ReplaySubject<string>();
subject.OnCompleted();
```

Observable.Never

The *Observable.Never<T>()* method will return infinite sequence without any notifications.

```
var never = Observable.Never<string>();
//similar to a subject without notifications
var subject = new Subject<string>();
```

Observable.Throw

Observable.Throw<T>(Exception) method needs the type parameter information, it also need the *Exception* that it will *OnError* with. This method creates a sequence with just a single *OnError* notification containing the exception passed to the factory.

```
var throws = Observable.Throw<string>(new Exception());
//Behaviorally equivalent to
var subject = new ReplaySubject<string>();
subject.OnError(new Exception());
```

Observable.Create

The *Create* factory method is a little different to the above creation methods. The method signature itself may be a bit overwhelming at first, but becomes quite natural once you have used it.

```
//Creates an observable sequence from a specified Subscribe method implementation.
```

```
public static IObservable<TSource> Create<TSource>(
        Func<IObserver<TSource>, IDisposable> subscribe)
{...}
public static IObservable<TSource> Create<TSource>(
        Func<IObserver<TSource>, Action> subscribe)
{...}
```

Essentially this method allows you to specify a delegate that will be executed anytime a subscription is made. The *IObserver<T>* that made the subscription will be passed to your delegate so that you can call the *OnNext*/*OnError*/*OnCompleted* methods as you need. This is one of the few scenarios where you will need to concern yourself with the *IObserver<T>* interface. Your delegate is a *Func* that returns an *IDisposable*. This *IDisposable* will have its *Dispose()* method called when the subscriber disposes from their subscription.

The *Create* factory method is the preferred way to implement custom observable sequences. The usage of subjects should largely remain in the realms of samples and testing. Subjects are a great way to get started with Rx. They reduce the learning curve for new developers, however they pose several concerns that the *Create* method eliminates. Rx is effectively a functional programming paradigm. Using subjects means we are now managing state, which is potentially mutating. Mutating state and asynchronous programming are very hard to get right. Furthermore many of the operators (extension methods) have been carefully written to ensure correct and consistent lifetime of subscriptions and sequences are maintained. When you introduce subjects you can break this. Future releases may also see significant performance degradation if you explicitly use subjects.

The *Create* method is also preferred over creating custom types that implement the *IObservable* interface. There really is no need to implement the observer/observable interfaces yourself. Rx tackles the intricacies that you may not think of such as thread safety of notifications and subscriptions.

A significant benefit that the *Create* method has over subjects is that the sequence will be lazily evaluated. Lazy evaluation is a very important part of Rx. It opens doors to other powerful features such as scheduling and combination of sequences that we will see later. The delegate will only be invoked when a subscription is made.

In this example we show how we might first return a sequence via standard blocking eagerly evaluated call, and then we show the correct way to return an observable sequence without blocking by lazy evaluation.

```
private IObservable<string> BlockingMethod()
{
        var subject = new ReplaySubject<string>();
        subject.OnNext("a");
        subject.OnNext("b");
        subject.OnCompleted();
        Thread.Sleep(1000);
        return subject;
}
private IObservable<string> NonBlocking()
{
        return Observable.Create<string>(
        (IObserver<string> observer) =>
        {
        observer.OnNext("a");
        observer.OnNext("b");
        observer.OnCompleted();
        Thread.Sleep(1000);
        return Disposable.Create(() => Console.WriteLine("Observer has unsubscribed"));
        //or can return an Action like
        //return () => Console.WriteLine("Observer has unsubscribed");
        });
}
```

While the examples are somewhat contrived, the intention is to show that when a consumer calls the eagerly evaluated, blocking method, they will be blocked for at least 1 second before they even receive the *IObservable<string>*, regardless of if they do actually subscribe to it or not. The non

blocking method is lazily evaluated so the consumer immediately receives the *IObservable<string>* and will only incur the cost of the thread sleep if they subscribe.

As an exercise, try to build the *Empty*, *Return*, *Never* & *Throw* extension methods yourself using the *Create* method. If you have Visual Studio or LINQPad available to you right now, code it up as quickly as you can. If you don't (perhaps you are on the train on the way to work), try to conceptualize how you would solve this problem. When you are done move forward to see some examples of how it could be done...

Examples of *Empty*, *Return*, *Never* and *Throw* recreated with *Observable.Create*:

```
public static IObservable<T> Empty<T>()
{
        return Observable.Create<T>(o =>
        {
        o.OnCompleted();
        return Disposable.Empty;
        });
}
public static IObservable<T> Return<T>(T value)
{
        return Observable.Create<T>(o =>
        {
        o.OnNext(value);
        o.OnCompleted();
        return Disposable.Empty;
        });
}
public static IObservable<T> Never<T>()
{
        return Observable.Create<T>(o =>
        {
        return Disposable.Empty;
        });
}
public static IObservable<T> Throws<T>(Exception exception)
{
        return Observable.Create<T>(o =>
        {
        o.OnError(exception);
        return Disposable.Empty;
        });
}
```

You can see that *Observable.Create* provides the power to build our own factory methods if we wish. You may have noticed that in each of the examples we only are able to return our subscription token (the implementation of *IDisposable*) once we have produced all of our *OnNext* notifications. This is because inside of the delegate we provide, we are completely sequential. It also makes the token rather pointless. Now we look at how we can use the return value in a more useful way. First is an example where inside our delegate we create a Timer that will call the observer's *OnNext* each time the timer ticks.

```
//Example code only
public void NonBlocking_event_driven()
{
        var ob = Observable.Create<string>(
        observer =>
        {
        var timer = new System.Timers.Timer();
        timer.Interval = 1000;
        timer.Elapsed += (s, e) => observer.OnNext("tick");
        timer.Elapsed += OnTimerElapsed;
        timer.Start();
        return Disposable.Empty;
        });
        var subscription = ob.Subscribe(Console.WriteLine);
        Console.ReadLine();
        subscription.Dispose();
}
private void OnTimerElapsed(object sender, ElapsedEventArgs e)
{
        Console.WriteLine(e.SignalTime);
}
```

Output:

```
tick
01/01/2012 12:00:00
tick
01/01/2012 12:00:01
tick
01/01/2012 12:00:02
01/01/2012 12:00:03
01/01/2012 12:00:04
01/01/2012 12:00:05
```

The example above is broken. When we dispose of our subscription, we will stop seeing "tick" being written to the screen; however we have not released our second event handler "`OnTimerElasped`" and have not disposed of the instance of the timer, so it will still be writing the `ElapsedEventArgs.SignalTime` to the console after our disposal. The extremely simple fix is to return `timer` as the *IDisposable* token.

```
//Example code only
var ob = Observable.Create<string>(
        observer =>
        {
        var timer = new System.Timers.Timer();
        timer.Interval = 1000;
        timer.Elapsed += (s, e) => observer.OnNext("tick");
        timer.Elapsed += OnTimerElapsed;
        timer.Start();
        return timer;
        });
```

Now when a consumer disposes of their subscription, the underlying *Timer* will be disposed of too.

Observable.Create also has an overload that requires your *Func* to return an *Action* instead of an *IDisposable*. In a similar example to above, this one shows how you could use an action to un-register the event handler, preventing a memory leak by retaining the reference to the timer.

```
//Example code only
var ob = Observable.Create<string>(
        observer =>
        {
        var timer = new System.Timers.Timer();
        timer.Enabled = true;
        timer.Interval = 100;
        timer.Elapsed += OnTimerElapsed;
        timer.Start();
        return ()=>{
        timer.Elapsed -= OnTimerElapsed;
        timer.Dispose();
        };
        });
```

These last few examples showed you how to use the *Observable.Create* method. These were just examples; there are actually better ways to produce values from a timer that we will look at soon. The intention is to show that *Observable.Create* provides you a lazily evaluated way to create observable sequences. We will dig much deeper into lazy evaluation and application of the *Create* factory method throughout the book especially when we cover concurrency and scheduling.

Functional unfolds

As a functional programmer you would come to expect the ability to unfold a potentially infinite sequence. An issue we may face with *Observable.Create* is that is that it can be a clumsy way to produce an infinite sequence. Our timer example above is an example of an infinite sequence, and while this is a simple implementation it is an annoying amount of code for something that effectively is delegating all the work to the *System.Timers.Timer* class. The *Observable.Create* method also has poor support for unfolding sequences using corecursion.

Corecursion

Corecursion is a function to apply to the current state to produce the next state. Using corecursion by taking a value, applying a function to it that extends that value and repeating we can create a sequence. A simple example might be to take the value 1 as the seed and a function that increments the given value by one. This could be used to create sequence of [1,2,3,4,5...].

Using corecursion to create an *IEnumerable<int>* sequence is made simple with the `yield return` syntax.

```
private static IEnumerable<T> Unfold<T>(T seed, Func<T, T> accumulator)
{
        var nextValue = seed;
        while (true)
        {
        yield return nextValue;
        nextValue = accumulator(nextValue);
        }
}
```

The code above could be used to produce the sequence of natural numbers like this.

```
var naturalNumbers = Unfold(1, i => i + 1);
Console.WriteLine("1st 10 Natural numbers");
foreach (var naturalNumber in naturalNumbers.Take(10))
{
        Console.WriteLine(naturalNumber);
}
```

Output:

```
1st 10 Natural numbers
1
2
3
4
5
6
7
8
9
10
```

Note the `Take(10)` is used to terminate the infinite sequence.

Infinite and arbitrary length sequences can be very useful. First we will look at some that come with Rx and then consider how we can generalize the creation of infinite observable sequences.

Observable.Range

Observable.Range(int, int) simply returns a range of integers. The first integer is the initial value and the second is the number of values to yield. This example will write the values '10' through to '24' and then complete.

```
var range = Observable.Range(10, 15);
range.Subscribe(Console.WriteLine, ()=>Console.WriteLine("Completed"));
```

Observable.Generate

It is difficult to emulate the *Range* factory method using *Observable.Create*. It would be cumbersome to try and respect the principles that the code should be lazily evaluated and the consumer should be able to dispose of the subscription resources when they so choose. This is where we can use corecursion to provide a richer unfold. In Rx the unfold method is called *Observable.Generate*.

The simple version of *Observable.Generate* takes the following parameters:

- an initial state
- a predicate that defines when the sequence should terminate
- a function to apply to the current state to produce the next state
- a function to transform the state to the desired output

```
public static IObservable<TResult> Generate<TState, TResult>(
        TState initialState,
        Func<TState, bool> condition,
        Func<TState, TState> iterate,
        Func<TState, TResult> resultSelector)
```

As an exercise, write your own *Range* factory method using *Observable.Generate*.

Consider the *Range* signature `Range(int start, int count)`, which provides the seed and a value for the conditional predicate. You know how each new value is derived from the previous one; this becomes your iterate function. Finally, you probably don't need to transform the state so this makes the result selector function very simple.

Continue when you have built your own version...

Example of how you could use *Observable.Generate* to construct a similar *Range* factory method.

```
//Example code only
public static IObservable<int> Range(int start, int count)
{
        var max = start + count;
        return Observable.Generate(
        start,
        value => value < max,
        value => value + 1,
        value => value);
}
```

Observable.Interval

Earlier in the chapter we used a *System.Timers.Timer* in our observable to generate a continuous sequence of notifications. As mentioned in the example at the time, this is not the preferred way of working with timers in Rx. As Rx provides operators that give us this functionality it could be argued that to not use them is to re-invent the wheel. More importantly the Rx operators are the preferred way of working with timers due to their ability to substitute in schedulers which is desirable for easy substitution of the underlying timer. There are at least three various timers you could choose from for the example above:

- *System.Timers.Timer*
- *System.Threading.Timer*
- *System.Windows.Threading.DispatcherTimer*

By abstracting the timer away via a scheduler we are able to reuse the same code for multiple platforms. More importantly than being able to write platform independent code is the ability to substitute in a test-double scheduler/timer to enable testing. Schedulers are a complex subject that is out of scope for this chapter, but they are covered in detail in the later chapter on Scheduling and threading.

There are three better ways of working with constant time events, each being a further generalization of the former. The first is **Observable.Interval(TimeSpan)** which will publish incremental values starting from zero, based on a frequency of your choosing. This example publishes values every 250 milliseconds.

```
var interval = Observable.Interval(TimeSpan.FromMilliseconds(250));
interval.Subscribe(
        Console.WriteLine,
        () => Console.WriteLine("completed"));
```

Output:

```
0
1
2
3
4
5
```

Once subscribed, you must dispose of your subscription to stop the sequence. It is an example of an infinite sequence.

Observable.Timer

The second factory method for producing constant time based sequences is **Observable.Timer**. It has

several overloads; the first of which we will look at being very simple. The most basic overload of *Observable.Timer* takes just a *TimeSpan* as *Observable.Interval* does. The *Observable.Timer* will however only publish one value (0) after the period of time has elapsed, and then it will complete.

```
var timer = Observable.Timer(TimeSpan.FromSeconds(1));
timer.Subscribe(
        Console.WriteLine,
        () => Console.WriteLine("completed"));
```

Output:

```
0
completed
```

Alternatively, you can provide a *DateTimeOffset* for the `dueTime` parameter. This will produce the value 0 and complete at the due time.

A further set of overloads adds a *TimeSpan* that indicates the period to produce subsequent values. This now allows us to produce infinite sequences and also construct *Observable.Interval* from *Observable.Timer*.

```
public static IObservable<long> Interval(TimeSpan period)
{
        return Observable.Timer(period, period);
}
```

Note that this now returns an *IObservable* of `long` not `int`. While *Observable.Interval* would always wait the given period before producing the first value, this *Observable.Timer* overload gives the ability to start the sequence when you choose. With *Observable.Timer* you can write the following to have an interval sequence that started immediately.

```
Observable.Timer(TimeSpan.Zero, period);
```

This takes us to our third way and most general way for producing timer related sequences, back to *Observable.Generate*. This time however, we are looking at a more complex overload that allows you to provide a function that specifies the due time for the next value.

```
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)
```

Using this overload, and specifically the extra `timeSelector` argument, we can produce our own implementation of *Observable.Timer* and in turn, *Observable.Interval*.

```
public static IObservable<long> Timer(TimeSpan dueTime)
{
        return Observable.Generate(
        0l,
        i => i < 1,
        i => i + 1,
        i => i,
        i => dueTime);
}
public static IObservable<long> Timer(TimeSpan dueTime, TimeSpan period)
{
        return Observable.Generate(
        0l,
        i => true,
        i => i + 1,
        i => i,
        i => i == 0 ? dueTime : period);
}
public static IObservable<long> Interval(TimeSpan period)
{
        return Observable.Generate(
        0l,
        i => true,
        i => i + 1,
        i => i,
        i => period);
}
```

This shows how you can use *Observable.Generate* to produce infinite sequences. I will leave it up to you the reader, as an exercise using *Observable.Generate*, to produce values at variable rates. I find using these methods invaluable not only in day to day work but especially for producing dummy data.

Transitioning into IObservable<T>

Generation of an observable sequence covers the complicated aspects of functional programming i.e. corecursion and unfold. You can also start a sequence by simply making a transition from an existing synchronous or asynchronous paradigm into the Rx paradigm.

From delegates

The *Observable.Start* method allows you to turn a long running *Func<T>* or *Action* into a single value observable sequence. By default, the processing will be done asynchronously on a ThreadPool thread. If the overload you use is a *Func<T>* then the return type will be *IObservable<T>*. When the function returns its value, that value will be published and then the sequence completed. If you use the overload that takes an *Action*, then the returned sequence will be of type *IObservable<Unit>*. The *Unit* type is a functional programming construct and is analogous to `void`. In this case *Unit* is used to publish an acknowledgement that the *Action* is complete, however this is rather inconsequential as the sequence is immediately completed straight after *Unit* anyway. The *Unit* type itself has no value; it just serves as an empty payload for the *OnNext* notification. Below is an example of using both overloads.

```
static void StartAction()
{
        var start = Observable.Start(() =>
        {
        Console.Write("Working away");
        for (int i = 0; i < 10; i++)
        {
        Thread.Sleep(100);
        Console.Write(".");
        }
        });
        start.Subscribe(
        unit => Console.WriteLine("Unit published"),
        () => Console.WriteLine("Action completed"));
}
static void StartFunc()
{
        var start = Observable.Start(() =>
        {
        Console.Write("Working away");
        for (int i = 0; i < 10; i++)
        {
        Thread.Sleep(100);
        Console.Write(".");
        }
        return "Published value";
        });
        start.Subscribe(
        Console.WriteLine,
        () => Console.WriteLine("Action completed"));
}
```

Note the difference between *Observable.Start* and *Observable.Return*; *Start* lazily evaluates the value from a function, *Return* provided the value eagerly. This makes *Start* very much like a *Task*. This can also lead to some confusion on when to use each of the features. Both are valid tools and the choice come down to the context of the problem space. Tasks are well suited to parallelizing computational work and providing workflows via continuations for computationally heavy work. Tasks also have the benefit of documenting and enforcing single value semantics. Using *Start* is a good way to integrate computationally heavy work into an existing code base that is largely made up of observable sequences. We look at composition of sequences in more depth later in the book.

From events

As we discussed early in the book, .NET already has the event model for providing a reactive, event driven programming model. While Rx is a more powerful and useful framework, it is late to the party and so needs to integrate with the existing event model. Rx provides methods to take an event and turn

it into an observable sequence. There are several different varieties you can use. Here is a selection of common event patterns.

```
//Activated delegate is EventHandler
var appActivated = Observable.FromEventPattern(
        h => Application.Current.Activated += h,
        h => Application.Current.Activated -= h);
//PropertyChanged is PropertyChangedEventHandler
var propChanged = Observable.FromEventPattern
        <PropertyChangedEventHandler, PropertyChangedEventArgs>(
        handler => handler.Invoke,
        h => this.PropertyChanged += h,
        h => this.PropertyChanged -= h);
//FirstChanceException is EventHandler<FirstChanceExceptionEventArgs>
var firstChanceException = Observable.FromEventPattern<FirstChanceExceptionEventArgs>(
        h => AppDomain.CurrentDomain.FirstChanceException += h,
        h => AppDomain.CurrentDomain.FirstChanceException -= h);
```

So while the overloads can be confusing, they key is to find out what the event's signature is. If the signature is just the base *EventHandler* delegate then you can use the first example. If the delegate is a sub-class of the *EventHandler*, then you need to use the second example and provide the *EventHandler* sub-class and also its specific type of *EventArgs*. Alternatively, if the delegate is the newer generic *EventHandler<TEventArgs>*, then you need to use the third example and just specify what the generic type of the event argument is.

It is very common to want to expose property changed events as observable sequences. These events can be exposed via *INotifyPropertyChanged* interface, a *DependencyProperty* or perhaps by events named appropriately to the Property they are representing. If you are looking at writing your own wrappers to do this sort of thing, I would strongly suggest looking at the Rxx library on http://Rxx.codeplex.com first. Many of these have been catered for in a very elegant fashion.

From Task

Rx provides a useful, and well named set of overloads for transforming from other existing paradigms to the Observable paradigm. The *ToObservable()* method overloads provide a simple route to make the transition.

As we mentioned earlier, the *AsyncSubject<T>* is similar to a *Task<T>*. They both return you a single value from an asynchronous source. They also both cache the result for any repeated or late requests for the value. The first `ToObservable()` extension method overload we look at is an extension to *Task<T>*. The implementation is simple;

- if the task is already in a status of `RanToCompletion` then the value is added to the sequence and then the sequence completed
- if the task is Cancelled then the sequence will error with a *TaskCanceledException*
- if the task is Faulted then the sequence will error with the task's inner exception
- if the task has not yet completed, then a continuation is added to the task to perform the above actions appropriately

There are two reasons to use the extension method:

1. From Framework 4.5, almost all I/O-bound functions return *Task<T>*
2. I f *Task<T>* is a good fit, it's preferable to use it over *IObservable<T>* - because it

communicates single-value result in the type system. In other words, a function that returns a single value in the future should return a *Task<T>*, not an *IObservable<T>*. Then if you need to combine it with other observables, use `ToObservable()`.

Usage of the extension method is also simple.

```
var t = Task.Factory.StartNew(()=>"Test");
var source = t.ToObservable();
source.Subscribe(
        Console.WriteLine,
        () => Console.WriteLine("completed"));
```

Output:

```
Test
completed
```

There is also an overload that converts a *Task* (non generic) to an *IObservable<Unit>*.

From IEnumerable<T>

The final overload of *ToObservable* takes an *IEnumerable<T>*. This is semantically like a helper method for an *Observable.Create* with a `foreach` loop in it.

```
//Example code only
public static IObservable<T> ToObservable<T>(this IEnumerable<T> source)
{
        return Observable.Create<T>(o =>
        {
        foreach (var item in source)
        {
        o.OnNext(item);
        }
        //Incorrect disposal pattern
        return Disposable.Empty;
        });
}
```

This crude implementation however is naive. It does not allow for correct disposal, it does not handle exceptions correctly and as we will see later in the book, it does not have a very nice concurrency model. The version in Rx of course caters for all of these tricky details so you don't need to worry.

When transitioning from *IEnumerable<T>* to *IObservable<T>*, you should carefully consider what you are really trying to achieve. You should also carefully test and measure the performance impacts of your decisions. Consider that the blocking synchronous (pull) nature of *IEnumerable<T>* sometimes just does not mix well with the asynchronous (push) nature of *IObservable<T>*. Remember that it is completely valid to pass *IEnumerable*, *IEnumerable<T>*, arrays or collections as the data type for an observable sequence. If the sequence can be materialized all at once, then you may want to avoid exposing it as an *IEnumerable*. If this seems like a fit for you then also consider passing immutable types like an array or a *ReadOnlyCollection<T>*. We will see the use of *IObservable<IList<T>>* later for operators that provide batching of data.

From APM

Finally we look at a set of overloads that take you from the Asynchronous Programming Model (APM) to an observable sequence. This is the style of programming found in .NET that can be identified with the use of two methods prefixed with *Begin...* and *End...* and the iconic *IAsyncResult* parameter type. This is commonly seen in the I/O APIs.

```
class WebRequest
{
        public WebResponse GetResponse()
        {...}
        public IAsyncResult BeginGetResponse(
        AsyncCallback callback,
        object state)
        {...}
        public WebResponse EndGetResponse(IAsyncResult asyncResult)
        {...}
        ...
}
class Stream
{
        public int Read(
        byte[] buffer,
        int offset,
        int count)
        {...}
        public IAsyncResult BeginRead(
        byte[] buffer,
        int offset,
        int count,
        AsyncCallback callback,
        object state)
        {...}
        public int EndRead(IAsyncResult asyncResult)
        {...}
        ...
}
```

At time of writing .NET 4.5 was still in preview release. Moving forward with .NET 4.5 the APM model will be replaced with *Task* and new `async` and `await` keywords. Rx 2.0 which is also in a beta release will integrate with these features. .NET 4.5 and Rx 2.0 are not in the scope of this book.

APM, or the Async Pattern, has enabled a very powerful, yet clumsy way of for .NET programs to perform long running I/O bound work. If we were to use the synchronous access to IO, e.g. `WebRequest.GetResponse()` or `Stream.Read(...)`, we would be blocking a thread but not performing any work while we waited for the IO. This can be quite wasteful on busy servers performing a lot of concurrent work to hold a thread idle while waiting for I/O to complete. Depending on the implementation, APM can work at the hardware device driver layer and not require any threads while blocking. Information on how to follow the APM model is scarce. Of the documentation you can find it is pretty shaky, however, for more information on APM, see Jeffery Richter's brilliant book *CLR via C#* or Joe Duffy's comprehensive *Concurrent Programming on Windows*. Most stuff on the internet is blatant plagiary of Richter's examples from his book. An in-depth examination of APM is outside of the scope of this book.

To utilize the Asynchronous Programming Model but avoid its awkward API, we can use the *Observable.FromAsyncPattern* method. Jeffery van Gogh gives a brilliant walk through of the *Observable.FromAsyncPattern* in Part 1 of his *Rx on the Server* blog series. While the theory backing the Rx on the Server series is sound, it was written in mid 2010 and targets an old version of Rx.

With 30 overloads of *Observable.FromAsyncPattern* we will look at the general concept so that you can pick the appropriate overload for yourself. First if we look at the normal pattern of APM we will see that the BeginXXX method will take zero or more data arguments followed by an `AsyncCallback` and an *Object*. The BeginXXX method will also return an *IAsyncResult* token.

```
//Standard Begin signature
IAsyncResult BeginXXX(AsyncCallback callback, Object state);
//Standard Begin signature with data
IAsyncResult BeginYYY(string someParam1, AsyncCallback callback, object state);
```

The EndXXX method will accept an *IAsyncResult* which should be the token returned from the BeginXXX method. The EndXXX can also return a value.

```
//Standard EndXXX Signature
void EndXXX(IAsyncResult asyncResult);
//Standard EndXXX Signature with data
int EndYYY(IAsyncResult asyncResult);
```

The generic arguments for the *FromAsyncPattern* method are just the BeginXXX data arguments if any, followed by the EndXXX return type if any. If we apply that to our `Stream.Read(byte[], int, int, AsyncResult,`

`object)` example above we see that we have a `byte[]`, an `int` and another `int` as our data parameters for *BeginRead* method.

```
//IAsyncResult BeginRead(
//  byte[] buffer,
//  int offset,
//  int count,
//  AsyncCallback callback, object state) {...}
Observable.FromAsyncPattern<byte[], int, int ...
```

Now we look at the EndXXX method and see it returns an `int`, which completes the generic signature of our *FromAsyncPattern* call.

```
//int EndRead(
//  IAsyncResult asyncResult) {...}
Observable.FromAsyncPattern<byte[], int, int, int>
```

The result of the call to *Observable.FromAsyncPattern* does *not* return an observable sequence. It returns a delegate that returns an observable sequence. The signature for this delegate will match the generic arguments of the call to *FromAsyncPattern*, except that the return type will be wrapped in an observable sequence.

```
var fileLength = (int) stream.Length;
//read is a Func<byte[], int, int, IObservable<int>>
var read = Observable.FromAsyncPattern<byte[], int, int, int>(
        stream.BeginRead,
        stream.EndRead);
var buffer = new byte[fileLength];
var bytesReadStream = read(buffer, 0, fileLength);
bytesReadStream.Subscribe(
        byteCount =>
        {
        Console.WriteLine("Number of bytes read={0}, buffer should be populated with data now.",
        byteCount);
        });
```

Note that this implementation is just an example. For a very well designed implementation that is built against the latest version of Rx you should look at the Rxx project on http://rxx.codeplex.com.

This covers the first classification of query operators: creating observable sequences. We have looked at the various eager and lazy ways to create a sequence. We have introduced the concept of corecursion and show how we can use it with the *Generate* method to unfold potentially infinite sequences. We can now produce timer based sequences using the various factory methods. We should also be familiar with ways to transition from other synchronous and asynchronous paradigms and be able to decide when it is or is not appropriate to do so. As a quick recap:

- Factory Methods
  - Observable.Return
  - Observable.Empty
  - Observable.Never
  - Observable.Throw
  - Observable.Create
- Unfold methods
  - Observable.Range
  - Observable.Interval
  - Observable.Timer
  - Observable.Generate
- Paradigm Transition
  - Observable.Start

- Observable.FromEventPattern
- Task.ToObservable
- Task<T>.ToObservable
- IEnumerable<T>.ToObservable
- Observable.FromAsyncPattern

Creating an observable sequence is our first step to practical application of Rx: create the sequence and then expose it for consumption. Now that we have a firm grasp on how to create an observable sequence, we can discover the operators that allow us to query an observable sequence.

Reducing a sequence

We live in the information age. Data is being created, stored and distributed at a phenomenal rate. Consuming this data can be overwhelming, like trying to drink directly from the fire hose. We need the ability to pick out the data we need, choose what is and is not relevant, and roll up groups of data to make it relevant. Users, customers and managers need you do this with more data than ever before, while still delivering higher performance and tighter deadlines.

Given that we know how to create an observable sequence, we will now look at the various methods that can reduce an observable sequence. We can categorize operators that reduce a sequence to the following:

Filter and partition operators
: Reduce the source sequence to a sequence with at most the same number of elements

Aggregation operators
: Reduce the source sequence to a sequence with a single element

Fold operators
: Reduce the source sequence to a single element as a scalar value

We discovered that the creation of an observable sequence from a scalar value is defined as *anamorphism* or described as an '*unfold*'. We can think of the anamorphism from T to *IObservable<T>* as an 'unfold'. This could also be referred to as "entering the monad" where in this case (and for most cases in this book) the monad is *IObservable<T>*. What we will now start looking at are methods that eventually get us to the inverse which is defined as *catamorphism* or a *fold*. Other popular names for fold are 'reduce', 'accumulate' and 'inject'.

Where

Applying a filter to a sequence is an extremely common exercise and the most common filter is the *Where* clause. In Rx you can apply a where clause with the *Where* extension method. For those that are unfamiliar, the signature of the *Where* method is as follows:

```
IObservable<T> Where(this IObservable<T> source, Fun<T, bool> predicate)
```

Note that both the source parameter and the return type are the same. This allows for a fluent interface, which is used heavily throughout Rx and other LINQ code. In this example we will use the *Where* to filter out all even values produced from a *Range* sequence.

```
var oddNumbers = Observable.Range(0, 10)
        .Where(i => i % 2 == 0)
        .Subscribe(
        Console.WriteLine,
        () => Console.WriteLine("Completed"));
```

Output:

```
0
2
4
6
8
Completed
```

The *Where* operator is one of the many standard LINQ operators. This and other LINQ operators are common use in the various implementations of query operators, most notably the *IEnumerable<T>* implementation. In most cases the operators behave just as they do in the *IEnumerable<T>* implementations, but there are some exceptions. We will discuss each implementation and explain any variation as we go. By implementing these common operators Rx also gets language support for free via C# query comprehension syntax. For the examples in this book however, we will keep with using extension methods for consistency.

Distinct and DistinctUntilChanged

As I am sure most readers are familiar with the *Where* extension method for *IEnumerable<T>*, some will also know the *Distinct* method. In Rx, the *Distinct* method has been made available for observable sequences too. For those that are unfamiliar with *Distinct*, and as a recap for those that are, *Distinct* will only pass on values from the source that it has not seen before.

```
var subject = new Subject<int>();
var distinct = subject.Distinct();
subject.Subscribe(
        i => Console.WriteLine("{0}", i),
        () => Console.WriteLine("subject.OnCompleted()"));
distinct.Subscribe(
        i => Console.WriteLine("distinct.OnNext({0})", i),
        () => Console.WriteLine("distinct.OnCompleted()"));
subject.OnNext(1);
subject.OnNext(2);
subject.OnNext(3);
subject.OnNext(1);
subject.OnNext(1);
subject.OnNext(4);
subject.OnCompleted();
```

Output:

```
1
distinct.OnNext(1)
2
distinct.OnNext(2)
3
distinct.OnNext(3)
1
1
4
distinct.OnNext(4)
subject.OnCompleted()
distinct.OnCompleted()
```

Take special note that the value 1 is pushed 3 times but only passed through the first time. There are overloads to *Distinct* that allow you to specialize the way an item is determined to be distinct or not. One way is to provide a function that returns a different value to use for comparison. Here we look at an example that uses a property from a custom class to define if a value is distinct.

```
public class Account
{
        public int AccountId { get; set; }
        //... etc
}
public void Distinct_with_KeySelector()
{
        var subject = new Subject<Account>();
        var distinct = subject.Distinct(acc => acc.AccountId);
}
```

In addition to the `keySelector` function that can be provided, there is an overload that takes an *IEqualityComparer<T>* instance. This is useful if you have a custom implementation that you can reuse to compare instances of your type `T`. Lastly there is an overload that takes a `keySelector` and an instance of *IEqualityComparer<TKey>*. Note that the equality comparer in this case is aimed at the selected key type (`TKey`), not the type `T`.

A variation of *Distinct*, that is peculiar to Rx, is *DistinctUntilChanged*. This method will surface values only if they are different from the previous value. Reusing our first *Distinct* example, note the change in output.

```
var subject = new Subject<int>();
var distinct = subject.DistinctUntilChanged();
subject.Subscribe(
        i => Console.WriteLine("{0}", i),
        () => Console.WriteLine("subject.OnCompleted()"));
distinct.Subscribe(
        i => Console.WriteLine("distinct.OnNext({0})", i),
        () => Console.WriteLine("distinct.OnCompleted()"));
subject.OnNext(1);
subject.OnNext(2);
subject.OnNext(3);
subject.OnNext(1);
subject.OnNext(1);
subject.OnNext(4);
subject.OnCompleted();
```

Output:

```
1
distinct.OnNext(1)
2
distinct.OnNext(2)
3
distinct.OnNext(3)
1
distinct.OnNext(1)
1
4
distinct.OnNext(4)
subject.OnCompleted()
distinct.OnCompleted()
```

The difference between the two examples is that the value 1 is pushed twice. However the third time that the source pushes the value 1, it is immediately after the second time value 1 is pushed. In this case it is ignored. Teams I have worked with have found this method to be extremely useful in reducing any noise that a sequence may provide.

IgnoreElements

The *IgnoreElements* extension method is a quirky little tool that allows you to receive the *OnCompleted* or *OnError* notifications. We could effectively recreate it by using a *Where* method with a predicate that always returns false.

```
var subject = new Subject<int>();
//Could use subject.Where(_=>false);
var noElements = subject.IgnoreElements();
subject.Subscribe(
        i=>Console.WriteLine("subject.OnNext({0})", i),
        () => Console.WriteLine("subject.OnCompleted()"));
noElements.Subscribe(
        i=>Console.WriteLine("noElements.OnNext({0})", i),
        () => Console.WriteLine("noElements.OnCompleted()"));
subject.OnNext(1);
subject.OnNext(2);
subject.OnNext(3);
subject.OnCompleted();
```

Output:

```
subject.OnNext(1)
subject.OnNext(2)
subject.OnNext(3)
subject.OnCompleted()
noElements.OnCompleted()
```

As suggested earlier we could use a *Where* to produce the same result

```
subject.IgnoreElements();
//Equivalent to
subject.Where(value=>false);
//Or functional style that implies that the value is ignored.
subject.Where(_=>false);
```

Just before we leave *Where* and *IgnoreElements*, I wanted to just quickly look at the last line of code. Until recently, I personally was not aware that '_' was a valid variable name; however it is commonly used by functional programmers to indicate an ignored parameter. This is perfect for the above example; for each value we receive, we ignore it and always return false. The intention is to improve the readability of the code via convention.

Skip and Take

The other key methods to filtering are so similar I think we can look at them as one big group. First we will look at *Skip* and *Take*. These act just like they do for the *IEnumerable<T>* implementations. These are the most simple and probably the most used of the Skip/Take methods. Both methods just have the one parameter; the number of values to skip or to take.

If we first look at *Skip*, in this example we have a range sequence of 10 items and we apply a `Skip(3)` to it.

```
Observable.Range(0, 10)
        .Skip(3)
        .Subscribe(Console.WriteLine, () => Console.WriteLine("Completed"));
```

Output:

```
3
4
5
6
7
8
9
Completed
```

Note the first three values (0, 1 & 2) were all ignored from the output. Alternatively, if we used `Take(3)` we would get the opposite result; i.e. we would only get the first 3 values and then the Take operator would complete the sequence.

```
Observable.Range(0, 10)
        .Take(3)
        .Subscribe(Console.WriteLine, () => Console.WriteLine("Completed"));
```

Output:

```
0
1
2
Completed
```

Just in case that slipped past any readers, it is the *Take* operator that completes once it has received its count. We can prove this by applying it to an infinite sequence.

```
Observable.Interval(TimeSpan.FromMilliseconds(100))
        .Take(3)
        .Subscribe(Console.WriteLine, () => Console.WriteLine("Completed"));
```

Output:

```
0
1
2
Completed
```

SkipWhile and TakeWhile

The next set of methods allows you to skip or take values from a sequence while a predicate evaluates to true. For a *SkipWhile* operation this will filter out all values until a value fails the predicate, then the remaining sequence can be returned.

```
var subject = new Subject<int>();
subject
        .SkipWhile(i => i < 4)
        .Subscribe(Console.WriteLine, () => Console.WriteLine("Completed"));
subject.OnNext(1);
subject.OnNext(2);
subject.OnNext(3);
subject.OnNext(4);
```

```
subject.OnNext(3);
subject.OnNext(2);
subject.OnNext(1);
subject.OnNext(0);
subject.OnCompleted();
```

Output:

```
4
3
2
1
0
Completed
```

TakeWhile will return all values while the predicate passes, and when the first value fails the sequence will complete.

```
var subject = new Subject<int>();
subject
        .TakeWhile(i => i < 4)
        .Subscribe(Console.WriteLine, () => Console.WriteLine("Completed"));
subject.OnNext(1);
subject.OnNext(2);
subject.OnNext(3);
subject.OnNext(4);
subject.OnNext(3);
subject.OnNext(2);
subject.OnNext(1);
subject.OnNext(0);
subject.OnCompleted();
```

Output:

```
1
2
3
Completed
```

SkipLast and TakeLast

These methods become quite self explanatory now that we understand Skip/Take and SkipWhile/TakeWhile. Both methods require a number of elements at the end of a sequence to either skip or take. The implementation of the *SkipLast* could cache all values, wait for the source sequence to complete, and then replay all the values except for the last number of elements. The Rx team however, has been a bit smarter than that. The real implementation will queue the specified number of notifications and once the queue size exceeds the value, it can be sure that it may drain a value from the queue.

```
var subject = new Subject<int>();
subject
        .SkipLast(2)
        .Subscribe(Console.WriteLine, () => Console.WriteLine("Completed"));
Console.WriteLine("Pushing 1");
subject.OnNext(1);
Console.WriteLine("Pushing 2");
subject.OnNext(2);
Console.WriteLine("Pushing 3");
subject.OnNext(3);
Console.WriteLine("Pushing 4");
subject.OnNext(4);
subject.OnCompleted();
```

Output:

```
Pushing 1
Pushing 2
Pushing 3
1
Pushing 4
2
Completed
```

Unlike *SkipLast*, *TakeLast* does have to wait for the source sequence to complete to be able to push its results. As per the example above, there are `Console.WriteLine` calls to indicate what the program is doing at each stage.

```
var subject = new Subject<int>();
subject
        .SkipLast(2)
```

```
        .Subscribe(Console.WriteLine, () => Console.WriteLine("Completed"));
Console.WriteLine("Pushing 1");
subject.OnNext(1);
Console.WriteLine("Pushing 2");
subject.OnNext(2);
Console.WriteLine("Pushing 3");
subject.OnNext(3);
Console.WriteLine("Pushing 4");
subject.OnNext(4);
Console.WriteLine("Completing");
subject.OnCompleted();
```

Output:

```
Pushing 1
Pushing 2
Pushing 3
Pushing 4
Completing
3
4
Completed
```

SkipUntil and TakeUntil

Our last two methods make an exciting change to the methods we have previously looked. These will be the first two methods that we have discovered together that require two observable sequences.

SkipUntil will skip all values until any value is produced by a secondary observable sequence.

```
var subject = new Subject<int>();
var otherSubject = new Subject<Unit>();
subject
        .SkipUntil(otherSubject)
        .Subscribe(Console.WriteLine, () => Console.WriteLine("Completed"));
subject.OnNext(1);
subject.OnNext(2);
subject.OnNext(3);
otherSubject.OnNext(Unit.Default);
subject.OnNext(4);
subject.OnNext(5);
subject.OnNext(6);
subject.OnNext(7);
subject.OnNext(8);
subject.OnCompleted();
```

Output:

```
4
5
6
7
Completed
```

Obviously, the converse is true for *TakeWhile*. When the secondary sequence produces a value, then the *TakeWhile* operator will complete the output sequence.

```
var subject = new Subject<int>();
var otherSubject = new Subject<Unit>();
subject
        .TakeUntil(otherSubject)
        .Subscribe(Console.WriteLine, () => Console.WriteLine("Completed"));
subject.OnNext(1);
subject.OnNext(2);
subject.OnNext(3);
otherSubject.OnNext(Unit.Default);
subject.OnNext(4);
subject.OnNext(5);
subject.OnNext(6);
subject.OnNext(7);
subject.OnNext(8);
subject.OnCompleted();
```

Output:

```
1
2
3
Completed
```

That was our quick run through of the filtering methods available in Rx. While they are pretty simple, as we will see, the power in Rx is down to the composability of its operators.

These operators provide a good introduction to the filtering in Rx. The filter operators are your first

stop for managing the potential deluge of data we can face in the information age. We now know how to remove unmatched data, duplicate data or excess data. Next, we will move on to the other two sub classifications of the reduction operators, inspection and aggregation.

Inspection

Making sense of all the data we consume is not always about just filtering out the redundant and superfluous. Sometimes we need to pluck out data that is relevant or validate that a sequence even meets our expectations. Does this data have any values that meet this specification? Is this specific value in the sequence? Get me that specific value from the sequence!

In the last chapter we looked at a series of ways to reduce your observable sequence via a variety of filters. Next we will look at operators that provide inspection functionality. Most of these operators will reduce your observable sequence down to a sequence with a single value in it. As the return value of these methods is not a scalar (it is still IObservable<T>) these methods do not actually satisfy our definition of catamorphism, but suit our examination of reducing a sequence to a single value.

The series of methods we will look at next are useful for inspecting a given sequence. Each of them returns an observable sequence with the single value containing the result. This proves useful, as by their nature they are asynchronous. They are all quite simple so we will be brief with each of them.

Any

First we can look at the parameterless overload for the extension method *Any*. This will simply return an observable sequence that has the single value of `false` if the source completes without any values. If the source does produce a value however, then when the first value is produced, the result sequence will immediately push `true` and then complete. If the first notification it gets is an error, then it will pass that error on.

```
var subject = new Subject<int>();
subject.Subscribe(Console.WriteLine, () => Console.WriteLine("Subject completed"));
var any = subject.Any();
any.Subscribe(b => Console.WriteLine("The subject has any values? {0}", b));
subject.OnNext(1);
subject.OnCompleted();
```

Output:

```
1
The subject has any values? True
subject completed
```

If we now remove the OnNext(1), the output will change to the following

```
subject completed
The subject has any values? False
```

If the source errors it would only be interesting if it was the first notification, otherwise the *Any* method will have already pushed true. If the first notification is an error then *Any* will just pass it along as an *OnError* notification.

```
var subject = new Subject<int>();
subject.Subscribe(Console.WriteLine,
        ex => Console.WriteLine("subject OnError : {0}", ex),
        () => Console.WriteLine("Subject completed"));
var any = subject.Any();
any.Subscribe(b => Console.WriteLine("The subject has any values? {0}", b),
        ex => Console.WriteLine(".Any() OnError : {0}", ex),
        () => Console.WriteLine(".Any() completed"));
subject.OnError(new Exception());
```

Output:

```
subject OnError : System.Exception: Fail
.Any() OnError : System.Exception: Fail
```

The *Any* method also has an overload that takes a predicate. This effectively makes it a *Where* with an *Any* appended to it.

```
subject.Any(i => i > 2);
//Functionally equivalent to
subject.Where(i => i > 2).Any();
```

As an exercise, write your own version of the two *Any* extension method overloads. While the answer may not be immediately obvious, we have covered enough material for you to create this using the methods you know...

Example of the *Any* extension methods written with Observable.Create:

```
public static IObservable<bool> MyAny<T>(
        this IObservable<T> source)
{
        return Observable.Create<bool>(
        o =>
        {
        var hasValues = false;
        return source
        .Take(1)
        .Subscribe(
                _ => hasValues = true,
                o.OnError,
                () =>
                {
                o.OnNext(hasValues);
                o.OnCompleted();
                });
        });
}
public static IObservable<bool> MyAny<T>(
        this IObservable<T> source,
        Func<T, bool> predicate)
{
        return source
        .Where(predicate)
        .MyAny();
}
```

All

The *All*() extension method works just like the *Any* method, except that all values must meet the predicate. As soon as a value does not meet the predicate a `false` value is returned then the output sequence completed. If the source is empty, then *All* will push `true` as its value. As per the *Any* method, and errors will be passed along to the subscriber of the *All* method.

```
var subject = new Subject<int>();
subject.Subscribe(Console.WriteLine, () => Console.WriteLine("Subject completed"));
var all = subject.All(i => i < 5);
all.Subscribe(b => Console.WriteLine("All values less than 5? {0}", b));
subject.OnNext(1);
subject.OnNext(2);
subject.OnNext(6);
subject.OnNext(2);
subject.OnNext(1);
subject.OnCompleted();
```

Output:

```
1
2
6
All values less than 5? False
all completed
2
1
subject completed
```

Early adopters of Rx may notice that the *IsEmpty* extension method is missing. You can easily replicate the missing method using the *All* extension method.

```
//IsEmpty() is deprecated now.
//var isEmpty = subject.IsEmpty();
var isEmpty = subject.All(_ => false);
```

Contains

The *Contains* extension method overloads could sensibly be overloads to the *Any* extension method. The *Contains* extension method has the same behavior as *Any*, however it specifically targets the use of *IComparable* instead of the usage of predicates and is designed to seek a specific value instead of a value that fits the predicate. I believe that these are not overloads of *Any* for consistency with *IEnumerable*.

```
var subject = new Subject<int>();
subject.Subscribe(
        Console.WriteLine,
        () => Console.WriteLine("Subject completed"));
var contains = subject.Contains(2);
contains.Subscribe(
        b => Console.WriteLine("Contains the value 2? {0}", b),
        () => Console.WriteLine("contains completed"));
subject.OnNext(1);
subject.OnNext(2);
subject.OnNext(3);
subject.OnCompleted();
```

Output:

```
1
2
Contains the value 2? True
contains completed
3
Subject completed
```

There is also an overload to *Contains* that allows you to specify an implementation of *IEqualityComparer<T>* other than the default for the type. This can be useful if you have a sequence of custom types that may have some special rules for equality depending on the use case.

DefaultIfEmpty

The *DefaultIfEmpty* extension method will return a single value if the source sequence is empty. Depending on the overload used, it will either be the value provided as the default, or `Default(T)`. `Default(T)` will be the zero value for struct types and will be null for classes. If the source is not empty then all values will be passed straight on through.

In this example the source produces values, so the result of *DefaultIfEmpty* is just the source.

```
var subject = new Subject<int>();
subject.Subscribe(
        Console.WriteLine,
        () => Console.WriteLine("Subject completed"));
var defaultIfEmpty = subject.DefaultIfEmpty();
defaultIfEmpty.Subscribe(
        b => Console.WriteLine("defaultIfEmpty value: {0}", b),
        () => Console.WriteLine("defaultIfEmpty completed"));
subject.OnNext(1);
subject.OnNext(2);
subject.OnNext(3);
subject.OnCompleted();
```

Output:

```
1
defaultIfEmpty value: 1
2
defaultIfEmpty value: 2
3
defaultIfEmpty value: 3
Subject completed
defaultIfEmpty completed
```

If the source is empty, we can use either the default value for the type (i.e. 0 for int) or provide our own value in this case 42.

```
var subject = new Subject<int>();
subject.Subscribe(
        Console.WriteLine,
        () => Console.WriteLine("Subject completed"));
var defaultIfEmpty = subject.DefaultIfEmpty();
defaultIfEmpty.Subscribe(
        b => Console.WriteLine("defaultIfEmpty value: {0}", b),
        () => Console.WriteLine("defaultIfEmpty completed"));
var default42IfEmpty = subject.DefaultIfEmpty(42);
default42IfEmpty.Subscribe(
        b => Console.WriteLine("default42IfEmpty value: {0}", b),
        () => Console.WriteLine("default42IfEmpty completed"));
subject.OnCompleted();
```

Output:

```
Subject completed
defaultIfEmpty value: 0
defaultIfEmpty completed
default42IfEmpty value: 42
default42IfEmpty completed
```

ElementAt

The **ElementAt** extension method allows us to "cherry pick" out a value at a given index. Like the *IEnumerable<T>* version it is uses a 0 based index.

```
var subject = new Subject<int>();
subject.Subscribe(
        Console.WriteLine,
        () => Console.WriteLine("Subject completed"));
var elementAt1 = subject.ElementAt(1);
elementAt1.Subscribe(
        b => Console.WriteLine("elementAt1 value: {0}", b),
        () => Console.WriteLine("elementAt1 completed"));
subject.OnNext(1);
subject.OnNext(2);
subject.OnNext(3);
subject.OnCompleted();
```

Output

```
1
2
elementAt1 value: 2
elementAt1 completed
3
subject completed
```

As we can't check the length of an observable sequence it is fair to assume that sometimes this method could cause problems. If your source sequence only produces five values and we ask for `ElementAt(5)`, the result sequence will error with an *ArgumentOutOfRangeException* inside when the source completes. There are three options we have:

- Handle the OnError gracefully
- Use .Skip(5).Take(1); This will ignore the first 5 values and the only take the 6th value. If the sequence has less than 6 elements we just get an empty sequence, but no errors.
- Use ElementAtOrDefault

ElementAtOrDefault extension method will protect us in case the index is out of range, by pushing the `Default(T)` value. Currently there is not an option to provide your own default value.

SequenceEqual

Finally *SequenceEqual* extension method is perhaps a stretch to put in a chapter that starts off talking about catamorphism and fold, but it does serve well for the theme of inspection. This method allows us to compare two observable sequences. As each source sequence produces values, they are compared to a cache of the other sequence to ensure that each sequence has the same values in the same order and that the sequences are the same length. This means that the result sequence can return `false` as soon as the source sequences produce diverging values, or `true` when both sources complete with the same values.

```
var subject1 = new Subject<int>();
subject1.Subscribe(
        i=>Console.WriteLine("subject1.OnNext({0})", i),
        () => Console.WriteLine("subject1 completed"));
var subject2 = new Subject<int>();
subject2.Subscribe(
        i=>Console.WriteLine("subject2.OnNext({0})", i),
        () => Console.WriteLine("subject2 completed"));
var areEqual = subject1.SequenceEqual(subject2);
areEqual.Subscribe(
        i => Console.WriteLine("areEqual.OnNext({0})", i),
        () => Console.WriteLine("areEqual completed"));
subject1.OnNext(1);
subject1.OnNext(2);
subject2.OnNext(1);
subject2.OnNext(2);
subject2.OnNext(3);
subject1.OnNext(3);
subject1.OnCompleted();
subject2.OnCompleted();
```

Output:

```
subject1.OnNext(1)
subject1.OnNext(2)
subject2.OnNext(1)
subject2.OnNext(2)
subject2.OnNext(3)
subject1.OnNext(3)
subject1 completed
subject2 completed
areEqual.OnNext(True)
areEqual completed
```

This chapter covered a set of methods that allow us to inspect observable sequences. The result of each, generally, returns a sequence with a single value. We will continue to look at methods to reduce our sequence until we discover the elusive functional fold feature.

Aggregation

Data is not always valuable is its raw form. Sometimes we need to consolidate, collate, combine or condense the mountains of data we receive into more consumable bite sized chunks. Consider fast moving data from domains like instrumentation, finance, signal processing and operational intelligence. This kind of data can change at a rate of over ten values per second. Can a person actually consume this? Perhaps for human consumption, aggregate values like averages, minimums and maximums can be of more use.

Continuing with the theme of reducing an observable sequence, we will look at the aggregation functions that are available to us in Rx. Our first set of methods continues on from our last chapter, as they take an observable sequence and reduce it to a sequence with a single value. We then move on to find operators that can transition a sequence back to a scalar value, a functional fold.

Just before we move on to introducing the new operators, we will quickly create our own extension method. We will use this 'Dump' extension method to help build our samples.

```
public static class SampleExtentions
{
        public static void Dump<T>(this IObservable<T> source, string name)
        {
        source.Subscribe(
        i=>Console.WriteLine("{0}-->{1}", name, i),
        ex=>Console.WriteLine("{0} failed-->{1}", name, ex.Message),
        ()=>Console.WriteLine("{0} completed", name));
        }
}
```

Those who use LINQPad will recognize that this is the source of inspiration. For those who have not used LINQPad, I highly recommend it. It is perfect for whipping up quick samples to validate a snippet of code. LINQPad also fully supports the *IObservable<T>* type.

Count

Count is a very familiar extension method for those that use LINQ on *IEnumerable<T>*. Like all good method names, it "does what it says on the tin". The Rx version deviates from the *IEnumerable<T>* version as Rx will return an observable sequence, not a scalar value. The return sequence will have a single value being the count of the values in the source sequence. Obviously we cannot provide the count until the source sequence completes.

```
var numbers = Observable.Range(0,3);
numbers.Dump("numbers");
numbers.Count().Dump("count");
```

Output:

```
numbers-->1
numbers-->2
numbers-->3
numbers Completed
count-->3
count Completed
```

If you are expecting your sequence to have more values than a 32 bit integer can hold, there is the option to use the *LongCount* extension method. This is just the same as *Count* except it returns an *IObservable<long>*.

Min, Max, Sum and Average

Other common aggregations are *Min*, *Max*, *Sum* and *Average*. Just like *Count*, these all return a sequence with a single value. Once the source completes the result sequence will produce its value and then complete.

```
var numbers = new Subject<int>();
numbers.Dump("numbers");
numbers.Min().Dump("Min");
numbers.Average().Dump("Average");
numbers.OnNext(1);
numbers.OnNext(2);
numbers.OnNext(3);
numbers.OnCompleted();
```

Output:

```
numbers-->1
numbers-->2
numbers-->3
numbers Completed
min-->1
min Completed
avg-->2
avg Completed
```

The *Min* and *Max* methods have overloads that allow you to provide a custom implementation of an *IComparer<T>* to sort your values in a custom way. The *Average* extension method specifically calculates the mean (as opposed to median or mode) of the sequence. For sequences of integers (int or long) the output of *Average* will be an *IObservable<double>*. If the source is of nullable integers then the output will be *IObservable<double?>*. All other numeric types (*float*, *double*, *decimal* and their nullable equivalents) will result in the output sequence being of the same type as the input sequence.

Functional folds

Finally we arrive at the set of methods in Rx that meet the functional description of catamorphism/fold. These methods will take an *IObservable<T>* and produce a `T`.

Caution should be prescribed whenever using any of these fold methods on an observable sequence, as they are all blocking. The reason you need to be careful with blocking methods is that you are moving from an asynchronous paradigm to a synchronous one, and without care you can introduce concurrency problems such as locking UIs and deadlocks. We will take a deeper look into these problems in a later chapter when we look at concurrency.

> It is worth noting that in the soon to be released .NET 4.5 and Rx 2.0 will provide support for avoiding these concurrency problems. The new `async/await` keywords and related features in Rx 2.0 can help exit the monad in a safer way.

First

The *First()* extension method simply returns the first value from a sequence.

```
var interval = Observable.Interval(TimeSpan.FromSeconds(3));
//Will block for 3s before returning
Console.WriteLine(interval.First());
```

If the source sequence does not have any values (i.e. is an empty sequence) then the *First* method will throw an exception. You can cater for this in three ways:

- Use a try/catch blocks around the *First()* call
- Use *Take(1)* instead. However, this will be asynchronous, not blocking.
- Use *FirstOrDefault* extension method instead

The *FirstOrDefault* will still block until the source produces any notification. If the notification is an *OnError* then it will be thrown. If the notification is an *OnNext* then that value will be returned, otherwise if it is an *OnCompleted* the default will be returned. As we have seen in earlier methods, we can either choose to use the parameterless method in which the default value will be `default(T)` (i.e. null for reference types or the zero value for value types), alternatively we can provide our own default value to use.

A special mention should be made for the unique relationship that *BehaviorSubject* and the *First()* extension method has. The reason behind this is that the *BehaviorSubject* is guaranteed to have a notification, be it a value, an error or a completion. This effectively removes the blocking nature of the *First* extension method when used with a *BehaviorSubject*. This can be used to make behavior subjects act like properties.

Last

The *Last* and *LastOrDefault* will block until the source completes and then return the last value. Just like the *First()* method any *OnError* notifications will be thrown. If the sequence is empty then *Last()* will throw an *InvalidOperationException*, but you can use *LastOrDefault* to avoid this.

Single

The *Single* extension method is for getting the single value from a sequence. The difference between this and *First()* or *Last()* is that it helps to assert your assumption that the sequence will only contain a single value. The method will block until the source produces a value and then completes. If the sequence produces any other combination of notifications then the method will throw. This method works especially well with *AsyncSubject* instances as they only produce a single value sequences.

Build your own aggregations

If the provided aggregations do not meet your needs, you can build your own. Rx provides two different ways to do this.

Aggregate

The *Aggregate* method allows you to apply an accumulator function to the sequence. For the basic overload, you need to provide a function that takes the current state of the accumulated value and the value that the sequence is pushing. The result of the function is the new accumulated value. This overload signature is as follows:

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

If you wanted to produce your own version of *Sum* for `int` values, you could do so by providing a function that just adds to the current state of the accumulator.

```
var sum = source.Aggregate((acc, currentValue) => acc + currentValue);
```

This overload of *Aggregate* has several problems. First is that it requires the aggregated value must be the same type as the sequence values. We have already seen in other aggregates like *Average* this is not always the case. Secondly, this overload needs at least one value to be produced from the source or the output will error with an *InvalidOperationException*. It should be completely valid for us to use *Aggregate* to create our own *Count* or *Sum* on an empty sequence. To do this you need to use the other overload. This overload takes an extra parameter which is the seed. The seed value provides an initial accumulated value. It also allows the aggregate type to be different to the value type.

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

To update our *Sum* implementation to use this overload is easy. Just add the seed which will be 0. This will now return 0 as the sum when the sequence is empty which is just what we want. You also now can also create your own version of *Count*.

```
var sum = source.Aggregate(0, (acc, currentValue) => acc + currentValue);
var count = source.Aggregate(0, (acc, currentValue) => acc + 1);
//or using '_' to signify that the value is not used.
var count = source.Aggregate(0, (acc, _) => acc + 1);
```

As an exercise write your own *Min* and *Max* methods using *Aggregate*. You will probably find the *IComparer<T>* interface useful, and in particular the static `Comparer<T>.Default` property. When you have done the exercise, continue to the example implementations...

Examples of creating *Min* and *Max* from *Aggregate*:

```
public static IObservable<T> MyMin<T>(this IObservable<T> source)
{
        return source.Aggregate(
        (min, current) => Comparer<T>
        .Default
        .Compare(min, current) > 0
        ? current
        : min);
}
public static IObservable<T> MyMax<T>(this IObservable<T> source)
{
        var comparer = Comparer<T>.Default;
        Func<T, T, T> max =
        (x, y) =>
        {
        if(comparer.Compare(x, y) < 0)
        {
        return y;
        }
        return x;
        };
        return source.Aggregate(max);
}
```

Scan

While *Aggregate* allows us to get a final value for sequences that will complete, sometimes this is not what we need. If we consider a use case that requires that we get a running total as we receive values, then *Aggregate* is not a good fit. *Aggregate* is also not a good fit for infinite sequences. The *Scan* extension method however meets this requirement perfectly. The signatures for both *Scan* and *Aggregate* are the same; the difference is that *Scan* will push the *result* from every call to the accumulator function. So instead of being an aggregator that reduces a sequence to a single value sequence, it is an accumulator that we return an accumulated value for each value of the source sequence. In this example we produce a running total.

```
var numbers = new Subject<int>();
var scan = numbers.Scan(0, (acc, current) => acc + current);
numbers.Dump("numbers");
scan.Dump("scan");
numbers.OnNext(1);
numbers.OnNext(2);
numbers.OnNext(3);
numbers.OnCompleted();
```

Output:

```
numbers-->1
sum-->1
numbers-->2
sum-->3
numbers-->3
sum-->6
numbers completed
sum completed
```

It is probably worth pointing out that you use *Scan* with *TakeLast()* to produce *Aggregate*.

```
source.Aggregate(0, (acc, current) => acc + current);
//is equivalent to
source.Scan(0, (acc, current) => acc + current).TakeLast();
```

As another exercise, use the methods we have covered so far in the book to produce a sequence of running minimum and running maximums. The key here is that each time we receive a value that is less than (or more than for a Max operator) our current accumulator we should push that value and update the accumulator value. We don't however want to push duplicate values. For example, given a sequence of [2, 1, 3, 5, 0] we should see output like [2, 1, 0] for the running minimum, and [2, 3, 5] for the running maximum. We don't want to see [2, 1, 2, 2, 0] or [2, 2, 3, 5, 5]. Continue to see an example implementation.

Example of a running minimum:

```
var comparer = Comparer<T>.Default;
Func<T,T,T> minOf = (x, y) => comparer.Compare(x, y) < 0 ? x: y;
var min = source.Scan(minOf)
        .DistinctUntilChanged();
```

Example of a running maximum:

```
public static IObservable<T> RunningMax<T>(this IObservable<T> source)
{
        return source.Scan(MaxOf)
        .Distinct();
}
private static T MaxOf<T>(T x, T y)
{
        var comparer = Comparer<T>.Default;
        if (comparer.Compare(x, y) < 0)
        {
        return y;
        }
        return x;
}
```

While the only functional differences between the two examples is checking greater instead of less than, the examples show two different styles. Some people prefer the terseness of the first example, others like their curly braces and the verbosity of the second example. The key here was to compose the *Scan* method with the *Distinct* or *DistinctUntilChanged* methods. It is probably preferable to use the *DistinctUntilChanged* so that we internally are not keeping a cache of all values.

Partitioning

Rx also gives you the ability to partition your sequence with features like the standard LINQ operator *GroupBy*. This can be useful for taking a single sequence and fanning out to many subscribers or perhaps taking aggregates on partitions.

MinBy and MaxBy

The *MinBy* and *MaxBy* operators allow you to partition your sequence based on a key selector function. Key selector functions are common in other LINQ operators like the *IEnumerable<T>ToDictionary* or *GroupBy* and the *Distinct* method. Each method will return you the values from the key that was the minimum or maximum respectively.

```
// Returns an observable sequence containing a list of zero or more elements that have a
//  minimum key value.
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)
{...}
// Returns an observable sequence containing a list of zero or more elements that have a
//  maximum key value.
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)
{...}
```

Take note that each *Min* and *Max* operator has an overload that takes a comparer. This allows for comparing custom types or custom sorting of standard types.

Consider a sequence from 0 to 10. If we apply a key selector that partitions the values in to groups based on their modulus of 3, we will have 3 groups of values. The values and their keys will be as follows:

```
Func<int, int> keySelector = i => i % 3;
```

- 0, key: 0
- 1, key: 1
- 2, key: 2
- 3, key: 0
- 4, key: 1
- 5, key: 2
- 6, key: 0
- 7, key: 1
- 8, key: 2
- 9, key: 0

We can see here that the minimum key is 0 and the maximum key is 2. If therefore, we applied the *MinBy* operator our single value from the sequence would be the list of [0,3,6,9]. Applying the *MaxBy* operator would produce the list [2,5,8]. The *MinBy* and *MaxBy* operators will only yield a

single value (like an *AsyncSubject*) and that value will be an *IList<T>* with zero or more values.

If instead of the values for the minimum/maximum key, you wanted to get the minimum value for each key, then you would need to look at *GroupBy*.

GroupBy

The *GroupBy* operator allows you to partition your sequence just as *IEnumerable<T>*'s *GroupBy* operator does. In a similar fashion to how the *IEnumerable<T>* operator returns an *IEnumerable<IGrouping<TKey, T>>*, the *IObservable<T>GroupBy* operator returns an *IObservable<IGroupedObservable<TKey, T>>*.

```
// Transforms a sequence into a sequence of observable groups,
//  each of which corresponds to a unique key value,
//  containing all elements that share that same key value.
public static IObservable<IGroupedObservable<TKey, TSource>> GroupBy<TSource, TKey>(
        this IObservable<TSource> source,
        Func<TSource, TKey> keySelector)
{...}
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)
{...}
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)
{...}
```

I find the last two overloads a little redundant as we could easily just compose a *Select* operator to the query to get the same functionality.

In a similar fashion that the *IGrouping<TKey, T>* type extends the *IEnumerable<T>*, the *IGroupedObservable<T>* just extends *IObservable<T>* by adding a *Key* property. The use of the *GroupBy* effectively gives us a nested observable sequence.

To use the *GroupBy* operator to get the minimum/maximum value for each key, we can first partition the sequence and then *Min*/*Max* each partition.

```
var source = Observable.Interval(TimeSpan.FromSeconds(0.1)).Take(10);
var group = source.GroupBy(i => i % 3);
group.Subscribe(
        grp =>
        grp.Min().Subscribe(
        minValue =>
        Console.WriteLine("{0} min value = {1}", grp.Key, minValue)),
        () => Console.WriteLine("Completed"));
```

The code above would work, but it is not good practice to have these nested subscribe calls. We have lost control of the nested subscription, and it is hard to read. When you find yourself creating nested subscriptions, you should consider how to apply a better pattern. In this case we can use *SelectMany* which we will look at in the next chapter.

```
var source = Observable.Interval(TimeSpan.FromSeconds(0.1)).Take(10);
var group = source.GroupBy(i => i % 3);
group.SelectMany(
        grp =>
        grp.Max()
        .Select(value => new { grp.Key, value }))
.Dump("group");
```

Nested observables

The concept of a sequence of sequences can be somewhat overwhelming at first, especially if both

sequence types are *IObservable*. While it is an advanced topic, we will touch on it here as it is a common occurrence with Rx. I find it easier if I can conceptualize a scenario or example to understand concepts better.

Examples of Observables of Observables:

Partitions of Data
: You may partition data from a single source so that it can easily be filtered and shared to many sources. Partitioning data may also be useful for aggregates as we have seen. This is commonly done with the *GroupBy* operator.

Online Game servers
: Consider a sequence of servers. New values represent a server coming online. The value itself is a sequence of latency values allowing the consumer to see real time information of quantity and quality of servers available. If a server went down then the inner sequence can signify that by completing.

Financial data streams
: New markets or instruments may open and close during the day. These would then stream price information and could complete when the market closes.

Chat Room
: Users can join a chat (outer sequence), leave messages (inner sequence) and leave a chat (completing the inner sequence).

File watcher
: As files are added to a directory they could be watched for modifications (outer sequence). The inner sequence could represent changes to the file, and completing an inner sequence could represent deleting the file.

Considering these examples, you could see how useful it could be to have the concept of nested observables. There are a suite of operators that work very well with nested observables such as *SelectMany*, *Merge* and *Switch* that we look at in future chapters.

When working with nested observables, it can be handy to adopt the convention that a new sequence represents a creation (e.g. A new partition is created, new game host comes online, a market opens, users joins a chat, creating a file in a watched directory). You can then adopt the convention for what a completed inner sequence represents (e.g. Game host goes offline, Market Closes, User leave chat, File being watched is deleted). The great thing with nested observables is that a completed inner sequence can effectively be restarted by creating a new inner sequence.

In this chapter we are starting to uncover the power of LINQ and how it applies to Rx. We chained methods together to recreate the effect that other methods already provide. While this is academically nice, it also allows us to starting thinking in terms of functional composition. We have also seen that some methods work nicely with certain types: `First()` + *BehaviorSubject<T>*, `Single()` + *AsyncSubject<T>*, `Single()` + `Aggregate()` etc. We have covered the second of our three classifications of operators, *catamorphism*. Next we will discover more methods to add to our functional composition tool belt and also find how Rx deals with our third functional concept, *bind*.

Consolidating data into groups and aggregates enables sensible consumption of mass data. Fast

moving data can be too overwhelming for batch processing systems and human consumption. Rx provides the ability to aggregate and partition on the fly, enabling real-time reporting without the need for expensive CEP or OLAP products.

Transformation of sequences

The values from the sequences we consume are not always in the format we need. Sometimes there is too much noise in the data so we strip the values down. Sometimes each value needs to be expanded either into a richer object or into more values. By composing operators, Rx allows you to control the quality as well as the quantity of values in the observable sequences you consume.

Up until now, we have looked at creation of sequences, transition into sequences, and, the reduction of sequences by filtering, aggregating or folding. In this chapter we will look at *transforming* sequences. This allows us to introduce our third category of functional methods, *bind*. A bind function in Rx will take a sequence and apply some set of transformations on each element to produce a new sequence. To review:

Ana(morphism) T --> IObservable<T>

Cata(morphism) IObservable<T> --> T

Bind IObservable<T1> --> IObservable<T2>

Now that we have been introduced to all three of our higher order functions, you may find that you already know them. Bind and Cata(morphism) were made famous by MapReduce framework from Google. Here Google refer to Bind and Cata by their perhaps more common aliases; Map and Reduce.

It may help to remember our terms as the *ABCs* of higher order functions.

Ana enters the sequence. T --> IObservable<T>

Bind modifies the sequence. IObservable<T1> --> IObservable<T2>

Cata leaves the sequence. IObservable<T> --> T

Select

The classic transformation method is *Select*. It allows you provide a function that takes a value of `TSource` and return a value of `TResult`. The signature for *Select* is nice and simple and suggests that its most common usage is to transform from one type to another type, i.e. *IObservable<TSource>* to *IObservable<TResult>*.

```
IObservable<TResult> Select<TSource, TResult>(
        this IObservable<TSource> source,
        Func<TSource, TResult> selector)
```

Note that there is no restriction that prevents `TSource` and `TResult` being the same thing. So for our first example, we will take a sequence of integers and transform each value by adding 3, resulting in another sequence of integers.

```
var source = Observable.Range(0, 5);
source.Select(i=>i+3)
        .Dump("+3")
```

Output:

```
+3-->3
+3-->4
+3-->5
+3-->6
+3-->7
+3 completed
```

While this can be useful, more common use is to transform values from one type to another. In this example we transform integer values to characters.

```
Observable.Range(1, 5);
        .Select(i =>(char)(i + 64))
        .Dump("char");
```

Output:

```
char-->A
char-->B
char-->C
char-->D
char-->E
char completed
```

If we really want to take advantage of LINQ we could transform our sequence of integers to a sequence of anonymous types.

```
Observable.Range(1, 5)
        .Select(
        i => new { Number = i, Character = (char)(i + 64) })
        .Dump("anon");
```

Output:

```
anon-->{ Number = 1, Character = A }
anon-->{ Number = 2, Character = B }
anon-->{ Number = 3, Character = C }
anon-->{ Number = 4, Character = D }
anon-->{ Number = 5, Character = E }
anon completed
```

To further leverage LINQ we could write the above query using query comprehension syntax.

```
var query = from i in Observable.Range(1, 5)
        select new {Number = i, Character = (char) (i + 64)};
query.Dump("anon");
```

In Rx, *Select* has another overload. The second overload provides two values to the `selector` function. The additional argument is the element's index in the sequence. Use this method if the index of the

element in the sequence is important to your selector function.

Cast and OfType

If you were to get a sequence of objects i.e. *IObservable<object>*, you may find it less than useful. There is a method specifically for *IObservable<object>* that will cast each element to a given type, and logically it is called *Cast<T>()*.

```
var objects = new Subject<object>();
objects.Cast<int>().Dump("cast");
objects.OnNext(1);
objects.OnNext(2);
objects.OnNext(3);
objects.OnCompleted();
```

Output:

```
cast-->1
cast-->2
cast-->3
cast completed
```

If however we were to add a value that could not be cast into the sequence then we get errors.

```
var objects = new Subject<object>();
objects.Cast<int>().Dump("cast");
objects.OnNext(1);
objects.OnNext(2);
objects.OnNext("3");//Fail
```

Output:

```
cast-->1
cast-->2
cast failed -->Specified cast is not valid.
```

Thankfully, if this is not what we want, we could use the alternative extension method *OfType<T>()*.

```
var objects = new Subject<object>();
objects.OfType<int>().Dump("OfType");
objects.OnNext(1);
objects.OnNext(2);
objects.OnNext("3");//Ignored
objects.OnNext(4);
objects.OnCompleted();
```

Output:

```
OfType-->1
OfType-->2
OfType-->4
OfType completed
```

It is fair to say that while these are convenient methods to have, we could have created them with the operators we already know about.

```
//source.Cast<int>(); is equivalent to
source.Select(i=>(int)i);
//source.OfType<int>();
source.Where(i=>i is int).Select(i=>(int)i);
```

Timestamp and TimeInterval

As observable sequences are asynchronous it can be convenient to know timings for when elements are received. The *Timestamp* extension method is a handy convenience method that wraps elements of a sequence in a light weight *Timestamped<T>* structure. The *Timestamped<T>* type is a struct that exposes the value of the element it wraps, and the timestamp it was created with as a *DateTimeOffset*.

In this example we create a sequence of three values, one second apart, and then transform it to a time stamped sequence. The handy implementation of `ToString()` on *Timestamped<T>* gives us a readable output.

```
Observable.Interval(TimeSpan.FromSeconds(1))
        .Take(3)
        .Timestamp()
        .Dump("TimeStamp");
```

Output

```
TimeStamp-->0@01/01/2012 12:00:01 a.m. +00:00
TimeStamp-->1@01/01/2012 12:00:02 a.m. +00:00
TimeStamp-->2@01/01/2012 12:00:03 a.m. +00:00
TimeStamp completed
```

We can see that the values 0, 1 & 2 were each produced one second apart. An alternative to getting an absolute timestamp is to just get the interval since the last element. The *TimeInterval* extension method provides this. As per the *Timestamp* method, elements are wrapped in a light weight structure. This time the structure is the *TimeInterval<T>* type.

```
Observable.Interval(TimeSpan.FromSeconds(1))
        .Take(3)
        .TimeInterval()
        .Dump("TimeInterval");
```

Output:

```
TimeInterval-->0@00:00:01.0180000
TimeInterval-->1@00:00:01.0010000
TimeInterval-->2@00:00:00.9980000
TimeInterval completed
```

As you can see from the output, the timings are not exactly one second but are pretty close.

Materialize and Dematerialize

The *Timestamp* and *TimeInterval* transform operators can prove useful for logging and debugging sequences, so too can the *Materialize* operator. *Materialize* transitions a sequence into a metadata representation of the sequence, taking an *IObservable<T>* to an *IObservable<Notification<T>>*. The *Notification* type provides meta data for the events of the sequence.

If we materialize a sequence, we can see the wrapped values being returned.

```
Observable.Range(1, 3)
        .Materialize()
        .Dump("Materialize");
```

Output:

```
Materialize-->OnNext(1)
Materialize-->OnNext(2)
Materialize-->OnNext(3)
Materialize-->OnCompleted()
Materialize completed
```

Note that when the source sequence completes, the materialized sequence produces an 'OnCompleted' notification value and then completes. *Notification<T>* is an abstract class with three implementations:

- OnNextNotification
- OnErrorNotification
- OnCompletedNotification

Notification<T> exposes four public properties to help you discover it: `Kind`, `HasValue`, `Value` and `Exception`. Obviously only *OnNextNotification* will return true for `HasValue` and have a useful implementation of `Value`. It should also be obvious that `OnErrorNotification` is the only implementation that will have a value for `Exception`. The `Kind` property returns an `enum` which should allow you to know which methods are appropriate to use.

```
public enum NotificationKind
{
        OnNext,
        OnError,
        OnCompleted,
}
```

In this next example we produce a faulted sequence. Note that the final value of the materialized sequence is an *OnErrorNotification*. Also that the materialized sequence does not error, it completes successfully.

```
var source = new Subject<int>();
source.Materialize()
        .Dump("Materialize");
source.OnNext(1);
source.OnNext(2);
source.OnNext(3);
source.OnError(new Exception("Fail?"));
```

Output:

```
Materialize-->OnNext(1)
Materialize-->OnNext(2)
Materialize-->OnNext(3)
Materialize-->OnError(System.Exception)
Materialize completed
```

Materializing a sequence can be very handy for performing analysis or logging of a sequence. You can unwrap a materialized sequence by applying the *Dematerialize* extension method. The *Dematerialize* will only work on *IObservable<Notification<TSource>>*.

SelectMany

Of the transformation operators above, we can see that *Select* is the most useful. It allows very broad flexibility in its transformation output and can even be used to reproduce some of the other transformation operators. The *SelectMany* operator however is even more powerful. In LINQ and therefore Rx, the *bind* method is *SelectMany*. Most other transformation operators can be built with *SelectMany*. Considering this, it is a shame to think that *SelectMany* may be one of the most misunderstood methods in LINQ.

In my personal discovery of Rx, I struggled to grasp the *SelectMany* extension method. One of my colleagues helped me understand *SelectMany* better by suggesting I think of it as “from one, select many”. An even better definition is “From one, select zero or more”. If we look at the signature for *SelectMany* we see that it takes a source sequence and a function as its parameters.

```
IObservable<TResult> SelectMany<TSource, TResult>(
        this IObservable<TSource> source,
        Func<TSource, IObservable<TResult>> selector)
```

The `selector` parameter is a function that takes a single value of `T` and returns a sequence. Note that the sequence the `selector` returns does not have to be of the same type as the `source`. Finally, the *SelectMany* return type is the same as the `selector` return type.

This method is very important to understand if you wish to work with Rx effectively, so let's step through this slowly. It is also important to note its subtle differences to *IEnumerable<T>*'s *SelectMany* operator, which we will look at soon.

Our first example will take a sequence with the single value '3' in it. The selector function we provide will produce a further sequence of numbers. This result sequence will be a range of numbers from 1 to the value provided i.e. 3. So we take the sequence [3] and return the sequence [1,2,3] from our `selector` function.

```
Observable.Return(3)
        .SelectMany(i => Observable.Range(1, i))
        .Dump("SelectMany");
```

Output:

```
SelectMany-->1
SelectMany-->2
SelectMany-->3
SelectMany completed
```

If we modify our source to be a sequence of [1,2,3] like this...

```
Observable.Range(1,3)
        .SelectMany(i => Observable.Range(1, i))
        .Dump("SelectMany");
```

...we will now get an output with the result of each sequence ([1], [1,2] and [1,2,3]) flattened to produce [1,1,2,1,2,3].

```
SelectMany-->1
SelectMany-->1
SelectMany-->2
SelectMany-->1
SelectMany-->2
SelectMany-->3
SelectMany completed
```

This last example better illustrates how *SelectMany* can take a *single* value and expand it to many values. When we then apply this to a *sequence* of values, the result is each of the child sequences combined to produce the final sequence. In both examples, we have returned a sequence that is the same type as the source. This is not a restriction however, so in this next example we return a different type. We will reuse the *Select* example of transforming an integer to an ASCII character. To do this, the `selector` function just returns a char sequence with a single value.

```
Func<int, char> letter = i => (char)(i + 64);
Observable.Return(1)
        .SelectMany(i => Observable.Return(letter(i)));
        .Dump("SelectMany");
```

So with the input of [1] we return a sequence of [A].

```
SelectMany-->A
SelectMany completed
```

Extending the source sequence to have many values, will give us a result with many values.

```
Func<int, char> letter = i => (char)(i + 64);
Observable.Range(1,3)
        .SelectMany(i => Observable.Return(letter(i)))
        .Dump("SelectMany");
```

Now the input of [1,2,3] produces [[A], [B], [C]] which is flattened to just [A,B,C].

```
SelectMany-->A
SelectMany-->B
SelectMany-->C
```

Note that we have effectively recreated the *Select* operator.

The last example maps a number to a letter. As there are only 26 letters, it would be nice to ignore values greater than 26. This is easy to do. While we must return a sequence for each element of the source, there aren't any rules that prevent it from being an empty sequence. In this case if the element value is a number outside of the range 1-26 we return an empty sequence.

```
Func<int, char> letter = i => (char)(i + 64);
Observable.Range(1, 30)
        .SelectMany(
        i =>
        {
        if (0 < i && i < 27)
        {
        return Observable.Return(letter(i));
        }
        else
        {
        return Observable.Empty<char>();
        }
        })
        .Dump("SelectMany");
```

Output:

```
A
B
C
...
X
Y
Z
Completed
```

To be clear, for the source sequence [1..30], the value 1 produced a sequence [A], the value 2 produced a sequence [B] and so on until value 26 produced a sequence [Z]. When the source produced value 27, the `selector` function returned the empty sequence []. Values 28, 29 and 30 also produced empty sequences. Once all the sequences from the calls to the selector had been fattened to produce the final result, we end up with the sequence [A..Z].

Now that we have covered the third of our three higher order functions, let us take time to reflect on some of the methods we have already learnt. First we can consider the *Where* extension method. We first looked at this method in the chapter on Reducing a sequence. While this method does reduce a sequence, it is not a fit for a functional *fold* as the result is still a sequence. Taking this into account, we find that *Where* is actually a fit for *bind*. As an exercise, try to write your own extension method version of *Where* using the *SelectMany* operator. Review the last example for some help...

An example of a *Where* extension method written using *SelectMany*:

```
public static IObservable<T> Where<T>(this IObservable<T> source, Func<T, bool> predicate)
{
        return source.SelectMany(
        item =>
        {
        if (predicate(item))
        {
        return Observable.Return(item);
        }
        else
        {
        return Observable.Empty<T>();
        }
        });
}
```

Now that we know we can use *SelectMany* to produce *Where*, it should be a natural progression for you the reader to be able to extend this to reproduce other filters like *Skip* and *Take*.

As another exercise, try to write your own version of the *Select* extension method using *SelectMany*. Refer to our example where we use *SelectMany* to convert `int` values into `char` values if you need some help...

An example of a *Select* extension method written using *SelectMany*:

```
public static IObservable<TResult> MySelect<TSource, TResult>(
        this IObservable<TSource> source,
        Func<TSource, TResult> selector)
{
        return source.SelectMany(value => Observable.Return(selector(value)));
}
```

IEnumerable<T> vs. IObservable<T> SelectMany

It is worth noting the difference between the implementations of *IEnumerable<T>SelectMany* and *IObservable<T>SelectMany*. Consider that *IEnumerable<T>* sequences are pull based and blocking. This means that when an *IEnumerable<T>* is processed with a *SelectMany* it will pass one item at a time to the `selector` function and wait until it has processed all of the values from the `selector` before requesting (pulling) the next value from the source.

Consider an *IEnumerable<T>* source sequence of [1,2,3]. If we process that with a *SelectMany* operator that returns a sequence of [x*10, (x*10)+1, (x*10)+2], we would get the [[10,11,12], [20,21,22], [30,31,32]].

```
private IEnumerable<int> GetSubValues(int offset)
{
        yield return offset * 10;
        yield return (offset * 10) + 1;
        yield return (offset * 10) + 2;
}
```

We then apply the `GetSubValues` method with the following code:

```
var enumerableSource = new [] {1, 2, 3};
var enumerableResult = enumerableSource.SelectMany(GetSubValues);
foreach (var value in enumerableResult)
{
        Console.WriteLine(value);
}
```

The resulting child sequences are flattened into [10,11,12,20,21,22,30,31,32].

```
10
11
12
20
21
22
30
31
32
```

The difference with *IObservable<T>* sequences is that the call to the *SelectMany*'s `selector` function is not blocking and the result sequence can produce values over time. This means that subsequent 'child' sequences can overlap. Let us consider again a sequence of [1,2,3], but this time values are produced three second apart. The `selector` function will also produce sequence of [x*10, (x*10)+1, (x*10)+2] as per the example above, however these values will be four seconds apart.

To visualize this kind of asynchronous data we need to represent space and time.

Visualizing sequences

Let's divert quickly and talk about a technique we will use to help communicate the concepts relating to sequences. Marble diagrams are a way of visualizing sequences. Marble diagrams are great for sharing Rx concepts and describing composition of sequences. When using marble diagrams there are only a few things you need to know

1. a sequence is represented by a horizontal line
2. time moves to the right (i.e. things on the left happened before things on the right)
3. notifications are represented by symbols:
    1. '0' for OnNext
    2. 'X' for an OnError
    3. '|' for OnCompleted
4. many concurrent sequences can be visualized by creating rows of sequences

This is a sample of a sequence of three values that completes:

```
--0--0--0-|
```

This is a sample of a sequence of four values then an error:

```
--0--0--0--0--X
```

Now going back to our *SelectMany* example, we can visualize our input sequence by using values in instead of the 0 marker. This is the marble diagram representation of the sequence [1,2,3] spaced three seconds apart (note each character represents one second).

```
--1--2--3|
```

Now we can leverage the power of marble diagrams by introducing the concept of time and space. Here we see the visualization of the sequence produced by the first value 1 which gives us the sequence [10,11,12]. These values were spaced four seconds apart, but the initial value is produce immediately.

```
1---1---1|

0   1   2|
```

As the values are double digit they cover two rows, so the value of 10 is not confused with the value 1 immediately followed by the value 0. We add a row for each sequence produced by the `selector` function.

```
--1--2--3|
  1---1---1|
  0   1   2|
     2---2---2|
     0   1   2|
        3---3---3|
        0   1   2|
```

Now that we can visualize the source sequence and its child sequences, we should be able to deduce the expected output of the *SelectMany* operator. To create a result row for our marble diagram, we simple allow the values from each child sequence to 'fall' into the new result row.

```
--1--2--3|
  1---1---1|
  0   1   2|
     2---2---2|
```

```
     0   1   2|

        3---3---3|

        0   1   2|

--1--21-321-32--3|

  0  01 012 12  2|
```

If we take this exercise and now apply it to code, we can validate our marble diagram. First our method that will produce our child sequences:

```
private IObservable<long> GetSubValues(long offset)
{
        //Produce values [x*10, (x*10)+1, (x*10)+2] 4 seconds apart, but starting immediately.
        return Observable.Timer(TimeSpan.Zero, TimeSpan.FromSeconds(4))
        .Select(x => (offset*10) + x)
        .Take(3);
}
```

This is the code that takes the source sequence to produce our final output:

```
// Values [1,2,3] 3 seconds apart.
Observable.Interval(TimeSpan.FromSeconds(3))
        .Select(i => i + 1) //Values start at 0, so add 1.
        .Take(3)            //We only want 3 values
        .SelectMany(GetSubValues) //project into child sequences
        .Dump("SelectMany");
```

The output produced matches our expectations from the marble diagram.

```
SelectMany-->10
SelectMany-->20
SelectMany-->11
SelectMany-->30
SelectMany-->21
SelectMany-->12
SelectMany-->31
SelectMany-->22
SelectMany-->32
SelectMany completed
```

We have previously looked at the *Select* operator when it is used in Query Comprehension Syntax, so it is worth noting how you use the *SelectMany* operator. The *Select* extension method maps quite obviously to query comprehension syntax, *SelectMany* is not so obvious. As we saw in the earlier example, the simple implementation of just suing select is as follows:

```
var query = from i in Observable.Range(1, 5)
        select i;
```

If we wanted to add a simple `where` clause we can do so like this:

```
var query = from i in Observable.Range(1, 5)
        where i%2==0
        select i;
```

To add a *SelectMany* to the query, we actually add an extra `from` clause.

```
var query = from i in Observable.Range(1, 5)
        where i%2==0
        from j in GetSubValues(i)
        select j;
//Equivalent to
var query = Observable.Range(1, 5)
        .Where(i=>i%2==0)
        .SelectMany(GetSubValues);
```

An advantage of using the query comprehension syntax is that you can easily access other variables in the scope of the query. In this example we select into an anon type both the value from the source and the child value.

```
var query = from i in Observable.Range(1, 5)
        where i%2==0
        from j in GetSubValues(i)
        select new {i, j};
query.Dump("SelectMany");
```

Output

```
SelectMany-->{ i = 2, j = 20 }
SelectMany-->{ i = 4, j = 40 }
SelectMany-->{ i = 2, j = 21 }
SelectMany-->{ i = 4, j = 41 }
SelectMany-->{ i = 2, j = 22 }
SelectMany-->{ i = 4, j = 42 }
SelectMany completed
```

This brings us to a close on Part 2. The key takeaways from this were to allow you the reader to understand a key principal to Rx: functional composition. As we move through Part 2, examples became progressively more complex. We were leveraging the power of LINQ to chain extension methods together to compose complex queries.

We didn't try to tackle all of the operators at once, we approached them in groups.

- Creation
- Reduction
- Inspection
- Aggregation
- Transformation

On deeper analysis of the operators we find that most of the operators are actually specialization of the higher order functional concepts. We named them the ABC's of functional programming:

- Anamorphism, aka:
  - Ana
  - Unfold
  - Generate
- Bind, aka:
  - Map
  - SelectMany
  - Projection
  - Transform
- Catamorphism, aka:
  - Cata
  - Fold
  - Reduce
  - Accumulate
  - Inject

Now you should feel that you have a strong understanding of how a sequence can be manipulated. What we have learnt up to this point however can all largely be applied to *IEnumerable* sequences too. Rx can be much more complex than what many people will have dealt with in *IEnumerable* world, as we have seen with the *SelectMany* operator. In the next part of the book we will uncover features specific to the asynchronous nature of Rx. With the foundation we have built so far we should be able to tackle the far more challenging and interesting features of Rx.

PART 3 - Taming the sequence

In the third part to this book we will look the features that allow us to apply Rx to more than just sample code. When building production quality code we often need to be able to handle error scenarios, log workflow, retry in certain circumstances, dispose of resources and other real-life problems that are regularly excluded from examples and demos.

Part 3 of this book aims to equip you with the tools you need to be able to use Rx as more than just a toy. If you use Rx properly, you will find it pervasive in your code base. You should not shy away from this, just like you would not shy away from using the `foreach` syntax with *IEnumerable* types, or, the `using` syntax with *IDisposable* types. Understanding and embracing Rx will improve your code base by reducing it, by making it more declarative, by identifying and eliminating race conditions, and therefore making it more maintainable.

Maintenance of Rx code obviously requires Rx knowledge but this creates a "chicken and egg" problem. I choose to believe that Rx is here to stay. I believe this because it solves a targeted set of problems very well. It is also complimentary to other libraries and features such as TPL (Task Parallel Library) and the future `async`/`await` features of .NET 4.5. Considering this, if Rx improves our code base then we should embrace it!

Side effects

Non-functional requirements of production systems often demand high availability, quality monitoring features and low lead time for defect resolution. Logging, debugging, instrumentation and journaling are common non-functional requirements that developers need to consider for production ready systems. These artifacts could be considered side effects of the main business workflow. Side effects are a real life problem that code samples and how-to guides often ignore, however Rx provides tools to help.

In this chapter we will discuss the consequences of introducing side effects when working with an observable sequence. A function is considered to have a side effect if, in addition to any return value, it has some other observable effect. Generally the 'observable effect' is a modification of state. This observable effect could be

- modification of a variable with a wider scope than the function (i.e. global, static or perhaps an argument)
- I/O such as a read/write from a file or network
- updating a display

Issues with side effects

Functional programming in general tries to avoid creating any side effects. Functions with side effects, especially which modify state, require the programmer to understand more than just the inputs and outputs of the function. The surface area they are required to understand needs to now extend to the history and context of the state being modified. This can greatly increase the complexity of a function, and thus make it harder to correctly understand and maintain.

Side effects are not always accidental, nor are they always intentional. An easy way to reduce the accidental side effects is to reduce the surface area for change. The simple actions coders can take are to reduce the visibility or scope of state and to make what you can immutable. You can reduce the visibility of a variable by scoping it to a code block like a method. You can reduce visibility of class members by making them private or protected. By definition immutable data can't be modified so cannot exhibit side effects. These are sensible encapsulation rules that will dramatically improve the maintainability of your Rx code.

To provide a simple example of a query that has a side effect, we will try to output the index and value of the elements received by updating a variable (closure).

```
var letters = Observable.Range(0, 3)
        .Select(i => (char)(i + 65));
var index = -1;
var result = letters.Select(
        c =>
        {
        index++;
        return c;
        });
result.Subscribe(
        c => Console.WriteLine("Received {0} at index {1}", c, index),
        () => Console.WriteLine("completed"));
```

Output:

```
Received A at index 0
Received B at index 1
Received C at index 2
completed
```

While this seems harmless enough, imagine if another person sees this code and understands it to be the pattern the team is using. They in turn adopt this style themselves. For the sake of the example, we will add a duplicate subscription to our previous example.

```
var letters = Observable.Range(0, 3)
        .Select(i => (char)(i + 65));
var index = -1;
var result = letters.Select(
        c =>
        {
        index++;
        return c;
        });
result.Subscribe(
        c => Console.WriteLine("Received {0} at index {1}", c, index),
        () => Console.WriteLine("completed"));
result.Subscribe(
        c => Console.WriteLine("Also received {0} at index {1}", c, index),
        () => Console.WriteLine("2nd completed"));
```

Output

```
Received A at index 0
Received B at index 1
Received C at index 2
completed
Also received A at index 3
Also received B at index 4
Also received C at index 5
2nd completed
```

Now the second person's output is clearly nonsense. They will be expecting index values to be 0, 1

and 2 but get 3, 4 and 5 instead. I have seen far more sinister versions of side effects in code bases. The nasty ones often modify state that is a Boolean value e.g. `hasValues`, `isStreaming` etc. We will see in a later chapter far better ways of controlling workflow with observable sequences than using shared state.

In addition to creating potentially unpredictable results in existing software, programs that exhibit side effects are far more difficult to test and maintain. Future refactoring, enhancements or other maintenance on programs that exhibits side effects are far more likely to be brittle. This is especially so in asynchronous or concurrent software.

Composing data in a pipeline

The preferred way of capturing state is to introduce it to the pipeline. Ideally, we want each part of the pipeline to be independent and deterministic. That is, each function that makes up the pipeline should have its inputs and output as its only state. To correct our example we could enrich the data in the pipeline so that there is no shared state. This would be a great example where we could use the *Select* overload that exposes the index.

```
var source = Observable.Range(0, 3);
var result = source.Select(
        (idx, value) => new
        {
        Index = idx,
        Letter = (char) (value + 65)
        });
result.Subscribe(
        x => Console.WriteLine("Received {0} at index {1}", x.Letter, x.Index),
        () => Console.WriteLine("completed"));
result.Subscribe(
        x => Console.WriteLine("Also received {0} at index {1}", x.Letter, x.Index),
        () => Console.WriteLine("2nd completed"));
```

Output:

```
Received A at index 0
Received B at index 1
Received C at index 2
completed
Also received A at index 0
Also received B at index 1
Also received C at index 2
2nd completed
```

Thinking outside of the box, we could also use other features like *Scan* to achieve similar results. Here is an example.

```
var result = source.Scan(
        new
        {
        Index = -1,
        Letter = new char()
        },
        (acc, value) => new
        {
        Index = acc.Index + 1,
        Letter = (char)(value + 65)
        });
```

The key here is to isolate the state, and reduce or remove any side effects like mutating state.

Do

We should aim to avoid side effects, but in some cases it is unavoidable. The *Do* extension method allows you to inject side effect behavior. The signature of the *Do* extension method looks very much like the *Select* method;

- They both have various overloads to cater for combinations of *OnNext*, *OnError* and *OnCompleted* handlers
- They both return and take an observable sequence

```
// Invokes an action with side effecting behavior for each element in the observable
//  sequence.
public static IObservable<TSource> Do<TSource>(
        this IObservable<TSource> source,
        Action<TSource> onNext)
{...}
// Invokes an action with side effecting behavior for each element in the observable
//  sequence and invokes an action with side effecting behavior upon graceful termination
//  of the observable sequence.
public static IObservable<TSource> Do<TSource>(
        this IObservable<TSource> source,
        Action<TSource> onNext,
        Action onCompleted)
{...}
// Invokes an action with side effecting behavior for each element in the observable
//  sequence and invokes an action with side effecting behavior upon exceptional
//  termination of the observable sequence.
public static IObservable<TSource> Do<TSource>(
        this IObservable<TSource> source,
        Action<TSource> onNext,
        Action<Exception> onError)
{...}
// Invokes an action with side effecting behavior for each element in the observable
//  sequence and invokes an action with side effecting behavior upon graceful or
//  exceptional termination of the observable sequence.
public static IObservable<TSource> Do<TSource>(
        this IObservable<TSource> source,
        Action<TSource> onNext,
        Action<Exception> onError,
        Action onCompleted)
{...}
// Invokes the observer's methods for their side effects.
public static IObservable<TSource> Do<TSource>(
        this IObservable<TSource> source,
        IObserver<TSource> observer)
{...}
```

The *Select* overloads take *Func* arguments for their *OnNext* handlers and also provide the ability to return an observable sequence that is a different type to the source. In contrast, the *Do* methods only take an *Action<T>* for the *OnNext* handler, and therefore can only return a sequence that is the same type as the source. As each of the arguments that can be passed to the *Do* overloads are actions, they implicitly cause side effects.

For the next example, we first define the following methods for logging:

```
private static void Log(object onNextValue)
{
        Console.WriteLine("Logging OnNext({0}) @ {1}", onNextValue, DateTime.Now);
}
private static void Log(Exception onErrorValue)
{
        Console.WriteLine("Logging OnError({0}) @ {1}", onErrorValue, DateTime.Now);
}
private static void Log()
{
        Console.WriteLine("Logging OnCompleted()@ {0}", DateTime.Now);
}
```

This code can use *Do* to introduce some logging using the methods from above.

```
var source = Observable
        .Interval(TimeSpan.FromSeconds(1))
        .Take(3);
var result = source.Do(
        i => Log(i),
        ex => Log(ex),
        () => Log());
result.Subscribe(
        Console.WriteLine,
        () => Console.WriteLine("completed"));
```

Output:

```
Logging OnNext(0) @ 01/01/2012 12:00:00
0
Logging OnNext(1) @ 01/01/2012 12:00:01
1
Logging OnNext(2) @ 01/01/2012 12:00:02
2
Logging OnCompleted() @ 01/01/2012 12:00:02
completed
```

Note that because the *Do* is earlier in the query chain than the *Subscribe*, it will receive the values first and therefore write to the console first. I like to think of the *Do* method as a wire tap to a sequence. It gives you the ability to listen in on the sequence, without the ability to modify it.

The most common acceptable side effect I see in Rx is the need to log. The signature of *Do* allows you to inject it into a query chain. This allows us to add logging into our sequence and retain encapsulation. When a repository, service agent or provider exposes an observable sequence, they have the ability to add their side effects (e.g. logging) to the sequence before exposing it publicly. Consumers can then append operators to the query (e.g. *Where*, *SelectMany*) and this will not affect the logging of the provider.

Consider the method below. It produces numbers but also logs what it produces (to the console for simplicity). To the consuming code the logging is transparent.

```
private static IObservable<long> GetNumbers()
{
        return Observable.Interval(TimeSpan.FromMilliseconds(250))
        .Do(i => Console.WriteLine("pushing {0} from GetNumbers", i));
}
```

We then call it with this code.

```
var source = GetNumbers();
var result = source.Where(i => i%3 == 0)
        .Take(3)
        .Select(i => (char) (i + 65));
result.Subscribe(
        Console.WriteLine,
        () => Console.WriteLine("completed"));
```

Output:

```
pushing 0 from GetNumbers
A
pushing 1 from GetNumbers
pushing 2 from GetNumbers
pushing 3 from GetNumbers
D
pushing 4 from GetNumbers
pushing 5 from GetNumbers
pushing 6 from GetNumbers
G
completed
```

This example shows how producers or intermediaries can apply logging to the sequence regardless of what the end consumer does.

One overload to *Do* allows you to pass in an *IObserver<T>*. In this overload, each of the `OnNext`, `OnError` and `OnCompleted` methods are passed to the other *Do* overload as each of the actions to perform.

Applying a side effect adds complexity to a query. If side effects are a necessary evil, then being explicit will help your fellow coder understand your intentions. Using the *Do* method is the favored approach to doing so. This may seem trivial, but given the inherent complexity of a business domain mixed with asynchrony and concurrency, developers don't need the added complication of side effects hidden in a *Subscribe* or *Select* operator.

Encapsulating with AsObservable

Poor encapsulation is a way developers can leave the door open for unintended side effects. Here is a handful of scenarios where carelessness leads to leaky abstractions. Our first example may seem harmless at a glance, but has numerous problems.

```
public class UltraLeakyLetterRepo
{
        public ReplaySubject<string> Letters { get; set; }
        public UltraLeakyLetterRepo()
        {
        Letters = new ReplaySubject<string>();
        Letters.OnNext("A");
        Letters.OnNext("B");
        Letters.OnNext("C");
        }
}
```

In this example we expose our observable sequence as a property. The first problem here is that it is a settable property. Consumers could change the entire subject out if they wanted. This would be a very poor experience for other consumers of this class. If we make some simple changes we can make a class that seems safe enough.

```
public class LeakyLetterRepo
{
        private readonly ReplaySubject<string> _letters;
        public LeakyLetterRepo()
        {
        _letters = new ReplaySubject<string>();
        _letters.OnNext("A");
        _letters.OnNext("B");
        _letters.OnNext("C");
        }
        public ReplaySubject<string> Letters
        {
        get { return _letters; }
        }
}
```

Now the `Letters` property only has a getter and is backed by a read-only field. This is much better. Keen readers will note that the `Letters` property returns a *ReplaySubject<string>*. This is poor encapsulation, as consumers could call `OnNext`/`OnError`/`OnCompleted`. To close off that loophole we can simply make the return type an *IObservable<string>*.

```
public IObservable<string> Letters
{
        get { return _letters; }
}
```

The class now *looks* much better. The improvement, however, is only cosmetic. There is still nothing preventing consumers from casting the result back to an *ISubject<string>* and then calling whatever methods they like. In this example we see external code pushing their values into the sequence.

```
var repo = new ObscuredLeakinessLetterRepo();
var good = repo.GetLetters();
var evil = repo.GetLetters();
good.Subscribe(
        Console.WriteLine);
//Be naughty
var asSubject = evil as ISubject<string>;
if (asSubject != null)
{
        //So naughty, 1 is not a letter!
        asSubject.OnNext("1");
}
else
{
        Console.WriteLine("could not sabotage");
}
```

Output:

```
A
B
C
1
```

The fix to this problem is quite simple. By applying the *AsObservable* extension method, the `_letters`

field will be wrapped in a type that only implements *IObservable<T>*.

```
public IObservable<string> GetLetters()
{
        return _letters.AsObservable();
}
```

Output:

```
A
B
C
could not sabotage
```

While I have used words like 'evil' and 'sabotage' in these examples, it is more often than not an oversight rather than malicious intent that causes problems. The failing falls first on the programmer who designed the leaky class. Designing interfaces is hard, but we should do our best to help consumers of our code fall into the pit of success by giving them discoverable and consistent types. Types become more discoverable if we reduce their surface area to expose only the features we intend our consumers to use. In this example we reduced the type's surface area. We did so by removing the property setter and returning a simpler type via the *AsObservable* method.

Mutable elements cannot be protected

While the *AsObservable* method can encapsulate your sequence, you should still be aware that it gives no protection against mutable elements. Consider what consumers of a sequence of this class could do:

```
public class Account
{
        public int Id { get; set; }
        public string Name { get; set; }
}
```

Here is a quick example of the kind of mess we can make if we choose to modify elements in a sequence.

```
var source = new Subject<Account>();
//Evil code. It modifies the Account object.
source.Subscribe(account => account.Name = "Garbage");
//unassuming well behaved code
source.Subscribe(
        account=>Console.WriteLine("{0} {1}", account.Id, account.Name),
        ()=>Console.WriteLine("completed"));
source.OnNext(new Account {Id = 1, Name = "Microsoft"});
source.OnNext(new Account {Id = 2, Name = "Google"});
source.OnNext(new Account {Id = 3, Name = "IBM"});
source.OnCompleted();
```

Output:

```
1 Garbage
2 Garbage
3 Garbage
completed
```

Here the second consumer was expecting to get 'Microsoft', 'Google' and 'IBM' but received just 'Garbage'.

Observable sequences will be perceived to be a sequence of resolved events: things that have happened as a statement of fact. This implies two things: first, each element represents a snapshot of state at the time of publication, secondly, the information emanates from a trustworthy source. We want to eliminate the possibility of tampering. Ideally the type `T` will be immutable, solving both of these problems. This way, consumers of the sequence can be assured that the data they get is the data that the source produced. Not being able to mutate elements may seem limiting as a consumer, but these needs are best met via the Transformation operators which provide better encapsulation.

Side effects should be avoided where possible. Any combination of concurrency with shared state will commonly demand the need for complex locking, deep understanding of CPU architectures and how they work with the locking and optimization features of the language you use. The simple and preferred approach is to avoid shared state, favor immutable data types and utilize query composition and transformation. Hiding side effects into *Where* or *Select* clauses can make for very confusing code. If a side effect is required, then the *Do* method expresses intent that you are creating a side effect by being explicit.

Leaving the monad

An observable sequence is a useful construct, especially when we have the power of LINQ to compose complex queries over it. Even though we recognize the benefits of the observable sequence, sometimes it is required to leave the *IObservable<T>* paradigm for another paradigm, maybe to enable you to integrate with an existing API (i.e. use events or *Task<T>*). You might leave the observable paradigm if you find it easier for testing, or it may simply be easier for you to learn Rx by moving between an observable paradigm and a more familiar one.

What is a monad

We have casually referred to the term *monad* earlier in the book, but to most it will be a very foreign term. I am going to try to avoid overcomplicating what a monad is, but give enough of an explanation to help us out with our next category of methods. The full definition of a monad is quite abstract. Many others have tried to provide their definition of a monad using all sorts of metaphors from astronauts to Alice in Wonderland. Many of the tutorials for monadic programming use Haskell for the code examples which can add to the confusion. For us, a monad is effectively a programming structure that represents computations. Compare this to other programming structures:

Data structure
: Purely state e.g. a List, a Tree or a Tuple

Contract
: Contract definition or abstract functionality e.g. an interface or abstract class

Object-Orientated structure
: State and behavior together

Generally a monadic structure allows you to chain together operators to produce a pipeline, just as we do with our extension methods.

Monads are a kind of abstract data type constructor that encapsulate program logic instead of data in the domain model.

This neat definition of a monad lifted from Wikipedia allows us to start viewing sequences as monads; the abstract data type in this case is the *IObservable<T>* type. When we use an observable sequence, we compose functions onto the abstract data type (the *IObservable<T>*) to create a query. This query becomes our encapsulated programming logic.

The use of monads to define control flows is particularly useful when dealing with typically troublesome areas of programming such as IO, concurrency and exceptions. This just happens to be some of Rx's strong points!

Why leave the monad?

There is a variety of reasons you may want to consume an observable sequence in a different paradigm. Libraries that need to expose functionality externally may be required to present it as events or as *Task* instances. In demonstration and sample code you may prefer to use blocking methods to limit the number of asynchronous moving parts. This may help make the learning curve to Rx a little less steep!

In production code, it is rarely advised to 'break the monad', especially moving from an observable sequence to blocking methods. Switching between asynchronous and synchronous paradigms should be done with caution, as this is a common root cause for concurrency problems such as deadlock and scalability issues.

In this chapter, we will look at the methods in Rx which allow you to leave the *IObservable<T>* monad.

ForEach

The *ForEach* method provides a way to process elements as they are received. The key difference between *ForEach* and *Subscribe* is that *ForEach* will block the current thread until the sequence completes.

```
var source = Observable.Interval(TimeSpan.FromSeconds(1))
        .Take(5);
source.ForEach(i => Console.WriteLine("received {0} @ {1}", i, DateTime.Now));
Console.WriteLine("completed @ {0}", DateTime.Now);
```

Output:

```
received 0 @ 01/01/2012 12:00:01 a.m.
received 1 @ 01/01/2012 12:00:02 a.m.
received 2 @ 01/01/2012 12:00:03 a.m.
received 3 @ 01/01/2012 12:00:04 a.m.
received 4 @ 01/01/2012 12:00:05 a.m.
completed @ 01/01/2012 12:00:05 a.m.
```

Note that the completed line is last, as you would expect. To be clear, you can get similar functionality from the *Subscribe* extension method, but the *Subscribe* method will not block. So if we substitute the call to *ForEach* with a call to *Subscribe*, we will see the completed line happen first.

```
var source = Observable.Interval(TimeSpan.FromSeconds(1))
        .Take(5);
source.Subscribe(i => Console.WriteLine("received {0} @ {1}", i, DateTime.Now));
Console.WriteLine("completed @ {0}", DateTime.Now);
```

Output:

```
completed @ 01/01/2012 12:00:00 a.m.
received 0 @ 01/01/2012 12:00:01 a.m.
received 1 @ 01/01/2012 12:00:02 a.m.
received 2 @ 01/01/2012 12:00:03 a.m.
received 3 @ 01/01/2012 12:00:04 a.m.
received 4 @ 01/01/2012 12:00:05 a.m.
```

Unlike the *Subscribe* extension method, *ForEach* has only the one overload; the one that take an *Action<T>* as its single argument. In contrast, previous (pre-release) versions of Rx, the *ForEach* method had most of the same overloads as *Subscribe*. Those overloads of *ForEach* have been deprecated, and I think rightly so. There is no need to have an `OnCompleted` handler in a synchronous call, it is unnecessary. You can just place the call immediately after the *ForEach* call as we have done above. Also, the `OnError` handler can now be replaced with standard Structured Exception Handling like you would use for any other synchronous code, with a `try`/`catch` block. This also gives symmetry to the *ForEach* instance method on the *List<T>* type.

```
var source = Observable.Throw<int>(new Exception("Fail"));
try
{
        source.ForEach(Console.WriteLine);
}
catch (Exception ex)
{
        Console.WriteLine("error @ {0} with {1}", DateTime.Now, ex.Message);
}
finally
{
        Console.WriteLine("completed @ {0}", DateTime.Now);
}
```

Output:

```
error @ 01/01/2012 12:00:00 a.m. with Fail
completed @ 01/01/2012 12:00:00 a.m.
```

The *ForEach* method, like its other blocking friends (*First* and *Last* etc.), should be used with care. I would leave the *ForEach* method for spikes, tests and demo code only. We will discuss the problems with introducing blocking calls when we look at concurrency.

ToEnumerable

An alternative way to switch out of the *IObservable<T>* is to call the *ToEnumerable* extension method. As a simple example:

```
var period = TimeSpan.FromMilliseconds(200);
var source = Observable.Timer(TimeSpan.Zero, period)
        .Take(5);
var result = source.ToEnumerable();
foreach (var value in result)
{
        Console.WriteLine(value);
}
Console.WriteLine("done");
```

Output:

```
0
1
2
3
4
done
```

The source observable sequence will be subscribed to when you start to enumerate the sequence (i.e. lazily). In contrast to the *ForEach* extension method, using the *ToEnumerable* method means you are only blocked when you try to move to the next element and it is not available. Also, if the sequence produces values faster than you consume them, they will be cached for you.

To cater for errors, you can wrap your `foreach` loop in a `try`/`catch` as you do with any other enumerable sequence:

```
try
{
        foreach (var value in result)
        {
        Console.WriteLine(value);
        }
}
catch (Exception e)
{
        Console.WriteLine(e.Message);
}
```

As you are moving from a push to a pull model (non-blocking to blocking), the standard warning applies.

To a single collection

To avoid having to oscillate between push and pull, you can use one of the next four methods to get the entire list back in a single notification. They all have the same semantics, but just produce the data in a different format. They are similar to their corresponding *IEnumerable<T>* operators, but the return values differ in order to retain asynchronous behavior.

ToArray and ToList

Both *ToArray* and *ToList* take an observable sequence and package it into an array or an instance of *List<T>* respectively. Once the observable sequence completes, the array or list will be pushed as the single value of the result sequence.

```
var period = TimeSpan.FromMilliseconds(200);
var source = Observable.Timer(TimeSpan.Zero, period).Take(5);
var result = source.ToArray();
result.Subscribe(
        arr => {
        Console.WriteLine("Received array");
        foreach (var value in arr)
        {
        Console.WriteLine(value);
        }
        },
        () => Console.WriteLine("Completed")
);
Console.WriteLine("Subscribed");
```

Output:

```
Subscribed
Received array
0
1
2
3
4
Completed
```

As these methods still return observable sequences we can use our `OnError` handler for errors. Note that the source sequence is packaged to a single notification; you either get the whole sequence **or** the error. If the source produces values and then errors, you will not receive any of those values. All four operators (*ToArray*, *ToList*, *ToDictionary* and *ToLookup*) handle errors like this.

ToDictionary and ToLookup

As an alternative to arrays and lists, Rx can package an observable sequence into a dictionary or lookup with the *ToDictionary* and *ToLookup* methods. Both methods have the same semantics as the *ToArray* and *ToList* methods, as they return a sequence with a single value and have the same error handling features.

The *ToDictionary* extension method overloads:

```
// Creates a dictionary from an observable sequence according to a specified key selector
// function, a comparer, and an element selector function.
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)
{...}
// Creates a dictionary from an observable sequence according to a specified key selector
// function, and an element selector function.
public static IObservable<IDictionary<TKey, TElement>> ToDictionary<TSource, TKey, TElement>(
        this IObservable<TSource> source,
        Func<TSource, TKey> keySelector,
        Func<TSource, TElement> elementSelector)
{...}
// Creates a dictionary from an observable sequence according to a specified key selector
// function, and a comparer.
public static IObservable<IDictionary<TKey, TSource>> ToDictionary<TSource, TKey>(
        this IObservable<TSource> source,
        Func<TSource, TKey> keySelector,
```

```
        IEqualityComparer<TKey> comparer)
{...}
// Creates a dictionary from an observable sequence according to a specified key selector
// function.
public static IObservable<IDictionary<TKey, TSource>> ToDictionary<TSource, TKey>(
        this IObservable<TSource> source,
        Func<TSource, TKey> keySelector)
{...}
```

The *ToLookup* extension method overloads:

```
// Creates a lookup from an observable sequence according to a specified key selector
// function, a comparer, and an element selector function.
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)
{...}
// Creates a lookup from an observable sequence according to a specified key selector
// function, and a comparer.
public static IObservable<ILookup<TKey, TSource>> ToLookup<TSource, TKey>(
        this IObservable<TSource> source,
        Func<TSource, TKey> keySelector,
        IEqualityComparer<TKey> comparer)
{...}
// Creates a lookup from an observable sequence according to a specified key selector
// function, and an element selector function.
public static IObservable<ILookup<TKey, TElement>> ToLookup<TSource, TKey, TElement>(
        this IObservable<TSource> source,
        Func<TSource, TKey> keySelector,
        Func<TSource, TElement> elementSelector)
{...}
// Creates a lookup from an observable sequence according to a specified key selector
// function.
public static IObservable<ILookup<TKey, TSource>> ToLookup<TSource, TKey>(
        this IObservable<TSource> source,
        Func<TSource,
        TKey> keySelector)
{...}
```

Both *ToDictionary* and *ToLookup* require a function that can be applied each value to get its key. In addition, the *ToDictionary* method overloads mandate that all keys should be unique. If a duplicate key is found, it terminate the sequence with a *DuplicateKeyException*. On the other hand, the *ILookup<TKey, TElement>* is designed to have multiple values grouped by the key. If you have many values per key, then *ToLookup* is probably the better option.

ToTask

We have compared *AsyncSubject<T>* to *Task<T>* and even showed how to transition from a task to an observable sequence. The *ToTask* extension method will allow you to convert an observable sequence into a *Task<T>*. Like an *AsyncSubject<T>*, this method will ignore multiple values, only returning the last value.

```
// Returns a task that contains the last value of the observable sequence.
public static Task<TResult> ToTask<TResult>(
        this IObservable<TResult> observable)
{...}
// Returns a task that contains the last value of the observable sequence, with state to
//  use as the underlying task's AsyncState.
public static Task<TResult> ToTask<TResult>(
        this IObservable<TResult> observable,
        object state)
{...}
// Returns a task that contains the last value of the observable sequence. Requires a
//  cancellation token that can be used to cancel the task, causing unsubscription from
//  the observable sequence.
public static Task<TResult> ToTask<TResult>(
        this IObservable<TResult> observable,
        CancellationToken cancellationToken)
{...}
// Returns a task that contains the last value of the observable sequence, with state to
//  use as the underlying task's AsyncState. Requires a cancellation token that can be used
//  to cancel the task, causing unsubscription from the observable sequence.
public static Task<TResult> ToTask<TResult>(
        this IObservable<TResult> observable,
        CancellationToken cancellationToken,
        object state)
{...}
```

This is a simple example of how the *ToTask* operator can be used. Note, the *ToTask* method is in the `System.Reactive.Threading.Tasks` namespace.

```
var source = Observable.Interval(TimeSpan.FromSeconds(1))
        .Take(5);
var result = source.ToTask(); //Will arrive in 5 seconds.
Console.WriteLine(result.Result);
```

Output:

```
4
```

If the source sequence was to manifest error then the task would follow the error-handling semantics of tasks.

```
var source = Observable.Throw<long>(new Exception("Fail!"));
var result = source.ToTask();
try
{
        Console.WriteLine(result.Result);
}
catch (AggregateException e)
{
        Console.WriteLine(e.InnerException.Message);
}
```

Output:

```
Fail!
```

Once you have your task, you can of course engage in all the features of the TPL such as continuations.

ToEvent<T>

Just as you can use an event as the source for an observable sequence with *FromEventPattern*, you can also make your observable sequence look like a standard .NET event with the *ToEvent* extension methods.

```
// Exposes an observable sequence as an object with a .NET event.
public static IEventSource<unit> ToEvent(this IObservable<Unit> source)
{...}
// Exposes an observable sequence as an object with a .NET event.
public static IEventSource<TSource> ToEvent<TSource>(
        this IObservable<TSource> source)
{...}
// Exposes an observable sequence as an object with a .NET event.
public static IEventPatternSource<TEventArgs> ToEventPattern<TEventArgs>(
        this IObservable<EventPattern<TEventArgs>> source)
        where TEventArgs : EventArgs
{...}
```

The *ToEvent* method returns an *IEventSource<T>*, which will have a single event member on it: `OnNext`.

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

When we convert the observable sequence with the *ToEvent* method, we can just subscribe by providing an *Action<T>*, which we do here with a lambda.

```
var source = Observable.Interval(TimeSpan.FromSeconds(1))
        .Take(5);
var result = source.ToEvent();
result.OnNext += val => Console.WriteLine(val);
```

Output:

```
0
1
2
3
4
```

ToEventPattern

Note that this does not follow the standard pattern of events. Normally, when you subscribe to an event, you need to handle the `sender` and *EventArgs* parameters. In the example above, we just get the value. If you want to expose your sequence as an event that follows the standard pattern, you will need to use *ToEventPattern*.

The *ToEventPattern* will take an *IObservable<EventPattern<TEventArgs>>* and convert that into an *IEventPatternSource<TEventArgs>*. The public interface for these types is quite simple.

```
public class EventPattern<TEventArgs> : IEquatable<EventPattern<TEventArgs>>
        where TEventArgs : EventArgs
{
        public EventPattern(object sender, TEventArgs e)
        {
        this.Sender = sender;
        this.EventArgs = e;
        }
        public object Sender { get; private set; }
        public TEventArgs EventArgs { get; private set; }
        //...equality overloads
}
public interface IEventPatternSource<TEventArgs> where TEventArgs : EventArgs
{
        event EventHandler<TEventArgs> OnNext;
}
```

These look quite easy to work with. So if we create an *EventArgs* type and then apply a simple transform using *Select*, we can make a standard sequence fit the pattern.

The *EventArgs* type:

```
public class MyEventArgs : EventArgs
{
        private readonly long _value;
        public MyEventArgs(long value)
        {
        _value = value;
        }
        public long Value
        {
        get { return _value; }
        }
}
```

The transform:

```
var source = Observable.Interval(TimeSpan.FromSeconds(1))
        .Select(i => new EventPattern<MyEventArgs>(this, new MyEventArgs(i)));
```

Now that we have a sequence that is compatible, we can use the *ToEventPattern*, and in turn, a standard event handler.

```
var result = source.ToEventPattern();
result.OnNext += (sender, eventArgs) => Console.WriteLine(eventArgs.Value);
```

Now that we know how to get back into .NET events, let's take a break and remember why Rx is a better model.

- In C#, events have a curious interface. Some find the `+=` and `-=` operators an unnatural way to register a callback
- Events are difficult to compose
- Events do not offer the ability to be easily queried over time
- Events are a common cause of accidental memory leaks
- Events do not have a standard pattern for signaling completion
- Events provide almost no help for concurrency or multithreaded applications. For instance, raising an event on a separate thread requires you to do all of the plumbing

---

The set of methods we have looked at in this chapter complete the circle started in the Creating a Sequence chapter. We now have the means to enter and leave the observable sequence monad. Take care when opting in and out of the *IObservable<T>* monad. Doing so excessively can quickly make a mess of your code base, and may indicate a design flaw.

Advanced error handling

Exceptions happen. Exceptions themselves are not bad or good, however the way we raise or catch them can. Some exceptions are predictable and are due to sloppy code, for example a *DivideByZeroException*. Other exceptions cannot be prevented with defensive coding, for example an I/O exception like *FileNotFoundException* or *TimeoutException*. In these cases, we need to cater for the exception gracefully. Providing some sort of error message to the user, logging the error or perhaps retrying are all potential ways to handle these exceptions.

The *IObserver<T>* interface and *Subscribe* extension methods provide the ability to cater for sequences that terminate in error, however they leave the sequence terminated. They also do not offer a composable way to cater for different *Exception* types. A functional approach that enables composition of error handlers, allowing us to remain in the monad, would be more useful. Again, Rx delivers.

Control flow constructs

Using marble diagrams, we will examine various ways to handle different control flows. Just as with normal .NET code, we have flow control constructs such as `try`/`catch`/`finally`. In this chapter we see how they can be applied to observable sequences.

Catch

Just like a catch in SEH (Structured Exception Handling), with Rx you have the option of swallowing an exception, wrapping it in another exception or performing some other logic.

We already know that observable sequences can handle erroneous situations with the *OnError* construct. A useful method in Rx for handling an *OnError* notification is the *Catch* extension method. Catch allows you to intercept a specific *Exception* type and then continue with another sequence.

Below is the signature for the simple overload of catch:

```
public static IObservable<TSource> Catch<TSource>(
        this IObservable<TSource> first,
        IObservable<TSource> second)
{
        ...
}
```

Swallowing exceptions

With Rx, you can catch and swallow exceptions in a similar way to SEH. It is quite simple; we use the *Catch* extension method and provide an empty sequence as the second value.

We can represent an exception being swallowed like this with a marble diagram.

```
S1--1--2--3--X

S2           -|

R --1--2--3----|
```

Here S1 represents the first sequence that ends with an error (X). S2 is the continuation sequence, an empty sequence. R is the result sequence which starts as S1, then continues with S2 when S1 terminates.

```
var source = new Subject<int>();
var result = source.Catch(Observable.Empty<int>());
result.Dump("Catch"););
source.OnNext(1);
source.OnNext(2);
source.OnError(new Exception("Fail!"));
```

Output:

```
Catch-->1
Catch-->2
Catch completed
```

The example above will catch and swallow all types of exceptions. This is somewhat equivalent to the following with SEH:

```
try
{
        DoSomeWork();
}
catch
```

```
{
}
```

Just as it is generally avoided in SEH, you probably also want to limit your use of swallowing errors in Rx. You may, however, have a specific exception you want to handle. Catch has an overload that enables you specify the type of exception. Just as the following code would allow you to catch a *TimeoutException*:

```
try
{
        //
}
catch (TimeoutException tx)
{
        //
}
```

Rx also offers an overload of *Catch* to cater for this.

```
public static IObservable<TSource> Catch<TSource, TException>(
        this IObservable<TSource> source,
        Func<TException, IObservable<TSource>> handler)
        where TException : Exception
{
        ...
}
```

The following Rx code allows you to catch a *TimeoutException*. Instead of providing a second sequence, we provide a function that takes the exception and returns a sequence. This allows you to use a factory to create your continuation. In this example, we add the value -1 to the error sequence and then complete it.

```
var source = new Subject<int>();
var result = source.Catch<int, TimeoutException>(tx=>Observable.Return(-1));
result.Dump("Catch");
source.OnNext(1);
source.OnNext(2);
source.OnError(new TimeoutException());
```

Output:

```
Catch-->1
Catch-->2
Catch-->-1
Catch completed
```

If the sequence was to terminate with an *Exception* that could not be cast to a *TimeoutException*, then the error would not be caught and would flow through to the subscriber.

```
var source = new Subject<int>();
var result = source.Catch<int, TimeoutException>(tx=>Observable.Return(-1));
result.Dump("Catch");
source.OnNext(1);
source.OnNext(2);
source.OnError(new ArgumentException("Fail!"));
```

Output:

```
Catch-->1
Catch-->2
Catch failed-->Fail!
```

Finally

Similar to the `finally` statement with SEH, Rx exposes the ability to execute code on completion of a sequence, regardless of how it terminates. The *Finally* extension method accepts an *Action* as a parameter. This *Action* will be invoked if the sequence terminates normally or erroneously, or if the subscription is disposed of.

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

```
        ...
}
```

In this example, we have a sequence that completes. We provide an action and see that it is called after our `OnCompleted` handler.

```
var source = new Subject<int>();
var result = source.Finally(() => Console.WriteLine("Finally action ran"));
result.Dump("Finally");
source.OnNext(1);
source.OnNext(2);
source.OnNext(3);
source.OnCompleted();
```

Output:

```
Finally-->1
Finally-->2
Finally-->3
Finally completed
Finally action ran
```

In contrast, the source sequence could have terminated with an exception. In that case, the exception would have been sent to the console, and then the delegate we provided would have been executed.

Alternatively, we could have disposed of our subscription. In the next example, we see that the *Finally* action is invoked even though the sequence does not complete.

```
var source = new Subject<int>();
var result = source.Finally(() => Console.WriteLine("Finally"));
var subscription = result.Subscribe(
        Console.WriteLine,
        Console.WriteLine,
        () => Console.WriteLine("Completed"));
source.OnNext(1);
source.OnNext(2);
source.OnNext(3);
subscription.Dispose();
```

Output:

```
1
2
3
Finally
```

Note that there is an anomaly in the current implementation of *Finally*. If there is no *OnError* handler provided, the error will be promoted to an exception and thrown. This will be done before the *Finally* action is invoked. We can reproduce this behavior easily by removing the *OnError* handler from our examples above.

```
var source = new Subject<int>();
var result = source.Finally(() => Console.WriteLine("Finally"));
result.Subscribe(
        Console.WriteLine,
        //Console.WriteLine,
        () => Console.WriteLine("Completed"));
source.OnNext(1);
source.OnNext(2);
source.OnNext(3);
//Brings the app down. Finally action is not called.
source.OnError(new Exception("Fail"));
```

Hopefully this will be identified as a bug and fixed by the time you read this in the next release of Rx. Out of academic interest, here is a sample of a *Finally* extension method that would work as expected.

```
public static IObservable<T> MyFinally<T>(
        this IObservable<T> source,
        Action finallyAction)
{
        return Observable.Create<T>(o =>
        {
        var finallyOnce = Disposable.Create(finallyAction);
        var subscription = source.Subscribe(
        o.OnNext,
        ex =>
        {
        try { o.OnError(ex); }
        finally { finallyOnce.Dispose(); }
        },
        () =>
```

```
        {
        try { o.OnCompleted(); }
        finally { finallyOnce.Dispose(); }
        });
        return new CompositeDisposable(subscription, finallyOnce);
        });
}
```

Using

The *Using* factory method allows you to bind the lifetime of a resource to the lifetime of an observable sequence. The signature itself takes two factory methods; one to provide the resource and one to provide the sequence. This allows everything to be lazily evaluated.

```
public static IObservable<TSource> Using<TSource, TResource>(
        Func<TResource> resourceFactory,
        Func<TResource, IObservable<TSource>> observableFactory)
        where TResource : IDisposable
{
        ...
}
```

The *Using* method will invoke both the factories when you subscribe to the sequence. The resource will be disposed of when the sequence is terminated gracefully, terminated erroneously or when the subscription is disposed.

To provide an example, we will reintroduce the *TimeIt* class from Chapter 3. I could use this handy little class to time the duration of a subscription. In the next example we create an observable sequence with the *Using* factory method. We provide a factory for a *TimeIt* resource and a function that returns a sequence.

```
var source = Observable.Interval(TimeSpan.FromSeconds(1));
var result = Observable.Using(
        () => new TimeIt("Subscription Timer"),
        timeIt => source);
result.Take(5).Dump("Using");
```

Output:

```
Using-->0
Using-->1
Using-->2
Using-->3
Using-->4
Using completed
Subscription Timer took 00:00:05.0138199
```

Due to the `Take(5)` decorator, the sequence completes after five elements and thus the subscription is disposed of. Along with the subscription, the *TimeIt* resource is also disposed of, which invokes the logging of the elapsed time.

This mechanism can find varied practical applications in the hands of an imaginative developer. The resource being an *IDisposable* is convenient; indeed, it makes it so that many types of resources can be bound to, such as other subscriptions, stream reader/writers, database connections, user controls and, with *Disposable.Create(Action)*, virtually anything else.

OnErrorResumeNext

Just the title of this section will send a shudder down the spines of old VB developers! In Rx, there is an extension method called *OnErrorResumeNext* that has similar semantics to the VB keywords/statement that share the same name. This extension method allows the continuation of a sequence with another sequence regardless of whether the first sequence completes gracefully or due to an error. Under normal use, the two sequences would merge as below:

```
S1--0--0--|

S2        --0--|

R --0--0----0--|
```

In the event of a failure in the first sequence, then the sequences would still merge:

```
S1--0--0--X

S2        --0--|

R --0--0----0--|
```

The overloads to *OnErrorResumeNext* are as follows:

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

It is simple to use; you can pass in as many continuations sequences as you like using the various overloads. Usage should be limited however. Just as the *OnErrorResumeNext* keyword warranted mindful use in VB, so should it be used with caution in Rx. It will swallow exceptions quietly and can leave your program in an unknown state. Generally, this will make your code harder to maintain and debug.

Retry

If you are expecting your sequence to encounter predictable issues, you might simply want to retry. One such example when you want to retry is when performing I/O (such as web request or disk access). I/O is notorious for intermittent failures. The *Retry* extension method offers the ability to retry on failure a specified number of times or until it succeeds.

```
//Repeats the source observable sequence until it successfully terminates.
public static IObservable<TSource> Retry<TSource>(
        this IObservable<TSource> source)
{
        ...
}
// Repeats the source observable sequence the specified number of times or until it
//  successfully terminates.
public static IObservable<TSource> Retry<TSource>(
        this IObservable<TSource> source, int retryCount)
{
        ...
}
```

In the diagram below, the sequence (S) produces values then fails. It is re-subscribed, after which it produces values and fails again; this happens a total of two times. The result sequence (R) is the concatenation of all the successive subscriptions to (S).

```
S --1--2--X

           --1--2--3--X

                       --1

R --1--2------1--2--3------1
```

In the next example, we just use the simple overload that will always retry on any exception.

```
public static void RetrySample<T>(IObservable<T> source)
```

```
{
        source.Retry().Subscribe(t=>Console.WriteLine(t)); //Will always retry
        Console.ReadKey();
}
```

Given the source [0,1,2,X], the output would be:

```
0
1
2
0
1
2
0
1
2
```

This output would continue forever, as we throw away the token from the subscribe method. As a marble diagram it would look like this:

```
S--0--1--2--x

            --0--1--2--x

                        --0--

R--0--1--2-----0--1--2-----0--
```

Alternatively, we can specify the maximum number of retries. In this example, we only retry once, therefore the error that gets published on the second subscription will be passed up to the final subscription. Note that to retry once you pass a value of 2. Maybe the method should have been called *Try*?

```
source.Retry(2).Dump("Retry(2)");
```

Output:

```
Retry(2)-->0
Retry(2)-->1
Retry(2)-->2
Retry(2)-->0
Retry(2)-->1
Retry(2)-->2
Retry(2) failed-->Test Exception
```

As a marble diagram, this would look like:

```
S--0--1--2--x

            --0--1--2--x

R--0--1--2-----0--1--2--x
```

Proper care should be taken when using the infinite repeat overload. Obviously if there is a persistent problem with your underlying sequence, you may find yourself stuck in an infinite loop. Also, take note that there is no overload that allows you to specify the type of exception to retry on.

A useful extension method to add to your own library might be a "Back Off and Retry" method. The teams I have worked with have found such a feature useful when performing I/O, especially network requests. The concept is to try, and on failure wait for a given period of time and then try again. Your version of this method may take into account the type of *Exception* you want to retry on, as well as the maximum number of times to retry. You may even want to lengthen the to wait period to be less aggressive on each subsequent retry.

Requirements for exception management that go beyond simple *OnError* handlers are commonplace. Rx delivers the basic exception handling operators which you can use to compose complex and robust

queries. In this chapter we have covered advanced error handling and some more resource management features from Rx. We looked at the *Catch*, *Finally* and *Using* methods as well as the other methods like *OnErrorResumeNext* and *Retry*, that allow you to play a little 'fast and loose'. We have also revisited the use of marble diagrams to help us visualize the combination of multiple sequences. This will help us in our next chapter where we will look at other ways of composing and aggregating observable sequences.

Combining sequences

Data sources are everywhere, and sometimes we need to consume data from more than just a single source. Common examples that have many inputs include: multi touch surfaces, news feeds, price feeds, social media aggregators, file watchers, heart-beating/polling servers, etc. The way we deal with these multiple stimuli is varied too. We may want to consume it all as a deluge of integrated data, or one sequence at a time as sequential data. We could also get it in an orderly fashion, pairing data values from two sources to be processed together, or perhaps just consume the data from the first source that responds to the request.

We have uncovered the benefits of operator composition; now we turn our focus to sequence composition. Earlier on, we briefly looked at operators that work with multiple sequences such as *SelectMany*, *TakeUntil*/*SkipUntil*, *Catch* and *OnErrorResumeNext*. These give us a hint at the potential that sequence composition can deliver. By uncovering the features of sequence composition with Rx, we find yet another layer of game changing functionality. Sequence composition enables you to create complex queries across multiple data sources. This unlocks the possibility to write some very powerful and succinct code.

Now we will build upon the concepts covered in the Advanced Error Handling chapter. There we were able to provide continuations for sequences that failed. We will now examine operators aimed at composing sequences that are still operational instead of sequences that have terminated due to an error.

Sequential concatenation

The first methods we will look at are those that concatenate sequences sequentially. They are very similar to the methods we have seen before for dealing with faulted sequences.

Concat

The *Concat* extension method is probably the most simple composition method. It simply concatenates two sequences. Once the first sequence completes, the second sequence is subscribed to and its values are passed on through to the result sequence. It behaves just like the *Catch* extension method, but will concatenate operational sequences when they complete, instead of faulted sequences when they *OnError*. The simple signature for *Concat* is as follows.

```
// Concatenates two observable sequences. Returns an observable sequence that contains the
//  elements of the first sequence, followed by those of the second the sequence.
public static IObservable<TSource> Concat<TSource>(
        this IObservable<TSource> first,
        IObservable<TSource> second)
{
        ...
}
```

Usage of *Concat* is familiar. Just like *Catch* or *OnErrorResumeNext*, we pass the continuation sequence to the extension method.

```
//Generate values 0,1,2
var s1 = Observable.Range(0, 3);
//Generate values 5,6,7,8,9
var s2 = Observable.Range(5, 5);
s1.Concat(s2)
        .Subscribe(Console.WriteLine);
```

Returns:

```
s1 --0--1--2-|

s2          -5--6--7--8--|

r  --0--1--2--5--6--7--8--|
```

If either sequence was to fault so too would the result sequence. In particular, if *s1* produced an `OnError` notification, then *s2* would never be used. If you wanted *s2* to be used regardless of how s1 terminates, then *OnErrorResumeNext* would be your best option.

Concat also has two useful overloads. These overloads allow you to pass multiple observable sequences as either a `params` array or an *IEnumerable<IObservable<T>>*.

```
public static IObservable<TSource> Concat<TSource>(
        params IObservable<TSource>[] sources)
{...}
public static IObservable<TSource> Concat<TSource>(
        this IEnumerable<IObservable<TSource>> sources)
{...}
```

The ability to pass an *IEnumerable<IObservable<T>>* means that the multiple sequences can be lazily evaluated. The overload that takes a `params` array is well-suited to times when we know how many sequences we want to merge at compile time, whereas the *IEnumerable<IObservable<T>>* overload is a better fit when we do not know this ahead of time.

In the case of the lazily evaluated *IEnumerable<IObservable<T>>*, the *Concat* method will take one sequence, subscribe until it is completed and then switch to the next sequence. To help illustrate this, we create a method that returns a sequence of sequences and is sprinkled with logging. It returns three

observable sequences each with a single value [1], [2] and [3]. Each sequence returns its value on a timer delay.

```
public IEnumerable<IObservable<long>> GetSequences()
{
        Console.WriteLine("GetSequences() called");
        Console.WriteLine("Yield 1st sequence");
        yield return Observable.Create<long>(o =>
        {
        Console.WriteLine("1st subscribed to");
        return Observable.Timer(TimeSpan.FromMilliseconds(500))
        .Select(i=>1L)
        .Subscribe(o);
        });
        Console.WriteLine("Yield 2nd sequence");
        yield return Observable.Create<long>(o =>
        {
        Console.WriteLine("2nd subscribed to");
        return Observable.Timer(TimeSpan.FromMilliseconds(300))
        .Select(i=>2L)
        .Subscribe(o);
        });
        Thread.Sleep(1000);     //Force a delay
        Console.WriteLine("Yield 3rd sequence");
        yield return Observable.Create<long>(o =>
        {
        Console.WriteLine("3rd subscribed to");
        return Observable.Timer(TimeSpan.FromMilliseconds(100))
        .Select(i=>3L)
        .Subscribe(o);
        });
        Console.WriteLine("GetSequences() complete");
}
```

When we call our `GetSequences` method and concatenate the results, we see the following output using our `Dump` extension method.

```
GetSequences().Concat().Dump("Concat");
```

Output:

```
GetSequences() called
Yield 1st sequence
1st subscribed to
Concat-->1
Yield 2nd sequence
2nd subscribed to
Concat-->2
Yield 3rd sequence
3rd subscribed to
Concat-->3
GetSequences() complete
Concat completed
```

Below is a marble diagram of the *Concat* operator applied to the `GetSequences` method. 's1', 's2' and 's3' represent sequence 1, 2 and 3. Respectively, 'rs' represents the result sequence.

```
s1-----1|

s2      ---2|

s3          -3|

rs-----1---2-3|
```

You should note that the second sequence is only yielded once the first sequence has completed. To prove this, we explicitly put in a 500ms delay on producing a value and completing. Once that happens, the second sequence is then subscribed to. When that sequence completes, then the third sequence is processed in the same fashion.

Repeat

Another simple extension method is *Repeat*. It allows you to simply repeat a sequence, either a specified or an infinite number of times.

```
// Repeats the observable sequence indefinitely and sequentially.
public static IObservable<TSource> Repeat<TSource>(
        this IObservable<TSource> source)
{...}
//Repeats the observable sequence a specified number of times.
public static IObservable<TSource> Repeat<TSource>(
        this IObservable<TSource> source,
        int repeatCount)
{...}
```

If you use the overload that loops indefinitely, then the only way the sequence will stop is if there is an error or the subscription is disposed of. The overload that specifies a repeat count will stop on error, un-subscription, or when it reaches that count. This example shows the sequence [0,1,2] being repeated three times.

```
var source = Observable.Range(0, 3);
var result = source.Repeat(3);
result.Subscribe(
        Console.WriteLine,
        () => Console.WriteLine("Completed"));
```

Output:

```
0
1
2
0
1
2
0
1
2
Completed
```

StartWith

Another simple concatenation method is the *StartWith* extension method. It allows you to prefix values to a sequence. The method signature takes a `params` array of values so it is easy to pass in as many or as few values as you need.

```
// prefixes a sequence of values to an observable sequence.
public static IObservable<TSource> StartWith<TSource>(
        this IObservable<TSource> source,
        params TSource[] values)
{
        ...
}
```

Using *StartWith* can give a similar effect to a *BehaviorSubject<T>* by ensuring a value is provided as soon as a consumer subscribes. It is not the same as a *BehaviorSubject* however, as it will not cache the last value.

In this example, we prefix the values -3, -2 and -1 to the sequence [0,1,2].

```
//Generate values 0,1,2
var source = Observable.Range(0, 3);
var result = source.StartWith(-3, -2, -1);
result.Subscribe(
        Console.WriteLine,
        () => Console.WriteLine("Completed"));
```

Output:

```
-3
-2
-1
0
1
2
Completed
```

Concurrent sequences

The next set of methods aims to combine observable sequences that are producing values concurrently. This is an important step in our journey to understanding Rx. For the sake of simplicity, we have avoided introducing concepts related to concurrency until we had a broad understanding of the simple concepts.

Amb

The *Amb* method was a new concept to me when I started using Rx. It is a non-deterministic function, first introduced by John McCarthy and is an abbreviation of the word *Ambiguous*. The Rx implementation will return values from the sequence that is first to produce values, and will completely ignore the other sequences. In the examples below I have three sequences that all produce values. The sequences can be represented as the marble diagram below.

```
s1 -1--1--|

s2 --2--2--|

s3 ---3--3--|

r  -1--1--|
```

The code to produce the above is as follows.

```
var s1 = new Subject<int>();
var s2 = new Subject<int>();
var s3 = new Subject<int>();
var result = Observable.Amb(s1, s2, s3);
result.Subscribe(
        Console.WriteLine,
        () => Console.WriteLine("Completed"));
s1.OnNext(1);
s2.OnNext(2);
s3.OnNext(3);
s1.OnNext(1);
s2.OnNext(2);
s3.OnNext(3);
s1.OnCompleted();
s2.OnCompleted();
s3.OnCompleted();
```

Output:

```
1
1
Completed
```

If we comment out the first `s1.OnNext(1);` then s2 would produce values first and the marble diagram would look like this.

```
s1 ---1--|

s2 -2--2--|

s3 --3--3--|

r  -2--2--|
```

The *Amb* feature can be useful if you have multiple cheap resources that can provide values, but latency is widely variable. For an example, you may have servers replicated around the world. Issuing a query is cheap for both the client to send and for the server to respond, however due to network conditions the latency is not predictable and varies considerably. Using the *Amb* operator, you can send the same request out to many servers and consume the result of the first that responds.

There are other useful variants of the *Amb* method. We have used the overload that takes a `params` array of sequences. You could alternatively use it as an extension method and chain calls until you have included all the target sequences (e.g. s1.Amb(s2).Amb(s3)). Finally, you could pass in an *IEnumerable<IObservable<T>>*.

```
// Propagates the observable sequence that reacts first.
public static IObservable<TSource> Amb<TSource>(
        this IObservable<TSource> first,
        IObservable<TSource> second)
{...}
public static IObservable<TSource> Amb<TSource>(
        params IObservable<TSource>[] sources)
{...}
public static IObservable<TSource> Amb<TSource>(
        this IEnumerable<IObservable<TSource>> sources)
{...}
```

Reusing the `GetSequences` method from the *Concat* section, we see that the evaluation of the outer (IEnumerable) sequence is eager.

```
GetSequences().Amb().Dump("Amb");
```

Output:

```
GetSequences() called
Yield 1st sequence
Yield 2nd sequence
Yield 3rd sequence
GetSequences() complete
1st subscribed to
2nd subscribed to
3rd subscribed to
Amb-->3
Amb completed
```

Marble:

```
s1-----1|

s2---2|

s3-3|

rs-3|
```

Take note that the inner observable sequences are not subscribed to until the outer sequence has yielded them all. This means that the third sequence is able to return values the fastest even though there are two sequences yielded one second before it (due to the `Thread.Sleep`).

Merge

The *Merge* extension method does a primitive combination of multiple concurrent sequences. As values from any sequence are produced, those values become part of the result sequence. All sequences need to be of the same type, as per the previous methods. In this diagram, we can see *s1* and *s2* producing values concurrently and the values falling through to the result sequence as they occur.

```
s1 --1--1--1--|

s2 ---2---2---2|

r  --12-1-21--2|
```

The result of a *Merge* will complete only once all input sequences complete. By contrast, the *Merge* operator will error if any of the input sequences terminates erroneously.

```
//Generate values 0,1,2
var s1 = Observable.Interval(TimeSpan.FromMilliseconds(250))
        .Take(3);
```

```
//Generate values 100,101,102,103,104
var s2 = Observable.Interval(TimeSpan.FromMilliseconds(150))
        .Take(5)
        .Select(i => i + 100);
s1.Merge(s2)
        .Subscribe(
        Console.WriteLine,
        ()=>Console.WriteLine("Completed"));
```

The code above could be represented by the marble diagram below. In this case, each unit of time is 50ms. As both sequences produce a value at 750ms, there is a race condition and we cannot be sure which value will be notified first in the result sequence (sR).

```
s1 ----0----0----0|


s2 --0--0--0--0--0|


sR --0-00--00-0--00|
```

Output:

```
100
0
101
102
1
103
104 //Note this is a race condition. 2 could be
2 // published before 104.
```

You can chain this overload of the *Merge* operator to merge multiple sequences. *Merge* also provides numerous other overloads that allow you to pass more than two source sequences. You can use the static method *Observable.Merge* which takes a `params` array of sequences that is known at compile time. You could pass in an *IEnumerable* of sequences like the *Concat* method. *Merge* also has the overload that takes an *IObservable<IObservable<T>>*, a nested observable. To summarize:

- Chain *Merge* operators together e.g. `s1.Merge(s2).Merge(s3)`
- Pass a `params` array of sequences to the *Observable.Merge* static method. e.g. `Observable.Merge(s1,s2,s3)`
- Apply the *Merge* operator to an *IEnumerable<IObservable<T>>*.
- Apply the *Merge* operator to an *IObservable<IObservable<T>>*.

```
/// Merges two observable sequences into a single observable sequence.
/// Returns a sequence that merges the elements of the given sequences.
public static IObservable<TSource> Merge<TSource>(
        this IObservable<TSource> first,
        IObservable<TSource> second)
{...}
// Merges all the observable sequences into a single observable sequence.
// The observable sequence that merges the elements of the observable sequences.
public static IObservable<TSource> Merge<TSource>(
        params IObservable<TSource>[] sources)
{...}
// Merges an enumerable sequence of observable sequences into a single observable sequence.
public static IObservable<TSource> Merge<TSource>(
        this IEnumerable<IObservable<TSource>> sources)
{...}
// Merges an observable sequence of observable sequences into an observable sequence.
// Merges all the elements of the inner sequences in to the output sequence.
public static IObservable<TSource> Merge<TSource>(
        this IObservable<IObservable<TSource>> sources)
{...}
```

For merging a known number of sequences, the first two operators are effectively the same thing and which style you use is a matter of taste: either provide them as a `params` array or chain the operators together. The third and fourth overloads allow to you merge sequences that can be evaluated lazily at run time. The *Merge* operators that take a sequence of sequences make for an interesting concept. You can either pull or be pushed observable sequences, which will be subscribed to immediately.

If we again reuse the `GetSequences` method, we can see how the *Merge* operator works with a sequence of sequences.

```
GetSequences().Merge().Dump("Merge");
```

Output:

```
GetSequences() called
Yield 1st sequence
1st subscribed to
Yield 2nd sequence
2nd subscribed to
Merge-->2
Merge-->1
Yield 3rd sequence
3rd subscribed to
GetSequences() complete
Merge-->3
Merge completed
```

As we can see from the marble diagram, s1 and s2 are yielded and subscribed to immediately. s3 is not yielded for one second and then is subscribed to. Once all input sequences have completed, the result sequence completes.

```
s1-----1|

s2---2|

s3          -3|

rs---2-1-----3|
```

Switch

Receiving all values from a nested observable sequence is not always what you need. In some scenarios, instead of receiving everything, you may only want the values from the most recent inner sequence. A great example of this is live searches. As you type, the text is sent to a search service and the results are returned to you as an observable sequence. Most implementations have a slight delay before sending the request so that unnecessary work does not happen. Imagine I want to search for "Intro to Rx". I quickly type in "Into to" and realize I have missed the letter 'r'. I stop briefly and change the text to "Intro ". By now, two searches have been sent to the server. The first search will return results that I do not want. Furthermore, if I were to receive data for the first search merged together with results for the second search, it would be a very odd experience for the user. This scenario fits perfectly with the *Switch* method.

In this example, there is a source that represents a sequence of search text. Values the user types are represented as the source sequence. Using *Select*, we pass the value of the search to a function that takes a `string` and returns an *IObservable<string>*. This creates our resulting nested sequence, *IObservable<IObservable<string>>*.

Search function signature:

```
private IObservable<string> SearchResults(string query)
{
        ...
}
```

Using *Merge* with overlapping search:

```
IObservable<string> searchValues = ....;
IObservable<IObservable<string>> search = searchValues
        .Select(searchText=>SearchResults(searchText));
var subscription = search
        .Merge()
        .Subscribe(
        Console.WriteLine);
```

If we were lucky and each search completed before the next element from `searchValues` was produced, the output would look sensible. It is much more likely, however that multiple searches will result in overlapped search results. This marble diagram shows what the *Merge* function could do in such a

situation.

- *SV* is the searchValues sequence
- *S1* is the search result sequence for the first value in searchValues/SV
- *S2* is the search result sequence for the second value in searchValues/SV
- *S3* is the search result sequence for the third value in searchValues/SV
- *RM* is the result sequence for the merged (*R*esult *M*erge) sequences

```
SV--1---2---3---|

S1  -1--1--1--1|

S2      --2-2--2--2|

S3          -3--3|

RM---1--1-2123123-2|
```

Note how the values from the search results are all mixed together. This is not what we want. If we use the *Switch* extension method we will get much better results. *Switch* will subscribe to the outer sequence and as each inner sequence is yielded it will subscribe to the new inner sequence and dispose of the subscription to the previous inner sequence. This will result in the following marble diagram where *RS* is the result sequence for the Switch (*R*esult *S*witch) sequences

```
SV--1---2---3---|

S1  -1--1--1--1|

S2      --2-2--2--2|

S3          -3--3|

RS --1--1-2-23--3|
```

Also note that, even though the results from S1 and S2 are still being pushed, they are ignored as their subscription has been disposed of. This eliminates the issue of overlapping values from the nested sequences.

Pairing sequences

The previous methods allowed us to flatten multiple sequences sharing a common type into a result sequence of the same type. These next sets of methods still take multiple sequences as an input, but attempt to pair values from each sequence to produce a single value for the output sequence. In some cases, they also allow you to provide sequences of different types.

CombineLatest

The *CombineLatest* extension method allows you to take the most recent value from two sequences, and with a given function transform those into a value for the result sequence. Each input sequence has the last value cached like `Replay(1)`. Once both sequences have produced at least one value, the latest output from each sequence is passed to the `resultSelector` function every time either sequence produces a value. The signature is as follows.

```
// Composes two observable sequences into one observable sequence by using the selector
//  function whenever one of the observable sequences produces an element.
public static IObservable<TResult> CombineLatest<TFirst, TSecond, TResult>(
        this IObservable<TFirst> first,
        IObservable<TSecond> second,
        Func<TFirst, TSecond, TResult> resultSelector)
{...}
```

The marble diagram below shows off usage of *CombineLatest* with one sequence that produces numbers (N), and the other letters (L). If the `resultSelector` function just joins the number and letter together as a pair, this would be the result (R):

```
N---1---2---3---

L--a------bc----

R---1---2-223---

    a   a bcc
```

If we slowly walk through the above marble diagram, we first see that `L` produces the letter 'a'. `N` has not produced any value yet so there is nothing to pair, no value is produced for the result (R). Next, `N` produces the number '1' so we now have a pair '1a' that is yielded in the result sequence. We then receive the number '2' from `N`. The last letter is still 'a' so the next pair is '2a'. The letter 'b' is then produced creating the pair '2b', followed by 'c' giving '2c'. Finally the number 3 is produced and we get the pair '3c'.

This is great in case you need to evaluate some combination of state which needs to be kept up-to-date when the state changes. A simple example would be a monitoring system. Each service is represented by a sequence that returns a Boolean indicating the availability of said service. The monitoring status is green if all services are available; we can achieve this by having the result selector perform a logical AND. Here is an example.

```
IObservable<bool> webServerStatus = GetWebStatus();
IObservable<bool> databaseStatus = GetDBStatus();
//Yields true when both systems are up.
var systemStatus = webServerStatus
        .CombineLatest(
        databaseStatus,
        (webStatus, dbStatus) => webStatus && dbStatus);
```

Some readers may have noticed that this method could produce a lot of duplicate values. For example, if the web server goes down the result sequence will yield '`false`'. If the database then goes

down, another (unnecessary) '`false`' value will be yielded. This would be an appropriate time to use the *DistictUntilChanged* extension method. The corrected code would look like the example below.

```
//Yields true when both systems are up, and only on change of status
var systemStatus = webServerStatus
        .CombineLatest(
        databaseStatus,
        (webStatus, dbStatus) => webStatus && dbStatus)
        .DistinctUntilChanged();
```

To provide an even better service, we could provide a default value by prefixing `false` to the sequence.

```
//Yields true when both systems are up, and only on change of status
var systemStatus = webServerStatus
        .CombineLatest(
        databaseStatus,
        (webStatus, dbStatus) => webStatus && dbStatus)
        .DistinctUntilChanged()
        .StartWith(false);
```

Zip

The *Zip* extension method is another interesting merge feature. Just like a zipper on clothing or a bag, the *Zip* method brings together two sequences of values as pairs; two by two. Things to note about the *Zip* function is that the result sequence will complete when the first of the sequences complete, it will error if either of the sequences error and it will only publish once it has a pair of fresh values from each source sequence. So if one of the source sequences publishes values faster than the other sequence, the rate of publishing will be dictated by the slower of the two sequences.

```
//Generate values 0,1,2
var nums = Observable.Interval(TimeSpan.FromMilliseconds(250))
        .Take(3);
//Generate values a,b,c,d,e,f
var chars = Observable.Interval(TimeSpan.FromMilliseconds(150))
        .Take(6)
        .Select(i => Char.ConvertFromUtf32((int)i + 97));
//Zip values together
nums.Zip(chars, (lhs, rhs) => new { Left = lhs, Right = rhs })
        .Dump("Zip");
```

This can be seen in the marble diagram below. Note that the result uses two lines so that we can represent a complex type, i.e. the anonymous type with the properties Left and Right.

```
nums  ----0----1----2|

chars --a--b--c--d--e--f|

result----0----1----2|

          a    b    c|
```

The actual output of the code:

```
{ Left = 0, Right = a }
{ Left = 1, Right = b }
{ Left = 2, Right = c }
```

Note that the `nums` sequence only produced three values before completing, while the `chars` sequence produced six values. The result sequence thus has three values, as this was the most pairs that could be made.

The first use I saw of *Zip* was to showcase drag and drop. The example tracked mouse movements from a `MouseMove` event that would produce event arguments with its current X,Y coordinates. First, the example turns the event into an observable sequence. Then they cleverly zipped the sequence with a `Skip(1)` version of the same sequence. This allows the code to get a delta of the mouse position, i.e. where it is now (sequence.Skip(1)) minus where it was (sequence). It then applied the delta to the control it was dragging.

To visualize the concept, let us look at another marble diagram. Here we have the mouse movement (MM) and the Skip 1 (S1). The numbers represent the index of the mouse movement.

```
MM --1--2--3--4--5

S1    --2--3--4--5

Zip   --1--2--3--4

        2  3  4  5
```

Here is a code sample where we fake out some mouse movements with our own subject.

```
var mm = new Subject<Coord>();
var s1 = mm.Skip(1);
var delta = mm.Zip(s1,
        (prev, curr) => new Coord
        {
        X = curr.X - prev.X,
        Y = curr.Y - prev.Y
        });
delta.Subscribe(
        Console.WriteLine,
        () => Console.WriteLine("Completed"));
mm.OnNext(new Coord { X = 0, Y = 0 });
mm.OnNext(new Coord { X = 1, Y = 0 }); //Move across 1
mm.OnNext(new Coord { X = 3, Y = 2 }); //Diagonally up 2
mm.OnNext(new Coord { X = 0, Y = 0 }); //Back to 0,0
mm.OnCompleted();
```

This is the simple Coord(inate) class we use.

```
public class Coord
{
        public int X { get; set; }
        public int Y { get; set; }
        public override string ToString()
        {
        return string.Format("{0},{1}", X, Y);
        }
}
```

Output:

```
0,1
2,2
-3,-2
Completed
```

It is also worth noting that *Zip* has a second overload that takes an *IEnumerable<T>* as the second input sequence.

```
// Merges an observable sequence and an enumerable sequence into one observable sequence
//  containing the result of pair-wise combining the elements by using the selector function.
public static IObservable<TResult> Zip<TFirst, TSecond, TResult>(
        this IObservable<TFirst> first,
        IEnumerable<TSecond> second,
        Func<TFirst, TSecond, TResult> resultSelector)
{...}
```

This allows us to zip sequences from both *IEnumerable<T>* and *IObservable<T>* paradigms!

And-Then-When

If *Zip* only taking two sequences as an input is a problem, then you can use a combination of the three *And*/*Then*/*When* methods. These methods are used slightly differently from most of the other Rx methods. Out of these three, *And* is the only extension method to *IObservable<T>*. Unlike most Rx operators, it does not return a sequence; instead, it returns the mysterious type *Pattern<T1, T2>*. The *Pattern<T1, T2>* type is public (obviously), but all of its properties are internal. The only two (useful) things you can do with a *Pattern<T1, T2>* are invoking its *And* or *Then* methods. The *And* method called on the *Pattern<T1, T2>* returns a *Pattern<T1, T2, T3>*. On that type, you will also find the *And* and *Then* methods. The generic *Pattern* types are there to allow you to chain multiple *And* methods together, each one extending the generic type parameter list by one. You then bring them

all together with the *Then* method overloads. The *Then* methods return you a *Plan* type. Finally, you pass this *Plan* to the *Observable.When* method in order to create your sequence.

It may sound very complex, but comparing some code samples should make it easier to understand. It will also allow you to see which style you prefer to use.

To *Zip* three sequences together, you can either use *Zip* methods chained together like this:

```
var one = Observable.Interval(TimeSpan.FromSeconds(1)).Take(5);
var two = Observable.Interval(TimeSpan.FromMilliseconds(250)).Take(10);
var three = Observable.Interval(TimeSpan.FromMilliseconds(150)).Take(14);
//lhs represents 'Left Hand Side'
//rhs represents 'Right Hand Side'
var zippedSequence = one
        .Zip(two, (lhs, rhs) => new {One = lhs, Two = rhs})
        .Zip(three, (lhs, rhs) => new {One = lhs.One, Two = lhs.Two, Three = rhs});
zippedSequence.Subscribe(
        Console.WriteLine,
        () => Console.WriteLine("Completed"));
```

Or perhaps use the nicer syntax of the *And/Then/When*:

```
var pattern = one.And(two).And(three);
var plan = pattern.Then((first, second, third)=>new{One=first, Two=second, Three=third});
var zippedSequence = Observable.When(plan);
zippedSequence.Subscribe(
        Console.WriteLine,
        () => Console.WriteLine("Completed"));
```

This can be further reduced, if you prefer, to:

```
var zippedSequence = Observable.When(
        one.And(two)
        .And(three)
        .Then((first, second, third) =>
        new {
        One = first,
        Two = second,
        Three = third
        })
        );
zippedSequence.Subscribe(
        Console.WriteLine,
        () => Console.WriteLine("Completed"));
```

The *And/Then/When* trio has more overloads that enable you to group an even greater number of sequences. They also allow you to provide more than one 'plan' (the output of the *Then* method). This gives you the *Merge* feature but on the collection of 'plans'. I would suggest playing around with them if this functionality is of interest to you. The verbosity of enumerating all of the combinations of these methods would be of low value. You will get far more value out of using them and discovering for yourself.

As we delve deeper into the depths of what the Rx libraries provide us, we can see more practical usages for it. Composing sequences with Rx allows us to easily make sense of the multiple data sources a problem domain is exposed to. We can concatenate values or sequences together sequentially with *StartWith*, *Concat* and *Repeat*. We can process multiple sequences concurrently with *Merge*, or process a single sequence at a time with *Amb* and *Switch*. Pairing values with *CombineLatest*, *Zip* and the *And/Then/When* operators can simplify otherwise fiddly operations like our drag-and-drop examples and monitoring system status.

Time-shifted sequences

When working with observable sequences, the time axis is an unknown quantity: when will the next notification arrive? When consuming an *IEnumerable* sequence, asynchrony is not a concern; when we call `MoveNext()`, we are blocked until the sequence yields. This chapter looks at the various methods we can apply to an observable sequence when its relationship with time is a concern.

Buffer

Our first subject will be the *Buffer* method. In some situations, you may not want a deluge of individual notifications to process. Instead, you might prefer to work with batches of data. It may be the case that processing one item at a time is just too expensive, and the trade-off is to deal with messages in batches, at the cost of accepting a delay.

The *Buffer* operator allows you to store away a range of values and then re-publish them as a list once the buffer is full. You can temporarily withhold a specified number of elements, stash away all the values for a given time span, or use a combination of both count and time. *Buffer* also offers more advanced overloads that we will look at in a future chapter.

```
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,
        TimeSpan timeSpan,
        int count)
{...}
```

The two overloads of *Buffer* are straight forward and should make it simple for other developers to understand the intent of the code.

```
IObservable<IList<T>> bufferedSequence;
bufferedSequence = mySequence.Buffer(4);
//or
bufferedSequence = mySequence.Buffer(TimeSpan.FromSeconds(1))
```

For some use cases, it may not be enough to specify only a buffer size and a maximum delay period. Some systems may have a sweet spot for the size of a batch they can process, but also have a time constraint to ensure that data is not stale. In this case buffering by both time and count would be suitable.

In this example below, we create a sequence that produces the first ten values one second apart, then a further hundred values within another second. We buffer by a maximum period of three seconds and a maximum batch size of fifteen values.

```
var idealBatchSize = 15;
var maxTimeDelay = TimeSpan.FromSeconds(3);
var source = Observable.Interval(TimeSpan.FromSeconds(1)).Take(10)
        .Concat(Observable.Interval(TimeSpan.FromSeconds(0.01)).Take(100));
source.Buffer(maxTimeDelay, idealBatchSize)
        .Subscribe(
        buffer => Console.WriteLine("Buffer of {1} @ {0}", DateTime.Now, buffer.Count),
        () => Console.WriteLine("Completed"));
```

Output:

```
Buffer of 3 @ 01/01/2012 12:00:03
Buffer of 3 @ 01/01/2012 12:00:06
Buffer of 3 @ 01/01/2012 12:00:09
Buffer of 15 @ 01/01/2012 12:00:10
Buffer of 15 @ 01/01/2012 12:00:10
Buffer of 15 @ 01/01/2012 12:00:10
Buffer of 15 @ 01/01/2012 12:00:11
Buffer of 15 @ 01/01/2012 12:00:11
Buffer of 15 @ 01/01/2012 12:00:11
Buffer of 11 @ 01/01/2012 12:00:11
```

Note the variations in time and buffer size. We never get a buffer containing more than fifteen elements, and we never wait more than three seconds. A practical application of this is when you are loading data from an external source into an *ObservableCollection<T>* in a WPF application. It may be the case that adding one item at a time is just an unnecessary load on the dispatcher (especially if

you are expecting over a hundred items). You may have also measured, for example that processing a batch of fifty items takes 100ms. You decide that this is the maximum amount of time you want to block the dispatcher, to keep the application responsive. This could give us two reasonable values to use: `source.Buffer(TimeSpan.FromMilliseconds(100), 50)`. This means the longest we will block the UI is about 100ms to process a batch of 50 values, and we will never have values waiting for longer than 100ms before they are processed.

Overlapping buffers

Buffer also offers overloads to manipulate the overlapping of the buffers. The variants we have looked at so far do not overlap and have no gaps between buffers, i.e. all values from the source are propagated through.

```
public static IObservable<IList<TSource>> Buffer<TSource>(
        this IObservable<TSource> source,
        int count,
        int skip)
{...}
public static IObservable<IList<TSource>> Buffer<TSource>(
        this IObservable<TSource> source,
        TimeSpan timeSpan,
        TimeSpan timeShift)
{...}
```

There are three interesting things you can do with overlapping buffers:

Overlapping behavior
: Ensure that current buffer includes some or all values from previous buffer

Standard behavior
: Ensure that each new buffer only has new data

Skip behavior
: Ensure that each new buffer not only contains new data exclusively, but also ignores one or more values since the previous buffer

Overlapping buffers by count

If you are specifying a buffer size as a count, then you need to use this overload.

```
public static IObservable<IList<TSource>> Buffer<TSource>(
        this IObservable<TSource> source,
        int count,
        int skip)
{...}
```

You can apply the above scenarios as follows:

Overlapping behavior
: *skip* < *count**

Standard behavior
: *skip* = *count*

Skip behavior
: *skip* > *count*

*The *skip* parameter cannot be less than or equal to zero. If you want to use a value of zero (i.e. each buffer contains all values), then consider using the *Scan* method instead with an *IList<T>* as the accumulator.

Let's see each of these in action. In this example, we have a source that produces values every second. We apply each of the variations of the buffer overload.

```
var source = Observable.Interval(TimeSpan.FromSeconds(1)).Take(10);
source.Buffer(3, 1)
        .Subscribe(
        buffer =>
        {
        Console.WriteLine("--Buffered values");
        foreach (var value in buffer)
        {
        Console.WriteLine(value);
        }
        }, () => Console.WriteLine("Completed"));
```

Output

```
--Buffered values
0
1
2
--Buffered values
1
2
3
--Buffered values
2
3
4
--Buffered values
3
4
5
etc....
```

Note that in each buffer, one value is skipped from the previous batch. If we change the *skip* parameter from 1 to 3 (same as the buffer size), we see standard buffer behavior.

```
var source = Observable.Interval(TimeSpan.FromSeconds(1)).Take(10);
source.Buffer(3, 3)
        ...
```

Output

```
--Buffered values
0
1
2
--Buffered values
3
4
5
--Buffered values
6
7
8
--Buffered values
9
Completed
```

Finally, if we change the *skip* parameter to 5 (a value greater than the count of 3), we can see that two values are lost between each buffer.

```
var source = Observable.Interval(TimeSpan.FromSeconds(1)).Take(10);
source.Buffer(3, 5)
        ...
```

Output

```
--Buffered values
0
1
2
--Buffered values
5
6
7
Completed
```

Overlapping buffers by time

You can, of course, apply the same three behaviors with buffers defined by time instead of count.

```
public static IObservable<IList<TSource>> Buffer<TSource>(
        this IObservable<TSource> source,
        TimeSpan timeSpan,
        TimeSpan timeShift)
{...}
```

To exactly replicate the output from our Overlapping Buffers By Count examples, we only need to provide the following arguments:

```
var source = Observable.Interval(TimeSpan.FromSeconds(1)).Take(10);
var overlapped = source.Buffer(TimeSpan.FromSeconds(3), TimeSpan.FromSeconds(1));
var standard = source.Buffer(TimeSpan.FromSeconds(3), TimeSpan.FromSeconds(3));
var skipped = source.Buffer(TimeSpan.FromSeconds(3), TimeSpan.FromSeconds(5));
```

As our source produces values consistently every second, we can use the same values from our count example but as seconds.

Delay

The *Delay* extension method is a purely a way to time-shift an entire sequence. You can provide either a relative time the sequence should be delayed by using a *TimeSpan*, or an absolute point in time that the sequence should wait for using a *DateTimeOffset*. The relative time intervals between the values are preserved.

```
// Time-shifts the observable sequence by a relative time.
public static IObservable<TSource> Delay<TSource>(
        this IObservable<TSource> source,
        TimeSpan dueTime)
{...}
// Time-shifts the observable sequence by a relative time.
public static IObservable<TSource> Delay<TSource>(
        this IObservable<TSource> source,
        TimeSpan dueTime,
        IScheduler scheduler)
{...}
// Time-shifts the observable sequence by an absolute time.
public static IObservable<TSource> Delay<TSource>(
        this IObservable<TSource> source,
        DateTimeOffset dueTime)
{...}
// Time-shifts the observable sequence by an absolute time.
public static IObservable<TSource> Delay<TSource>(
        this IObservable<TSource> source,
        DateTimeOffset dueTime,
        IScheduler scheduler)
{...}
```

To show the *Delay* method in action, we create a sequence of values one second apart and timestamp them. This will show that it is not the subscription that is being delayed, but the actual forwarding of the notifications to our final subscriber.

```
var source = Observable.Interval(TimeSpan.FromSeconds(1))
        .Take(5)
        .Timestamp();
var delay = source.Delay(TimeSpan.FromSeconds(2));
source.Subscribe(
        value => Console.WriteLine("source : {0}", value),
        () => Console.WriteLine("source Completed"));
delay.Subscribe(
        value => Console.WriteLine("delay : {0}", value),
        () => Console.WriteLine("delay Completed"));
```

Output:

```
source : 0@01/01/2012 12:00:00 pm +00:00
source : 1@01/01/2012 12:00:01 pm +00:00
source : 2@01/01/2012 12:00:02 pm +00:00
delay : 0@01/01/2012 12:00:00 pm +00:00
source : 3@01/01/2012 12:00:03 pm +00:00
delay : 1@01/01/2012 12:00:01 pm +00:00
source : 4@01/01/2012 12:00:04 pm +00:00
source Completed
delay : 2@01/01/2012 12:00:02 pm +00:00
delay : 3@01/01/2012 12:00:03 pm +00:00
delay : 4@01/01/2012 12:00:04 pm +00:00
delay Completed
```

It is worth noting that *Delay* will not time-shift *OnError* notifications. These will be propagated immediately.

Sample

The *Sample* method simply takes the last value for every specified *TimeSpan*. This is great for getting timely data from a sequence that produces too much information for your requirements. This example shows sample in action.

```
var interval = Observable.Interval(TimeSpan.FromMilliseconds(150));
interval.Sample(TimeSpan.FromSeconds(1))
        .Subscribe(Console.WriteLine);
```

Output:

```
5
12
18
```

This output is interesting and this is the reason why I choose the value of 150ms. If we plot the underlying sequence of values against the time they are produced, we can see that *Sample* is taking the last value it received for each period of one second.

Relative time (ms)	Source value	Sampled value
0		
50		
100		
150	0	
200		
250		
300	1	
350		
400		
450	2	
500		
550		
600	3	
650		
700		
750	4	
800		
850		
900	5	
950		
1000		5
1050	6	
1100		
1150		
1200	7	
1250		
1300		
1350	8	
1400		
1450		
1500	9	
1550		
1600		
1650	10	
1700		
1750		
1800	11	
1850		
1900		
1950	12	
2000		12
2050		
2100	13	
2150		
2200		
2250	14	
2300		

2350

2400 15

2450

2500

2550 16

2600

2650

2700 17

2750

2800

2850 18

2900

2950

3000 19 19

Throttle

The *Throttle* extension method provides a sort of protection against sequences that produce values at variable rates and sometimes too quickly. Like the *Sample* method, *Throttle* will return the last sampled value for a period of time. Unlike *Sample* though, *Throttle*'s period is a sliding window. Each time *Throttle* receives a value, the window is reset. Only once the period of time has elapsed will the last value be propagated. This means that the *Throttle* method is only useful for sequences that produce values at a variable rate. Sequences that produce values at a constant rate (like *Interval* or *Timer*) either would have all of their values suppressed if they produced values faster than the throttle period, or all of their values would be propagated if they produced values slower than the throttle period.

```
// Ignores values from an observable sequence which are followed by another value before
//  dueTime.
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)
{...}
```

A great application of the *Throttle* method would be to use it with a live search like "Google Suggest". While the user is still typing we can hold off on the search. Once there is a pause for a given period, we can execute the search with what they have typed. The Rx team has a great example of this scenario in the Rx Hands On Lab

Timeout

We have considered handling timeout exceptions previously in the chapter on Flow control. The *Timeout* extension method allows us terminate the sequence with an error if we do not receive any notifications for a given period. We can either specify the period as a sliding window with a *TimeSpan*, or as an absolute time that the sequence must complete by providing a *DateTimeOffset*.

```
// Returns either the observable sequence or a TimeoutException if the maximum duration
//  between values elapses.
public static IObservable<TSource> Timeout<TSource>(
        this IObservable<TSource> source,
        TimeSpan dueTime)
{...}
public static IObservable<TSource> Timeout<TSource>(
        this IObservable<TSource> source,
        TimeSpan dueTime,
        IScheduler scheduler)
{...}
// Returns either the observable sequence or a TimeoutException if dueTime elapses.
public static IObservable<TSource> Timeout<TSource>(
        this IObservable<TSource> source,
        DateTimeOffset dueTime)
{...}
public static IObservable<TSource> Timeout<TSource>(
        this IObservable<TSource> source,
        DateTimeOffset dueTime,
        IScheduler scheduler)
{...}
```

If we provide a *TimeSpan* and no values are produced within that time span, then the sequence fails with a *TimeoutException*.

```
var source = Observable.Interval(TimeSpan.FromMilliseconds(100)).Take(10)
        .Concat(Observable.Interval(TimeSpan.FromSeconds(2)));
var timeout = source.Timeout(TimeSpan.FromSeconds(1));
timeout.Subscribe(
        Console.WriteLine,
        Console.WriteLine,
        () => Console.WriteLine("Completed"));
```

Output:

```
0
1
2
3
4
System.TimeoutException: The operation has timed out.
```

Like the *Throttle* method, this overload is only useful for sequences that produce values at a variable rate.

The alternative use of *Timeout* is to set an absolute time; the sequence must be completed by then.

```
var dueDate = DateTimeOffset.UtcNow.AddSeconds(4);
var source = Observable.Interval(TimeSpan.FromSeconds(1));
var timeout = source.Timeout(dueDate);
timeout.Subscribe(
        Console.WriteLine,
        Console.WriteLine,
        () => Console.WriteLine("Completed"));
```

Output:

```
0
1
2
System.TimeoutException: The operation has timed out.
```

Perhaps an even more interesting usage of the *Timeout* method is to substitute in an alternative sequence when a timeout occurs. The *Timeout* method has overloads the provide the option of specifying a continuation sequence to use if a timeout occurs. This functionality behaves much like the Catch operator. It is easy to imagine that the simple overloads actually just call through to these over loads and specify an `Observable.Throw<TimeoutException>` as the continuation sequence.

```
// Returns the source observable sequence or the other observable sequence if the maximum
//  duration between values elapses.
```

```
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,
        IObservable<TSource> other,
        IScheduler scheduler)
{...}
// Returns the source observable sequence or the other observable sequence if dueTime
//  elapses.
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,
        IObservable<TSource> other,
        IScheduler scheduler)
{...}
```

Rx provides features to tame the unpredictable element of time in a reactive paradigm. Data can be buffered, throttled, sampled or delayed to meet your needs. Entire sequences can be shifted in time with the delay feature, and timeliness of data can be asserted with the *Timeout* operator. These simple yet powerful features further extend the developer's tool belt for querying data in motion.

Hot and Cold observables

In this chapter, we will look at how to describe and handle two styles of observable sequences:

1. Sequences that are passive and start producing notifications on request (when subscribed to), and
2. Sequences that are active and produce notifications regardless of subscriptions.

In this sense, passive sequences are *Cold* and active are described as being *Hot*. You can draw some similarities between implementations of the *IObservable<T>* interface and implementations of the *IEnumerable<T>* interface with regards to hot and cold. With *IEnumerable<T>*, you could have an on-demand collection via the yield return syntax, or you could have an eagerly-evaluated collection by returning a populated *List<T>*. We can compare the two styles by attempting to read just the first value from a sequence. We can do this with a method like this:

```
public void ReadFirstValue(IEnumerable<int> list)
{
        foreach (var i in list)
        {
        Console.WriteLine("Read out first value of {0}", i);
        break;
        }
}
```

As an alternative to the `break` statement, we could apply a `Take(1)` to the `list`. If we then apply this to an eagerly-evaluated sequence, such as a list, we see the entire list is first constructed, and then returned.

```
public static void Main()
{
        ReadFirstValue(EagerEvaluation());
}
public IEnumerable<int> EagerEvaluation()
{
        var result = new List<int>();
        Console.WriteLine("About to return 1");
        result.Add(1);
        //code below is executed but not used.
        Console.WriteLine("About to return 2");
        result.Add(2);
        return result;
}
```

Output:

```
About to return 1
About to return 2
Read out first value of 1
```

We now apply the same code to a lazily-evaluated sequence.

```
public IEnumerable<int> LazyEvaluation()
{
        Console.WriteLine("About to return 1");
        yield return 1;
        //Execution stops here in this example
        Console.WriteLine("About to return 2");
        yield return 2;
}
```

Output:

```
About to return 1
Read out first value of 1
```

The lazily-evaluated sequence did not have to yield any more values than required. Lazy evaluation is good for on-demand queries whereas eager evaluation is good for sharing sequences so as to avoid re-evaluating multiple times. Implementations of *IObservable<T>* can exhibit similar variations in style.

Examples of hot observables that could publish regardless of whether there are any subscribers would be:

- mouse movements
- timer events
- broadcasts like ESB channels or UDP network packets.
- price ticks from a trading exchange

Some examples of cold observables would be:

- asynchronous request (e.g. when using *Observable.FromAsyncPattern*)
- whenever *Observable.Create* is used
- subscriptions to queues
- on-demand sequences

Cold observables

In this example, we fetch a list of products from a database. In our implementation, we choose to return an *IObservable<string>* and, as we get the results, we publish them until we have the full list, then complete the sequence.

```
private const string connectionString = @"Data Source=.\SQLSERVER;"+
        @"Initial Catalog=AdventureWorksLT2008;Integrated Security=SSPI;"
private static IObservable<string> GetProducts()
{
        return Observable.Create<string>(
        o =>
        {
        using(var conn = new SqlConnection(connectionString))
        using (var cmd = new SqlCommand("Select Name FROM SalesLT.ProductModel", conn))
        {
        conn.Open();
        SqlDataReader reader = cmd.ExecuteReader(CommandBehavior.CloseConnection);
        while (reader.Read())
        {
        o.OnNext(reader.GetString(0));
        }
        o.OnCompleted();
        return Disposable.Create(()=>Console.WriteLine("--Disposed--"));
        }
        });
}
```

This code is just like many existing data access layers that return an *IEnumerable<T>*, however it would be much easier with Rx to access this in an asynchronous manner (using SubscribeOn and ObserveOn). This example of a data access layer is lazily evaluated and provides no caching. Each time the method is used, we reconnect to the database. This is typical of cold observables; calling the method does nothing. Subscribing to the returned *IObservable<T>* will however invoke the create delegate which connects to the database.

Here we have a consumer of our above code, but it explicitly only wants up to three values (the full set has 128 values). This code illustrates that the `Take(3)` expression will restrict what the consumer receives but `GetProducts()` method will still publish *all* of the values.

```
public void ColdSample()
{
        var products = GetProducts().Take(3);
        products.Subscribe(Console.WriteLine);
        Console.ReadLine();
}
```

The *GetProducts()* code above is a pretty naive example, as it lacks the ability to cancel at any time. This means all values are read even though only three were requested. In the later chapter on scheduling, we cover examples on how to provide cancellation correctly.

Hot observables

In our example above, the database was not accessed until the consumer of the *GetProducts()* method subscribed to the return value. Subsequent or even parallel calls to *GetProducts()* would return independent observable sequences and would each make their own independent calls to the database. By contrast, a hot observable is an observable sequence that is producing notifications even if there are no subscribers. The classic cases of hot observables are UI Events and Subjects. For example, if the mouse moves then the *MouseMove* event will be raised. If there are no event handlers registered for the event, then nothing happens. If, on the other hand, we create a *Subject<int>*, we can inject values into it using `OnNext`, regardless of whether there are observers subscribed to the subject or not.

Some observable sequences can appear to be hot when they are in fact cold. A couple of examples that surprise many is *Observable.Interval* and *Observable.Timer* (though it should not come as a shock to attentive readers of the Creating observable sequences chapter). In the example below, we subscribe twice to the same instance, created via the *Interval* factory method. The delay between the two subscriptions should demonstrate that while they are subscribed to the same observable instance, the values each subscription receives are independent, i.e. *Interval* is cold.

```
public void SimpleColdSample()
{
        var period = TimeSpan.FromSeconds(1);
        var observable = Observable.Interval(period);
        observable.Subscribe(i => Console.WriteLine("first subscription : {0}", i));
        Thread.Sleep(period);
        observable.Subscribe(i => Console.WriteLine("second subscription : {0}", i));
        Console.ReadKey();
        /* Output:
        first subscription : 0
        first subscription : 1
        second subscription : 0
        first subscription : 2
        second subscription : 1
        first subscription : 3
        second subscription : 2
        */
}
```

Publish and Connect

If we want to be able to share the actual data values and not just the observable instance, we can use the *Publish()* extension method. This will return an *IConnectableObservable<T>*, which extends *IObservable<T>* by adding a single *Connect()* method. By using the *Publish()* then *Connect()* method, we can get this sharing functionality.

```
var period = TimeSpan.FromSeconds(1);
var observable = Observable.Interval(period).Publish();
observable.Connect();
observable.Subscribe(i => Console.WriteLine("first subscription : {0}", i));
Thread.Sleep(period);
observable.Subscribe(i => Console.WriteLine("second subscription : {0}", i));
```

Output:

```
first subscription : 0
first subscription : 1
second subscription : 1
first subscription : 2
second subscription : 2
```

In the example above, the *observable* variable is an *IConnectableObservable<T>*, and by calling *Connect()* it will subscribe to the underlying (the *Observable.Interval*). In this case, we are quick enough to subscribe before the first item is published, but only on the first subscription. The second subscription subscribes late and misses the first publication. We could move the invocation of the *Connect()* method until after all subscriptions have been made. That way, even with the call to *Thread.Sleep* we will not really subscribe to the underlying until after both subscriptions are made. This would be done as follows:

```
var period = TimeSpan.FromSeconds(1);
var observable = Observable.Interval(period).Publish();
observable.Subscribe(i => Console.WriteLine("first subscription : {0}", i));
Thread.Sleep(period);
observable.Subscribe(i => Console.WriteLine("second subscription : {0}", i));
observable.Connect();
first subscription : 0
second subscription : 0
first subscription : 1
second subscription : 1
first subscription : 2
second subscription : 2
```

As you can imagine, this is quite useful whenever an application needs to share sequences of data. In a financial trading application, if you wanted to consume a price stream for a certain asset in more than one place, you would want to try to reuse a single, common stream and avoid making another subscription to the server providing that data. In a social media application, many widgets may need to be notified whenever someone connects. *Publish* and *Connect* are perfect solutions for this.

Disposal of connections and subscriptions

A point of interest is how disposal is performed. Indeed, we have not covered yet the fact that *Connect* returns an *IDisposable*. By disposing of the 'connection', you can turn the sequence on and off (`Connect()` to toggle it on, disposing toggles it off). In this example, we see that the sequence can be connected and disconnected multiple times.

```
var period = TimeSpan.FromSeconds(1);
var observable = Observable.Interval(period).Publish();
observable.Subscribe(i => Console.WriteLine("subscription : {0}", i));
var exit = false;
while (!exit)
{
        Console.WriteLine("Press enter to connect, esc to exit.");
        var key = Console.ReadKey(true);
        if(key.Key== ConsoleKey.Enter)
        {
        var connection = observable.Connect(); //--Connects here--
        Console.WriteLine("Press any key to dispose of connection.");
        Console.ReadKey();
```

```
        connection.Dispose(); //--Disconnects here--
        }
        if(key.Key==ConsoleKey.Escape)
        {
        exit = true;
        }
}
```

Output:

```
Press enter to connect, esc to exit.
Press any key to dispose of connection.
subscription : 0
subscription : 1
subscription : 2
Press enter to connect, esc to exit.
Press any key to dispose of connection.
subscription : 0
subscription : 1
subscription : 2
Press enter to connect, esc to exit.
```

Let us finally consider automatic disposal of a connection. We want a single sequence to be shared between subscriptions, as per the price stream example mentioned above. We also want to only have the sequence running hot if there are any subscribers. It seems therefore, not only obvious that there should be a mechanism for automatically connecting (once a subscription has been made), but also a mechanism for disconnecting (once there are no more subscriptions) from a sequence. First let us look at what happens to a sequence when we connect with no subscribers, and then later unsubscribe:

```
var period = TimeSpan.FromSeconds(1);
var observable = Observable.Interval(period)
        .Do(l => Console.WriteLine("Publishing {0}", l)) //Side effect to show it is running
        .Publish();
observable.Connect();
Console.WriteLine("Press any key to subscribe");
Console.ReadKey();
var subscription = observable.Subscribe(i => Console.WriteLine("subscription : {0}", i));
Console.WriteLine("Press any key to unsubscribe.");
Console.ReadKey();
subscription.Dispose();
Console.WriteLine("Press any key to exit.");
Console.ReadKey();
```

Output:

```
Press any key to subscribe
Publishing 0
Publishing 1
Press any key to unsubscribe.
Publishing 2
subscription : 2
Publishing 3
subscription : 3
Press any key to exit.
Publishing 4
Publishing 5
```

A few things to note here:

1. I use the *Do* extension method to create side effects on the sequence (i.e. write to the console). This allows us to see when the sequence is actually connected.
2. We connect first and then subscribe, which means that we can publish without any live subscriptions i.e. make the sequence hot.
3. We dispose of our subscription but do not dispose of the connection, which means the sequence will still be running.

RefCount

Let us modify that last example by replacing uses of `Connnect()` by the extension method *RefCount*. This will "magically" implement our requirements for automatic disposal and lazy connection. *RefCount* will take an *IConnectableObservable<T>* and turn it back into an *IObservable<T>* while automatically implementing the "connect" and "disconnect" behavior we are looking for.

```
var period = TimeSpan.FromSeconds(1);
var observable = Observable.Interval(period)
        .Do(l => Console.WriteLine("Publishing {0}", l)) //side effect to show it is running
        .Publish()
        .RefCount();
//observable.Connect(); Use RefCount instead now
Console.WriteLine("Press any key to subscribe");
Console.ReadKey();
var subscription = observable.Subscribe(i => Console.WriteLine("subscription : {0}", i));
Console.WriteLine("Press any key to unsubscribe.");
Console.ReadKey();
subscription.Dispose();
Console.WriteLine("Press any key to exit.");
Console.ReadKey();
```

Output:

```
Press any key to subscribe
Press any key to unsubscribe.
Publishing 0
subscription : 0
Publishing 1
subscription : 1
Publishing 2
subscription : 2
Press any key to exit.
```

The *Publish*/*RefCount* pair is extremely useful for taking a cold observable and sharing it as a hot observable sequence for subsequent observers. *RefCount()* also allows us to avoid a race condition. In the example above, we subscribed to the sequence before a connection was established. This is not always possible, especially if we are exposing the sequence from a method. By using the *RefCount* method we can mitigate the subscribe/connect race condition because of the auto-connect behavior.

Other connectable observables

The *Connect* method is not the only method that returns *IConnectableObservable<T>* instances. The ability to connect or defer an operator's functionality is useful in other areas too.

PublishLast

The *PublishLast()* method is effectively a non-blocking *Last()* call. You can consider it similar to an *AsyncSubject<T>* wrapping your target sequence. You get equivalent semantics to *AsyncSubject<T>* where only the last value is published, and only once the sequence completes.

```
var period = TimeSpan.FromSeconds(1);
var observable = Observable.Interval(period)
        .Take(5)
        .Do(l => Console.WriteLine("Publishing {0}", l)) //side effect to show it is running
        .PublishLast();
observable.Connect();
Console.WriteLine("Press any key to subscribe");
Console.ReadKey();
var subscription = observable.Subscribe(i => Console.WriteLine("subscription : {0}", i));
Console.WriteLine("Press any key to unsubscribe.");
Console.ReadKey();
subscription.Dispose();
Console.WriteLine("Press any key to exit.");
Console.ReadKey();
```

Output:

```
Press any key to subscribe
Publishing 0
Publishing 1
Press any key to unsubscribe.
Publishing 2
Publishing 3
Publishing 4
subscription : 4
Press any key to exit.
```

Replay

The *Replay* extension method allows you take an existing observable sequence and give it 'replay' semantics as per *ReplaySubject<T>*. As a reminder, the *ReplaySubject<T>* will cache all values so that any late subscribers will also get all of the values. In this example, two subscriptions are made on time, and then a third subscription can be made after the sequence completes. Even though the third subscription is made after the underlying sequence has completed, we can still get all of the values.

```
var period = TimeSpan.FromSeconds(1);
var hot = Observable.Interval(period)
        .Take(3)
        .Publish();
hot.Connect();
Thread.Sleep(period); //Run hot and ensure a value is lost.
var observable = hot.Replay();
observable.Connect();
observable.Subscribe(i => Console.WriteLine("first subscription : {0}", i));
Thread.Sleep(period);
observable.Subscribe(i => Console.WriteLine("second subscription : {0}", i));
Console.ReadKey();
observable.Subscribe(i => Console.WriteLine("third subscription : {0}", i));
Console.ReadKey();
```

Output:

```
first subscription : 1
second subscription : 1
first subscription : 2
second subscription : 2
third subscription : 1
third subscription : 2
```

The *Replay* extension method has several overloads that match the *ReplaySubject<T>* constructor overloads; you are able to specify the buffer size by count or by time.

Multicast

The *PublishLast* and *Replay* methods effectively apply *AsyncSubject<T>* and *ReplaySubject<T>* functionality to the underlying observable sequence. We could attempt to build a crude implementation ourselves.

```
var period = TimeSpan.FromSeconds(1);
//var observable = Observable.Interval(period).Publish();
var observable = Observable.Interval(period);
var shared = new Subject<long>();
shared.Subscribe(i => Console.WriteLine("first subscription : {0}", i));
observable.Subscribe(shared);   //'Connect' the observable.
Thread.Sleep(period);
Thread.Sleep(period);
shared.Subscribe(i => Console.WriteLine("second subscription : {0}", i));
```

Output:

```
first subscription : 0
first subscription : 1
second subscription : 1
first subscription : 2
second subscription : 2
```

The Rx library supplies us with a great method to do this well though. You can apply subject behavior via the *Multicast* extension method. This allows you to share or "multicast" an observable sequence with the behavior of a specific subject. For example

- *.Publish() = .Multicast(new Subject<T>)*
- *.PublishLast() = .Multicast(new AsyncSubject<T>)*
- *.Replay() = .Multicast(new ReplaySubject<T>)*

Hot and cold observables are two different styles of sharing an observable sequence. Both have equally valid applications but behave in different ways. Cold observables allow you to lazily evaluate an observable sequence independently for each subscriber. Hot observables allow you to share notifications by multicasting your sequence, even if there are no subscribers. The use of *RefCount* allows you to have lazily-evaluated, multicast observable sequences, coupled with eager disposal semantics once the last subscription is disposed.

PART 4 - Concurrency

Rx is primarily a system for querying *data in motion* asynchronously. To effectively provide the level of asynchrony that developers require, some level of concurrency control is required. We need the ability to generate sequence data concurrently to the consumption of the sequence data.

In this fourth and final part of the book, we will look at the various concurrency considerations one must undertake when querying data in motion. We will look how to avoid concurrency when possible and use it correctly when justifiable. We will look at the excellent abstractions Rx provides, that enable concurrency to become declarative and also unit testable. In my opinion, theses two features are enough reason alone to adopt Rx into your code base. We will also look at the complex issue of querying concurrent sequences and analyzing data in sliding windows of time.

Scheduling and threading

So far, we have managed to avoid any explicit usage of threading or concurrency. There are some methods that we have covered that implicitly introduce some level of concurrency to perform their jobs (e.g. *Buffer*, *Delay*, *Sample* each require a separate thread/scheduler/timer to work their magic). Most of this however, has been kindly abstracted away from us. This chapter will look at the elegant beauty of the Rx API and its ability to effectively remove the need for *WaitHandle* types, and any explicit calls to *Thread*s, the *ThreadPool* or *Task*s.

Rx is single-threaded by default

A popular misconception is that Rx is multithreaded by default. It is perhaps more an idle assumption than a strong belief, much in the same way some assume that standard .NET events are multithreaded until they challenge that notion. We debunk this myth and assert that events are most certainly single threaded and synchronous in the Appendix.

Like events, Rx is just a way of chaining callbacks together for a given notification. While Rx is a free-threaded model, this does not mean that subscribing or calling `OnNext` will introduce multi-threading to your sequence. Being free-threaded means that you are not restricted to which thread you choose to do your work. For example, you can choose to do your work such as invoking a subscription, observing or producing notifications, on any thread you like. The alternative to a free-threaded model is a *Single Threaded Apartment* (STA) model where you must interact with the system on a given thread. It is common to use the STA model when working with User Interfaces and some COM interop. So, just as a recap: if you do not introduce any scheduling, your callbacks will be invoked on the same thread that the *OnNext*/*OnError*/*OnCompleted* methods are invoked from.

In this example, we create a subject then call *OnNext* on various threads and record the threadId in our handler.

```
Console.WriteLine("Starting on threadId:{0}", Thread.CurrentThread.ManagedThreadId);
var subject = new Subject<object>();
subject.Subscribe(
        o => Console.WriteLine("Received {1} on threadId:{0}",
        Thread.CurrentThread.ManagedThreadId,
        o));
ParameterizedThreadStart notify = obj =>
{
        Console.WriteLine("OnNext({1}) on threadId:{0}",
        Thread.CurrentThread.ManagedThreadId,
        obj);
        subject.OnNext(obj);
};
notify(1);
new Thread(notify).Start(2);
new Thread(notify).Start(3);
```

Output:

```
Starting on threadId:9
OnNext(1) on threadId:9
Received 1 on threadId:9
OnNext(2) on threadId:10
Received 2 on threadId:10
OnNext(3) on threadId:11
Received 3 on threadId:11
```

Note that each *OnNext* was called back on the same thread that it was notified on. This is not always what we are looking for. Rx introduces a very handy mechanism for introducing concurrency and multithreading to your code: Scheduling.

SubscribeOn and ObserveOn

In the Rx world, there are generally two things you want to control the concurrency model for:

1. The invocation of the subscription
2. The observing of notifications

As you could probably guess, these are exposed via two extension methods to *IObservable<T>* called *SubscribeOn* and *ObserveOn*. Both methods have an overload that take an *IScheduler* (or *SynchronizationContext*) and return an *IObservable<T>* so you can chain methods together.

```
public static class Observable
{
        public static IObservable<TSource> ObserveOn<TSource>(
        this IObservable<TSource> source,
        IScheduler scheduler)
        {...}
        public static IObservable<TSource> ObserveOn<TSource>(
        this IObservable<TSource> source,
        SynchronizationContext context)
        {...}
        public static IObservable<TSource> SubscribeOn<TSource>(
        this IObservable<TSource> source,
        IScheduler scheduler)
        {...}
        public static IObservable<TSource> SubscribeOn<TSource>(
        this IObservable<TSource> source,
        SynchronizationContext context)
        {...}
}
```

One pitfall I want to point out here is, the first few times I used these overloads, I was confused as to what they actually do. You should use the *SubscribeOn* method to describe how you want any warm-up and background processing code to be scheduled. For example, if you were to use *SubscribeOn* with *Observable.Create*, the delegate passed to the *Create* method would be run on the specified scheduler.

In this example, we have a sequence produced by *Observable.Create* with a standard subscription.

```
Console.WriteLine("Starting on threadId:{0}", Thread.CurrentThread.ManagedThreadId);
var source = Observable.Create<int>(
        o =>
        {
        Console.WriteLine("Invoked on threadId:{0}", Thread.CurrentThread.ManagedThreadId);
        o.OnNext(1);
        o.OnNext(2);
        o.OnNext(3);
        o.OnCompleted();
        Console.WriteLine("Finished on threadId:{0}",
        Thread.CurrentThread.ManagedThreadId);
        return Disposable.Empty;
        });
source
        //.SubscribeOn(Scheduler.ThreadPool)
        .Subscribe(
        o => Console.WriteLine("Received {1} on threadId:{0}",
        Thread.CurrentThread.ManagedThreadId,
        o),
        () => Console.WriteLine("OnCompleted on threadId:{0}",
        Thread.CurrentThread.ManagedThreadId));
Console.WriteLine("Subscribed on threadId:{0}", Thread.CurrentThread.ManagedThreadId);
```

Output:

```
Starting on threadId:9
Invoked on threadId:9
Received 1 on threadId:9
Received 2 on threadId:9
Received 3 on threadId:9
OnCompleted on threadId:9
Finished on threadId:9
Subscribed on threadId:9
```

You will notice that all actions were performed on the same thread. Also, note that everything is sequential. When the subscription is made, the *Create* delegate is called. When `OnNext(1)` is called, the *OnNext* handler is called, and so on. This all stays synchronous until the *Create* delegate is finished, and the *Subscribe* line can move on to the final line that declares we are subscribed on thread 9.

If we apply *SubscribeOn* to the chain (i.e. un-comment it), the order of execution is quite different.

```
Starting on threadId:9
Subscribed on threadId:9
Invoked on threadId:10
Received 1 on threadId:10
Received 2 on threadId:10
Received 3 on threadId:10
OnCompleted on threadId:10
Finished on threadId:10
```

Observe that the subscribe call is now non-blocking. The *Create* delegate is executed on the thread pool and so are all our handlers.

The *ObserveOn* method is used to declare where you want your notifications to be scheduled to. I would suggest the *ObserveOn* method is most useful when working with STA systems, most commonly UI applications. When writing UI applications, the *SubscribeOn*/*ObserveOn* pair is very useful for two reasons:

1. you do not want to block the UI thread
2. but you do need to update UI objects on the UI thread.

It is critical to avoid blocking the UI thread, as doing so leads to a poor user experience. General guidance for Silverlight and WPF is that any work that blocks for longer than 150-250ms should not be performed on the UI thread (Dispatcher). This is approximately the period of time over which a user can notice a lag in the UI (mouse becomes sticky, animations sluggish). In the upcoming Metro style apps for Windows 8, the maximum allowed blocking time is only 50ms. This more stringent rule is to ensure a consistent "fast and fluid" experience across applications. With the processing power offered by current desktop processors, you can achieve a lot of processing 50ms. However, as processor become more varied (single/multi/many core, plus high power desktop vs. lower power ARM tablet/phones), how much you can do in 50ms fluctuates widely. In general terms: any I/O, computational intensive work or any processing unrelated to the UI should be marshaled off the UI thread. The general pattern for creating responsive UI applications is:

- respond to some sort of user action
- do work on a background thread
- pass the result back to the UI thread
- update the UI

This is a great fit for Rx: responding to events, potentially composing multiple events, passing data to chained method calls. With the inclusion of scheduling, we even have the power to get off and back onto the UI thread for that responsive application feel that users demand.

Consider a WPF application that used Rx to populate an *ObservableCollection<T>*. You would almost certainly want to use *SubscribeOn* to leave the *Dispatcher*, followed by *ObserveOn* to ensure you were notified back on the Dispatcher. If you failed to use the *ObserveOn* method, then your *OnNext* handlers would be invoked on the same thread that raised the notification. In Silverlight/WPF, this would cause some sort of not-supported/cross-threading exception. In this example, we subscribe to a sequence of `Customers`. We perform the subscription on a new thread and

ensure that as we receive `Customer` notifications, we add them to the `Customers` collection on the *Dispatcher*.

```
_customerService.GetCustomers()
        .SubscribeOn(Scheduler.NewThread)
        .ObserveOn(DispatcherScheduler.Instance)
        //or .ObserveOnDispatcher()
        .Subscribe(Customers.Add);
```

Schedulers

The *SubscribeOn* and *ObserveOn* methods required us to pass in an *IScheduler*. Here we will dig a little deeper and see what schedulers are, and what implementations are available to us.

There are two main types we use when working with schedulers:

The *IScheduler* interface
: A common interface for all schedulers

The static *Scheduler* class
: Exposes both implementations of *IScheduler* and helpful extension methods to the *IScheduler* interface

The *IScheduler* interface is of less importance right now than the types that implement the interface. The key concept to understand is that an *IScheduler* in Rx is used to schedule some action to be performed, either as soon as possible or at a given point in the future. The implementation of the *IScheduler* defines how that action will be invoked i.e. asynchronously via a thread pool, a new thread or a message pump, or synchronously on the current thread. Depending on your platform (Silverlight 4, Silverlight 5, .NET 3.5, .NET 4.0), you will be exposed most of the implementations you will need via a static class *Scheduler*.

Before we look at the *IScheduler* interface in detail, let's look at the extension method we will use the most often and then introduce the common implementations.

This is the most commonly used (extension) method for *IScheduler*. It simply sets an action to be performed as soon as possible.

```
public static IDisposable Schedule(this IScheduler scheduler, Action action)
{...}
```

You could use the method like this:

```
IScheduler scheduler = ...;
scheduler.Schedule(()=>{ Console.WriteLine("Work to be scheduled"); });
```

These are the static properties that you can find on the *Scheduler* type.

Scheduler.Immediate will ensure the action is not scheduled, but rather executed immediately.

Scheduler.CurrentThread ensures that the actions are performed on the thread that made the original call. This is different from *Immediate*, as *CurrentThread* will queue the action to be performed. We will compare these two schedulers using a code example soon.

Scheduler.NewThread will schedule work to be done on a new thread.

Scheduler.ThreadPool will schedule all actions to take place on the Thread Pool.

Scheduler.TaskPool will schedule actions onto the TaskPool. This is not available in Silverlight 4 or .NET 3.5 builds.

If you are using WPF or Silverlight, then you will also have access to *DispatcherScheduler.Instance*. This allows you to schedule tasks onto the *Dispatcher* with the common interface, either now or in the future. There is the *SubscribeOnDispatcher()* and *ObserveOnDispatcher()* extension methods to *IObservable<T>*, that also help you access the Dispatcher. While they appear useful, you will want to avoid these two methods for production code, and we explain why in the Testing Rx chapter.

Most of the schedulers listed above are quite self explanatory for basic usage. We will take an in-depth look at all of the implementations of *IScheduler* later in the chapter.

Concurrency pitfalls

Introducing concurrency to your application will increase its complexity. If your application is not noticeably improved by adding a layer of concurrency, then you should avoid doing so. Concurrent applications can exhibit maintenance problems with symptoms surfacing in the areas of debugging, testing and refactoring.

The common problem that concurrency introduces is unpredictable timing. Unpredictable timing can be caused by variable load on a system, as well as variations in system configurations (e.g. varying core clock speed and availability of processors). These can ultimately can result in race conditions. Symptoms of race conditions include out-of-order execution, deadlocks, livelocks and corrupted state.

In my opinion, the biggest danger when introducing concurrency haphazardly to an application, is that you can silently introduce bugs. These defects may slip past Development, QA and UAT and only manifest themselves in Production environments.

Rx, however, does such a good job of simplifying the concurrent processing of observable sequences that many of these concerns can be mitigated. You can still create problems, but if you follow the guidelines then you can feel a lot safer in the knowledge that you have heavily reduced the capacity for unwanted race conditions.

In a later chapter, Testing Rx, we will look at how Rx improves your ability to test concurrent workflows.

Lock-ups

When working on my first commercial application that used Rx, the team found out the hard way that Rx code can most certainly deadlock. When you consider that some calls (like *First*, *Last*, *Single* and *ForEach*) are blocking, and that we can schedule work to be done in the future, it becomes obvious that a race condition can occur. This example is the simplest block I could think of. Admittedly, it is fairly elementary but it will get the ball rolling.

```
var sequence = new Subject<int>();
Console.WriteLine("Next line should lock the system.");
var value = sequence.First();
sequence.OnNext(1);
Console.WriteLine("I can never execute....");
```

Hopefully, we won't ever write such code though, and if we did, our tests would give us quick feedback that things went wrong. More realistically, race conditions often slip into the system at integration points. The next example may be a little harder to detect, but is only small step away from our first, unrealistic example. Here, we block in the constructor of a UI element which will always be created on the dispatcher. The blocking call is waiting for an event that can only be raised from the dispatcher, thus creating a deadlock.

```
public Window1()
{
        InitializeComponent();
        DataContext = this;
        Value = "Default value";
        //Deadlock!
        //We need the dispatcher to continue to allow me to click the button to produce a value
        Value = _subject.First();
```

```
        //This will give same result but will not be blocking (deadlocking).
        _subject.Take(1).Subscribe(value => Value = value);
}
private void MyButton_Click(object sender, RoutedEventArgs e)
{
        _subject.OnNext("New Value");
}
public string Value
{
        get { return _value; }
        set
        {
        _value = value;
        var handler = PropertyChanged;
        if (handler != null) handler(this, new PropertyChangedEventArgs("Value"));
        }
}
```

Next, we start seeing things that can become more sinister. The button's click handler will try to get the first value from an observable sequence exposed via an interface.

```
public partial class Window1 : INotifyPropertyChanged
{
        //Imagine DI here.
        private readonly IMyService _service = new MyService();
        private int _value2;
        public Window1()
        {
        InitializeComponent();
        DataContext = this;
        }
        public int Value2
        {
        get { return _value2; }
        set
        {
        _value2 = value;
        var handler = PropertyChanged;
        if (handler != null) handler(this, new PropertyChangedEventArgs("Value2"));
        }
        }
        #region INotifyPropertyChanged Members
        public event PropertyChangedEventHandler PropertyChanged;
        #endregion
        private void MyButton2_Click(object sender, RoutedEventArgs e)
        {
        Value2 = _service.GetTemperature().First();
        }
}
```

There is only one small problem here in that we block on the *Dispatcher* thread (`First` is a blocking call), however this manifests itself into a deadlock if the service code is written incorrectly.

```
class MyService : IMyService
{
        public IObservable<int> GetTemperature()
        {
        return Observable.Create<int>(
        o =>
        {
        o.OnNext(27);
        o.OnNext(26);
        o.OnNext(24);
        return () => { };
        })
        .SubscribeOnDispatcher();
        }
}
```

This odd implementation, with explicit scheduling, will cause the three `OnNext` calls to be scheduled once the `First()` call has finished; however, *that* is waiting for an *OnNext* to be called: we are deadlocked.

So far, this chapter may seem to say that concurrency is all doom and gloom by focusing on the problems you could face; this is not the intent though. We do not magically avoid classic concurrency problems simply by adopting Rx. Rx will however make it easier to get it right, provided you follow these two simple rules.

1. Only the final subscriber should be setting the scheduling
2. Avoid using blocking calls: e.g. *First*, *Last* and *Single*

The last example came unstuck with one simple problem; the service was dictating the scheduling paradigm when, really, it had no business doing so. Before we had a clear idea of where we should be doing the scheduling in my first Rx project, we had all sorts of layers adding 'helpful' scheduling

code. What it ended up creating was a threading nightmare. When we removed all the scheduling code and then confined it it in a single layer (at least in the Silverlight client), most of our concurrency problems went away. I recommend you do the same. At least in WPF/Silverlight applications, the pattern should be simple: "Subscribe on a Background thread; Observe on the Dispatcher".

Advanced features of schedulers

We have only looked at the most simple usage of schedulers so far:

- Scheduling an action to be executed as soon as possible
- Scheduling the subscription of an observable sequence
- Scheduling the observation of notifications coming from an observable sequence

Schedulers also provide more advanced features that can help you with various problems.

Passing state

In the extension method to *IScheduler* we have looked at, you could only provide an *Action* to execute. This *Action* did not accept any parameters. If you want to pass state to the *Action*, you could use a closure to share the data like this:

```
var myName = "Lee";
Scheduler.NewThread.Schedule(
        () => Console.WriteLine("myName = {0}", myName));
```

This could create a problem, as you are sharing state across two different scopes. I could modify the variable *myName* and get unexpected results.

In this example, we use a closure as above to pass state. I immediately modify the closure and this creates a race condition: will my modification happen before or after the state is used by the scheduler?

```
var myName = "Lee";
scheduler.Schedule(
        () => Console.WriteLine("myName = {0}", myName));
myName = "John";//What will get written to the console?
```

In my tests, "John" is generally written to the console when *scheduler* is an instance of *NewThreadScheduler*. If I use the *ImmediateScheduler* then "Lee" would be written. The problem with this is the non-deterministic nature of the code.

A preferable way to pass state is to use the *Schedule* overloads that accept state. This example takes advantage of this overload, giving us certainty about our state.

```
var myName = "Lee";
scheduler.Schedule(myName,
        (_, state) =>
        {
        Console.WriteLine(state);
        return Disposable.Empty;
        });
myName = "John";
```

Here, we pass *myName* as the state. We also pass a delegate that will take the state and return a disposable. The disposable is used for cancellation; we will look into that later. The delegate also takes an *IScheduler* parameter, which we name "_" (underscore). This is the convention to indicate we are ignoring the argument. When we pass *myName* as the state, a reference to the state is kept internally. So when we update the *myName* variable to "John", the reference to "Lee" is still maintained by the scheduler's internal workings.

Note that in our previous example, we modify the *myName* variable to point to a new instance of a string. If we were to instead have an instance that we actually modified, we could still get unpredictable behavior. In the next example, we now use a list for our state. After scheduling an action to print out the element count of the list, we modify that list.

```
var list = new List<int>();
scheduler.Schedule(list,
        (innerScheduler, state) =>
        {
        Console.WriteLine(state.Count);
        return Disposable.Empty;
        });
list.Add(1);
```

Now that we are modifying shared state, we can get unpredictable results. In this example, we don't even know what type the scheduler is, so we cannot predict the race conditions we are creating. As with any concurrent software, you should avoid modifying shared state.

Future scheduling

As you would expect with a type called "IScheduler", you are able to schedule an action to be executed in the future. You can do so by specifying the exact point in time an action should be invoked, or you can specify the period of time to wait until the action is invoked. This is clearly useful for features such as buffering, timers etc.

Scheduling in the future is thus made possible by two styles of overloads, one that takes a *TimeSpan* and one that takes a *DateTimeOffset*. These are the two most simple overloads that execute an action in the future.

```
public static IDisposable Schedule(
        this IScheduler scheduler,
        TimeSpan dueTime,
        Action action)
{...}
public static IDisposable Schedule(
        this IScheduler scheduler,
        DateTimeOffset dueTime,
        Action action)
{...}
```

You can use the *TimeSpan* overload like this:

```
var delay = TimeSpan.FromSeconds(1);
Console.WriteLine("Before schedule at {0:o}", DateTime.Now);
scheduler.Schedule(delay,
        () => Console.WriteLine("Inside schedule at {0:o}", DateTime.Now));
Console.WriteLine("After schedule at  {0:o}", DateTime.Now);
```

Output:

```
Before schedule at 2012-01-01T12:00:00.000000+00:00
After schedule at 2012-01-01T12:00:00.058000+00:00
Inside schedule at 2012-01-01T12:00:01.044000+00:00
```

We can see therefore that scheduling is non-blocking as the 'before' and 'after' calls are very close together in time. You can also see that approximately one second after the action was scheduled, it was invoked.

You can specify a specific point in time to schedule the task with the *DateTimeOffset* overload. If, for some reason, the point in time you specify is in the past, then the action is scheduled as soon as possible.

Cancelation

Each of the overloads to *Schedule* returns an *IDisposable*; this way, a consumer can cancel the scheduled work. In the previous example, we scheduled work to be invoked in one second. We could cancel that work by disposing of the cancellation token (i.e. the return value).

```
var delay = TimeSpan.FromSeconds(1);
Console.WriteLine("Before schedule at {0:o}", DateTime.Now);
var token = scheduler.Schedule(delay,
        () => Console.WriteLine("Inside schedule at {0:o}", DateTime.Now));
Console.WriteLine("After schedule at  {0:o}", DateTime.Now);
token.Dispose();
```

Output:

```
Before schedule at 2012-01-01T12:00:00.000000+00:00
After schedule at 2012-01-01T12:00:00.058000+00:00
```

Note that the scheduled action never occurs, as we have cancelled it almost immediately.

When the user cancels the scheduled action method before the scheduler is able to invoke it, that action is just removed from the queue of work. This is what we see in example above. If you want to cancel scheduled work that is already running, then you can use one of the overloads to the *Schedule* method that takes a *Func<IDisposable>*. This gives a way for users to cancel out of a job that may already be running. This job could be some sort of I/O, heavy computations or perhaps usage of *Task* to perform some work.

Now this may create a problem; if you want to cancel work that has already been started, you need to dispose of an instance of *IDisposable*, but how do you return the disposable if you are still doing the work? You could fire up another thread so the work happens concurrently, but creating threads is something we are trying to steer away from.

In this example, we have a method that we will use as the delegate to be scheduled. It just fakes some work by performing a spin wait and adding values to the *list* argument. The key here is that we allow the user to cancel with the *CancellationToken* via the disposable we return.

```
public IDisposable Work(IScheduler scheduler, List<int> list)
{
        var tokenSource = new CancellationTokenSource();
        var cancelToken = tokenSource.Token;
        var task = new Task(() =>
        {
        Console.WriteLine();
        for (int i = 0; i < 1000; i++)
        {
        var sw = new SpinWait();
        for (int j = 0; j < 3000; j++) sw.SpinOnce();
        Console.Write(".");
        list.Add(i);
        if (cancelToken.IsCancellationRequested)
        {
        Console.WriteLine("Cancelation requested");
        //cancelToken.ThrowIfCancellationRequested();
        return;
        }
        }
        }, cancelToken);
        task.Start();
        return Disposable.Create(tokenSource.Cancel);
}
```

This code schedules the above code and allows the user to cancel the processing work by pressing Enter

```
var list = new List<int>();
Console.WriteLine("Enter to quit:");
var token = scheduler.Schedule(list, Work);
Console.ReadLine();
Console.WriteLine("Cancelling...");
token.Dispose();
Console.WriteLine("Cancelled");
```

Output:

```
Enter to quit:
```

```
........
Cancelling...
Cancelled
Cancelation requested
```

The problem here is that we have introduced explicit use of *Task*. We can avoid explicit usage of a concurrency model if we use the Rx recursive scheduler features instead.

Recursion

The more advanced overloads of *Schedule* extension methods take some strange looking delegates as parameters. Take special note of the final parameter in each of these overloads of the *Schedule* extension method.

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

Each of these overloads take a delegate "action" that allows you to call "action" recursively. This may seem a very odd signature, but it makes for a great API. This effectively allows you to create a recursive delegate call. This may be best shown with an example.

This example uses the most simple recursive overload. We have an Action that can be called recursively.

```
Action<Action> work = (Action self)
        =>
        {
        Console.WriteLine("Running");
        self();
        };
var token = s.Schedule(work);
Console.ReadLine();
Console.WriteLine("Cancelling");
token.Dispose();
Console.WriteLine("Cancelled");
```

Output:

```
Enter to quit:
Running
Running
Running
Running
Cancelling
Cancelled
Running
```

Note that we didn't have to write any cancellation code in our delegate. Rx handled the looping and checked for cancellation on our behalf. Brilliant! Unlike simple recursive methods in C#, we are also protected from stack overflows, as Rx provides an extra level of abstraction. Indeed, Rx takes our recursive method and transforms it to a loop structure instead.

Creating your own iterator

Earlier in the book, we looked at how we can use Rx with APM. In our example, we just read the entire file into memory. We also referenced Jeffery van Gogh's blog post, which sadly is now out of date; however, his concepts are still sound. Instead of the Iterator method from Jeffery's post, we can use schedulers to achieve the same result.

The goal of the following sample is to open a file and stream it in chunks. This enables us to work with files that are larger than the memory available to us, as we would only ever read and cache a portion of the file at a time. In addition to this, we can leverage the compositional nature of Rx to apply multiple transformations to the file such as encryption and compression. By reading chunks at a time, we are able to start the other transformations before we have finished reading the file.

First, let us refresh our memory with how to get from the *FileStream*'s APM methods into Rx.

```
var source = new FileStream(@"C:\Somefile.txt", FileMode.Open, FileAccess.Read);
var factory = Observable.FromAsyncPattern<byte[], int, int, int>(
        source.BeginRead,
        source.EndRead);
var buffer = new byte[source.Length];
IObservable<int> reader = factory(buffer, 0, (int)source.Length);
reader.Subscribe(
        bytesRead =>
        Console.WriteLine("Read {0} bytes from file into buffer", bytesRead));
```

The example above uses *FromAsyncPattern* to create a factory. The factory will take a byte array (buffer), an offset (0) and a length (source.Length); it effectively returns the count of the bytes read as a single-value sequence. When the sequence (reader) is subscribed to, *BeginRead* will read values, starting from the offset, into the buffer. In this case, we will read the whole file. Once the file has been read into the buffer, the sequence (reader) will push the single value (bytesRead) in to the sequence.

This is all fine, but if we want to read chunks of data at a time then this is not good enough. We need to specify the buffer size we want to use. Let's start with 4KB (4096 bytes).

```
var bufferSize = 4096;
var buffer = new byte[bufferSize];
IObservable<int> reader = factory(buffer, 0, bufferSize);
reader.Subscribe(
        bytesRead =>
        Console.WriteLine("Read {0} bytes from file", bytesRead));
```

This works but will only read a max of 4KB from the file. If the file is larger, we want to keep reading all of it. As the *Position* of the *FileStream* will have advanced to the point it stopped reading, we can reuse the *factory* to reload the buffer. Next, we want to start pushing these bytes into an observable sequence. Let's start by creating the signature of an extension method.

```
public static IObservable<byte> ToObservable(
        this FileStream source,
        int buffersize,
        IScheduler scheduler)
{...}
```

We can ensure that our extension method is lazily evaluated by using *Observable.Create*. We can also ensure that the *FileStream* is closed when the consumer disposes of the subscription by taking advantage of the *Observable.Using* operator.

```
public static IObservable<byte> ToObservable(
        this FileStream source,
        int buffersize,
        IScheduler scheduler)
{
        var bytes = Observable.Create<byte>(o =>
        {
        ...
        });
        return Observable.Using(() => source, _ => bytes);
}
```

Next, we want to leverage the scheduler's recursive functionality to continuously read chunks of data while still providing the user with the ability to dispose/cancel when they choose. This creates a bit of a pickle; we can only pass in one state parameter but need to manage multiple moving parts (buffer, factory, filestream). To do this, we create our own private helper class:

```
private sealed class StreamReaderState
{
        private readonly int _bufferSize;
        private readonly Func<byte[], int, int, IObservable<int>> _factory;
        public StreamReaderState(FileStream source, int bufferSize)
        {
        _bufferSize = bufferSize;
        _factory = Observable.FromAsyncPattern<byte[], int, int, int>(
        source.BeginRead,
        source.EndRead);
        Buffer = new byte[bufferSize];
        }
        public IObservable<int> ReadNext()
        {
        return _factory(Buffer, 0, _bufferSize);
        }
        public byte[] Buffer { get; set; }
}
```

This class will allow us to read data into a buffer, then read the next chunk by calling `ReadNext()`. In our *Observable.Create* delegate, we instantiate our helper class and use it to push the buffer into our observable sequence.

```
public static IObservable<byte> ToObservable(
        this FileStream source,
        int buffersize,
        IScheduler scheduler)
{
        var bytes = Observable.Create<byte>(o =>
        {
        var initialState = new StreamReaderState(source, buffersize);
        initialState
        .ReadNext()
        .Subscribe(bytesRead =>
        {
        for (int i = 0; i < bytesRead; i++)
        {
                o.OnNext(initialState.Buffer[i]);
        }
        });
        ...
        });
        return Observable.Using(() => source, _ => bytes);
}
```

So this gets us off the ground, but we are still do not support reading files larger than the buffer. Now, we need to add recursive scheduling. To do this, we need a delegate to fit the required signature. We will need one that accepts a *StreamReaderState* and can recursively call an *Action<StreamReaderState>*.

```
public static IObservable<byte> ToObservable(
        this FileStream source,
        int buffersize,
        IScheduler scheduler)
{
        var bytes = Observable.Create<byte>(o =>
        {
        var initialState = new StreamReaderState(source, buffersize);
        Action<StreamReaderState, Action<StreamReaderState>> iterator;
        iterator = (state, self) =>
        {
        state.ReadNext()
        .Subscribe(bytesRead =>
                {
                for (int i = 0; i < bytesRead; i++)
                {
                o.OnNext(state.Buffer[i]);
                }
                self(state);
                });
        };
        return scheduler.Schedule(initialState, iterator);
        });
        return Observable.Using(() => source, _ => bytes);
}
```

We now have an *iterator* action that will:

1. call *ReadNext()*
2. subscribe to the result
3. push the buffer into the observable sequence
4. and recursively call itself.

We also schedule this recursive action to be called on the provided scheduler. Next, we want to complete the sequence when we get to the end of the file. This is easy, we maintain the recursion until the *bytesRead* is 0.

```
public static IObservable<byte> ToObservable(
        this FileStream source,
        int buffersize,
        IScheduler scheduler)
{
        var bytes = Observable.Create<byte>(o =>
        {
        var initialState = new StreamReaderState(source, buffersize);
        Action<StreamReaderState, Action<StreamReaderState>> iterator;
        iterator = (state, self) =>
        {
        state.ReadNext()
        .Subscribe(bytesRead =>
                {
                for (int i = 0; i < bytesRead; i++)
                {
                o.OnNext(state.Buffer[i]);
                }
                if (bytesRead > 0)
                self(state);
                else
                o.OnCompleted();
                });
        };
        return scheduler.Schedule(initialState, iterator);
        });
        return Observable.Using(() => source, _ => bytes);
}
```

At this point, we have an extension method that iterates on the bytes from a file stream. Finally, let us apply some clean up so that we correctly manage our resources and exceptions, and the finished method looks something like this:

```
public static IObservable<byte> ToObservable(
        this FileStream source,
        int buffersize,
        IScheduler scheduler)
{
        var bytes = Observable.Create<byte>(o =>
        {
        var initialState = new StreamReaderState(source, buffersize);
        var currentStateSubscription = new SerialDisposable();
        Action<StreamReaderState, Action<StreamReaderState>> iterator =
        (state, self) =>
        currentStateSubscription.Disposable = state.ReadNext()
        .Subscribe(
                bytesRead =>
                {
                for (int i = 0; i < bytesRead; i++)
                {
                o.OnNext(state.Buffer[i]);
                }
                if (bytesRead > 0)
                self(state);
                else
                o.OnCompleted();
                },
                o.OnError);
        var scheduledWork = scheduler.Schedule(initialState, iterator);
        return new CompositeDisposable(currentStateSubscription, scheduledWork);
        });
        return Observable.Using(() => source, _ => bytes);
}
```

This is example code and your mileage may vary. I find that increasing the buffer size and returning *IObservable<IList<byte>>* suits me better, but the example above works fine too. The goal here was to provide an example of an iterator that provides concurrent I/O access with cancellation and resource-efficient buffering.

Combinations of scheduler features

We have discussed many features that you can use with the *IScheduler* interface. Most of these examples, however, are actually using extension methods to invoke the functionality that we are looking for. The interface itself exposes the richest overloads. The extension methods are effectively just making a trade-off; improving usability/discoverability by reducing the richness of the overload. If you want access to passing state, cancellation, future scheduling and recursion, it is all available directly from the interface methods.

```
namespace System.Reactive.Concurrency
{
        public interface IScheduler
        {
        //Gets the scheduler's notion of current time.
        DateTimeOffset Now { get; }
```

```
        // Schedules an action to be executed with given state.
        //  Returns a disposable object used to cancel the scheduled action (best effort).
        IDisposable Schedule<TState>(
        TState state,
        Func<IScheduler, TState, IDisposable> action);
        // Schedules an action to be executed after dueTime with given state.
        //  Returns a disposable object used to cancel the scheduled action (best effort).
        IDisposable Schedule<TState>(
        TState state,
        TimeSpan dueTime,
        Func<IScheduler, TState, IDisposable> action);
        //Schedules an action to be executed at dueTime with given state.
        //  Returns a disposable object used to cancel the scheduled action (best effort).
        IDisposable Schedule<TState>(
        TState state,
        DateTimeOffset dueTime,
        Func<IScheduler, TState, IDisposable> action);
        }
}
```

Schedulers in-depth

We have largely been concerned with the abstract concept of a scheduler and the *IScheduler* interface. This abstraction allows low-level plumbing to remain agnostic towards the implementation of the concurrency model. As in the file reader example above, there was no need for the code to know which implementation of *IScheduler* was passed, as this is a concern of the consuming code.

Now we take an in-depth look at each implementation of *IScheduler*, consider the benefits and tradeoffs they each make, and when each is appropriate to use.

ImmediateScheduler

The *ImmediateScheduler* is exposed via the *Scheduler.Immediate* static property. This is the most simple of schedulers as it does not actually schedule anything. If you call *Schedule(Action)* then it will just invoke the action. If you schedule the action to be invoked in the future, the *ImmediateScheduler* will invoke a *Thread.Sleep* for the given period of time and then execute the action. In summary, the *ImmediateScheduler* is synchronous.

CurrentThreadScheduler

Like the *ImmediateScheduler*, the *CurrentThreadScheduler* is single-threaded. It is exposed via the *Scheduler.Current* static property. The key difference is that the *CurrentThreadScheduler* acts like a message queue or a *Trampoline*. If you schedule an action that itself schedules an action, the *CurrentThreadScheduler* will queue the inner action to be performed later; in contrast, the *ImmediateScheduler* would start working on the inner action straight away. This is probably best explained with an example.

In this example, we analyze how *ImmediateScheduler* and *CurrentThreadScheduler* perform nested scheduling differently.

```
private static void ScheduleTasks(IScheduler scheduler)
{
        Action leafAction = () => Console.WriteLine("----leafAction.");
        Action innerAction = () =>
        {
        Console.WriteLine("--innerAction start.");
        scheduler.Schedule(leafAction);
        Console.WriteLine("--innerAction end.");
        };
        Action outerAction = () =>
        {
        Console.WriteLine("outer start.");
        scheduler.Schedule(innerAction);
        Console.WriteLine("outer end.");
        };
        scheduler.Schedule(outerAction);
}
public void CurrentThreadExample()
{
        ScheduleTasks(Scheduler.CurrentThread);
        /*Output:
        outer start.
        outer end.
        --innerAction start.
        --innerAction end.
        ----leafAction.
        */
}
public void ImmediateExample()
{
        ScheduleTasks(Scheduler.Immediate);
        /*Output:
        outer start.
        --innerAction start.
        ----leafAction.
        --innerAction end.
        outer end.
        */
}
```

Note how the *ImmediateScheduler* does not really "schedule" anything at all, all work is performed

immediately (synchronously). As soon as *Schedule* is called with a delegate, that delegate is invoked. The *CurrentThreadScheduler*, however, invokes the first delegate, and, when nested delegates are scheduled, queues them to be invoked later. Once the initial delegate is complete, the queue is checked for any remaining delegates (i.e. nested calls to *Schedule*) and they are invoked. The difference here is quite important as you can potentially get out-of-order execution, unexpected blocking, or even deadlocks by using the wrong one.

DispatcherScheduler

The *DispatcherScheduler* is found in *System.Reactive.Window.Threading.dll* (for WPF, Silverlight 4 and Silverlight 5). When actions are scheduled using the *DispatcherScheduler*, they are effectively marshaled to the *Dispatcher*' s *BeginInvoke* method. This will add the action to the end of the dispatcher's *Normal* priority queue of work. This provides similar queuing semantics to the *CurrentThreadScheduler* for nested calls to *Schedule*.

When an action is scheduled for future work, then a *DispatcherTimer* is created with a matching interval. The callback for the timer's tick will stop the timer and re-schedule the work onto the *DispatcherScheduler*. If the *DispatcherScheduler* determines that the *dueTime* is actually not in the future then no timer is created, and the action will just be scheduled normally.

I would like to highlight a hazard of using the *DispatcherScheduler*. You can construct your own instance of a *DispatcherScheduler* by passing in a reference to a *Dispatcher*. The alternative way is to use the static property *DispatcherScheduler.Instance*. This can introduce hard to understand problems if it is not used properly. The static property does not return a reference to a static field, but creates a new instance each time, with the static property *Dispatcher.CurrentDispatcher* as the constructor argument. If you access *Dispatcher.CurrentDispatcher* from a thread that is not the UI thread, it will thus give you a new instance of a *Dispatcher*, but it will not be the instance you were hoping for.

For example, imagine that we have a WPF application with an *Observable.Create* method. In the delegate that we pass to *Observable.Create*, we want to schedule the notifications on the dispatcher. We think this is a good idea because any consumers of the sequence would get the notifications on the dispatcher for free.

```
var fileLines = Observable.Create<string>(
        o =>
        {
        var dScheduler = DispatcherScheduler.Instance;
        var lines = File.ReadAllLines(filePath);
        foreach (var line in lines)
        {
        var localLine = line;
        dScheduler.Schedule(
        () => o.OnNext(localLine));
        }
        return Disposable.Empty;
        });
```

This code may intuitively seem correct, but actually takes away power from consumers of the sequence. When we subscribe to the sequence, we decide that reading a file on the UI thread is a bad idea. So we add in a `SubscribeOn(Scheduler.NewThread)` to the chain as below:

```
fileLines
        .SubscribeOn(Scheduler.ThreadPool)
        .Subscribe(line => Lines.Add(line));
```

This causes the create delegate to be executed on a new thread. The delegate will read the file then get an instance of a *DispatcherScheduler*. The *DispatcherScheduler* tries to get the *Dispatcher* for the current thread, but we are no longer on the UI thread, so there isn't one. As such, it creates a new dispatcher that is used for the *DispatcherScheduler* instance. We schedule some work (the notifications), but, as the underlying *Dispatcher* has not been run, nothing happens; we do not even get an exception. I have seen this on a commercial project and it left quite a few people scratching their heads.

This takes us to one of our guidelines regarding scheduling: "the use of *SubscribeOn* and *ObserveOn* should only be invoked by the final subscriber". If you introduce scheduling in your own extension methods or service methods, you should allow the consumer to specify their own scheduler. We will see more reasons for this guidance in the next chapter.

EventLoopScheduler

The *EventLoopScheduler* allows you to designate a specific thread to a scheduler. Like the *CurrentThreadScheduler* that acts like a trampoline for nested scheduled actions, the *EventLoopScheduler* provides the same trampoline mechanism. The difference is that you provide an *EventLoopScheduler* with the thread you want it to use for scheduling instead, of just picking up the current thread.

The *EventLoopScheduler* can be created with an empty constructor, or you can pass it a thread factory delegate.

```
// Creates an object that schedules units of work on a designated thread.
public EventLoopScheduler()
{...}
// Creates an object that schedules units of work on a designated thread created by the
//  provided factory function.
public EventLoopScheduler(Func<ThreadStart, Thread> threadFactory)
{...}
```

The overload that allows you to pass a factory enables you to customize the thread before it is assigned to the *EventLoopScheduler*. For example, you can set the thread name, priority, culture and most importantly whether the thread is a background thread or not. Remember that if you do not set the thread's property *IsBackground* to false, then your application will not terminate until it the thread is terminated. The *EventLoopScheduler* implements *IDisposable*, and calling Dispose will allow the thread to terminate. As with any implementation of *IDisposable*, it is appropriate that you explicitly manage the lifetime of the resources you create.

This can work nicely with the *Observable.Using* method, if you are so inclined. This allows you to bind the lifetime of your *EventLoopScheduler* to that of an observable sequence - for example, this `GetPrices` method that takes an *IScheduler* for an argument and returns an observable sequence.

```
private IObservable<Price> GetPrices(IScheduler scheduler)
{...}
```

Here we bind the lifetime of the *EventLoopScheduler* to that of the result from the `GetPrices` method.

```
Observable.Using(()=>new EventLoopScheduler(), els=> GetPrices(els))
        .Subscribe(...)
```

New Thread

If you do not wish to manage the resources of a thread or an *EventLoopScheduler*, then you can use *NewThreadScheduler*. You can create your own instance of *NewThreadScheduler* or get access to the static instance via the property *Scheduler.NewThread*. Like *EventLoopScheduler*, you can use the parameterless constructor or provide your own thread factory function. If you do provide your own factory, be careful to set the *IsBackground* property appropriately.

When you call *Schedule* on the *NewThreadScheduler*, you are actually creating an *EventLoopScheduler* under the covers. This way, any nested scheduling will happen on the same thread. Subsequent (non-nested) calls to *Schedule* will create a new *EventLoopScheduler* and call the thread factory function for a new thread too.

In this example we run a piece of code reminiscent of our comparison between *Immediate* and *Current* schedulers. The difference here, however, is that we track the `ThreadId` that the action is performed on. We use the *Schedule* overload that allows us to pass the Scheduler instance into our nested delegates. This allows us to correctly nest calls.

```
private static IDisposable OuterAction(IScheduler scheduler, string state)
{
        Console.WriteLine("{0} start. ThreadId:{1}",
        state,
        Thread.CurrentThread.ManagedThreadId);
        scheduler.Schedule(state + ".inner", InnerAction);
        Console.WriteLine("{0} end. ThreadId:{1}",
        state,
        Thread.CurrentThread.ManagedThreadId);
        return Disposable.Empty;
}
private static IDisposable InnerAction(IScheduler scheduler, string state)
{
        Console.WriteLine("{0} start. ThreadId:{1}",
        state,
        Thread.CurrentThread.ManagedThreadId);
        scheduler.Schedule(state + ".Leaf", LeafAction);
        Console.WriteLine("{0} end. ThreadId:{1}",
        state,
        Thread.CurrentThread.ManagedThreadId);
        return Disposable.Empty;
}
private static IDisposable LeafAction(IScheduler scheduler, string state)
{
        Console.WriteLine("{0}. ThreadId:{1}",
        state,
        Thread.CurrentThread.ManagedThreadId);
        return Disposable.Empty;
}
```

When executed with the *NewThreadScheduler* like this:

```
Console.WriteLine("Starting on thread :{0}",
        Thread.CurrentThread.ManagedThreadId);
Scheduler.NewThread.Schedule("A", OuterAction);
```

Output:

```
Starting on thread :9
A start. ThreadId:10
A end. ThreadId:10
A.inner start . ThreadId:10
A.inner end. ThreadId:10
A.inner.Leaf. ThreadId:10
```

As you can see, the results are very similar to the *CurrentThreadScheduler*, except that the trampoline happens on a separate thread. This is in fact exactly the output we would get if we used an *EventLoopScheduler*. The differences between usages of the *EventLoopScheduler* and the *NewThreadScheduler* start to appear when we introduce a second (non-nested) scheduled task.

```
Console.WriteLine("Starting on thread :{0}",
        Thread.CurrentThread.ManagedThreadId);
Scheduler.NewThread.Schedule("A", OuterAction);
Scheduler.NewThread.Schedule("B", OuterAction);
```

Output:

```
Starting on thread :9
A start. ThreadId:10
A end. ThreadId:10
```

```
A.inner start . ThreadId:10
A.inner end. ThreadId:10
A.inner.Leaf. ThreadId:10
B start. ThreadId:11
B end. ThreadId:11
B.inner start . ThreadId:11
B.inner end. ThreadId:11
B.inner.Leaf. ThreadId:11
```

Note that there are now three threads at play here. Thread 9 is the thread we started on and threads 10 and 11 are performing the work for our two calls to Schedule.

Thread Pool

The *ThreadPoolScheduler* will simply just tunnel requests to the *ThreadPool*. For requests that are scheduled as soon as possible, the action is just sent to *ThreadPool.QueueUserWorkItem*. For requests that are scheduled in the future, a *System.Threading.Timer* is used.

As all actions are sent to the *ThreadPool*, actions can potentially run out of order. Unlike the previous schedulers we have looked at, nested calls are not guaranteed to be processed serially. We can see this by running the same test as above but with the *ThreadPoolScheduler*.

```
Console.WriteLine("Starting on thread :{0}",
        Thread.CurrentThread.ManagedThreadId);
Scheduler.ThreadPool.Schedule("A", OuterAction);
Scheduler.ThreadPool.Schedule("B", OuterAction);
```

The output

```
Starting on thread :9
A start. ThreadId:10
A end. ThreadId:10
A.inner start . ThreadId:10
A.inner end. ThreadId:10
A.inner.Leaf. ThreadId:10
B start. ThreadId:11
B end. ThreadId:11
B.inner start . ThreadId:10
B.inner end. ThreadId:10
B.inner.Leaf. ThreadId:11
```

Note, that as per the *NewThreadScheduler* test, we initially start on one thread but all the scheduling happens on two other threads. The difference is that we can see that part of the second run "B" runs on thread 11 while another part of it runs on 10.

TaskPool

The *TaskPoolScheduler* is very similar to the *ThreadPoolScheduler* and, when available (depending on your target framework), you should favor it over the later. Like the *ThreadPoolScheduler*, nested scheduled actions are not guaranteed to be run on the same thread. Running the same test with the *TaskPoolScheduler* shows us similar results.

```
Console.WriteLine("Starting on thread :{0}",
        Thread.CurrentThread.ManagedThreadId);
Scheduler.TaskPool.Schedule("A", OuterAction);
Scheduler.TaskPool.Schedule("B", OuterAction);
```

Output:

```
Starting on thread :9
A start. ThreadId:10
A end. ThreadId:10
B start. ThreadId:11
B end. ThreadId:11
A.inner start . ThreadId:10
A.inner end. ThreadId:10
A.inner.Leaf. ThreadId:10
B.inner start . ThreadId:11
B.inner end. ThreadId:11
B.inner.Leaf. ThreadId:10
```

TestScheduler

It is worth noting that there is also a *TestScheduler* accompanied by its base classes *VirtualTimeScheduler* and *VirtualTimeSchedulerBase*. The latter two are not really in the scope of an introduction to Rx, but the former is. We will cover all things testing including the *TestScheduler* in the next chapter, Testing Rx.

Selecting an appropriate scheduler

With all of these options to choose from, it can be hard to know which scheduler to use and when. Here is a simple check list to help you in this daunting task:

UI Applications

- The final subscriber is normally the presentation layer and should control the scheduling.
- Observe on the *DispatcherScheduler* to allow updating of ViewModels
- Subscribe on a background thread to prevent the UI from becoming unresponsive
  - If the subscription will not block for more than 50ms then
    - Use the *TaskPoolScheduler* if available, or
    - Use the *ThreadPoolScheduler*
  - If any part of the subscription could block for longer than 50ms, then you should use the *NewThreadScheduler*.

Service layer

- If your service is reading data from a queue of some sort, consider using a dedicated *EventLoopScheduler*. This way, you can preserve order of events
- If processing an item is expensive (>50ms or requires I/O), then consider using a *NewThreadScheduler*
- If you just need the scheduler for a timer, e.g. for *Observable.Interval* or *Observable.Timer*, then favor the *TaskPool*. Use the *ThreadPool* if the *TaskPool* is not available for your platform.

The *ThreadPool* (and the *TaskPool* by proxy) have a time delay before they will increase the number of threads that they use. This delay is 500ms. Let us consider a PC with two cores that we will schedule four actions onto. By default, the thread pool size will be the number of cores (2). If each action takes 1000ms, then two actions will be sitting in the queue for 500ms before the thread pool size is increased. Instead of running all four actions in parallel, which would take one second in total, the work is not completed for 1.5 seconds as two of the actions sat in the queue for 500ms. For this reason, you should only schedule work that is very fast to execute (guideline 50ms) onto the ThreadPool or TaskPool. Conversely, creating a new thread is not free, but with the power of processors today the creation of a thread for work over 50ms is a small cost.

Concurrency is hard. We can choose to make our life easier by taking advantage of Rx and its scheduling features. We can improve it even further by only using Rx where appropriate. While Rx has concurrency features, these should not be mistaken for a concurrency framework. Rx is designed for querying data, and as discussed in the first chapter, parallel computations or composition of asynchronous methods is more appropriate for other frameworks.

Rx solves the issues for concurrently generating and consuming data via the *ObserveOn*/*SubscribeOn* methods. By using these appropriately, we can simplify our code base, increase responsiveness and reduce the surface area of our concurrency concerns. Schedulers provide a rich platform for processing work concurrently without the need to be exposed directly to threading primitives. They also help with common troublesome areas of concurrency such as cancellation, passing state and recursion. By reducing the concurrency surface area, Rx provides a (relatively) simple yet powerful set of concurrency features paving the way to the pit of success.

Testing Rx

Testing software has its roots in debugging and demonstrating code. Having largely matured past manual tests that try to "break the application", modern quality assurance standards demand a level of automation that can help evaluate and prevent bugs. While teams of testing specialists are common, more and more coders are expected to provide quality guarantees via automated test suites.

Up to this point, we have covered a broad scope of Rx, and we have almost enough knowledge to start using Rx in anger! Still, many developers would not dream of coding without first being able to write tests. Tests can be used to prove that code is in fact satisfying requirements, provide a safety net against regression and can even help document the code. This chapter makes the assumption that you are familiar with the concepts of dependency injection and unit testing with test-doubles, such as mocks or stubs.

Rx poses some interesting problems to our Test-Driven community:

- Scheduling, and therefore threading, is generally avoided in test scenarios as it can introduce race conditions which may lead to non-deterministic tests
- Tests should run as fast as possible
- For many, Rx is a new technology/library. Naturally, as we progress on our journey to mastering Rx, we may want to refactor some of our previous Rx code. We want to use tests to ensure that our refactoring has not altered the internal behavior of our code base
- Likewise, tests will ensure nothing breaks when we upgrade versions of Rx.

While we do want to test our code, we don't want to introduce slow or non-deterministic tests; indeed, the later would introduce false-negatives or false-positives. If we look at the Rx library, there are plenty of methods that involve scheduling (implicitly or explicitly), so using Rx effectively makes it hard to avoid scheduling. This LINQ query shows us that there are at least 26 extension methods that accept an *IScheduler* as a parameter.

```
var query = from method in typeof(Observable).GetMethods()
        from parameter in method.GetParameters()
        where typeof (IScheduler).IsAssignableFrom(parameter.ParameterType)
        group method by method.Name into m
        orderby m.Key
        select m.Key;
foreach (var methodName in query)
{
        Console.WriteLine(methodName);
}
```

Output:

```
Buffer
Delay
Empty
Generate
Interval
Merge
ObserveOn
Range
Repeat
Replay
Return
Sample
Start
StartWith
Subscribe
SubscribeOn
Take
Throttle
Throw
TimeInterval
Timeout
Timer
Timestamp
```

```
ToAsync
ToObservable
Window
```

Many of these methods also have an overload that does not take an *IScheduler* and instead uses a default instance. TDD/Test First coders will want to opt for the overload that accepts the *IScheduler*, so that they can have some control over scheduling in our tests. I will explain why soon.

Consider this example, where we create a sequence that publishes values every second for five seconds.

```
var interval = Observable
        .Interval(TimeSpan.FromSeconds(1))
        .Take(5);
```

If we were to write a test that ensured that we received five values and they were each one second apart, it would take five seconds to run. That would be no good; I want hundreds if not thousands of tests to run in five seconds. Another very common requirement is to test a timeout. Here, we try to test a timeout of one minute.

```
var never = Observable.Never<int>();
var exceptionThrown = false;
never.Timeout(TimeSpan.FromMinutes(1))
        .Subscribe(
        i => Console.WriteLine("This will never run."),
        ex => exceptionThrown = true);
Assert.IsTrue(exceptionThrown);
```

We have two problems here:

1. either the *Assert* runs too soon, and the test is pointless as it always fails, or
2. we have to add a delay of one minute to perform an accurate test

For this test to be useful, it would therefore take one minute to run. Unit tests that take one minute to run are not acceptable.

TestScheduler

To our rescue comes the *TestScheduler*; it introduces the concept of a virtual scheduler to allow us to emulate and control time.

A virtual scheduler can be conceptualized as a queue of actions to be executed. Each are assigned a point in time when they should be executed. We use the *TestScheduler* as a substitute, or test double, for the production *IScheduler* types. Using this virtual scheduler, we can either execute all queued actions, or only those up to a specified point in time.

In this example, we schedule a task onto the queue to be run immediately by using the simple overload (*Schedule(Action)*). We then advance the virtual clock forward by one tick. By doing so, we execute everything scheduled up to that point in time. Note that even though we schedule an action to be executed immediately, it will not actually be executed until the clock is manually advanced.

```
var scheduler = new TestScheduler();
var wasExecuted = false;
scheduler.Schedule(() => wasExecuted = true);
Assert.IsFalse(wasExecuted);
scheduler.AdvanceBy(1); //execute 1 tick of queued actions
Assert.IsTrue(wasExecuted);
```

Running and debugging this example may help you to better understand the basics of the *TestScheduler*.

The *TestScheduler* implements the *IScheduler* interface (naturally) and also extends it to allow us to control and monitor virtual time. We are already familiar with the *IScheduler.Schedule* methods, however the *AdvanceBy(long)*, *AdvanceTo(long)* and *Start()* methods unique to the *TestScheduler* are of most interest. Likewise, the *Clock* property will also be of interest, as it can help us understand what is happening internally.

```
public class TestScheduler : ...
{
        //Implementation of IScheduler
        public DateTimeOffset Now { get; }
        public IDisposable Schedule<TState>(
        TState state,
        Func<IScheduler, TState, IDisposable> action)
        public IDisposable Schedule<TState>(
        TState state,
        TimeSpan dueTime,
        Func<IScheduler, TState, IDisposable> action)
        public IDisposable Schedule<TState>(
        TState state,
        DateTimeOffset dueTime,
        Func<IScheduler, TState, IDisposable> action)
        //Useful extensions for testing
        public bool IsEnabled { get; private set; }
        public TAbsolute Clock { get; protected set; }
        public void Start()
        public void Stop()
        public void AdvanceTo(long time)
        public void AdvanceBy(long time)
        //Other methods
        ...
}
```

AdvanceTo

The *AdvanceTo(long)* method will execute all the actions that have been scheduled up to the absolute time specified. The *TestScheduler* uses ticks as its measurement of time. In this example, we schedule actions to be invoked now, in 10 ticks, and in 20 ticks.

```
var scheduler = new TestScheduler();
scheduler.Schedule(() => Console.WriteLine("A")); //Schedule immediately
scheduler.Schedule(TimeSpan.FromTicks(10), () => Console.WriteLine("B"));
scheduler.Schedule(TimeSpan.FromTicks(20), () => Console.WriteLine("C"));
Console.WriteLine("scheduler.AdvanceTo(1);");
scheduler.AdvanceTo(1);
Console.WriteLine("scheduler.AdvanceTo(10);");
scheduler.AdvanceTo(10);
Console.WriteLine("scheduler.AdvanceTo(15);");
scheduler.AdvanceTo(15);
Console.WriteLine("scheduler.AdvanceTo(20);");
scheduler.AdvanceTo(20);
```

Output:

```
scheduler.AdvanceTo(1);
A
scheduler.AdvanceTo(10);
B
scheduler.AdvanceTo(15);
scheduler.AdvanceTo(20);
C
```

Note that nothing happened when we advanced to 15 ticks. All work scheduled before 15 ticks had been performed and we had not advanced far enough yet to get to the next scheduled action.

AdvanceBy

The *AdvanceBy(long)* method allows us to move the clock forward a relative amount of time. Again, the measurements are in ticks. We can take the last example and modify it to use *AdvanceBy(long)*.

```
var scheduler = new TestScheduler();
scheduler.Schedule(() => Console.WriteLine("A")); //Schedule immediately
scheduler.Schedule(TimeSpan.FromTicks(10), () => Console.WriteLine("B"));
scheduler.Schedule(TimeSpan.FromTicks(20), () => Console.WriteLine("C"));
Console.WriteLine("scheduler.AdvanceBy(1);");
scheduler.AdvanceBy(1);
Console.WriteLine("scheduler.AdvanceBy(9);");
scheduler.AdvanceBy(9);
Console.WriteLine("scheduler.AdvanceBy(5);");
scheduler.AdvanceBy(5);
Console.WriteLine("scheduler.AdvanceBy(5);");
scheduler.AdvanceBy(5);
```

Output:

```
scheduler.AdvanceBy(1);
A
scheduler.AdvanceBy(9);
B
scheduler.AdvanceBy(5);
scheduler.AdvanceBy(5);
C
```

Start

The *TestScheduler*'s *Start()* method is an effective way to execute everything that has been scheduled. We take the same example again and swap out the *AdvanceBy(long)* calls for a single *Start()* call.

```
var scheduler = new TestScheduler();
scheduler.Schedule(() => Console.WriteLine("A")); //Schedule immediately
scheduler.Schedule(TimeSpan.FromTicks(10), () => Console.WriteLine("B"));
scheduler.Schedule(TimeSpan.FromTicks(20), () => Console.WriteLine("C"));
Console.WriteLine("scheduler.Start();");
scheduler.Start();
Console.WriteLine("scheduler.Clock:{0}", scheduler.Clock);
```

Output:

```
scheduler.Start();
A
B
C
scheduler.Clock:20
```

Note that once all of the scheduled actions have been executed, the virtual clock matches our last scheduled item (20 ticks).

We further extend our example by scheduling a new action to happen after *Start()* has already been called.

```
var scheduler = new TestScheduler();
scheduler.Schedule(() => Console.WriteLine("A"));
scheduler.Schedule(TimeSpan.FromTicks(10), () => Console.WriteLine("B"));
scheduler.Schedule(TimeSpan.FromTicks(20), () => Console.WriteLine("C"));
Console.WriteLine("scheduler.Start();");
scheduler.Start();
Console.WriteLine("scheduler.Clock:{0}", scheduler.Clock);
scheduler.Schedule(() => Console.WriteLine("D"));
```

Output:

```
scheduler.Start();
A
B
C
scheduler.Clock:20
```

Note that the output is exactly the same; If we want our fourth action to be executed, we will have to call *Start()* again.

Stop

In previous releases of Rx, the *Start()* method was called *Run()*. Now there is a *Stop()* method whose name seems to imply some symmetry with *Start()*. All it does however, is set the *IsEnabled* property to false. This property is used as an internal flag to check whether the internal queue of actions should continue being executed. The processing of the queue may indeed be instigated by *Start()*, however *AdvanceTo* or *AdvanceBy* can be used too.

In this example, we show how you could use *Stop()* to pause processing of scheduled actions.

```
var scheduler = new TestScheduler();
scheduler.Schedule(() => Console.WriteLine("A"));
scheduler.Schedule(TimeSpan.FromTicks(10), () => Console.WriteLine("B"));
scheduler.Schedule(TimeSpan.FromTicks(15), scheduler.Stop);
scheduler.Schedule(TimeSpan.FromTicks(20), () => Console.WriteLine("C"));
Console.WriteLine("scheduler.Start();");
scheduler.Start();
Console.WriteLine("scheduler.Clock:{0}", scheduler.Clock);
```

Output:

```
scheduler.Start();
A
B
scheduler.Clock:15
```

Note that "C" never gets printed as we stop the clock at 15 ticks. I have been testing Rx successfully for nearly two years now, yet I have not found the need to use the *Stop()* method. I imagine that there are cases that warrant its use; however I just wanted to make the point that you do not have to be concerned about the lack of use of it in your tests.

Schedule collisions

When scheduling actions, it is possible and even likely that many actions will be scheduled for the same point in time. This most commonly would occur when scheduling multiple actions for *now*. It could also happen that there are multiple actions scheduled for the same point in the future. The *TestScheduler* has a simple way to deal with this. When actions are scheduled, they are marked with the clock time they are scheduled for. If multiple items are scheduled for the same point in time, they are queued in order that they were scheduled; when the clock advances, all items for that point in time are executed in the order that they were scheduled.

```
var scheduler = new TestScheduler();
scheduler.Schedule(TimeSpan.FromTicks(10), () => Console.WriteLine("A"));
scheduler.Schedule(TimeSpan.FromTicks(10), () => Console.WriteLine("B"));
scheduler.Schedule(TimeSpan.FromTicks(10), () => Console.WriteLine("C"));
Console.WriteLine("scheduler.Start();");
scheduler.Start();
Console.WriteLine("scheduler.Clock:{0}", scheduler.Clock);
```

Output:

```
scheduler.AdvanceTo(10);
```

```
A
B
C
scheduler.Clock:10
```

Note that the virtual clock is at 10 ticks, the time we advanced to.

Testing Rx code

Now that we have learnt a little bit about the *TestScheduler*, let's look at how we could use it to test our two initial code snippets that use *Interval* and *Timeout*. We want to execute tests as fast as possible but still maintain the semantics of time. In this example we generate our five values one second apart but pass in our *TestScheduler* to the *Interval* method to use instead of the default scheduler.

```
[TestMethod]
public void Testing_with_test_scheduler()
{
        var expectedValues = new long[] {0, 1, 2, 3, 4};
        var actualValues = new List<long>();
        var scheduler = new TestScheduler();
        var interval = Observable
        .Interval(TimeSpan.FromSeconds(1), scheduler)
        .Take(5);
        interval.Subscribe(actualValues.Add);
        scheduler.Start();
        CollectionAssert.AreEqual(expectedValues, actualValues);
        //Executes in less than 0.01s "on my machine"
}
```

While this is mildly interesting, what I think is more important is how we would test a real piece of code. Imagine, if you will, a ViewModel that subscribes to a stream of prices. As prices are published, it adds them to a collection. Assuming this is a WPF or Silverlight implementation, we take the liberty of enforcing that the subscription be done on the *ThreadPool* and the observing is executed on the *Dispatcher*.

```
public class MyViewModel : IMyViewModel
{
        private readonly IMyModel _myModel;
        private readonly ObservableCollection<decimal> _prices;
        public MyViewModel(IMyModel myModel)
        {
        _myModel = myModel;
        _prices = new ObservableCollection<decimal>();
        }
        public void Show(string symbol)
        {
        //TODO: resource mgt, exception handling etc...
        _myModel.PriceStream(symbol)
        .SubscribeOn(Scheduler.ThreadPool)
        .ObserveOn(Scheduler.Dispatcher)
        .Timeout(TimeSpan.FromSeconds(10), Scheduler.ThreadPool)
        .Subscribe(
        Prices.Add,
        ex=>
                {
                if(ex is TimeoutException)
                IsConnected = false;
                });
        IsConnected = true;
        }
        public ObservableCollection<decimal> Prices
        {
        get { return _prices; }
        }
        public bool IsConnected { get; private set; }
}
```

Injecting scheduler dependencies

While the snippet of code above may do what we want it to, it will be hard to test as it is accessing the schedulers via static properties. To help my testing, I have created my own interface that exposes the same *IScheduler* implementations that the *Scheduler* type does, i suggest you adopt this interface too.

```
public interface ISchedulerProvider
{
        IScheduler CurrentThread { get; }
        IScheduler Dispatcher { get; }
        IScheduler Immediate { get; }
        IScheduler NewThread { get; }
        IScheduler ThreadPool { get; }
        //IScheduler TaskPool { get; }
}
```

Whether the `TaskPool` property should be included or not depends on your target platform. If you adopt this concept, feel free to name this type in accordance with your naming conventions e.g. `SchedulerService`,

Schedulers. The default implementation that we would run in production is implemented as follows:

```
public sealed class SchedulerProvider : ISchedulerProvider
{
        public IScheduler CurrentThread
        {
        get { return Scheduler.CurrentThread; }
        }
        public IScheduler Dispatcher
        {
        get { return DispatcherScheduler.Instance; }
        }
        public IScheduler Immediate
        {
        get { return Scheduler.Immediate; }
        }
        public IScheduler NewThread
        {
        get { return Scheduler.NewThread; }
        }
        public IScheduler ThreadPool
        {
        get { return Scheduler.ThreadPool; }
        }
        //public IScheduler TaskPool { get { return Scheduler.TaskPool; } }
}
```

This now allows me to substitute implementations of *ISchedulerProvider* to help with testing. I could mock the *ISchedulerProvider*, but I find it easier to provide a test implementation. My implementation for testing is as follows.

```
public sealed class TestSchedulers : ISchedulerProvider
{
        private readonly TestScheduler _currentThread = new TestScheduler();
        private readonly TestScheduler _dispatcher = new TestScheduler();
        private readonly TestScheduler _immediate = new TestScheduler();
        private readonly TestScheduler _newThread = new TestScheduler();
        private readonly TestScheduler _threadPool = new TestScheduler();
        #region Explicit implementation of ISchedulerService
        IScheduler ISchedulerProvider.CurrentThread { get { return _currentThread; } }
        IScheduler ISchedulerProvider.Dispatcher { get { return _dispatcher; } }
        IScheduler ISchedulerProvider.Immediate { get { return _immediate; } }
        IScheduler ISchedulerProvider.NewThread { get { return _newThread; } }
        IScheduler ISchedulerProvider.ThreadPool { get { return _threadPool; } }
        #endregion
        public TestScheduler CurrentThread { get { return _currentThread; } }
        public TestScheduler Dispatcher { get { return _dispatcher; } }
        public TestScheduler Immediate { get { return _immediate; } }
        public TestScheduler NewThread { get { return _newThread; } }
        public TestScheduler ThreadPool { get { return _threadPool; } }
}
```

Note that *ISchedulerProvider* is implemented explicitly. This means that, in our tests, we can access the *TestScheduler* instances directly, but our system under test (SUT) still just sees the interface implementation. I can now write some tests for my ViewModel. Below, we test a modified version of the *MyViewModel* class that takes an *ISchedulerProvider* and uses that instead of the static schedulers from the *Scheduler* class. We also use the popular Moq framework in order to mock out our model.

```
[TestInitialize]
public void SetUp()
{
        _myModelMock = new Mock<IMyModel>();
        _schedulerProvider = new TestSchedulers();
        _viewModel = new MyViewModel(_myModelMock.Object, _schedulerProvider);
}
[TestMethod]
public void Should_add_to_Prices_when_Model_publishes_price()
{
        decimal expected = 1.23m;
        var priceStream = new Subject<decimal>();
        _myModelMock.Setup(svc => svc.PriceStream(It.IsAny<string>())).Returns(priceStream);
        _viewModel.Show("SomeSymbol");
        //Schedule the OnNext
        _schedulerProvider.ThreadPool.Schedule(() => priceStream.OnNext(expected));
        Assert.AreEqual(0, _viewModel.Prices.Count);
        //Execute the OnNext action
        _schedulerProvider.ThreadPool.AdvanceBy(1);
        Assert.AreEqual(0, _viewModel.Prices.Count);
        //Execute the OnNext handler
        _schedulerProvider.Dispatcher.AdvanceBy(1);
        Assert.AreEqual(1, _viewModel.Prices.Count);
        Assert.AreEqual(expected, _viewModel.Prices.First());
}
[TestMethod]
public void Should_disconnect_if_no_prices_for_10_seconds()
{
        var timeoutPeriod = TimeSpan.FromSeconds(10);
        var priceStream = Observable.Never<decimal>();
        _myModelMock.Setup(svc => svc.PriceStream(It.IsAny<string>())).Returns(priceStream);
        _viewModel.Show("SomeSymbol");
        _schedulerProvider.ThreadPool.AdvanceBy(timeoutPeriod.Ticks - 1);
        Assert.IsTrue(_viewModel.IsConnected);
        _schedulerProvider.ThreadPool.AdvanceBy(timeoutPeriod.Ticks);
        Assert.IsFalse(_viewModel.IsConnected);
}
```

Output:

```
2 passed, 0 failed, 0 skipped, took 0.41 seconds (MSTest 10.0).
```

These two tests ensure five things:

1. That the *Price* property has prices added to it as the model produces them
2. That the sequence is subscribed to on the ThreadPool
3. That the *Price* property is updated on the Dispatcher i.e. the sequence is observed on the Dispatcher
4. That a timeout of 10 seconds between prices will set the ViewModel to disconnected.
5. The tests run fast. While the time to run the tests is not that impressive, most of that time seems to be spent warming up my test harness. Moreover, increasing the test count to 10 only adds 0.03seconds. In general, on a modern CPU, I expect to see unit tests run at a rate of +1000 tests per second

Usually, I would not have more than one assert/verify per test, but here it does help illustrate a point. In the first test, we can see that only once both the `ThreadPool` and the `Dispatcher` schedulers have been run will we get a result. In the second test, it helps to verify that the timeout is not less than 10 seconds.

In some scenarios, you are not interested in the scheduler and you want to be focusing your tests on other functionality. If this is the case, then you may want to create another test implementation of the *ISchedulerProvider* that returns the *ImmediateScheduler* for all of its members. That can help reduce the noise in your tests.

```
public sealed class ImmediateSchedulers : ISchedulerService
{
        public IScheduler CurrentThread { get { return Scheduler.Immediate; } }
        public IScheduler Dispatcher { get { return Scheduler.Immediate; } }
        public IScheduler Immediate { get { return Scheduler.Immediate; } }
        public IScheduler NewThread { get { return Scheduler.Immediate; } }
        public IScheduler ThreadPool { get { return Scheduler.Immediate; } }
}
```

Advanced features - ITestableObserver

The *TestScheduler* provides further advanced features. I find that I am able to get by quite well without these methods, but others may find them useful. Perhaps this is because I have found myself accustomed to testing without them from using earlier versions of Rx.

Start(Func<IObservable<T>>)

There are three overloads to *Start*, which are used to start an observable sequence at a given time, record the notifications it makes and dispose of the subscription at a given time. This can be confusing at first, as the parameterless overload of *Start* is quite unrelated. These three overloads return an *ITestableObserver<T>* which allows you to record the notifications from an observable sequence, much like the *Materialize* method we saw in the Transformation chapter.

```
public interface ITestableObserver<T> : IObserver<T>
{
        // Gets recorded notifications received by the observer.
        IList<Recorded<Notification<T>>> Messages { get; }
}
```

While there are three overloads, we will look at the most specific one first. This overload takes four parameters:

1. an observable sequence factory delegate
2. the point in time to invoke the factory
3. the point in time to subscribe to the observable sequence returned from the factory
4. the point in time to dispose of the subscription

The *time* for the last three parameters is measured in ticks, as per the rest of the *TestScheduler* members.

```
public ITestableObserver<T> Start<T>(
        Func<IObservable<T>> create,
        long created,
        long subscribed,
        long disposed)
{...}
```

We could use this method to test the *Observable.Interval* factory method. Here, we create an observable sequence that spawns a value every second for 4 seconds. We use the *TestScheduler.Start* method to create and subscribe to it immediately (by passing 0 for the second and third parameters). We dispose of our subscription after 5 seconds. Once the *Start* method has run, we output what we have recorded.

```
var scheduler = new TestScheduler();
var source = Observable.Interval(TimeSpan.FromSeconds(1), scheduler)
        .Take(4);
var testObserver = scheduler.Start(
        () => source,
        0,
        0,
        TimeSpan.FromSeconds(5).Ticks);
Console.WriteLine("Time is {0} ticks", scheduler.Clock);
Console.WriteLine("Received {0} notifications", testObserver.Messages.Count);
foreach (Recorded<Notification<long>> message in testObserver.Messages)
{
        Console.WriteLine("{0} @ {1}", message.Value, message.Time);
}
```

Output:

```
Time is 50000000 ticks
```

```
Received 5 notifications
OnNext(0) @ 10000001
OnNext(1) @ 20000001
OnNext(2) @ 30000001
OnNext(3) @ 40000001
OnCompleted() @ 40000001
```

Note that the *ITestObserver<T>* records `OnNext` and `OnCompleted` notifications. If the sequence was to terminate in error, the *ITestObserver<T>* would record the `OnError` notification instead.

We can play with the input variables to see the impact it makes. We know that the *Observable.Interval* method is a Cold Observable, so the virtual time of the creation is not relevant. Changing the virtual time of the subscription can change our results. If we change it to 2 seconds, we will notice that if we leave the disposal time at 5 seconds, we will miss some messages.

```
var testObserver = scheduler.Start(
        () => Observable.Interval(TimeSpan.FromSeconds(1), scheduler).Take(4),
        0,
        TimeSpan.FromSeconds(2).Ticks,
        TimeSpan.FromSeconds(5).Ticks);
```

Output:

```
Time is 50000000 ticks
Received 2 notifications
OnNext(0) @ 30000000
OnNext(1) @ 40000000
```

We start the subscription at 2 seconds; the *Interval* produces values after each second (i.e. second 3 and 4), and we dispose on second 5. So we miss the other two `OnNext` messages as well as the `OnCompleted` message.

There are two other overloads to this *TestScheduler.Start* method.

```
public ITestableObserver<T> Start<T>(Func<IObservable<T>> create, long disposed)
{
        if (create == null)
        throw new ArgumentNullException("create");
        else
        return this.Start<T>(create, 100L, 200L, disposed);
}
public ITestableObserver<T> Start<T>(Func<IObservable<T>> create)
{
        if (create == null)
        throw new ArgumentNullException("create");
        else
        return this.Start<T>(create, 100L, 200L, 1000L);
}
```

As you can see, these overloads just call through to the variant we have been looking at, but passing some default values. I am not sure why these default values are special; I can not imagine why you would want to use these two methods, unless your specific use case matched that specific configuration exactly.

CreateColdObservable

Just as we can record an observable sequence, we can also use *CreateColdObservable* to playback a set of *Recorded<Notification<int>>*. The signature for *CreateColdObservable* simply takes a `params` array of recorded notifications.

```
// Creates a cold observable from an array of notifications.
// Returns a cold observable exhibiting the specified message behavior.
public ITestableObservable<T> CreateColdObservable<T>(
        params Recorded<Notification<T>>[] messages)
{...}
```

T h e *CreateColdObservable* returns an *ITestableObservable<T>*. This interface extends *IObservable<T>* by exposing the list of "subscriptions" and the list of messages it will produce.

```
public interface ITestableObservable<T> : IObservable<T>
{
        // Gets the subscriptions to the observable.
        IList<Subscription> Subscriptions { get; }
        // Gets the recorded notifications sent by the observable.
        IList<Recorded<Notification<T>>> Messages { get; }
}
```

Using *CreateColdObservable*, we can emulate the *Observable.Interval* test we had earlier.

```
var scheduler = new TestScheduler();
var source = scheduler.CreateColdObservable(
        new Recorded<Notification<long>>(10000000, Notification.CreateOnNext(0L)),
        new Recorded<Notification<long>>(20000000, Notification.CreateOnNext(1L)),
        new Recorded<Notification<long>>(30000000, Notification.CreateOnNext(2L)),
        new Recorded<Notification<long>>(40000000, Notification.CreateOnNext(3L)),
        new Recorded<Notification<long>>(40000000, Notification.CreateOnCompleted<long>())
        );
var testObserver = scheduler.Start(
        () => source,
        0,
        0,
        TimeSpan.FromSeconds(5).Ticks);
Console.WriteLine("Time is {0} ticks", scheduler.Clock);
Console.WriteLine("Received {0} notifications", testObserver.Messages.Count);
foreach (Recorded<Notification<long>> message in testObserver.Messages)
{
        Console.WriteLine("  {0} @ {1}", message.Value, message.Time);
}
```

Output:

```
Time is 50000000 ticks
Received 5 notifications
OnNext(0) @ 10000001
OnNext(1) @ 20000001
OnNext(2) @ 30000001
OnNext(3) @ 40000001
OnCompleted() @ 40000001
```

Note that our output is exactly the same as the previous example with *Observable.Interval*.

CreateHotObservable

We can also create hot test observable sequences using the *CreateHotObservable* method. It has the same parameters and return value as *CreateColdObservable*; the difference is that the virtual time specified for each message is now relative to when the observable was created, not when it is subscribed to as per the *CreateColdObservable* method.

This example is just that last "cold" sample, but creating a Hot observable instead.

```
var scheduler = new TestScheduler();
var source = scheduler.CreateHotObservable(
        new Recorded<Notification<long>>(10000000, Notification.CreateOnNext(0L)),
...
```

Output:

```
Time is 50000000 ticks
Received 5 notifications
OnNext(0) @ 10000000
OnNext(1) @ 20000000
OnNext(2) @ 30000000
OnNext(3) @ 40000000
OnCompleted() @ 40000000
```

Note that the output is almost the same. Scheduling of the creation and subscription do not affect the Hot Observable, therefore the notifications happen 1 tick earlier than their Cold counterparts.

We can see the major difference a Hot Observable bears by changing the virtual create time and virtual subscribe time to be different values. With a Cold Observable, the virtual create time has no real impact, as subscription is what initiates any action. This means we can not miss any early message on a Cold Observable. For Hot Observables, we can miss messages if we subscribe too late. Here, we create the Hot Observable immediately, but only subscribe to it after 1 second (thus missing the first message).

```
var scheduler = new TestScheduler();
var source = scheduler.CreateHotObservable(
        new Recorded>Notification>long<<(10000000, Notification.CreateOnNext(0L)),
        new Recorded>Notification>long<<(20000000, Notification.CreateOnNext(1L)),
        new Recorded>Notification>long<<(30000000, Notification.CreateOnNext(2L)),
        new Recorded>Notification>long<<(40000000, Notification.CreateOnNext(3L)),
        new Recorded>Notification>long<<(40000000, Notification.CreateOnCompleted>long<())
        );
var testObserver = scheduler.Start(
        () =< source,
        0,
        TimeSpan.FromSeconds(1).Ticks,
        TimeSpan.FromSeconds(5).Ticks);
Console.WriteLine("Time is {0} ticks", scheduler.Clock);
Console.WriteLine("Received {0} notifications", testObserver.Messages.Count);
foreach (Recorded>Notification>long<< message in testObserver.Messages)
{
        Console.WriteLine("  {0} @ {1}", message.Value, message.Time);
}
```

Output:

```
Time is 50000000 ticks
Received 4 notifications
OnNext(1) @ 20000000
OnNext(2) @ 30000000
OnNext(3) @ 40000000
OnCompleted() @ 40000000
```

CreateObserver

Finally, if you do not want to use the *TestScheduler.Start* methods, and you need more fine-grained control over your observer, you can use *TestScheduler.CreateObserver()*. This will return an *ITestObserver* that you can use to manage the subscriptions to your observable sequences with. Furthermore, you will still be exposed to the recorded messages and any subscribers.

Current industry standards demand broad coverage of automated unit tests to meet quality assurance standards. Concurrent programming, however, is often a difficult area to test well. Rx delivers a well-designed implementation of testing features, allowing deterministic and high-throughput testing. The *TestScheduler* provides methods to control virtual time and produce observable sequences for testing. This ability to easily and reliably test concurrent systems sets Rx apart from many other libraries.

Sequences of coincidence

We can conceptualize events that have *duration* as *windows*. For example;

- a server is up
- a person is in a room
- a button is pressed (and not yet released).

The first example could be re-worded as "for this window of time, the server was up". An event from one source may have a greater value if it coincides with an event from another source. For example, while at a music festival, you may only be interested in tweets (event) about an artist while they are playing (window). In finance, you may only be interested in trades (event) for a certain instrument while the New York market is open (window). In operations, you may be interested in the user sessions (window) that remained active during an upgrade of a system (window). In that example, we would be querying for coinciding windows.

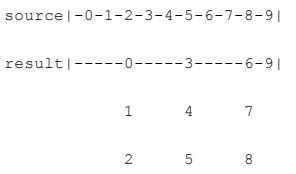
Rx provides the power to query sequences of coincidence, sometimes called 'sliding windows'. We already recognize the benefit that Rx delivers when querying data in motion. By additionally providing the power to query sequences of coincidence, Rx exposes yet another dimension of possibilities.

Buffer revisited

Buffer is not a new operator to us; however, it can now be conceptually grouped with the window operators. Each of these windowing operators act on a sequence and a window of time. Each operator will open a window when the source sequence produces a value. The way the window is closed, and which values are exposed, are the main differences between each of the operators. Let us just quickly recap the internal working of the *Buffer* operator and see how this maps to the concept of "windows of time".

Buffer will create a window when the first value is produced. It will then put that value into an internal cache. The window will stay open until the count of values has been reached; each of these values will have been cached. When the count has been reached, the window will close and the cache will be published to the result sequence as an *IList<T>*. When the next value is produced from the source, the cache is cleared and we start again. This means that *Buffer* will take an *IObservable<T>* and return an *IObservable<IList<T>>*.

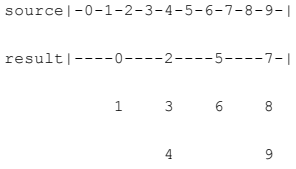
Example Buffer with count of 3



In this marble diagram, I have represented the list of values being returned at a point in time as a column of data, i.e. the values 0, 1 & 2 are all returned in the first buffer.

Understanding buffer with time is only a small step away from understanding buffer with count; instead of passing a count, we pass a *TimeSpan*. The closing of the window (and therefore the buffer's cache) is now dictated by time instead of the number of values. This is now more complicated as we have introduced some sort of scheduling. To produce the *IList<T>* at the correct point in time, we need a scheduler assigned to perform the timing. Incidentally, this makes testing a lot easier.

Example Buffer with time of 5 units



Window

The *Window* operators are very similar to the *Buffer* operators; they only really differ by their return type. Where *Buffer* would take an *IObservable<T>* and return an *IObservable<IList<T>>*, the Window operators return an *IObservable<IObservable<T>>*. It is also worth noting that the *Buffer* operators will not yield their buffers until the window closes.

Here we can see the simple overloads to *Window*. There is a surprising symmetry with the *Window* and *Buffer* overloads.

```
public static IObservable<IObservable<TSource>> Window<TSource>(
        this IObservable<TSource> source,
        int count)
{...}
public static IObservable<IObservable<TSource>> Window<TSource>(
        this IObservable<TSource> source,
        int count,
        int skip)
{...}
public static IObservable<IObservable<TSource>> Window<TSource>(
        this IObservable<TSource> source,
        TimeSpan timeSpan)
{...}
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,
        TimeSpan timeShift)
{...}
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,
        IScheduler scheduler)
{...}
public static IObservable<IObservable<TSource>> Window<TSource>(
        this IObservable<TSource> source,
        TimeSpan timeSpan,
        int count,
        IScheduler scheduler)
{...}
```

This is an example of *Window* with a count of 3 as a marble diagram:

```
source |-0-1-2-3-4-5-6-7-8-9|

window0|-0-1-2|

window1        3-4-5|

window2              6-7-8|

window3                    9|
```

For demonstration purposes, we could reconstruct that with this code.

```
var windowIdx = 0;
var source = Observable.Interval(TimeSpan.FromSeconds(1)).Take(10);
source.Window(3)
        .Subscribe(window =>
        {
        var id = windowIdx++;
        Console.WriteLine("--Starting new window");
        var windowName = "Window" + thisWindowIdx;
        window.Subscribe(
        value => Console.WriteLine("{0} : {1}", windowName, value),
        ex => Console.WriteLine("{0} : {1}", windowName, ex),
        () => Console.WriteLine("{0} Completed", windowName));
        },
        () => Console.WriteLine("Completed"));
```

Output:

```
--Starting new window
window0 : 0
window0 : 1
window0 : 2
window0 Completed
--Starting new window
window1 : 3
window1 : 4
window1 : 5
window1 Completed
```

```
--Starting new window
window2 : 6
window2 : 7
window2 : 8
window2 Completed
--Starting new window
window3 : 9
window3 Completed
Completed
```

Example of Window with time of 5 units

```
source |-0-1-2-3-4-5-6-7-8-9|

window0|-0-1-|

window1      2-3-4|

window2           -5-6-|

window3                7-8-9|
```

A major difference we see here is that the *Window* operators can notify you of values from the source as soon as they are produced. The *Buffer* operators, on the other hand, must wait until the window closes before the values can be notified as an entire list.

Flattening a Window operation

I think it is worth noting, at least from an academic standpoint, that the *Window* operators produce *IObservable<IObservable<T>>*. We have explored the concept of nested observables in the earlier chapter on Aggregation. *Concat*, *Merge* and *Switch* each have an overload that takes an *IObservable<IObservable<T>>* and returns an *IObservable<T>*. As the *Window* operators ensure that the windows (child sequences) do not overlap, we can use either of the *Concat*, *Switch* or *Merge* operators to turn a windowed sequence back into its original form.

```
//is the same as Observable.Interval(TimeSpan.FromMilliseconds(200)).Take(10)
        var switchedWindow = Observable.Interval(TimeSpan.FromMilliseconds(200)).Take(10)
        .Window(TimeSpan.FromMilliseconds(500))
        .Switch();
```

Customizing windows

The overloads above provide simple ways to break a sequence into smaller nested windows using a count and/or a time span. Now we will look at the other overloads, that provide more flexibility over how windows are managed.

```
//Projects each element of an observable sequence into consecutive non-overlapping windows.
//windowClosingSelector : A function invoked to define the boundaries of the produced
//  windows. A new window is started when the previous one is closed.
public static IObservable<IObservable<TSource>> Window<TSource, TWindowClosing>
(
        this IObservable<TSource> source,
        Func<IObservable<TWindowClosing>> windowClosingSelector
)
{...}
```

The first of these complex overloads allows us to control when windows should close. The *windowClosingSelector* function is called each time a window is created. Windows are created on subscription and immediately after a window closes; windows close when the sequence from the *windowClosingSelector* produces a value. The value is disregarded so it doesn't matter what type the sequence values are; in fact you can just complete the sequence from *windowClosingSelector* to close the window instead.

In this example, we create a window with a closing selector. We return the same subject from that selector every time, then notify from the subject whenever a user presses enter from the console.

```
var windowIdx = 0;
var source = Observable.Interval(TimeSpan.FromSeconds(1)).Take(10);
var closer = new Subject<Unit>();
source.Window(() => closer)
        .Subscribe(window =>
        {
        var thisWindowIdx = windowIdx++;
        Console.WriteLine("--Starting new window");
        var windowName = "Window" + thisWindowIdx;
        window.Subscribe(
        value => Console.WriteLine("{0} : {1}", windowName, value),
        ex => Console.WriteLine("{0} : {1}", windowName, ex),
        () => Console.WriteLine("{0} Completed", windowName));
        },
        () => Console.WriteLine("Completed"));
var input = "";
while (input!="exit")
{
        input = Console.ReadLine();
        closer.OnNext(Unit.Default);
}
```

Output (when I hit enter after '1' and '5' are displayed):

```
--Starting new window
window0 : 0
window0 : 1
window0 Completed
--Starting new window
window1 : 2
window1 : 3
window1 : 4
window1 : 5
window1 Completed
--Starting new window
window2 : 6
window2 : 7
window2 : 8
window2 : 9
window2 Completed
Completed
```

The most complex overload of *Window* allows us to create potentially overlapping windows.

```
//Projects each element of an observable sequence into zero or more windows.
// windowOpenings : Observable sequence whose elements denote the creation of new windows.
// windowClosingSelector : A function invoked to define the closing of each produced window.
public static IObservable<IObservable<TSource>> Window
        <TSource, TWindowOpening, TWindowClosing>
(
        this IObservable<TSource> source,
        IObservable<TWindowOpening> windowOpenings,
        Func<TWindowOpening, IObservable<TWindowClosing>> windowClosingSelector
)
{...}
```

This overload takes three arguments

1. The source sequence
2. A sequence that indicates when a new window should be opened
3. A function that takes a window opening value, and returns a window closing sequence

This overload offers great flexibility in the way windows are opened and closed. Windows can be largely independent from each other; they can overlap, vary in size and even skip values from the source.

To ease our way into this more complex overload, let's first try to use it to recreate a simpler version of *Window* (the overload that takes a count). To do so, we need to open a window once on the initial subscription, and once each time the source has produced then specified count. The window needs to close each time that count is reached. To achieve this we only need the source sequence. We will share it by using the *Publish* method, then supply 'views' of the source as each of the arguments.

```
public static IObservable<IObservable<T>> MyWindow<T>(
        this IObservable<T> source,
        int count)
{
        var shared = source.Publish().RefCount();
        var windowEdge = shared
        .Select((i, idx) => idx % count)
        .Where(mod => mod == 0)
        .Publish()
        .RefCount();
        return shared.Window(windowEdge, _ => windowEdge);
}
```

If we now want to extend this method to offer skip functionality, we need to have two different sequences: one for opening and one for closing. We open a window on subscription and again after the *skip* items have passed. We close those windows after '`count`' items have passed since the window opened.

```
public static IObservable<IObservable<T>> MyWindow<T>(
        this IObservable<T> source,
        int count,
        int skip)
{
        if (count <= 0) throw new ArgumentOutOfRangeException();
        if (skip <= 0) throw new ArgumentOutOfRangeException();
        var shared = source.Publish().RefCount();
        var index = shared
        .Select((i, idx) => idx)
        .Publish()
        .RefCount();
        var windowOpen = index.Where(idx => idx % skip == 0);
        var windowClose = index.Skip(count-1);
        return shared.Window(windowOpen, _ => windowClose);
}
```

We can see here that the `windowClose` sequence is re-subscribed to each time a window is opened, due to it being returned from a function. This allows us to reapply the skip (`Skip(count-1)`) for each window. Currently, we ignore the value that the `windowOpen` pushes to the `windowClose` selector, but if you require it for some logic, it is available to you.

As you can see, the *Window* operator can be quite powerful. We can even use *Window* to replicate other operators; for instance we can create our own implementation of *Buffer* that way. We can have the *SelectMany* operator take a single value (the window) to produce zero or more values of another type (in our case, a single *IList<T>*). To create the *IList<T>* without blocking, we can apply the *Aggregate* method and use a new *List<T>* as the seed.

```
public static IObservable<IList<T>> MyBuffer<T>(
        this IObservable<T> source,
        int count)
{
        return source.Window(count)
        .SelectMany(window =>
        window.Aggregate(
        new List<T>(),
        (list, item) =>
        {
                list.Add(item);
                return list;
        }));
}
```

It may be an interesting exercise to try implementing other time shifting methods, like *Sample* or *Throttle*, with *Window*.

Join

The *Join* operator allows you to logically join two sequences. Whereas the *Zip* operator would pair values from the two sequences together by index, the *Join* operator allows you join sequences by intersecting windows. Like the *Window* overload we just looked at, you can specify when a window should close via an observable sequence; this sequence is returned from a function that takes an opening value. The *Join* operator has two such functions, one for the first source sequence and one for the second source sequence. Like the *Zip* operator, we also need to provide a selector function to produce the result item from the pair of values.

```
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
)
```

This is a complex signature to try and understand in one go, so let's take it one parameter at a time.

IObservable<TLeft> left is the source sequence that defines when a window starts. This is just like the *Buffer* and *Window* operators, except that every value published from this source opens a new window. In *Buffer* and *Window*, by contrast, some values just fell into an existing window.

I like to think of *IObservable<TRight> right* as the window value sequence. While the left sequence controls opening the windows, the right sequence will try to pair up with a value from the left sequence.

Let us imagine that our left sequence produces a value, which creates a new window. If the right sequence produces a value while the window is open, then the `resultSelector` function is called with the two values. This is the crux of join, pairing two values from a sequence that occur within the same window. This then leads us to our next question; when does the window close? The answer illustrates both the power and the complexity of the *Join* operator.

When *left* produces a value, a window is opened. That value is also passed, at that time, to the *leftDurationSelector* function, which returns an *IObservable<TLeftDuration>*. When that sequence produces a value or completes, the window for that value is closed. Note that it is irrelevant what the type of *TLeftDuration* is. This initially left me with the feeling that *IObservable<TLeftDuration>* was a bit excessive as you effectively just need some sort of event to say 'Closed'. However, by being allowed to use *IObservable<T>*, you can do some clever manipulation as we will see later.

Let us now imagine a scenario where the left sequence produces values twice as fast as the right sequence. Imagine that in addition we never close the windows; we could do this by always returning *Observable.Never<Unit>()* from the *leftDurationSelector* function. This would result in the following pairs being produced.

Left Sequence

```
L 0-1-2-3-4-5-
```

Right Sequence

```
R --A---B---C-


0, A
1, A
0, B
1, B
2, B
3, B
0, C
1, C
2, C
3, C
4, C
5, C
```

As you can see, the left values are cached and replayed each time the right produces a value.

Now it seems fairly obvious that, if I immediately closed the window by returning *Observable.Empty<Unit>*, or perhaps *Observable.Return(0)*, windows would never be opened thus no pairs would ever get produced. However, what could I do to make sure that these windows did not overlap- so that, once a second value was produced I would no longer see the first value? Well, if we returned the `left` sequence from the `leftDurationSelector`, that could do the trick. But wait, when we return the sequence `left` from the `leftDurationSelector`, it would try to create another subscription and that may introduce side effects. The quick answer to that is to *Publish* and *RefCount* the `left` sequence. If we do that, the results look more like this.

```
left   |-0-1-2-3-4-5|

right |---A---B---C|

result|---1---3---5

          A   B   C
```

The last example is very similar to *CombineLatest*, except that it is only producing a pair when the right sequence changes. We could use *Join* to produce our own version of *CombineLatest*. If the values from the left sequence expire when the next value from left was notified, then I would be well on my way to implementing my version of *CombineLatest*. However I need the same thing to happen for the right. Luckily the *Join* operator provides a `rightDurationSelector` that works just like the `leftDurationSelector`. This is simple to implement; all I need to do is return a reference to the same left sequence when a left value is produced, and do the same for the right. The code looks like this.

```
public static IObservable<TResult> MyCombineLatest<TLeft, TRight, TResult>
(
        IObservable<TLeft> left,
        IObservable<TRight> right,
        Func<TLeft, TRight, TResult> resultSelector
)
{
        var refcountedLeft = left.Publish().RefCount();
        var refcountedRight = right.Publish().RefCount();
        return Observable.Join(
        refcountedLeft,
        refcountedRight,
        value => refcountedLeft,
        value => refcountedRight,
        resultSelector);
}
```

While the code above is not production quality (it would need to have some gates in place to mitigate race conditions), it shows how powerful *Join* is; we can actually use it to create other operators!

GroupJoin

When the *Join* operator pairs up values that coincide within a window, it will pass the scalar values left and right to the *resultSelector*. The *GroupJoin* operator takes this one step further by passing the left (scalar) value immediately to the `resultSelector` with the right (sequence) value. The right parameter represents all of the values from the right sequences that occur within the window. Its signature is very similar to *Join*, but note the difference in the `resultSelector` parameter.

```
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
)
```

If we went back to our first *Join* example where we had

- the `left` producing values twice as fast as the right,
- the left never expiring
- the right immediately expiring

this is what the result may look like

```
left              |-0-1-2-3-4-5|

right             |---A---B---C|

0th window values   --A---B---C|

1st window values     A---B---C|

2nd window values       --B---C|

3rd window values         B---C|

4th window values           --C|

5th window values             C|
```

We could switch it around and have the left expired immediately and the right never expire. The result would then look like this:

```
left              |-0-1-2-3-4-5|

right             |---A---B---C|

0th window values   |

1st window values     A|

2nd window values       A|

3rd window values         AB|

4th window values           AB|

5th window values             ABC|
```

This starts to make things interesting. Perceptive readers may have noticed that with *GroupJoin* you could effectively re-create your own *Join* method by doing something like this:

```
public IObservable<TResult> MyJoin<TLeft, TRight, TLeftDuration, TRightDuration, TResult>(
        IObservable<TLeft> left,
        IObservable<TRight> right,
        Func<TLeft, IObservable<TLeftDuration>> leftDurationSelector,
```

```
        Func<TRight, IObservable<TRightDuration>> rightDurationSelector,
        Func<TLeft, TRight, TResult> resultSelector)
{
        return Observable.GroupJoin
        (
        left,
        right,
        leftDurationSelector,
        rightDurationSelector,
        (leftValue, rightValues)=>
        rightValues.Select(rightValue=>resultSelector(leftValue, rightValue))
        )
        .Merge();
}
```

You could even create a crude version of *Window* with this code:

```
public IObservable<IObservable<T>> MyWindow<T>(
        IObservable<T> source,
        TimeSpan windowPeriod)
{
        return Observable.Create<IObservable<T>>(o
        =>
        {
        var sharedSource = source
        .Publish()
        .RefCount();
        var intervals = Observable.Return(0L)
        .Concat(Observable.Interval(windowPeriod))
        .TakeUntil(sharedSource.TakeLast(1))
        .Publish()
        .RefCount();
        return intervals.GroupJoin(
        sharedSource,
        _ => intervals,
        _ => Observable.Empty<Unit>(),
        (left, sourceValues) => sourceValues)
        .Subscribe(o);
        });
}
```

For an alternative summary of reducing operators to a primitive set see Bart DeSmet's excellent MINLINQ post (and follow-up video). Bart is one of the key members of the team that built Rx, so it is great to get some insight on how the creators of Rx think.

Showcasing *GroupJoin* and the use of other operators turned out to be a fun academic exercise. While watching videos and reading books on Rx will increase your familiarity with it, nothing replaces the experience of actually picking it apart and using it in earnest.

GroupJoin and other window operators reduce the need for low-level plumbing of state and concurrency. By exposing a high-level API, code that would be otherwise difficult to write, becomes a cinch to put together. For example, those in the finance industry could use *GroupJoin* to easily produce real-time Volume or Time Weighted Average Prices (VWAP/TWAP).

Rx delivers yet another way to query data in motion by allowing you to interrogate sequences of coincidence. This enables you to solve the intrinsically complex problem of managing state and concurrency while performing matching from multiple sources. By encapsulating these low level operations, you are able to leverage Rx to design your software in an expressive and testable fashion. Using the Rx operators as building blocks, your code effectively becomes a composition of many simple operators. This allows the complexity of the domain code to be the focus, not the otherwise incidental supporting code.

Summary

When LINQ was first released, it brought the ability to query static data sources directly into the language. With the volume of data produced in modern times, only being able to query data-at-rest, limits your competitive advantage. Being able to make sense of information as it flows, opens an entirely new spectrum of software. We need more than just the ability to react to events, we have been able to do this for years. We need the ability to construct complex queries across multiple sources of flowing data.

Rx brings event processing to the masses by allowing you to query data-in-motion directly from your favorite .NET language. Composition is king: you compose operators to create queries and you compose sequences to enrich the data. Rx leverages common types, patterns and language features to deliver an incredibly powerful library that can change the way you write modern software.

Throughout the book you will have learnt the basic types and principle of Rx. You have discovered functional programming concepts and how they apply to observable sequences. You can identify potential pitfalls of certain patterns and how to avoid them. You understand the internal working of the operators and are even able to build your own implementations of many of them. Finally you are able to construct complex queries that manage concurrency in a safe and declarative way while still being testable.

You have everything you need to confidently build applications using the Reactive Extensions for .NET. If you do find yourself at any time stuck, and not sure how to solve a problem or need help, you can probably solve it without outside stimulus. Remember to first draw a marble diagram of what you think the problem space is. This should allow you to see the patterns in the flow which will help you choose the correct operators. Secondly, remember to follow the Guidelines. Third, write a spike. Use LINQPad or a blank Visual Studio project to flesh out a small sample. Finally, if you are still stuck, your best place to look for help is the MSDN Rx forum. StackOverflow.com is another useful resource too, but with regards to Rx questions, the MSDN forum is dedicated to Rx and seems to have a higher quality of answers.

Appendix

Usage guidelines

This is a list of quick guidelines intended to help you when writing Rx queries.

- Members that return a sequence should never return null. This applies to *IEnumerable<T>* and *IObservable<T>* sequences. Return an empty sequence instead.
- Dispose of subscriptions.
- Subscriptions should match their scope.
- Always provide an `OnError` handler.
- Avoid breaking the monad with blocking operators such as `First`, `FirstOrDefault`, `Last`, `LastOrDefault`, `Single`, `SingleOrDefault` and `ForEach`.
- Avoid switching between monads, i.e. going from *IObservable<T>* to *IEnumerable<T>* and back to *IObservable<T>*.
- Favor lazy evaluation over eager evaluation.
- Break large queries up into parts. Key indicators of a large query:
  1. nesting
  2. over 10 lines of query comprehension syntax
  3. using the into statement
- Name your queries well, i.e. avoid using the names like `query`, `q`, `xs`, `ys`, `subject` etc.
- Avoid creating side effects. If you must create side effects, be explicit by using the *Do* operator.
- Avoid the use of the subject types. Rx is effectively a functional programming paradigm. Using subjects means we are now managing state, which is potentially mutating. Dealing with both mutating state and asynchronous programming at the same time is very hard to get right. Furthermore, many of the operators (extension methods) have been carefully written to ensure that correct and consistent lifetime of subscriptions and sequences is maintained; when you introduce subjects, you can break this. Future releases may also see significant performance degradation if you explicitly use subjects.
- Avoid creating your own implementations of the *IObservable<T>* interface. Favor using the *Observable.Create* factory method overloads instead.
- Avoid creating your own implementations of the *IObserver<T>* interface. Favor using the *Subscribe* extension method overloads instead.
- The subscriber should define the concurrency model. The *SubscribeOn* and *ObserveOn* operators should only ever precede a *Subscribe* method.

Dispelling event myths

The previous parts in this book should have given you a solid and broad foundation in the fundamentals of Rx. We will use this based to learn the really fun and sometimes complex parts of Rx. Just before we do, I want to first make sure we are all on the same page and dispel some common myths and misunderstandings. Carrying these misconceptions into a world of concurrency will make things seem magic and mysterious. This normally leads to problematic code.

Event myths

Often in my career, I have found myself involved in the process of interviewing new candidates for developer roles. I have often been surprised about the lack of understanding developers had surrounding .NET events. Carrying these misconceptions into a world of concurrency will make things seem magic and mysterious. This normally leads to problematic code. Here is a short list of verifiable facts about events.

Events are just a syntactic implementation of the observer pattern
: The += and -= syntax in c# may lead you to think that there is something clever going on here, but it is just the observer pattern; you are providing a delegate to get called back on. Most events pass you data in the form of the sender and the *EventArgs*.

Events are multicast
: Many consumers can register for the same event. Each delegate (handler) will be internally added to a callback list.

Events are single threaded.
: There is nothing multithreaded about an event. Each of the callback handlers are each just called in the order that they registered, and they are called sequentially.

Event handlers that throw exceptions stop other handlers being called
: Since handlers are called sequentially, they are also called on the same thread that raised the event. This means that, if one handler throws an exception, there cannot be a chance for any user code to intercept the exception. This means that the remaining handlers will not be called.

Common myths about events that I have heard (or even believed at some point) include:

- Handlers are called all at the same time on the thread pool, in parallel
- All handlers will get called. Throwing an exception from a handler will not affect other handlers
- You don't need to unregister an event handler, .NET is managed so it will garbage collect everything for you.

The silly thing about these myths is that it only takes fifteen minutes to prove them all wrong; you just open up Visual Studio or LINQPad and test them out. In my opinion, there is something satisfying about proving something you only believed in good faith to be true.

Disposables

Rx leverages the existing *IDisposable* interface for subscription management. This is an incredibly useful design decision, as users can work with a familiar type and reuse existing language features. Rx further extends its usage of the *IDisposable* type by providing several public implementations of the interface. These can be found in the *System.Reactive.Disposables* namespace. Here, we will briefly enumerate each of them.

Disposable.Empty

This static property exposes an implementation of *IDisposable* that performs no action when the *Dispose* method is invoked. This can be useful whenever you need to fulfil an interface requirement, like *Observable.Create*, but do not have any resource management that needs to take place.

Disposable.Create(Action)

This static method exposes an implementation of *IDisposable* that performs the action provided when the *Dispose* method is invoked. As the implementation follows the guidance to be idempotent, the action will only be called on the first time the *Dispose* method is invoked.

BooleanDisposable

This class simply has the *Dispose* method and a read-only property *IsDisposed*. *IsDisposed* is `false` when the class is constructed, and is set to `true` when the *Dispose* method is invoked.

CancellationDisposable

The *CancellationDisposable* class offers an integration point between the .NET cancellation paradigm (*CancellationTokenSource*) and the resource management paradigm (*IDisposable*). You can create an instance of the *CancellationDisposable* class by providing a *CancellationTokenSource* to the constructor, or by having the parameterless constructor create one for you. Calling *Dispose* will invoke the *Cancel* method on the *CancellationTokenSource*. There are two properties (*Token* and *IsDisposed*) that *CancellationDisposable* exposes; they are wrappers for the *CancellationTokenSource* properties, respectively *Token* and *IsCancellationRequested*.

CompositeDisposable

The *CompositeDisposable* type allows you to treat many disposable resources as one. Common usage is to create an instance of *CompositeDisposable* by passing in a `params` array of disposable resources. Calling *Dispose* on the *CompositeDisposable* will call dispose on each of these resources in the order they were provided. Additionally, the *CompositeDisposable* class implements *ICollection<IDisposable>*; this allows you to add and remove resources from the collection. After the *CompositeDisposable* has been disposed of, any further resources that are added to this collection will be disposed of instantly. Any item that is removed from the collection is also disposed of, regardless of whether the collection itself has been disposed of. This includes usage of both the *Remove* and *Clear* methods.

ContextDisposable

ContextDisposable allows you to enforce that disposal of a resource is performed on a given *SynchronizationContext*. The constructor requires both a *SynchronizationContext* and an *IDisposable* resource. When the *Dispose* method is invoked on the *ContextDisposable*, the provided resource will be disposed of on the specified context.

MultipleAssignmentDisposable

The *MultipleAssignmentDisposable* exposes a read-only *IsDisposed* property and a read/write property *Disposable*. Invoking the *Dispose* method on the *MultipleAssignmentDisposable* will dispose of the current value held by the *Disposable* property. It will then set that value to null. As long as the *MultipleAssignmentDisposable* has not been disposed of, you are able to set the *Disposable* property to *IDisposable* values as you would expect. Once the *MultipleAssignmentDisposable* has been disposed, attempting to set the *Disposable* property will cause the value to be instantly disposed of; meanwhile, *Disposable* will remain null.

RefCountDisposable

The *RefCountDisposable* offers the ability to prevent the disposal of an underlying resource until all dependent resources have been disposed. You need an underlying *IDisposable* value to construct a *RefCountDisposable*. You can then call the *GetDisposable* method on the *RefCountDisposable* instance to retrieve a dependent resource. Each time a call to *GetDisposable* is made, an internal counter is incremented. Each time one of the dependent disposables from *GetDisposable* is disposed, the counter is decremented. Only if the counter reaches zero will the underlying be disposed of. This allows you to call *Dispose* on the *RefCountDisposable* itself before or after the count is zero.

ScheduledDisposable

In a similar fashion to *ContextDisposable*, the *ScheduledDisposable* type allows you to specify a scheduler, onto which the underlying resource will be disposed. You need to pass both the instance of *IScheduler* and instance of *IDisposable* to the constructor. When the *ScheduledDisposable* instance is disposed of, the disposal of the underlying resource will be scheduled onto the provided scheduler.

SerialDisposable

SerialDisposable is very similar to *MultipleAssignmentDisposable*, as they both expose a read/write *Disposable* property. The contrast between them is that whenever the *Disposable* property is set on a *SerialDisposable*, the previous value is disposed of. Like the *MultipleAssignmentDisposable*, once the *SerialDisposable* has been disposed of, the *Disposable* property will be set to null and any further attempts to set it will have the value disposed of. The value will remain as null.

SingleAssignmentDisposable

The *SingleAssignmentDisposable* class also exposes *IsDisposed* and *Disposable* properties. Like *MultipleAssignmentDisposable* and *SerialDisposable*, the *Disposable* value will be set to null when the *SingleAssignmentDisposable* is disposed of. The difference in implementation here is that the *SingleAssignmentDisposable* will throw an *InvalidOperationException* if there is an attempt to set the *Disposable* property while the value is not null and the *SingleAssignmentDisposable* has not been disposed of.